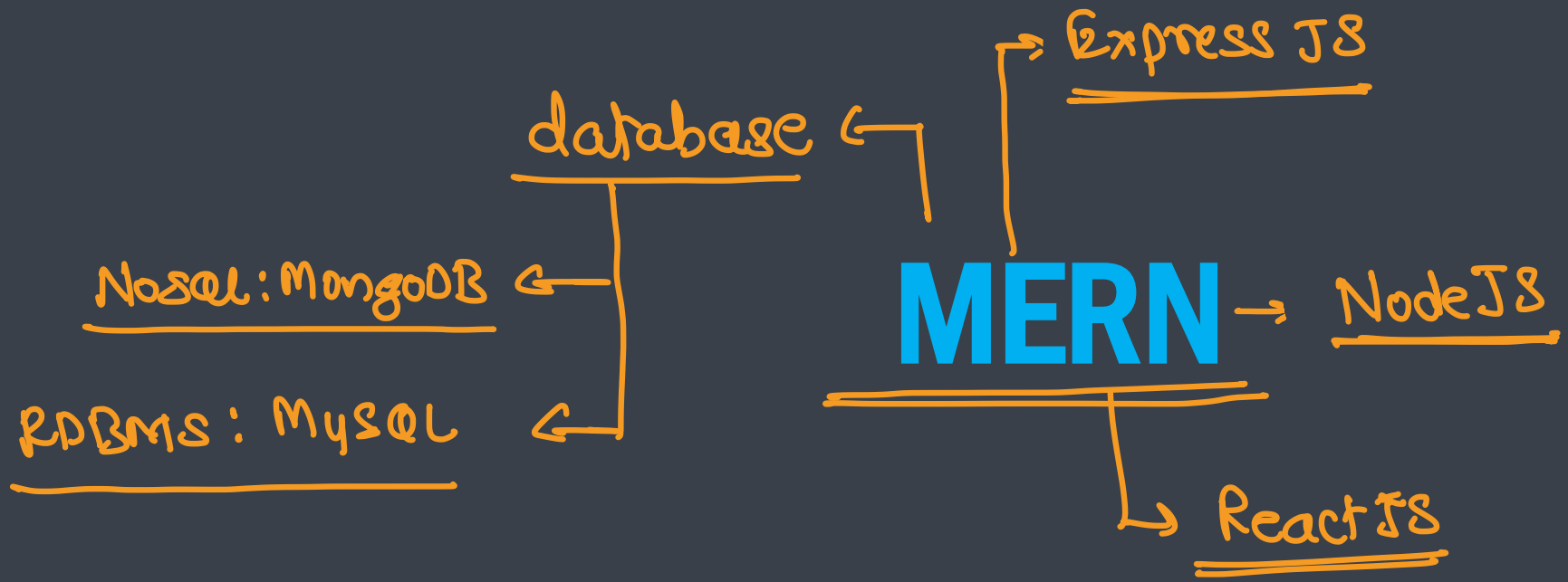
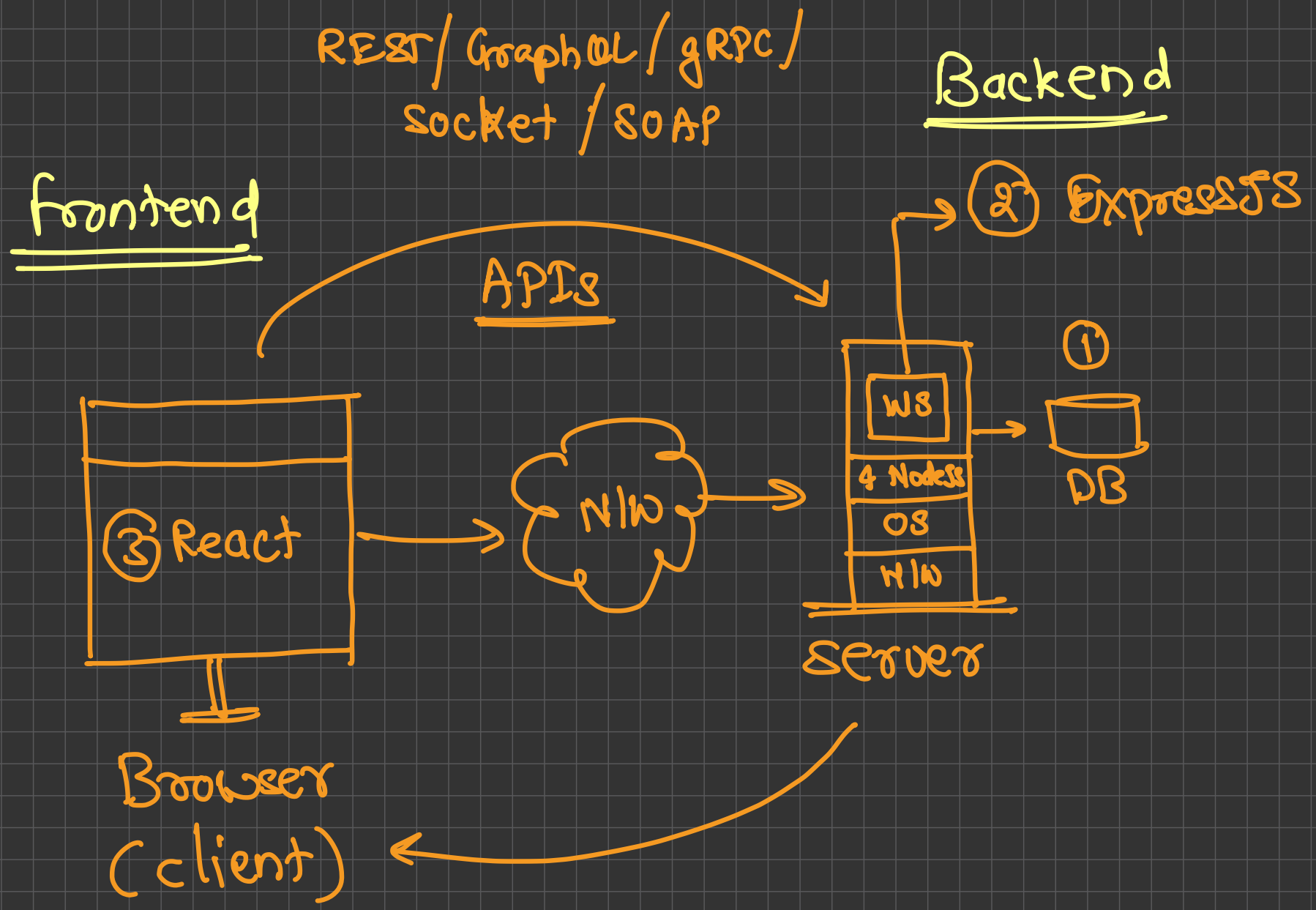

MERN
Express JS
database
NoSQL: MongoDB
RDBMS: MySQL
NodeJS
ReactJS

Full stack application = Frontend + Backend



Course Contents – NodeJS → platform

Getting Started

HTTP / HTTPS client-server architecture

Request - Response

Web Architecture

Javascript Engine → Browser Engine

Browser Architecture

architecture, frameworks

JS apps work

NodeJS Foundation

Web Assembly

V8 Engine

Operating System Access

Libuv

Single threaded architecture

Event Loop

reusability → helmet, cors, MySQL2 ....

npm Modules

async / await / promises

Sync vs Async

web server

HTTP Server

Course Contents – Express → Web server → GraphQL

REST APIs → JSON

	HTTP Server vs Express	Authentication & Authorization (→ check user is valid)	(→ checking authenticated user has enough permissions → JWT token)
HTTP methods, parameters, Design pattern	REST Foundation	File Uploading	multer
simple and advanced	Routing in Express	Implementing Security	encryption, helmet
db / filter / custom	Middleware	Socket Programming	Live chat using socket.io
MySQL + mongoDB / mongoose	Database Connection	Sending Emails	nodemailer
sanitization	Error Handling & Validation	Deployment	AWS → cloud-docker

Course Contents – React (frontend) (19.0) → Graph QL

SPA → index.html

lite weight library

Js + XML designing UI

reusable entity types → class / function

properties → R/O
state → R/W

useState(), useReducer(), useRef(), useCallback(), Custom Hooks

Intro to React	
Using JSX	
Components	
Props and State	
React Hooks ***	
Implementing Themes	light + Dark = CSS

empty, regular expression,

User Input & Validation	React forms Errors
Routing in React	switching between components (pages)
Consuming Services	axios, react query, Js Fetch → REST
Context API vs Redux	context API - inbuilt Redux → global store
Payment	stripe gateway
Deployment	AWS - cloud Docker

Live chat

testing → Jest

Capstone Project – Food Delivery Application

Customer Portal

- User workflow → custom / social → signup / sign / forgot pass
- Food Items
  - Searching and Filtering
  - Review Management → Review (Rating, comments)
- Cart Implementation → Redux → checkout
- Payment using Stripe
- Orders
  - Listings
  - Cancel Order → email / notification
- Live Chat with Administrators

Admin Panel

- Dashboard → food items, users, orders & listing
- Food Item management → add / delete / updating
- Order management → listing / updating / cancel
- User management → listing / block / unblock
- Live Chat with customers → socket.io

Pre-requisites

- HTML
- JavaScript
- CSS
- A little bit of AWS → EC2 (vm)
- git → GitHub

About Instructor

- 18+ years of experience
- Associate Technical Director at Sunbeam
- Freelance Developer working with real world projects → React Native
- Developed numerous mobile applications on iOS and Android platforms
- Developed various websites using LAMP, MEAN and MERN stacks
- Languages I love: C, C++, Python, JavaScript, TypeScript, PHP, Go

Foundation

① client → Browser → URL

② Server : Not a Hardware, it is software/app

→ Web server → used to listen http/https requests
  ↳ apache, nginix, ExpressJS

→ Database server → used to store the data permanantly
  ↳ RDBMS → MySQL, PostgreSQL, SQL server, oracle..
  ↳ NoSQL → MongoDB, Redis, Neo4J, Redshift..

→ file servers → used to share files
  ↳ FTP / NFS / Samba (SMB)

Software stack (server)

① Web server ② Database ③ UI framework ④ platforms

LAMP → Linux Apache MySQL Python / Pearl / PHP

WAMP → Windows Apache MySQL Python / Pearl / PHP

MAMP → mac OS Apache MySQL Python / Pearl / PHP

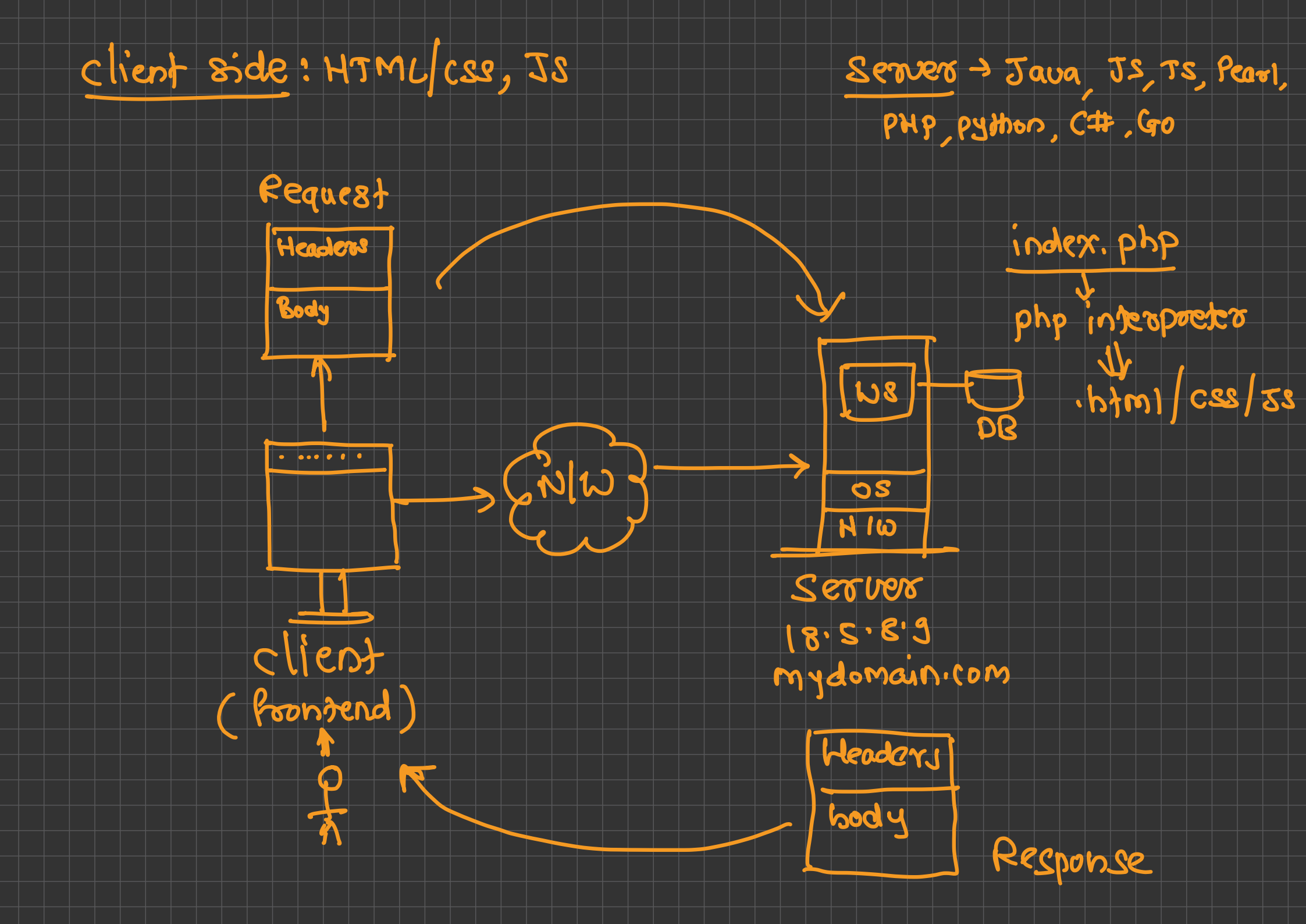
MERN → MySQL / MongoDB Express React NodeJS

MEAN → MySQL / MongoDB Express Angular NodeJS

WISA → Windows IIS server SQL server ASP.net

Web Architecture → uses client-server architecture and request-response pattern

- **Web architecture** refers to the layout and design of technologies that define how data and services are delivered over the internet, enabling interaction between **clients (users)** and **servers (applications)**.
- **Client (Frontend)**
  - The user-facing part of the application.
  - Runs in the browser.
  - Built with **HTML, CSS, JavaScript.**
  - Examples: Websites, mobile apps, SPA (Single Page Applications like React or Angular).
- **Server (Backend)**
  - Receives requests from the client and sends back responses.
  - Handles business logic, data processing, authentication.
  - Built using **Node.js, Express, Django, Ruby on Rails**, etc.
- **Database**
  - Stores and manages application data.
  - Types:
    - **Relational (SQL)**: MySQL, PostgreSQL
    - **Non-relational (NoSQL)**: MongoDB, Redis


client side : HTML/CSS, JS
Server → Java, JS, TS, Perl, PHP, Python, C#, Go
Request
Headers
Body
index.php
php interpreter
.html/css/JS
WS
DB
OS
H/W
N/W
Server
18.5.8.9
mydomain.com
client
(frontend)
Headers
body
Response

How web communication works

- **Client sends a request** to a server (usually HTTP/HTTPS).
- **Server processes** the request.
- **Server communicates with a database** (if needed).
- **Server sends a response** (usually in JSON or HTML).
- **Client displays** the result.

HTTP Request

send data → query string / url parameters / body

- An HTTP request is a message sent by the client (usually a browser) to the server to request data or perform an action (e.g., view a page, submit a form, fetch API data)

- Headers → collection of key-value pairs
  - Host: Domain name of the server
  - User-Agent: Browser or client info
  - Accept: Types of responses the client accepts
  - Content-Type: Type of data being sent (for POST/PUT)
  - Authorization: Credentials (e.g., Bearer token)
  - Cookie: Cookies sent by the client
  - Referer: The URL of the previous page
  - Origin: Origin of the request (used for CORS)

method : GET

- Body
  - Only included in some request types like POST, PUT, or PATCH.
  - Contains **data sent to the server**, such as:
  - Form data
  - JSON objects
  - File uploads

Request Methods

- **GET:** Retrieve data
- **POST:** Send new data
- **PUT:** Update existing data → full update
- **DELETE:** Delete a resource
- **PATCH:** Partially update data → partially update
- **OPTIONS:** Ask the server what is allowed

HTTP Response

- It is a message sent by the **server** to the **client (browser, app, etc.)** in reply to an HTTP request
- It contains:
  - A status line
  - Response headers
- Headers
  - **Content-Type:** Format of the response (e.g., text/html, application/json)
  - **Content-Length:** Size of the response body in bytes
  - **Set-Cookie:** Instructs client to store a cookie
  - **Cache-Control:** How and how long to cache the response
  - **Access-Control-Allow-Origin:** For CORS handling
  - **Date:** Date and time of the response
  - **Server:** Info about the server software

Body

- The actual data returned from the server. May include:
  - HTML content
  - JSON data (for APIs)
  - File (e.g., image, PDF)
  - Text/plain content

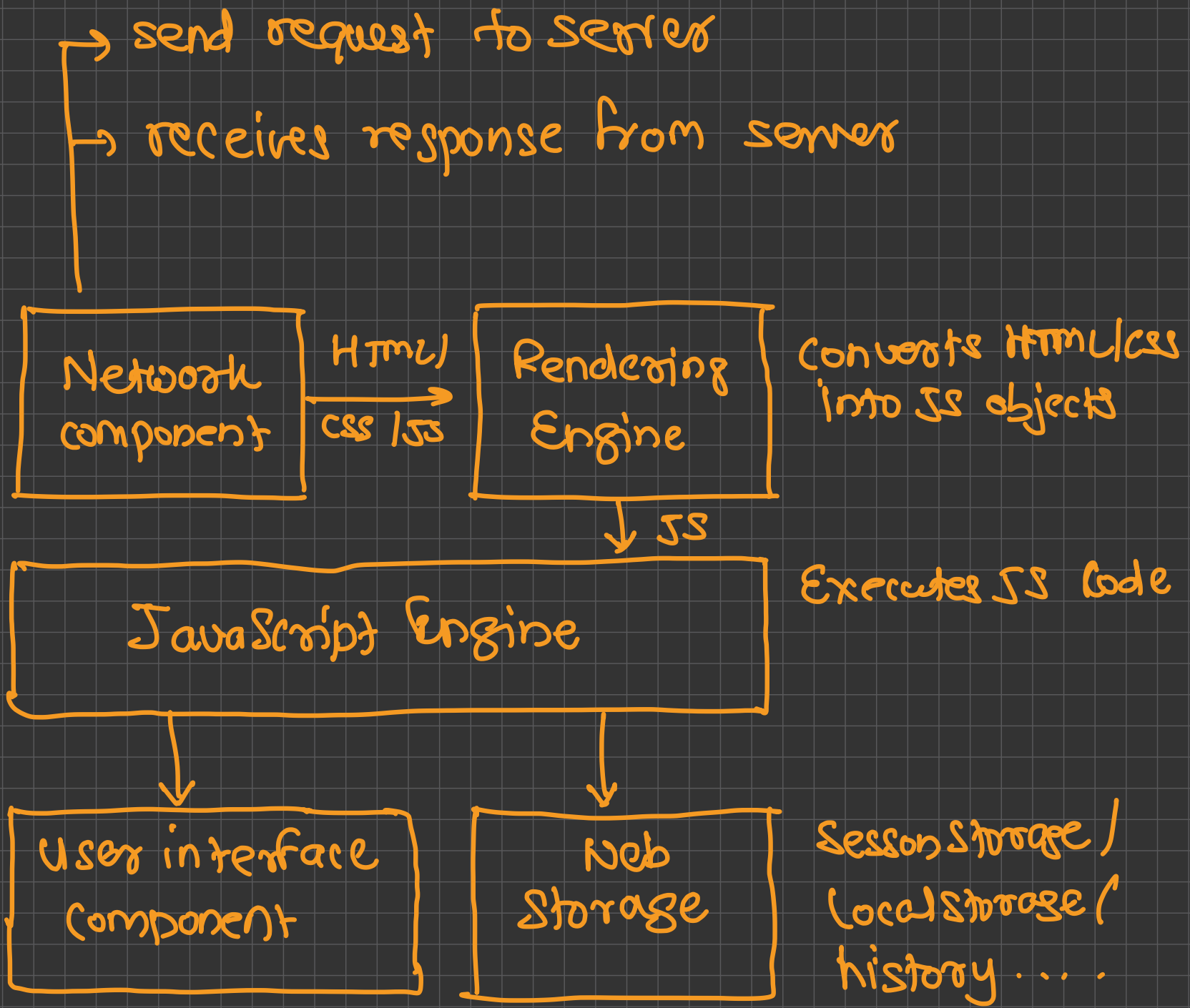
Response Code Categories → status code

- Informational responses (100 – 199) → 101 → switching protocol
- Successful responses (200– 299) → 200 ok
- Redirection responses (300 – 399)
- Client error responses (400 – 499)
- Server error responses (500– 599)

Tools and Technologies

SPA

- **Frontend:** HTML, CSS, JS, React, Vue, Angular
- **Backend:** Node.js, Express, Django, Flask
- **Database:** MongoDB, PostgreSQL, MySQL
- **Web Server:** Nginx, Apache
- **API Format:** REST, GraphQL
- **DevOps:** Docker, Git, GitHub, CI/CD, Nginx
- **Hosting:** Vercel, Netlify, Heroku, AWS, Render


send request to server
receives response from server
Network component
Html/
css /js
Rendering Engine
converts Html/css into JS objects
JS
JavaScript Engine
Executes JS Code
User interface component
Web Storage
Session Storage /
LocalStorage /
history ....

Boowser	Rendering Engine	Js Engine
Edge	EdgeHTML	Chakra
Firefox	Gecko	SpiderMonkey
Safari	WebKit	Nitro Js Engine
Chrome	Blink	V8

fastest / c++

Browser Architecture

- **Network Component**
  - It is responsible for web browsers communicating with the internet and fetching resources such as HTML, CSS, JavaScript, images, and more from web servers
  - It works alongside the rendering engine, browser UI, and, most importantly, the JavaScript engine
  - They depend heavily on the networking layer for fetching external JavaScript files, APIs, or other asynchronous data
  - Working closely with JavaScript engines ensures web pages load efficiently, maintain real-time communication and that users can interact with dynamic web content securely
- **Rendering Engine**
  - A rendering engine is a core component of web browser architecture
  - It transforms HTML, CSS, and JavaScript into a visual web page representation and works alongside the JavaScript engine to create dynamic interactions and behaviors on web pages
  - The rendering engine and JavaScript work together in these various functions:
    - **Parallel operation:** While the rendering engine constructs the DOM and CSSOM, the JavaScript engine runs the JavaScript code. They work in parallel but also synchronize when necessary
    - **DOM and CSSOM manipulation:** The JavaScript engine can modify the DOM and CSSOM so the rendering engine works to rerender the page or parts of it when such changes occur
    - **Reflow or repaints:** When JavaScript manipulates the layout or content of the page, the rendering engine may need to recompute the layout (reflow) or update the visual content (repaint) of the portions of the page

Browser Architecture

- JavaScript Engine
  - A **JavaScript engine** is a program or interpreter designed to execute JavaScript code
  - It is embedded within web browsers such as Chrome, Firefox, and Safari
  - The JavaScript engine transforms code into machine code that can be executed by browser
  - Common JavaScript engines you should be familiar with include:
    - V8 (used by Google Chrome, Microsoft Edge, and Node.js)
    - SpiderMonkey (developed by Mozilla for Firefox)
    - JavaScriptCore Nitro (used by Safari)
    - Chakra (previously used by Microsoft's Internet Explorer and Edge)
  - **Key features of JavaScript Engines**
    - **Optimized code execution:** Techniques such as inline caching and speculative optimization allow JavaScript engines to predict the types of operations and optimize for them
    - **Multithreading for heavy tasks:** JavaScript engines can offload specific tasks to background threads, such as garbage collection or complex calculations
    - **Lazy parsing and compilation:** The existence of JavaScript engines means that instead of parsing and compiling the entire code upfront, they can delay parsing less important code until it is needed

Browser Architecture

- User Interface Component
  - The user interface (UI) layer renders the visual elements users interact with when browsing the web
  - It has to work alongside JavaScript engines because they can dynamically alter the UI in various ways:
    - **DOM manipulation:** JavaScript engines can update the DOM tree, for example, by adding or removing elements
    - **Event handling:** The user interactions that the UI layer captures, such as clicks, typing, scrolling, etc., are passed to the JavaScript engine for processing
    - **Repainting:** After JavaScript updates the DOM, the UI layer has to repaint or re-render the affected parts of the page
- Data Storage Component
  - The data storage layer is essential and works with JavaScript engines to provide the ability to store, retrieve, and manage data locally within the web browser
  - This allows web browser architecture to manage user data and perform offline operations if needed
  - JavaScript engines do not provide data persistence or storage capabilities, and data storage layers enable web applications to store data persistently across or temporarily during sessions

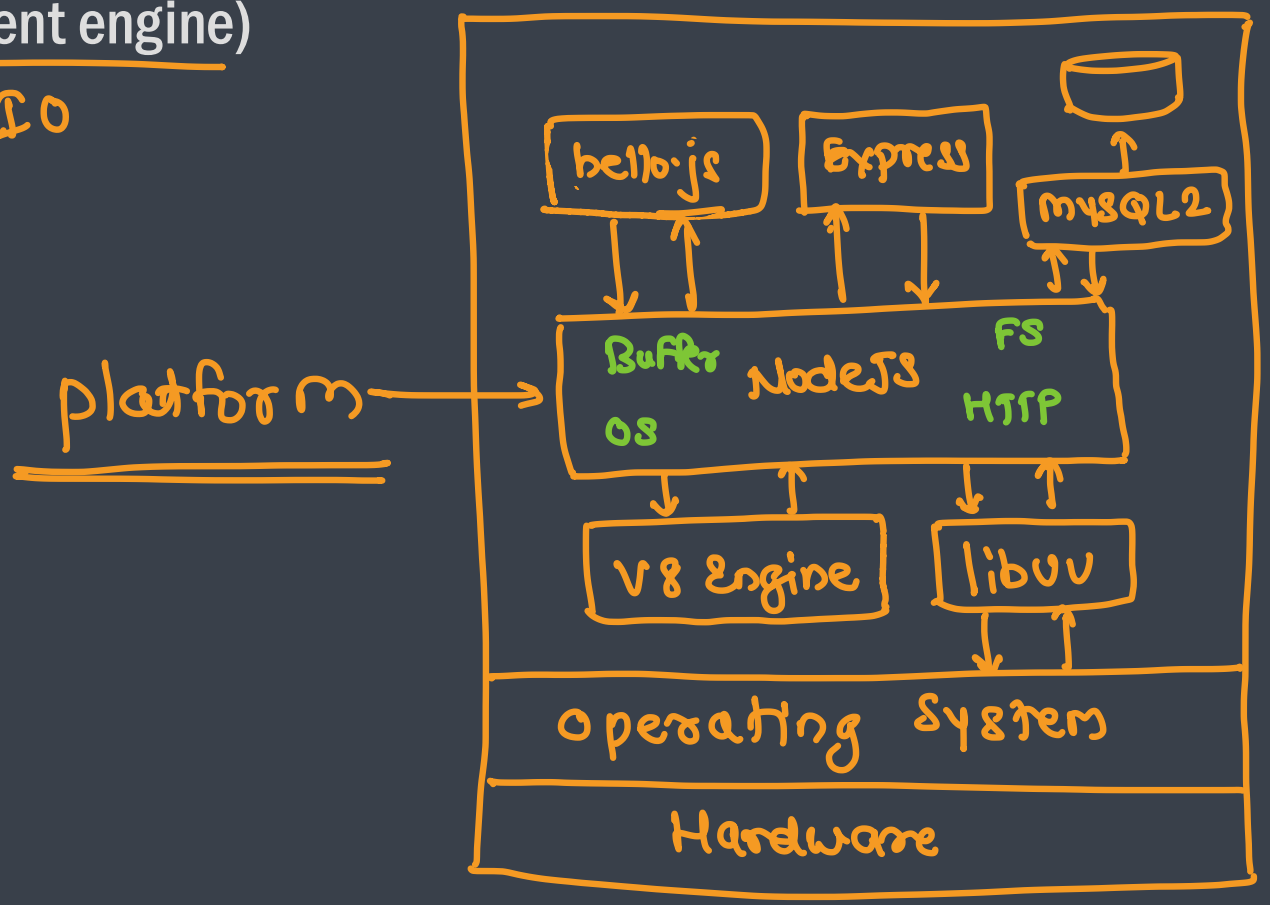
Node JS

What is Node ?

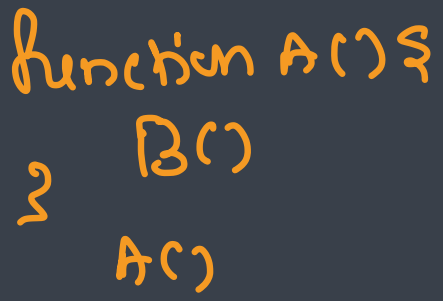
- Node.js is a JavaScript runtime built on Google's V8 JavaScript engine
  - The same V8 engine also powers the chrome
- It is a platform to develop server sided and networking applications in JS
- Developed Ryan Dahl in 2009
- It is free and open source
- It can run on various Operating Systems like macOS, Linux and Windows

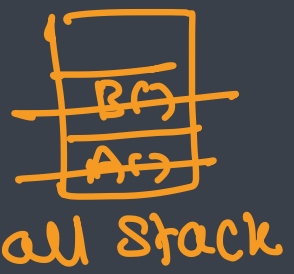
Node JS Components

- JavaScript Runtime: V8 (you can use different engine)
- Event Loop: Libuv → asynchronous IO
- Standard Library
  - Filesystem Access
  - Crypto
  - TCP/UDP
  - HTTP
  - Buffer
  - etc



Misconceptions

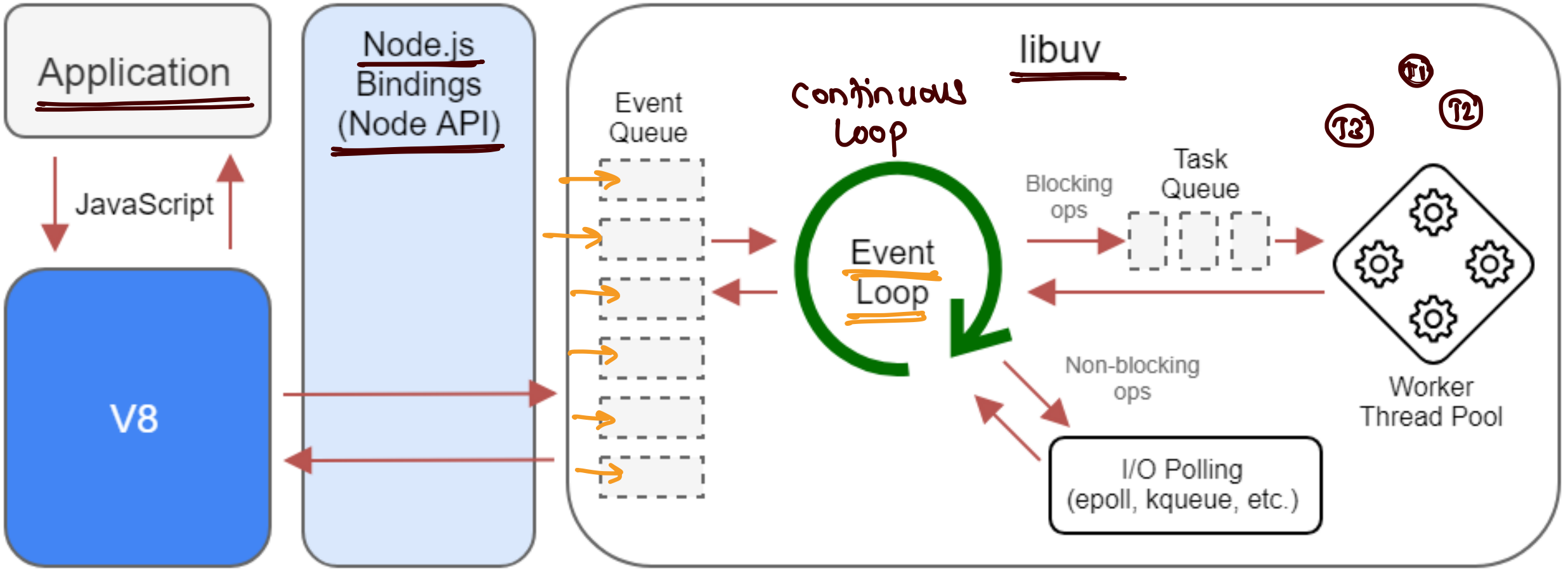




- Node JS is single threaded
  - Not all parts of NodeJs are single threaded
  - Event Loop and V8 run in single thread. It has single main call stack.
  - Your code has single threaded. If you run every expensive application then you block event loop
- Node JS uses a thread pool for all IO
  - Threads need to block while we do the IO so Node does not use Thread Pool for IO
  - It uses thread pool
    - when there is no OS async API that allows it do the non-blocking API
    - For crypto work
    - For DNS lookup
- The Event Loop runs in one thread and V8 runs in separate thread

Architecture

Asynchronous IO



NodeJS Features

- It uses event-driven, non-blocking I/O model
- It has a single threaded but highly scalable architecture
- It is a lightweight and efficient
- It has its own package ecosystem which has thousands of packages for different purpose
- It never buffers the data. Rather it simply outputs the data in chunks → streaming

NodeJS Use Cases

- Where to use NodeJs
  - I/O bound applications
  - Data streaming applications
  - IoT applications
  - JSON API based applications → REST API
  - Single Page Applications → React
- Where not to use NodeJs
  - CPU intensive applications
  - Heavy server sided processing

Advantages of NodeJS