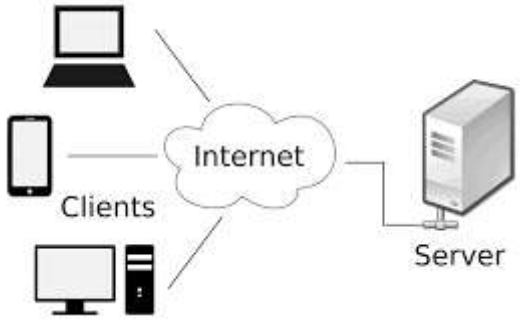
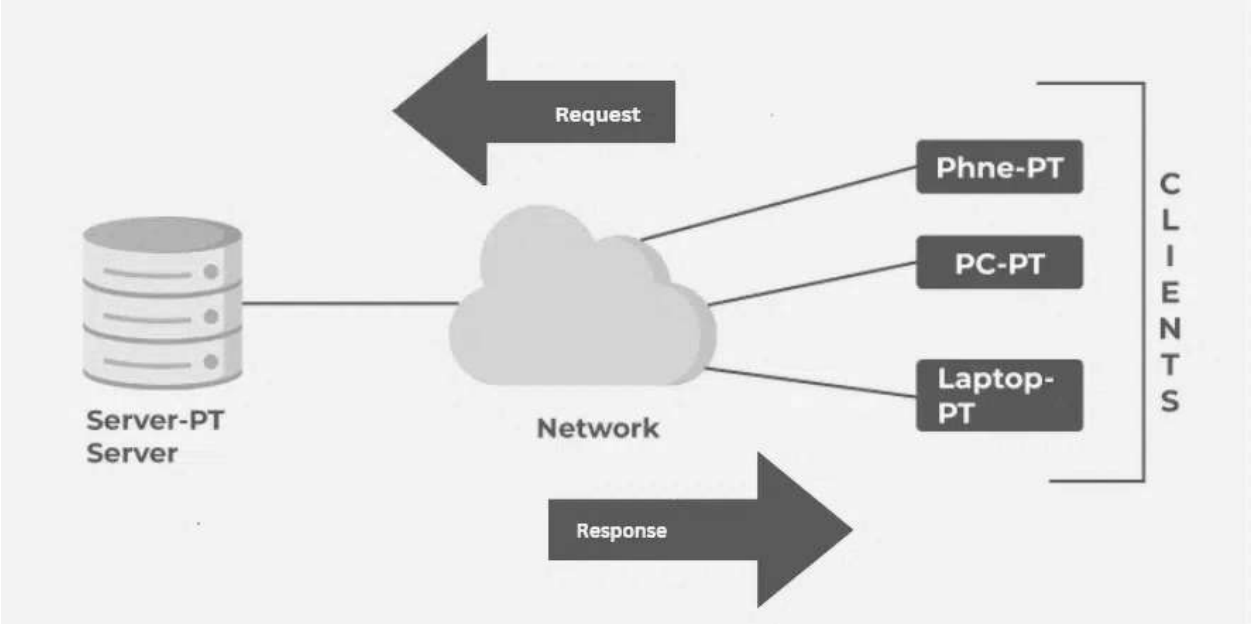
Client-Server Architecture

1. Introduction to Client-Server Architecture



- **Definition**: A computing model where tasks and services are distributed between two entities: clients (requesters) and servers (providers).
- **Purpose**: To efficiently share resources and process requests over a network.
- **Key Characteristics**:
    - Centralized resources on servers.
    - Communication through standardized protocols.
    - Scalability and modularity in application design.

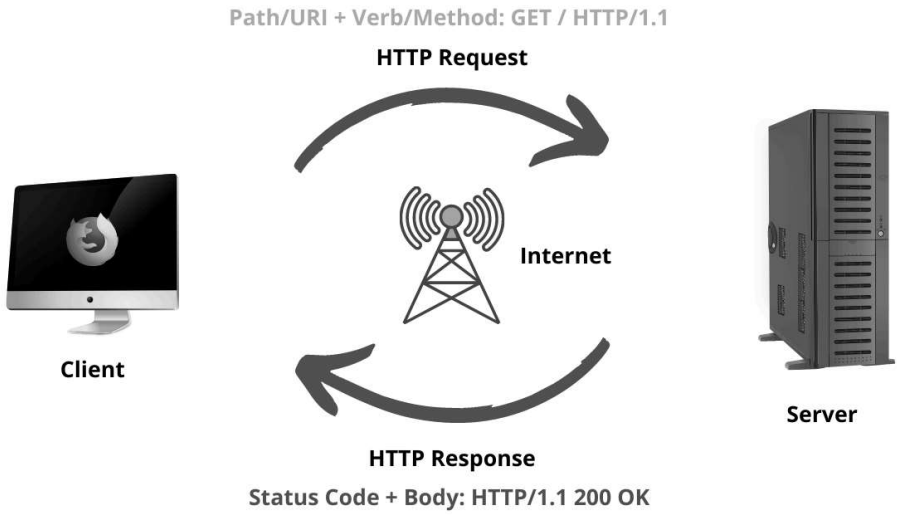
2. Components of Client-Server Architecture



- **Client**:
    - **Role**: Initiates requests to the server for services or resources.

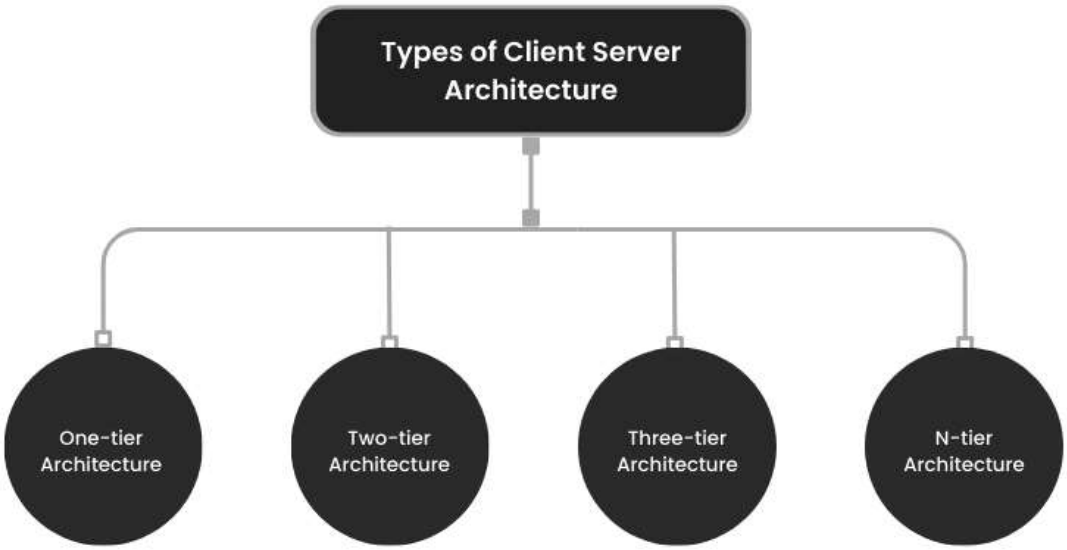
    - **Examples**:
        - Web browsers (Google Chrome, Firefox).
        - Mobile applications.
    - **Responsibilities**:
        - User interface (UI) rendering.
        - Sending user input to servers.
        - Displaying responses from servers.
- **Server**:
    - **Role**: Responds to client requests by providing the requested data or service.
    - **Examples**:
        - Web servers (Apache, Nginx).
        - Application servers (Tomcat, Node.js).
        - Database servers (MySQL, MongoDB).
    - **Responsibilities**:
        - Data processing and storage.
        - Managing client requests.
        - Ensuring security and authentication.

3. Communication in Client-Server Architecture

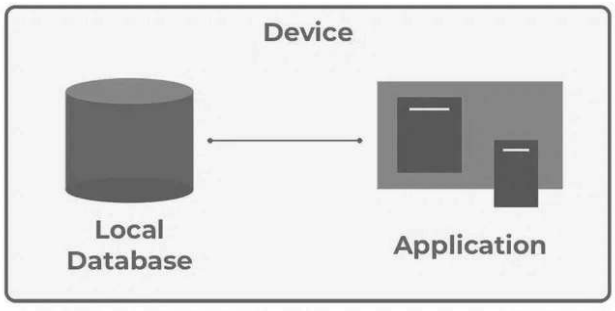


- **Request-Response Cycle**:
    - Client sends a request to the server.
    - Server processes the request.
    - Server sends a response back to the client.
- **Protocols**:
    - **HTTP/HTTPS**: For web communication.
    - **FTP**: For file transfers.
    - **SMTP/IMAP/POP3**: For email services.
    - **WebSockets**: For real-time, bidirectional communication.
- **Formats**:
    - **Text Formats**: Plain text, XML.
    - **Lightweight Formats**: JSON for modern web applications.

4. Types of Client-Server Architectures

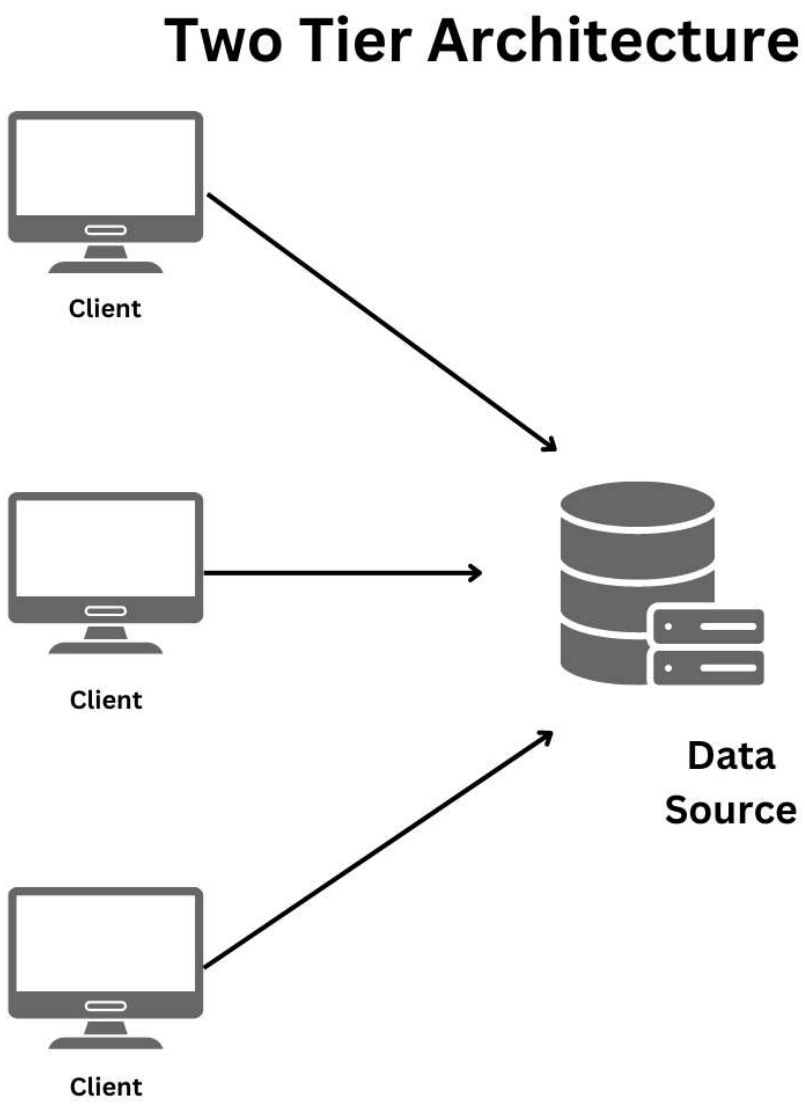


- **1-Tier Architecture**:



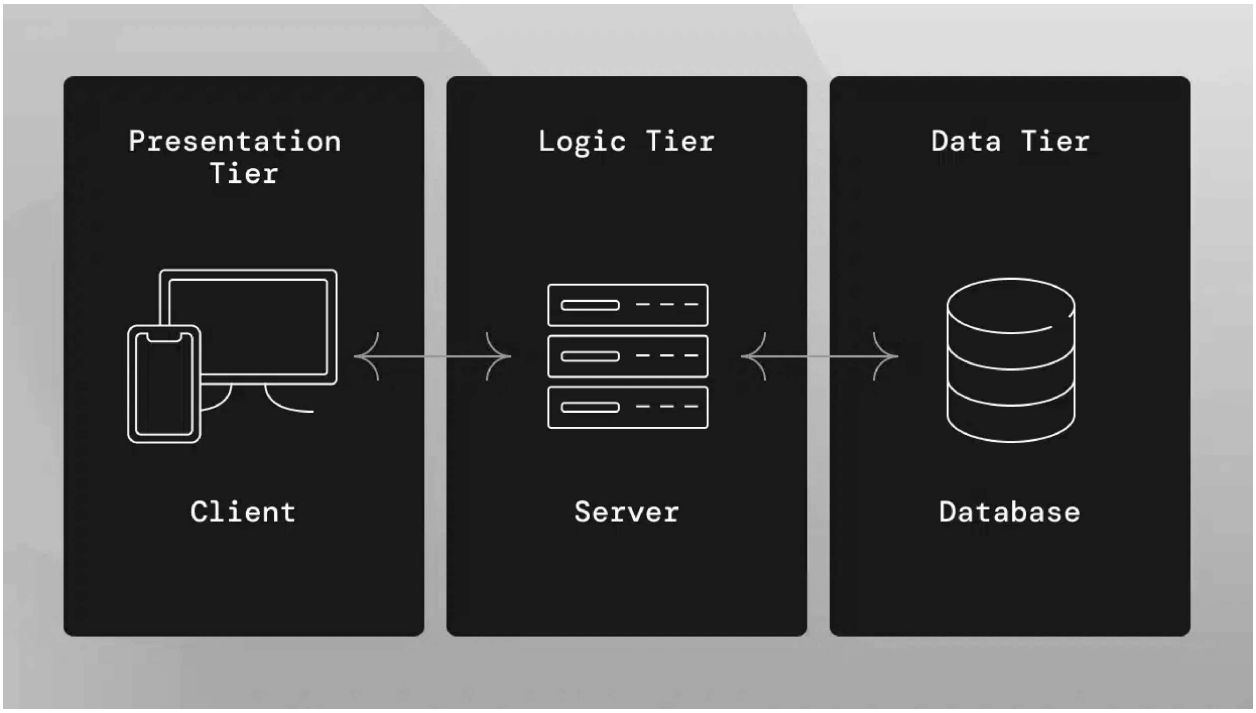
- All functionalities (UI, logic, and data) are in a single layer.
- Example: Standalone applications.
- **Limitations**: Poor scalability, limited sharing.

- **2-Tier Architecture**:

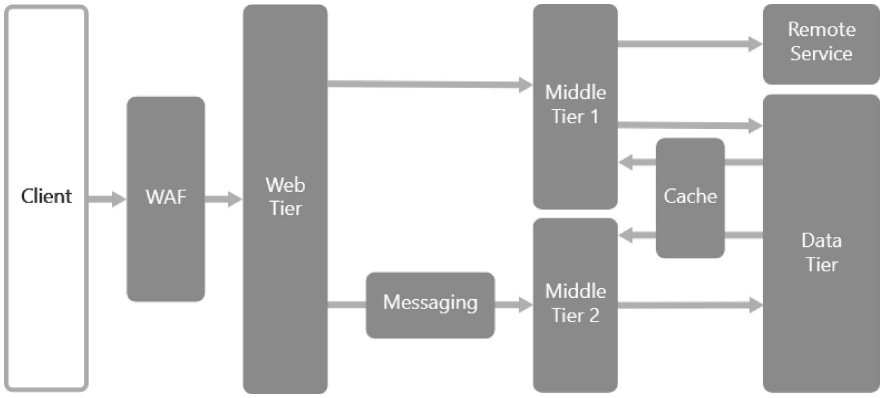


- **Structure**:
  - Client handles UI and logic.
  - Server handles data storage.
- Example: A client application accessing a database directly.

    - **Advantages**: Simplified communication.
    - **Disadvantages**: Limited scalability.
- **3-Tier Architecture**:



    - **Structure**:
        - **Client**: User interface.
        - **Application Server**: Business logic.
        - **Database Server**: Data storage.
    - Example: Most web applications (e.g., e-commerce sites).
    - **Advantages**: Improved scalability, better security.
    - **Disadvantages**: Increased complexity.
- **N-Tier Architecture**:



- Expands 3-tier to include additional layers like caching, analytics, etc.
- Example: Large-scale enterprise applications.
- **Advantages**: Flexibility, modularity.
- **Disadvantages**: High development and maintenance costs.

5. Advantages of Client-Server Architecture

- **Centralized Resources**: Easier to manage and update server-side components.
- **Scalability**: Servers can handle increasing client loads with proper infrastructure.
- **Security**: Sensitive data and processes remain on the server.
- **Collaboration**: Multiple clients can share resources simultaneously.
- **Maintainability**: Changes or updates only need to be applied on the server.

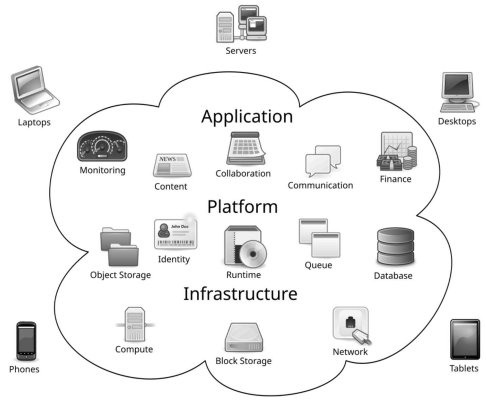
6. Challenges in Client-Server Architecture

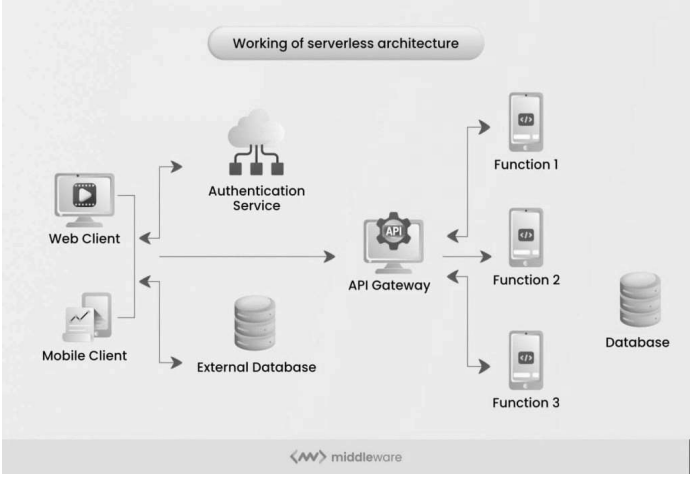
- **Single Point of Failure**: Server downtime affects all clients.
- **Network Dependency**: Requires a reliable and fast network connection.
- **Scalability Limits**: High traffic can overwhelm servers without proper load balancing.
- **Security Risks**: Vulnerable to attacks like DDoS, unauthorized access.

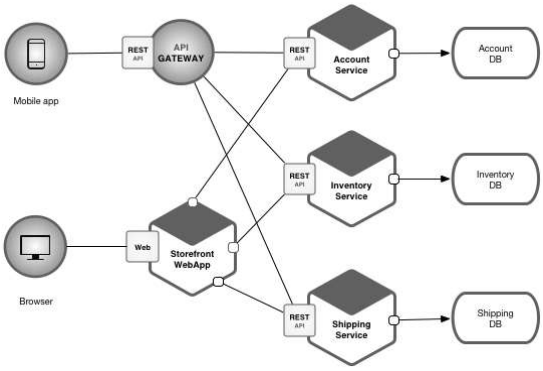
7. Real-World Applications of Client-Server Architecture

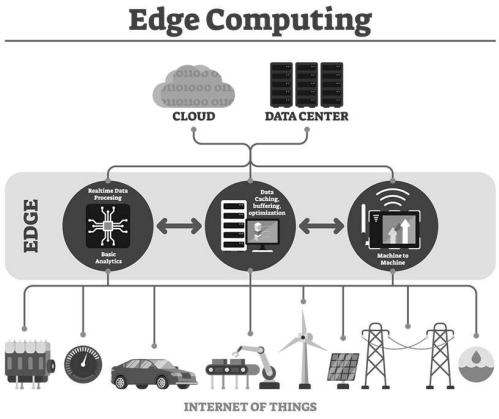
- **Web Browsing**: Example: Accessing websites via browsers (client) and web servers.
- **Email Services**: Example: Sending emails via SMTP servers.
- **Online Banking**: Secure transaction processing with backend servers.
- **Social Media Platforms**: Example: Instagram, Facebook.
- **E-Commerce**: Example: Shopping platforms like Amazon, Flipkart.

8. Emerging Trends in Client-Server Architecture









- **Cloud Computing**:
  - Servers hosted in distributed cloud data centers.
  - Example: AWS, Azure, Google Cloud.
- **Serverless Architectures**:
  - Functions executed in the cloud without managing servers.
  - Example: AWS Lambda, Google Cloud Functions.
- **Microservices**:
  - Applications divided into small, independent services.
  - Communicate using APIs.
- **Edge Computing**: Processing data closer to the client for reduced latency.

9. Comparison with Peer-to-Peer (P2P) Architecture

Feature	Client-Server Architecture	Peer-to-Peer Architecture
Structure	Centralized	Decentralized
Control	Managed by the server	Shared among peers
Examples	Websites, databases	File sharing (e.g., BitTorrent)
Scalability	Limited by server capacity	Highly scalable
Reliability	Dependent on server uptime	Redundant connections

Web Protocols (HTTP/HTTPS)

1. Introduction to Web Protocols

- **Definition**: Web protocols are standardized rules that define how data is exchanged between a client (e.g., browser) and a server over the internet.
- **Importance**:
    - Enable seamless communication across diverse devices and platforms.
    - Provide a framework for delivering content and services efficiently.

2. Overview of HTTP (Hypertext Transfer Protocol)

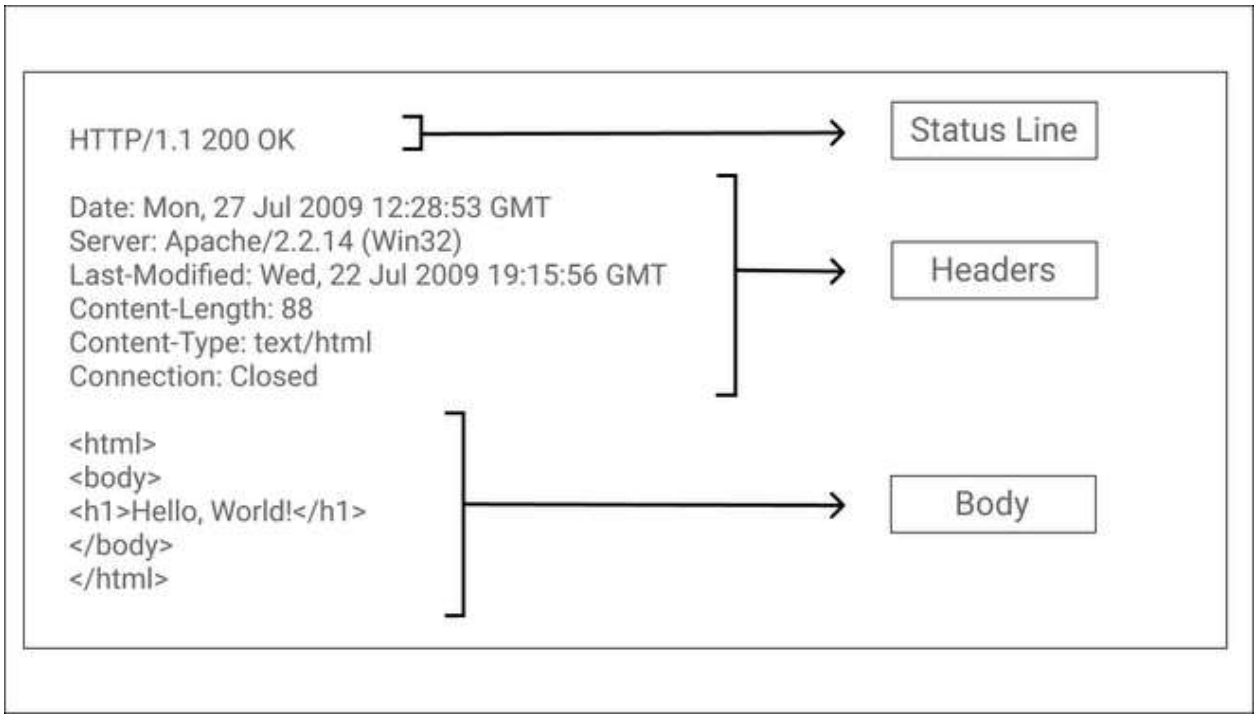
- **Definition**:
    - HTTP is a stateless, application-layer protocol used for transmitting hypertext documents like HTML.
    - Initially designed for communication between web browsers and servers.
- **Key Features**:
    - **Stateless**: Each request is independent; no information is retained between requests.
    - **Text-Based**: Requests and responses are plain text, making them easy to debug.
    - **Client-Server Model**: The client sends requests, and the server responds.

3. HTTP Request-Response Model

- **HTTP Request**:

- **Structure**:
  - Request Line: Method, URL, HTTP version.
  - Headers: Metadata about the request (e.g., `Content-Type`).
  - Optional Body: Data sent with certain methods (e.g., POST).
- **Common Methods**:
  - `GET`: Retrieve data from the server.
  - `POST`: Submit data to be processed by the server.
  - `PUT`: Update resources on the server.
  - `DELETE`: Remove resources.
  - `HEAD`: Retrieve headers without the body.

- **HTTP Response**:



- **Structure**:
  - Status Line: HTTP version, status code, status message.
  - Headers: Metadata about the response.
  - Optional Body: Content of the response.
- **Common Status Codes**:
  - 1xx: Informational (e.g., 100 Continue).
  - 2xx: Success (e.g., 200 OK, 201 Created).
  - 3xx: Redirection (e.g., 301 Moved Permanently, 302 Found).
  - 4xx: Client Errors (e.g., 404 Not Found, 403 Forbidden).

- 5xx: Server Errors (e.g., 500 Internal Server Error, 503 Service Unavailable).

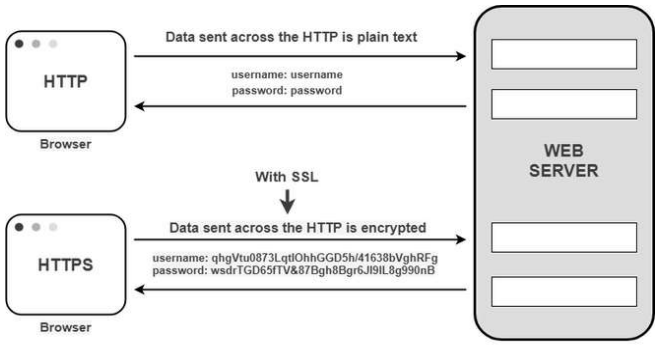
4. Limitations of HTTP

- **Statelessness**: No session information is maintained by default, requiring workarounds like cookies or session storage.
- **Security**: Data is transmitted in plain text, making it vulnerable to eavesdropping or tampering.

5. Introduction to HTTPS (Hypertext Transfer Protocol Secure)

- **Definition**: HTTPS is an extension of HTTP that incorporates encryption through SSL/TLS to secure data transmission.
- **Key Features**:
  - Encrypts data, ensuring confidentiality.
  - Validates server authenticity using certificates.
  - Protects data integrity to prevent tampering.

6. How HTTPS Works



- **1. SSL/TLS Handshake**:
  - The client requests a secure connection.
  - The server responds with its digital certificate.
  - Both parties agree on encryption protocols and keys.
- **2. Data Encryption**: Once the handshake is complete, all data exchanged is encrypted using symmetric encryption.
- **3. Authentication**: The server proves its identity using a trusted Certificate Authority (CA).
- **4. Integrity**: Data is checked for tampering using cryptographic hash functions.

7. Key Components in HTTPS

- **SSL/TLS**:
  - **Secure Sockets Layer (SSL)**: Deprecated.

    - **Transport Layer Security (TLS)**: Modern standard for secure communication.
- **Digital Certificates**:
    - Issued by Certificate Authorities (CAs) like Let's Encrypt or DigiCert.
    - Include details like the server's public key and domain name.
- **Encryption**: Ensures that data is readable only to intended parties.

8. HTTP vs HTTPS

Feature	HTTP	HTTPS
Encryption	None	Encrypted using SSL/TLS
Port	80	443
Security	Vulnerable to eavesdropping	Secure from interception
Performance	Slightly faster	Slightly slower (encryption overhead)
Use Case	Non-sensitive data transmission	Secure transactions (e.g., banking, login)

9. Advantages of HTTPS

- **Confidentiality**: Protects sensitive data like passwords and credit card details.
- **Integrity**: Ensures that data is not tampered with during transmission.
- **Authentication**: Confirms the identity of the website, preventing phishing attacks.
- **SEO Benefits**: Search engines like Google prioritize HTTPS websites.

10. HTTP/HTTPS in Real-World Applications

- **Web Browsing**: HTTPS is now a standard for most websites (e.g., Gmail, Facebook).
- **E-Commerce**: Secure payment processing.
- **APIs**: Use HTTPS to ensure secure API communication.
- **IoT Devices**: Securing data exchanges between devices and servers.

11. Emerging Trends in Web Protocols

- **HTTP/2**: Improved speed and efficiency with features like multiplexing and header compression.
- **HTTP/3**: Uses QUIC protocol for faster and more reliable connections.
- **Zero Trust Security**: Enhanced authentication and encryption mechanisms.
- **TLS 1.3**: Latest version of TLS, offering better security and faster performance.
