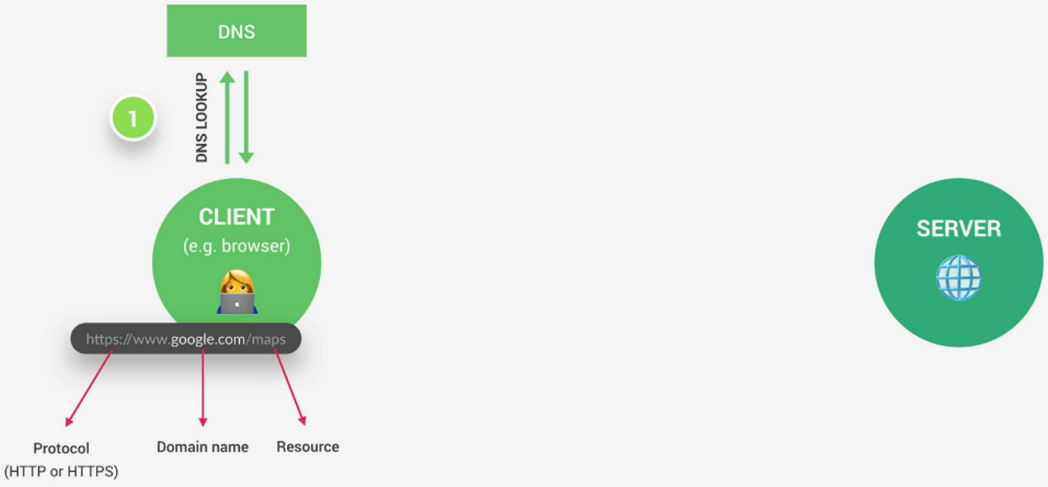
Section 3: Introduction to Back-End Web Development

23 May 2025 19:15

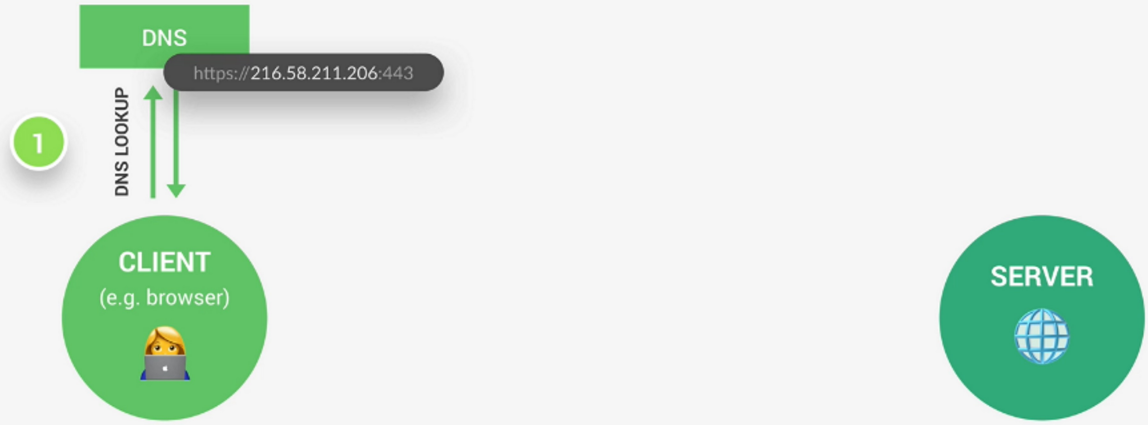
WHAT HAPPENS WHEN WE ACCESS A WEBPAGE

👉 **Request-response model** or **Client-server architecture**

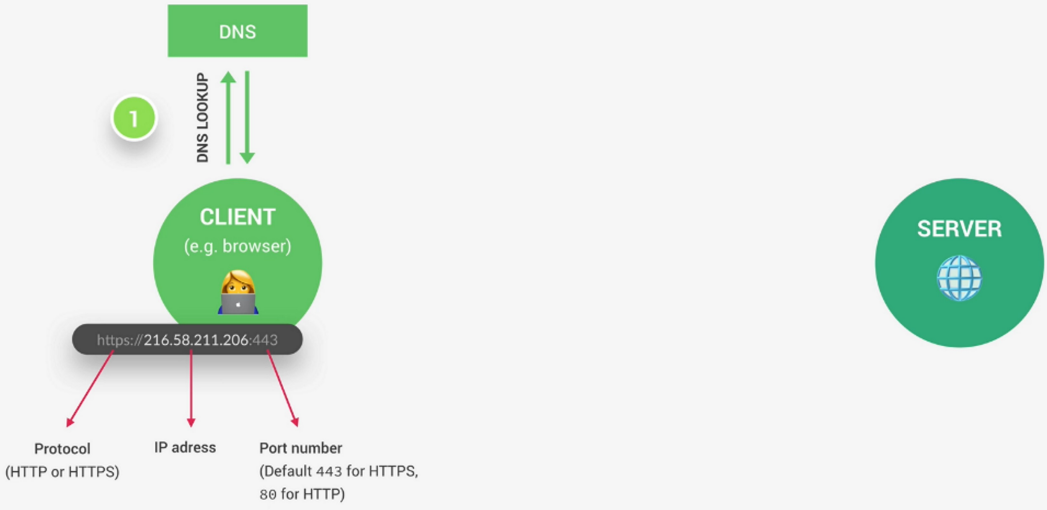
WHAT HAPPENS WHEN WE ACCESS A WEBPAGE

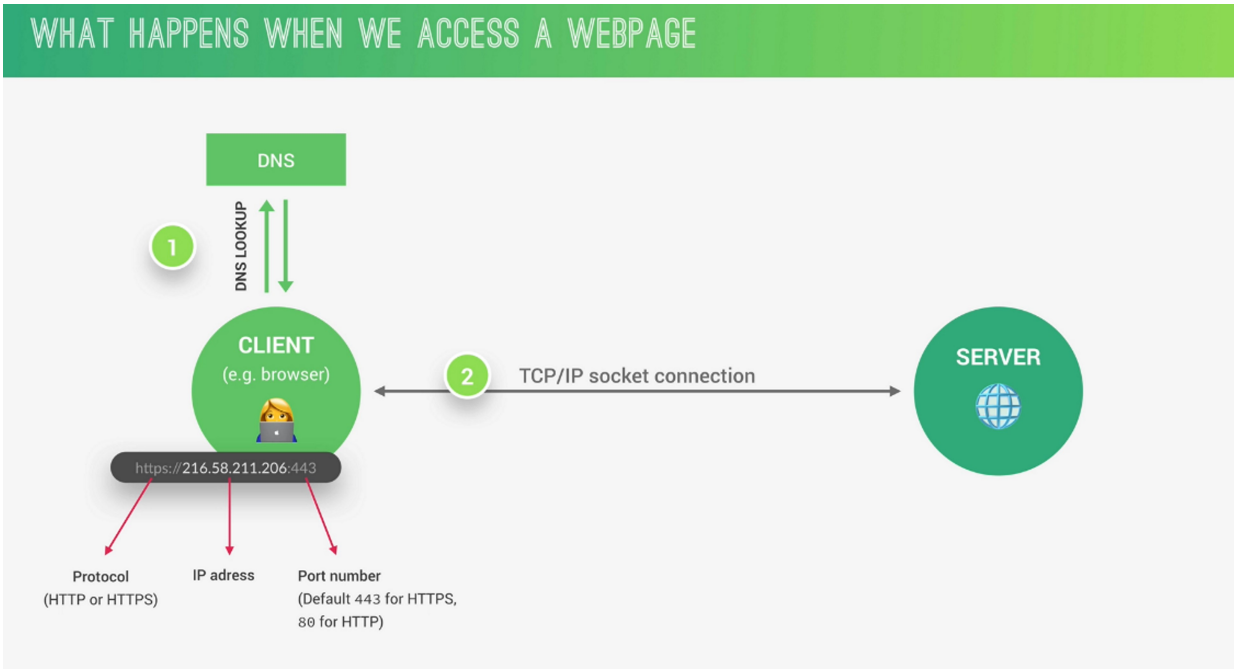


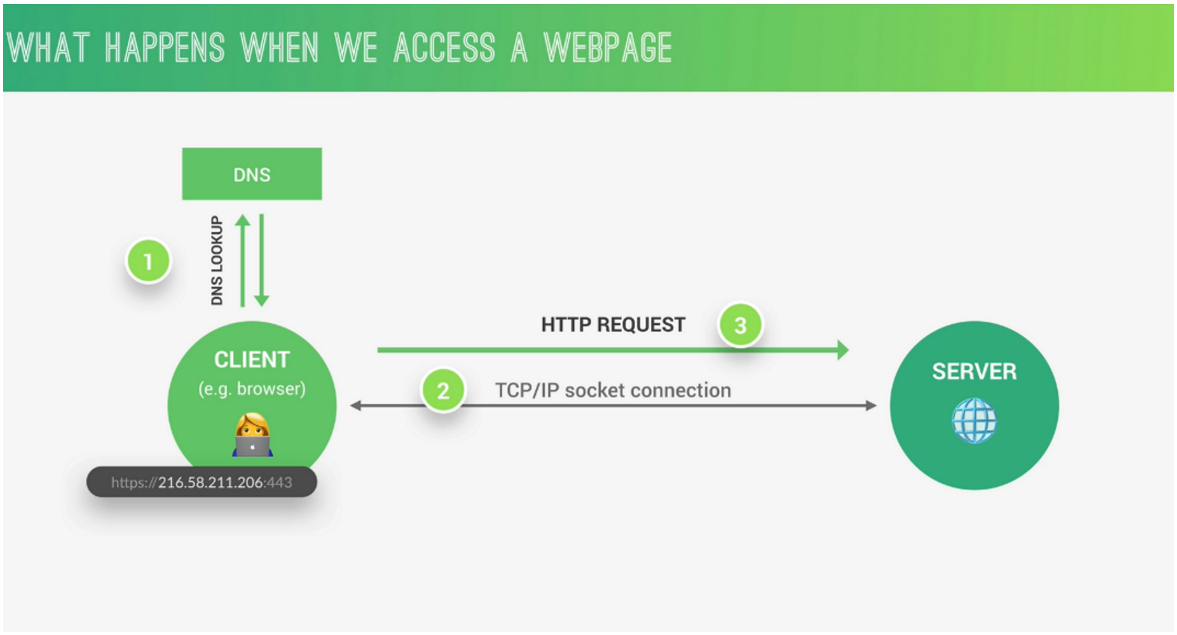
WHAT HAPPENS WHEN WE ACCESS A WEBPAGE

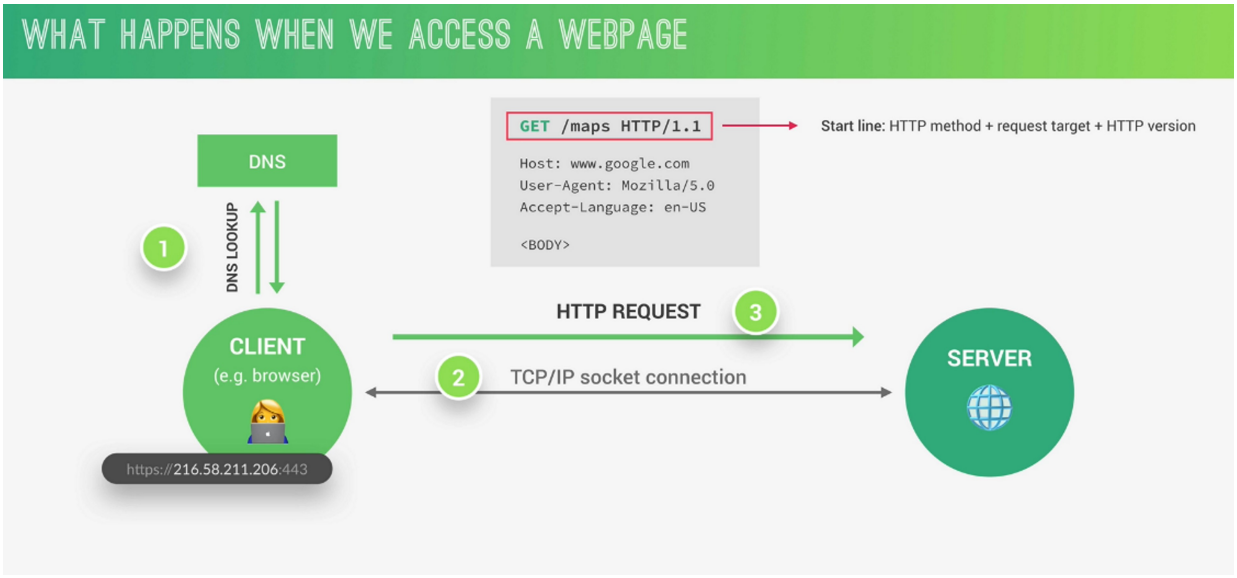


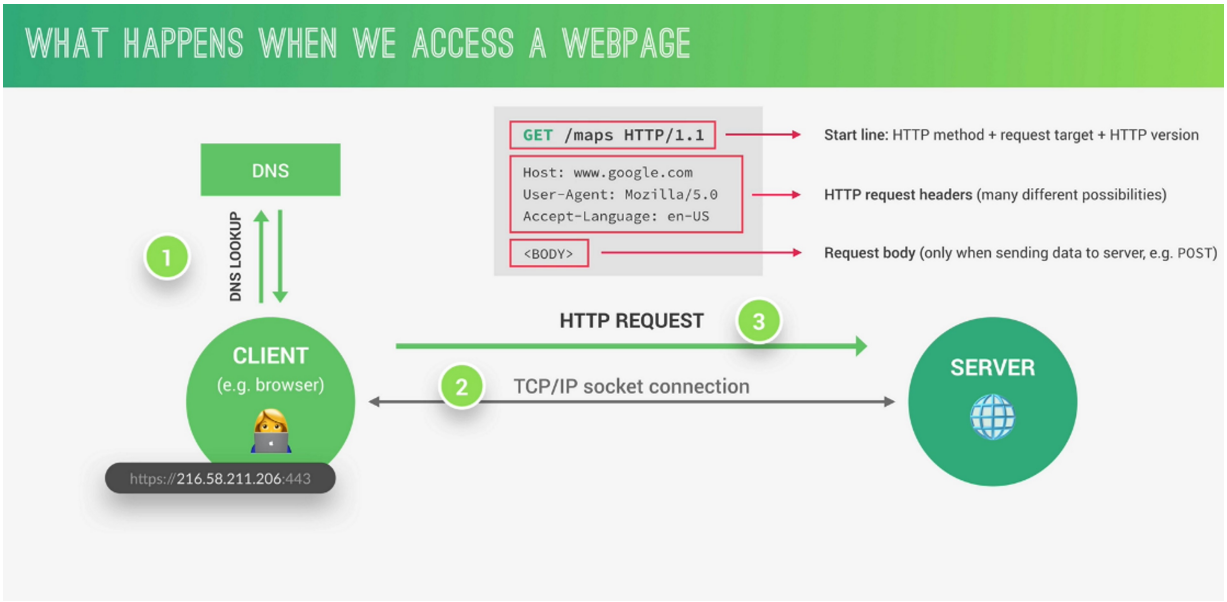
WHAT HAPPENS WHEN WE ACCESS A WEBPAGE



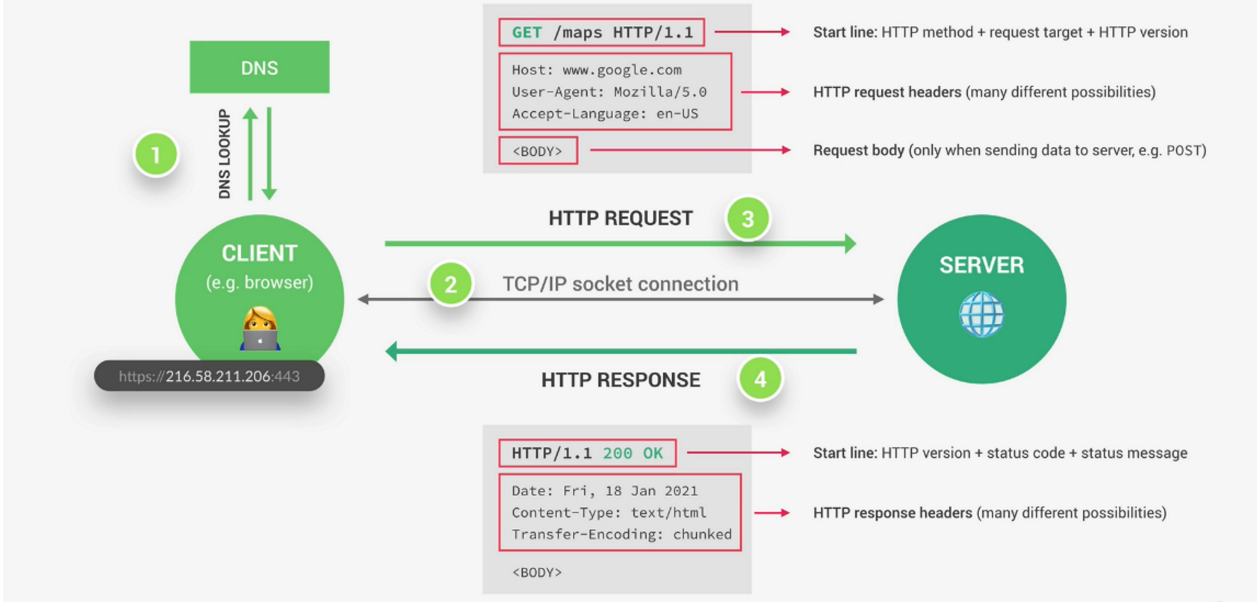

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE
DNS
1
DNS LOOKUP
CLIENT
(e.g. browser)
2
TCP/IP socket connection
SERVER
https://216.58.211.206:443
Protocol
(HTTP or HTTPS)
IP adress
Port number
(Default 443 for HTTPS,
80 for HTTP)


WHAT HAPPENS WHEN WE ACCESS A WEBPAGE
DNS
1
DNS LOOKUP
HTTP REQUEST
3
CLIENT
(e.g. browser)
2
TCP/IP socket connection
SERVER
https://216.58.211.206:443

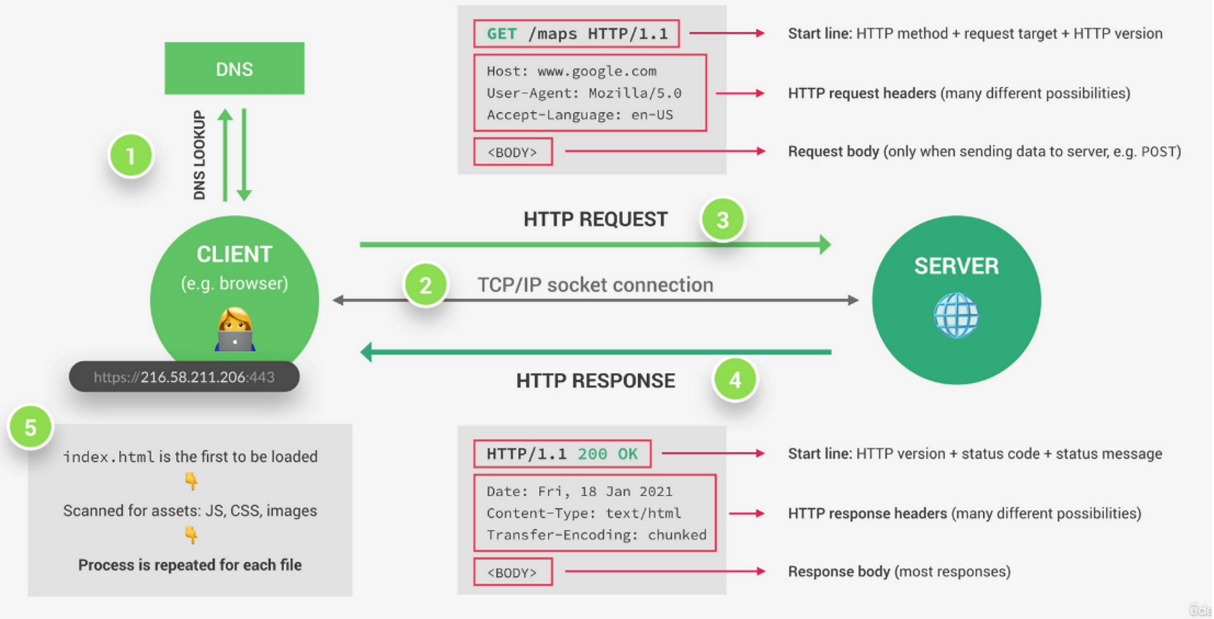

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE
GET /maps HTTP/1.1
Start line: HTTP method + request target + HTTP version
Host: www.google.com
User-Agent: Mozilla/5.0
Accept-Language: en-US
<BODY>
DNS
DNS LOOKUP
1
CLIENT
(e.g. browser)
https://216.58.211.206:443
HTTP REQUEST
3
2
TCP/IP socket connection
SERVER

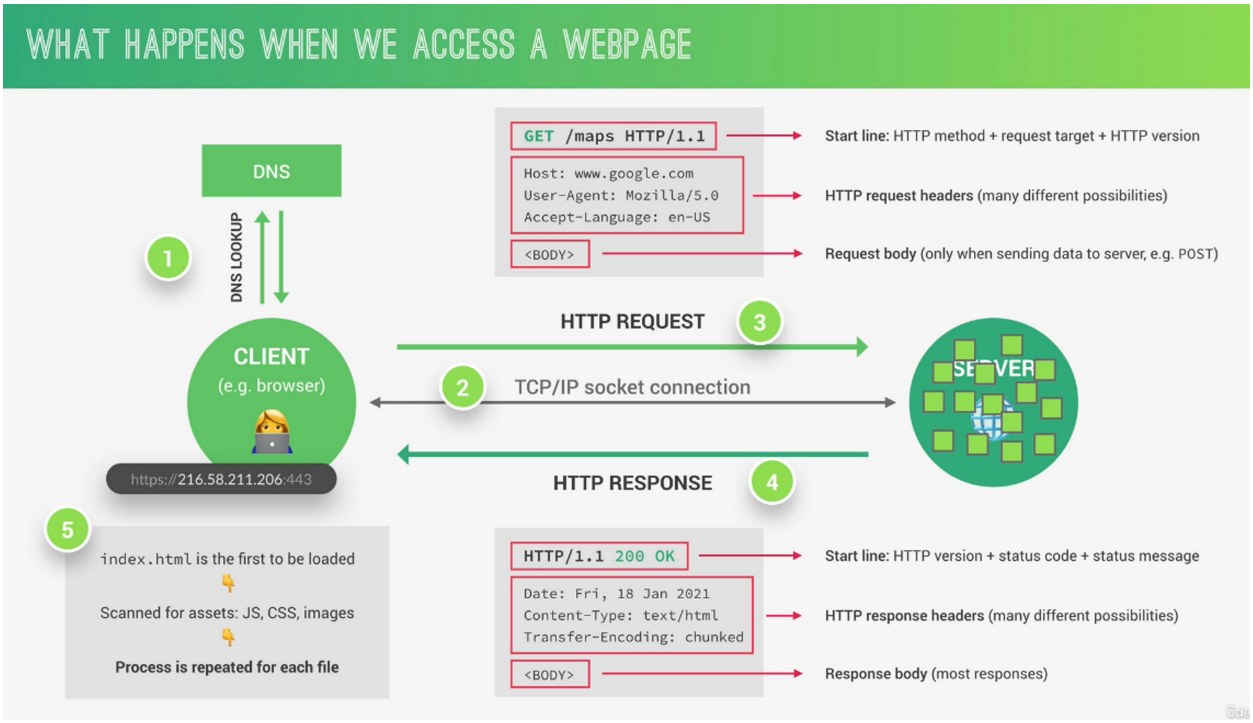

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE
GET /maps HTTP/1.1
Start line: HTTP method + request target + HTTP version
Host: www.google.com
User-Agent: Mozilla/5.0
Accept-Language: en-US
HTTP request headers (many different possibilities)
<BODY>
Request body (only when sending data to server, e.g. POST)
DNS
DNS LOOKUP
1
CLIENT
(e.g. browser)
https://216.58.211.206:443
HTTP REQUEST
3
2
TCP/IP socket connection
SERVER

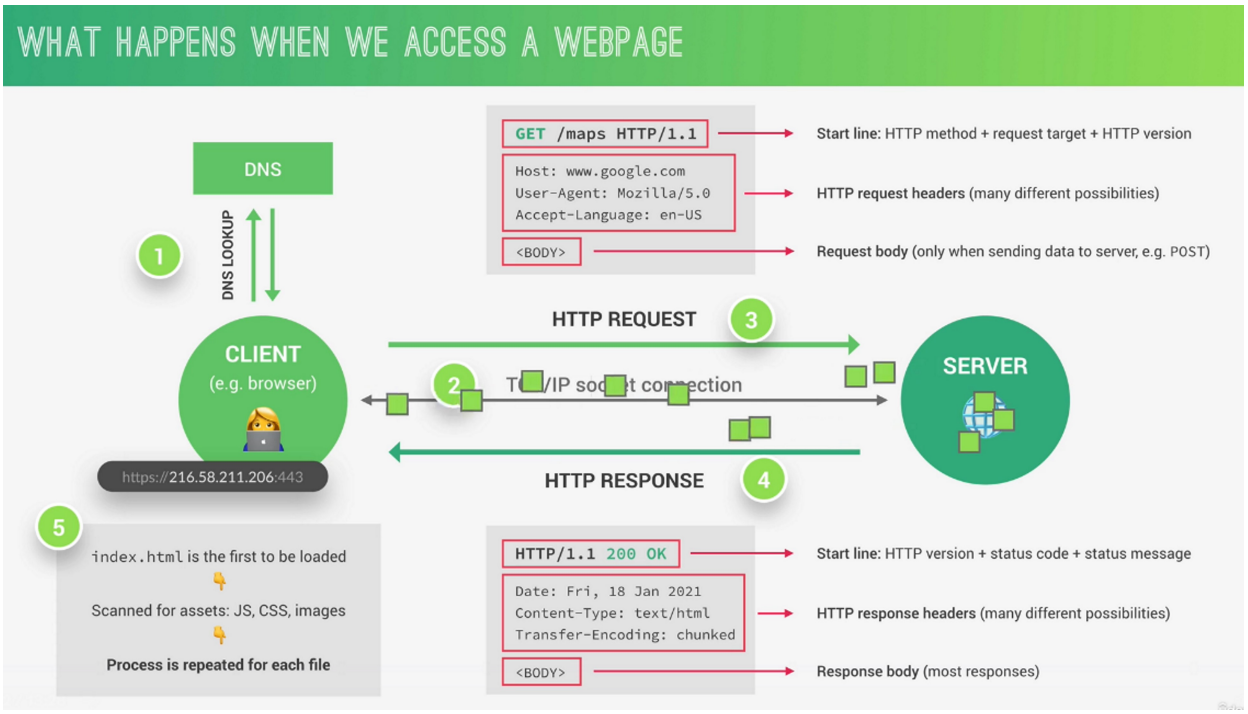
WHAT HAPPENS WHEN WE ACCESS A WEBPAGE



WHAT HAPPENS WHEN WE ACCESS A WEBPAGE







When you visit google.com/maps — here's the full journey:

1. URL Breakdown

- https://google.com/maps
  - **Protocol**: https — this means we're using **secure HTTP**.
  - **Domain**: google.com — easy-to-remember name.
  - **Path/Resource**: /maps — specific part of the site we wanna access.

2. DNS Lookup

- Browser hits up a **DNS server** (think of it as the Internet's phonebook) to convert google.com into an **IP address** like 142.250.190.14.
- Usually, your **ISP** handles this DNS stuff behind the scenes.

3. Establishing a Connection

- A **TCP connection** is made with the server.
- It uses **TCP/IP**:
  - **TCP** = breaks the data into **packets** + reassembles them.
  - **IP** = routes those packets to the correct **destination** using IP addresses.

4. Making the HTTP Request

- The browser now sends an **HTTP request**:

  http
  CopyEdit
  GET /maps HTTP/1.1
  Host: google.com
  User-Agent: Chrome...
  - **Method**: GET, POST, PUT, etc.
  - **Headers**: Extra info like browser, language, etc.
  - **Body**: Only when sending data (like from a form).

5. Server Responds with HTTP Response

- Server sends back:

  http
  CopyEdit
  HTTP/1.1 200 OK
  Content-Type: text/html
  - **Status code**: 200 (OK), 404 (Not Found), 500 (Server Error), etc.
  - **Headers**: Info about the response.
  - **Body**: The actual HTML/JSON/image/etc.

6. Browser Starts Rendering the Website

- It parses the HTML and then starts sending **more requests** for:
  - Images
  - CSS
  - JavaScript
  - Fonts, etc.
- Each of these triggers its own **HTTP request/response cycle**.

7. All Files Downloaded → Website is Rendered!

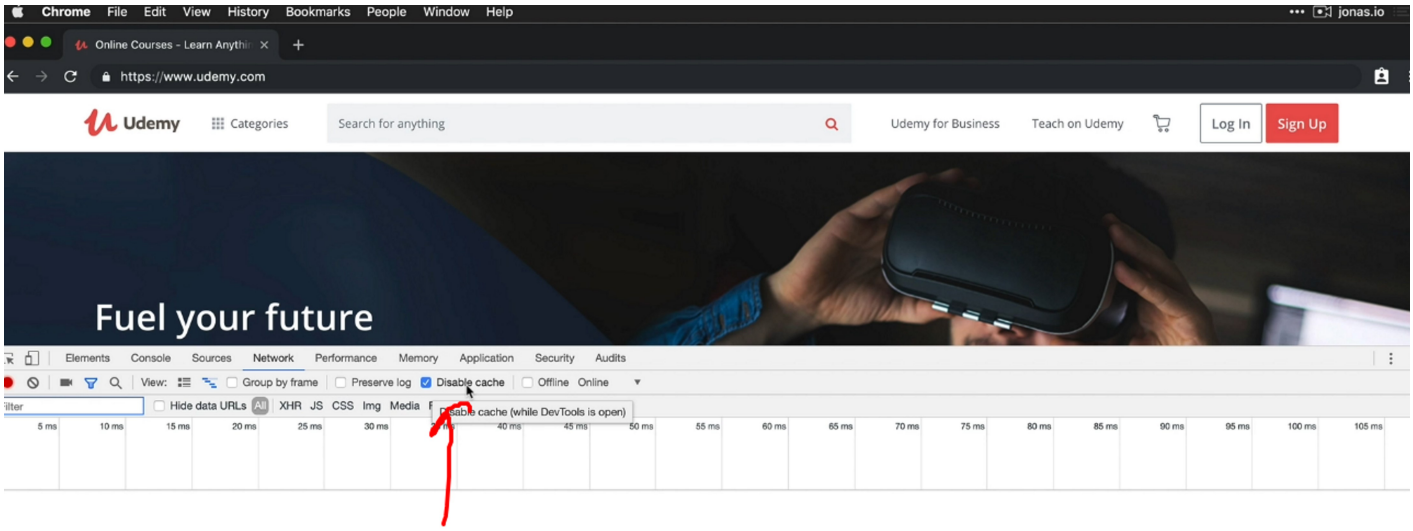
- After everything arrives, browser finally puts it all together using **HTML, CSS, and JS**.
- 🎉 You see the full Google Maps interface.

💡 **Bonus: HTTP vs HTTPS**

- **HTTPS** = HTTP + encryption using **TLS/SSL** (so stuff stays private + secure).
- Same request/response idea, but **encrypted** 🔒.

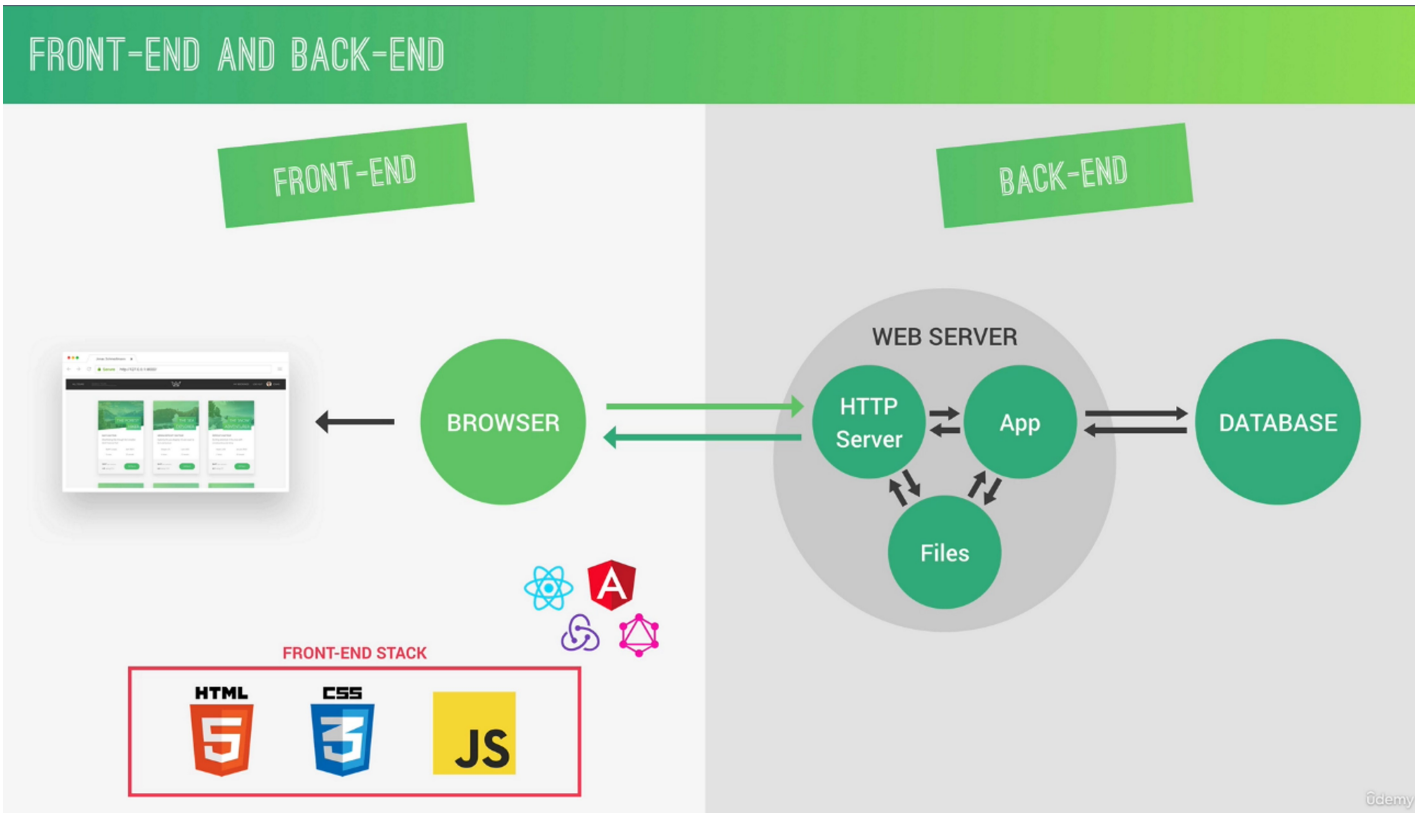
⚙️ **Why This Matters for Devs Like You:**

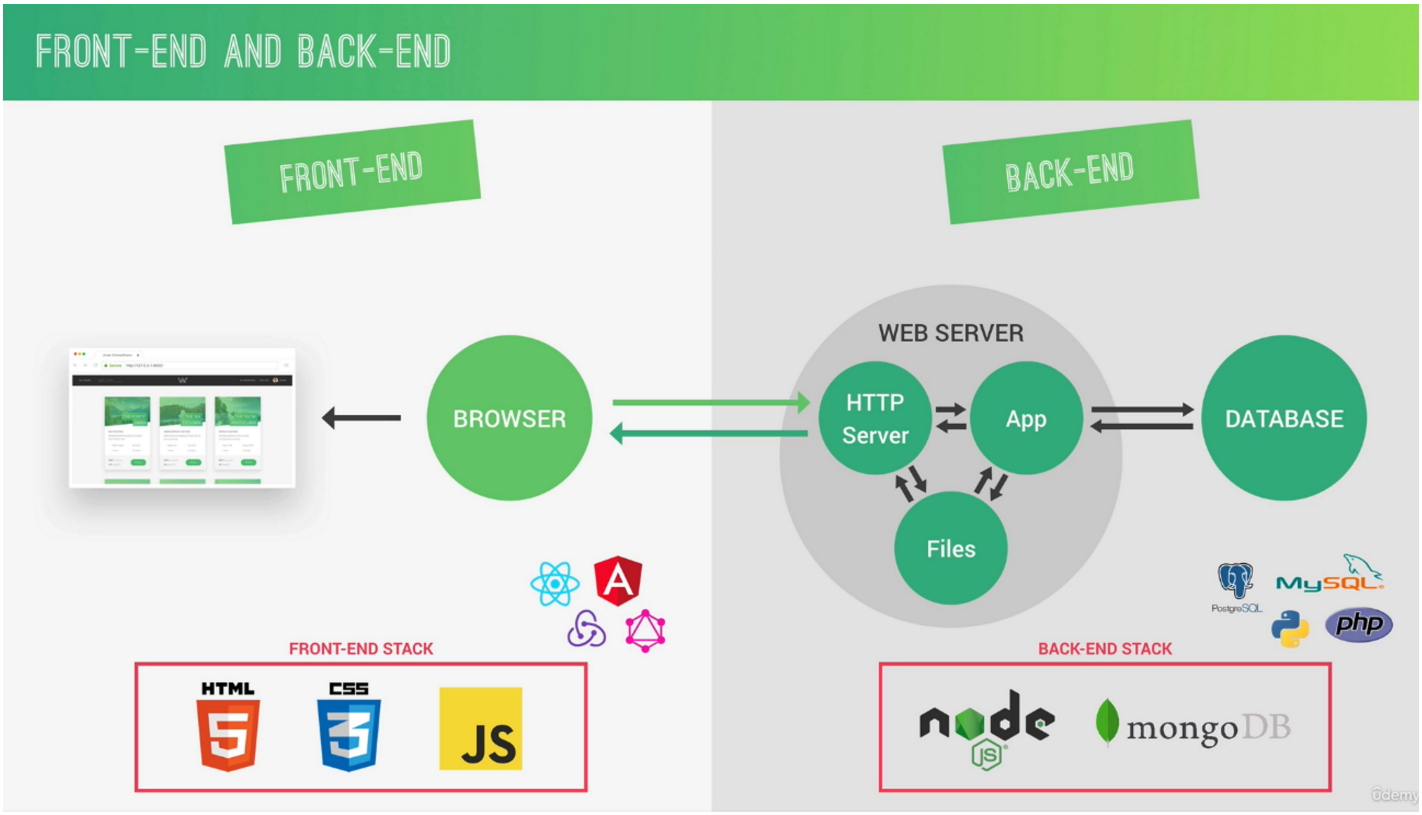
- You'll use this knowledge **daily** as a backend or full-stack dev.
- Helps when working with **APIs**, debugging network issues, or optimizing performance.
- Plus, this is a **common topic in interviews** for backend roles!



Recording network activity...

Perform a request or hit ⌘ R to record the reload.


FRONT-END AND BACK-END
FRONT-END
BACK-END
WEB SERVER
BROWSER
HTTP Server
App
DATABASE
Files
FRONT-END STACK
HTML
CSS
JS


FRONT-END AND BACK-END
FRONT-END
BACK-END
WEB SERVER
BROWSER
HTTP Server
App
DATABASE
Files
FRONT-END STACK
HTML
CSS
JS
BACK-END STACK
node
mongoDB
MySQL
PostgreSQL
php

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO BACK-END WEB DEVELOPMENT

LECTURE

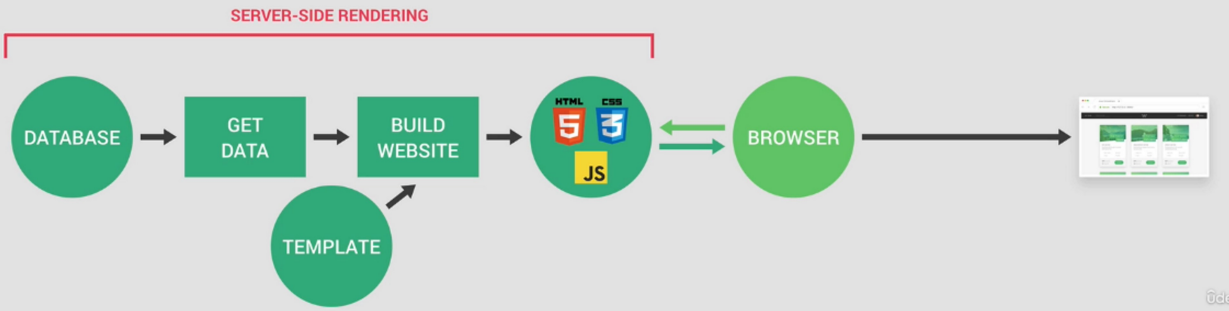
STATIC VS DYNAMIC VS API

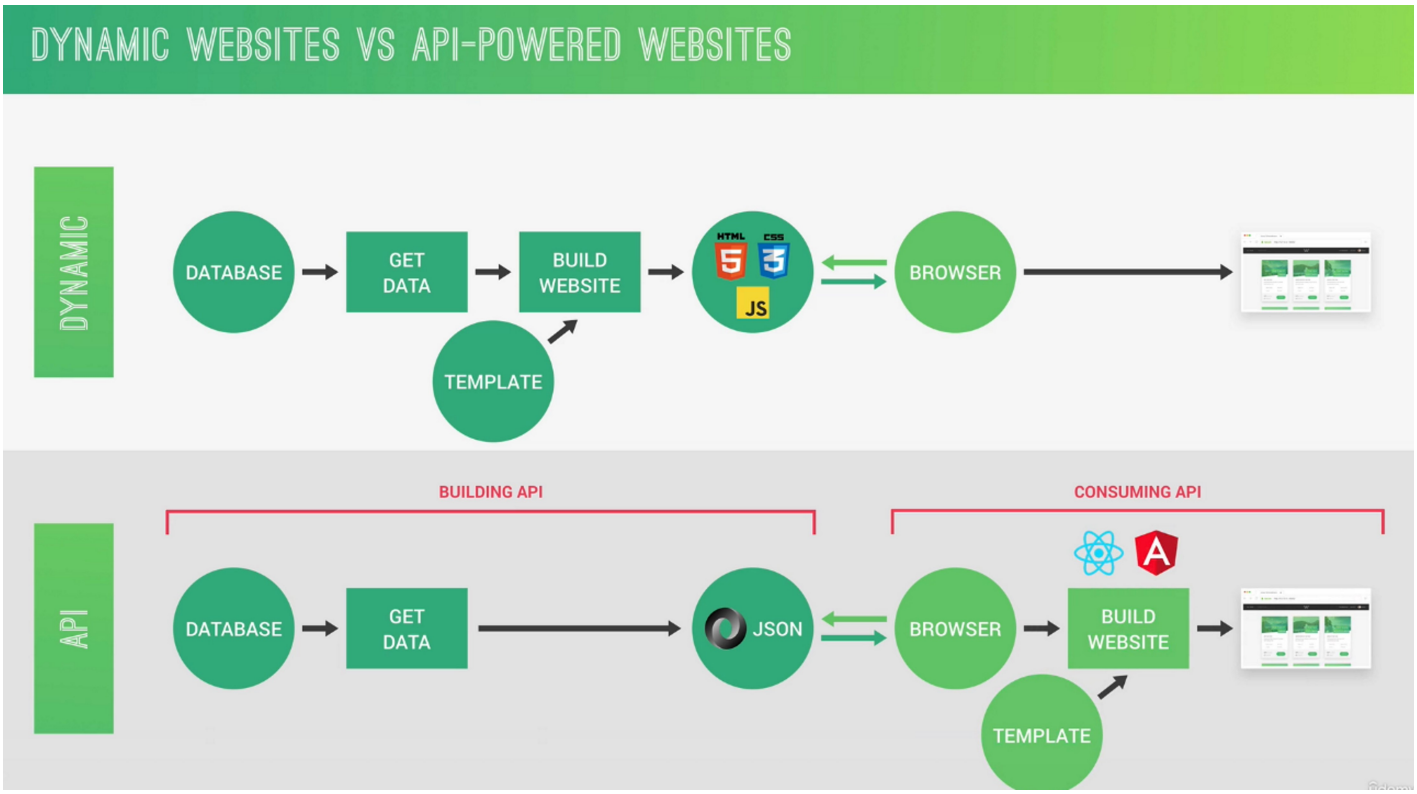
STATIC WEBSITES VS DYNAMIC WEBSITES

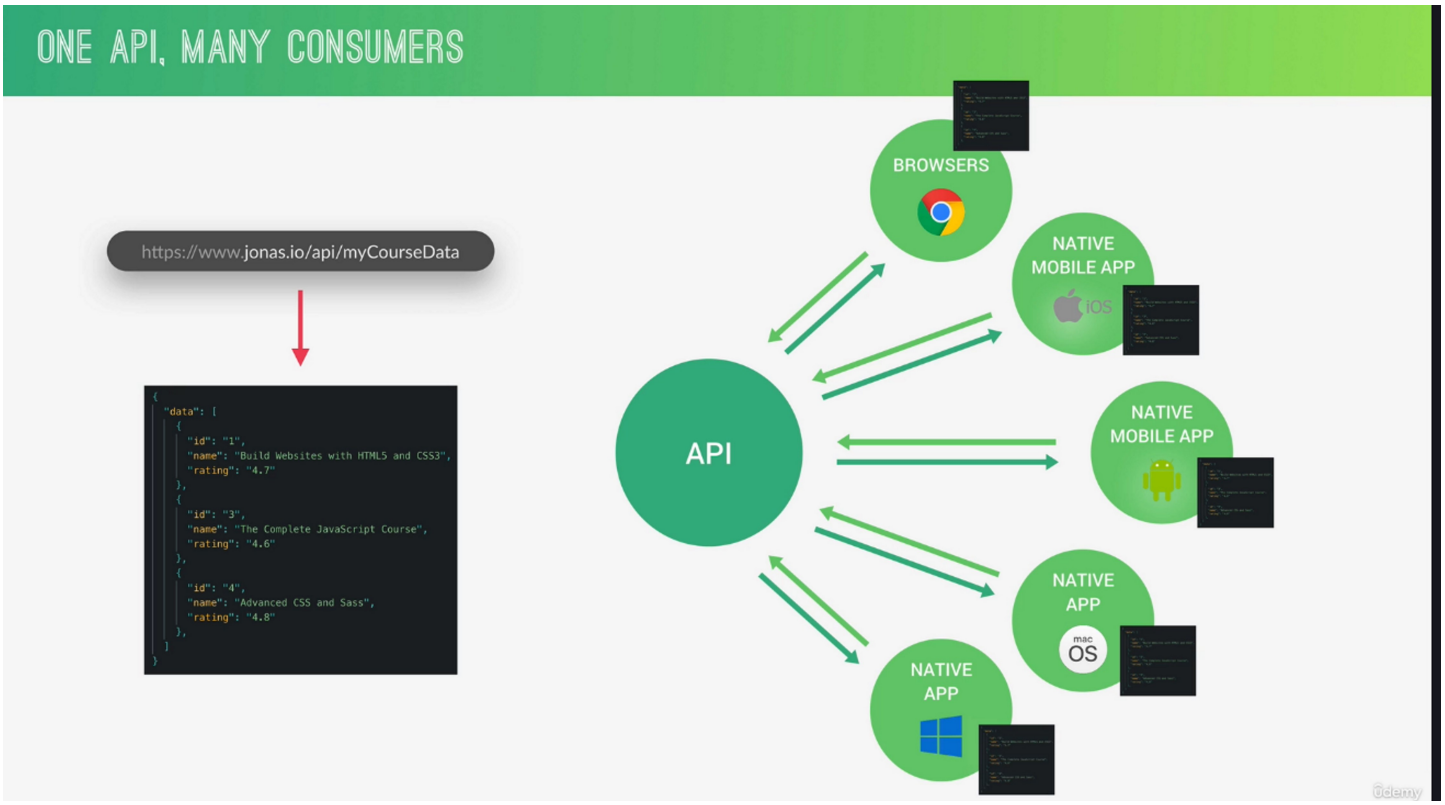
STATIC

JavaScript ≠ Dynamic

DYNAMIC







🔥 Static vs Dynamic vs API-Powered Websites – What's the Real Deal?

□ 1. Static Websites = Set in Stone

- These are like printed posters. Once they're made,

they never change unless someone **manually** edits and re-uploads the files.

- Think: pure HTML, CSS, maybe a sprinkle of JavaScript.
- There's **no back-end**, no server thinking, no database.
- Example: A portfolio site where the same info shows every time.

  💡 Even if there's JavaScript making cool animations on a static page, it's still *static* because the content doesn't change based on who's visiting or what data they bring.

⚙️ 2. Dynamic Websites = Built on the Fly

- These are like a restaurant menu that changes every time based on who's ordering 🍔🥗🍜.
- When someone visits, the server says: "Hold up, let me cook something up for you based on your data."
- It pulls stuff from a **database**, mixes it with a **template**, and sends it over as a full HTML page.
- Example: Facebook, Twitter, e-commerce sites like Amazon.

  🔁 This process is called **Server-Side Rendering (SSR)** – because the server assembles the page *dynamically* before sending it to your browser.

🧠 3. API-Powered Websites = Data Only, Build in the Browser

- These are smarter. They split the job: the **backend just sends the data** (usually as JSON), and the **frontend builds the page**.
- Example: React app hits an API endpoint, gets a list of blog posts, and renders it nicely.
- Nothing comes pre-built from the server — it's like giving ingredients to the browser and letting it cook the dish 🍱.

  ⚡ This is called **Client-Side Rendering (CSR)** — since the browser does the heavy lifting.

💡 So...What's an API Anyway?

API = Application Programming Interface

It's like a **middleman** that lets software talk to each other.

- You hit an API endpoint (like jonas.io/api/myCourseData)…
- The server gives you just the **data** (not the whole website)…
- Then your frontend (React, Vue, etc.) uses that data to show cool stuff.

APIs aren't just for browsers. They can be used by mobile apps, desktop apps, smartwatches, fridges… basically anything with code in it.

Big Picture Time

Type	Built Where?	Sends What?	Good For
Static Website	During deploy	Full HTML/CSS/JS	Simple blogs, portfolios
Dynamic Website	On the server	Full HTML (SSR)	News sites, social networks
API-Powered Site	In the browser	Just JSON data (CSR)	SPAs, mobile/web hybrid systems

Why Should Back-End Devs Care?

- APIs are **modular**. One API can feed **multiple clients**: web, iOS, Android, smartwatch — you name it.
- It's **scalable**, **clean**, and **future-proof**.
- Businesses even sell just their APIs to other devs. That's a whole revenue stream .

TL;DR for you, bro:

- Static = no backend, just files.
- Dynamic = backend builds pages on each request.
- API-powered = backend sends data only, frontend builds the UI.
- APIs are king when you want flexibility and cross-platform power.