Why we are studying this?

pandas → Python API

functions, modules, classes

Why? → .csv → .xlsx → Easy access to data

Data ingestion

Databases, Server → Exposed

local.

helps you to get data.

Data is costly

Offline, online

DE, Analysis, DS, ML, DL, LLM, Gen AI

Data → Key

API → Powerful

letter → document

data.

API ← talk ← x → Y

API

Postal system

black box

→ read_csv(_) path

df

API

class, subject, lang

API → NCERT

API → NCERT

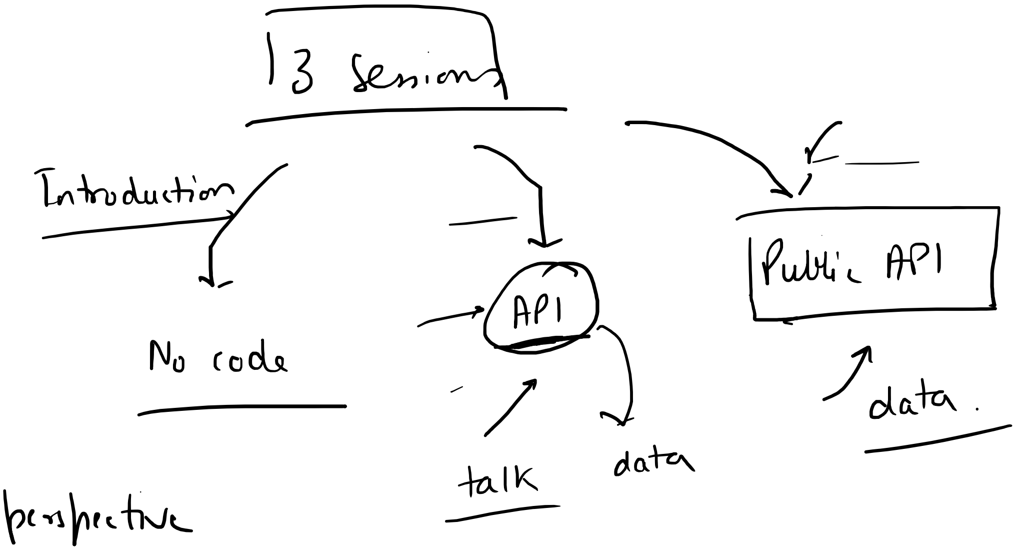
User / demand → Weather

API → Weather

Weather Satellite

location, Time,

°C, °F, °K



user's perspective

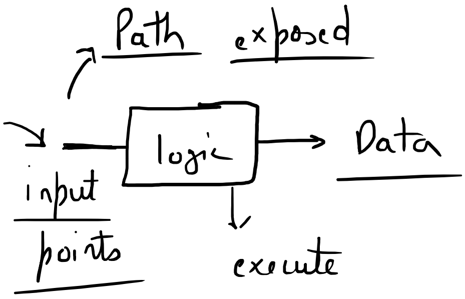
Why do we study about API:

Excel files, .csv data → clean, well organised.

real life → incomplete

API

- ✓ access to public data.
- ✓ incomplete
- ✓ messy
- ✓



- messy

points   execute

Costly →

Application →

Programming → access

interface → access point

Application Programming interface

API → logic implemented

API → access points

logic

read_csv() ← path

concat()

merge()   value counts()

Documentation

Numpy

Pandas

Python API

Real world data → Messy
→ Difficult to gather
→ Costly
→ Incomplete

APIs

Demand of cabs → Weather

→ access point → API → Weather information

|point|

(city, date, time)

},

Temp, weather pattern,

forecast

Introduction

request → API → response System → Weather satellite

◈ What is an API?

An **API (Application Programming Interface)** is a set of rules that allows one piece of software or system to interact with another. Think of it as a **messenger** that takes requests, tells a system what you want to do, and returns the response.

API secure

◈ Why are APIs important?

- They allow applications to communicate with each other.
- Enable access to **data** and **functionality** without exposing internal code.
- Widely used in **data science** to fetch data from sources like Twitter, weather services, stock markets, etc.

◈ Real-life analogy:

Imagine a restaurant:

- You (the client) want food (data).
- The waiter (API) takes your order to the kitchen (server) and brings back your food (response).
- You don't need to know how the kitchen works — you just use the menu (API documentation).

◈ Common Examples:

- **Weather API** – Get current weather or forecasts. ✓
- **Twitter API** – Access tweets, users, and trends. ✓
- **Google Maps API** – Get directions, places, geolocation. ✓

counter windows.

API → Bank →

form input

↓

Respose

Account

Cards.

loans

Sentiment analysis Tweets

API → twitter accounts

↓

tweets

Analogy

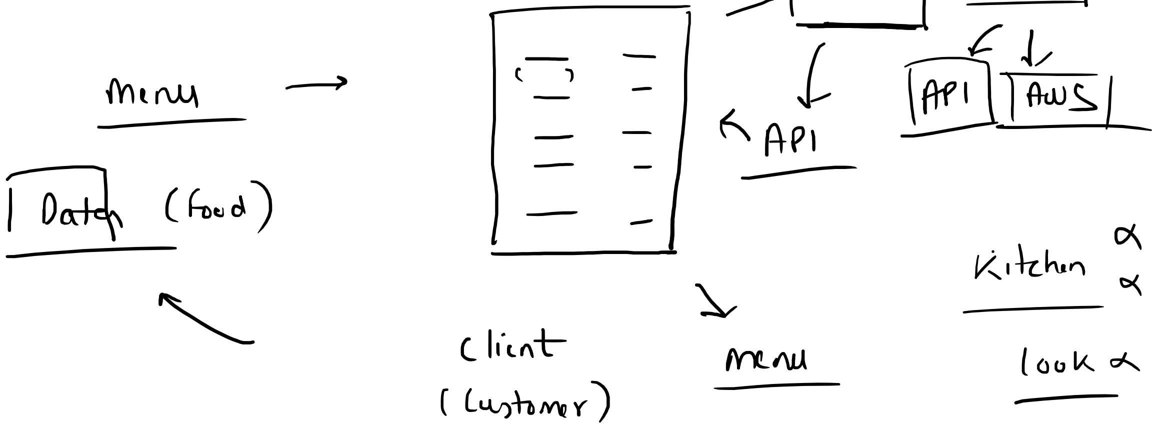
🍽️ **Restaurant Analogy for APIs**

- **You** are the **client** (the one making the request).
- The **menu** is the **API documentation** — it tells you what is available and how to ask for it (e.g., endpoints, parameters).
- The **waiter** is the **API itself** — the one who takes your order (request) and delivers it to the kitchen.
- The **kitchen** is the **server** — it processes your request and prepares the response (data). → food
- Finally, the **waiter brings back the dish** (response) from the kitchen to your table.

Example:
You want to know today's weather in Delhi.

- You check the **API documentation** (menu) and find there's a GET /weather endpoint.
- You send a **GET request** to that endpoint with parameters like city=Delhi.
- The **API (waiter)** passes your request to the **server (kitchen)**.
- The server fetches the weather data and the API sends it back to you — like getting your food at the table.

Restaurant Analogy



waiter → — — — API

Waiter → Documentation

→ take order

→ serve food

Bill

Order → cook

exposed → menu

cook → Function logic → dish → waiter

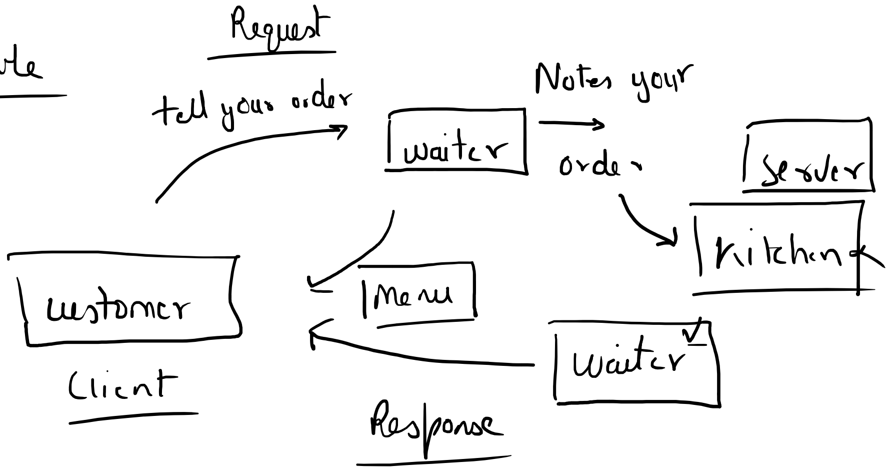
API → waiter → API → Limited powers

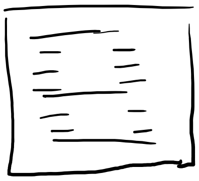
API ↓ Data.

Public API → Open AI API → model

Restaurant ↓ have food

table





Request → API → Response

API ↓ waiter

Menu → Documentation → access points

Client → access points

REST APIs

Internet → request ↓ response ←

✓ What is a REST API ?

A **REST API (Representational State Transfer)** is a way for two systems (like your app and a server) to talk to each other over the internet using simple **HTTP requests** — like GET, POST, PUT, DELETE.
It follows some rules and principles to make communication simple, fast, and scalable.

HTTP Methods in REST:

HTTP methods

- GET → Get data ✓
- POST → Add new data ✓
- PUT → Update data ✓
- DELETE → Remove data ✓

JSON ↓ Universal

Advantages of REST API:

→ addresses

1. **Easy to use** – Uses simple HTTP and URLs.
2. **Lightweight** – Mostly uses JSON (easy to read and process).
3. **Stateless** – Each request is independent; server doesn't store session info.
4. **Flexible** – Works with any language (Python, JavaScript, etc.).
5. **Scalable** – Can handle large numbers of requests efficiently.

dict ← JSON

lightweight → Hardware reqs

HTTP requests → API over the internet ↓ URL

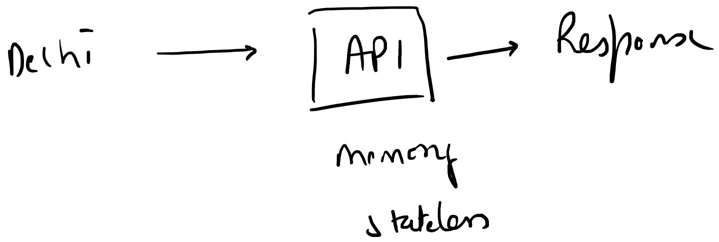
4 types → Get → URL → data.

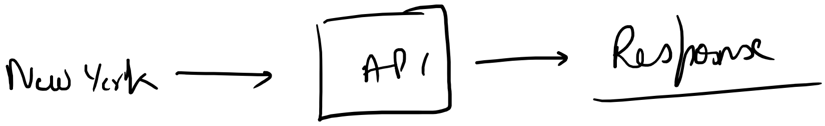
Post → URL → send data

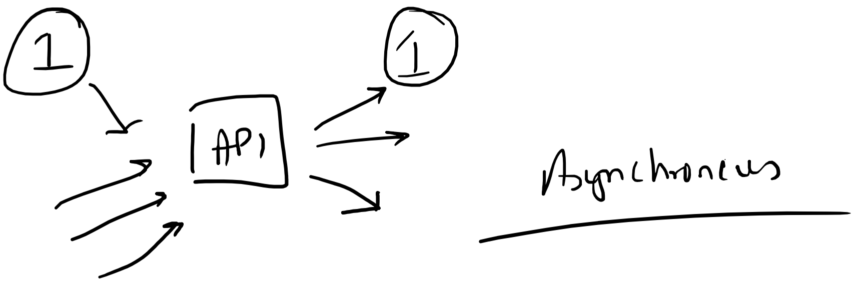
Put → Edit → update

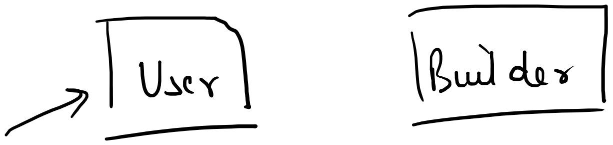
Delete → Delete

JSON →


Delhi
API
Response
memory
stateless


New York
API
Response


1
API
1
Asynchronous


User
Builder

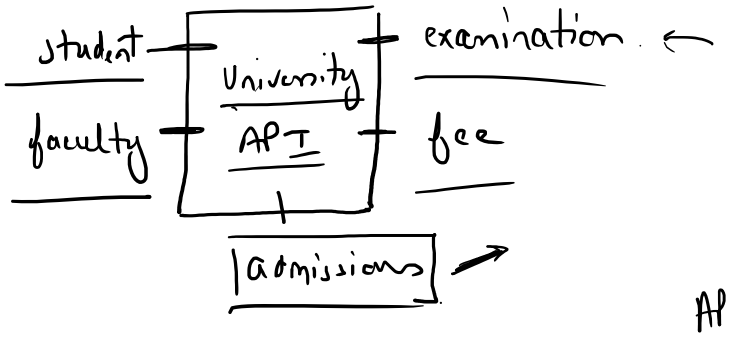
Components

1) Endpoints → Segmentation

2) Request → Request method.

3) Response

Endpoint. → Windows to the inside.



API → student
Roll : 101
name: 'XYZ'.

Request → Get → Attachment α
Post → Data
Put → update
Delete → Delete.

API → CRUD operation

CRUD → API

C → Create → Post

R → Read → Get

U → Update → Put

D → Delete → Delete

Response → Get → Data.

Post → Status message

Response

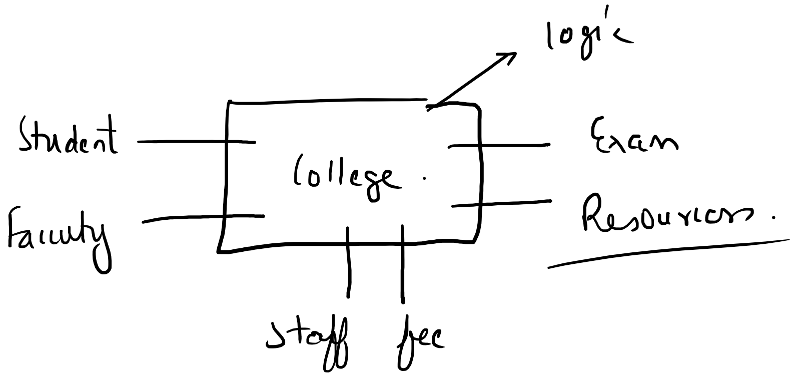
Post → Status message

Put → Status

Delete → Status

1) Endpoint → Access points → URL's → segments

menu



2) Request → Body → Authorisation . → Input

3) Response → Body → Output

CRUD databases .

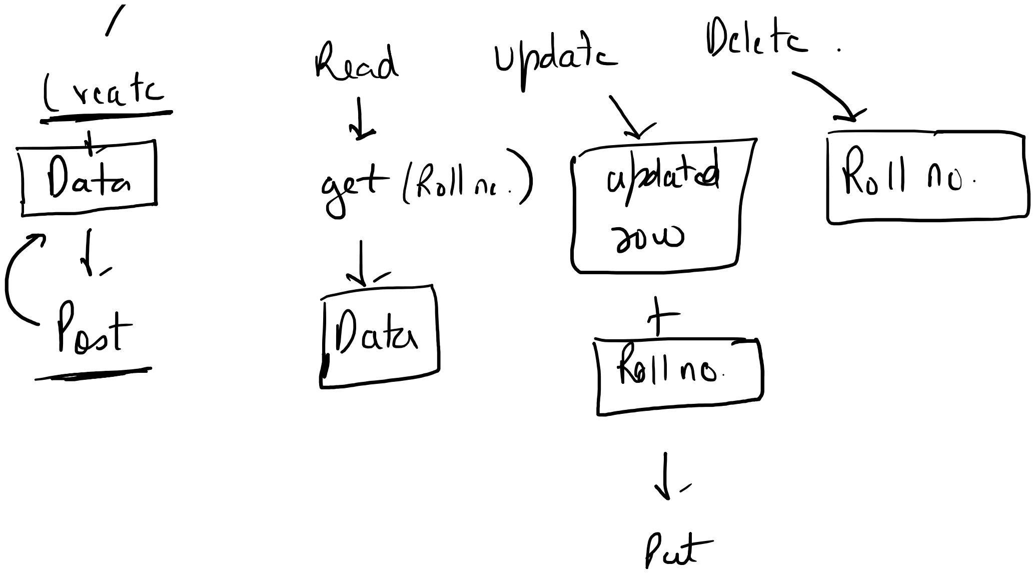
4 HTTP methods

C / Create

R ↓ Read

U ↓ Update

D ↓ Delete .

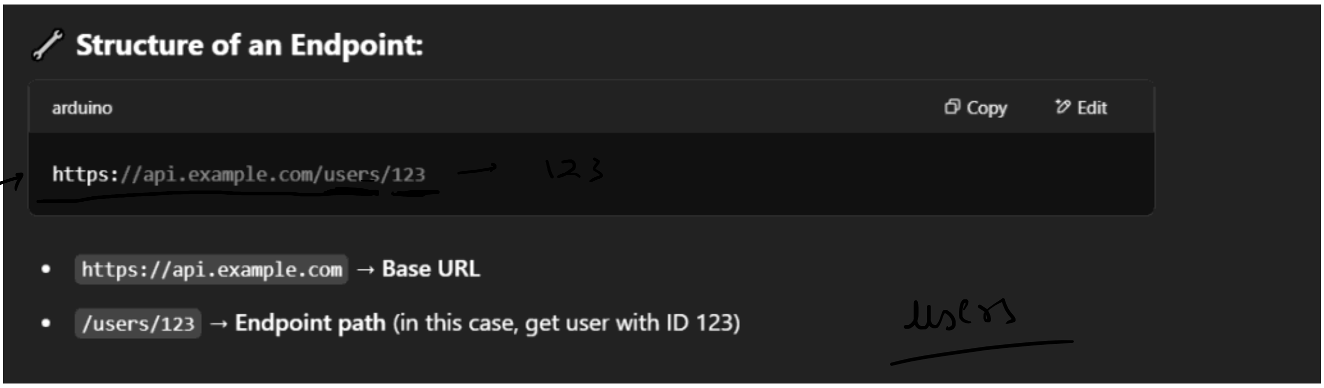


HTTP methods → CRUD operations.

Endpoint

🔷 What are Endpoints in an API?

An **endpoint** is a specific **URL** where an API can be accessed to perform a particular action or get specific data.
Think of it as a **door** to a specific resource or service on the server.



🔧 Structure of an Endpoint:

```arduino
https://api.example.com/users/123
```

123

- `https://api.example.com` → **Base URL**
- `/users/123` → **Endpoint path** (in this case, get user with ID 123)

users

📌 Examples of Endpoints:

- GET /users → Get list of all users
- GET /users/45 → Get user with ID 45
- POST /users → Add a new user
- DELETE /users/45 → Delete user with ID 45

Each endpoint is tied to an **HTTP method** (GET, POST, etc.) and performs a specific task.

📦 Why Endpoints Matter

They define:
- What data you can access
- How you access it
- What actions you can perform on it → methods

Post → C → Create
↑ ↘
Data Error

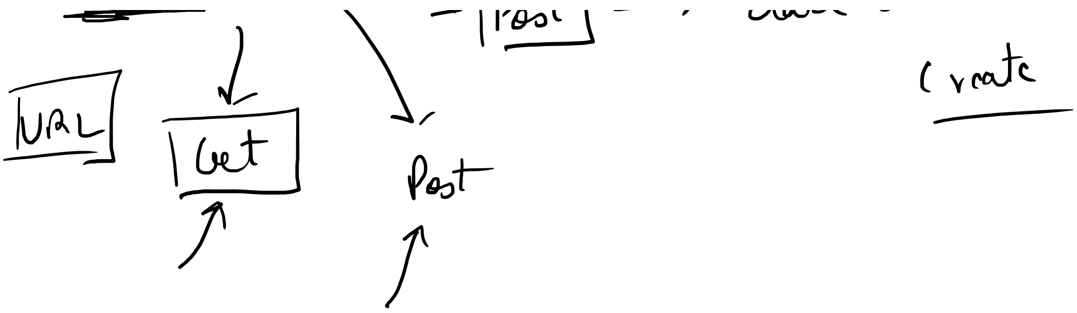
Endpoint → Method

Reading.

/users → Get → Users data

→ Post → data add

Create


Post
URL
Get
Post
Create

HTTP Methods

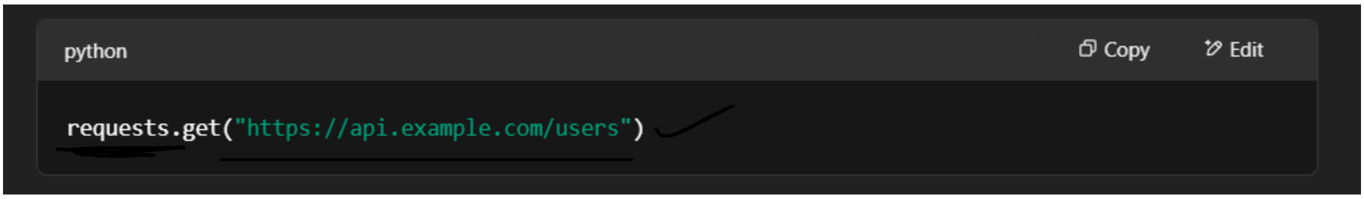
◈ Common HTTP Methods in APIs

endpoint

HTTP methods define what kind of action you want to perform on a resource through an API.
Here are the **4 most commonly used** methods in REST APIs:

☑ 1. GET – Retrieve data ✓

- Used to fetch data from the server.
- **Does not modify** anything.

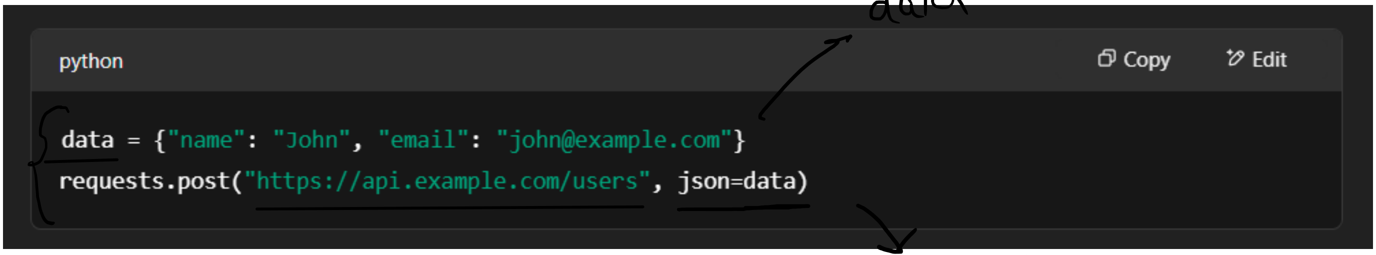


```python
requests.get("https://api.example.com/users") ✓
```

🆕 2. POST – Create new data

- Sends new data to the server to create a resource.

data

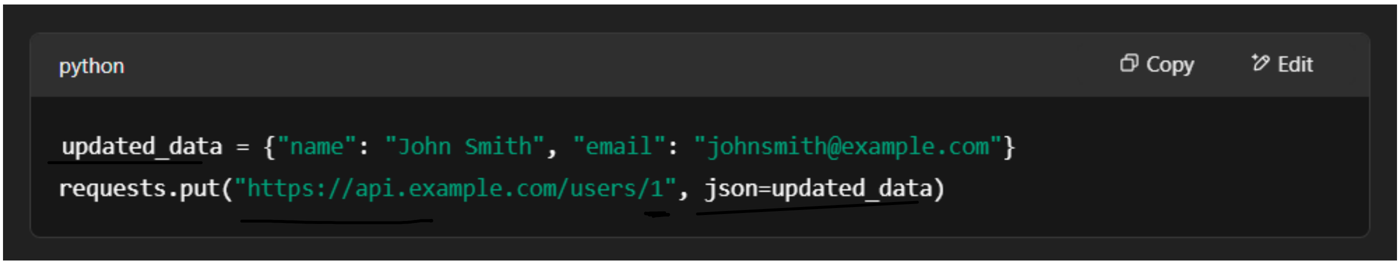


```python
data = {"name": "John", "email": "john@example.com"}
requests.post("https://api.example.com/users", json=data)
```

Endpoints → 2

🔁 3. PUT – Update existing data

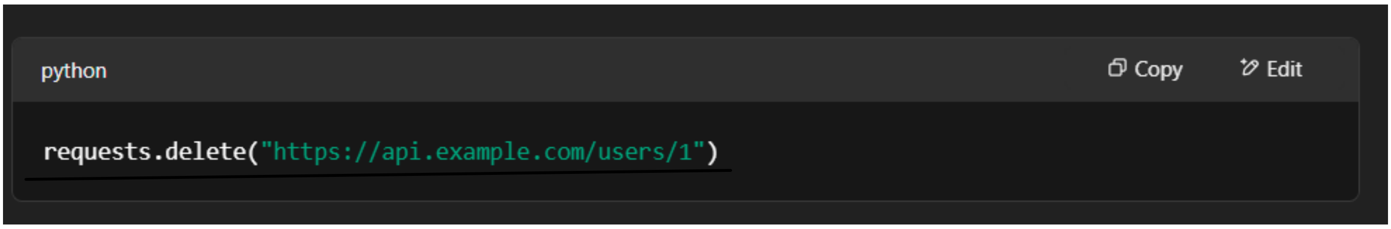
- Replaces the entire resource with new data.



```python
updated_data = {"name": "John Smith", "email": "johnsmith@example.com"}
requests.put("https://api.example.com/users/1", json=updated_data)
```

✕ 4. DELETE – Delete data

- Removes a specific resource.



```python
requests.delete("https://api.example.com/users/1")
```

📌 Less common methods:

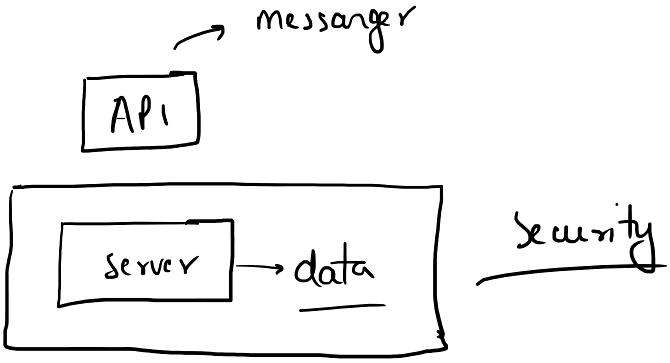
- PATCH → Partially update a resource

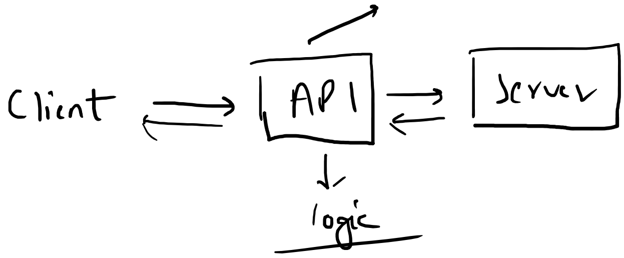
📌 **Less common methods:**

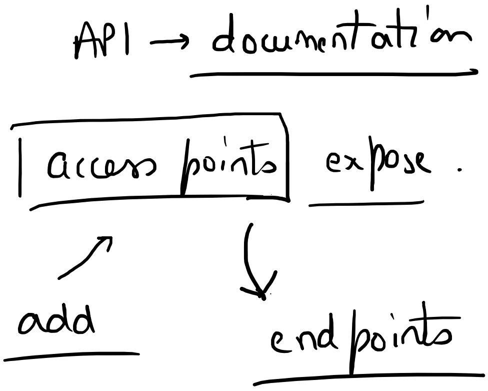
- PATCH → Partially update a resource
- HEAD → Same as GET, but returns only headers

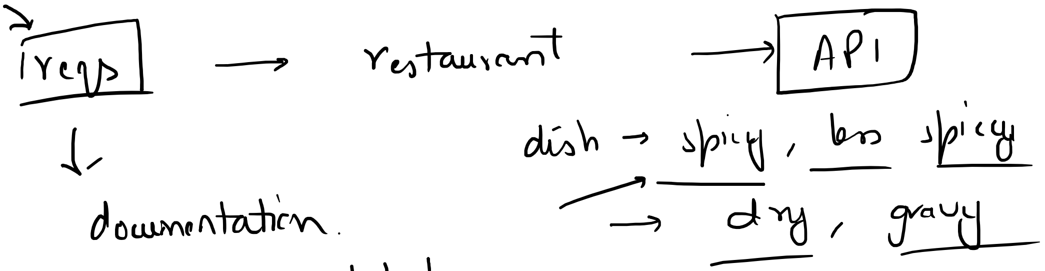
Recap

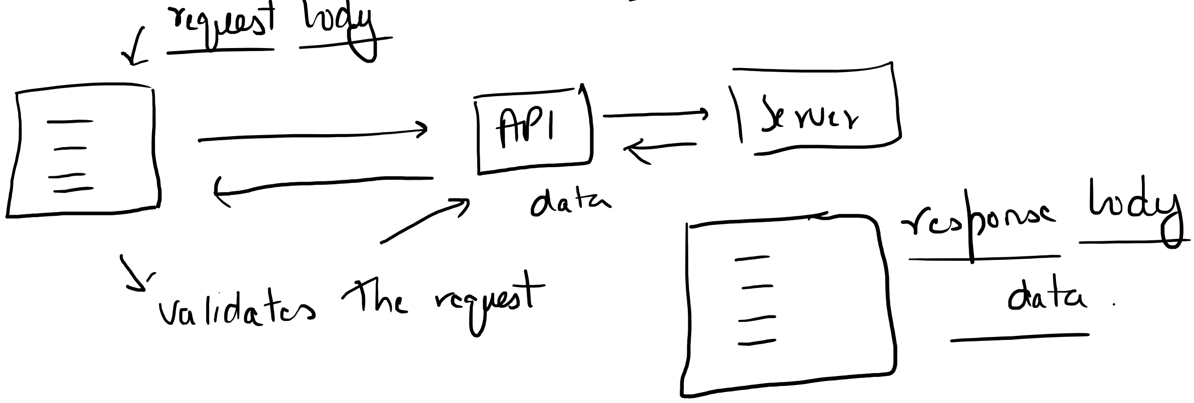
01 May 2025 20:07











REST API's → internet HTTP requests

methods → C → Create → Post

R → Read → Get

U → Update → Put

D → Delete → Delete .

D → Delete → Delete .

endpoints → number
endpoints → address
endpoints → functionality
endpoints → HTTP method
endpoints → inputs / outputs

} Builder side

get

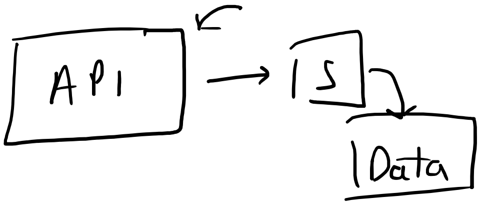
read_csv()

to_csv()

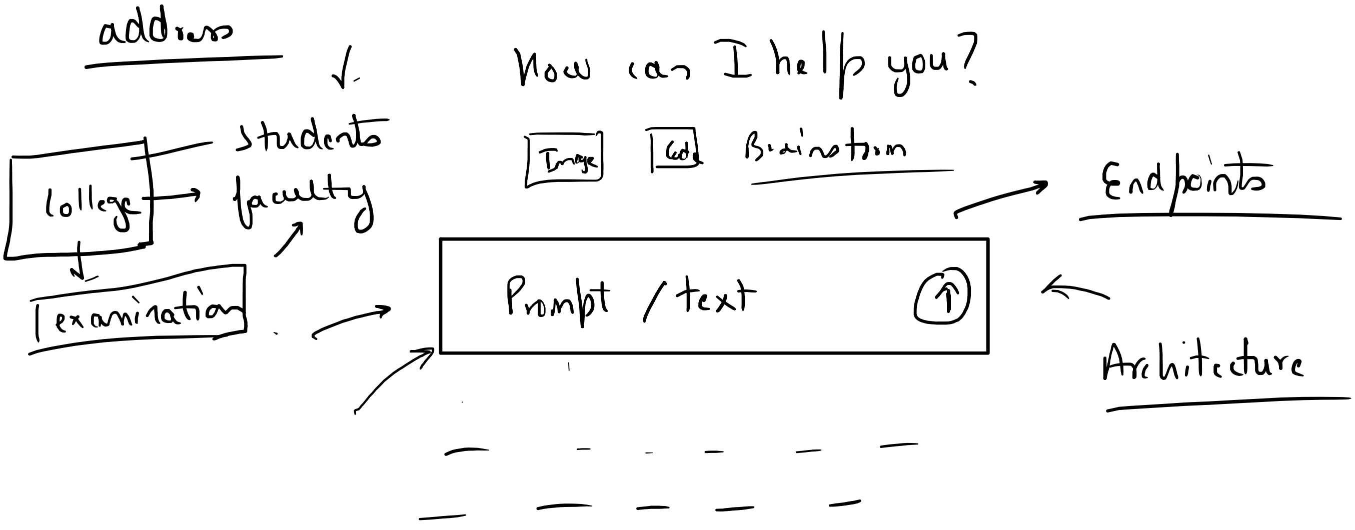
} csv files

post

functions in python.



ChatGPT → Interface



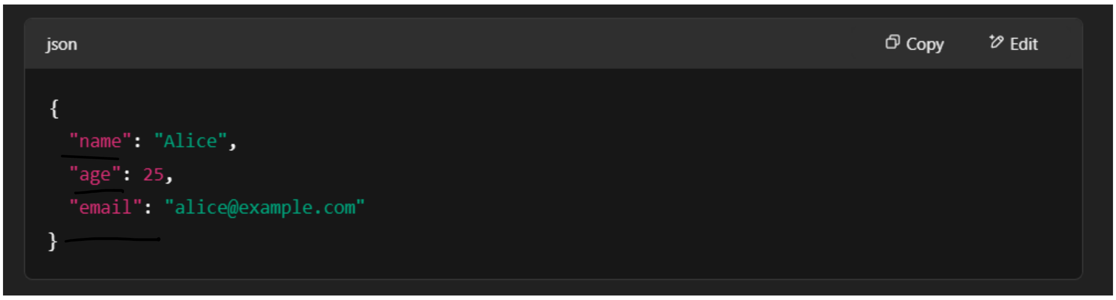
JSON

English of data

◈ What is JSON?

JSON (JavaScript Object Notation) is a lightweight data format used to store and exchange data. It is easy for humans to read and write, and easy for machines to parse and generate.

True



```json
{
  "name": "Alice",
  "age": 25,
  "email": "alice@example.com"
}
```

dict1

dict1["age"]

YAML → MLOPs

🔍 Why is JSON used in APIs?

1. ☑ **Lightweight** – Smaller in size compared to XML.
2. 🤖 **Language-independent** – Can be used across Python, JavaScript, Java, etc.
3. 🔄 **Easily parsed** – Languages like Python have built-in support to parse JSON. → dictionary
4. 🌐 **Web-friendly** – JSON is a native format in JavaScript, making it ideal for web APIs.
5. 📊 **Structured** – Perfect for sending nested or structured data (like lists, objects).

API → language → JSON

data

data → Post

json

request

response → JSON

import json

Python JSON Parse → Dictionary.

{ "name": { "first": "Rahul", "last": "Kumar" } }

YAML

name:

first: Rahul.

last: kumar.

pandas read_json →

to_csv

|to_csv|

last: kumar.

Age: 32

Subjects:

- English
- maths

Request Body

{ name : ___ ,
add : ___ ,
}

📤 What is a Request Body?

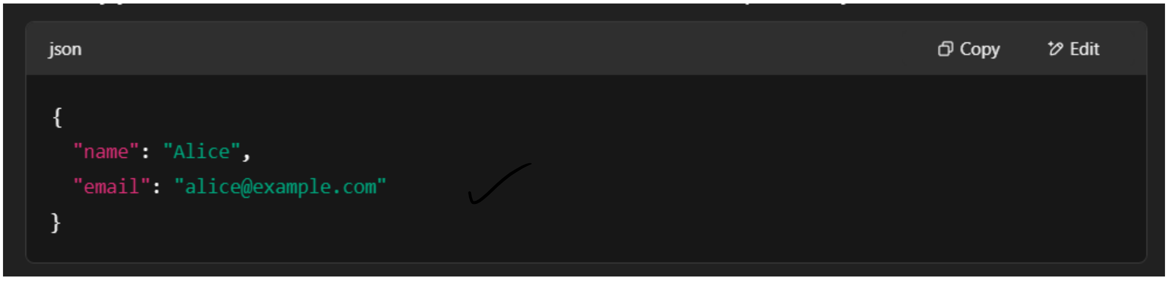
The **request body** is the **data you send to the server** when making an API call. It's like filling out a form and submitting it.

- You usually send a request body with **POST**, **PUT**, or **~~PATCH~~** requests.
- This body contains the **details the server needs** to process your request.
- The data is typically sent in **JSON format**, although other formats (like XML, form-data) are possible.
- Usually in **JSON format**. ✓
- Sent in the **body** of the HTTP request.

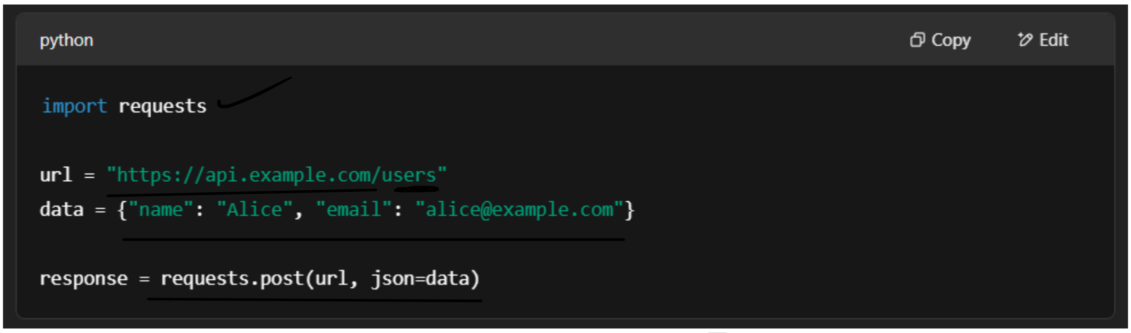
Post

Post → Create → add

Put → Update → Updated data.



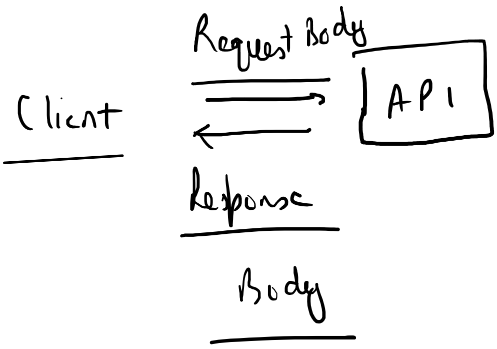
```json
{
  "name": "Alice",
  "email": "alice@example.com"
}
```

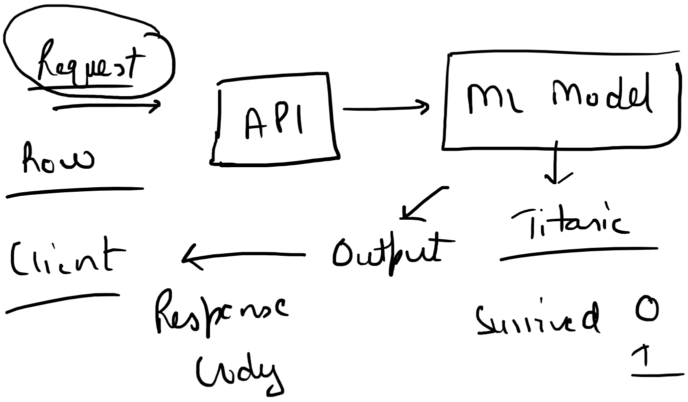


```python
import requests

url = "https://api.example.com/users"
data = {"name": "Alice", "email": "alice@example.com"}

response = requests.post(url, json=data)
```





metadata → data for your data

↓

time of creation

Post → expose α

time of creation   Post → expose α

date

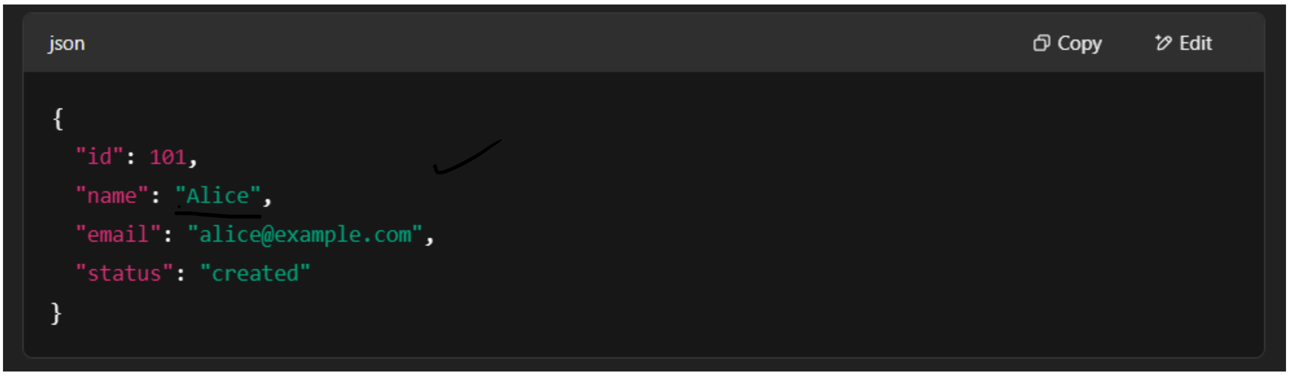
modify

Response Body

📥 What is a Response Body?

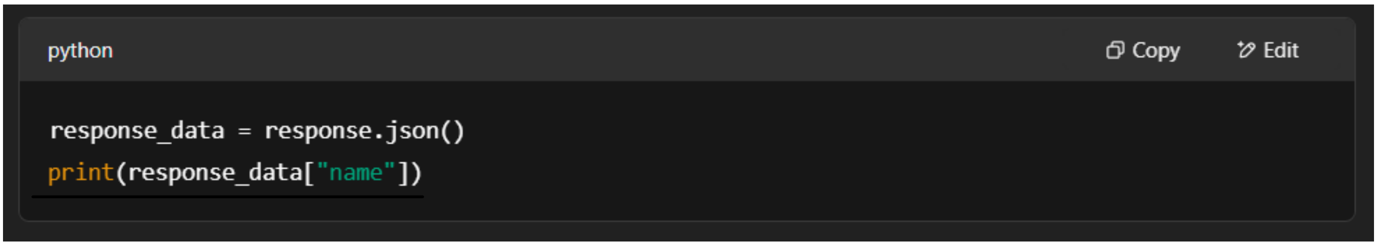
The **response body** is the **data the server sends back to you** after processing your request.

- It usually contains useful information, like:
  - The data you asked for (in GET requests)
  - A confirmation of what was created or changed (in POST/PUT)
  - An error message, if something went wrong
- Like the request body, it's most often in **JSON format**.



json

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

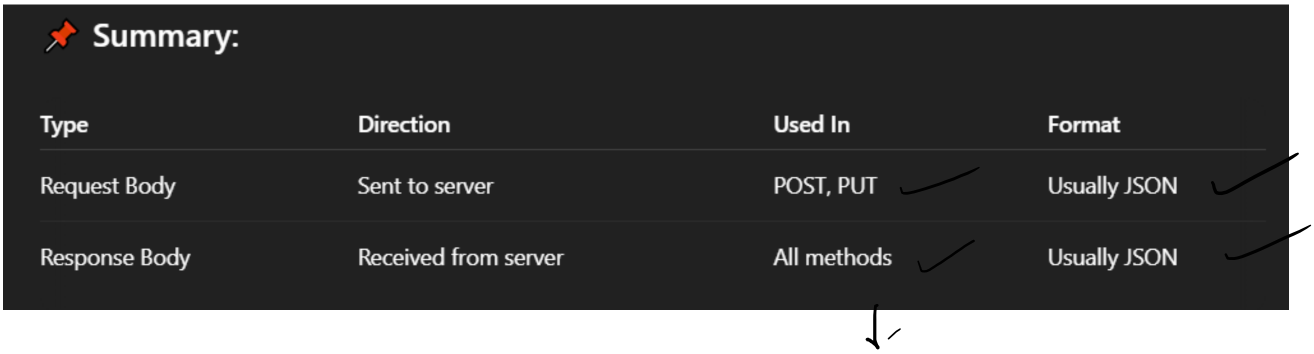


python

```python
response_data = response.json()
print(response_data["name"])
```

🧠 Analogy:

- **Request Body** = What you tell the server ("Here's the user I want to create").
- **Response Body** = What the server tells you back ("User created successfully, here are the details").



📌 **Summary:**

Type	Direction	Used In	Format
Request Body	Sent to server	POST, PUT	Usually JSON
Response Body	Received from server	All methods	Usually JSON

Get

Headers

◈ What are Headers in an API?

Headers are key-value pairs sent with both the **request** and the **response** in an API call. They contain important **metadata** about the request or response, but **not the actual data** (which is in the body).

📤 Request Headers

When you make an API call, **headers** are used to provide additional information that the server may need to process the request properly.

- **Authorization**: Used to pass credentials (like API keys or tokens) for authentication.
- **Content-Type**: Specifies the format of the data being sent in the request body (e.g., JSON, XML, etc.). → JSON
- **Accept**: Tells the server which data formats the client can handle (like JSON or XML).
- **User-Agent**: Contains information about the client making the request (browser, app, etc.).

Python

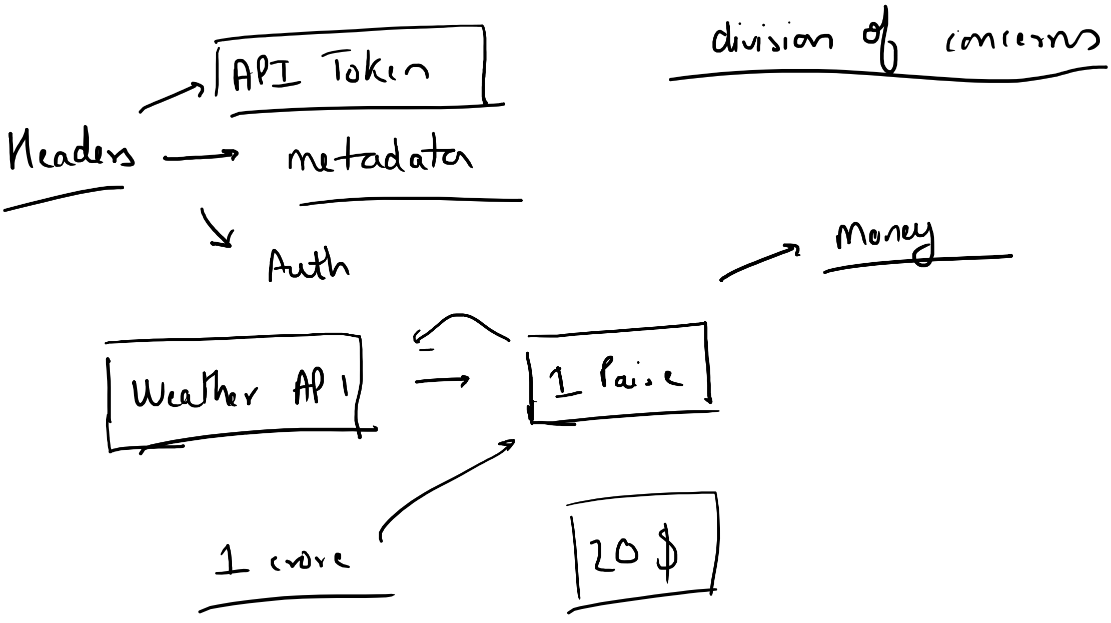
📥 Response Headers

These are sent by the server in the response to provide additional details about the response.

- **Content-Type**: Specifies the format of the data returned (e.g., JSON, XML). ✓
- **Status**: Provides status information about the request (e.g., HTTP status code). ✓ get
- **Location**: Often used in response to a POST request to provide the URL of the newly created resource.

📌 Why are Headers Important?

- **Security**: Headers like Authorization are crucial for identifying who is making the request.
- **Content Negotiation**: The Accept and Content-Type headers allow you to specify and understand the format of the data you're exchanging.
- **State Management**: Headers like Cache-Control help in managing how responses are cached or handled.
- **Efficiency**: By passing relevant information in headers, you avoid unnecessary data being sent in the body, making the request faster.



Chat GPT → GPT → '4o'

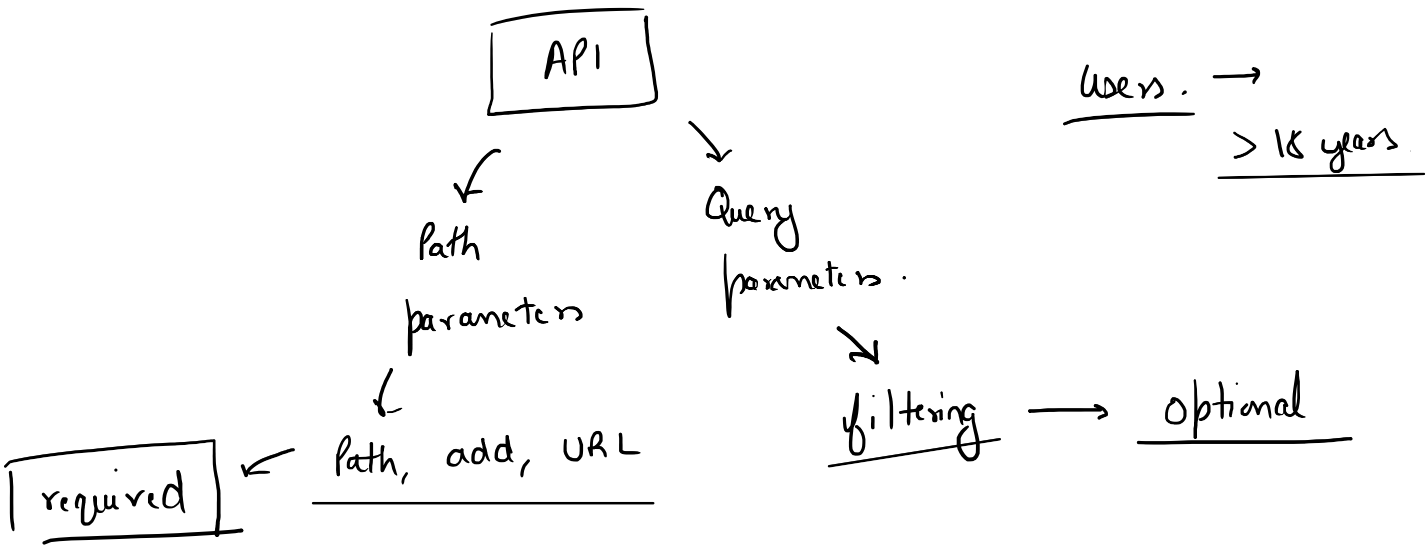
↓ Authorization

↓ 10 chats

Parameters

Parameters → Python, function.

↓ input, function → control → parameters.



API

Path parameters

Query parameters.

Path, add, URL

required

filtering → optional

Users. → > 18 years

```
def test (a, b):
    pass
```

```
test ()
```

```
def test (a, b=0):
    pass.
```

→ test (1, 0)

→ test (1, 2)

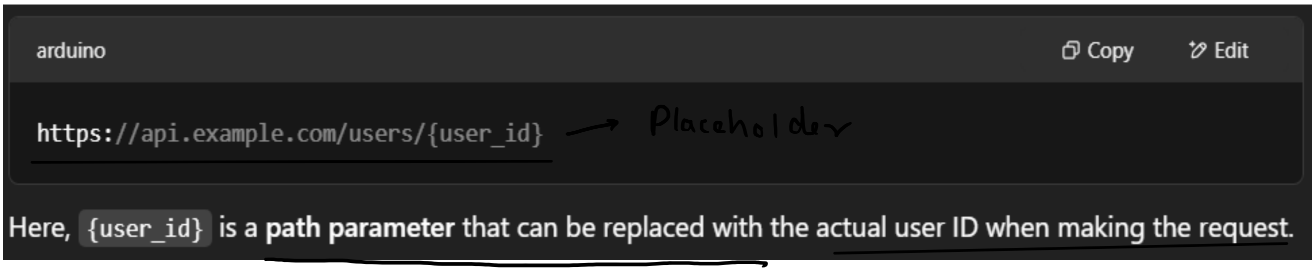
✓ → test ()

Path Parameters

◈ What are Path Parameters in APIs?

Path parameters are values that are part of the URL and are used to identify a specific resource or provide additional information in the request. They are placed **within the URL path**, typically following a specific pattern (usually denoted by curly braces {} in API documentation).
Path parameters are often used in **REST APIs** to uniquely identify a resource or specify something dynamic (like a user ID or product ID).

/121

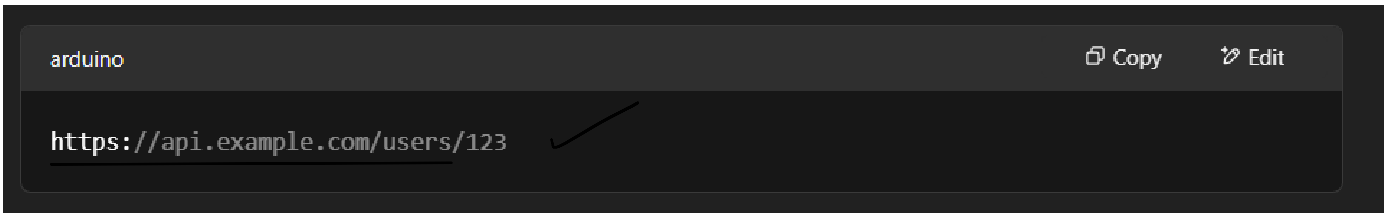


arduino

```
https://api.example.com/users/{user_id}
```

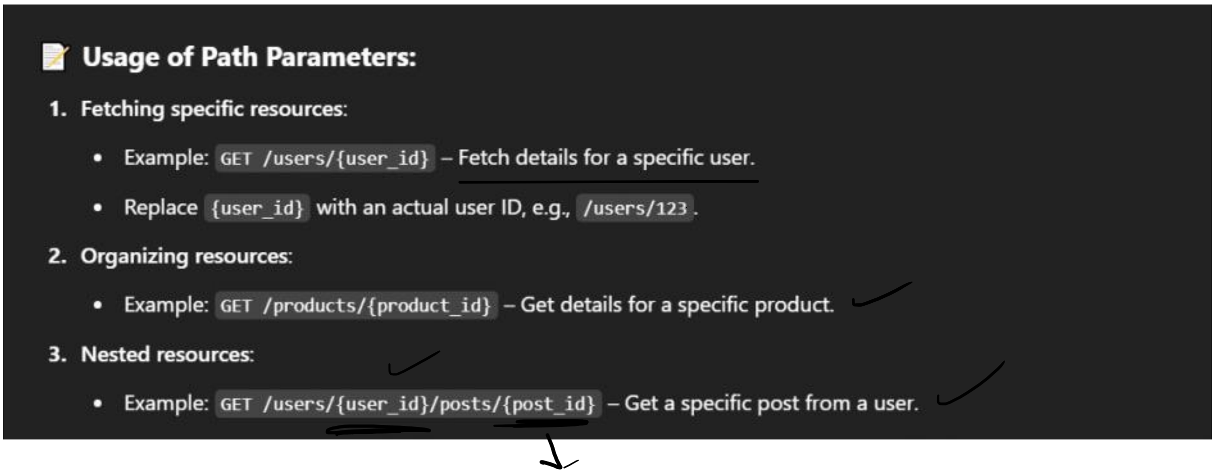
Placeholder

Here, `{user_id}` is a **path parameter** that can be replaced with the actual user ID when making the request.



arduino

```
https://api.example.com/users/123
```



📝 Usage of Path Parameters:

1. **Fetching specific resources:**
   - Example: `GET /users/{user_id}` – Fetch details for a specific user.
   - Replace `{user_id}` with an actual user ID, e.g., `/users/123`.
2. **Organizing resources:**
   - Example: `GET /products/{product_id}` – Get details for a specific product.
3. **Nested resources:**
   - Example: `GET /users/{user_id}/posts/{post_id}` – Get a specific post from a user.

tweet

format , {

Endpoint, method, JSON, parameters.

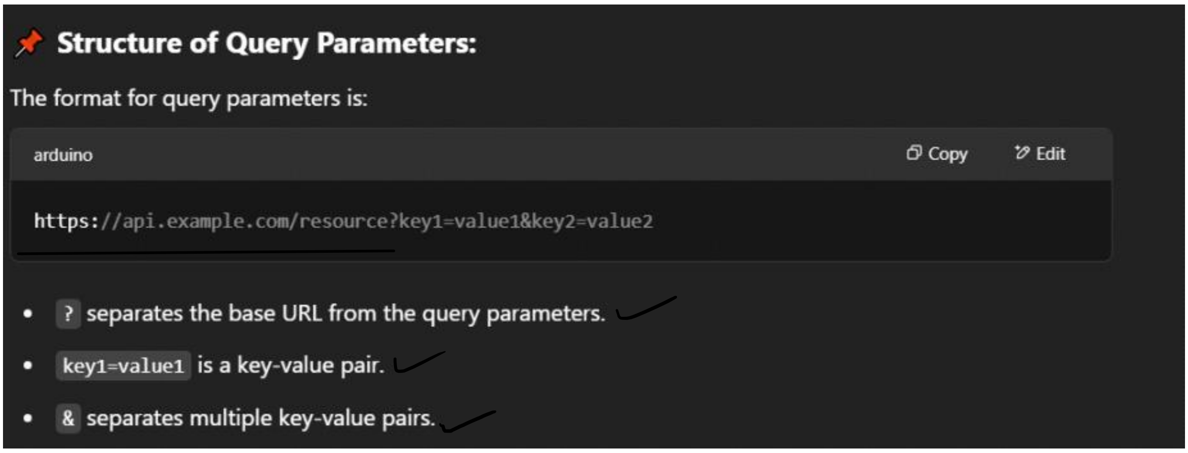
Query Parameters

? $Key_1 = Value_1$ $Key_2 = Value_2$.

◈ What are Query Parameters in APIs?

Query parameters are additional key-value pairs added to the **URL** after a question mark (?). They are used to **filter, modify, or customize** the data being requested without changing the primary resource specified in the path.
Query parameters are used to **refine or filter** the data returned by the API, but they do **not identify a specific resource** (like path parameters do).

→ resource → endpoint



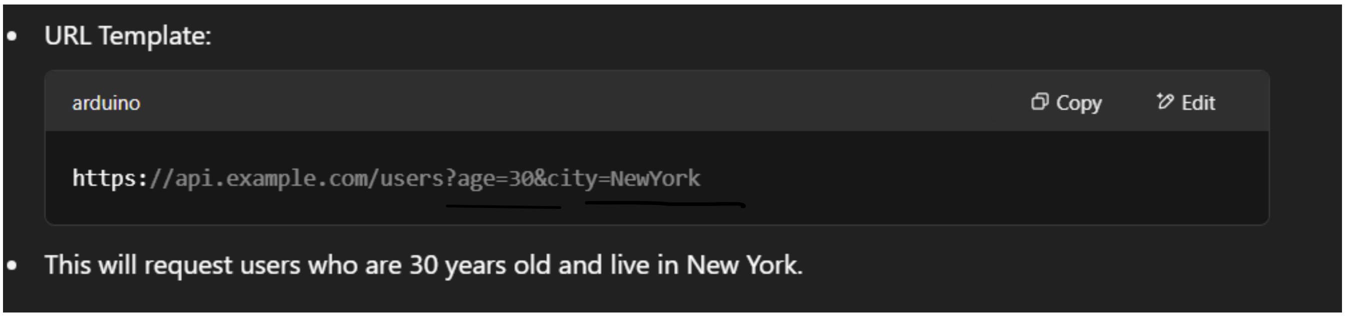
📌 **Structure of Query Parameters:**

The format for query parameters is:

arduino

```arduino
https://api.example.com/resource?key1=value1&key2=value2
```

- `?` separates the base URL from the query parameters.
- `key1=value1` is a key-value pair.
- `&` separates multiple key-value pairs.

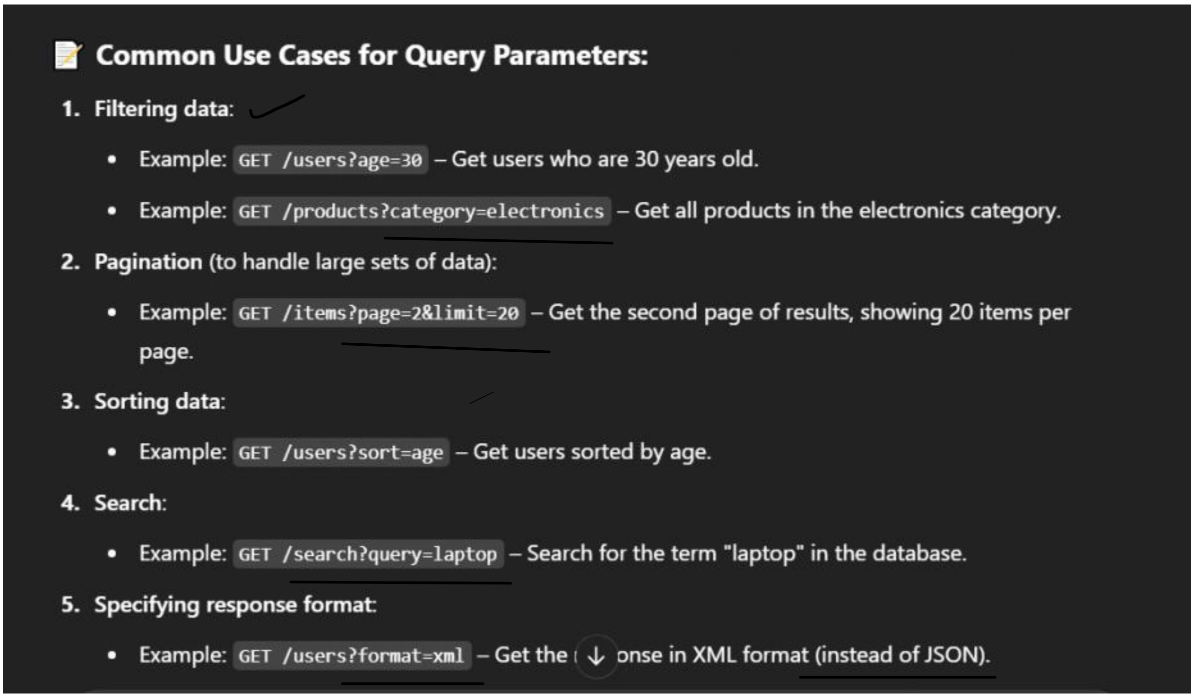


- URL Template:

arduino

```arduino
https://api.example.com/users?age=30&city=NewYork
```

- This will request users who are 30 years old and live in New York.



📝 Common Use Cases for Query Parameters:

1. **Filtering data**:
   - Example: `GET /users?age=30` – Get users who are 30 years old.
   - Example: `GET /products?category=electronics` – Get all products in the electronics category.
2. **Pagination** (to handle large sets of data):
   - Example: `GET /items?page=2&limit=20` – Get the second page of results, showing 20 items per page.
3. **Sorting data**:
   - Example: `GET /users?sort=age` – Get users sorted by age.
4. **Search**:
   - Example: `GET /search?query=laptop` – Search for the term "laptop" in the database.
5. **Specifying response format**:
   - Example: `GET /users?format=xml` – Get the ↓ onse in XML format (instead of JSON).

Response Codes

Status codes

📃 HTTP Response Codes

HTTP response codes indicate the **status** of a request made to the server. These codes are grouped into five categories (ranges), each representing a different class of response.

1. 1xx – Informational Responses

- **Description**: These codes indicate that the server has received the request and is continuing the process.
- **Range**: 100–199
- **Common codes**:
  - 100 Continue – The server has received the request, and the client should continue with the request.
  - 101 Switching Protocols – The server is switching protocols as requested by the client.

2. 2xx – Successful Responses

- **Description**: These codes indicate that the request was **successful** and the server has successfully processed it.
- **Range**: 200–299
- **Common codes**:
  - 200 OK – The request was successful, and the server has returned the requested data. ✓ ✓
  - 201 Created – The request was successful, and a new resource has been created (typically in response to a POST request).
  - 204 No Content – The request was successful, but there's no content to return (often used with DELETE or PUT).

3. 3xx – Redirection Responses

- **Description**: These codes indicate that the client **must take additional actions** to complete the request, like following a redirect.
- **Range**: 300–399
- **Common codes**:
  - 301 Moved Permanently – The resource has been permanently moved to a new location.
  - 302 Found – The resource is temporarily located elsewhere.
  - 304 Not Modified – The resource has not been modified since the last request (used in caching).

4. 4xx – Client Errors

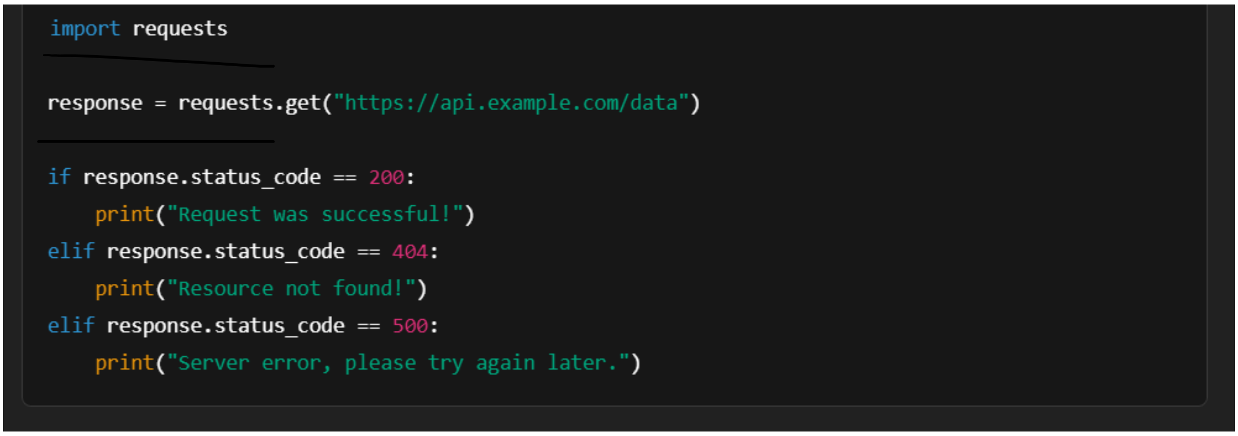
- **Description**: These codes indicate that the **client** has made an error in the request.
- **Range**: 400–499
- **Common codes**:
  - ✓ 400 Bad Request – The server cannot process the request due to invalid syntax.
  - ✓ 401 Unauthorized – The request requires authentication, or authentication failed (e.g., missing or invalid API key).
  - 403 Forbidden – The client does not have permission to access the requested resource.
  - ✓ 404 Not Found – The requested resource could not be found on the server.
  - 408 Request Timeout – The client's request timed out before the server could respond.

5. 5xx – Server Errors

- **Description**: These codes indicate that the server has encountered an **error** or is otherwise incapable of performing the request.
- **Range**: 500–599 → Server
- **Common codes**:
  - 500 Internal Server Error – A generic error when the server encounters an unexpected condition.
  - 502 Bad Gateway – The server received an invalid response from an upstream server.
  - 503 Service Unavailable – The server is temporarily unable to handle the request (often due to being overloaded).
  - 504 Gateway Timeout – The server did not receive a timely response from an upstream server.

📌 Summary of Key Ranges:

- **1xx**: Informational – Request received, continuing process.
- **2xx**: Success – The action was successfully received, understood, and accepted.
- **3xx**: Redirection – Further action is needed to fulfil the request.
- **4xx**: Client Error – The request contains bad syntax or cannot be fulfilled.
- **5xx**: Server Error – The server failed to fulfil a valid request.



```python
import requests

response = requests.get("https://api.example.com/data")

if response.status_code == 200:
    print("Request was successful!")
elif response.status_code == 404:
    print("Resource not found!")
elif response.status_code == 500:
    print("Server error, please try again later.")
```