BackEnd
Understanding Servers and Request Handling

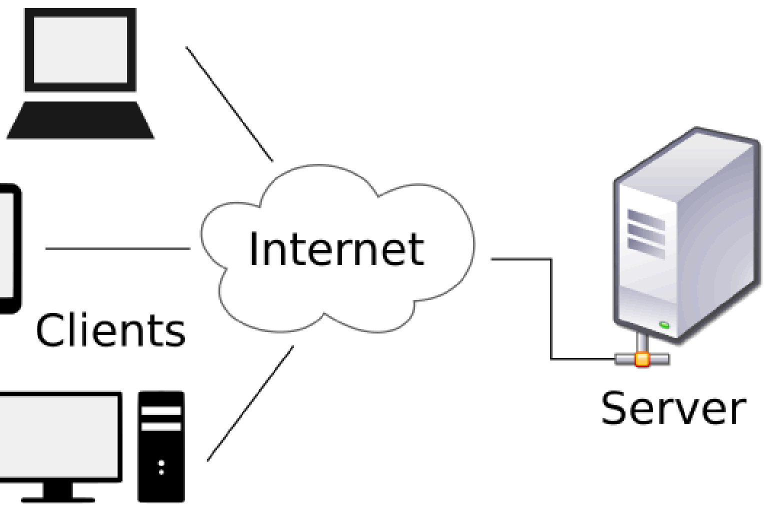
Introduction

A server is a powerful machine designed to compute, store and manage data, devices and systems over a network. This Sophisticated computer system provides resources to networking units to render specialized services such as displaying web pages and sending or receiving mails among other functionalities.

Computer hardware, software or even virtual machines with requisite software capabilities can act as a server.

Servers answer user requests via a client-server model. Here the ghost device that powers network devices is referred to as host server, and the in-network devices that utilize the resources the host device offers are termed clients.

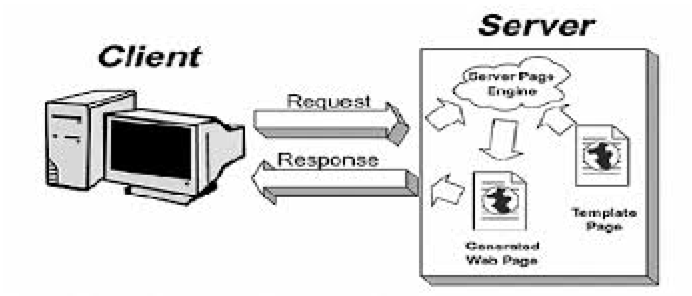
A good Server requires a processor, RAM, storage and Bandwidth, which is used to deliver content via the internet or a combination of networks such as WAN or LAN



Communication Protocols

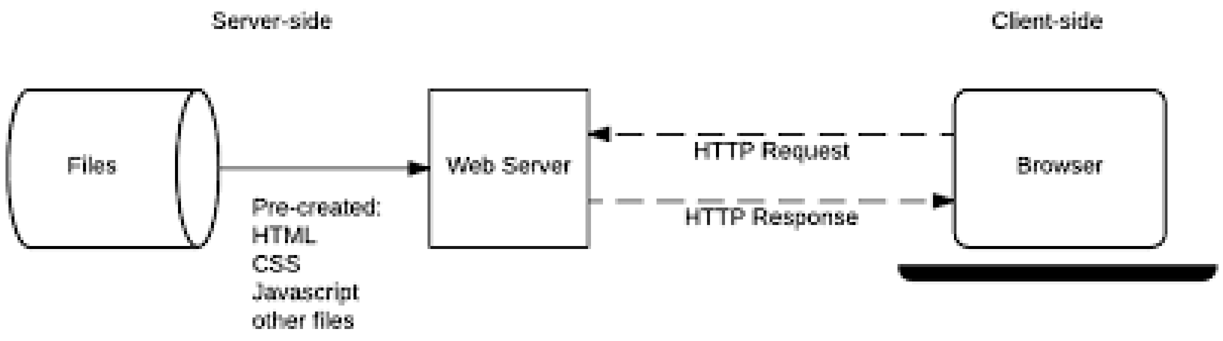
Clients and Servers communicate using protocols, which are sets of rules that define how data is exchanged. Common protocols include:

- **Http/Https:** Used for Web Services.
- **FTP:** Used for file transfers.
- **SMTP/IMAP/POP3**: Used for Email Services.
- **TCP/IP:** Underlies most Internet communications.



Request Handling Process

a. **Client Sends a Request**
   1. **Initiation:** The Client initiates a connection to the server using the servers IP address and a specific port number associated with the Service(ex., port 80 for HTTP)
   2. **Request Formation:** The client forms a request, typically in a structured format such as an HTTP request for web services, which includes:
      - **Method:** Action to be perfomed(ex., GET, POST, PUT, DELETE in HTTP)
      - **URL/URI:** The resources being Requested.
      - **Headers:** Metadata about the request(ex., content type, user-agent).
      - **Body:** Optional data sent with the request(ex., form data in a POST request).



b. **Server Receives the Request**
   1. **Listening:** The server listens for incoming requests on specific ports. Server software(like Apache, Nginx, IIS, etc.)is responsible for managing these connections.
   2. **Accepting connections:** The server accepts the connection from the client and reads the request date.

c. **Processing the Request**
   1. **Routing:** The Server routes the request to the appropriate handler based on the request method, URL, and headers.
   2. **Authentication and Authorization:** If required, the server checks if the client is authorized to access the requested resources.
   3. **Logic Execution:** The server executes the necessary logic, which might involve:
      - Querying a Database
      - Processing Data
      - Communication with other services
   4. **Error Handling:** The Server handles any errors that occur during processing and prepares an appropriate response.

d. **Generating a Response**
   1. **Response Formation:** The server forms a response, typically including:

- **Status Code:** Indicates the result of the request(ex., 200OK, 404 Not Found, 500 Internal Server Error).
- **Headers:** Metadata about the response(ex., content type, caching policies).
- **Body:** The requested data or result(ex., HTML content, JSON data, files).

2. **Sending Responses:** The server sends the response back to the client over the network.

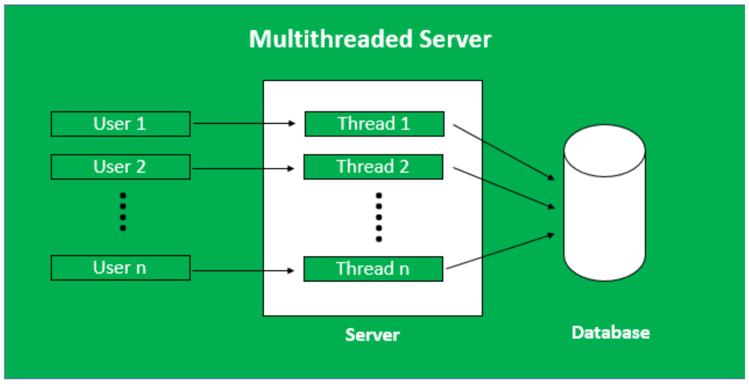
e. Client Receives the Response

1. **Receiving Data:** The client receives the server's response.
2. **Processing Responses:** The client processes the response, which might involve:
    - Rendering a webpage
    - Displaying Data
    - Storing Information
3. **Closing Connection:** The Client and server may close the connection or keep it open for additional requests(persistent connections).

Handling Concurrent Requests

a. **Concurrency:** Servers are designed to handle multiple requests simultaneously. They achieve this using :

- **Multi-Threading:** Creating multiple threads to handle each request separately.



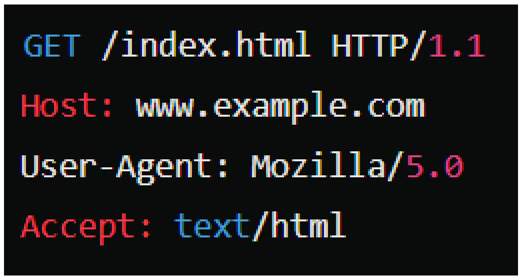
- **Event-Driven Architecture:** Using a single-threaded event loop to manage non-blocking operations efficiently(e.x., Node.js).
- **Load Balancing:** Distributing requests across multiple servers to balance the load and improve performance.

Security Measures

- **Encryption:** HTTPS encrypts data between the client and server to ensure privacy and security.
- **Firewalls:** Protect Servers from unauthorized access and attacks.
- **Authentication Mechanisms:** Use of tokens, passwords, and certificates to verify client identities.

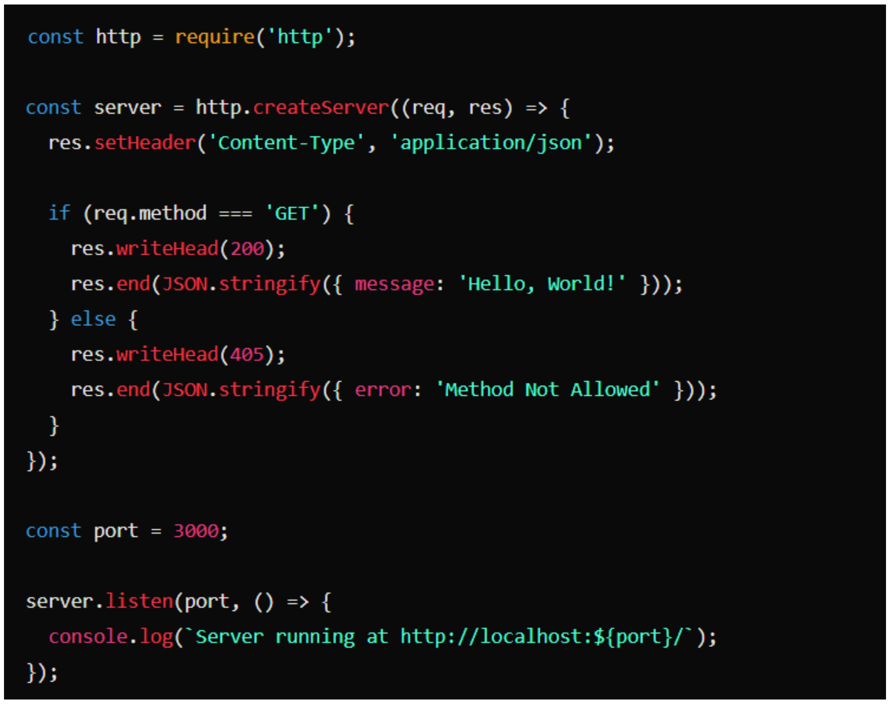
Example: HTTP Request Handling

Client Request:



```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
```

Server Response:



```
const http = require('http');

const server = http.createServer((req, res) => {
  res.setHeader('Content-Type', 'application/json');

  if (req.method === 'GET') {
    res.writeHead(200);
    res.end(JSON.stringify({ message: 'Hello, World!' }));
  } else {
    res.writeHead(405);
    res.end(JSON.stringify({ error: 'Method Not Allowed' }));
  }
});

const port = 3000;

server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```