CLIENT-SERVER ARCHITECTURE

WHAT IS CLIENT-SERVER ARCHITECTURE?

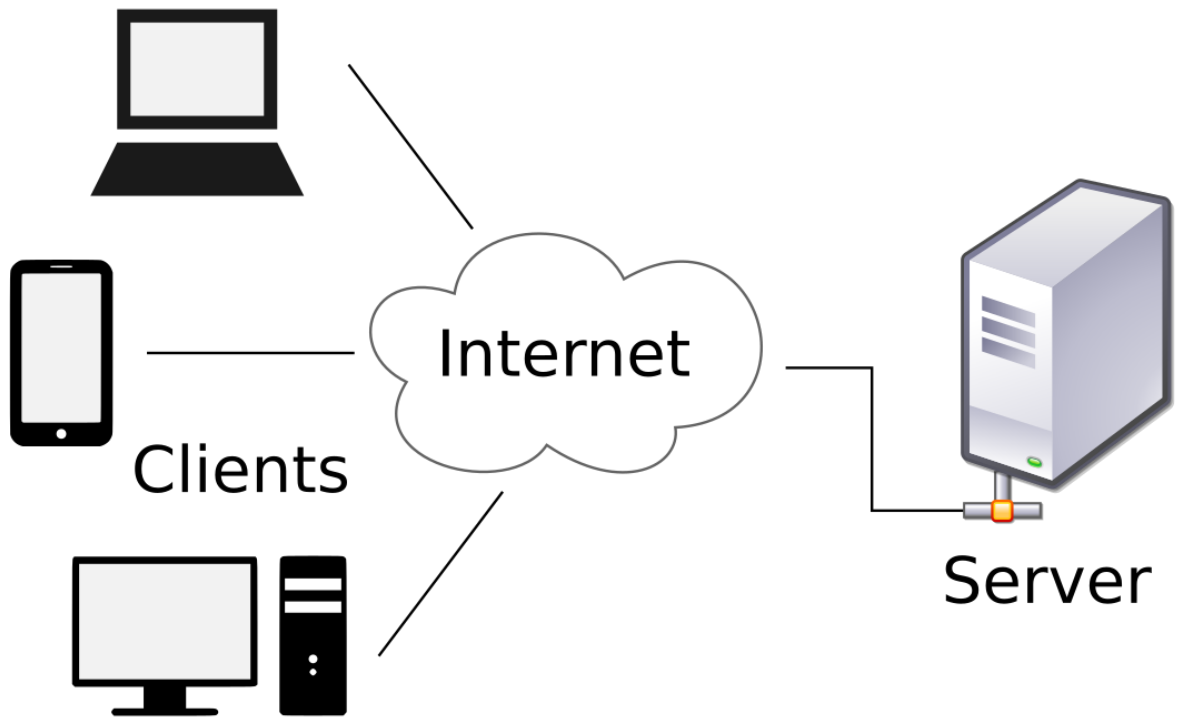
- **Client-Server Architecture** is a system structure where the **client (user's device)** requests a service, and the **server (central computer)** provides the service.

- Client : The device or application that makes a request (e.g., browser, mobile app)
- Server : The system that listens to requests, processes them, and sends back data (e.g., a website's backend)

REAL WORLD ANALOGY

- Think of a **restaurant**:
- You (client) order food.
- The **kitchen (server)** prepares it and sends it back.
- The **waiter (internet)** carries the request and response.

- Client → sends request → Server
- Server → processes → sends response → Client


Internet
Clients
Server

IP ADDRESSES

- In a **Client-Server system**, both the **client** and the **server** need to **identify each other** to send and receive data.
- IP stands for **Internet Protocol** address.
- It is a **unique numerical label** assigned to every device connected to a network.
- Used to **identify and locate devices** on the internet or local networks.

DOMAIN NAMES

- A **domain name** is a **human-friendly address** used to access websites on the internet instead of typing the numeric IP address.
- IP addresses (like `192.168.1.1`) are hard to remember.
- Domain names (like `google.com`) are easier for humans to remember and use.
- They act as a **nickname** or **alias** for the IP address of a server.

REAL WORLD ANALOGY:

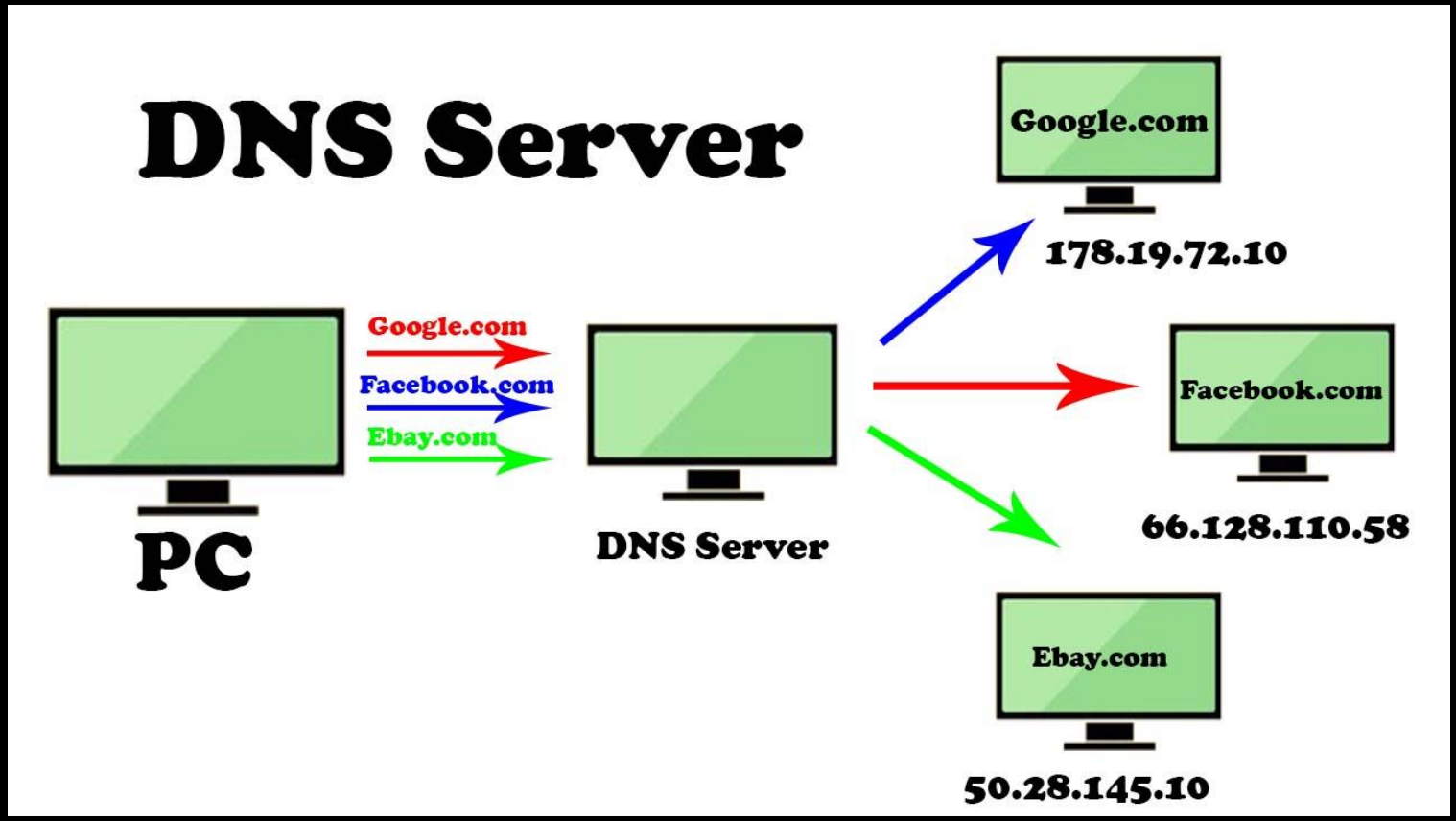
- Imagine calling a friend:
- You remember their **name** (domain name), not their phone number (IP address).
- You look up their phone number in your contacts (DNS).
- Then call them using the phone number.

DOMAIN NAME SYSTEM

- The **Domain Name System (DNS)** is like the **internet's phonebook** that translates **domain names** (like `www.google.com`) into **IP addresses** (like `172.217.6.68`) so browsers can load websites.
- Humans remember domain names easily, but computers use IP addresses.
- DNS **automates the translation** between domain names and IP addresses.
- Without DNS, you would have to memorize complex IP addresses to visit websites.

CLIENT REQUEST

- How DNS Works (Step-by-Step):
- You type `www.example.com` in the browser.
- The browser asks the **DNS resolver** (usually your ISP) for the IP address.
- If the resolver doesn't know, it asks higher-level DNS servers.
- Eventually, the **authoritative DNS server** for the domain replies with the correct IP address.
- The browser uses this IP to connect to the server and load the website.


DNS Server
Google.com
178.19.72.10
Google.com
Facebook.com
Ebay.com
Facebook.com
66.128.110.58
PC
DNS Server
Ebay.com
50.28.145.10

PROXY SERVER

- A **proxy server** sits **between the client and the internet**.
- The **client** sends the request to the proxy.
- The proxy **forwards** that request to the actual server on behalf of the client.
- Why Use It?

  To hide the client's IP address.

  To filter requests (like parental controls or office restrictions).

  To cache responses for faster loading.

  To monitor or control access to internet usage

HTTP&HTTPS

- It's a **protocol** (set of rules) used by web browsers to **communicate with web servers**.
- It helps **transfer web pages, images, videos, etc.** over the internet.
- When you visit a website like `http://example.com`, you're using HTTP.
- HTTPS = **HTTP + Security** (uses **SSL/TLS encryption**).
- It makes sure the data you send/receive is **encrypted** — like a secret message.
- You use it when visiting `https://example.com`.

- Real-World Example (Simple Language):
- Imagine you go to a restaurant:
- You (the **client/browser**) ask: “Can I have the menu?”
- The waiter (the **HTTP protocol**) brings your request to the **kitchen/server**, and then brings the **menu (webpage)** back to you.
- This "asking and delivering" is what HTTP does between your browser and a website.

API

An **API** allows **two applications to talk to each other** — it's like a waiter that takes your order and brings your food back.

Imagine:

- You (client) go to a travel agent (API)
- You ask: "Find me flights to Goa"
- The agent checks airline databases (server), then responds
- You don't care how they did it — you just get your flight info

That's **what an API does**: hides the backend complexity, gives you what you asked for.

REST API

- REST is an architecture that defines **standard ways to access data using HTTP**.
- GET /users/123
- → gets user with ID 123

- It uses **HTTP methods**: GET, POST, PUT, DELETE.

GRAPHQL

- GraphQL is a **query language** for APIs developed by Facebook. It lets clients **ask exactly for the data they want**, no more, no less.
- {
- user(id: 123) {
- name
- email
- orders {
- product
- price
- }
- }
- }

Concept	REST	GraphQL
Protocol	Follows HTTP standards	Uses HTTP, but custom query language
Calls	Many endpoints	One smart endpoint
Flexibility	Low (fixed data shapes)	High (client chooses fields)
Simplicity	Easier to learn, widespread	More complex setup, modern apps prefer it

DATABASE

- A **database** is an organized collection of data stored electronically, which the **server uses to store, retrieve, and manage information**.
- How It Works:
- **Client** sends a request (e.g., "Show my order history").
- **Server** processes this request.
- **Server queries the database** to fetch the order details.
- **Database** returns the data.
- **Server** sends the data back to the client.

CONTENT DELIVERY NETWORK

- A **CDN** is a network of geographically distributed servers that **cache and deliver web content (like images, videos, scripts, stylesheets) to users faster** by serving it from the nearest server location.
- How Does CDN Work?
- User requests content (e.g., image on a webpage).
- Request goes to the **nearest CDN server** (based on user's location).
- CDN server serves the cached content.
- If the content is not cached, CDN fetches it from the **origin server** and caches it for future requests.
- When you open **YouTube**, videos and thumbnails are served from CDN servers closest to you, so videos start quickly.

MIDDLEWARE

- **Middleware** is software that acts as a **bridge** between the client's request and the server's response, processing requests **before** they reach the main server logic and sometimes processing responses **before** they go back to the client.
- Real-World Analogy:
- Middleware is like a **restaurant host** who checks the reservation, directs you to the right table, and passes your order to the kitchen. Before food leaves the kitchen, the host might check it's correct and ready for the customer.

WEBSOCKETS

- **WebSocket** is a communication protocol that enables **full-duplex, real-time, two-way communication** between a client (like a browser) and a server over a single, long-lived connection.
- Unlike HTTP requests that are one-way and short-lived (client sends a request → server sends response → connection closes), WebSockets keep the connection **open**.
- This allows **both client and server to send messages to each other anytime**, instantly.
- **Chat applications:** Messages appear instantly.
- **Online gaming:** Fast real-time updates.

- How It Works (Simplified):
- Client sends a **WebSocket handshake request** to server over HTTP.
- Server upgrades the connection to a WebSocket protocol.
- Both keep the connection open.
- Either client or server can send data anytime without new requests.
- Connection closes only when either side ends it.

LOAD BALANCER

- A **Load Balancer** is a system (hardware or software) that **distributes incoming network traffic evenly across multiple servers** to ensure no single server gets overwhelmed.
- How It Works:
- Client sends a request.
- The **load balancer** receives the request first.
- It decides which backend server should handle it based on criteria (like least busy, round-robin, server health).
- The selected server processes the request and sends the response back via the load balancer.
- Client receives the response seamlessly.

CONCLUSION

- Client-Server Architecture is the backbone of modern web and network applications. It organizes computing into two main roles — clients that request services, and servers that provide them — enabling efficient communication, scalability, and resource sharing.
- Key components like **IP addresses**, **DNS**, **HTTP/HTTPS protocols**, **APIs**, **load balancers**, **caching**, and **API gateways** work together to make this interaction fast, secure, and reliable.
- Understanding these elements helps developers design systems that can handle millions of users, provide real-time data, and maintain high availability. As technology evolves, mastering client-server principles remains essential for building robust and scalable applications.