Q1.What is a RESTful API ?

- A RESTful API (Representational State Transfer API) is a way for two computer systems to communicate over the internet using the principles of REST architecture. It is one of the most common types of web APIs used in modern web and mobile applications.

Q2.Explain the concept of API specification ?

- Ans:- An API specification is a detailed description of how an Application Programming Interface (API) works. It defines the rules, endpoints, data formats, and behaviors that software components use to communicate with each other.

🔧 What Does an API Specification Include?

1.Endpoints

- The available URLs for accessing resources.
- Example: /users, /products/{id}

2.HTTP Methods

- GET, POST, PUT, DELETE, etc.

3.Request Parameters

- Query parameters (?page=2), headers, path variables (/users/{id}), and request body.

4.Response Format

- Structure of the data returned by the API, usually in JSON or XML.

5.Status Codes

- Standard HTTP codes to indicate the result of the request:
- 200 OK, 201 Created, 400 Bad Request, 404 Not Found, etc.

6.Authentication Requirements

- Tokens, API keys, OAuth, etc.

7.Example Requests and Responses

- Sample input and output data for better understanding

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

- 🐍 Flask is a lightweight, Python-based web framework used to build web applications and RESTful APIs. It is known for being simple, flexible, and easy to get started with.

🚀 Why Flask is Popular for Building APIs:-

✅ Minimal & Lightweight

- You only use what you need—no bloat. Perfect for small to medium projects.

🧩 Extensible

- You can add libraries like Flask-RESTful, SQLAlchemy, or JWT for more features.

🛠️ Easy to Learn

- Great for beginners and very readable Python code.

🗂️ Clear Routing

- You can define endpoints using decorators (@app.route) very intuitively.

📦 Built-in Development Server

- No setup required—run your app with just one command.

🔧 REST API Friendly

- Easy to build GET, POST, PUT, DELETE routes with JSON responses.

Q4. What is routing in Flask ?

- In Flask, routing refers to the process of mapping URLs to specific functions (called view functions) that return responses to the browser. This is a core part of web development, as it defines how a web application responds to different URL requests.

Q5.How do you create a simple Flask application ?

🧰1. Install Flask

- First, make sure Flask is installed. Run this in your terminal: pip install flask

📝 2. Create a Python File Create a file called app.py and add the following code:

```python
from flask import Flask


app = Flask(__name__)


@app.route('/')
def home():
    return "Hello, World! This is your first Flask app."


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

▶️ 3. Run the Application In your terminal, run:

python app.py

You'll see output like:

- Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

"Hello, World! This is your first Flask app."

Q6.What are HTTP methods used in RESTful APIs ?

- 🌐 HTTP Methods Used in RESTful APIs In RESTful APIs, HTTP methods define the type of action a client wants to perform on a resource (like users, products, posts, etc.). These methods map to CRUD operations: Create, Read, Update, Delete.

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

- 🧩 In Flask, the @app.route() decorator is used to map a URL to a specific Python function (called a view function). It tells Flask:

🔧 Basic Usage

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

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

🎯 Purpose of @app.route():-

📍 URL Mapping-

- Connects a specific URL (like /about) to a function.

🏞️ Routing-

- Helps Flask know what function to call for each URL.

🧠 Readable & Clean-

- Makes it clear which URL triggers which function.

⚙️ Supports HTTP Methods-

- You can specify allowed methods like GET, POST, etc.

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

- 🔄 Difference Between GET and POST HTTP Methods Feature-

- 📌 Purpose

- 📦 Data Location

- 🔐 Security

- 📚 Caching

- 🔁 Idempotent

- 📏 Data Limit

- 💡 Use For

GET-

- Retrieve data from the server
- URL (query parameters)
- Less secure (data visible in URL)
- Can be cached
- Yes (repeating gives same result)
- Limited (URL length restriction)
- Reading/searching data (safe ops)

POST

- Send data to the server
- HTTP request body
- More secure (data hidden from URL)
- Not cached by default
- No (can create duplicate entries)
- Can send large data
- Submitting forms, uploading data

Q9.How do you handle errors in Flask APIs ?

- ❗ Error handling in Flask is important for providing clear feedback to clients and ensuring your API doesn't crash when something goes wrong.

🔧 1. Using abort() for Common HTTP Errors

Flask provides a built-in abort() function to stop a request and return an error code:

```
from flask import Flask, abort

app = Flask(__name__)

@app.route('/user/<int:user_id>')
def get_user(user_id):
    if user_id != 1:
        abort(404)
    return {"id": 1, "name": "Sonu"}
```

🚨 2. Custom Error Handlers

You can define custom responses for specific errors:

```
from flask import jsonify

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

@app.errorhandler(500)
def internal_error(error):
    return jsonify({"error": "Internal server error"}), 500
```

🧪 3. Returning Custom Error Responses in Views

Instead of letting an error crash the app, return a message manually:

```
@app.route('/divide')
def divide():
    try:
        result = 10 / 0
        return jsonify({"result": result})
    except ZeroDivisionError:
        return jsonify({"error": "Division by zero is not allowed"}), 400
```

🛡️ 4. Validation Errors (With request)

Example when processing form data:

```
from flask import request

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

```
    def login():
        data = request.get_json()
        if not data or 'username' not in data:
            return jsonify({"error": "Username is required"}), 400
        return jsonify({"message": "Login successful"})
```

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

- 🧩Flask doesn't connect to a database directly — you use extensions like Flask-SQLAlchemy to handle database integration easily.

🔧 Step-by-Step Guide to Connect Flask with a SQL Database

1️⃣ Install Flask and Flask-SQLAlchemy

pip install flask flask-sqlalchemy

2️⃣ Create Your Flask App

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///students.db'  # or PostgreSQL/
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db = SQLAlchemy(app)
```

3️⃣ Define Your Database Model (Table)

```
class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    age = db.Column(db.Integer)
```

4️⃣ Create the Database In a Python shell or inside your script:

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

5️⃣ Insert Data into the Table

```
    @app.route('/add')
    def add_student():
        new_student = Student(name="Sonu", age=21)
        db.session.add(new_student)
        db.session.commit()
        return "Student added!"
```

6️⃣ Retrieve Data from the Table

```
    @app.route('/students')
    def get_students():
        students = Student.query.all()
        return { "students": [ {"id": s.id, "name": s.name, "age": s.age} for s i
```

Q11.What is the role of Flask-SQLAlchemy ?

🧩Flask-SQLAlchemy is an extension for Flask that adds SQL database support to your Flask applications using SQLAlchemy ORM (Object Relational Mapper).

🎯 Key Roles of Flask-SQLAlchemy

🛠️ Database Setup-

- Connects Flask to a SQL database easily.

🧠 ORM Functionality-

- Lets you interact with database tables as Python classes.

✅ Query Simplification-

- Helps you write Pythonic queries instead of raw SQL.

📦 Data Models-

- Allows you to define tables using Python classes (db.Model)

🔄 Session Management-

- Handles database transactions (insert, update, delete, etc.)

🧪 Migration Support-

- Works well with tools like Flask-Migrate for version control of your DB schema

Q12.What are Flask blueprints, and how are they useful ?

- 🧩Flask Blueprints are a way to organize your Flask application into smaller, reusable components. They allow you to split a large app into modular parts, making the code cleaner

and easier to manage.

🎯 Why Use Blueprints?

📦 Modularity

- Keep routes, views, and logic separate for different features (e.g., auth, dashboard).

🔄 Reusability

- You can reuse the same blueprint in multiple projects.

🧩 Scalability

- Makes your app structure more maintainable as it grows.

✅ Clarity

Keeps your app.py clean and focused.

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

- 📥The request object in Flask is used to access incoming request data sent by the client (like a browser, mobile app, or Postman). It is part of Flask's flask module and provides all the info related to an HTTP request.

🎯 Key Purposes of the request Object:-

📦 request.args-

- Get data from the URL query string (GET method).

📄 request.form-

- Get data from an HTML form (POST method).

🧪 request.json-

- Get JSON data from API requests.

📂 request.files-

- Handle file uploads.

🧭 request.method-

- Know whether the request is GET, POST, etc.

🌐 request.headers-

- Access HTTP request headers.

🌍 request.remote_addr-

- Get the client's IP address.

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

- 🛠️A RESTful API endpoint allows clients to interact with your server using standard HTTP methods like GET, POST, PUT, and DELETE.

✅ Step-by-Step Example

- Let's create a simple Flask API that manages a list of users.

1️⃣ Install Flask

pip install flask

2️⃣ Basic Flask API Code

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

app = Flask(__name__)

users = [
    {"id": 1, "name": "Sonu"},
    {"id": 2, "name": "Kumar"}
]

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

@app.route('/users', methods=['POST'])
def add_user():
    data = request.get_json()
    new_user = {
        "id": len(users) + 1,
        "name": data['name']
    }
    users.append(new_user)
    return jsonify(new_user), 201

@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
```

```
    @app.route('/users/<int:user_id>', methods=['DELETE'])
    def delete_user(user_id):
        global users
        users = [u for u in users if u["id"] != user_id]
        return jsonify({"message": "User deleted"})

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

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

🧪 The jsonify() function in Flask is used to convert Python data (like dictionaries and lists) into JSON format, which is the standard format used in APIs.

🎯 Why Use jsonify()? 🔁 Converts Python → JSON-

- Automatically converts Python dict, list, etc. into JSON.

🧾 Sets Proper Headers-

- Sets the Content-Type: application/json header.

🚀 Returns Valid API Responses-

- Ensures API clients (like Postman or frontend apps) receive JSON.

🔐 Safe Encoding-

- Prevents security issues like XSS by escaping characters properly

Q16.Explain Flask's url_for() function ?

✅ Definition:

url_for() is a built-in Flask function used to dynamically build URLs for your Flask app based on the function name of a route, rather than hardcoding the actual URL path.

🎯 Purpose:

- The main goal of url_for() is to ensure that URL building in your app is:
- Dynamic (updates automatically if route changes)
- Reliable (avoids typos and broken links)

Clean and maintainable

🧠 How It Works:

- Instead of writing URLs as plain strings (like /home), you refer to the name of the view function, and Flask will generate the correct URL for you.

```
Example-


@app.route('/dashboard')
def dashboard():
    return "Welcome to dashboard"
```

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

- Flask uses a special folder named static/ to serve static files like:
- 🎨 CSS files
- ⚙️ JavaScript files
- 🖼️ Images
- 📁 Fonts and other frontend assets
- 📁 Default Project Structure

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

🧠 How It Works

- Flask automatically looks for static files in the /static/ directory.
- These files can be accessed in your browser using the URL:

```
http://localhost:5000/static/filename
```

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

✅An API specification is a formal, structured description of how an API should behave. It defines the endpoints, HTTP methods, request and response formats, parameters, authentication rules,

and error codes that the API will use.

📌 Key Elements Typically Included:

- Base URL: The root path of the API (e.g., https://api.example.com)

- Endpoints: Paths like /users, /products/

- HTTP Methods: Such as GET, POST, PUT, DELETE

- Request Parameters: Inputs via path, query, or body

- Response Format: Usually JSON or XML

- Status Codes: Such as 200 OK, 404 Not Found, 500 Internal Server Error

- Authentication: Like API keys, JWT tokens, OAuth

🛠️ How It Helps in Building a Flask API

1.Clear Structure and Planning

- The specification acts as a design document, helping developers plan the structure and behavior of the Flask API before writing code.

2.Improves Collaboration

- Frontend and backend developers can work independently using the API spec as a shared contract.

3.Documentation Generation

- Tools like Swagger (OpenAPI) can generate interactive documentation directly from the spec, making it easier for others to understand and use the API.

4.Easier Testing and Validation

- With a defined specification, it becomes easier to test endpoints and validate inputs and outputs consistently.

5.Better Maintainability

- A spec ensures that the API remains consistent and understandable even as the project grows or changes over time

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

- ✅HTTP status codes are standardized 3-digit numerical codes that a web server returns in response to an HTTP request made by a client (like a browser or an API consumer).

- These codes indicate the outcome of the request, such as:

- Whether it was successful,

- If there was an error,

- Or if further action is needed.

🔑 Why Are They Important in a Flask API?

1.Clear Communication

- Status codes allow the server (Flask app) to communicate the result of a request clearly to the client (e.g., browser, mobile app, frontend).

2.Improved Error Handling

- The client can detect problems (like 404 Not Found or 400 Bad Request) and take appropriate actions.

3.Helps Debugging

- Developers can easily identify what went wrong in the request/response cycle.

4.Follows REST Standards

- Proper use of status codes makes a Flask API RESTful and aligns it with industry standards.

5.Better User Experience

- Applications using the API can show meaningful messages or actions to users based on the status code.

Q20.How do you handle POST requests in Flask ?

- ✅ What is a POST Request? A POST request is used to send data from the client (such as a form or app) to the server, typically to create a new resource. In Flask, you can handle POST requests using the @app.route() decorator with methods=["POST"].

🧪 Example: Handling a POST Request in Flask

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/submit', methods=['POST'])
def submit_data():
    # Get JSON data from the request
    data = request.get_json()
    name = data.get('name')
    age = data.get('age')

    return jsonify({"message": "Data received", "name": name, "age": age}), 2
```

Q21.How would you secure a Flask API ?

- 🔒 Securing a Flask API is essential to protect data, prevent unauthorized access, and ensure safe communication between clients and the server. Here's a theoretical overview of how to secure a Flask API effectively:

✅ 1. Authentication

Authentication verifies who the user is.

Common methods:

- API Keys: Send a secret key in headers (X-API-Key) with each request.

JWT (JSON Web Tokens): Token-based system for stateless authentication.

OAuth 2.0: Used for third-party logins (like Google, GitHub).

🔐 Example with JWT:

from flask_jwt_extended import JWTManager, jwt_required

```
# Initialize JWT
app.config['JWT_SECRET_KEY'] = 'your-secret-key'
jwt = JWTManager(app)

@app.route('/protected')
@jwt_required()
def protected():
    return jsonify(message="Access granted")
```

✅ 2. Authorization

Authorization determines what the user is allowed to do.

Example: A user may be allowed to read a post, but not delete it. You implement this by checking user roles or permissions in each endpoint.

✅ 3. HTTPS (SSL/TLS)

Always run your Flask app over HTTPS, not HTTP. It encrypts the communication between client and server to protect sensitive data.

✅ 4. Input Validation and Sanitization

- Never trust user input. Use validation to prevent:
- SQL Injection

- XSS (Cross-Site Scripting)

- CSRF (Cross-Site Request Forgery)

✔️ Use libraries like:

marshmallow for JSON schema validation

WTForms for form validation

✅ 5. Rate Limiting

Prevent abuse or DDoS attacks by limiting the number of requests per user/IP.

📦 Use Flask extensions like:

Flask-Limiter ✅ 6. Use Environment Variables for Secrets

Never hardcode API keys or passwords. Store them securely using:

.env files + python-dotenv

Environment variables (os.environ)

✅ 7. Error Handling

Never expose internal server errors to users. Instead:

Show generic error messages

Log details on the backend for debugging

✅ 8. Enable CORS Carefully

If your API is accessed from browsers, configure Cross-Origin Resource Sharing (CORS) securely.

📦 Use:

from flask_cors import CORS CORS(app, resources={r"/api/*": {"origins": "https://your-frontend.com"}})

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

- ✅Flask-RESTful is an extension for Flask that helps you quickly build RESTful APIs by providing tools and abstractions for:

- Routing

- Request parsing

- Input validation

- Standardized response formatting

- It is built on top of Flask and simplifies the process of creating clean, modular, and maintainable API code.

🎯 Significance / Benefits of Using Flask-RESTful

1. ✅ Class-Based Resource Handling

- Allows you to define API endpoints using Python classes instead of functions.

- Each HTTP method is defined as a method (get(), post(), etc.) in a class.

- from flask_restful import Resource

```
class Hello(Resource):
    def get(self):
        return {"message": "Hello, World!"}
```

2. 🛠️ Simplified Routing with Api Class

- You use the Api object to manage routes cleanly.

  from flask_restful import Api api = Api(app) api.add_resource(Hello, '/hello')

3. 🔍 Automatic Request

- Easily handle JSON and form data using reqparse.

  from flask_restful import reqparse parser = reqparse.RequestParser() parser.add_argument('name', type=str, required=True) args = parser.parse_args()

4. 📦 Standardized Response Format Returns are automatically converted into JSON, with proper HTTP status codes.

return {'message': 'Item created'}, 201

5. 🧩 Modular and Clean Code

- Encourages RESTful design patterns and separates logic for each resource.

6. 🔒 Easy Integration with Flask Ecosystem

- Works well with Flask extensions like:

- Flask-JWT-Extended (for authentication)

- Flask-SQLAlchemy (for database)

- Flask-CORS (for cross-origin requests)

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

- In Flask, the session object is used to store data across requests for a specific user. It allows you to remember information (like login status or user preferences) from one request to the next.

🧠 Why is session important:-

- HTTP is a stateless protocol, meaning each request is independent and doesn't "remember" the previous one.

- The session object gives Flask the ability to maintain state between different requests from the same client.

🛠️ How It Works:

- Flask uses secure cookies to store session data on the client side.

- The data is signed cryptographically using a secret key so it cannot be tampered with.

Example:

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

app = Flask(__name__)
app.secret_key = 'your_secret_key'

@app.route('/login', methods=['POST'])
def login():
    username = request.form['username']
    session['username'] = username
    return f"Welcome, {username}!"

@app.route('/logout')
def logout():
    session.pop('username', None)
    return "You have been logged out."
```

Practical Questions

1.How do you create a basic Flask application?

```python
pip install flask
from flask import Flask

# Create the Flask application
app = Flask(__name__)
```

```
    # Define a basic route
    @app.route('/')
    def home():
        return "Welcome to your first Flask app!"


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

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

```
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)
```

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

```
from flask import Flask, request

app = Flask(__name__)

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

# GET and POST methods
@app.route('/submit', methods=['GET', 'POST'])
def submit():
    if request.method == 'POST':
        name = request.form.get('name')
        return f"Hello, {name} (from POST)"
```

```
    return '''
        <form method="POST">
            <input type="text" name="name" placeholder="Enter your name">
            <input type="submit" value="Submit">
        </form>
    '''


# PUT method
@app.route('/update', methods=['PUT'])
def update():
    return "Data updated using PUT request"


# DELETE method
@app.route('/delete', methods=['DELETE'])
def delete():
    return "Data deleted using DELETE request"


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

4.How do you render HTML templates in Flask?

```
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)
```

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

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

app = Flask(__name__)

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

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


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


@app.route('/go-to-about')
def go_to_about():
        # Redirect to 'about' route using url_for
        return redirect(url_for('about'))


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

6.How do you handle forms in flasks?

```
    from flask import Flask, request, render_template


    app = Flask(__name__)


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


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


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

7.How can you validate form data in Flask?

```
    from flask import Flask, request, render_template


    app = Flask(__name__)


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

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

    # Manual validation
    if not name or not email:
        return "Name and email are required!"
    if '@' not in email:
        return "Invalid email format!"

    return f"Welcome {name}, your email is {email}"

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

8.How do you manage sessions in flask?

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

app = Flask(__name__)
app.secret_key = 'your_secret_key_here'  # Required for session security

@app.route('/')
def home():
    return render_template('login.html')

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

    if username:
        session['user'] = username  # Store data in session
        return redirect(url_for('dashboard'))
    return "Username required!"

@app.route('/dashboard')
def dashboard():
    if 'user' in session:
        return render_template('dashboard.html', user=session['user'])
    return redirect(url_for('home'))
```

```python
    @app.route('/logout')
    def logout():
        session.pop('user', None)   # Clear session
        return redirect(url_for('home'))


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

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

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

    app = Flask(__name__)

    @app.route('/')
    def home():
        return '<h1>Home Page</h1><a href="/login">Go to Login</a>'

    @app.route('/login')
    def login():
        # Imagine login is successful, now redirect to dashboard
        return redirect(url_for('dashboard'))

    @app.route('/dashboard')
    def dashboard():
        return '<h1>Welcome to the Dashboard!</h1>'

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

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

```python
    from flask import Flask, render_template

    app = Flask(__name__)

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

    @app.route('/cause-error')
```

```
def cause_error():
    # This will raise an internal error
    return 1 / 0

# Custom 404 Error Handler
@app.errorhandler(404)
def not_found(error):
    return render_template('404.html'), 404

# Custom 500 Error Handler
@app.errorhandler(500)
def server_error(error):
    return render_template('500.html'), 500

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

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

```
from flask import Flask

# Import blueprints
from auth.routes import auth_bp
from main.routes import main_bp

app = Flask(__name__)
app.secret_key = 'your_secret_key'

# Register blueprints
app.register_blueprint(auth_bp, url_prefix='/auth')
app.register_blueprint(main_bp)

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

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

```
from flask import Flask, render_template

app = Flask(__name__)
```

```python
# Define custom filter to reverse a string
@app.template_filter('reverse')
def reverse_filter(s):
    return s[::-1]


@app.route('/')
def home():
    return render_template('home.html', name='Sonu Kumar')


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

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

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

app = Flask(__name__)

@app.route('/')
def home():
    return '<a href="/go">Go with query params</a>'

@app.route('/go')
def go():
    # Redirect to /destination with query parameters
    return redirect(url_for('destination', name='Sonu', age=24))

@app.route('/destination')
def destination():
    # Access query parameters
    name = request.args.get('name')
    age = request.args.get('age')
    return f"Name: {name}, Age: {age}"

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

14. How do you return JSON responses in Flask?

```python
from flask import Flask, jsonify
```

```
app = Flask(__name__)


@app.route('/api/data')
def api_data():
    data = {
        'name': 'Sonu Kumar',
        'role': 'Data Analyst',
        'skills': ['Python', 'SQL', 'Flask']
    }
    return jsonify(data)


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

15.How do you capture URL parameters in Flask?

```
from flask import Flask


app = Flask(__name__)


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


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