Fast API

FastAPI is a modern, high-performance web framework in Python that's widely used for building APIs.

Before diving deeper into FastAPI, let's first understand what an API actually is.

API Stands for Application Programming Interface. We can say that an **API (Application Programming Interface)** is a **set of rules, definitions, and protocols** that allows two software applications to communicate with each other.

Think of an API as a **messenger** between two systems — We have three main steps

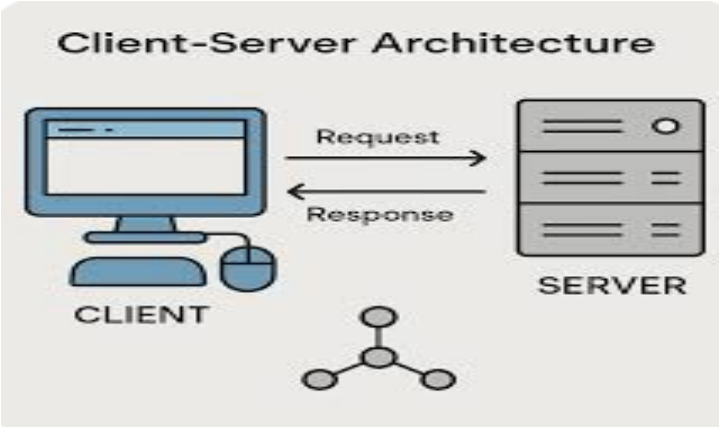
1. it takes a request,
2. tells the system what you want to do, and
3. then returns the response back to you.

To understand an API's workflow, we have to understand the Client-Server architecture.

What is Client–Server Architecture?

It's a model where:

- The **client** (browser, mobile app, backend service) makes requests.
- The **server** (API service) processes those requests and returns responses.
  They communicate over a network (usually HTTPS), using a well-defined contract (the API).



Clients handle user interactions and send specific requests to the server. The server processes these requests and returns the appropriate responses.

The client–server architecture is highly scalable, as it can support an increasing number of clients by enhancing the server's capacity or by adding additional servers to distribute the workload.

In real time…

When a client sends a request, it is not sent directly to the server. Instead, the request first goes to the Domain Name System (DNS). The **DNS** acts like the internet's phonebook — it takes the **domain name** (for example, *www.example.com*) and translates it into the corresponding **IP address** of the server that hosts the requested resource.

Once the IP address is resolved:

1. The client's browser uses this IP address to establish a connection with the server using **TCP (Transmission Control Protocol)** or **UDP (User Datagram Protocol)**, depending on the type of request.
2. After the connection is established, the client sends the **HTTP or HTTPS request** to the server.
3. The **server processes** the request, retrieves the necessary data (from databases or other services), and sends back an **HTTP response** containing the requested information or resource.
4. The **client (browser)** then interprets the response — for example, rendering a web page or displaying an error message if something went wrong.

Step-by-Step Client–Server Communication Process

Step 1: Client Initiates a Request

- A user enters a website URL (like `www.example.com`) in the browser or clicks a link.
- The browser prepares to send an **HTTP/HTTPS request**, but first, it needs the **IP address** of the server where the website is hosted.

Step 2: Request Goes to DNS

- The browser checks its **local DNS cache** to see if it already knows the IP address of `www.example.com`.
- If not found, it contacts the **DNS resolver** (usually provided by your Internet Service Provider or a public DNS like Google's `8.8.8.8`).

Step 3: DNS Resolution Process
The DNS resolver performs several lookups to find the IP address:

1. **Root DNS Server** – It points the resolver to the correct **Top-Level Domain (TLD)** server (e.g., `.com`).
2. **TLD DNS Server** – It directs the resolver to the **Authoritative Name Server** for `example.com`.
3. **Authoritative Name Server** – This server provides the actual **IP address** associated with `www.example.com`. The IP address is then returned to the client's browser.

Step 4: Browser Connects to the Web Server

- Using the IP address obtained from DNS, the browser establishes a **TCP connection** with the server (3-way handshake).
- If it's an HTTPS site, a **TLS/SSL handshake** also occurs to establish a secure encrypted channel.

Step 5: Client Sends the HTTP/HTTPS Request

- The browser sends a structured request (e.g., `GET /index.html`) to the server along with headers that specify details such as browser type, accepted formats, and cookies.

Step 6: Server Processes the Request

- The web server receives the request, processes it, and may communicate with **databases or other APIs** to fetch data.
- The server then generates a response (such as an HTML page, JSON data, or an error message).

Step 7: Server Sends the Response

- The server sends back an **HTTP response** to the client, which includes the requested content and a **status code** (like `200 OK`, `404 Not Found`, or `500 Internal Server Error`).

Step 8: Browser Renders the Content

- The browser interprets the response, downloads any additional resources (like CSS, JavaScript, or images), and **renders the web page** for the user.

Step 9: Connection Closed or Reused

- Once the response is fully received, the TCP connection may be **closed** or **kept alive** for future requests to the same server.

How APIs and client-server communication work.

When a client (like your browser, mobile app, or a frontend system) communicates with a server, it sends HTTP requests. *Each request asks the server to perform a specific action,* and in return, the server sends back a response.

We can do various actions such as

→ Fetch data from the server
→ Add new data
→ Update entire data
→ Partially update data
→ Remove data
→ Fetch headers only (fetching meta data)

Each action is associated with the HTTP Method.

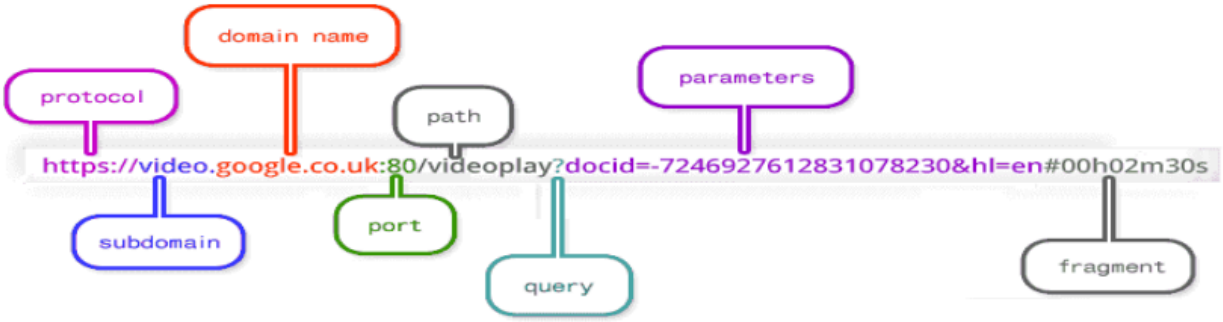
HTTP Method	Purpose / Action Client Requests	What the Server Does in Return	Example
GET	"Fetch data from the server"	Retrieves data from the database and sends it back to the client	`GET /users` → Returns all users
POST	"Add new data"	Creates a new record in the database and confirms creation	`POST /users` → Adds a new user
PUT	"Update entire data"	Replaces an existing record with new data	`PUT /users/5` → Updates user with ID 5
PATCH	"Partially update data"	Updates only specific fields of an existing record	`PATCH /users/5` → Updates user's email only
DELETE	"Remove data"	Deletes a record from the database	`DELETE /users/5` → Deletes user with ID 5
OPTIONS	"Tell me what I can do"	Informs the client which HTTP methods are allowed for a resource	`OPTIONS /users` → Returns allowed methods
HEAD	"Fetch headers only"	Returns only metadata (headers) without the body	`HEAD /users` → Checks if data exists

Understanding URL and its components in detail:

A **URL (Uniform Resource Locator)** is the address you type in your browser to visit a website or access a resource online. Example : ***https://mail.google.com/inbox?folder=unread#top***

Now let us understand the various components of the URL in details

here's an example of a more complex-looking URL:



Protocol

- Tells the browser **how to communicate** with the server.
- Common examples:
  - `http` → Hypertext Transfer Protocol (normal websites)
  - `https` → Secure version (encrypted, used for banking or login pages)
- Always prefer `https` for security.

Subdomain

- A smaller section of the main website.
- Example:
  - `mail.google.com` → "mail" is the subdomain.
  - `calendar.google.com` → "calendar" is another subdomain.
  - online.innomatics.com → "online" is the subdomain

Domain Name

- The main name of the website.
- Example:In `google.com`,

  `google` = domain name
  `.com` = Top-Level Domain (TLD)
  In `co.uk`, `.uk` is TLD and `.co` is part of the second-level domain.

Port

- A “doorway” used by the browser to connect to the server.
- Usually hidden, but defaults are:
    - Port `80` → for `http`
    - Port `443` → for `https`

Example: https://example.com:443

Path

- Shows the **exact location** of a page or file on the server.
- Example: https://example.com/blog/article.html
- `/blog/article.html` is the path.

Note : If no file is specified, the browser looks for `index.html` or `default.html`.

Query

- Comes **after a question mark (?)**.
- Used to send **extra information or filters** to the server.

Example: `https://example.com/search?q=python`

Here, `q=python` means you are searching for the word *python*.

Parameters

- Inside the query, there can be **multiple key-value pairs**, separated by `&`.

Example: `?q=python&lang=en&sort=asc`

`q=python` → search term
`lang=en` → language filter
`sort=asc` → sort order

Fragment

- Comes after a **# (hash)** symbol.
- Points to a specific section within a page (like a bookmark).

Example:
`https://example.com/about#team`
This jumps directly to the “team” section on the page.

HTTP Response Code Categories with Examples:

When a client sends a request to the server, the server can **accept**, **process**, **reject**, or sometimes **fail to respond** due to various reasons. In each of these cases, the server sends back a **predefined HTTP response code** to indicate the outcome of the request.

These **HTTP status codes** help both clients and developers understand what happened during the communication.

They are grouped into five main categories based on the type of response:

🔢 Categories of HTTP Status Codes

Category	Code Range	Meaning
1xx – Informational	100–199	Request received and understood; the process is continuing.
2xx – Success	200–299	The request was successfully received, understood, and processed.
3xx – Redirection	300–399	The client must take additional action, like following a new URL.
4xx – Client Error	400–499	The request was invalid or cannot be processed due to client-side issues.
5xx – Server Error	500–599	The server failed to fulfill a valid request due to an internal error.

✅ Examples of Common HTTP Response Codes

Code	Meaning	Description
200 OK	Success	The request was successful and the server returned the expected response.
201 Created	Resource Created	A new resource has been successfully created on the server.
204 No Content	Success, No Data	The action was successful, but there's no content to return.
301 Moved Permanently	Redirect	The resource has been moved to a new URL permanently.
302 Found	Temporary Redirect	The resource temporarily resides under a different URL.
400 Bad Request	Client Error	The server couldn't understand the request due to invalid syntax.
401 Unauthorized	Authentication Needed	The client must authenticate before making this request.
403 Forbidden	Access Denied	The client doesn't have permission to access the resource.
404 Not Found	Resource Missing	The server couldn't find the requested URL or resource.
500 Internal Server Error	Server Fault	The server encountered an unexpected condition.
503 Service Unavailable	Server Busy	The server is temporarily unable to handle the request (e.g., maintenance).

Hands On:

```
1  import requests # type: ignore
2
3  response = requests.get('https://api.github.com')
4  print(response.status_code)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
[Running] python -u "C:\Users\venum\AppData\Local\Temp\tempCodeRunnerFile.python"
200
```

Client-Side Components:

These are the parts that run on the **user's device** — usually in the browser.

Component	Description	Example Technologies
1. User Interface (UI)	What the user interacts with — forms, buttons, images, links.	HTML, CSS
2. Client-Side Logic	Executes inside the browser — handles interactions, validations, dynamic updates.	JavaScript, TypeScript
3. Front-End Frameworks	Libraries and frameworks to build rich, interactive UIs.	React, Angular, Vue.js
4. HTTP Client / Fetch Logic	Sends API requests to the server and processes responses.	`fetch()`, `axios`, AJAX
5. Caching / Storage	Temporarily stores data on the user's browser for faster access.	LocalStorage, SessionStorage, Cookies
6. Rendering Engine	Converts HTML, CSS, and JS into a visual web page.	Chrome's Blink, Firefox's Gecko
7. Security Layer (Client-Side)	Handles encryption, cookies, and token storage.	HTTPS, JWT tokens in local storage

Server-Side Components:

These run on the **server or backend** — where business logic and data processing happen.

Component	Description	Example Technologies
1. Web Server	Accepts client requests and routes them to appropriate handlers.	Apache, Nginx, Microsoft IIS, Uvicorn (FastAPI)
2. Application Server / Backend	Contains the logic that processes requests and generates responses.	FastAPI, Flask, Django, ASP.NET Core, Node.js
3. Database Server	Stores, retrieves, and manages data.	MySQL, SQL Server, PostgreSQL, MongoDB
4. Business Logic Layer	Implements core application rules and workflows.	Custom Python / C# / Java code
5. API Layer	Defines endpoints for communication with clients (CRUD operations).	REST APIs, GraphQL APIs
6. Authentication & Authorization	Manages user identity and access control.	JWT, OAuth 2.0, Identity Server
7. File Storage / Media Server	Stores uploaded files, images, or media content.	AWS S3, Azure Blob Storage
8. Caching Layer (Server-Side)	Improves performance by storing frequently accessed data.	Redis, Memcached
9. Logging & Monitoring	Tracks server activity and errors.	ELK Stack, Prometheus, Serilog
10. Security Layer (Server-Side)	Protects from attacks like SQL Injection, CSRF, XSS.	HTTPS, Firewalls, Input Validation