REST API Basics

Here's a beginner-friendly explanation of the **basics of REST API**:

🔹 What is a REST API?

REST stands for **Representational State Transfer**.
An **API** (Application Programming Interface) allows different software systems to communicate with each other.

A **REST API** is a set of rules that allows programs to **communicate over HTTP**, the same protocol used by web browsers.

🔹 Key Concepts

1. Client-Server Architecture

- The **client** (frontend) sends requests.
- The **server** (backend) processes the request and sends a response.

2. HTTP Methods (Verbs)

REST APIs use standard HTTP methods to perform actions:

HTTP Method	Description	Example
`GET`	Read data	Get a list of users
`POST`	Create new data	Add a new user
`PUT`	Update existing data	Update an entire user profile
`PATCH`	Partially update data	Update a user's email
`DELETE`	Remove data	Delete a user

3. Resources & Endpoints

- Everything in REST is a **resource** (e.g., user, product).
- Each resource is accessed via a **URL (endpoint)**.

Example:

```
nginx

GET https://api.example.com/users/1
```

→ Gets details of user with ID `1`.

4. Request and Response

- A **request** contains the method, URL, headers, and sometimes a body.
- A **response** contains a status code, headers, and optionally a body.

---

5. Status Codes

Common HTTP response codes:

Code	Meaning	Description
200	OK	Request succeeded
201	Created	Resource successfully created
400	Bad Request	Malformed request
401	Unauthorized	Auth required/not valid
404	Not Found	Resource doesn't exist
500	Internal Server Error	Server failed to respond

---

6. JSON Format

Most REST APIs use **JSON** for sending and receiving data.

Example JSON Response:

```
json

{
  "id": 1,
  "name": "John Doe",
  "email": "john@example.com"
}
```

🔹 Example REST API Interaction

✅ GET Request

Request:

```
bash

GET /users/1
```

Response:

```
json

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

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

✅ POST Request

Request:

```
bash

POST /users
Content-Type: application/json
```

Body:

```
json

{
  "name": "Bob",
  "email": "bob@example.com"
}
```

Response:

```
201 Created
```

🔹 Characteristics of REST

- **Stateless** – Each request is independent.
- **Cacheable** – Responses can be cached.
- **Uniform Interface** – Same approach to access resources.
- **Client-Server Separation** – Frontend and backend are separated.

Would you like to try a hands-on REST API testing using tools like **Postman** or **curl**?

Great! Let's now break down the **Headers** and **Body** in a REST API request and response.

🔹 1. Headers

📌 What are Headers?

Headers are key-value pairs sent in both **requests** and **responses**.
They carry metadata (information *about* the request or response).

✅ Common Request Headers

Header	Purpose	Example
`Content-Type`	Tells the server the data format being sent	`application/json`
`Authorization`	Sends credentials or tokens	`Bearer <token>`
`Accept`	Tells the server what data format the client wants	`application/json`
`User-Agent`	Info about the client making the request	`PostmanRuntime/7.36.0`

✅ Common Response Headers

Header	Purpose	Example
`Content-Type`	Format of the response data	`application/json`
`Cache-Control`	Caching rules	`no-cache`
`Server`	Info about the server	`nginx/1.21.6`
`Set-Cookie`	Sends cookies to the client	`sessionId=abc123; Path=/`

🔹 2. Body

📌 What is the Body?

The **body** contains the **actual data** being sent with the request or received in the response.

✅ Request Body (Used in POST, PUT, PATCH)

You send the **request body** when you want to send data to the server (e.g., to create or update a resource).

Example: POST Request Body

```
json

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

- You must set the header:

```
pgsql

Content-Type: application/json
```

✅ Response Body

The server sends back a **response body** with data (usually in JSON).

Example: JSON Response Body

```
json

{
  "id": 101,
  "name": "Alice",
  "email": "alice@example.com"
}
```

🔹 Complete Example: POST Request

Endpoint:

```
nginx

POST https://api.example.com/users
```

Request Headers:

```
pgsql

Content-Type: application/json
Authorization: Bearer abc123
```

Request Body:

```
json

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

Response (201 Created):

```
json

{
  "id": 101,
  "name": "Alice",
  "email": "alice@example.com"
}
```

Would you like to try testing this using **Postman** or **curl**, or should we move to **query parameters** and **path variables** next?

Great question! Understanding **Path Parameters** and **Query Parameters** is essential when working with REST APIs.

🔹 Path Parameters

📌 What are Path Parameters?

Path parameters are **part of the URL path**, and are used to identify **a specific resource**.

✅ Example:

```
bash

GET /users/101
```

- `101` is a **path parameter**.
- It refers to a specific **user** with ID 101.

Full URL:

```
arduino

https://api.example.com/users/101
```

🔸 Path parameters are **mandatory** and usually not optional.

✅ URL Format:

```
bash

/resource/{id}
```

Example:

```
http

GET /products/45
```

→ Fetch product with ID 45.

🔹 Query Parameters

📌 What are Query Parameters?

Query parameters appear **after a `?` in the URL** and are used to **filter, sort, or paginate** data.

✅ Example:

```pgsql
GET /users?role=admin&active=true
```

- **`role=admin`** and **`active=true`** are **query parameters**.
- They're used to filter the list of users.

Full URL:

```pgsql
https://api.example.com/users?role=admin&active=true
```

🔸 Query parameters are usually **optional**.

✅ URL Format:

```bash
/resource?key1=value1&key2=value2
```

Example:

```http
GET /products?category=books&sort=price_desc
```

→ Get books sorted by price in descending order.

🔹 Key Differences

Feature	Path Parameter	Query Parameter
Location	In the URL path	After the **?** in the URL
Purpose	Identify a **specific** item	Filter/search/multiple options
Mandatory?	Usually **yes**	Usually **no** (optional)
Format	**`/resource/{id}`**	**`?key=value&key2=value2`**
Example	**`/users/123`**	**`/users?name=John&sort=asc`**

✅ Combined Example:

```bash
GET /users/123/orders?status=shipped&limit=5
```

- Path: **`/users/123/orders`** → Orders of user ID 123
- Query: **`status=shipped&limit=5`** → Only 5 shipped orders

Would you like to practice with a live API or simulate this using Postman or curl commands?

Excellent! Let's look at how **JSON** and **XML** are used in APIs, especially REST APIs.

🔹 What are JSON and XML?

Both **JSON (JavaScript Object Notation)** and **XML (eXtensible Markup Language)** are data formats used to:

- **Send data in requests** (e.g. POST, PUT)
- **Receive data in responses** (e.g. GET)

🔸 1. JSON (Most common today)

✅ Example JSON:

```
json

{
  "id": 101,
  "name": "Alice",
  "email": "alice@example.com"
}
```

🔹 Pros of JSON:

- Lightweight and easy to read
- Native to JavaScript (but works in all languages)
- Faster to parse

🔹 Content-Type for JSON:

```
http

Content-Type: application/json
```

🔸 2. XML (Older but still used in some APIs)

✅ Example XML:

```
xml

<user>
  <id>101</id>
  <name>Alice</name>
  <email>alice@example.com</email>
</user>
```

🔹 Pros of XML:

- Supports complex structures, attributes, and namespaces
- Still used in SOAP APIs, older systems, banking, etc.

🔹 Content-Type for XML:

```
http

Content-Type: application/xml
```

🔹 REST API Example in Both Formats

✅ JSON Request:

Request:

```
bash

POST /users
Content-Type: application/json
```

Body:

```
json

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

✅ XML Request:

Request:

```
bash

POST /users
Content-Type: application/xml
```

Body:

```
xml

<user>
  <name>Alice</name>
  <email>alice@example.com</email>
</user>
```

🔹 How to Tell Server What Format You Want

Use the **Accept** header to tell the server what kind of response you want:

Purpose	Header
Send JSON	`Content-Type: application/json`
Send XML	`Content-Type: application/xml`
Accept JSON response	`Accept: application/json`
Accept XML response	`Accept: application/xml`

🔹 Summary

Feature	JSON	XML
Format	Lightweight, human-readable	Verbose, tag-based
Popularity	Most modern APIs	Used in older or SOAP APIs
Parsing Speed	Faster	Slower
Content-Type	`application/json`	`application/xml`

Would you like to test both formats using **Postman** or see a **Java or Python** example using them in an API client?