1. What is Node.js?

Node.js is a JavaScript runtime environment built on Chrome's V8 engine that allows us to run JavaScript on the server using a non-blocking, event-driven architecture.

Before Node.js, JavaScript worked **only inside browsers**
After Node.js, JavaScript can run on **servers**, **APIs**, **databases**, **files**, etc.

Node.js is made of **5 main parts**:

- JavaScript Code (Your App)
- V8 Engine
- Event Loop
- Libuv (C++ Library)
- Operating System (OS)

Blocking vs non-blocking

```
const data = fs.readFileSync("file.txt");
console.log(data);
```

- Thread waits
- Server stuck
- Other users wait

Non-Blocking (Node's Superpower)

```
fs.readFile("file.txt", (err, data) => {
  console.log(data);
});
```

- Task goes to background
- Thread is free
- Other users are served

Event Loop (Heart of Node.js)

The **Event Loop** decides:

- What runs now
- What waits
- What runs later

Event Loop phases (simplified)

1. Timers (setTimeout)
2. I/O callbacks
3. Poll phase
4. Check phase (setImmediate)
5. Close callbacks

Event loop keeps checking:

“Any task finished? Okay, execute callback.”

Libuv (The Unsung Hero)

Libuv is a **C++ library** that handles:

- File system
- Network calls
- DNS
- Threads (internally)

It creates a **thread pool** for:

- File I/O
- Crypto
- Compression

So even though Node.js is single-threaded:

Heavy tasks are offloaded internally

2. Why is Node Js single-threaded?

Node.js is single-threaded because JavaScript itself is single-threaded. Instead of using multiple threads, Node.js uses an event-driven, non-blocking I/O model where one thread and an event loop efficiently handle thousands of concurrent requests, avoiding the complexity and overhead of multi-threading.

So:

- One call stack
- One event loop
- One main JS thread

Node.js model:

One Thread
↓
Event Loop
↓
Async Tasks → Background
↓
Callbacks handled when ready

- No waiting
- No blocking
- High throughput

3. What is a Node framework?

A Node.js framework is a set of tools, rules, and pre-written code built on top of Node.js that helps us create backend applications faster, cleaner, and in a structured way.

Why frameworks exist (core reason)

Node.js by itself gives you:

- File system access
- Network access
- OS access

But Node.js **does NOT** give:

- Routing
- Controllers
- Middleware system
- Project structure
- Request handling helpers

Popular Node.js Frameworks (important)

1. **Express.js (Most popular)**

Express.js is a minimal, flexible, and unopinionated Node.js framework used to build REST APIs and web applications.

It gives you full control over project structure.

Key Features :-

- Minimal & lightweight
- Unopinionated (no fixed structure)
- Middleware-based
- Large ecosystem
- Easy to learn

```
const express = require("express");
const app = express();

app.get("/users", (req, res) => {
  res.send("Users");
});

app.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

Advantages

- Simple syntax
- Huge community
- Tons of middleware

Disadvantages

- No enforced structure
- Can become messy in large apps

2. NestJS?

NestJS is a **highly structured, scalable Node.js framework** inspired by **Java Spring Boot**.

It uses **TypeScript, decorators, and Dependency Injection** by default.

Key Features

- Built on Express / Fastify
- Strong architecture
- Dependency Injection
- Modular structure
- TypeScript-first

When to Use NestJS?

- Large-scale applications
- Enterprise systems
- Team-based projects
- Long-term maintainability

```
import { Controller, Get } from "@nestjs/common";

@Controller("users")
export class UserController {
  @Get()
  getUsers() {
    return "Users";
  }
}
```

Advantages

- Clean architecture
- Easy testing
- Scales very well
- Best practices enforced

Disadvantages

- Steeper learning curve
- Slightly verbose

3. Fastify (High Performance)

Fastify is a **high-performance Node.js framework** designed for **speed and low overhead**.

Focuses heavily on **performance and JSON schema validation**.

Key Features

- Extremely fast
- Schema-based validation
- Low memory usage
- Plugin system
- Async/await friendly

When to Use Fastify?

- High-performance APIs
- Microservices
- Real-time systems
- Performance-critical apps

```
const fastify = require("fastify")();

fastify.get("/users", async (request, reply) => {
  return { users: [] };
});

fastify.listen({ port: 3000 });
```

Advantages

- Very fast
- Built-in validation
- Better performance than Express

Disadvantages

- Smaller ecosystem than Express
- Slight learning curve

Feature	Express.js	NestJS	Fastify
Type	Minimal	Enterprise	Performance
Structure	Unopinionated	Highly structured	Plugin-based
TypeScript	Optional	Default	Optional
Dependency Injection	❌	✅	❌
Performance	Medium	Medium	Very High
Learning Curve	Easy	Medium	Medium

4. Node.js Process Model

Node.js scales horizontally using multiple processes rather than vertically using multiple threads.

What is a “process” in Node.js?

A running instance of a program with its own memory, event loop, and resources.

This process has:

- Its own memory
- One main JS thread
- One event loop

Inside a Node.js Process (Important)

A single Node.js process contains:

Node Process

├─ Call Stack (JS execution)

├─ Event Loop

├─ Heap Memory

├─ Libuv Thread Pool

└─ OS Resources

Limitations of single-process model

Even though Node is efficient, this model has limits:

1. **Uses only one CPU core**
   - Modern CPUs have 4–16 cores
   - Single process uses **only one**
2. **Crash = app down**
   - If process crashes → server down
3. **CPU-heavy tasks block app**
   - Infinite loop blocks everything

Cluster Module (Built-in scaling)

The **cluster module** allows:

- Create **multiple worker processes**
- Use **all CPU cores**
- Run workers on the **same server port**
- Achieve **load balancing**

Core Properties & Methods

Function / Property	Definition
cluster.isPrimary	Returns true if the current process is the **primary (master)** process.
cluster.isWorker	Returns true if the current process is a **worker** process.
cluster.fork()	Creates a **new worker process** that runs the same Node.js application.
cluster.workers	An object that contains **all active worker processes** indexed by their IDs.
worker.id	A **unique numeric ID** assigned to each worker process.
worker.process	Provides access to the **child process object** of the worker (PID, signals, etc.).
worker.kill()	**Terminates** the worker process gracefully or forcefully.

Cluster Events

Event	Definition
cluster.on("fork")	Triggered when a **new worker is created** using cluster.fork().
cluster.on("online")	Triggered when a worker is **running and ready to receive requests**.
cluster.on("exit")	Triggered when a worker **exits or crashes**, optionally used to restart it.

Worker Threads vs Processes (Quick clarity)

Feature	Process	Worker Thread
Memory	Separate	Shared
Crash safety	High	Lower
Performance	Lower	Higher
Complexity	Low	High

5. Difference between single thread and multi thread

Feature	Single Thread	Multi Thread
Threads	One	Multiple
Execution	Sequential	Parallel
Complexity	Simple	Complex
Debugging	Easy	Difficult
Memory usage	Low	High
CPU usage	Low	High
Risk	Low	High (deadlock, race)

Node.js is:

- **Single-threaded at JS level**
- **Multi-threaded internally** (libuv thread pool)

“Node.js has a single JS thread but uses background threads internally.”

Use Single-Threaded when:

- I/O-heavy apps
- APIs
- Real-time apps
- Chat systems

Use Multi-Threaded when:

- CPU-heavy processing
- Image/video processing
- Scientific calculations

6. Node.js Console - REPL

What is REPL?

REPL stands for **:- Read – Evaluate – Print – Loop**

Node.js REPL is an interactive shell that allows developers to execute JavaScript code line by line. It follows the Read–Evaluate–Print–Loop pattern and is mainly used for testing, learning, and debugging Node.js features.

It is an **interactive Node.js console** where you can:

- Write JavaScript code
- Execute it immediately
- See the result instantly

No file
No server
No browser

Using REPL for learning Node.js

REPL is perfect for:

- Testing small logic
- Understanding APIs
- Learning async behavior
- Debugging quickly

REPL special commands

Command	Meaning
.help	Show all commands
.exit	Exit REPL
.clear	Clear context
.load file.js	Load a JS file
.save file.js	Save REPL session

_ (underscore) in REPL

```
> 10 + 5
15
> _ * 2
30
```

Multiline code in REPL

```
> function greet(name) {
... return "Hello " + name
... }
> greet("Node")
"Hello Node"
```

REPL vs Script file

REPL	JS File
Interactive	Static
Immediate output	Needs run
Testing & learning	Production code
No file needed	File required

7. Default Node.js Module (Node.js Built-in Modules)

Node.js built-in modules are core modules that come pre-installed with Node.js and provide essential functionalities like file system access, networking, OS interaction, and process management without requiring external packages.

- No installation needed
- No npm install
- Directly available after installing Node.js

Why Node.js has Built-in Modules?

Because JavaScript alone **cannot**:

- Read/write files
- Create servers
- Access OS info
- Handle processes

Node.js provides these abilities using **core modules written in C++ and JavaScript**.

Built-in Modules vs NPM Packages

Built-in Modules	NPM Packages
Pre-installed	Need installation
Core features	Extra features
Fast & stable	Community-driven
No version issues	Version management needed

1. **fs Module (File System)**

The **File System (fs) module** is a core Node.js module that allows applications to **interact with the operating system's file system**.

It enables backend applications to **create, read, update, delete files and directories**.

Why the fs Module Is Important

Backend applications frequently need to:

- Store application logs
- Upload and manage files
- Read configuration files
- Generate reports (PDF, CSV, etc.)
- Serve static content

Without file system access, a backend application is incomplete.

Ways to Use the fs Module

The fs module supports **two execution modes**:

1. **Synchronous (Blocking)**
2. **Asynchronous (Non-blocking)**

Node.js applications should **prefer asynchronous methods** to keep the event loop free.

Why it exists

JavaScript cannot access files by default.
The fs module provides **controlled file system access** through Node.js.

Types of fs APIs

Type	Description
Asynchronous	Non-blocking, recommended for production
Synchronous	Blocking, not recommended for large apps

Synchronous fs Methods (Blocking)

- **Reading a File (Sync)**

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

❌ Blocks the event loop
❌ Not recommended for servers

- **Writing a File (Sync)**

```
fs.writeFileSync("file.txt", "Hello Node");
```

Asynchronous fs Methods (Non-Blocking)

- **Reading a File (Async)**

```
fs.readFile("data.txt", "utf8", (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

✅ Non-blocking
✅ Suitable for production servers

- **Writing a File (Async)**

```
fs.writeFile("file.txt", "Hello Node", (err) => {
  if (err) throw err;
  console.log("File written");
});
```

Promise-Based fs (Modern Best Practice)

Node.js provides promise-based APIs for cleaner asynchronous code.

```
const fs = require("fs/promises");

async function readFile() {
  const data = await fs.readFile("data.txt", "utf8");
  console.log(data);
}

readFile();
```

Advantages:

- Clean syntax
- Async/Await support
- Recommended for modern Node.js apps

Common fs Operations (CRUD)

1. **Create a File**

```
fs.writeFile("new.txt", "Content", () => {});
```

2. **Read a File**

```
fs.readFile("new.txt", "utf8", () => {});
```

3. **Update a File**

```
fs.appendFile("new.txt", "\nMore content", () => {});
```

4. Delete a File

```
fs.unlink("new.txt", () => {});
```

Working with Directories (Folders)

1. Create a Folder

```
fs.mkdir("uploads", { recursive: true }, () => {});
```

2. Read Folder Contents

```
fs.readdir("uploads", (err, files) => {
  console.log(files);
});
```

3. Delete a Folder

```
fs.rmdir("uploads", { recursive: true }, () => {});
```

File Streams (Handling Large Files)

Streams process data **chunk by chunk**, saving memory.

```
const readStream = fs.createReadStream("bigfile.txt");

readStream.on("data", (chunk) => {
  console.log(chunk);
});
```

Use Cases:

- Video streaming
- Large file uploads/downloads
- Log file processing

Using fs with path Module (Best Practice)

```
const path = require("path");

const filePath = path.join(__dirname, "files", "data.txt");
```

✓ Prevents OS-specific path issues
❌ Never hardcode file paths

Real-World MERN Stack Usage

- Profile image uploads
- Log file generation
- PDF and report exports
- File-based backups

2. path Module

The **path module** provides utilities to **work with file and directory paths** in a **cross-platform safe way**.

Why it exists

Different OS use different path formats (\ in Windows, / in Linux/macOS).
The path module **prevents path-related bugs** by handling this automatically.

Commonly Used Path Methods

Method	Purpose
path.join()	Safely joins path segments
path.resolve()	Returns absolute path
path.basename()	Gets file name
path.dirname()	Gets directory name
path.extname()	Gets file extension
__dirname	Current file directory

Simple Examples :

```
path.join(__dirname, "uploads", "file.txt");
path.resolve("uploads", "file.txt");
path.basename("/users/file.txt"); // file.txt
path.extname("image.png"); // .png
```

Real-World Usage

- File uploads
- Static file serving
- Config & env files
- Works with fs module

Example :

```
app.use(express.static(path.join(__dirname, "public")));
```

Key Points

- Cross-platform
- Prevents path errors
- Used in almost every backend project

3. os Module (Operating System)

The **os module** provides information about the **operating system, hardware, and system resources** on which a Node.js application is running.

Why it exists

JavaScript cannot access system-level details like CPU, memory, or OS type.
The os module safely exposes this information to Node.js applications.

Important os Methods

Method	Purpose
os.platform()	Returns OS name (win32, linux, darwin)
os.arch()	Returns CPU architecture
os.cpus()	Returns CPU core information
os.totalmem()	Returns total system memory
os.freemem()	Returns free system memory
os.hostname()	Returns system hostname
os.uptime()	Returns system uptime

Simple Examples

```
os.platform();
os.arch();
os.cpus().length;
os.totalmem();
os.freemem();
```

Real-World Usage

- Scaling Node apps
- Cluster worker count
- Performance monitoring
- System logging

Key Points

- Provides system-level info
- Read-only and safe
- Used with cluster and PM2
- Important for performance tuning

4. http Module

The **http** module allows Node.js to create an HTTP server and handle client requests and responses.

"**Express** internally uses the Node.js http module to **handle** requests and responses, then adds routing and middleware on **top** of it."

Why the http module exists

JavaScript alone **cannot**:

- Listen on ports
- Accept HTTP requests
- Send HTTP responses

Node.js uses the http module to:
communicate over the **HTTP protocol**

Creating a basic HTTP server

```
const http = require("http");

const server = http.createServer((req, res) => {
  res.end("Hello from Node.js HTTP server");
});

server.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

What happens internally (important concept)

- Client sends request
- Node http server receives it
- Callback runs
- Response is sent back

Client → http server → response

Common http Module Methods

Method	Definition
http.createServer()	Creates a new HTTP server
server.listen()	Starts server on a given port
server.close()	Stops the server
http.get()	Sends an HTTP GET request
http.request()	Sends custom HTTP requests

req (Request) Object Properties

Property	Definition
req.url	Requested URL path
req.method	HTTP method (GET, POST, etc.)
req.headers	Request headers
req.socket	Underlying socket info
req.httpVersion	HTTP version used

res (Response) Object Methods

Method / Property	Definition
res.write()	Writes response body
res.end()	Ends response
res.setHeader()	Sets HTTP response header
res.getHeader()	Reads a response header
res.statusCode	Sets HTTP status code
res.writeHead()	Sets status + headers together

HTTP Status Codes (Quick Reference)

Category	Code	Meaning
Success (2xx)	200	OK
	201	Created
	202	Accepted
	204	No Content
	206	Partial Content
Redirection (3xx)	301	Moved Permanently
	302	Found (Temporary Redirect)
	304	Not Modified
	307	Temporary Redirect
	308	Permanent Redirect
Client Error (4xx)	400	Bad Request
	401	Unauthorized
	403	Forbidden

	404	Not Found
	405	Method Not Allowed
	408	Request Timeout
	409	Conflict
	410	Gone
	415	Unsupported Media Type
	422	Unprocessable Entity
	429	Too Many Requests
Server Error (5xx)	500	Internal Server Error
	502	Bad Gateway
	503	Service Unavailable
	504	Gateway Timeout
	507	Insufficient Storage

5. https Module (Secure HTTP)

The **https module** allows Node.js to create **secure HTTP servers** using **SSL/TLS encryption** to protect data between client and server.

Why it exists

Normal http sends data in plain text (insecure).
The https module ensures **encrypted, safe communication**, especially for passwords, tokens, and payments.

http vs https

http	https
Insecure	Secure
Plain text	Encrypted
Port 80	Port 443
Not for auth	Safe for auth

Basic HTTPS Server Example

```
const https = require("https");
const fs = require("fs");

const options = {
  key: fs.readFileSync("key.pem"),
  cert: fs.readFileSync("cert.pem"),
};

https.createServer(options, (req, res) => {
  res.end("Secure Server");
}).listen(443);
```

SSL/TLS (Simple Meaning)

- Encrypts data
- Prevents data theft
- Verifies server identity

Real-World Usage

- Secure APIs
- Authentication systems
- Payment services
  (Usually handled by Nginx or cloud providers in MERN apps)

6. process Module (Node.js Built-in)

The **process module** provides information and control over the **currently running Node.js process**.

Why it exists

It allows Node.js applications to:

- Access environment variables
- Read command-line arguments
- Know process ID and OS
- Control app lifecycle (exit, shutdown)

Global Module

process is **global**, no import required.

Important process Properties

Property	Purpose
process.env	Access environment variables
process.argv	Command-line arguments
process.pid	Process ID
process.platform	OS type
process.cwd()	Current working directory

Simple Examples

```
process.env.NODE_ENV;
process.argv;
process.pid;
process.platform;
process.cwd();
```

Important process Methods

Method	Purpose
process.exit()	Terminates the process
process.on()	Listens to process events

```
process.on("exit", code => {
  console.log("Exit code:", code);
});
```

Real-World Usage

- .env configuration
- Docker & CI/CD
- NestJS config modules
- Graceful shutdown
- Production environment checks

Key Points

- Core Node.js module
- Used in every backend app
- Essential for production
- Handles runtime behavior

One-Liner Definition

The process module provides access to environment variables, command-line arguments, and control over the Node.js runtime process.

7. events Module (Node.js Built-in)

The **events module** allows Node.js to work with **Event-Driven Architecture**, reacting to events asynchronously.

Why it exists

Node.js is non-blocking and asynchronous. Events let Node **notify when something happens** instead of waiting.

Examples of events:

- HTTP request arrives
- File read completes
- Database query finishes

Import

```
const EventEmitter = require("events");
```

Core Concept : EventEmitter

- EventEmitter is a **class**
- Can **emit** events and **listen** to events

```
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();
```

Listening and Emitting Events

```
myEmitter.on("orderPlaced", () => {
  console.log("Order received");
});

myEmitter.emit("orderPlaced");
```

Passing data:

```
myEmitter.on("login", user => console.log(user));
myEmitter.emit("login", { id: 1, name: "Kevin" });
```

Other Features

- Multiple listeners per event → all execute
- on() → runs every time
- once() → runs only once

```
myEmitter.once("connect", () => console.log("Connected once"));
```

Real-World Usage

- Node core: http, fs, stream, process
- Express: request events & middleware chain
- NestJS: event-based microservices, @nestjs/event-emitter

```
req.on("data", chunk => {});
req.on("end", () => {});
```

Key Points

- Heart of Node.js architecture
- Event-driven, non-blocking
- Core for understanding NestJS

8. stream Module (Node.js Built-in)

The **stream module** provides an efficient way to **read and write data in chunks** instead of loading everything into memory at once.

- Ideal for **big files**
- Reduces memory usage
- Improves performance

Why it exists

Problem ❌:

```
const data = fs.readFileSync("bigfile.mp4");
```

- Loads entire file into memory
- Crashes on large files

Solution ✅:

- Read file in **chunks**
- Process each chunk as it arrives
- Memory efficient

Types of streams

1. **Readable** – read data
2. **Writable** – write data
3. **Duplex** – read + write
4. **Transform** – modify data while reading/writing

Import

```
const fs = require("fs"); // stream used in fs
```

Streams are built into **fs, http, process**

Example: Read stream

```
const readStream = fs.createReadStream("bigfile.txt", "utf8");

readStream.on("data", (chunk) => {
  console.log("Received chunk:", chunk);
});

readStream.on("end", () => {
  console.log("Finished reading file");
});
```

Example: Write stream

```
const writeStream = fs.createWriteStream("output.txt");

writeStream.write("Hello ");
```

```
writeStream.write("Node.js Streams\n");
writeStream.end();
```

Example: Pipe

```
const readStream = fs.createReadStream("bigfile.txt");
const writeStream = fs.createWriteStream("copy.txt");

readStream.pipe(writeStream);
```

Copies file **efficiently**

Real-world usage

- Video streaming (YouTube)
- File upload/download (MERN)
- Real-time logs
- Data pipelines

stream module in NestJS

- File upload streaming
- Streaming responses via HTTP adapters

```
@Get("file")
getFile(@Res() res) {
  const file = createReadStream("sample.txt");
  file.pipe(res);
}
```

Key points

- Streams = big files / large data
- Chunked processing = efficient
- Pipes simplify read/write
- Core for high-performance Node.js

9. crypto Module (Node.js Built-in)

The **crypto module** provides cryptographic functionality: **hashing, encryption, decryption, and signing**.

- Makes Node.js apps **secure**
- Used for passwords, tokens, and data integrity

Why it exists

Backend apps need to:

- Store passwords securely
- Generate tokens
- Encrypt sensitive data
- Verify data integrity

Without crypto ❌ → data is insecure

Hashing Example (Password)

```
const hash = crypto
  .createHash("sha256")
  .update("myPassword123")
  .digest("hex");

console.log(hash);
```

✅ One-way hash → cannot reverse

Hash-based Message Authentication (HMAC) (Keyed Hash)

```
const hmac = crypto
  .createHmac("sha256", "secretKey")
  .update("data")
  .digest("hex");

console.log(hmac);
```

Used for: API signature validation, JWT verification

Encryption & Decryption

```
const algorithm = "aes-256-cbc";
const key = crypto.randomBytes(32);
const iv = crypto.randomBytes(16);

// Encrypt
const cipher = crypto.createCipheriv(algorithm, key, iv);
let encrypted = cipher.update("Hello Node", "utf8", "hex");
encrypted += cipher.final("hex");
```

```
// Decrypt
const decipher = crypto.createDecipheriv(algorithm, key, iv);
let decrypted = decipher.update(encrypted, "hex", "utf8");
decrypted += decipher.final("utf8");

console.log(decrypted); // Hello Node
```

Real-World Usage

- Password hashing → bcrypt / crypto
- Token generation → JWT secrets
- Data encryption → files, messages
- Digital signatures → payments

crypto in NestJS

- Auth module
- Passwords → hashed
- API secrets → encrypted
- Tokens → generated

```
const hash = crypto.createHash("sha256").update(password).digest("hex");
```

Key Points

- Hashing → passwords (one-way)
- Encryption → sensitive data (reversible)
- HMAC → API signature / token integrity
- Security backbone of backend apps

10. child_process Module

The **child_process module** allows Node.js to **spawn new processes, execute system commands, and communicate with them via streams**.

- Run other programs, scripts, or shell commands from Node.js

Why it exists

- Node.js is **single-threaded**
- Use child processes to:
  - Run scripts
  - Perform parallel tasks
  - Automate system commands
- Prevents CPU-heavy tasks from blocking the main thread
-

exec() – runs a command in a shell and buffers output

```
exec("ls -l", (err, stdout, stderr) => {
  if (err) console.error(err);
  else console.log(stdout);
});
```

spawn() – spawns a process with stream interface

```
const ls = spawn("ls", ["-l"]);

ls.stdout.on("data", (data) => console.log(`Output: ${data}`));
ls.stderr.on("data", (data) => console.error(`Error: ${data}`));
```

fork() – special spawn for Node.js scripts with message passing

```
const child = fork("child.js");

child.on("message", (msg) => console.log("Message from child:", msg));
child.send({ hello: "world" });
```

Real-World Usage

- Running Python scripts
- PDF/report generation
- Video/image processing
- CPU-heavy task offloading
- CI/CD automation

MERN Example: Video conversion

```
const ffmpeg = spawn("ffmpeg", ["-i", "input.mp4", "output.mp4"]);

ffmpeg.on("close", () => console.log("Conversion done"));
```

Key Points

- exec → simple commands (buffers output)
- spawn → stream output, large tasks
- fork → Node-to-Node parallel processing
- Child processes **don't block main thread**

11. timers Module

Timers don’t guarantee exact timing

When you do:

```
setTimeout(fn, 2000);
```

It means:

“Run fn *after at least* 2000ms, when the event loop is free.”

If:

- CPU is busy
- Event loop is blocked
- Many callbacks are queued

Execution can be **delayed**.

This is why timers are called **minimum delay timers**.

Where timers fit in the Event Loop

Simplified Event Loop phases:

1. **Timers phase** → setTimeout, setInterval
2. **I/O callbacks**
3. **Idle / prepare**
4. **Poll phase**
5. **Check phase** → setImmediate
6. **Close callbacks**

process.nextTick runs **before all phases**.

setInterval gotcha (common bug)

If the callback takes longer than interval time:

```
setInterval(() => {
  // heavy task
}, 1000);
```

Safer pattern (recursive timeout):

```
function runTask() {
  setTimeout(() => {
    // task
    runTask();
  }, 1000);
}
runTask();
```

setImmediate vs setTimeout(0)

```
setTimeout(() => console.log("timeout"), 0);
setImmediate(() => console.log("immediate"));
```

Key idea:

- setImmediate → after I/O phase
- setTimeout(0) → timers phase (minimum delay)

In **I/O-heavy apps**, setImmediate is more predictable.

process.nextTick warning

```
process.nextTick(() => {
  console.log("nextTick");
});
```

Runs **before**:

- timers
- I/O
- setImmediate

Overuse can **starve the event loop** (dangerous).

cluster Module

The cluster module allows Node.js to create multiple worker processes that share the same server port, enabling the application to utilize multiple CPU cores.

Simple words:
cluster lets Node.js **run multiple instances of your server** to handle more traffic.

Why the cluster module exists

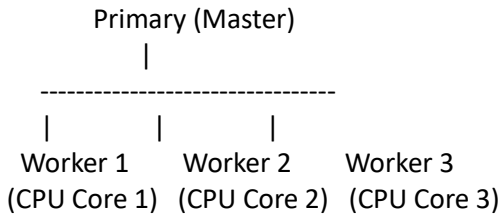
❌ **Problem**

- Node.js is **single-threaded**
- Uses **only one CPU core**
- Modern servers have **multiple cores**

✅ **Solution (cluster)**

- One **Primary (Master) process**
- Multiple **Worker processes**
- Each worker runs on a **separate CPU core**
- All workers share the **same port**

How cluster works (architecture)



- Primary manages workers
- Workers handle client requests
- Load balancing handled by Node.js

Importing cluster module

```
const cluster = require("cluster");
const os = require("os");
```

Core cluster properties

Property	Definition
cluster.isPrimary	Checks if current process is master
cluster.isWorker	Checks if current process is worker
cluster.workers	Object containing all active workers

Worker properties

Property	Definition
worker.id	Unique worker ID
worker.process	Access to worker process
worker.process.pid	Worker process ID

cluster methods

Method	Definition
cluster.fork()	Creates a new worker process
worker.kill()	Stops a worker process

cluster lifecycle events

Event	Meaning
cluster.on("fork")	Worker created
cluster.on("online")	Worker ready
cluster.on("exit")	Worker crashed / exited

Basic cluster example (MOST IMPORTANT)

```
const cluster = require("cluster");
const os = require("os");

if (cluster.isPrimary) {
  const cpuCount = os.cpus().length;

  for (let i = 0; i < cpuCount; i++) {
    cluster.fork();
  }

  cluster.on("exit", (worker) => {
    console.log(`Worker ${worker.process.pid} died`);
    cluster.fork(); // restart worker
  });
} else {
  require("./server"); // Express / HTTP server
}
```

- Each CPU core runs one worker
- Automatic recovery
- Same server port

Real-world backend usage

- High-traffic APIs
- Production servers
- Load-balanced Node apps
- Fault-tolerant systems

Example use cases:

- E-commerce platforms
- Payment gateways
- Streaming APIs

cluster vs child_process

Feature	cluster	child_process
Purpose	Scaling servers	Running tasks
Port sharing	✅ Yes	❌ No
Load balancing	Automatic	Manual
Best for	HTTP APIs	Scripts, jobs

cluster vs worker_threads (interview bonus)

cluster	worker_threads
Multi-process	Multi-thread
Separate memory	Shared memory
Best for servers	Best for CPU tasks

Key takeaways

- cluster = multi-core scaling
- Uses child processes
- Same server port
- Improves performance
- Mostly replaced by infrastructure solutions

8. Export Module in Node.js

1. What is a Module in Node.js?

A **module** is a reusable block of code encapsulated inside a file.

Key points:

- Each file in Node.js is treated as a **separate module**
- Variables, functions, and classes are **private by default**
- To share them across files, we use **exports**

Why Do We Need Export?

Without export:

- Code remains private
- Cannot be reused
- Poor project structure

With export:

- Code becomes reusable
- Clean and scalable architecture
- Similar to **Java packages**

Module Systems in Node.js

Node.js supports two module systems:

Module System	Syntax Used
CommonJS	require, module.exports
ES Modules	import, export

CommonJS Module System (Default)

Exporting a Function

```
module.exports = add;
```

Importing:

```
const add = require("./math");
```

Exporting Multiple Values

```
module.exports = { add, sub };
```

Importing:

```
const { add, sub } = require("./math");
```

Exporting a Class (Recommended for Backend)

```
module.exports = UserService;
```

Importing:

```
const UserService = require("./user.service");

const userService = new UserService();
userService.getUser();
```

module.exports vs exports

Syntax	Description
module.exports = value	Correct and recommended
exports.value = value	Works (reference)
exports = value	❌ Breaks reference

Best practice: Always use module.exports

ES Modules (Modern Node.js)

```
import { add } from "./math.js";
```

Default Export (Very Common)

```
export default class UserService {
  getUser() {
    return "User data";
  }
}
```

Import:

```
import UserService from "./user.service.js";
```

CommonJS vs ES Modules

Feature	CommonJS	ES Modules
Syntax	require	import
Export	module.exports	export
Default in Node	Yes	No
Used in	Express	NestJS

9. NPM – Node Package Manager

What is NPM?

NPM (Node Package Manager) is the default package manager for Node.js used to **install, manage, update, and share JavaScript libraries (packages)**.

When Node.js is installed, **NPM is installed automatically**.

In simple terms:

NPM helps developers **reuse code instead of writing everything from scratch**.

What is a Package?

A **package** is reusable JavaScript code written by developers and shared publicly or privately.

Common Examples:

- **express** – Web framework
- **mongoose** – MongoDB ODM
- **bcrypt** – Password hashing
- **cors** – Cross-origin resource sharing

Packages save time and make applications scalable.

What is the NPM Registry?

The **NPM Registry** is an online public database that stores all published NPM packages.

- Website: **npmjs.com**
- Contains millions of packages
- Anyone can publish packages

When you run:

```
npm install express
```

NPM performs:

1. Searches the registry
2. Downloads the package
3. Adds it to your project

Important NPM Files

1. package.json (MOST IMPORTANT)

package.json is the **heart of a Node.js project**.

It stores:

- Project metadata
- Dependencies
- Scripts
- Version information

Example:

```
{
  "name": "mern-backend",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

2. **node_modules/**
   - Stores all installed packages
   - Can be very large
   - ❌ Should never be pushed to GitHub

3. **package-lock.json**

package-lock.json **locks exact package versions**.

Why it matters:

- Same versions for all developers
- Prevents “works on my machine” issues
- Ensures consistent deployments

Initializing an NPM Project

```
npm init
```

Quick setup:

```
npm init -y
```

Creates:

- package.json

Installing Packages

1. **Local Installation (Project Dependency)**

npm install express

Saved under:

```
"dependencies": {
  "express": "^4.18.2"
}
```

2. **Development Dependency**

npm install nodemon --save-dev

- Used only during development.

Saved under:

```
"devDependencies": {
  "nodemon": "^3.0.0"
}
```

3. **Global Installation**

npm install -g nodemon

Used for:

- CLI tools
- System-wide commands

Dependencies vs DevDependencies

dependencies	devDependencies
Required in production	Required only during development
express	nodemon
mongoose	eslint
bcrypt	jest

NPM Scripts

Scripts automate tasks and are defined inside package.json.

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "test": "jest"
}
```

Run scripts using:

npm run dev

Used in:

- Development
- CI/CD pipelines
- Production servers

Updating Packages

npm update

Check outdated packages:

npm outdated

Removing Packages

npm uninstall express

NPM Cache

npm cache clean --force

Used when:

- Installation errors occur
- Cache becomes corrupted

NPM vs Yarn vs PNPM

Tool	Features
NPM	Default, widely used
Yarn	Faster installs, good caching
PNPM	Fastest, disk-efficient, monorepo-friendly

NestJS commonly uses **NPM or Yarn**.

NPM in MERN Stack

- Installs backend libraries
- Manages frontend dependencies
- Runs development & production scripts
- Handles build and deployment tasks

10. Debugging Node.js Applications

Debugging is the process of **finding, understanding, and fixing errors (bugs)** in an application.

In backend development, bugs can cause:

- Application crashes
- APIs not responding
- Wrong data being returned
- Performance issues
- Memory leaks

Debugging helps developers **see what is happening inside the Node.js application while it is running**.

Types of Errors in Node.js

Common errors developers face:

- **Syntax errors** – wrong code structure
- **Runtime errors** – errors while program runs
- **Logical errors** – code runs but gives wrong output
- **Async issues** – promises not resolved, callbacks not executed
- **Performance problems** – slow APIs, blocking code
- **Memory leaks** – increasing memory usage

Basic Debugging Using Console Methods

1. **console.log()**

```
console.log("User data:", user);
```

- Prints values to the console
- Most common and beginner-friendly debugging method

2. **console.error()**

```
console.error("Error occurred:", err);
```

- Used to print error messages
- Helps identify failures quickly

3. **console.table()**

```
console.table(users);
```

- Displays array or object data in table format
- Very useful for debugging database responses

4. **console.time() and console.timeEnd()**

```
console.time("dbCall");
await fetchData();
console.timeEnd("dbCall");
```

- Measures execution time
- Used for **performance debugging**

Node.js Built-in Debugger

Node.js comes with a **built-in debugger**, no extra tools required.

4Start Application in Debug Mode

node inspect app.js

This allows you to **step through code line by line**.

Important Debugger Commands

Command	Meaning
n	Move to next line
c	Continue execution
s	Step into function
o	Step out of function
repl	Inspect variables
bt	Show stack trace

Used mainly for **command-line debugging**.

Debugging with Chrome DevTools (Very Popular)

Run Node with Inspect Flag

node --inspect app.js

Pause at start:

node --inspect-brk app.js

Open Chrome DevTools

1. Open Chrome browser
2. Go to chrome://inspect
3. Click **Open dedicated DevTools**

What You Can Do in DevTools

- Set breakpoints
- Inspect variables
- Step through code
- View call stack
- Debug async functions
- Monitor memory usage

This method is **widely used in real projects**.

Debugging with VS Code (Most Recommended)

VS Code provides **powerful and beginner-friendly debugging**.

Create launch.json

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug Node App",
      "program": "${workspaceFolder}/app.js"
    }
  ]
}
```

Using Breakpoints

- Click on line number to add breakpoint
- Press **F5**
- Application pauses at that line

VS Code Debug Features

- Watch variables
- Call stack inspection
- Step over / step into
- Debug async functions
- View errors clearly

VS Code is the **best debugging tool for Node.js beginners and professionals**.

Debugging Express Applications

```
app.get("/user", (req, res) => {
  debugger;
  res.send("User");
});
```

- debugger keyword pauses execution
- Useful inside API routes and middleware

Debugging Async and Promise Issues

```
async function fetchData() {
  try {
    const data = await getData();
    console.log(data);
  } catch (err) {
    console.error(err);
```

```
  }
}
```

Best practices:

- Always use try/catch
- Log errors properly
- Avoid unhandled promise rejections

Debugging with Nodemon

```
nodemon --inspect app.js
```

Advantages:

- Auto-reload on code change
- Debugging enabled
- Saves development time

Debugging Best Practices

- Reproduce the bug
- Use breakpoints instead of too many logs
- Check async flow carefully
- Inspect request & response
- Read stack traces line by line

11. Node.js Event Loop

The Event Loop is the mechanism that allows Node.js to perform non-blocking asynchronous operations using a single thread.

Simple words :
Node.js has **one main thread**, but thanks to the Event Loop, it can **handle many tasks at the same time**.

Why Event Loop Exists

Problem (traditional blocking):

- One request blocks others → slow & inefficient

Node.js solution:

- Non-blocking
- Event-driven
- Asynchronous execution

Node.js is single-threaded, but NOT single-tasked ✅

High-Level Architecture

Client Requests
↓
Call Stack
↓
Event Loop
↓
Callback / Task Queues
↓
Call Stack

Core Components

1. **Call Stack** – executes functions
2. **Node APIs (libuv, C++ APIs)** – handle async tasks
3. **Event Loop** – monitors queues and call stack
4. **Callback / Task Queues** – store callbacks ready to execute

Simple Example

```
console.log("A");
setTimeout(() => console.log("B"), 0);
console.log("C");
```

Output:

```
A
C
B
```

- setTimeout callback goes to **Node APIs**, waits in queue, executed later
- console.log executes immediately

Event Loop Phases

The Event Loop runs in **phases**:

1. **Timers Phase**
   - Executes: setTimeout, setInterval

   ```
   setTimeout(() => console.log("timeout"), 0);
   ```

2. **Pending Callbacks Phase**
   - Executes system-level I/O callbacks

3. **Idle, Prepare Phase**
   - Internal Node.js use (ignore for most cases)

4. **Poll Phase** ⭐
   - Executes I/O callbacks (read/write)
   - Waits for new events or moves to next phase

5. **Check Phase**
   - Executes setImmediate

   ```
   setImmediate(() => console.log("immediate"));
   ```

6. **Close Callbacks Phase**
   - Executes cleanup tasks
   - Example: socket.on("close")

Special Queues

1. **process.nextTick**

```
process.nextTick(() => console.log("nextTick"));
```

   - Runs **before all phases**
   - Priority over timers, I/O, setImmediate

2. **Microtask Queue**
   - Contains Promise.then and async/await

   ```
   Promise.resolve().then(() => console.log("promise"));
   ```

   - Executes **after current stack, before Event Loop phases**

Execution Order Example

```
console.log("start");
setTimeout(() => console.log("timeout"));
Promise.resolve().then(() => console.log("promise"));
process.nextTick(() => console.log("nextTick"));
console.log("end");
```

Output:

start
end
nextTick
promise
timeout

Why Event Loop is Fast

- Non-blocking
- Uses OS for I/O
- Callbacks executed only when ready
- Efficiently handles thousands of requests

Event Loop in Real Applications

MERN Stack:

- Handles multiple API requests concurrently
- Non-blocking database calls
- Efficient real-time applications (chat, notifications)

NestJS:

- Runs on Node Event Loop
- Heavy use of async/await
- Same behavior, structured modularly

Common Mistakes ❌

❌ Blocking code:

while(true) {}

❌ Heavy CPU tasks on main thread

Solution ✅:

- Use **worker threads**
- Use **child_process**
- Offload tasks to queues or background jobs

Key Takeaways

- Event Loop = **heart of Node.js**
- Node.js = single-threaded but **async**
- Execution order is determined by **phases**
- Understanding it is **critical for backend performance**

12. Node.js EventEmitter (Documentation)

EventEmitter is a core Node.js class that allows objects to emit named events and register listeners to handle those events.

Simple terms:
Node.js apps are **event-driven**: things happen, and your code can **react** without blocking execution.

Why EventEmitter Exists

- Node.js is **asynchronous**.
- Instead of waiting/blocking, Node.js uses **events** to notify when tasks complete.
- Many core Node.js modules use EventEmitter internally:
  - fs → readable, end events
  - http → request, response events
  - Streams, sockets, timers, etc.
- Decouples code logic: producers emit events, consumers react.

Importing EventEmitter

```
const EventEmitter = require("events");
```

Creating an EventEmitter Instance

```
const myEmitter = new EventEmitter();
```

Listening to Events

```
myEmitter.on("greet", () => {
  console.log("Hello World!");
});
```

Registers a listener that runs every time the event is emitted.

Emitting Events

```
myEmitter.emit("greet");
```

- Calling emit triggers all listeners attached to that event.

Passing Arguments with Events

```
myEmitter.on("greet", (name) => {
  console.log(`Hello ${name}`);
});

myEmitter.emit("greet", "Kevin");
```

Pass multiple arguments if needed:

```
myEmitter.on("order", (id, status) => {
  console.log(`Order ${id} is ${status}`);
});

myEmitter.emit("order", 101, "shipped");
```

Listening Once

```
myEmitter.once("welcome", () => {
  console.log("Welcome event triggered");
});

myEmitter.emit("welcome");
myEmitter.emit("welcome"); // Won't run again
```

- once ensures a **one-time execution** for critical events.

Removing Event Listeners

```
function greet() {
  console.log("Hello again");
}

myEmitter.on("greet", greet);
myEmitter.removeListener("greet", greet);

myEmitter.emit("greet"); // Nothing happens
```

- Useful to avoid **memory leaks** or unwanted repeated events.

Real-world Usage

- **Streams:** readable, data, end events
- **HTTP servers:** request, response events
- **File uploads:** finish, error
- **Custom events:** userRegistered, orderShipped

EventEmitter in MERN Backend

```
// userEvent.js
const EventEmitter = require("events");
class UserEvents extends EventEmitter {}
module.exports = new UserEvents();

// main.js
const userEvents = require("./userEvent");

userEvents.on("userRegistered", (user) => {
  console.log(`New user: ${user.name}`);
});

userEvents.emit("userRegistered", { name: "Kevin" });
```

✅ Decouples logic: registration does not directly trigger emails or logs, it just emits an event.

Tips & Best Practices

1. Use once for single-use events to prevent memory leaks.
2. Always remove listeners if they're no longer needed.
3. Emit meaningful events for clear application flow.
4. Avoid heavy logic inside event listeners; use async functions when necessary.

Key Takeaways

- EventEmitter is **core to Node.js async design**.
- Use on, once, and emit to handle events.
- Pass data through events easily.
- Forms the foundation for **streams, HTTP, and custom events**.
- Directly maps to **NestJS event system**, making it easier to scale large apps.

13. Express.js (Most IMP)

Express.js is a fast, unopinionated, and minimalist web framework built on top of Node.js, used to create web servers and REST APIs easily.

In simple terms:

- Node.js provides the runtime to run JavaScript on the server
- Express.js provides **structure, tools, and simplicity** to build backend applications

Without Express, we must use the raw http module which requires a lot of boilerplate code.

Why Express.js is Important in MERN Stack

- Most backend APIs in MERN use Express
- It is lightweight and flexible
- Forms the foundation of **NestJS**
- Industry-standard backend framework
- Easy to learn, powerful to scale

Understanding Express properly means understanding **real backend development**.

Express.js Architecture (Request Flow)

Client
↓
HTTP Request
↓
Middleware Functions
↓
Route Handler
↓
Controller / Business Logic
↓
Response

Important:

👉 Everything in Express works as **middleware**

Creating a Basic Express Server

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  res.send("Hello Express");
});

app.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

Explanation:

- express() creates an application instance
- app.get() defines a route
- app.listen() starts the server

Request (req) and Response (res) Objects

Request Object (req)

Used to get data sent by the client.

Common properties:

- req.body → Data sent in request body
- req.params → URL parameters
- req.query → Query string values
- req.headers → Request headers

Response Object (res)

Used to send data back to the client.

Common methods:

- res.send()
- res.json()
- res.status()
- res.redirect()

Routing in Express

Routing determines **how the server responds to a client request at a specific URL and HTTP method**.

Basic Routes

```
app.get("/users", getUsers);
app.post("/users", createUser);
app.put("/users/:id", updateUser);
app.delete("/users/:id", deleteUser);
```

Route Parameters

```
app.get("/users/:id", (req, res) => {
  res.send(req.params.id);
});
```

Used when a value is part of the URL path.

Query Parameters

```
app.get("/search", (req, res) => {
  res.send(req.query.q);
});
```

Used for filtering, pagination, search, etc.

Middleware in Express (MOST IMPORTANT)
Middleware is a function that executes between the request and the response.

Syntax:

```
(req, res, next) => {}
```
next() passes control to the next middleware.

Types of Middleware

1. Built-in Middleware

```
app.use(express.json());
```
Used to parse incoming JSON data.

2. Custom Middleware

```
app.use((req, res, next) => {
  console.log("Request received");
  next();
});
```
Used for logging, authentication, validation, etc.

3. Third-party Middleware

Common examples:

- cors
- morgan
- helmet

```
app.use(cors());
```

Middleware Execution Flow

```
app.use(authMiddleware);
app.use(logMiddleware);

app.get("/profile", controller);
```

Execution order:

1. Authentication middleware
2. Logging middleware
3. Controller logic

Error Handling Middleware
Handles errors globally in the application.

```
app.use((err, req, res, next) => {
  res.status(500).json({ message: err.message });
});
```

Rule:
Error middleware must have **4 parameters**.

Express Router (Modular Routing)

Used to organize routes.

```
const express = require("express");
const router = express.Router();

router.get("/", getUsers);
router.post("/", createUser);

module.exports = router;
app.use("/users", router);
```

MVC Pattern in Express

MVC = Model View Controller

Folder structure:

```
routes/
controllers/
services/
models/
```

Flow:
Route → Controller → Service → Database

This structure improves scalability and readability.

Async/Await in Express

```
app.get("/data", async (req, res, next) => {
  try {
    const data = await fetchData();
    res.json(data);
  } catch (error) {
    next(error);
  }
});
```

Used to handle asynchronous operations safely.

Express with MongoDB (MERN)

```
app.post("/user", async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});
```

Express acts as the API layer between frontend and database.

Security Best Practices

- Use helmet
- Validate user input
- Sanitize data
- Handle errors centrally
- Protect routes using authentication middleware

Performance Best Practices

- Avoid blocking code
- Use async I/O
- Implement clustering
- Cache frequently used data

Express vs NestJS

Express.js	NestJS
Function-based	Class-based
Minimal structure	Opinionated architecture
Lightweight	Enterprise-ready
Middleware only	Guards, Pipes, Interceptors

NestJS internally uses Express.

14. Node JS middleware? (Most imp)

Middleware is a function that executes between the incoming request and the outgoing response and controls how the request is processed.

In simple terms:
Middleware works like a **checkpoint** in the backend that can:

- Modify the request
- Validate data
- Block the request
- Pass it forward

Middleware decides **what happens next**.

Middleware Execution Flow

Client Request
↓
Middleware 1
↓
Middleware 2
↓
Route Handler (Controller)
↓
Response

Each middleware has three choices:

- Call next() → continue
- Send response → stop
- Throw error → go to error handler

Middleware Function Signature

```
(req, res, next) => {}
```

Parameters:

- req → request object (data from client)
- res → response object (send data to client)
- next() → passes control to the next middleware

Why Middleware is the MOST IMPORTANT Concept

Middleware handles:

- Authentication
- Authorization
- Logging
- Input validation
- Error handling

- Rate limiting
- Body parsing

👉 **Almost the entire Express framework is built on middleware**

Types of Middleware

1. **Application-Level Middleware**

Applied globally to all routes.

```
app.use((req, res, next) => {
  console.log("Incoming request");
  next();
});
```

Use cases:

- Logging
- Global authentication
- Global validation

2. **Router-Level Middleware**

Applied only to a specific router.

```
router.use(authMiddleware);
```

Use cases:

- Protecting routes like /admin
- Module-specific logic

3. **Built-in Middleware (Express)**

Provided by Express itself.

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Purpose:

- Parse JSON data
- Parse form data

4. **Third-Party Middleware**

Installed using npm.

Common examples:

- cors → enable cross-origin requests
- helmet → security headers
- morgan → request logging

```
app.use(cors());
```

5. **Error-Handling Middleware** ⭐

Special middleware used only for errors.

Signature:

```
(err, req, res, next) => {}
```

Example:

```
app.use((err, req, res, next) => {
  res.status(500).json({ message: err.message });
});
```

Rule:
Must be defined **after all routes**.

Middleware Order (VERY IMPORTANT)

Middleware executes **top to bottom**.

```
app.use(auth);
app.use(logger);

app.get("/profile", controller);
```

Execution order:

1. auth
2. logger
3. controller

⚠ Wrong order can break authentication or security.

Authentication Middleware Example

```
function auth(req, res, next) {
  if (!req.headers.authorization) {
    return res.status(401).send("Unauthorized");
  }
  next();
}
```

Purpose:

- Protect private routes
- Check tokens or headers

Middleware That Stops Request

```
app.use((req, res, next) => {
  if (!req.user) {
    return res.status(403).send("Access denied");
  }
});
```

If next() is not called:

👉 Request ends immediately

Asynchronous Middleware

```
const asyncMiddleware = async (req, res, next) => {
  try {
    await someAsyncTask();
    next();
  } catch (error) {
    next(error);
  }
};
```

Used when middleware performs:

- Database calls
- API calls
- Async validation

Error Flow in Middleware

```
next(new Error("Something went wrong"));
```

Flow:

Middleware → Error Middleware → Response

Automatically skips normal middleware.

Middleware vs Controller

Middleware	Controller
Runs before route	Sends final response
Reusable	Route-specific
Cross-cutting logic	Business logic
Controls flow	Handles data

Express Middleware vs NestJS Concepts

Express	NestJS
Function-based	Class-based
Manual structure	Opinionated
app.use()	Decorators
Middleware only	Guards, Pipes, Interceptors

NestJS mapping:

- Middleware → Middleware
- Auth → Guards
- Validation → Pipes
- Logging → Interceptors

Real-World MERN Middleware Flow

Request
↓
Logger
↓
Authentication
↓
Authorization
↓
Validation
↓
Controller
↓
Error Handler

Common Mistakes

- Forgetting next()
- Wrong middleware order
- Heavy business logic inside middleware
- No centralized error handling

15. What is body-parser?

body-parser is Express middleware that reads incoming request bodies and converts them into JavaScript objects available in req.body.

In simple terms:
When a client sends data (JSON, form data, text), Node.js receives it as a **raw stream**.
body-parser converts that raw data into a **usable object**.

Why body-parser is Needed

Problem Without body-parser ❌

```
app.post("/user", (req, res) => {
  console.log(req.body); // undefined
});
```

Reason:

- Node.js does not automatically parse request bodies
- Data arrives as raw buffers

Solution With body-parser ✅

```
app.use(bodyParser.json());
```

Now the same request gives:

```
req.body = { name: "Kevin", age: 22 };
```

How Request Data Comes from Client

Frontend sends data like this:

```
fetch("/user", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name: "Kevin" })
});
```

Backend receives:

- Raw data stream
- body-parser reads it
- Converts it into a JavaScript object

Installing body-parser (Legacy Method)

```
npm install body-parser
const bodyParser = require("body-parser");

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Types of body-parser Middleware

1. JSON Parser

bodyParser.json()
Used for:

```
{
  "name": "Kevin",
  "age": 22
}
```

Most common for APIs.

2. URL-Encoded Parser (Form Data)

bodyParser.urlencoded({ extended: true })

Used for:

name=Kevin&age=22
Common in HTML forms.

3. Raw Body Parser

bodyParser.raw()

Used for:

- Binary data
- Webhooks
- File streams

4. Text Body Parser

bodyParser.text()

Used for:

- Plain text requests

body-parser is Built Into Express Now

From **Express 4.16+**, you do **NOT need to install body-parser separately**.

Modern Recommended Way

```
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

👉 Internally, Express uses **body-parser logic itself**.

body-parser vs express.json()

body-parser	express.json()
External package	Built-in
Older projects	Modern projects
Same functionality	Same functionality

Middleware Order (VERY IMPORTANT)

Correct order:

```
app.use(express.json());
app.use("/api", routes);
```

Incorrect order ❌:

```
app.use("/api", routes);
app.use(express.json());
```

If order is wrong:

- req.body will be undefined
- Validation and auth fail

Where body-parser Fits in Request Lifecycle

Client Request
↓
body-parser
↓
Authentication Middleware
↓
Validation Middleware
↓
Controller
↓
Response

Common Mistakes

- Forgetting body-parser / express.json()
- Wrong Content-Type header
- Using middleware after routes
- Expecting req.body in GET requests

body-parser in NestJS

NestJS uses Express internally.

```
@Post()
createUser(@Body() body) {
  return body;
}
```

👉 @Body() works because **body-parser is already enabled internally**.

Final Summary

- body-parser parses request body data
- Converts raw streams into JavaScript objects
- Built into Express now
- Core concept for MERN and NestJS backend development

16. Serving Static Resources in Node.js

Static resources are files that do not change on the server and are sent to the client exactly as they are.

In simple terms:
These files are **already created** and the backend just **delivers them**, without running any business logic.

Examples of Static Resources

- HTML files
- CSS files
- JavaScript files
- Images (PNG, JPG, SVG)
- Fonts
- PDFs

Why Serving Static Files is Important

Serving static resources is required because:

- Frontend needs CSS, JS, and images
- React build must be served in production
- Faster response (no processing needed)
- Supports browser caching & CDN
- Backend and frontend can run on same server

👉 Without static file serving, **frontend cannot work properly**.

Serving Static Files Using Express

Express provides a **built-in middleware**:

```
express.static()
```

This middleware:

- Reads files from a folder
- Sends them directly to the client
- Skips route handlers and controllers

Basic Example (Most Common)

Project Structure

```
project/
 ├─ server.js
 └─ public/
    ├─ index.html
    ├─ style.css
    └─ app.js
```

Server Code

```
const express = require("express");
const app = express();

app.use(express.static("public"));

app.listen(3000);
```

Result

You can open directly in browser:

- /index.html
- /style.css
- /app.js

How Express Serves Static Files (Internal Flow)

```
Client Request
    ↓
express.static middleware
    ↓
File System check
    ↓
File sent as response
```

If file exists:

- Express sends it
- No controller is executed

If file does not exist:

- Request moves to next middleware or route

Serving Static Files with URL Prefix

```
app.use("/static", express.static("public"));
```

Now files are accessed as:

/static/style.css

/static/app.js

Useful for:

- Better organization
- CDN-like paths

Serving Multiple Static Folders

```
app.use(express.static("public"));
app.use(express.static("uploads"));
```

Express checks folders **in order**.

Use case:

- public → frontend assets
- uploads → user uploaded files

Real MERN Use Case (VERY IMPORTANT)

Serving React Build Using Node.js

```
const path = require("path");

app.use(express.static("client/build"));

app.get("*", (req, res) => {
  res.sendFile(
    path.join(__dirname, "client", "build", "index.html")
  );
});
```

Why this is important

- Single server for frontend + backend
- Easy deployment
- Used in production MERN apps

Serving Uploaded Files (Images, PDFs)

```
app.use("/uploads", express.static("uploads"));
```

Now uploaded files are accessible at:

```
http://localhost:3000/uploads/image.png
```

Common use cases:

- Profile images
- Documents
- Product images

Serving Static Files Without Express (Low Level)

```
const http = require("http");
const fs = require("fs");

http.createServer((req, res) => {
```

```
  fs.readFile("index.html", (err, data) => {
    res.end(data);
  });
}).listen(3000);
```

❌ Too much boilerplate
✅ Express handles this efficiently with one line

Security Best Practices

Never expose sensitive folders:

- .env
- node_modules
- config

Correct way:

```
app.use(express.static(path.join(__dirname, "public")));
```

Only expose files meant for the client.

Caching Static Files (Performance)

Express supports caching automatically.

You can configure cache headers:

```
app.use(express.static("public", {
  maxAge: "1d"
}));
```

Benefits:

- Faster page load
- Reduced server load

Static Files vs Dynamic Content

Static Content	Dynamic Content
No logic	Business logic runs
Faster	Slower
Cacheable	Hard to cache
Files	API responses

Static Resources in NestJS

```
app.useStaticAssets(join(__dirname, "..", "public"));
```

Key points:

- Same concept as Express
- Class-based configuration
- Uses Express internally

Common Mistakes

- Wrong folder path
- Static middleware after routes
- Exposing sensitive files
- Forgetting SPA fallback (index.html)
- Incorrect use of relative paths

Final Summary

- Static resources are unchanged files
- Express serves them using built-in middleware
- Essential for MERN production apps
- Same concept applies to NestJS
- Improves performance and scalability

17. Data Access in Node.js

Data access in Node.js refers to the process of reading, writing, updating, and deleting data from a data source such as a database, file system, or external API.

In simple terms:
👉 Data access means **how your backend talks to data**.

Examples:

- Get user from database
- Save product details
- Update profile information
- Delete records

Why Data Access is Important

Every backend application needs data.

Without data access:

- No login system
- No user records
- No products
- No orders

👉 **Backend = Logic + Data Access**

Types of Data Sources in Node.js

Node.js can access data from multiple sources:

1. **Databases**
   - MongoDB
   - MySQL
   - PostgreSQL
   - SQLite
2. **File System**
   - JSON files
   - Logs
   - Uploaded files
3. **External APIs**
   - Payment gateways
   - Weather APIs
   - Third-party services

Data Access Flow in Node.js (Big Picture)

Client Request
↓
Controller
↓
Service Layer
↓
Data Access Layer
↓
Database / File / API

👉 This separation makes the app **clean, scalable, and testable**.

Data Access Using File System (Basic)

Node.js provides the fs module for file-based data access.

Example: Reading Data from a File

```
const fs = require("fs");

const data = fs.readFileSync("users.json", "utf8");
console.log(data);
```

Use cases:

- Small projects
- Config files
- Logs

❌ Not suitable for large applications

Data Access Using Databases (MOST IMPORTANT)

Database	Type
MongoDB	NoSQL
MySQL	SQL
PostgreSQL	SQL

👉 **MongoDB is most common in MERN stack**

Data Access with MongoDB (Using Mongoose)

What is Mongoose?

Mongoose is an ODM (Object Data Modeling) library for MongoDB and Node.js.

It helps:

- Define schemas
- Validate data
- Interact with database easily

Connecting to MongoDB

```
const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/mydb")
  .then(() => console.log("DB connected"))
  .catch(err => console.error(err));
```

Creating a Model (Schema)

```
const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  age: Number
});

const User = mongoose.model("User", userSchema);
```

Create (Insert Data)

```
const user = await User.create({
  name: "Kevin",
  email: "kevin@mail.com",
  age: 22
});
```

Read (Fetch Data)

```
const users = await User.find();
```

Update Data

```
await User.findByIdAndUpdate(id, { age: 23 });
```

Delete Data

```
await User.findByIdAndDelete(id);
```

👉 These operations are called **CRUD operations**.

Data Access Layer (DAL) – BEST PRACTICE

Data Access Layer is a separate layer responsible only for database operations.

Why use DAL?

- Clean code
- Easy testing
- Database independent
- Scalable architecture

Example: Data Access Layer

```
// user.repository.js
class UserRepository {
  createUser(data) {
    return User.create(data);
  }

  getAllUsers() {
    return User.find();
  }
}

module.exports = new UserRepository();
```

Using it in Service Layer

```
const userRepo = require("./user.repository");

class UserService {
  getUsers() {
    return userRepo.getAllUsers();
  }
}
```

Async/Await in Data Access

All data access in Node.js is **asynchronous**.

```
async function getUser() {
  const user = await User.findOne({ email });
  return user;
}
```

Why async?

- Non-blocking
- High performance
- Handles multiple users

Error Handling in Data Access

```
try {
  const user = await User.findById(id);
} catch (error) {
  console.error("DB Error:", error.message);
}
```

Always handle:

- Connection errors
- Query errors
- Validation errors

Data Access with SQL (Example)

```
connection.query(
  "SELECT * FROM users",
  (err, results) => {
    if (err) throw err;
    console.log(results);
  }
);
```

Used in:

- Enterprise systems
- Financial apps

Data Access in Express (Typical Pattern)

```
app.get("/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});
```

Flow:

Route → Controller → Service → DB

Data Access in NestJS

```
@Injectable()
export class UserService {
  constructor(
    @InjectModel(User.name) private userModel: Model<User>
  ) {}

  findAll() {
    return this.userModel.find();
  }
}
```

18. File uploading in Node.js

What is Multer Storage?

Storage defines where and how uploaded files are temporarily stored before final use.

Multer provides **two main storage types**:

Disk Storage – stores file on server
Memory Storage – stores file in RAM (buffer)

DISK STORAGE (Most Common for Local / Simple Apps)

What is Disk Storage?

Files are saved physically on the server's file system (folder).

- Easy
- Beginner-friendly
- Not ideal for large-scale production

When to use Disk Storage?

- Local development
- Small apps
- Document uploads
- No cloud integration

Disk Storage Example

📁 **Project Structure**

```
project/
 ├─ server.js
 ├─ uploads/
 └─ routes/
```

Multer Disk Storage Config

```
const multer = require("multer");
const path = require("path");

const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "uploads/");
  },
  filename: (req, file, cb) => {
    const uniqueName =
      Date.now() + "-" + Math.round(Math.random() * 1e9);

    cb(null, uniqueName + path.extname(file.originalname));
  }
});

const upload = multer({ storage });
```

Upload Route (Single File)

```
app.post("/upload-disk", upload.single("file"), (req, res) => {
  res.json({
    message: "File uploaded using disk storage",
    file: req.file
  });
});
```

Serve Uploaded Files

```
app.use("/uploads", express.static("uploads"));
```

Access:

http://localhost:3000/uploads/filename.jpg

Disk Storage Output (req.file)

```
{
  filename: "170000000-photo.jpg",
  path: "uploads/170000000-photo.jpg",
  mimetype: "image/jpeg",
  size: 45678
}
```

Disk Storage Pros & Cons

Pros	Cons
Easy setup	Server storage fills
Files persist	Slower than RAM
Debug friendly	Manual cleanup

MEMORY STORAGE (BEST FOR PRODUCTION & CLOUD)

What is Memory Storage?

Files are stored in RAM as Buffer, not saved to disk.

- Fast
- Best for cloud upload
- Not stored permanently

When to use Memory Storage?

- Cloudinary / S3 upload
- Microservices
- Scalable apps
- Production systems

Memory Storage Example

Multer Memory Storage Config

```
const multer = require("multer");

const storage = multer.memoryStorage();

const upload = multer({ storage });
```

Upload Route (Single File)

```
app.post("/upload-memory", upload.single("file"), async (req, res) => {
  res.json({
    message: "File received in memory",
    fileInfo: {
      originalname: req.file.originalname,
      size: req.file.size,
      mimetype: req.file.mimetype
    }
  });
});
```

Memory Storage Output (req.file)

```
{
  originalname: "photo.png",
  buffer: <Buffer ...>,
  mimetype: "image/png",
  size: 34567
}
```

📌 buffer = raw binary data in RAM

Upload to Cloud using Memory Storage (REAL USE)

Example (Cloudinary-style):

```
cloudinary.uploader.upload_stream(
  { resource_type: "image" },
  (error, result) => {
    if (error) return res.status(500).json(error);
    res.json(result);
  }
).end(req.file.buffer);
```

👉 No file ever touches server disk.

Memory Storage Pros & Cons

Pros	Cons
Very fast	RAM usage
No cleanup	Large files risky
Best for cloud	Not persistent

FILE VALIDATION (Works for BOTH)

File Type Validation

```
const upload = multer({
  storage,
  fileFilter: (req, file, cb) => {
    if (file.mimetype.startsWith("image/")) {
      cb(null, true);
    } else {
      cb(new Error("Only images allowed"), false);
    }
  }
});
```

File Size Limit

```
const upload = multer({
  storage,
  limits: {
    fileSize: 1024 * 1024 * 2 // 2MB
  }
});
```

FRONTEND (React) – SAME FOR BOTH

```
const formData = new FormData();
formData.append("file", file);

axios.post("/upload-disk", formData, {
  headers: {
    "Content-Type": "multipart/form-data"
  }
});
```

DISK vs MEMORY (INTERVIEW MUST)

Feature	Disk Storage	Memory Storage
Stored in	File system	RAM
Speed	Medium	Fast
Cleanup	Required	Auto
Best for	Local apps	Cloud uploads
Production	❌	✅

19. RESTful API

What is an API?

API (Application Programming Interface) allows two software systems to communicate with each other.

Simple example:

- Frontend (React) → API → Backend (Node)
- Backend → API → Database

What is REST?

REST (Representational State Transfer) is an architectural style for designing networked applications.

Important point ⭐
👉 REST is **NOT a library or framework**, it's a **set of rules**.

What is a RESTful API?

A RESTful API is an API that follows REST principles.

Means:

- Proper URLs
- Proper HTTP methods
- Stateless communication
- Standard responses

Core REST Principles

1. **Client–Server Separation**
   - Frontend & backend are independent
   - React doesn't care about DB
   - Backend doesn't care about UI

2. **Statelessness (VERY IMPORTANT)**

Each request contains all the information needed to process it.

- No server-side sessions
- JWT / tokens

3. **Resource-Based URLs**

Use **nouns**, not verbs ❌

```
/getUsers
/users
```

---

4. **HTTP Methods (CRITICAL)**

Method	Purpose
GET	Read
POST	Create
PUT	Update (full)
PATCH	Update (partial)
DELETE	Delete

5. **Standard HTTP Status Codes**

Code	Meaning
200	OK
201	Created
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not Found
500	Server Error

6. **REST API URL Design (BEST PRACTICES)**

6.1. Resource Collection

```
GET   /users
POST  /users
```

6.2. Single Resource

```
GET   /users/:id
PUT   /users/:id
DELETE /users/:id
```

RESTful API Example (Express)

```
app.get("/users", getUsers);
app.post("/users", createUser);
app.get("/users/:id", getUser);
app.put("/users/:id", updateUser);
app.delete("/users/:id", deleteUser);
```

Clean, readable, scalable ✔

Request Components in REST

Request includes:

- URL
- HTTP Method
- Headers
- Body (POST/PUT)
- Query Params

Response Structure (STANDARD)

```
{
  "success": true,
  "data": {},
  "message": "User created"
}
```

Consistency is key.

RESTful API in MERN

React
↓
Axios / Fetch
↓
REST API (Express)
↓
MongoDB

Frontend never touches DB directly.

REST vs GraphQL

REST	GraphQL
Multiple endpoints	Single endpoint
Fixed response	Flexible response
Simple	Complex

👉 REST is best for most apps.

Versioning REST APIs

/api/v1/users
/api/v2/users

Important for production apps.

Pagination & Filtering (REST Style)

```
GET /users?page=1&limit=10
GET /users?role=admin
```

Authentication in REST APIs

- Stateless
- JWT
- Tokens in headers

Authorization: Bearer token

RESTful API Error Handling

```
{
  "error": "User not found"
}
```

Use proper HTTP codes.

REST in NestJS

```
@Controller("users")
export class UserController {
  @Get()
  findAll() {}

  @Post()
  create() {}
}
```

👉 Same REST principles, class-based.

Common REST Mistakes ❌

- Using verbs in URLs
- Ignoring HTTP status codes
- Inconsistent response formats
- Server-side sessions

20. Node Global Objects

What are Global Objects?

Global objects are built-in objects in Node.js that are available in all modules without importing them.

Simple explanation:
Like window in browsers, Node provides objects to manage **environment, paths, timers, buffers, and logs**.

Key point: No require() needed — accessible everywhere.

Examples of Node Global Objects

Object	Purpose
global	Node's global object, like window
process	Current process info and control
Buffer	Handle binary data
__dirname	Absolute path of current directory
__filename	Absolute path of current file
console	Logging & debugging
setTimeout	Schedule function execution
setInterval	Repeat execution
setImmediate	Execute after current event loop phase
process.nextTick	Execute before next event loop phase
globalThis	Standard JS global object (Node equivalent = global)

global Object

- Like window in the browser.
- Stores globally accessible variables.

```
global.myVar = "Hello World";
console.log(global.myVar); // Hello World
```

✅ Useful for config or shared constants (use carefully to avoid conflicts).

__dirname & __filename

- __dirname → Absolute path of current directory
- __filename → Absolute path of current file

```
console.log(__dirname);  // e.g., /Users/kevin/project
console.log(__filename); // e.g., /Users/kevin/project/app.js
```

Used for **file paths**, static resources, and uploads.

module, exports, require

Node files = modules.

```
// math.js
module.exports.add = (a, b) => a + b;

// app.js
const math = require("./math");
console.log(math.add(2, 3)); // 5
```

process Object (VERY IMPORTANT)

Provides **info & control over Node process**.

```
console.log(process.pid);      // process ID
console.log(process.version);  // Node version
console.log(process.platform); // OS platform
console.log(process.env);      // Environment variables

process.on("exit", (code) => {
  console.log("Exit code:", code);
});
```

Exit manually:

```
process.exit(1);
```

Buffer Object

Handles **binary data**.

```
const buf = Buffer.from("Hello");
console.log(buf);          // <Buffer 48 65 6c 6c 6f>
console.log(buf.toString()); // Hello
```

Used in:

- File uploads
- Streams
- TCP/HTTP sockets

Timers

Function	Purpose
setTimeout(fn, ms)	Execute function after delay
setInterval(fn, ms)	Execute repeatedly
setImmediate(fn)	Run after current event loop phase
process.nextTick(fn)	Run **before** next event loop phase

```
setTimeout(() => console.log("timeout"), 0);
setImmediate(() => console.log("immediate"));
process.nextTick(() => console.log("nextTick"));
```

console Object

```
console.log("Log");
console.error("Error");
console.warn("Warning");
console.table([{ name: "Kevin", age: 22 }]);
```

- Debugging & logging is easier
- Works in Node & NestJS

globalThis

- Modern, standard global object
- Equivalent to Node's global

```
globalThis.myValue = 100;
console.log(globalThis.myValue); // 100
```

Practical Example — File Upload Path

```
const path = require("path");
const fs = require("fs");

const uploadDir = path.join(__dirname, "uploads");

if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir);
}


console.log("Upload directory ready at:", uploadDir);
```

- __dirname → get current folder
- fs → file system module
- path → build cross-platform paths

Global Objects in NestJS

- Node global objects **work the same in NestJS**
- process.env → environment configs
- Buffer → file upload/stream handling
- setTimeout, setInterval → background tasks

NestJS internally uses Node globals.

21. What is typeScript in Nodejs?

TypeScript is a superset of JavaScript that adds static types and modern features to JS.

Key points:

- Compiles to plain JS → runs in Node
- Detects errors **at compile time**
- Supports ES6+ features (classes, modules, decorators)
- Works perfectly with Node + Express + NestJS

Why use TypeScript in Node.js?

Problem in JS	TypeScript Solution
No type checking	Static typing (string, number, boolean, etc.)
Hard to refactor	Types make refactoring safe
No IDE hints	Autocomplete & IntelliSense
Bugs at runtime	Errors detected during compile

Installing TypeScript in Node.js

```
npm install -g typescript
tsc –version
```

- Initialize a Node + TS project:

```
npm init -y
tsc --init
```

Generates **tsconfig.json**.

tsconfig.json (Important)

Key options:

```
{
  "target": "ES6",      // JS version output
  "module": "commonjs",  // Node module system
  "outDir": "./dist",    // Compiled JS output
  "rootDir": "./src",    // TS source folder
  "strict": true,       // Type checking strict mode
  "esModuleInterop": true
}
```

Basic Node + TypeScript Example

Project structure:

```
project/
  ├– src/
```

```
|  └─ index.ts
└─ dist/
```

```
// src/index.ts
const greet = (name: string): string => {
  return `Hello ${name}`;
};

console.log(greet("Kevin"));
```

Compile & run:

```
tsc
node dist/index.js
```

TypeScript Features in Node.js

1. Types

```
let name: string = "Kevin";
let age: number = 22;
let isActive: boolean = true;
```

2. Interfaces

```
interface User {
  name: string;
  email: string;
}

const user: User = { name: "Kevin", email: "kevin@test.com" };
```

3. Classes & OOP

```
class User {
  constructor(public name: string, private email: string) {}

  getInfo() {
    return `${this.name} - ${this.email}`;
  }
}

const u = new User("Kevin", "kevin@test.com");
console.log(u.getInfo());
```

✅ This directly maps to **NestJS classes**

4. Modules

```
// user.ts
export class User {}

// index.ts
import { User } from "./user";
```

5. Enums

```
enum Role {
  ADMIN = "admin",
  USER = "user"
}

const myRole: Role = Role.ADMIN;
```

6. Generics

```
function wrap<T>(value: T): T {
  return value;
}

console.log(wrap<string>("Hello"));
```

7. TypeScript in Express + Node

```
import express, { Request, Response } from "express";

const app = express();

app.get("/users", (req: Request, res: Response) => {
  res.json({ message: "Hello TypeScript" });
});

app.listen(3000, () => console.log("Server running"));
```

- Types prevent errors in req and res
- Makes middleware safer

8. TypeScript in NestJS

NestJS is **built with TypeScript**:

```
@Controller("users")
export class UserController {
  @Get()
  findAll(): string {
    return "All users";
  }
}
```

- Types + classes + decorators = NestJS power
- Node.js + TS = foundation for enterprise backend

9. Advantages of Using TypeScript in Node

- Safer code & fewer runtime bugs
- IDE autocomplete & navigation
- Cleaner OOP structure → easier NestJS migration
- Scalable for big projects

Key Takeaways

- TypeScript = MUST for modern Node & NestJS
- Adds types, interfaces, classes
- Prevents runtime bugs
- Improves productivity & scalability
- NestJS is 100% TS-based → perfect preparation

22. What is the use strict?

"use strict" is a **directive in JavaScript that enables strict mode**, which enforces **more rigorous error checking** and **prevents certain unsafe actions**.

Simple words:
👉 It tells JS to **be stricter** about your code, helping avoid common mistakes.

How to use it

1. **Global strict mode**

```
"use strict";
x = 10; // ❌ Throws error: x is not defined
```

2. **Function-level strict mode**

```
function test() {
  "use strict";
  y = 20; // ❌ Error
}
```

What strict mode changes

Behavior	Normal JS	Strict Mode
Assigning undeclared variable	Creates global var	Throws error
this in functions	Global object (window / global)	undefined
Duplicates in object literals	Allowed	SyntaxError
Deleting variables	Ignored	Throws error
Octal literals (012)	Allowed	SyntaxError

Examples

1. **Prevent global leaks**

```
"use strict";

function hello() {
  message = "Hello"; // ❌ Error
}
hello();
```

Without strict mode → creates global.message silently.
With strict mode → **error immediately**. ✅

2. Safer this

```
"use strict";

function test() {
  console.log(this);
}

test(); // undefined in strict mode, global in normal JS
```

3. Prevent duplicate property names

```
"use strict";

const obj = { name: "Kevin", name: "Patel" };
// ❌ SyntaxError
```

4. Disallow deleting variables

```
"use strict";

let x = 10;
delete x; // ❌ SyntaxError
```

Why it matters in Node.js

- Node runs **strict mode by default in ES6 modules**
- Helps prevent accidental global variables in large apps
- Makes **TypeScript + Node + NestJS** safer
- Forces better code quality for **production-ready apps**

Comparison

Without "use strict"	With "use strict"
Mistakes may silently fail	Mistakes throw errors
this = global object	this = undefined (function)
Globals created accidentally	Error if undeclared
Less safe code	Safer, predictable code

In Node.js Today

- CommonJS modules (require) → not strict by default, but use strict can be used
- ES Modules (import/export) → strict mode **automatic**

```
// ES module
import fs from "fs"; // already strict mode
```

Best Practices

- Always write "use strict" in **vanilla JS files**
- Use **ES6 modules** → auto strict
- Prevent bugs in **backend code**
- Works perfectly with **TypeScript**, which is strict by default

Key Takeaways

- "use strict" = better, safer JS
- Helps in Node.js backend for **robust apps**
- Enforces rules like no undeclared variables, safer this, no duplicate keys
- Automatically enabled in ES modules / TypeScrip

23. What is process.nextTick, setimmediate, setinterval? with difference

Node.js Timers & nextTick

Node.js has **three main ways** to schedule code asynchronously:

1. process.nextTick
2. setImmediate
3. setInterval (and also setTimeout)

They **run at different phases of the event loop**.

1. **process.nextTick**

Schedules a callback to be executed immediately after the current operation completes, before the event loop continues.

Key Points

- Executes **before I/O events**
- Higher priority than setImmediate
- Good for **deferring execution without leaving current phase**

Example

```
console.log("Start");

process.nextTick(() => {
  console.log("Next Tick callback");
});

console.log("End");
```

Output:

```
Start
End
Next Tick callback
```

✅ Runs **before the next event loop tick**, after current code.

2. **setImmediate**

Schedules a callback to be executed on the next iteration of the event loop.

Key Points

- Executes **after I/O events in the current loop**
- Good for **running code after I/O**

Example

```
console.log("Start");

setImmediate(() => {
  console.log("setImmediate callback");
});

console.log("End");
```

Output:

```
Start
End
setImmediate callback
```

✅ Similar to nextTick, but happens **after I/O callbacks**, not before.

Example: nextTick vs setImmediate

```
const fs = require("fs");

fs.readFile(__filename, () => {
  setImmediate(() => console.log("setImmediate"));
  process.nextTick(() => console.log("nextTick"));
});
```

Output:

```
nextTick
setImmediate
```

✅ nextTick always runs **before setImmediate**, even in I/O callbacks.

3. **setInterval**

Schedules a function to run repeatedly at a fixed time interval.

Example

```
let count = 0;

const interval = setInterval(() => {
  console.log("Interval running", count++);
  if (count === 3) clearInterval(interval);
}, 1000);
```

Output:

```
Interval running 0
Interval running 1
Interval running 2
```

Key Points

- Repeats until clearInterval
- Timer-based → not precise (depends on event loop load)
- Great for repeated background tasks

Comparison Table

Feature	process.nextTick	setImmediate	setInterval
Execution phase	Microtask queue (before event loop continues)	Next event loop iteration	Timers phase repeatedly
Timing	After current operation	After I/O callbacks	Every X ms
Repeats	No	No	Yes, until cleared
Use case	Deferring small tasks immediately	Running after I/O	Repeated jobs / heartbeat
Priority	High	Medium	Low relative to microtasks

Quick Notes

- setTimeout(fn, 0) ≈ setImmediate, but not guaranteed order
- process.nextTick **can starve the event loop** if abused
- setInterval depends on **event loop availability**, not exact timing

Example Combined

```
console.log("Start");

setTimeout(() => console.log("setTimeout"), 0);
setImmediate(() => console.log("setImmediate"));
process.nextTick(() => console.log("nextTick"));

console.log("End");
```

Possible Output:

```
Start
End
nextTick
setTimeout
setImmediate
```

✅ nextTick > setTimeout / setImmediate order may vary slightly depending on environment.

Node.js MERN Usage Examples

- process.nextTick → small deferred DB validation
- setImmediate → heavy processing after I/O
- setInterval → background cleanup, cron jobs, cache refresh

NestJS Relevance

- process.nextTick → internal hooks / interceptors
- setInterval → @Interval() decorator for scheduled tasks
- setImmediate → rarely, for post-I/O deferred tasks

🔹 **Key Takeaways**

- process.nextTick → highest priority microtask
- setImmediate → next tick / post I/O
- setInterval → repeated jobs
- Understanding event loop phases is critical for **non-blocking MERN backend**
- Same principles apply in **NestJS + async services**

24. oops concepts in javascript (Most imp)

OOP is a programming paradigm that **organizes code using objects and classes**, making it modular, reusable, and maintainable.

1. Classes

What is a class?

Blueprint for creating objects with properties and methods.

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    return `Hello, ${this.name}`;
  }
}

const user1 = new User("Kevin", 22);
console.log(user1.greet()); // Hello, Kevin
```

Key points:

- constructor → initializes object
- Methods are shared via prototype

2. Objects

Objects are **instances of a class** or **plain JS objects**.

```
const user = {
  name: "Kevin",
  age: 22,
  greet() {
    return `Hi, ${this.name}`;
  }
};

console.log(user.greet());
```

3. Encapsulation

Encapsulation = bundling data + methods and controlling access.

```
class BankAccount {
  #balance; // private property

  constructor(balance) {
    this.#balance = balance;
  }
```

```
  deposit(amount) {
    this.#balance += amount;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount(1000);
account.deposit(500);
console.log(account.getBalance()); // 1500
```

Key points:

- Private properties → #
- Only accessible via methods

4. Inheritance

Inheritance = child class can use parent class properties & methods

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    return `Hello, ${this.name}`;
  }
}

class Employee extends Person {
  constructor(name, position) {
    super(name); // call parent constructor
    this.position = position;
  }
}

const emp = new Employee("Kevin", "Developer");
console.log(emp.greet()); // Hello, Kevin
console.log(emp.position); // Developer
```

5. Polymorphism

Polymorphism = same method, different behavior in subclasses

```
class Animal {
  speak() {
    return "Animal sound";
  }
}

class Dog extends Animal {
  speak() {
    return "Woof!";
```

```
  }
}

const dog = new Dog();
console.log(dog.speak()); // Woof!
```

6. Abstraction

Abstraction = hiding internal details, exposing only essential features

```
class Car {
  startEngine() {
    this.#igniteFuel();
  }

  #igniteFuel() {
    console.log("Fuel ignited!");
  }
}

const car = new Car();
car.startEngine(); // Fuel ignited!
// car.#igniteFuel() ❌ Not accessible
```

7. Constructor & super

- constructor() → initializes object
- super() → calls parent class constructor
- Must call super() in subclass before this

8. Getters & Setters

```
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name;
  }

  set name(value) {
    this._name = value;
  }
}

const p = new Person("Kevin");
console.log(p.name); // Kevin
p.name = "Patel";
console.log(p.name); // Patel
```

✅ Makes data access controlled and cleaner.

9. Static Methods & Properties

Belongs to the class, not instances

```
class MathHelper {
  static square(x) {
    return x * x;
  }
}

console.log(MathHelper.square(5)); // 25
```

10. OOP in Node.js & MERN

- **Node.js** → Classes for services, controllers
- **NestJS** → Fully class-based: @Injectable, @Controller
- **MERN Backend** → Model classes (Mongoose schemas + TS classes)

Common Mistakes ❌

- Forgetting super() in subclasses
- Overusing global variables instead of encapsulation
- Ignoring this context
- Confusing private (#) vs public properties

Quick Recap Table

OOP Concept	JS Implementation	Example
Class	class keyword	class User {}
Object	Instance	new User()
Encapsulation	Private #	#balance
Inheritance	extends	class Employee extends Person {}
Polymorphism	Method overriding	speak() in subclass
Abstraction	Hide details	Private methods
Getter/Setter	get, set	_name property
Static	static keyword	MathHelper.square()

Key Takeaways

- OOP in JS = classes, objects, encapsulation, inheritance, polymorphism, abstraction
- Used heavily in **Node.js backend + NestJS**
- Helps **modular, reusable, scalable backend code**
- Must practice with **MERN backend models, services, controllers**

25. mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js.

It provides:

- Schema-based structure for MongoDB documents
- Validation, type checking, and default values
- Easy querying and CRUD operations
- Middleware (hooks) support
- Relationship handling (population)

Why use Mongoose in MERN?

Without Mongoose	With Mongoose
Direct MongoDB queries (MongoClient)	Structured schemas & models
No validation	Schema-based validation
Hard to maintain large apps	Easier to scale & maintain
No middleware/hooks	Middleware (pre, post) for logic

✅ Mongoose helps **write scalable, maintainable MERN backends**.

Installing Mongoose

```
npm install mongoose
```

Connecting to MongoDB

```
const mongoose = require("mongoose");

mongoose.connect("mongodb://localhost:27017/mydb", {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log("MongoDB connected"))
.catch(err => console.error(err));
```

Notes:

- useNewUrlParser and useUnifiedTopology = recommended options
- Can also use **Atlas cloud MongoDB URI**

Defining a Schema

A **schema defines the structure of your documents**.

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
```

```
    trim: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  },
  age: {
    type: Number,
    min: 0
  },
  createdAt: {
    type: Date,
    default: Date.now
  }
});
```

Key points:

- type → JS type
- required → validation
- default → default value
- trim → remove spaces
- unique → ensures unique field

Creating a Model

```
const User = mongoose.model("User", userSchema);
```

- User = Model name (singular, Mongoose converts to plural collection users)
- Use the model to **interact with MongoDB**

CRUD Operations

1. Create a document

```
const newUser = new User({
  name: "Kevin",
  email: "kevin@test.com",
  age: 22
});

newUser.save()
  .then(user => console.log(user))
  .catch(err => console.error(err));
```

2. Read documents

```
User.find()
  .then(users => console.log(users));

User.findOne({ email: "kevin@test.com" })
  .then(user => console.log(user));
```

3. Update documents

```
User.updateOne({ email: "kevin@test.com" }, { age: 23 })
  .then(result => console.log(result));
```

Or with **findByIdAndUpdate**:

```
User.findByIdAndUpdate(userId, { name: "Kevin Patel" }, { new: true })
  .then(user => console.log(user));
```

4. Delete documents

```
User.deleteOne({ email: "kevin@test.com" })
  .then(result => console.log(result));

User.findByIdAndDelete(userId)
  .then(user => console.log(user));
```

Query Helpers

- find(), findOne() → read
- sort({ age: -1 }) → descending
- limit(5) → limit results
- select("name email") → project fields

```
User.find().sort({ age: -1 }).limit(5).select("name email");
```

Middleware (Hooks)

Mongoose supports **pre** and **post hooks** for events.

```
userSchema.pre("save", function(next) {
  console.log("Before saving:", this);
  next();
});

userSchema.post("save", function(doc) {
  console.log("After saving:", doc);
});
```

Virtuals (Computed Fields)

```
userSchema.virtual("info").get(function() {
  return `${this.name} (${this.email})`;
});

const user = await User.findOne();
console.log(user.info); // Kevin (kevin@test.com)
```

Populating References

- Supports **relationships between collections**

```
const postSchema = new mongoose.Schema({
  title: String,
```

```
  author: { type: mongoose.Schema.Types.ObjectId, ref: "User" }
});

const Post = mongoose.model("Post", postSchema);

Post.find().populate("author").then(posts => console.log(posts));
```

Validation & Error Handling

```
const invalidUser = new User({ email: "notanemail" });

invalidUser.save().catch(err => {
  console.log("Validation error:", err.message);
});
```

- Mongoose handles **type errors, required fields, unique constraints**

Connecting Mongoose with Express (MERN Example)

```
const express = require("express");
const mongoose = require("mongoose");
const app = express();

app.use(express.json());

mongoose.connect("mongodb://localhost:27017/mydb");

const userSchema = new mongoose.Schema({ name: String, email: String });
const User = mongoose.model("User", userSchema);

app.post("/users", async (req, res) => {
  try {
    const user = await User.create(req.body);
    res.json(user);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
});

app.get("/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});

app.listen(3000, () => console.log("Server running"));
```

✅ Simple MERN backend with **Mongoose + Express**.

Best Practices

- Always define **schemas**
- Use **timestamps**: { timestamps: true }
- Use **indexes** for faster queries
- Handle **errors and validation**
- Use **async/await** for cleaner code

NestJS + Mongoose

NestJS provides **@nestjs/mongoose** module:

```
@Schema()
export class User {
  @Prop({ required: true })
  name: string;

  @Prop()
  age: number;
}

export const UserSchema = SchemaFactory.createForClass(User);
```

- Classes + decorators = same concepts as Node.js OOP
- Models injected into services with @InjectModel()

Key Takeaways

- Mongoose = bridge between Node.js and MongoDB
- Provides **schema, validation, middleware, hooks**
- Essential for **MERN backend development**
- Works perfectly with **Express + NestJS**
- Makes MongoDB **structured, scalable, and maintainable**