Node.js Interview Q&As (Part 1)

🔹 Section 1: Node.js Basics

Q1. What is Node.js and why use it?
Answer: Node.js is a runtime environment that lets you run JavaScript outside the browser, powered by Chrome's V8 engine. It's event-driven, non-blocking, and ideal for APIs, real-time apps, and scalable services.
💡 Tip: Say → "Node.js is best for I/O-heavy apps (APIs, chat, streaming), not CPU-heavy tasks."

Q2. How does Node.js handle asynchronous code?
Answer: Node.js uses an **event loop** with a non-blocking I/O model. Instead of waiting for one task to finish, it registers a callback and continues executing.
Example:

```
fs.readFile("file.txt", () => console.log("Done!"));

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

💡 Tip: Always say → "Node.js is single-threaded but can handle thousands of requests asynchronously."

Q3. What are global objects in Node.js?
Answer: Global objects are built-in variables available everywhere in Node.js without importing modules.
Examples:

- __dirname → path of current directory
- __filename → full path of current file
- process → info about current process
- Buffer → handle binary data
- setTimeout() → scheduling functions

💡 Tip: Don't confuse Node.js globals with browser globals like window or document.

Q4. What is REPL in Node.js?
Answer: REPL = **Read, Eval, Print, Loop**. It’s a command-line tool for testing Node.js code snippets quickly. You type JavaScript, it executes instantly.
Example:

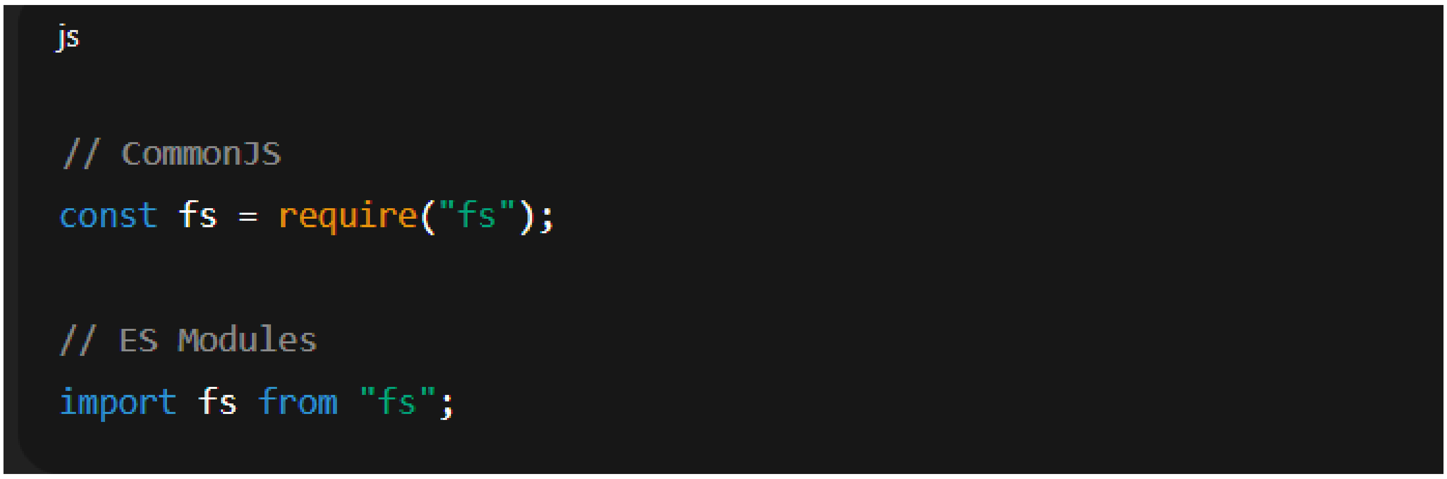
```
node
> 2 + 3
5
> require("fs")
```

💡 Tip: In interviews, mention → “REPL is useful for debugging and trying out code snippets quickly.”

🔹 Section 2: Modules & NPM

Q5. What are CommonJS vs ES Modules in Node.js?
Answer:

- **CommonJS** → Uses require() and module.exports.
- **ES Modules** → Use import and export.
  Example:

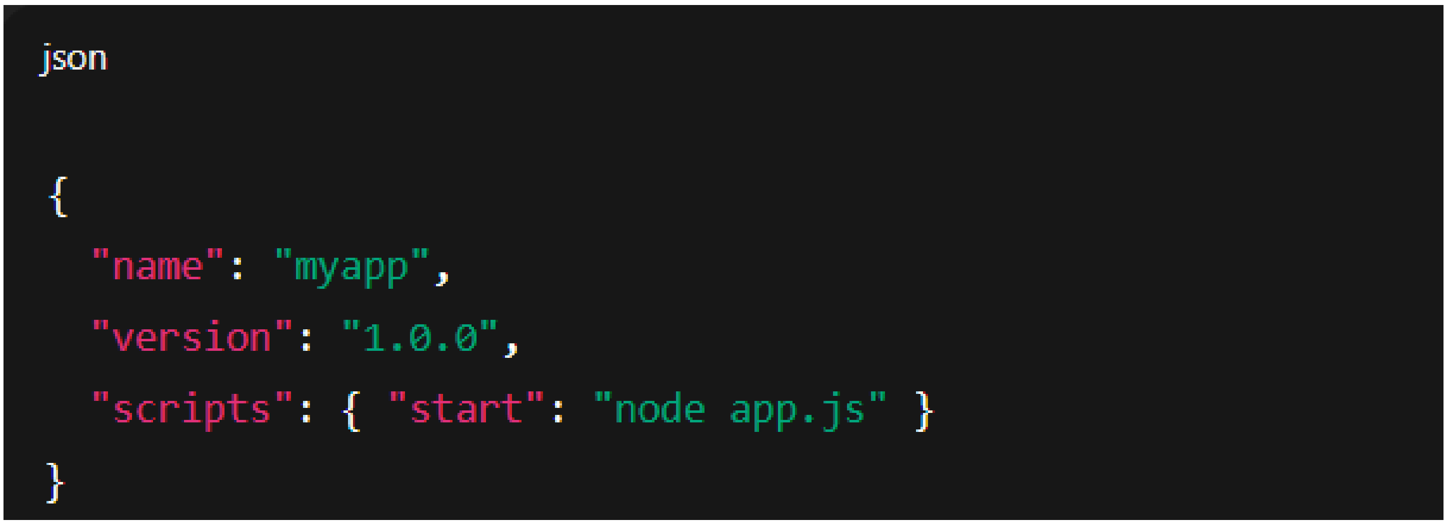


```js
// CommonJS
const fs = require("fs");

// ES Modules
import fs from "fs";
```

💡 Tip: Say → “CommonJS is default, ES Modules are modern standard in Node.js.”

Q6. What is package.json in Node.js?
Answer: package.json is the heart of a Node.js project. It stores project metadata, dependencies, scripts, and entry points.
Example:


json

```json
{
  "name": "myapp",
  "version": "1.0.0",
  "scripts": { "start": "node app.js" }
}
```

💡 Tip: Interviewers may ask → “How are versions locked?” → Answer: via package-lock.json.

Q7. What is the difference between npm and npx?
Answer:

- npm → Installs packages.
- npx → Runs executables from packages without installing globally.
  Example:

npx create-react-app myapp

npm create vite@latest my-app

💡 Tip: Say → “npx avoids version conflicts by using local binaries.”

🔹 Section 3: Async & Streams

Q8. What is process.nextTick() in Node.js?
Answer: It defers a function execution until the current operation

completes, before moving to the next event loop phase.

- In **Node.js**, the event loop has multiple **phases** (timers, I/O callbacks, idle, poll, check, close, etc.).
- process.nextTick() **adds a callback to the "next tick queue"**, which is processed **immediately after the current synchronous code finishes**, **before the event loop continues to the next phase**.

So it **runs earlier** than setTimeout, setImmediate, or even Promise.then.

Example:

```js
console.log("Start");
process.nextTick(() => console.log("NextTick"));
console.log("End");
// Output: Start → End → NextTick
```

💡 Tip: Always mention → "process.nextTick runs before Promises and setTimeout."

Q9. What are Streams in Node.js?
Answer: Streams let you process data chunk by chunk instead of loading it all at once (better for large files).

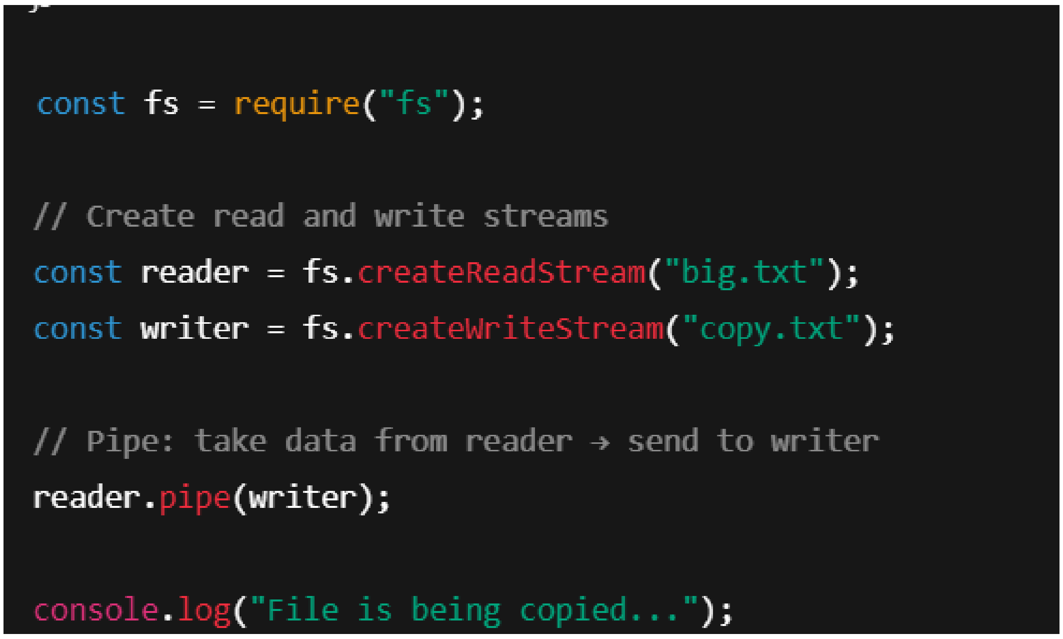
- A **stream** is like a continuous flow of data.
- Instead of **loading the entire file/data into memory at once**, Node.js allows you to **read/write in small chunks**.
- This is **memory-efficient** and **fast**, especially for large files, video/audio, or network transfers.

🔹 Why use Streams?

Imagine copying a **10GB video file**:

- If you use fs.readFile(), Node.js will load the **entire 10GB into memory** → your app may crash.
- With fs.createReadStream(), Node.js **reads small chunks** (default ~64KB) and writes them as they arrive, without keeping the whole file in memory.

  Example:



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

// Create read and write streams
const reader = fs.createReadStream("big.txt");
const writer = fs.createWriteStream("copy.txt");

// Pipe: take data from reader → send to writer
reader.pipe(writer);

console.log("File is being copied...");
```

💡 Tip: Say → "Streams are best for video/audio and large file transfers."

Q10. What are Buffers in Node.js?

Answer: Buffers are temporary memory used to store binary data (images, files) directly, especially useful with streams.

Example:

const buf = Buffer.from("Hello");

console.log(buf.toString()); // Hello

💡 Tip: Buffers = raw binary, Strings = human-readable.

🔹 Section 4: Express.js & APIs

Q11. What is Express.js and why use it?

Answer: Express.js is a lightweight web framework for Node.js used to

build APIs and web servers. It simplifies routing, middleware, and handling requests/responses.
Example:

```
app.get("/", (req, res) => res.send("Hello World"));
```

💡 Tip: Say → “Express is unopinionated and widely used.”

Q12. What is middleware in Express.js?
Answer: Middleware functions run between a request and response. They can log requests, authenticate users, or modify data before sending a response.
Example:

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

💡 Tip: Always mention → “Middleware order matters in Express.”

Q13. How do you handle authentication in Node.js?
Answer: Two main methods:

- **Sessions (cookies)** → Store login state on server.
- **JWT (JSON Web Token)** → Stateless, scalable authentication for APIs.
  Example (JWT):

```
jwt.sign({ id: user.id }, "secret", { expiresIn: "1h" });
```

💡 Tip: Say → “JWT scales better in distributed systems.”

Q14. Difference between res.send(), res.json(), and res.end()?
Answer:

- res.send() → Sends any type of response (string, buffer, object).
- res.json() → Sends JSON response only.
- res.end() → Ends the response without data.
  💡 Tip: Always use res.json() in REST APIs for clarity.
-

🔹 Section 5: Performance & Scaling

Q15. What is clustering in Node.js?
Answer: Node.js runs on a single core by default. Clustering allows using multiple CPU cores by running worker processes.
Example:

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

if (cluster.isMaster) { cluster.fork(); }
```

💡 Tip: Say → "Clustering improves performance on multi-core systems."

Q16. What is PM2 in Node.js?
Answer: PM2 is a process manager that helps run Node.js apps in production with features like clustering, restarts, and monitoring.

Problem

- If your Node.js app crashes, you'd have to restart it manually.
- If you want to run clustering, you'd have to manage worker processes yourself.
- You also need **monitoring, logging, and smooth restarts** for production.

Solution → PM2

- **PM2 is a process manager** for Node.js.
- It automates:
  - Running your app in **cluster mode** (multi-core).
  - **Restarting automatically** if it crashes.
  - **Zero-downtime reloads** (users don't notice when you deploy a new version).
  - Built-in monitoring (pm2 monit).
  - Log management.
    💡 Tip: Always say → "PM2 ensures zero-downtime deployments."

🔷 Section 6: Security & Best Practices

Q17. How do you handle environment variables in Node.js?
Answer: Use .env files with the dotenv package to load sensitive data securely.
Example:

```
require("dotenv").config();
```

```
console.log(process.env.DB_URL);
```

💡 Tip: Never commit .env files to GitHub.

Q18. How do you prevent SQL Injection in Node.js?
Answer: Use **parameterized queries** or ORMs (like Sequelize/Mongoose) instead of string concatenation.

- **SQL Injection** happens when user input is directly placed in queries.
  Example (❌ vulnerable):
- `` db.query(`SELECT * FROM users WHERE id = ${userId}`); ``

Prevention in SQL:

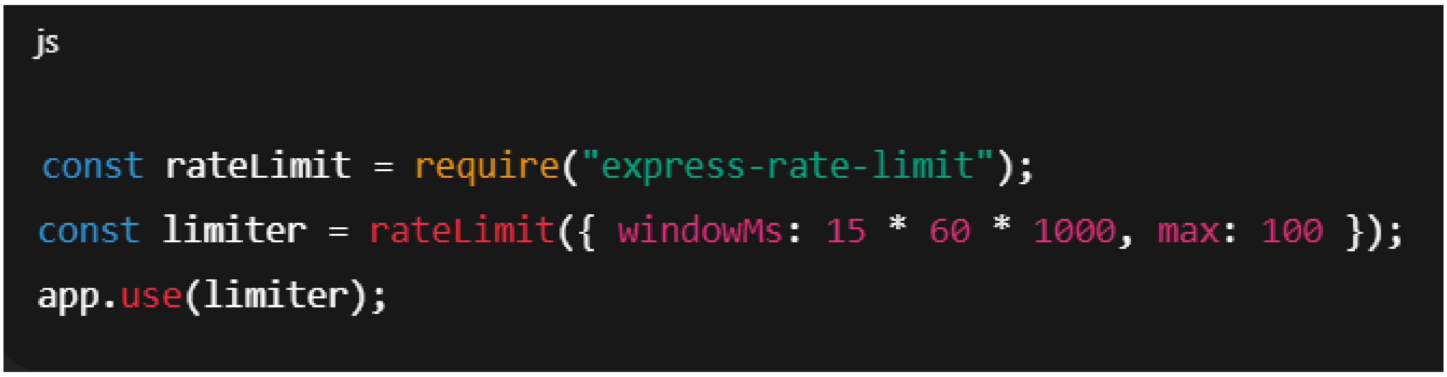
1. **Parameterized queries (prepared statements)**
2. **db.query("SELECT * FROM users WHERE id = ?", [userId]);**
3. **Use ORMs/Query Builders** (Sequelize, Prisma).
4. **Sanitize & validate inputs** (e.g., check that id is numeric).
   Example:

```
db.query("SELECT * FROM users WHERE id = ?", [userId]);
```

💡 Tip: Always sanitize user input.

Q19. How do you prevent DDoS or brute-force attacks in Node.js?
Answer: DDoS and brute-force attacks overwhelm your server with too many requests.
To prevent this:

- **Rate Limiting** → restrict how many requests a user/IP can make in a given time.



```js
const rateLimit = require("express-rate-limit");
const limiter = rateLimit({ windowMs: 15 * 60 * 1000, max: 100 });
app.use(limiter);
```

- **Caching** → serve repeated requests from cache (Redis, CDN) instead of hitting the server/database every time.
- **Web Application Firewall (WAF)** → blocks malicious traffic before it reaches your app (e.g., Cloudflare, AWS WAF).
- **Account Lockout / Delays** → after several failed login attempts, temporarily block or slow down responses.

💡 Tip: *"Throttling + caching prevents abuse."*

Q20. How do you handle error handling globally in Node.js?
Answer: Use centralized error-handling middleware and global process handlers.
Example:

app.use((err, req, res, next) => res.status(500).send(err.message));

process.on("uncaughtException", console.error);

💡 Tip: Say → "Centralized error handling makes debugging easier."