What Is a Server

A server is a central computer that is used to server data to different clients through a network.

Server Demo

I am gonna be skipping this section for now...
Just get a package and run it's service.
You can do apache2, ssh...

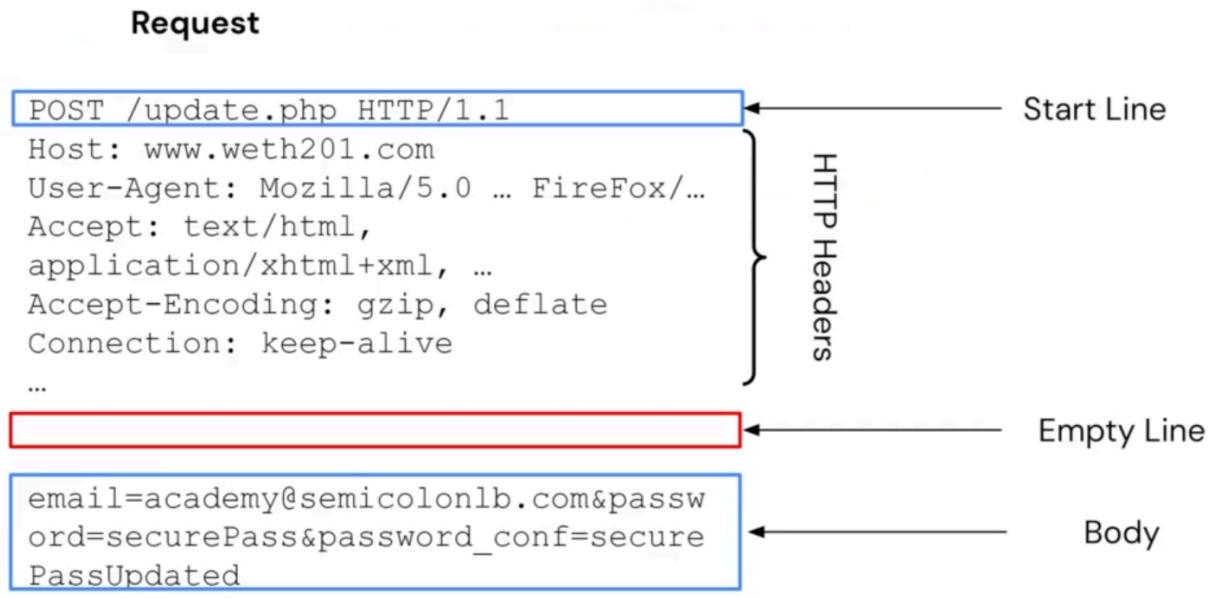
What Is DNS

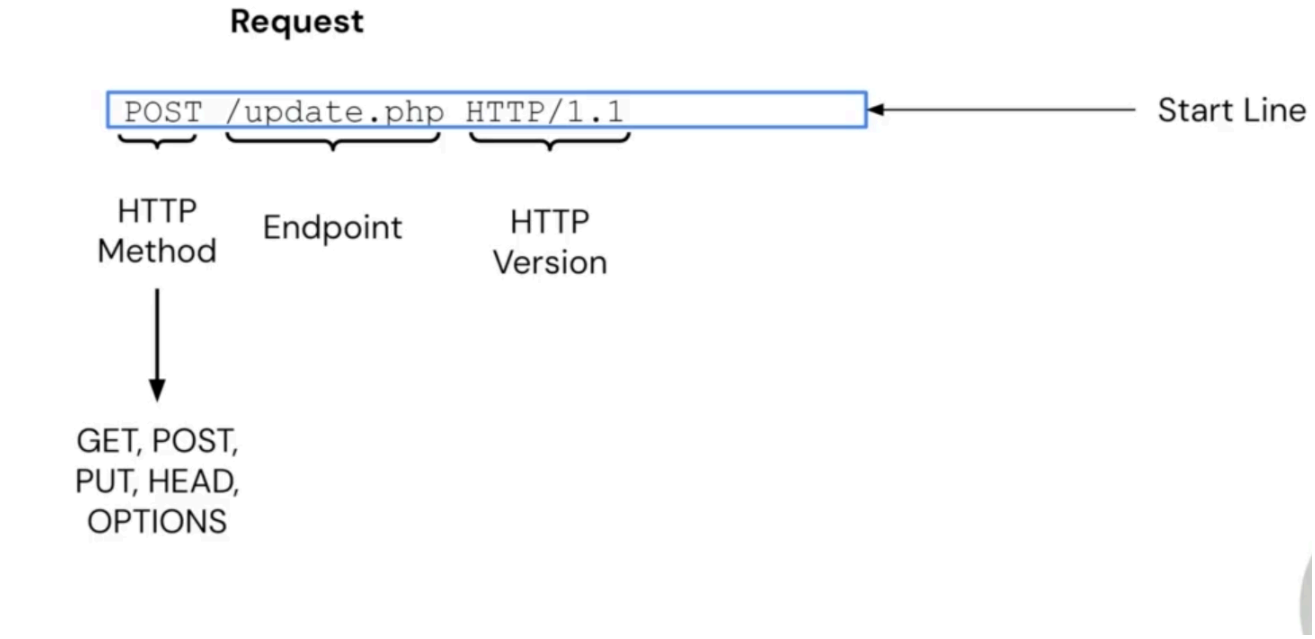
Domain Name System

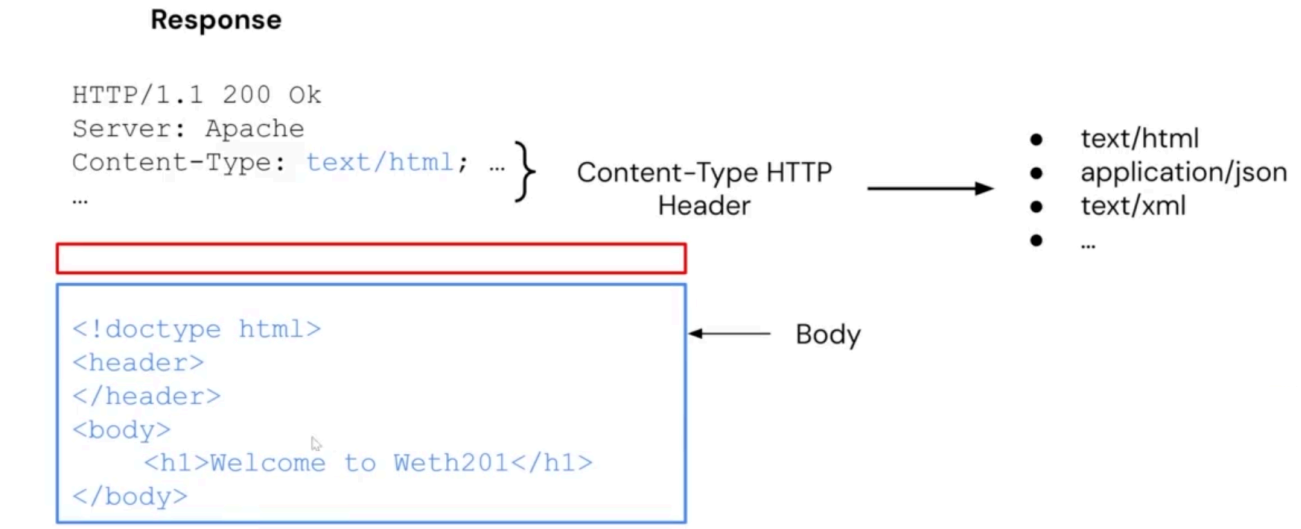
Used to link names to ips.

> To setup a local DNS you can use dnsmasq.

HTTP REQUEST RESPONSE







Modern Web Architecture

MPA

Multi Page Application, each navigation loads an entire new page.

SPA

Single Page Application, doesn't reload everything when navigating, only requests new stuff.

API - Application Programming Interface

An **API** is a way for different software systems to **communicate** with each other. In web development, it usually refers to a set of **HTTP-based endpoints** exposed by a server that clients (like web apps or mobile apps) can call.

Rest API

REST (Representational State Transfer) is the most common API style.

- Uses standard HTTP methods:
  - `GET` — Read data
  - `POST` — Create data
  - `PUT` — Update/replace data
  - `PATCH` — Partially update data
  - `DELETE` — Delete data
- Each URL (endpoint) typically represents a **resource**.
- Responses are usually in **JSON**.

GraphQL

GraphQL is a newer API style developed by **Meta (Facebook)**

- Uses a **single HTTP endpoint** (usually `/graphql`)
- Clients send **queries** or **mutations** in the request **body**
- Lets you **request exactly the data you need**
- Uses a strongly-typed schema

API Keys

What is an API Key?

An **API key** is a **unique identifier/token** used to authenticate a client (e.g. user, app, service) making a request to an API.

It says:

> “Hey, I'm allowed to access this API.”

What It Does:

- **Identifies** the caller
- Sometimes **limits** or **tracks** usage (rate limits, billing)
- Can be used to **authorize access** to specific features/data

---

🛠️ How It's Used:

Request Example:

```
GET /weather?city=beirut&units=metric
Host: api.example.com
Authorization: Bearer YOUR_API_KEY
```

Or:

```
GET /weather?city=beirut&apikey=YOUR_API_KEY
```

Or via headers:

```
x-api-key: YOUR_API_KEY
```

API Key vs Other Auth

Method	Auth Level	Who uses it
API Key	Basic identity	Public/open APIs
OAuth Token	User-level auth	Securing user data
JWT	Stateless auth	Web apps / APIs
Session Cookie	Web apps	Logged-in users

Security Tips

Mistake	Correct Way
Exposing key in frontend code	Keep keys secret, use backend as proxy
Committing keys to GitHub	Use `.env` files and `.gitignore` them
Using same key forever	Rotate keys, revoke unused ones
Not restricting usage	Apply IP/domain/rate limits if supported

---

🧪 How to Test APIs with API Keys

```
curl -H "x-api-key: YOUR_API_KEY" https://api.example.com/data
```

Or with query param:

```
curl "https://api.example.com/data?apikey=YOUR_API_KEY"
```

WebSockets

What are WebSockets?

- A protocol providing **full-duplex**, **bi-directional** communication over a **single TCP connection**.
- Enables servers and clients to send messages to each other **at any time**.
- Ideal for **real-time applications** (chat, gaming, live updates).

How WebSockets work:

1. **Handshake**:
   Client sends an HTTP/HTTPS request with an `Upgrade: websocket` header to initiate a WebSocket connection.
2. **Server Response**:
   If the server supports WebSockets, it replies with a `101 Switching Protocols` response and upgrades the connection.
3. **Persistent connection**:
   After upgrade, the TCP connection stays open for continuous two-way communication.

Key features:

Feature	Description
Full-duplex	Both client and server can send data independently at any time
Low latency	Messages sent instantly without HTTP overhead
Single connection	Uses one TCP connection, no repeated HTTP handshakes
Stateful	Connection remains open until closed

WebSocket Message Types:

- **Text** — UTF-8 encoded data (e.g., JSON)
- **Binary** — ArrayBuffer or other binary formats
- **Ping/Pong** — Keep connection alive, check latency
- **Close** — Close the connection gracefully

WebSocket URL scheme:

- `ws://` — WebSocket over plain TCP (not secure)
- `wss://` — WebSocket over TLS/SSL (secure, like HTTPS)

Basic example in JavaScript (client-side):

```
const ws = new WebSocket('wss://example.com/socket');

ws.onopen = () => {
  console.log('Connection opened');
  ws.send('Hello server!');
};

ws.onmessage = (event) => {
  console.log('Message from server:', event.data);
};

ws.onclose = () => {
  console.log('Connection closed');
```

```
};

ws.onerror = (error) => {
  console.error('WebSocket error:', error);
};
```

Use cases:

- Real-time chat apps
- Multiplayer games
- Live sports or stock updates
- Collaborative editing (Google Docs style)
- Notifications and alerts

Comparison: WebSocket vs HTTP Polling

Aspect	HTTP Polling	WebSocket
Latency	High (request/response for each poll)	Low (persistent connection)
Bandwidth	Higher (repeated HTTP headers)	Lower (less overhead)
Server load	Higher (many HTTP requests)	Lower (single connection per client)
Complexity	Simpler to implement	More complex server/client setup

---

Security considerations:

- Always use `wss://` (TLS encrypted) in production
- Authenticate users before upgrading connection
- Handle connection timeouts and errors gracefully
- Limit message size to prevent DoS attacks

Load Balancer

For large servers, at some point we might need to create multiple servers instead of one big server for easier scalability.

Traffic Distribution: Load balancers efficiently distribute incoming traffic onto the servers.

So they offer high reliability and availability.

They offer:

- Scalability
- Health Checks (for the backend servers)

Web Caching

Web caching stores frequently accessed data.
So instead of going to the server every time, a cache server will send cached data back.

Content Delivery Networks

CDNs are geographically distributed servers that cache web content close to users.

They are used for lower latency and better overall performance by being closer to users. (less packet travel times)

WAF - Web Application Firewall

Between the web application and the internet, checks for sussy traffic and blocks them before they get to the web servers.

URL

protocol://hostname[:port]/[path/]file[?parameter=value]
Protocol: Used protocol
Hostname: Host
Port: (optional) if not default port
Path: (optional) if not in parent directory
File: file to be accessed (might be optional, use default file)
Parameter: Query Parameters

Access Controls

It's goal is authorization.

Vertical

Admin > Mod > User

Horizontal

User - User - User

Context Dependent

Depending on the current state of the application.

IDOR - Insecure Direct Object References

Accessing objects by chancing the reference call in the request, for example changing the ID in a delete request.

How to mitigate:

Indirect Object Reference.

Add proper authentication and authorization. (Access controls)

XSS - Cross Site Scripting

XSS allows us to execute javascript code in the user's stead.

Three types:

- **Reflected**
- **Stored**
- **DOM-based**

Reflected XSS

On e commerce website, user uses the search feature.

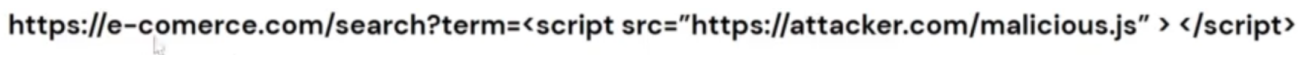
keyboard -> search

Page returned:

Search Results for **keboard**: -> **REFLECTION POINT**
blah blah blah
...

the search we used (keybaord, was then displayed by the html)

=> Might have Reflected XSS



Add the <script> into the search.

Stored XSS

More dangerous than reflected XSS.

Like reflected but stored in database or smth.

Everyone who loads the thing will also load my script.

No need to interact with the victim too.

If it was possible in comment for example, i submit the script, other ppl go to comments and they execute the script bcz database loaded it.

SQL Injection (SQLi) – Notes

What is SQLi?

SQL Injection is a vulnerability that occurs when user input is **unsafely** included in SQL queries, allowing attackers to:

- Read or modify the database
- Bypass login/authentication

- Dump sensitive data
- Execute administrative operations

How it works

Untrusted input is **injected directly** into an SQL query:

```
SELECT * FROM users WHERE username = 'admin' AND password = 'pass';
```

If unsanitized, attacker inputs:

```
' OR 1=1 --
```

Query becomes:

```
SELECT * FROM users WHERE username = '' OR 1=1 --' AND password = '';
```

Always true → Authentication bypassed.

Types of SQL Injection

---

1. Classic / In-band SQLi

Direct results in the browser/app.

a. Error-based

- Forces the database to generate errors that leak data.
- Example:

```
' AND extractvalue(1, concat(0x7e, version())) --
```

b. Union-based

- Uses `UNION` to combine results from multiple queries.
- Example:

```
' UNION SELECT null, username, password FROM users --
```

---

2. Blind SQLi

App doesn’t show errors, but you infer behavior from responses.

a. Boolean-based

- Send payloads that change the logic and observe differences.Z
- Example:

```
' AND 1=1 --  (returns normal)    ' AND 1=2 --  (returns blank/error)
```

b. Time-based

- Uses delays (e.g., `SLEEP(5)`) to detect if the query ran.
- Example:

```
  ' OR IF(1=1, SLEEP(5), 0) --
```

---

3. Out-of-Band SQLi

- Uses **external interactions** (DNS, HTTP) to exfiltrate data.
- Only possible if DB supports external calls (e.g., `LOAD_FILE`, `xp_dirtree` on MSSQL).

---

4. Second-Order SQLi

- Injection is stored in the database and executed later in a different query.
- Happens when an app stores unsafe input and reuses it insecurely.

Prevention

Technique	Description
Prepared statements / ORM	Use parameterized queries (no string concat)
Input validation	Whitelist expected input types
Escaping user input	Escape dangerous characters (fallback only)
Principle of Least Privilege	DB users should have only needed access
Web Application Firewalls	Can detect/stop some SQLi attempts

Real-world SQLi payloads (basic)

Goal	Payload
Bypass login	`' OR 1=1 --`
Detect blind SQLi	`' AND 1=1 --` vs `' AND 1=2 --`
Time-based detection	`' OR SLEEP(5) --`
Extract DB name	`' UNION SELECT database(), null --`
Dump all users	`' UNION SELECT user, password FROM users --`

Tools for testing:

- `sqlmap` (automated SQLi tool)
- `Burp Suite`
- `requestbin` (for OOB testing)
- Custom scripts with curl/python

Don't forget:

SQLi is **one of the most dangerous and common web vulnerabilities** (OWASP Top 10).

Always use **prepared statements** (e.g., `cursor.execute("SELECT * FROM users WHERE id = ?", [user_id])`)

Mitigation

Simply encode data, use prepared statements and validate inputs.