# REST API

REST (Representational State Transfer) is an architectural style for designing networked applications. A REST API is an application programming interface that follows REST principles and typically uses HTTP for communication.

REST was introduced by Roy Fielding in 2000 in his doctoral dissertation. It is not a strict protocol like JSON-RPC, but rather a set of architectural constraints.

REST is resource-oriented, which means the system is organized around resources instead of procedures. A resource can represent any entity, such as a user, product, order, or document. Each resource is identified by a unique URL.

For example:

```
GET /users/15
GET /products/42
```

In REST, the client does not call functions explicitly. Instead, it performs operations on resources using standard HTTP methods.

## Core REST Principles

### Resource Identification

Every resource must have a unique identifier, typically a URL. The URL represents the resource, not the action.

Bad example (RPC-style thinking):

```
GET /getUserById?id=10
```

### HTTP Methods (Verbs)

REST relies on standard HTTP methods to define operations:

| Method | Meaning | Example |
|---|---|---|
| GET | Retrieve resource | GET /users/1 |
| POST | Create new resource | POST /users |
| PUT | Replace resource | PUT /users/1 |
| PATCH | Partial update | PATCH /users/1 |
| DELETE | Remove resource | DELETE /users/1 |

The method defines the action, not the URL.

#### Statelessness

Like JSON-RPC, REST is stateless. Each request must contain all necessary information. The server does not store client session state between requests (unless explicitly implemented via tokens or cookies).

This improves scalability and simplifies distributed deployments.

#### Representation

Resources are transferred in representations, typically:

- JSON
- XML
- HTML
- Plain text

Today, JSON is the dominant format in REST APIs.

Example response:

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

### REST vs JSON-RPC

| **Feature** | **REST** | **JSON-RPC** |
|---|---|---|
| Architecture | Resource-based | Procedure-based |
| Endpoints | Multiple | Usually single |
| Uses HTTP verbs | Yes | Not required |
| Standardized by | Architectural constraints | Protocol specification |
| Typical use case | Public APIs | Internal APIs |

REST is typically used for public web APIs, while JSON-RPC is often used in internal systems and blockchain interfaces.

## REST API Server (Python + Flask)

In this section, we implement a simple REST API using Flask.

Install dependency:

```
pip install flask
```

Create file: *server.py*

```
```

```
from flask import Flask, jsonify, request

app = Flask(__name__)

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


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


@app.route("/users/<int:user_id>", methods=["GET"])
def get_user(user_id):
    for user in users:
        if user["id"] == user_id:
            return jsonify(user)
    return jsonify({"error": "User not found"}), 404


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


@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(port=5000)
```

Run:

```
python server.py
```

## Testing with curl

Get all users:

```
curl http://localhost:5000/users
```

Create new user:

```
curl -X POST http://localhost:5000/users \ -H "Content-Type: application/json" \ -d '{"name":"Charlie"}'
```

Delete user:

```
curl -X DELETE http://localhost:5000/users/1
```

## HTTP Status Codes

REST APIs rely heavily on HTTP status codes:

| Code | Meaning |
|---|---|
| 200 | OK |
| 201 | Created |
| 400 | Bad request |
| 401 | Unauthorized |
| 404 | Not found |
| 500 | Server error |

Status codes are an essential part of REST communication.

## Advantages and Limitations

REST is widely adopted, easy to understand, and integrates naturally with HTTP infrastructure. It supports caching, authentication mechanisms, and standard tooling.

However, REST can sometimes be verbose. Complex operations may require multiple endpoints, and over-fetching or under-fetching data can become an issue in large systems.