REST API Introduction

REST API stands for Representational State Transfer API. It is a type of API (Application Programming Interface) that allows communication between different systems over the internet. REST APIs work by sending requests and receiving responses, typically in JSON format, between the client and server.

- A request is sent from the client to the server via a web URL, using one of the HTTP methods.
- The server then responds with the requested resource, which could be HTML, XML, Image, or JSON, with JSON being the most commonly used format for modern web services.
- These methods map to CRUD operations (Create, Read, Update, Delete) for managing resources on the web.

Note => *REST is an architectural design style for APIs, while HTTP is the communication protocol used for data transfer over the web. REST APIs use HTTP methods to interact with resources, but they are not the same thing. REST defines how the APIs should behave, while HTTP defines the rules for communication over the web. They commonly work together, but they serve different purposes.*

Common HTTP Methods Used in REST API

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 that are less frequently used, like OPTIONS and HEAD.

1. GET Method

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).

```
GET /users/123
```

This request fetches data for the user with ID 123.

2. POST Method

The POST method is commonly used to create new resources. It is often used to create subordinate resources related to a parent resource. Upon successful creation, the server returns HTTP status 201 (Created) along with a Location header pointing to the newly created resource.

```
POST /users
{
  "name": "Anjali",
  "email": "gfg@example.com"
}
```

This request creates a new user with the given data.

NOTE: *POST is neither safe nor idempotent.*

3. PUT Method

PUT is an HTTP method used to update or create a resource on the server. When using PUT, the entire resource is sent in the request body, and it replaces the current resource at the specified URL. If the resource doesn’t exist, it can create a new one.

```
PUT /users/123
{
  "name": "Anjali",
  "email": "gfg@example.com"
}
```

This request updates the user with ID 123 or creates a new user if one doesn't exist.

4. PATCH Method

PATCH is an HTTP method used to partially update a resource on the server. Unlike PUT, PATCH only requires the fields that need to be updated to be sent in the request body. It modifies specific parts of the resource rather than replacing the entire resource.

```
PATCH /users/123
{
  "email": "new.email@example.com"
}
```

This request updates only the email of the user with ID 123, leaving the rest of the user data unchanged.

Differences Between PUT & PATCH

Both PATCH and PUT are used to update resources on the server, but they differ in how they handle the update process:

PUT	**PATCH**
Replaces the **entire** resource	Updates **only specified fields**
Must send full data	Only sends changes
Idempotent	Not always idempotent
Example: Updating a user’s **entire profile**	**Example**: Changing just a user’s **email**

5. DELETE Method

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

```
DELETE /users/123
```

This request deletes the user with ID 123.

Idempotence: *An idempotent HTTP method is a HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. Again, this only applies to the result, not the resource itself.*

Building a Simple REST API using Node.js and Express

Now let's create a REST AP and perform the various HTTP operations.

Step 1: Create the folder

Create the NodeJs project by using the following command:

```
mkdir node-app
cd node-app
```

Step 2: Install the package.json

```
npm init -y
```

Step 3: Install Express

To begin building a REST API in Node.js, you need to install Express. Run the following command in your terminal:

```
npm install express
```

Step 4: Create the Server

Here's a basic example of creating a REST API in Node.js using Express

```
// Import the Express module

const express = require('express');

const app = express();

const port = 3000;


app.use(express.json());


// Define a route for GET requests

app.get('/users', (req, res) => {

   res.json({ message: 'Returning list of users' });

});


// Define a route for POST requests

app.post('/users', (req, res) => {
  const newUser = req.body;
  res.json({ message: 'User created', user: newUser });
});

// Define a route for PUT requests
app.put('/users/:id', (req, res) => {
  const userId = req.params.id;
  const updatedUser = req.body;
  res.json({ message: `User with ID ${userId} updated`, updatedUser });
});

// Define a route for DELETE requests
app.delete('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.json({ message: `User with ID ${userId} deleted` });
});

// Start the server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Output: To test the API, open http://localhost:3000 in Postman or another API testing tool.

In this example

- **GET /users:** This route fetches the list of users (or mock data in this case).
- **POST /users:** This route accepts JSON data from the client to create a new user.
- **PUT /users/:id:** This route updates the information of a user based on the user ID provided in the URL.
- **DELETE /users/:id:** This route deletes a user with the specified ID.

Features of REST APIs

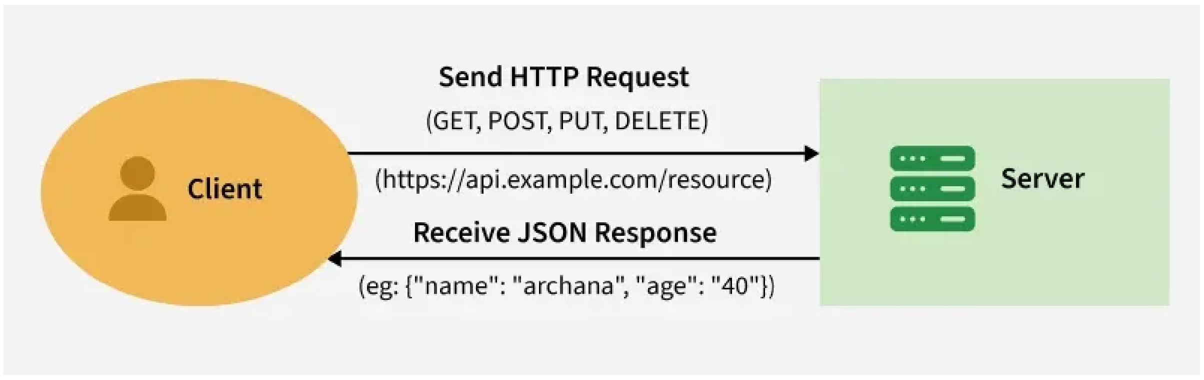
- **Stateless**: Each request from a client to a server must contain all the information the server needs to fulfill the request. No session state is stored on the server.

- **Client-Server Architecture**: RESTful APIs are based on a client-server model, where the client and server operate independently, allowing scalability.
- **Cacheable**: Responses from the server can be explicitly marked as cacheable or non-cacheable to improve performance.
- **Uniform Interface**: REST APIs follow a set of conventions and constraints, such as consistent URL paths, standardized HTTP methods, and status codes, to ensure smooth communication.
- **Layered System**: REST APIs can be deployed on multiple layers, which helps with scalability and security.

Real world Example of Rest APIs :

REST APIs are widely used across various industries to simplify communication between systems. Some common applications include:

- **Social Media:** Integrating third-party platforms like Facebook, Twitter, and Instagram for features like login, sharing, and posting.
- **E-Commerce:** Managing products, processing payments, handling orders, and customer management.
- **Geolocation Services**: GPS tracking, real-time location updates, and location-based services like finding nearby places.
- **Weather Forecasting:** Fetching weather data from external sources to provide real-time weather updates and forecasts.


Client
Send HTTP Request
(GET, POST, PUT, DELETE)
(https://api.example.com/resource)
Receive JSON Response
(eg: {"name": "archana", "age": "40"})
Server