Part I

Node JS:

We write JS code to make browser interactive. We execute JS code In the browser.

Browsers like Chrome, Firefox, Edge, and Safari were built with **a JavaScript engine** — a special program that can understand and execute JS code.

Examples of JS engines:

- **V8** → used by Google Chrome and Node.js
- **SpiderMonkey** → used by Firefox
- **JavaScriptCore** → used by Safari

These engines do two main things:

1. **Parse** JS code → understand it
2. **Compile** it into machine code → run it efficiently

When you open a webpage:

1. The browser loads the HTML.
2. Then it reads the JavaScript code.
3. The **JS engine** runs that code — letting it interact with the page (DOM), handle clicks, show alerts, etc.

JavaScript in the browser has access to special objects like:

- window
- document
- fetch
- localStorage

These are **not part of the JavaScript language itself** — they are **provided by the browser**.

So why does JS also run outside the browser (like in Node.js)?

JavaScript had:

- No access to files on your computer (for security)
- No access to the network except through limited APIs (window,document,fetch,localStorage)
- No ability to talk to databases, servers, or system files
  Because the **browser sandbox** isolated it from your system

Then in Node, embed **V8** into a C++ program, and then built new APIs around it — not browser APIs, but **system APIs**.

Example:

- Browser JS → can access document, window, fetch
- Node.js JS → can access fs (file system), http, os, etc.

What Node.js adds on top of V8?
Node.js is not just V8 — it's a **runtime environment** made of:

Components:

1. **V8 Engine** – Executes your JS code.
2. **libuv** – Handles asynchronous I/O (files, network, timers).
3. **Node APIs** – Built-in modules like fs, path, http.
4. **Bindings** – C++ code that connects JS functions to your operating system.

Code:

```
const fs = require('fs');
fs.writeFileSync('hello.txt', 'Hi there!');
```

What happens is:

1. JS code calls Node's fs module.
2. Node translates that to system-level operations using **libuv**.
3. The OS writes to your disk.
4. V8 runs everything as JS code.

Browser JS vs Node.js JS — Key Differences

Feature	**Browser JS**	**Node.js**
Engine	V8, SpiderMonkey, etc.	V8
Access to DOM	✅ Yes (document, window)	❌ No
Access to File System	❌ No	✅ Yes (fs)
Main Purpose	Frontend interactivity	Backend / Server-side JS
APIs	Web APIs	Node APIs
Security Sandbox	Yes	No (full system access)

Because of Node.js, JavaScript:

- Runs servers (e.g., Express.js)
- Builds command-line tools (e.g., npm)
- Runs desktop apps (e.g., Electron — VS Code is built on it)
- Even runs in IoT devices

Why specifically V8?

Reason 1: **Speed**
V8 compiles JavaScript **directly into machine code** instead of interpreting it line by line.

- It uses **JIT (Just-In-Time)** compilation.
- It has **Turbofan** (compiler) and **Ignition** (interpreter) that work together.
- The result: JavaScript runs **blazingly fast**, almost like C++.
- That made Node.js extremely efficient for backend operations — like handling thousands of network requests per second.

Reason 2**: V**8 powers **Google Chrome**, one of the world's most-used browsers.

- It's **constantly maintained and optimized** by Google engineers.
- It's **stable** and **secure**.
- It's already proven to run massive, real-world JavaScript workloads efficiently.

Reason 3: **Open-source & portable: V8 is open source, written in C++, and easy to embed in other C++ programs.**

Reason 4**: Garbage Collection: V8 has a highly optimized garbage collector that automatically frees unused memory.**

Reason **5**: Compatibility:
Because V8 strictly follows ECMAScript (JavaScript standard), Node.js automatically supports:
- **The latest JavaScript syntax (async/await, Promise, ES Modules, etc.)**
- **The same JS features you use in browsers**

Options like NodeJS:

Runtime	***Engine***	***Language Used***	***Key Feature***	***Best For***
Node.js	V8	C++	Huge ecosystem, stable	Backend servers, APIs
Deno	V8	Rust	Secure, modern, TypeScript native	Modern web apps
Bun	JavaScriptCore	Zig	Very fast, integrated toolchain	Startups, dev tools
GraalJS	GraalVM	Java	Polyglot, JVM integration	Enterprise systems
ChakraCore	Chakra	C++	Windows support (legacy)	Specialized/legacy use

We use NodeJS bcz it has more Ecosystem, it is proven, fast, large community

What is NPM?
npm stands for **Node Package Manager**. It gets installed along with NodeJS. npm helps you install, share, and manage code packages (modules) for JavaScript projects.

Why npm exists?
When developers build applications, they often need **reusable code** — for things like:
- Making HTTP requests
- Connecting to databases
- Validating forms
- Logging
- Authentication

Instead of writing that logic from scratch every time, you can just **install a pre-built package** from npm — written and published by other developers.

npm makes it easy to:
- Install existing libraries
- Update them
- Manage project dependencies
-

Npm has 2 parts:
1. npm CLI (Command-Line Interface): The tool you use in your terminal (comes with Node.js)
2. npm Registry: The online database that stores all public packages

What is npm init?
Purpose:
npm init is used to **create a new package.json file** in your project.
The package.json file is the **heart of every Node.js project** — it tells npm (and other developers) everything about your project:

- The project's name, version, and author
- Which dependencies (packages) it needs
- What scripts it can run
- Optional metadata like license, repository, etc.

Why it's important?
npm init is usually the first command you run when starting a new Node.js project because:

- npm uses package.json to track installed dependencies
- It stores metadata for package publishing
- Tools like express, react, vite, etc., rely on it
- It lets you define scripts (like npm start, npm run dev, etc.)

Without package.json, npm doesn't know what your project is or what it needs.

Modules:
Modular Programming is a software design principle where you **divide a large program into smaller, independent, and reusable parts**, called **modules**.

Each module:

- Has a **specific job or responsibility**
- Can be **developed, tested, and debugged** separately
- Can be **reused** in other programs

In **Node.js**, each file is treated as a module.
You can:
Export things from one file
Import them into another file
In **modern JavaScript (ES6 and above)**, you can use **export** and **import** to transfer data, functions, classes, or even entire modules between files.

require → old Node.js style
import/export → modern ES6 (ECMAScript) style

when we do "import x from "./something" we try to find something file from local directory
when we do "import x from "something" we try to find something from it's own installed packages

key Concept: Once Node or the browser loads a module:

- *It executes it **only once** per runtime.*
- *Then caches it.*
- *So even if 10 files import the same module, it won't run 10 times — they all share the same instance.*

When we export something it stores like:
```
exports = {
  add: [Function: add],
  PI: 3.14
};
```
When export with default:
```
exports = {
  default: [Function: add],
  PI: 3.14
};
```

When we import the function or variable, we import the reference of the function/variable
When the first line (import line) runs, it see the import file and go there and find the referenced function/variable and execute it

File Handling

File handling means reading from, writing to, updating, deleting, and managing files (like .txt, .json, .csv, etc.) from your Node.js program.
Basically — it's how your Node.js app **interacts with the file system** on your computer or server.
In the browser, JavaScript can't access your computer's files directly (for security reasons).
But Node.js runs on the **server (or your local machine)**, where it *can* access the filesystem — just like other languages (C, Python, Java, etc.).

Node provides a built-in module called **fs (File System)**.
Node.js includes the fs module for all file operations.
fs allows both **synchronous** and **asynchronous** (non-blocking) operations.

Common file operations:

Reading a File:

Asynchronous (non-blocking):
```
import fs from "fs";

fs.readFile("data.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error reading file:", err);
    return;
  }
  console.log("File content:", data);
});
```

Synchronous (blocking):
```
import fs from "fs";

const data = fs.readFileSync("data.txt", "utf8");
console.log("File content:", data);
```

Similarly,
Writing to a file

Delete a file (unlink)
Appending data to a file
Renaming a file
Copy file
Statistics of a file

You can also handle directories using fs.
Ex:
Create a folder:

```
fs.mkdir("newFolder", (err) => {
  if (err) throw err;
  console.log("Folder created!");
});
```

Read all files in a folder:

```
fs.readdir("./", (err, files) => {
  if (err) throw err;
  console.log("Files:", files);
});
```

Delete a folder:

```
fs.rmdir("newFolder", (err) => {
  if (err) throw err;
  console.log("Folder deleted!");
});
```

Step	**Synchronous**	**Asynchronous**
Waits for file?	Yes	No
Blocks other code?	Yes	No
Speed	Slower for large I/O	Faster overall
Thread behavior	Uses main thread	Uses worker thread in background
Example	fs.readFileSync()	fs.readFile()

What does the **industry use**?
Almost always asynchronous (non-blocking) file operations.
Node.js uses a **single-threaded event loop**.
If one synchronous operation blocks the thread, your **entire app freezes** — no requests, no events, nothing.
Asynchronous I/O allows Node.js to handle **many clients at once** efficiently.

Uses of file system:

1. Logging and Auditing
   Used in: Backend servers, APIs, authentication systems, financial software
   Purpose: To record important events (like logins, errors, transactions)
2. File Uploads and Media Handling
   Used in: Chat apps, CMS (e.g., WordPress clones), social media platforms
   Purpose: To store uploaded files (like images, videos, PDFs) temporarily or permanently.

3. Configuration and Template Management
Used in: Deployment systems, templating engines, report generators
Purpose: Read configuration files or templates dynamically
4. Static File Serving
Used in: Web servers, custom CDN layers
Purpose: To serve HTML, CSS, JS, and media files directly from disk.

5. Backup and Archiving
Used in: Enterprise systems, data pipelines, cloud sync tools
Purpose: Move/copy files for backup or archival.
6. Monitoring File Changes
Used in: Development servers, CI tools, file-sync systems
Purpose: Automatically reload or act when files change.

Node JS architecture

1. User send request to Server created in NodeJ js.
2. There is an **event queue** in node js
3. The **request is queued** inside the event queue
4. The requests go to event loop, event loop always look onto events. Use FIFO from queue
5. The requests could be synchronous or asynchronous. Event loop checks it.
6. If synchronous, it will process it and return the response
7. If asynchronous, the request goes to thread pool.
8. Thread pool has many threads, thread can be understood as a worker to make the request work. If there is a thread available it will work. Bcz it has limited threads
9. After work completion, the thread comeback to thread pool and return a response
10. If the workers/thread are busy, then the request have to wait until some thread get executed. It is a issue with scalability
11. It’s a good practice to do non-blocking executions

Node.js is single-threaded for JS execution,
but **uses a thread pool (libuv)** to handle async I/O tasks concurrently.
That’s why:
- It does not block the main thread.
- Responses come back as soon as each async task completes.
- The event loop picks up completed callbacks one by one in the order they’re done.

Http:

HTTP (HyperText Transfer Protocol) is a **communication protocol** that allows a **client** (usually a browser or app) to communicate with a **server** over the internet.
It defines:
how a client makes a **request**
how the server sends a **response**
how data is formatted, transferred, cached, secured, etc.

Why HTTP Exists?
The internet connects billions of devices, but they need a **standard way** to talk.

HTTP solves:

- How to ask for data
- How to send data
- How to structure URLs
- How to maintain sessions
- How to cache
- How to secure communication
- How to define verbs like GET, POST, etc.

Can we use the internet without HTTP?
YES — it's technically possible.

Then why do we STILL use HTTP everywhere?
Because **without HTTP the web becomes unusable**, inconsistent, insecure, incompatible, and chaotic.

Without HTTP:
Every app would invent its own communication rules
Your browser wouldn't know how to talk to any website
Servers wouldn't understand how to send data back

Browsers (Chrome, Firefox, Safari, Edge) are built around HTTP/HTTPS.
If a server doesn't speak HTTP:
The browser cannot render pages
You cannot load images, CSS, JS
You cannot submit forms
You cannot visit URLs

Browsers understand **one main protocol** → HTTP.

Without HTTP, we lose the Request–Response model
Web apps depend on this model:

- Client → sends request
- Server → sends response

Without HTTP:
You must design your own:
message structure
error formats
query system
method types
body formats
header formats

This adds huge complexity.
HTTP solves this by giving a **prebuilt communication framework**.

Could we build websites using a different protocol?

Yes.
We could use:

- WebSockets
- gRPC
- FTP
- Raw TCP
- QUIC
- Custom binary protocols

But these **do NOT replace HTTP**

Http is stateless

Stateless means:
Each HTTP request is independent.
Server does NOT remember anything about previous requests.

The server does **NOT automatically know** that both requests came from:

- the same person
- the same session
- the same browser

Unless you provide some identifier (cookie/session/token), the server treats EVERY request as brand new

Why is HTTP stateless?
Because it makes HTTP:

- Simple
- Fast
- Scalable
- Distributed
- Light on memory

If HTTP required servers to "remember" every user, the server would:

- Use huge memory
- Crash easily
- Scale poorly with millions of users
- Perform slower

If HTTP is stateless, how does it remember users?

we use **additional mechanisms on top of HTTP** to store state.

1. Cookies:
   Server says: Set-Cookie: sessionId=abc123
   Browser automatically sends this cookie back in every request: Cookie: sessionId=abc123
2. Sessions (stored on server)
   Server stores data: sessionId => { userId: 20, name: "Bibek" }
   Browser stores only the sessionId in cookies.
3. JWT Tokens
   Token sent in headers: Authorization: Bearer <jwt-token>
   Server reads token → decodes details → identifies the user.
4. LocalStorage + Cookies (client-side memory)
   Browsers store:

- tokens

- preferences
- cart items
- theme settings
  But HTTP itself stays stateless.

What is TCP?

TCP (Transmission Control Protocol) is a communication protocol that ensures data is sent between two computers *reliably*, *correctly*, and *in the right order* over the internet.

It is one of the core protocols of the internet and works together with IP (Internet Protocol).
That's why we say **TCP/IP**.

Why TCP Exists?
When you send data:

- Packets can get lost
- Packets can arrive in the wrong order
- Packets can get duplicated
- Packets can be delayed
- Packets can get corrupted

TCP fixes all these problems.

What TCP actually does?

1. **Breaks data into packets**
   When you send a message: `"Hello World"`, TCP breaks it into small chunks (packets) to send efficiently.
2. **Ensures all packets are delivered (Reliability)**
   If any packet is lost in the network:
   TCP notices
   TCP resends it
   Client receives 100% data
3. Ensures packets arrive in order (Sequencing)
4. Ensures packets are not duplicated
5. Checks for errors (Error detection)
6. Controls speed (Flow control)
7. Creates a reliable connection: Before sending data, TCP performs a connection handshake (3-way handshake).

What is HTTPS? (HyperText Transfer Protocol Secure)
HTTPS = HTTP + SSL/TLS encryption

HTTPS is the **secure version of HTTP**.
It still uses the **same request/response model**.

All communication in HTTPS is encrypted using TLS (Transport Layer Security)
So the main job of HTTPS is to:

1. **Encrypt data** (no one can read what you send)

2. **Verify identity** (you know you're talking to the real server)
3. **Prevent tampering** (data cannot be modified)

Without HTTPS
Anyone between client ↔ server can:

- Read your data
- Steal passwords
- Manipulate API responses
- Inject malware
- Impersonate a server

HTTPS fixes all these.

How HTTPS Works:
HTTPS uses **TLS Handshake**, which has two main steps:
Step 1: Client says "Hello"
Step 2: Server sends its Identity (Certificate)
Step 3: Client encrypts a secret key
Step 4: Both now share the same secret key
Every request & response after handshake is fully encrypted.

What HTTPS Protects Against

- Data Theft
- Man-in-the-middle Attacks
- Data Tampering
- Session Hijacking

How HTTPS Is Different from HTTP (Deep Comparison)

Feature	**HTTP**	**HTTPS**
Encryption	No	✓ Yes (TLS)
Security	Vulnerable	✓ Safe
MITM protection	No	✓ Yes
Authentication	None	✓ Server identity verified with certificate
SEO Ranking	Lower	✓ Google gives SEO boost
Speed	Faster in very old days	Now almost equal (TLS optimized)
Port	80	443
Certificate	Not needed	✓ Required (CA issues cert)

Why HTTPS Needs Certificates?
A **Certificate Authority (CA)** signs the certificate with its private key.
Browsers trust CAs, so:
→ If CA says the certificate is valid
→ Browser trusts the website
This prevents fake servers from impersonating real ones.

Why do we import http from "http"?

Because **Node.js does NOT automatically create servers**.

Node gives you a built-in module called **http**, which contains:

- Functions to create an HTTP server
- Classes for handling HTTP requests & responses
- Support for TCP communication under the hood

import http from "http";
you are importing the **core HTTP module** provided by Node.

What exactly does the http module do?
It gives you function: http.createServer()

This function:
 Creates a **TCP server**
 Adds **HTTP protocol logic** on top of TCP
 Handles:

- parsing HTTP requests
- handling headers
- handling methods (GET/POST/PUT/DELETE)
- sending responses
- status codes

It is a built-in Node.js module — **not a package from npm**.

why do we usually import http but not https?
Because **HTTPS requires certificates**.

HTTPS cannot run without SSL certificates
If you import:
import https from "https";
you MUST provide:

- key (private key)
- cert (SSL certificate)

In development, HTTP is fine
In production, HTTPS is handled by reverse proxy (Nginx, Cloudflare)

How to create server In Node?
Code:

```
Import http from (“http”)
Const myServer=http.createServer((req,res)=>{
  res.end(“Hello from Server”)
})
```

When run in browser with localhost:8000, it will give response as “Hello from server” and the connection is ended. In pure node js server res.end() is mandatory. But in express, express itself manages it internally.

Request Handler function:
(req,res)=>{}   => it is a callback function that handle our request and response
We need a port to run the server

On the port of the server we listen to the request
1 server must be run with 1 port only. One to One

myServer.listen(8000,()=>{console.log("Server Started")})   (PORT=8000)

if everything is in right place, the server will run and this console will work

Handling URL in Node

A **URL (Uniform Resource Locator)** is the **address** you type in a browser to access a resource on the internet.
It tells the browser **where the resource is located** and **how to fetch it.**

Parts of a URL:

1. Protocol
   Tells the browser *how* to communicate.
   Common ones:
   - **http** – HyperText Transfer Protocol
   - **https** – secured Http
   - **ftp** – File Transfer Protocol

2. Domain Name
   Ex: www.google.com
   Human-friendly name that maps to an IP address.
3. Path
   Ex: /search
   Which resource or page you want on the server.
4. **Query Parameters** (optional)
   Ex: ?q=chatgpt
   Extra data you send to the server.
   In query parameter always key-value pairs passed?
   Yes — query parameters are ALWAYS key–value pairs.
   But there are some *special cases* where the **value can be empty** or the **key can appear without a value**.
   Still, the structure is based on **key = value**.
5. **Fragment** (optional)
   Ex: #section2
   Used to jump to a specific section inside a webpage.
6. **TLD (Top-Level Domain):**
   It is the **last part of a domain name**, which comes **after the final dot**.

What is a IP address?
An **IP address** is the **numerical address** of a device or server on a network.

It identifies **where** a device is located on the internet.

Ex: 142.250.190.78

In simple terms:

- IP address = **Home address** of a device
- Every device/server on the internet has an IP address
- Browsers need IPs to connect to websites

Aspect	**IP Address**	**URL**
What it is	Numerical address	Human-friendly web address
Purpose	Identifies a machine/device	Points to a resource/page
Example	142.250.190.78	https://google.com
Used by	Network systems	Humans and browsers
Easy to remember?	✗ No	✓ Yes
Requires DNS?	✗ No	✓ Yes (to translate URL → IP)

How URL and IP work together?
When you type a URL, DNS translates it into an IP address. Then your browser connects to the IP to fetch the website.

Http Methods:
GET: fetch some data from server (by default)
POST: send or mutate data in the server
PUT: replace this entire object with what I'm sending
PATCH: modify only these fields
DELETE: delete the object/data from DB

Continue in PART 2…………………………………..