Programming in Python with Full Stack Development (303105257)

Anand Jawdekar

Assistant Professor CSE Department

Unit-4

RESTful APIs:

Unit-4

RESTful APIs:

Introduction to RESTful APIs and the REST architectural style
Understanding the HTTP protocol and its role in RESTful APIs
Designing and implementing RESTful APIs using common HTTP methods, such as GET, POST, PUT, and DELETE
Using URLs and resource representations to identify and transfer data in RESTful APIs
Implementing best practices for designing and implementing RESTful APIs, such as using HTTP status codes, versioning, and error handling
Consuming RESTful APIs using common tools and libraries, such as cURL, Postman, and the requests library in Python Building scalable and secure RESTful APIs using common frameworks and libraries Flask or FastAPI.

REST API Introduction

REpresentational **S**tate **T**ransfer (REST) is an architectural style that defines a set of constraints to be used for creating web services. **REST API** is a way of accessing web services in a simple and flexible way without having any processing.

REST technology is generally preferred to the more robust **Simple Object Access Protocol (SOAP)** technology because REST uses less bandwidth, simple and flexible making it more suitable for internet usage. It's used to fetch or give some information from a web service. All communication done via REST API uses only HTTP request.

Working: A request is sent from client to server in the form of a web URL as HTTP GET or POST or PUT or DELETE request. After that, a response comes back from the server in the form of a resource which can be anything like HTML, XML, Image, or JSON. But now JSON is the most popular format being used in Web Services.

REST API Introduction

In **HTTP** there are five methods that are commonly used in a REST-based Architecture i.e., POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations respectively. There are other methods which are less frequently used like OPTIONS and HEAD.

GET: The HTTP GET method is used to **read** (or retrieve) a representation of a resource. In the safe path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

POST: The POST verb is most often utilized to **create** new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

REST API Introduction

PUT: It is used for **updating** the capabilities. However, PUT can also be used to **create** a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. PUT is not safe operation but it's idempotent.

PATCH: It is used to **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource. This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch. PATCH is neither safe nor idempotent.

DELETE: It is used to **delete** a resource identified by a URI. On successful deletion, return HTTP status 200 (OK) along with a response body.

Example

```
1. a = 4 // It is Idempotence, as final value(a = 4)
        // would not change after executing it multiple
       // times.

2. a++ // It is not Idempotence because the final value
      // will depend upon the number of times the
     // statement is executed.
```

RESTful web services are very popular because they are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

REST API Architectural Constraints

REST stands for **Representational State Transfer** and **API** stands for **Application Program Interface**. REST is a software architectural style that defines the set of rules to be used for creating web services. Web services that follow the REST architectural style are known as RESTful web services. It allows requesting systems to access and manipulate web resources by using a uniform and predefined set of rules. Interaction in REST-based systems happens through the Internet's Hypertext Transfer Protocol (HTTP).

A Restful system consists of a:

A client who requests for the resources.

server who has the resources.

It is important to create REST API according to industry standards which results in ease of development and increase client adoption.

Architectural Constraints of RESTful API

There are six architectural constraints that makes any web service are listed below:

Uniform Interface

Stateless

Cacheable

Client-Server

Layered System

Code on Demand

The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful.

Using URLs and Resource Representations to Identify and Transfer Data in RESTful APIs

In **RESTful APIs**, **URLs (Uniform Resource Locators)** are used to uniquely **identify resources**, while **resource representations** (JSON, XML, etc.) are used to **transfer data** between clients and servers.

1. Identifying Resources Using URLs

Each **resource** in a REST API is identified by a **unique URL (endpoint)**.

URLs should be **meaningful, hierarchical, and descriptive**.

Method	URL Example	Description
GET	/users	Fetch all users
GET	/users/123	Fetch user with ID 123
POST	/users	Create a new user
PUT	/users/123	Update user 123
DELETE	/users/123	Delete user 123

Cont..

2. Transferring Data Using Resource Representations

Data in RESTful APIs is transferred in formats like:

JSON (JavaScript Object Notation)

XML (Extensible Markup Language)

YAML, CSV, or Plain Text (in some cases)

```
{
  "id": 123,
  "name": "John Doe",
  "email": "johndoe@example.com",
   "role": "admin"
}
```

Cont..

3. Using HTTP Headers for Content Negotiation
Clients and servers use **HTTP headers** to specify the data format.

Header	Example Value	Purpose
Content-Type	application/json	Specifies data format in request body
Accept	application/json	Specifies the format client expects in response

Example Request (Sending JSON Data)

```
POST /users HTTP/1.1
Host: api.example.com
Content-Type: application/json
Accept: application/json

{
  "name": "Alice",
  "email": "alice@example.com"
}
```

4. Example REST API Using Flask

```
from flask import Flask, jsonify, request

app = Flask(__name__)
# Sample data
users = [
    {"id": 1, "name": "Alice", "email": "alice@example.com"},
    {"id": 2, "name": "Bob", "email": "bob@example.com"}
]
@app.route('/users', methods=['GET'])
def get_users():
    return jsonify(users)  # Return JSON response

@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)
    return jsonify(user) if user else ("User not found", 404)

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

Implementing best practices for designing and implementing RESTful APIs

1. Use Proper HTTP Methods

Each HTTP method should be used appropriately based on its intended purpose:

GET – Retrieve a resource.

POST – Create a new resource.

PUT – Update a resource or create one if it doesn’t exist (idempotent).

PATCH – Partially update a resource.

DELETE – Remove a resource

Parul University

DIGITAL LEARNING CONTENT

Cont..

2. Use Meaningful Resource URIs

Keep URLs simple, predictable, and resource-oriented:

✅ **Good:**

GET /users/{id} POST /users PUT /users/{id} DELETE /users/{id}

❌ **Bad:**

GET /getUser?id=123 POST /createNewUser

3. Implement API Versioning

Versioning helps manage changes without breaking existing clients. Common versioning methods:

URI Versioning: /v1/users

Header Versioning: Accept: application/vnd.company.v1+json

Query Parameter: /users?version=1

Cont..

4. Use Proper HTTP Status Codes

Use appropriate status codes to indicate the outcome of an API request:

200 OK – Request successful.

201 Created – Resource created successfully.

204 No Content – Request successful, but no response body.

400 Bad Request – Client error (e.g., missing parameters).

401 Unauthorized – Authentication required.

403 Forbidden – Insufficient permissions.

404 Not Found – Resource doesn’t exist.

500 Internal Server Error – Unexpected server error

Cont..

5. Implement Error Handling

Use consistent error response formats with meaningful messages:

✅ **Example Error Response:**

```
{
"error": {
 "code": 400,
 "message": "Invalid email format",
"details": "Ensure email follows standard format (e.g., user@example.com)“
 }
 }
```

Parul University

Cont..

6. Use HATEOAS (Hypermedia as the Engine of Application State)

Provide links for navigation:

✅ **Example Response with HATEOAS:**

```
{
 "id": 123,
 "name": "John Doe",
 "email": "john@example.com",
 "_links": {
  "self": { "href": "/users/123" },
  "orders": { "href": "/users/123/orders" }
 }
}
```

PU

Cont..

7. Implement Authentication and Authorization
Use OAuth 2.0, JWT (JSON Web Tokens), or API keys to secure endpoints.
✅ **Example JWT Authorization Header:**
Authorization: Bearer <token>

8. Optimize Performance
Use **pagination** for large datasets (/users?page=2&size=50).
Implement **caching** using ETag and Cache-Control.
Use **compression** (Gzip) to reduce response size.

Cont..

9. Documentation with OpenAPI (Swagger)

Use **Swagger/OpenAPI** for API documentation.

Example:

```
paths:
  /users/{id}:
    get:
      summary: Get a user by ID
      responses:
        "200":
          description: Successful response
```

10. Logging and Monitoring

Implement logging and monitoring to track API usage and errors.

Consuming RESTful APIs using common tools and libraries

When interacting with RESTful APIs, various tools and libraries help send requests and process responses efficiently. Below are common methods for consuming APIs using **cURL**, **Postman**, and the **Requests library in Python**.

1. Using cURL (Command Line)

cURL is a powerful command-line tool for sending HTTP requests.

Basic GET Request

```
curl -X GET "https://jsonplaceholder.typicode.com/posts/1"
```

Sending Data with a POST Request

```
curl -X POST "https://jsonplaceholder.typicode.com/posts" \
    -H "Content-Type: application/json" \
    -d '{"title": "New Post", "body": "This is a test post", "userId": 1}'
```

Passing Headers (e.g., Authorization)

```
curl -X GET "https://api.example.com/data" \
    -H "Authorization: Bearer YOUR_ACCESS_TOKEN"
```

Cont..

2. Using Postman (GUI-Based)

ostman provides a user-friendly interface for making API requests.

Steps to Use Postman

Open Postman and enter the API **URL**.

Choose **HTTP Method** (GET, POST, PUT, DELETE).

Add **Headers** (e.g., Authorization, Content-Type).

Add a **Request Body** for POST/PUT requests (JSON format).

Click **Send** and view the response.

Cont..

3. Using the requests Library in Python

The requests library simplifies API consumption in Python.

Installing requests (if not already installed)

```
pip install requests
```

Basic GET Request

```
import requests

url = "https://jsonplaceholder.typicode.com/posts/1"
response = requests.get(url)

print(response.status_code)  # 200
print(response.json())       # JSON response
```

Cont..

Sending a POST Request

```
import requests

url = "https://jsonplaceholder.typicode.com/posts"
data = {
    "title": "New Post",
    "body": "This is a test post",
    "userId": 1
}

response = requests.post(url, json=data)
print(response.status_code)  # 201 Created
print(response.json())
```

Cont..

Adding Headers (e.g., Authorization)

```
headers = {
    "Authorization": "Bearer YOUR_ACCESS_TOKEN",
    "Content-Type": "application/json"
}

response = requests.get("https://api.example.com/data", headers=headers)
print(response.json())
```

Cont..

Handling Errors Gracefully

```
try:
    response = requests.get("https://jsonplaceholder.typicode.com/posts/99999")
    response.raise_for_status()  # Raise an error for HTTP errors
    print(response.json())
except requests.exceptions.HTTPError as err:
    print(f"HTTP Error: {err}")
except requests.exceptions.RequestException as err:
    print(f"Request Error: {err}")
```

Cont..

Comparison of Methods

Method	Ease of Use	Best For
cURL	Moderate	Quick testing in terminal
Postman	Easy	GUI-based API testing
Python requests	Easy	Automating API interactions

Building Scalable and Secure RESTful APIs with Flask & FastAPI

When building RESTful APIs, **Flask** and **FastAPI** are two popular Python frameworks:
Flask: Lightweight and widely used, but requires extensions for async support and performance optimizations.
FastAPI: Modern, high-performance, and built on ASGI, providing **automatic validation** and **async support**.

1. Building a Basic REST API with Flask

Install flask

```
pip install flask
Flask API Example
from flask import Flask, request, jsonify

app = Flask(__name__)

# Sample in-memory data
users = [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]

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

Building a Basic REST API with Flask

```
@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)
    return jsonify(user) if user else (jsonify({"error": "User not found"}), 404)

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

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

Building a Basic REST API with Flask

Pros of Flask

✅ Simple and easy to learn

✅ Large ecosystem of extensions

✅ Good for small to medium applications

Cons of Flask

❌ No built-in async support (requires additional libraries)

❌ Needs extra packages for request validation and security

2. Building a REST API with FastAPI

Install FastAPI

```
pip install fastapi uvicorn
```

FastAPI API Example

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI()

class User(BaseModel):
    name: str

users = [{"id": 1, "name": "Alice"}, {"id": 2, "name": "Bob"}]

@app.get("/users")
def get_users():
    return users
```

Cont..

```
@app.get("/users/{user_id}")
def get_user(user_id: int):
    user = next((u for u in users if u["id"] == user_id), None)
    if user is None:
        raise HTTPException(status_code=404, detail="User not found")
    return user

@app.post("/users", status_code=201)
def create_user(user: User):
    new_user = {"id": len(users) + 1, "name": user.name}
    users.append(new_user)
    return new_user
```

Cont..

Run the FastAPI server

```
uvicorn main:app --reload
```

Pros of FastAPI

✅ **Built-in async support** (Fast & efficient)
✅ **Automatic request validation** with Pydantic
✅ **Auto-generate API documentation** (Swagger UI & ReDoc)

Cons of FastAPI

❌ Slightly steeper learning curve
❌ Less mature ecosystem than Flask