Node.js

⭐ Node.js — HIGH-PROBABILITY INTERVIEW QUESTIONS (1–2 YOE)

1. Basics of Node.js

✅ 1. What is Node.js?

■ Explanation (simple, conceptual)

Node.js is a **JavaScript runtime environment** built on Chrome's **V8 engine** that allows developers to run JavaScript **outside the browser**.

It uses an **event-driven, non-blocking (asynchronous)** architecture, making it excellent for scalable server applications.

■ When/Why is it used

- Building backend REST APIs
- Real-time systems (chat, notifications)
- Streaming apps, IoT, microservices
- High-concurrency applications

■ Example

```
const http = require('http');
http.createServer((req, res) => {
  res.end("Hello from Node");
}).listen(3000);
```

■ Short interview line

"Node.js is a fast, event-driven JavaScript runtime used to build scalable backend services."

✅ 2. Why is Node.js single-threaded?

■ Explanation (simple, conceptual)

Node.js **uses a single-threaded event loop** to handle multiple concurrent requests without creating a new thread for each.

This avoids:

- Context switching

- Memory overhead
- Complexity of multithreading

Instead, heavy tasks are offloaded to **libuv's thread pool**.

■ When/Why

- Efficient for I/O-heavy operations (API calls, DB ops)
- Keeps programming simple while still supporting large traffic

■ Example (conceptual)

One thread → many requests because async callbacks run when ready.

■ Short interview line

"Node is single-threaded to reduce overhead and use the event loop to handle many requests efficiently."

✅ 3. Is Node.js good for CPU-intensive tasks? Why/why not?

■ Explanation (simple, conceptual)

❌ **No, Node.js is not ideal for CPU-heavy tasks** like:

- image processing
- encryption
- machine learning
- complex loops

Because **CPU-intensive tasks block the event loop**, preventing Node from handling other requests.

■ When/Why

- Node excels at **I/O-heavy**, not **CPU-heavy** workloads.
- If CPU-heavy work is needed, Node can use:
  - Worker Threads
  - Clustering
  - Offloading to another service

■ Example

A long loop will freeze the server:

```
while(true) {} // blocks the event loop
```

■ Short interview line

"Node isn't ideal for CPU-heavy work because it blocks the event loop and slows all requests."

✅ 4. What is the V8 Engine?

■ Explanation (simple, conceptual)

V8 is Google Chrome's open-source JavaScript engine written in C++.

It **compiles JavaScript to machine code** for extremely fast execution.

Node.js uses V8 internally.

■ When/Why it is used

- Provides high performance
- Efficient garbage collection
- Powers both Chrome and Node.js
- Enables Node to run JS outside browser

■ Example

```
console.log("Hello");
// V8 compiles this to native machine code
```

■ Short interview line

"V8 is Google's JavaScript engine that compiles JS to machine code and powers Node's high performance."

✅ 5. What makes Node.js fast?

■ Explanation (simple, conceptual)

Node.js is fast primarily because of:

✔ 1. V8 Engine

Compiles JS to native machine code.

✔ 2. Event-driven, non-blocking I/O

Does not wait for slow operations (DB/file/API).

Makes it highly scalable.

✔ 3. libuv library

Handles the event loop + thread pool for background tasks.

✔ 4. Lightweight, single-thread model

Avoids heavy thread creation overhead.

■ When/Why it is used

- Handling thousands of concurrent connections
- Real-time, high-performance apps

■ Example (conceptual)

```
fs.readFile('file.txt', () ⇒ {
  console.log("Done reading");
});
// Non-blocking — Node continues serving other requests
```

■ Short interview line

“Node is fast because of V8, non-blocking I/O, and the event-loop/libuv combination.”

---

2. Event Loop (MOST IMPORTANT)

✅ 1. Explain the event loop in Node.js.

■ Explanation (simple, conceptual)

The **event loop** is the core of Node.js.

It allows Node (which is **single-threaded**) to handle **multiple asynchronous operations** without blocking the main thread.

Node executes JavaScript on the **call stack**, but all async operations (timers, file I/O, database queries) are handled by **libuv** in the background, and their callbacks are pushed back into queues when ready.

The event loop continuously checks:

1. Is the call stack empty?
2. If yes → pick the next callback/microtask and run it.

■ When/Why is it used

- Enables non-blocking I/O
- Core reason why Node can handle thousands of requests
- Makes Node scalable without threads

■ Example

```
console.log("Start");
setTimeout(() ⇒ console.log("Timeout"), 0);
console.log("End");

// Output:
// Start
```

```
// End
// Timeout (runs later through event loop)
```

■ Short interview line

“The event loop lets Node handle async operations without blocking, even though it runs JavaScript on a single thread.”

✅ 2. What are the phases of the event loop?

■ Explanation (simple)

Node’s event loop has **six phases**:

1. **Timers Phase**

   Executes callbacks from `setTimeout()` and `setInterval()`.

2. **Pending Callbacks**

   Executes I/O callbacks deferred from the previous cycle.

3. **Idle/Prepare**

   Internal use (not for developers).

4. **Poll Phase**

   - Retrieves new I/O events
   - Executes I/O-related callbacks (file system, network)
   - Waits if nothing to do

5. **Check Phase**

   Executes callbacks from **setImmediate()**

6. **Close Callbacks**

   Runs close events (e.g., socket.on("close"))

■ When/Why it is used

- Important for understanding callback ordering
- Helps you explain timing behavior in callbacks and async functions

■ Short interview line

“Node’s event loop runs in phases like timers, poll, and check, ensuring async operations execute in order.”

✅ 3. What is the call stack, callback queue, microtask queue?

■ Explanation (simple, conceptual)

✔️ Call Stack

Where JavaScript executes code **line by line**.

Only one function runs at a time.

✔️ Callback Queue (Task Queue)

Contains callbacks from:

- setTimeout
- setInterval
- setImmediate
- I/O operations

Event loop moves them to the call stack when stack is empty.

✔️ Microtask Queue

Contains:

- Promise `.then()` callbacks
- `async/await` resolved callbacks
- `process.nextTick()` (Node-specific, highest priority)

Microtasks run **before** callbacks in the callback queue.

■ When/Why it is used

Understanding this explains:

- Why promises run before setTimeout
- Why some callbacks execute earlier than expected

■ Example

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

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

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

console.log("B");

// Output:
// A
// B
// Promise  ⟵ microtask
// Timeout  ⟵ callback queue
```

■ Short interview line

“Call stack runs code, microtask queue has promises, and callback queue handles timers/I/O; microtasks run first.”

✅ 4. Difference between process.nextTick() and setImmediate()

■ Explanation (simple)

Feature	process.nextTick()	setImmediate()
Priority	**Highest priority** (runs before microtasks)	Runs in **check phase**
Execution	Runs **immediately after current operation**	Runs **after I/O callbacks**
Danger	Can block the event loop if overused	Safe, designed for async callbacks

■ When/Why used

✔ process.nextTick()

Use when:

- You must run a callback **before the event loop continues**
- Clean-up, quick corrections

✔ setImmediate()

Use when:

- You want to run code **after I/O events**
- Avoid blocking async flow

■ Example

```
setImmediate(() ⇒ console.log("Immediate"));
process.nextTick(() ⇒ console.log("NextTick"));

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

Output:

```
Start
NextTick   ← runs first
Immediate
```

■ Short interview line

“nextTick runs before the event loop continues; setImmediate runs in the check phase after I/O.”

✅ 5. Why is Node.js non-blocking?

■ Explanation (simple, conceptual)

Node.js uses:

- **Event loop** → to manage async operations
- **libuv thread pool** → to perform heavy I/O in the background
- **Callbacks/promises** → to return results later

Because Node **does not wait** for I/O to finish, the main thread keeps serving other requests.

■ When/Why used

- Enables handling **thousands of concurrent connections**
- Perfect for network apps, real-time apps, I/O-heavy workloads

■ Example

```
fs.readFile("file.txt", () => {
  console.log("File read");
});
console.log("Continuing work...");
```

■ Short interview line

"Node is non-blocking because async operations run in libuv's thread pool while the event loop continues execution."

---

3. Modules

✅ 1. CommonJS vs ES Modules

■ Explanation (simple, conceptual)

Node.js supports two module systems:

CommonJS (CJS)

- Older, default system in Node
- Uses `require()` and `module.exports`
- Synchronous loading
- File extension: `.js`

ES Modules (ESM)

- Modern JavaScript module system
- Uses `import` and `export`
- Asynchronous loading

- File extension: `.mjs` or `"type": "module"` in package.json

■ When/Why is it used

- **CommonJS** → legacy Node apps, synchronous module loading
- **ESM** → modern apps, cleaner syntax, better tree-shaking, browser compatibility

■ Example

CommonJS

```
const utils = require("./utils");
module.exports = { sum };
```

ES Modules

```
import utils from "./utils.js";
export function sum(a, b) { return a + b; }
```

■ Short interview line

"CommonJS uses require/module.exports; ESM uses import/export with modern async loading."

✅ 2. What is `require()` vs `import`?

■ Explanation (simple)

`require()` → CommonJS

`import` → ES Modules

require()

- Loads modules dynamically
- Works anywhere in code
- Synchronous
- Node.js default

import

- Must be top-level
- Asynchronous loading
- Static structure → compiler can optimize
- Used in modern JavaScript

■ When/Why used

- Use `require()` for traditional Node.js projects
- Use `import` for modern apps, bundlers, or when mixing frontend & backend JS modules

■ Example

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

// ESM
import fs from "fs";
```

■ Short interview line

"require is CommonJS and synchronous; import is ES Modules and asynchronous with cleaner syntax."

✅ 3. How to export multiple functions in Node.js?

■ Explanation (simple)

In Node.js you can export multiple functions using:

✔ CommonJS

Using **module.exports = { }** or `exports.functionName =` .

✔ ES Modules

Using **named exports** (`export function` or `export { }`).

■ When/Why used

- To create reusable utility modules
- To separate logic into modular files (controllers, services, helpers)

■ Example

CommonJS

```
// utils.js
function add(a, b) { return a + b; }
function sub(a, b) { return a - b; }

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

Using

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

ES Modules

```
export function add(a, b) { return a + b; }
export function sub(a, b) { return a - b; }
```

Using

```
import { add, sub } from "./utils.js";
```

■ Short interview line

“In CommonJS use module.exports; in ESM use named exports with export {}.”

✅ 4. What is module caching?

■ Explanation (simple, conceptual)

When a module is loaded using `require()`, Node **caches it** so future `require()` calls return the **same instance** without reloading or re-running the file.

This improves performance and ensures **shared state** between modules.

■ When/Why is it used

- To avoid re-executing modules repeatedly
- To improve performance
- Useful in sharing in-memory objects (e.g., config, singletons)

■ Example

```
// file1.js
console.log("Module loaded");
module.exports = { counter: 1 };

// main.js
const a = require("./file1");  // prints "Module loaded"
const b = require("./file1");  // does NOT print again - cached

console.log(a === b); // true
```

■ Short interview line

“Node caches modules after first require(), so subsequent imports are fast and share the same instance.”

---

4. NPM & Package Management

✅ 1. What is NPM?

■ Explanation (simple, conceptual)

NPM (**Node Package Manager**) is the default package manager for Node.js.

It provides:

- A registry of open-source packages
- Tools to install, update, remove dependencies
- Script management (`npm start`, `npm test`)

■ When/Why is it used

- To install frameworks like Express, React, Lodash
- To manage versions of libraries
- To automate tasks using NPM scripts

■ Example

```
npm install express
```

■ Short interview line

"NPM is Node's package manager that installs and manages project dependencies."

✅ 2. Difference between dependencies vs devDependencies

■ Explanation (simple)

Category	Purpose	Installed in production?
dependencies	Required at runtime (Express, Mongoose)	✔ Yes
devDependencies	Needed only for development (Jest, ESLint, nodemon, Webpack)	❌ No

■ When/Why used

- **dependencies** → core application packages
- **devDependencies** → tools for testing, linting, compiling

■ Example

```
npm install express        # dependency
npm install jest --save-dev # devDependency
```

■ Short interview line

“dependencies run in production; devDependencies are only for development tools.”

✅ 3. What is package-lock.json?

■ Explanation (simple, conceptual)

`package-lock.json` **locks exact versions** of all installed packages and their sub-dependencies.

It ensures:

- Same versions across all environments
- Faster installs (cached tree)
- Predictable builds

■ When/Why used

- Prevents “works on my machine” issues
- Ensures consistent deployments in CI/CD
- Tracks integrity/security of dependencies

■ Example

```
"express": {
  "version": "4.18.2",
  "resolved": "...",
  "integrity": "sha512-..."
}
```

■ Short interview line

“package-lock.json freezes exact dependency versions to ensure consistent builds.”

✅ 4. What is semantic versioning (1.3.0)?

■ Explanation (simple)

Semantic versioning follows:

MAJOR.MINOR.PATCH

Example: **1.3.0**

- **1 → MAJOR** (breaking changes)
- **3 → MINOR** (new features, backward compatible)
- **0 → PATCH** (bug fixes, no new features)

■ When/Why used

- To understand the impact of upgrading a package
- To avoid breaking changes in production

■ Example

```
^1.3.0  # allows minor & patch updates
~1.3.0  # allows only patch updates
1.3.0   # fixed version, no updates
```

■ Short interview line

“Semantic versioning is MAJOR.MINOR.PATCH — major breaks, minor adds features, patch fixes bugs.”

✅ 5. How do you update and remove a package?

■ Explanation (simple)

✔️ Update a package

```
npm update <package-name>
```

To update globally:

```
npm install -g <package-name>
```

To check outdated:

```
npm outdated
```

✔️ Remove a package

```
npm uninstall <package-name>
```

■ When/Why used

- Keep libraries secure and up to date
- Remove unused dependencies to reduce bundle size

■ Example

```
npm update express
npm uninstall lodash
```

■ Short interview line

“Use npm update to upgrade packages and npm uninstall to remove them.”

---

5. Asynchronous Programming

✅ 1. What is asynchronous programming?

■ Explanation (simple, conceptual)

Asynchronous programming allows tasks to run **in the background** without blocking the main thread.

Instead of waiting for slow operations (API calls, DB queries, file reads), Node.js **registers callbacks** and continues executing other code.

■ When/Why used

- To handle multiple requests at once
- To keep UI or server responsive
- Required for I/O operations (network, file system, timers)

■ Example

```
console.log("Start");
setTimeout(() => console.log("Async Task"), 1000);
console.log("End");
// Start, End, Async Task
```

■ Short interview line

"Asynchronous programming lets slow tasks run in the background without blocking execution."

✅ 2. What is a callback?

■ Explanation (simple, conceptual)

A callback is a **function passed as an argument** to another function and executed **after the async operation completes**.

■ When/Why used

- File operations
- API calls
- Event handling
- Timers (`setTimeout`, `setInterval`)

■ Example

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

```
  else console.log(data.toString());
});
```

■ Short interview line

“A callback runs after an async task finishes, like reading a file or hitting an API.”

✅ 3. What problem do callbacks cause (callback hell)?

■ Explanation (simple)

When callbacks are **nested inside other callbacks**, code becomes:

- Hard to read
- Hard to maintain
- Hard to debug
- Error handling becomes messy

This pattern is called **callback hell** or “pyramid of doom.”

■ When/Why

Occurs in workflows where multiple async tasks must run in sequence.

■ Example

```
getUser(id, user => {
  getOrders(user, orders => {
    getPayment(orders, payment => {
      console.log("Done");
    });
  });
});
```

■ Short interview line

“Callback hell happens when callbacks are deeply nested, making code unreadable and error-prone.”

✅ 4. What is a Promise?

■ Explanation (simple, conceptual)

A **Promise** represents the result of an asynchronous operation.

It has three states:

- **pending**

- **fulfilled**
- **rejected**

Promises avoid callback hell and support chaining.

■ When/Why used

- To run async code cleanly
- Better error handling
- Sequencing async tasks

■ Example

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done"), 1000);
});

myPromise.then(result => console.log(result));
```

■ Short interview line

"A Promise represents a future value of an async operation and helps avoid callback hell."

✳️ 5. Explain async/await with an example.

■ Explanation (simple, conceptual)

`async/await` is syntactic sugar over Promises.

It allows writing async code that **looks synchronous**, improving readability.

■ When/Why used

- Sequential async operations
- Cleaner code
- Easier error handling with try/catch

■ Example

```
async function getData() {
  try {
    const res = await fetch("https://api.example.com");
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.log("Error:", err);
  }
}
```

■ Short interview line

“Async/await makes async code look synchronous and simplifies promise-based flows.”

✳️ 6. What is error handling in async/await?

■ Explanation (simple)

You handle async/await errors using **try...catch**, because async functions return Promises.

■ When/Why used

- To prevent app crashes
- To handle rejected promises
- To manage network, database, and file errors

■ Example

```
async function fetchUser() {
  try {
    const res = await fetch("/api/user");
    if (!res.ok) throw new Error("Request failed");
    return await res.json();
  } catch (err) {
    console.error("Error:", err);
  }
}
```

■ Short interview line

“In async/await, errors are handled using try/catch because async functions return Promises.”

---

6. Express.js (Most asked if Node.js is in resume)

✅ 1. What is Express.js?

■ Explanation (simple, conceptual)

Express.js is a **lightweight, fast, unopinionated web framework** built on top of Node.js.

It provides easy APIs to build:

- REST APIs
- Web servers
- Middleware-based pipelines

■ When/Why is it used

- To simplify routing, request handling, and middleware logic
- To build scalable backend services efficiently
- To avoid writing raw Node HTTP logic

■ Example

```
const express = require("express");
const app = express();
app.get("/", (req, res) => res.send("Hello"));
app.listen(3000);
```

■ Short interview line

"Express is a minimal, middleware-based framework for building Node.js web APIs."

✅ 2. How do you create a simple REST API in Express?

■ Explanation (simple)

A REST API in Express is created by:

1. Initializing Express
2. Defining routes (`GET`, `POST`, `PUT`, `DELETE`)
3. Sending responses using `res.json()` or `res.send()`

■ When/Why is it used

- For backend services
- For CRUD operations
- For connecting frontend → backend

■ Example

```
const express = require("express");
const app = express();
app.use(express.json());

// GET API
app.get("/users", (req, res) => {
  res.json([{ id: 1, name: "Mohini" }]);
});

// POST API
app.post("/users", (req, res) => {
  res.status(201).json(req.body);
});
```

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

■ Short interview line

"A REST API in Express is created by defining routes and sending JSON responses through request handlers."

✅ 3. What is middleware?

■ Explanation (simple, conceptual)

Middleware is a **function that runs between** the request and response cycle.

It can:

- Modify `req` or `res`
- Validate data
- Authenticate users
- Log requests
- Handle errors
- Call the next middleware using `next()`

■ When/Why used

- Authentication (JWT, sessions)
- Validation
- Parsing JSON
- Error handling
- Logging all incoming requests

■ Example

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

■ Short interview line

"Middleware processes requests before sending the response — ideal for auth, validation, and logging."

✅ 4. How are errors handled in Express?

■ Explanation (simple)

Express handles errors using **error-handling middleware** — functions with **four arguments**:

`(err, req, res, next)`.

Any thrown error or rejected promise inside controllers is passed here.

■ When/Why used

- To centralize all error logging
- To send consistent error responses
- To prevent crashes on unexpected failures

■ Example

```
// Error-handling middleware
app.use((err, req, res, next) => {
  console.error("Error:", err.message);
  res.status(500).json({ error: err.message });
});

// Throw error inside route
app.get("/test", (req, res) => {
  throw new Error("Something broke!");
});
```

■ Short interview line

"Express uses a special 4-argument middleware to catch and handle application errors globally."

✅ 5. What is `app.use()` ?

■ Explanation (simple)

`app.use()` registers **middleware** or **routes** in Express.

It runs for **every request** unless a specific path is provided.

■ When/Why used

- Register middleware (logging, auth, validation)
- Mount routers
- Parse JSON
- Serve static files

■ Example

```
app.use(express.json()); // JSON parsing middleware

app.use("/users", userRouter); // Mount router
```

■ Short interview line

“app.use() adds middleware or routers to the Express pipeline.”

✅ 6. What is routing in Express?

■ Explanation (simple, conceptual)

Routing means **defining endpoints** (URLs) and associating them with handler functions.

Example:

- `/users` → fetch user list
- `/users/:id` → fetch specific user

■ When/Why used

- Organize API endpoints
- Separate features using route files
- Improve maintainability in large apps

■ Example

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

■ Short interview line

“Routing is mapping URLs to handler functions for GET/POST/PUT/DELETE operations.”

---

7. Node.js Architecture

✅ 1. Explain Node’s architecture (single-thread + libuv)

■ Explanation (simple, conceptual)

Node.js uses a **single-threaded event loop** to execute JavaScript, but it is NOT entirely single-threaded internally.

Node’s architecture has two main parts:

1️⃣ Single-threaded JavaScript Engine (V8)

- Executes JS code on a **single main thread**
- Manages call stack, microtasks, and event loop

2️⃣ libuv (C/C++ library)

- Provides a **thread pool** for handling heavy tasks
- Handles async I/O (file system, network, DNS)
- Implements the **event loop** behind the scenes

So Node appears single-threaded to the developer, but internally it uses multithreading for I/O.

■ When/Why is it used

- Simplifies development (no complex multithreading needed)
- Supports massive concurrency using event loop
- Offloads heavy I/O to background threads

■ Example (conceptual)

JS thread: runs code

libuv: handles file reads, DB queries, timers → returns callback later

■ Short interview line

"Node runs JS in a single thread but uses libuv's thread pool and event loop for non-blocking I/O."

✅ 2. Why is Node.js good for I/O-heavy applications?

■ Explanation (simple, conceptual)

I/O-heavy tasks (file read, DB queries, API calls) do NOT require CPU.

Node uses **asynchronous, non-blocking I/O**, meaning:

- It sends I/O operations to **libuv**,
- Immediately continues executing next tasks,
- Gets callbacks only when data is ready.

So Node can serve **thousands of users at once** without blocking.

■ When/Why is it used

- Chat applications
- Real-time dashboards
- APIGateways
- Microservices
- Streaming apps
- CRUD systems (Expense Tracker, Fraud Services)

■ Example

```
// request goes to background (libuv)
fs.readFile("file.txt", () => console.log("done"));

console.log("Continue serving other users...");
```

■ Short interview line

“Node is ideal for I/O-heavy apps because non-blocking I/O lets it handle thousands of requests simultaneously.”

✅ 3. What is libuv and what role does it play?

■ Explanation (simple, conceptual)

libuv is a C/C++ library used by Node.js that powers:

- The **event loop**
- **Thread pool** (4 threads by default)
- Async I/O operations
- Timers
- Networking
- File system

Node delegates heavy tasks to libuv so the JS thread never blocks.

■ When/Why is it used

- For reading files, making network calls, DNS requests
- To keep the event loop free by using background threads
- To implement Node’s entire async architecture

■ Example (conceptual)

When you do:

```
fs.readFile("file.txt", callback);
```

Steps:

1. JS thread calls libuv
2. libuv runs file read in thread pool
3. When done → callback queued
4. Event loop executes callback

■ Short interview line

“libuv provides Node’s event loop and thread pool, enabling async and non-blocking operations.”

---

8. File System (fs module)

✅ 1. Difference between synchronous and asynchronous FS operations

■ Explanation (simple, conceptual)

Operation Type	Behavior	Blocks Event Loop?	Usage
Synchronous FS (`fs.readFileSync`)	Executes immediately and waits until the task completes	❌ YES — blocks the thread	Rarely used (small scripts)
Asynchronous FS (`fs.readFile`)	Delegates work to libuv thread pool and continues execution	✔ NO — non-blocking	Recommended for servers

■ When/Why is it used

- **Synchronous**: only for CLI scripts/startup code where blocking is acceptable
- **Asynchronous**: for API servers handling multiple clients (recommended for backend)

■ Example

```
// Sync
const data = fs.readFileSync("a.txt"); // blocks
console.log("After Sync");

// Async
fs.readFile("a.txt", (err, data) => console.log(data.toString()));
console.log("After Async"); // runs immediately
```

■ Short interview line

"Synchronous FS blocks the event loop, while asynchronous FS delegates work to libuv and keeps Node responsive."

✅ 2. How do you read a file in Node.js?

■ Explanation (simple)

Node.js provides two methods:

- `fs.readFile()` → async (recommended)
- `fs.readFileSync()` → sync (blocking)

■ When/Why used

- **readFile()** → for APIs, servers, microservices
- **readFileSync()** → for quick scripts or config loading during startup

■ Example

✔ Async (recommended)

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

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

✔ Sync (blocking)

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

■ Short interview line

"You read files using fs.readFile asynchronously or fs.readFileSync if blocking is acceptable."

✅ 3. What happens if you use blocking calls?

■ Explanation (simple, conceptual)

Blocking calls (like `fs.readFileSync`, `while` loops, heavy CPU work) **pause the event loop**, meaning:

- No other requests are processed
- Server becomes unresponsive
- Slow API responses
- User experience degrades
- Node loses its performance advantage

■ When/Why

Blocking is harmful in a real backend server because Node has only **one main thread** for JS execution.

■ Example

```
// This blocks the server
const data = fs.readFileSync("bigFile.txt");
```

While this runs → Node cannot serve any other user's request.

■ Short interview line

"Blocking calls freeze the event loop, stopping Node from handling other requests — harmful for server performance."

---

9. Streams & Buffers

✅ 1. What are streams in Node.js?

■ Explanation (simple, conceptual)

Streams are **data-handling pipelines** that allow processing data **chunk-by-chunk**, instead of loading the entire data into memory.

Node provides four types of streams:

1. **Readable** (e.g., reading files)
2. **Writable** (e.g., writing files)
3. **Duplex** (read + write, e.g., TCP sockets)
4. **Transform** (modify data while streaming, e.g., compression)

■ When/Why is it used

- To handle **large files** without loading them fully
- To improve performance and memory usage
- For streaming audio/video, logs, file uploads
- For data pipelines (compression, network streaming)

■ Example

```
const fs = require("fs");
const stream = fs.createReadStream("data.txt");

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

■ Short interview line

"Streams let Node process large data chunks efficiently without loading everything into memory."

✅ 2. What is the difference between read and write streams?

■ Explanation (simple)

Stream Type	Purpose	Example
Readable Stream	Reads data chunk-by-chunk	`fs.createReadStream()`
Writable Stream	Writes data chunk-by-chunk	`fs.createWriteStream()`

■ When/Why used

- **Readable** → reading logs, videos, or big JSON files
- **Writable** → writing logs, saving uploads, exporting files

■ Example

```
// Read Stream
const readStream = fs.createReadStream("input.txt");

// Write Stream
const writeStream = fs.createWriteStream("output.txt");

// Pipe (connect both)
readStream.pipe(writeStream);
```

■ Short interview line

"Readable streams pull data chunk-by-chunk; writable streams push data chunk-by-chunk."

✅ 3. What is a buffer?

■ Explanation (simple, conceptual)

A **buffer** is a temporary memory area used to store **binary data** (bytes) before processing.

Since JavaScript normally handles strings, Buffers allow Node to handle files, videos, images, and network streams.

■ When/Why used

- File reading/writing
- Networking (TCP packets)
- Streaming audio/video
- Handling binary data

■ Example

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

■ Short interview line

"Buffers store raw binary data and are used internally by streams, file I/O, and network operations."

✅ 4. Where did you use streams in your projects?

■ Explanation (simple, conceptual)

In your **Expense Tracker** and **Fraud Detection systems**, streams were used implicitly or explicitly for handling efficiency:

✔️ Expense Tracker – Exporting large reports

When exporting transactions (CSV/Excel), a full dataset might be large. A Read Stream + Write Stream pipeline prevents loading all rows into memory.

✔️ Fraud Detection – Log processing

When handling large log files or rule-engine outputs, streams allow reading piece by piece.

✔️ Backend Microservices – File and API logs

Streaming logs ensures:

- Low memory usage
- Real-time log processing
- Continuous monitoring

■ When/Why used

- To handle **large datasets** without blocking Node
- To generate files efficiently (CSV, logs)
- To improve performance for analytics/reporting

■ Example

```
const readStream = fs.createReadStream("transactions.log");
const writeStream = fs.createWriteStream("backup.log");

readStream.pipe(writeStream);
```

■ Short interview line

"I used streams for exporting large reports and processing logs efficiently without blocking the event loop."

---

10. Security in Node.js

✅ 1. How to secure APIs created in Node.js?

■ Explanation (simple, conceptual)

Securing APIs means protecting data, preventing unauthorized access, and making the service resilient to common attacks. It's a layered approach: authenticate & authorize users, validate and sanitize input, use secure transport, apply rate limits, log & monitor, and harden runtime/configuration.

■ When / Why is it used

Always — especially for financial systems (banking/fraud). Without these controls you risk data leaks, account takeover, injection attacks, denial-of-service, and regulatory non-compliance.

■ Example / Code snippet (checklist + small examples)

Important actions (practical checklist):

- Use **HTTPS** (TLS) for all traffic.
- **Authentication**: JWT / OAuth2 / session cookies (with HttpOnly + Secure flags).
- **Authorization**: role checks / scope validation per endpoint.
- **Input validation & sanitization**: `express-validator`, Joi.
- **Parameterized queries / ORM**: avoid string concatenation for SQL.
- **Rate limiting**: limit requests per IP or user.
- **Secure headers**: `helmet()` to set CSP, HSTS, X-Frame-Options, etc.
- **CORS**: restrict origins to trusted clients.
- **CSRF protection**: tokens or SameSite cookies for browser flows.
- **Secrets management**: environment variables or vaults — never commit secrets.
- **Dependency scanning**: `npm audit`, SCA tools.
- **Logging & monitoring**: structured logs, alerts on anomalies.

Small Express wiring example:

```
const express = require('express');
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');
const { body, validationResult } = require('express-validator');

const app = express();
app.use(helmet());
app.use(express.json());

app.use(rateLimit({ windowMs: 60_000, max: 100 })); // 100 req/min per IP

app.post('/expenses',
  body('amount').isFloat({ gt:0 }),
  body('category').isString().trim().isLength({min:1}),
  (req, res) => {
    const errs = validationResult(req);
    if (!errs.isEmpty()) return res.status(400).json({ errors: errs.array() });
    // use parameterized DB call here
    res.status(201).send();
  }
);
```

■ Short interview line

“Secure APIs with TLS, strong auth/authorization, input validation, parameterized queries, secure headers, rate limits and continuous monitoring.”

✅ 2. What is Helmet?

■ Explanation (simple, conceptual)

Helmet is an Express middleware that **sets a collection of HTTP security headers** (CSP, HSTS, X-Frame-Options, X-Content-Type-Options, etc.) to protect apps from common web vulnerabilities and reduce attack surface.

■ When / Why is it used

Use Helmet for almost every HTTP API or web app — it’s low-effort and prevents many client-side attack vectors (clickjacking, MIME-sniffing, weak content policies).

■ Example / Code snippet

Basic usage:

```
const helmet = require('helmet');
app.use(helmet()); // sensible defaults

// customize CSP (example)
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", 'apis.example.com'],
  }
}));
```

■ Short interview line

“Helmet is middleware that applies secure HTTP headers (CSP, HSTS, X-Frame-Options, etc.) to harden web apps.”

✅ 3. What is rate limiting?

■ Explanation (simple, conceptual)

Rate limiting controls how many requests a client (IP, user, API key) can make in a time window. It prevents abuse, brute-force logins, and reduces risk from accidental or malicious high traffic (DoS-like behavior).

■ When / Why is it used

Used on public endpoints, auth endpoints, and expensive APIs (search, report generation, fraud-check endpoints) to protect availability and limit cost.

■ Example / Code snippet

Simple express-rate-limit example:

```
const rateLimit = require('express-rate-limit');

const apiLimiter = rateLimit({
  windowMs: 60 * 1000, // 1 minute
  max: 60,             // limit each IP to 60 requests per window
  standardHeaders: true,
  legacyHeaders: false
});

app.use('/api/', apiLimiter);
```

Advanced approaches: token-bucket, leaky-bucket, sliding window, or distributed rate-limiting in Redis for multi-node setups.

■ Short interview line

"Rate limiting caps requests per client to prevent abuse and protect service availability — often implemented with in-memory or Redis-backed token buckets."

✅ 4. How to prevent SQL injection or XSS in Node?

■ Explanation (simple, conceptual)

SQL injection happens when user input is concatenated into SQL. **XSS** happens when untrusted input is rendered into HTML without escaping. Prevention uses parameterized queries / prepared statements, ORMs, strict input validation, and output encoding/Content Security Policy.

■ When / Why is it used

Critical for any app that accepts user input (forms, APIs, query strings). For banking/fraud systems, prevention is mandatory to avoid data theft or manipulation.

■ Example / Code snippet

Prevent SQL injection

- Use parameterized queries or prepared statements — never build SQL by string concatenation.

`mysql2` parameterized example:

```
// BAD (vulnerable)
const sql = `SELECT * FROM users WHERE email = '${email}'`;

// GOOD (parameterized)
const [rows] = await db.execute(
  'SELECT * FROM users WHERE email = ?',
```

```
  [email]
);
```

With `pg` (Postgres):

```
const res = await client.query('SELECT * FROM users WHERE id = $1', [userId]);
```

Or use an ORM (Sequelize, TypeORM) that uses bindings under the hood.

Prevent XSS

- Escape or HTML-encode user content before rendering in HTML.
- Use a safe templating engine that auto-escapes (e.g., Handlebars, EJS with escaping).
- Sanitize rich HTML inputs with libraries like `DOMPurify` (frontend) or `sanitize-html` (server) and enforce CSP.

Example (escaping in server-rendered HTML):

```
const escapeHtml = require('escape-html');
res.send(`<div>${escapeHtml(userInput)}</div>`);
```

Defense-in-depth

- Validate input types/lengths (whitelisting).
- Use HTTPOnly cookies for auth to reduce XSS-stealing risk.
- Set CSP via Helmet to restrict script sources.
- Principle of least privilege for DB users (read-only where appropriate).

■ Short interview line

"Prevent SQL injection with parameterized queries/ORMs and stop XSS by validating inputs, escaping output, sanitizing HTML, and enforcing CSP/secure headers."

⭐ Scenario-Based Questions

1. If your Expense Tracker dashboard fetches data using Node, how would you write a GET API?

■ Explanation (simple, conceptual)

A GET API should be thin (controller), validate inputs, call a service layer for business logic, use a repository/DB layer for queries, and return JSON. For dashboard data prefer returning **pre-aggregated** or **paged** results to keep response sizes small and predictable.

■ When/Why is it used

Used when the frontend needs summarized or filtered expense data for charts, tables, or analytics. Pre-aggregation avoids heavy queries on every request.

■ Example / Code snippet

```
// routes/expenses.routes.js
const express = require('express');
const { getExpensesSummary } = require('../controllers/expenses.controller');
const router = express.Router();
router.get('/summary', getExpensesSummary);
module.exports = router;

// controllers/expenses.controller.js
const { fetchExpensesSummary } = require('../services/expenses.service');

async function getExpensesSummary(req, res, next) {
 try {
  const userId = req.user.id;          // authenticated user
  const { from, to, groupBy = 'day' } = req.query;
  // validate query params minimally here or via middleware
  const result = await fetchExpensesSummary(userId, { from, to, groupBy });
  res.json({ success: true, data: result });
 } catch (err) {
  next(err); // pass to centralized error handler
 }
}
module.exports = { getExpensesSummary };

// services/expenses.service.js
const { queryExpensesSummary } = require('../repositories/expenses.repo');

async function fetchExpensesSummary(userId, opts) {
 // business rules: default date range, limit, cache lookup, etc.
 return queryExpensesSummary(userId, opts);
}
```

■ Short summary line to speak in interview

“I’d expose a GET `/expenses/summary` endpoint that validates input, calls a service which returns pre-aggregated/paged results, and uses centralized error handling.”

---

2. How would you handle errors in Node.js when fetching user expenses?

■ Explanation (simple, conceptual)

Use structured error handling: validate inputs early, catch errors in controllers, and forward to a central error-handling middleware that logs context and returns safe client-facing messages (no internal stack traces). Differentiate client errors (4xx) from server errors (5xx) and implement retries/backoff for transient DB/network errors.

■ When/Why is it used

To keep the API stable, return meaningful messages to frontend, and ensure proper logging/alerts for ops teams—especially critical for financial services.

■ Example / Code snippet

```
// middleware/errorHandler.js
function errorHandler(err, req, res, next) {
  console.error({ msg: err.message, path: req.path, user: req.user?.id, stack: err.stack });
  if (err.isClient) return res.status(err.status || 400).json({ success:false, message: err.message
});
  // transient errors could have a flag to retry on client side
  res.status(500).json({ success:false, message: "Something went wrong. Try again later." });
}
module.exports = errorHandler;

// usage in app.js
app.use('/api/expenses', expensesRouter);
app.use(errorHandler);
```

Use typed errors:

```
class BadRequestError extends Error { constructor(msg){ super(msg); this.isClient = true; this.s
tatus = 400; } }
```

Also add monitoring:

- Structured logs (correlation id)
- Alerts for repeated failures
- Circuit-breakers or retry strategies for transient DB/API faults

■ Short summary line to speak in interview

“I validate early, throw typed errors, route them to a centralized error middleware that logs safely and returns user-friendly messages while monitoring and retries handle transient faults.”

---

3. If a fraud detection service has to call another service, how will you handle async operations?

■ Explanation (simple, conceptual)

Use `async/await` for readability, add request timeouts, retries with exponential backoff for transient failures, idempotency keys where side-effects occur, and circuit breakers to protect downstream services. Use `Promise.all` or limited concurrency pools for parallel calls, and `AbortController` /cancellation to avoid wasted work when request is cancelled.

■ When/Why is it used

When the fraud service needs to enrich a transaction (call risk-score API, user profile, geo-lookup), these patterns keep latency predictable and system resilient to downstream failures.

■ Example / Code snippet

```
const pTimeout = require('p-timeout'); // conceptual helper

async function checkFraud(txn) {
 try {
  // parallel but limited concurrency example (using Promise.all for small number)
  const [userProfile, riskScore] = await Promise.all([
   fetchWithTimeout(`/users/${txn.userId}`),
   fetchWithTimeout(`/risk/score?txn=${txn.id}`)
  ]);
  // evaluate rules and return decision
  return evaluateRules(txn, userProfile, riskScore);
 } catch (err) {
  // transient failure handling: retry, fallback, or mark as "needs manual review"
  if (isTransient(err)) {
   // exponential backoff retry (pseudo)
   return retry(() => checkFraud(txn), { retries: 2 });
  }
  throw err; // bubble up to orchestrator
 }
}

async function fetchWithTimeout(url) {
 return pTimeout(fetchJson(url), 2000); // fail if >2s
}
```

Include:

- Timeouts (avoid infinite waits)
- Retries with limits (to not overload)
- Circuit breaker (e.g., `opossum`) to open when failure rate high
- Fallback strategy: safe default (e.g., escalate to manual review)

■ Short summary line to speak in interview

“I use async/await with timeouts, retries/backoff, limited parallelism, and circuit-breakers plus idempotency and cancellation to keep fraud flows resilient and low-latency.”

---

4. How do you validate user input in Node/Express?

■ Explanation (simple, conceptual)

Validate at multiple layers: request-level validation (schemas) to give immediate feedback, business-rule validation in services, and DB-level constraints for final safety. Use schema validators (Joi, express-validator, Zod) and sanitize inputs to avoid injection/XSS.

■ When/Why is it used

Prevents bad data, reduces unnecessary DB calls, improves UX with clear error messages, and defends against attacks.

■ Example / Code snippet

```
// routes/expenses.routes.js
const { body } = require('express-validator');

router.post('/',
  [
    body('amount').isFloat({ gt: 0 }).withMessage('Amount must be > 0'),
    body('category').isString().trim().notEmpty(),
    body('date').isISO8601().toDate()
  ],
  validationMiddleware,
  createExpenseController
);

// middleware/validationMiddleware.js
const { validationResult } = require('express-validator');
module.exports = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) return res.status(400).json({ errors: errors.array() });
  next();
};
```

Also:

- Use parameterized DB queries / ORM
- Limit field lengths and types
- Enforce DB constraints (NOT NULL, FK, CHECK) for defense-in-depth

■ Short summary line to speak in interview

“I validate at the request layer using express-validator/Joi, sanitize inputs, and enforce DB constraints to ensure correctness and security.”

---

5. How do you structure a Node.js project (folders)?

■ Explanation (simple, conceptual)

Use a modular, layered layout that separates concerns: routes, controllers, services (business logic), repositories/DAOs (DB access), models/schemas, middleware, config, and tests. This improves testability, maintainability, and allows teams to work independently.

■ When/Why is it used

Clean separation helps in onboarding, easier unit testing, clearer ownership, and scaling codebase as features grow.

■ Example / Folder structure

```
/src
  /config        # config and env loading
  /routes        # route definitions
  /controllers    # thin controllers: validate + call service
  /services      # business logic, orchestration, transactions
  /repositories   # DB queries / ORM models
  /models        # DB schemas/types
  /middleware     # auth, validation, error handling
  /utils         # helpers
  /jobs          # background jobs / workers
  /tests         # unit & integration tests
app.js           # express bootstrap
server.js        # server startup (listen)
```

■ Short summary line to speak in interview

"I organize code into routes/controllers/services/repositories with middleware and config folders to keep concerns separated and tests simple."

---

6. How do you handle response time optimization in Node APIs?

■ Explanation (simple, conceptual)

Optimize response time by reducing work on the critical path: use indexes and pre-aggregated queries in DB, add caching (in-memory or Redis) for hot endpoints, paginate responses, stream large payloads, use connection pooling, and offload heavy processing to background jobs. Monitor with APM and tune hotspots.

■ When/Why is it used

Crucial for user-facing dashboards and fraud checks that must be low-latency; reduces user wait time and improves throughput under load.

■ Example / Code snippet / Patterns

- **DB**: ensure proper indexes, use `EXPLAIN` to fix slow queries, return only required columns.
- **Caching**:

```
// simple Redis cache pattern
const cached = await redis.get(cacheKey);
if (cached) return res.json(JSON.parse(cached));

const data = await service.fetchHeavyData();
redis.setex(cacheKey, 60, JSON.stringify(data)); // TTL 60s
res.json(data);
```

- **Background processing**: offload heavy aggregations to worker (Bull, RabbitMQ) and return immediate response with status or precomputed results.
- **Compression & HTTP**: enable gzip, use HTTP keep-alive, and set proper cache headers.
- **Connection management**: reuse DB connections and tune pool sizes.
- **Profiling & monitoring**: APM (NewRelic, Datadog), log slow queries, add circuit breakers.

■ Short summary line to speak in interview

"I optimize response times by using DB indexes/pre-aggregates, Redis caching, pagination/streaming, offloading heavy tasks to background workers, and continuous monitoring to find hotspots."

⭐ Database Integration (Node + SQL/Mongo)

1. How do you connect Node.js to MySQL / MongoDB?

■ Explanation (simple, conceptual)

- **MySQL (relational)** — connect using a DB driver (e.g., `mysql2`) or an ORM (Sequelize). You open a connection (or pool) and execute parameterized queries.
- **MongoDB (NoSQL document DB)** — connect using the official driver (`mongodb`) or an ODM (Mongoose). You create a client/connection and use collections/models to read/write documents.

■ When/Why is it used

- Use MySQL when you need relational schema, joins, ACID transactions, and structured queries.
- Use MongoDB when you need flexible schema, rapid iteration or document-based storage (but you trade off some relational guarantees).

■ Example / Code snippet

MySQL (mysql2 + pool)

```
// db/mysql.js
const mysql = require('mysql2/promise');

const pool = mysql.createPool({
  host: 'localhost',
  user: 'app_user',
  password: 'secret',
  database: 'expense_db',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0
});
```

```
module.exports = pool;

// usage
const pool = require('./db/mysql');
const [rows] = await pool.execute('SELECT * FROM expenses WHERE user_id = ?', [userId]);
```

MongoDB (mongoose)

```
// db/mongo.js
const mongoose = require('mongoose');

await mongoose.connect('mongodb://localhost:27017/expense_db', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

// define model
const ExpenseSchema = new mongoose.Schema({
  userId: Number,
  amount: Number,
  category: String,
  txnTime: Date
});
const Expense = mongoose.model('Expense', ExpenseSchema);

// usage
const docs = await Expense.find({ userId: 1 }).limit(50);
```

■ Short summary line to speak in interview

"Use `mysql2`/Sequelize for MySQL and `mongodb`/Mongoose for MongoDB — create a connection/pool and then use parameterized queries or models to perform DB operations."

---

2. What are connection pools?

■ Explanation (simple, conceptual)

A **connection pool** is a cache of open database connections that can be reused. Instead of opening/closing a new connection per request (expensive), the app checks out a connection from the pool, uses it, and returns it.

■ When/Why is it used

- To reduce connection latency and overhead.
- To control and limit DB resource usage (max concurrent connections).
- To improve throughput under load (e.g., API servers).

■ Example / Code snippet

```
// create pool (mysql2)
const pool = mysql.createPool({ connectionLimit: 10, host, user, password, database });

// using a pooled connection
const [rows] = await pool.execute('SELECT * FROM users WHERE id = ?', [id]);
```

With ORMs, pool is typically configured in the ORM config (Sequelize, TypeORM).

■ Short summary line to speak in interview

"A connection pool reuses a set of DB connections to reduce overhead and limit concurrent DB usage for better performance."

---

3. Why use an ORM (Sequelize / Mongoose)?

■ Explanation (simple, conceptual)

An **ORM/ODM** maps database structures to language objects (models), providing higher-level APIs for CRUD, validation, relations, migrations, and often safer defaults (parameterization). It reduces boilerplate, improves maintainability, and helps with portability.

■ When/Why is it used

- When you want rapid development with model validation, associations, and migrations.
- When you prefer object-oriented access (e.g., `User.findByPk(id)`) over raw SQL.
- For team productivity and less error-prone queries (avoids manual string concatenation / SQL injections).

■ Tradeoffs

- Extra abstraction can hide SQL performance pitfalls and make complex queries harder — sometimes raw SQL is needed for performance-critical paths.

■ Example / Code snippet

Sequelize (MySQL)

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('expense_db','user','pass',{ dialect:'mysql' });

const Expense = sequelize.define('Expense', {
  userId: DataTypes.INTEGER,
  amount: DataTypes.DECIMAL,
  category: DataTypes.STRING
});

await Expense.findAll({ where: { userId: 1 }, limit: 10 });
```

Mongoose (MongoDB) — already shown above.

■ Short summary line to speak in interview

“ORMs/ODMs speed development by mapping DB tables/collections to models with built-in validation and safer query APIs, though raw SQL may still be needed for hotspots.”

---

4. How do you perform CRUD operations in Node?

■ Explanation (simple, conceptual)

CRUD = Create, Read, Update, Delete. Implementation differs slightly by library/DB:

- **MySQL**: parameterized SQL queries or ORM model methods.
- **MongoDB**: driver methods (`insertOne`, `find`, `updateOne`) or Mongoose model methods (`create`, `find`, `findOneAndUpdate`, `deleteOne`).

Always use parameterization/ORM to prevent injection and return minimal fields needed.

■ Example / Code snippet

MySQL (mysql2 raw queries)

```
// CREATE
await pool.execute(
  'INSERT INTO expenses (user_id, amount, category, txn_time) VALUES (?, ?, ?, ?)',
  [userId, amount, category, txnTime]
);

// READ
const [rows] = await pool.execute('SELECT id, amount FROM expenses WHERE user_id = ? LIM
IT 20', [userId]);

// UPDATE
await pool.execute('UPDATE expenses SET amount = ? WHERE id = ?', [newAmount, expensel
d]);

// DELETE
await pool.execute('DELETE FROM expenses WHERE id = ?', [expenseId]);
```

MySQL (Sequelize)

```
await Expense.create({ userId, amount, category });
const list = await Expense.findAll({ where: { userId }, limit: 20 });
await Expense.update({ amount: newAmount }, { where: { id: expenseId } });
await Expense.destroy({ where: { id: expenseId } });
```

MongoDB (Mongoose)

```
// CREATE
await Expense.create({ userId, amount, category, txnTime });

// READ
```

```
const docs = await Expense.find({ userId }).limit(20);

// UPDATE
await Expense.updateOne({ _id: expenseId }, { $set: { amount: newAmount } });

// DELETE
await Expense.deleteOne({ _id: expenseId });
```

■ Short summary line to speak in interview

"Perform CRUD via parameterized SQL or ORM methods for MySQL, and via driver/ODM methods (`insertOne` / `find` / `updateOne`) for MongoDB — always prefer parameterization/models for security and maintainability."

⭐ Tricky Node.js Questions

✅ 1. What is middleware chaining?

■ Explanation (simple, conceptual)

Middleware chaining in Express means **multiple middleware functions execute sequentially**, one after another, during a request.

Each middleware calls `next()` → which passes control to the next middleware in the stack.

■ When/Why is it used

- To break logic into small, reusable units
- To handle logging → authentication → validation → controller in order
- To ensure clean separation of concerns

■ Example / Code snippet

```
app.use((req, res, next) => {
  console.log("1. Logging");
  next();
});

app.use((req, res, next) => {
  console.log("2. Authentication");
  next();
});

app.get("/data", (req, res) => {
  res.send("3. Controller executed");
});
```

Output sequence:

```
1. Logging
2. Authentication
3. Controller executed
```

■ Short summary line

“Middleware chaining means multiple middleware run in order using next(), forming a pipeline for request processing.”

✅ 2. Why isn’t Node.js suitable for multi-CPU heavy tasks?

■ Explanation (simple, conceptual)

Node.js runs JavaScript on a **single thread**, meaning **heavy CPU tasks block the event loop**.

While Node can use multiple threads internally (libuv), JavaScript execution still happens on one main thread.

CPU-heavy tasks (image processing, encryption, ML) freeze the event loop and delay all other requests.

■ When/Why

- Node is ideal for I/O-heavy workloads (API calls, DB queries)
- Not ideal for tasks that require high CPU computation

■ Example

```
// This loop blocks Node.js completely
for (let i = 0; i < 1e10; i++) {}
```

■ Short summary line

“Node isn’t ideal for CPU-heavy tasks because they block the event loop, preventing it from serving other requests.”

✅ 3. What are worker threads?

■ Explanation (simple, conceptual)

Worker Threads in Node.js allow running **JavaScript in separate threads**, enabling parallel execution for heavy CPU tasks.

They solve Node's single-threaded limitation by giving developers true multi-thread support for:

- CPU-heavy computations
- Parallel processing

- Background processing

■ When/Why used

- Hashing/encryption
- Image/video processing
- Large JSON parsing
- ML computation
- Data transformation

■ Example / Code snippet

```
const { Worker } = require("worker_threads");

new Worker("./heavy-task.js", { workerData: { count: 1e9 } });
```

■ Short summary line

“Worker threads enable CPU-heavy tasks to run in parallel without blocking Node’s event loop.”

✅ 4. What is clustering in Node.js?

■ Explanation (simple, conceptual)

Clustering allows Node.js to use **multiple CPU cores** by creating multiple instances of the same server, each running on a different core.

A **master process** distributes incoming requests across worker processes.

■ When/Why used

- To scale Node.js apps horizontally
- To improve throughput under high load
- To utilize all CPU cores (since one Node process uses only one)

■ Example / Code snippet

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

if (cluster.isMaster) {
  os.cpus().forEach(() => cluster.fork());
} else {
  const express = require("express");
  const app = express();
  app.get("/", (req, res) => res.send("Hello"));
```

```
  app.listen(3000);
}
```

■ Short summary line

“Clustering creates multiple Node instances across CPU cores to handle more requests in parallel.”

✅ 5. How does Node handle multiple concurrent requests if it’s single-threaded?

■ Explanation (simple, conceptual)

Node handles concurrency using:

1. **Event loop** — coordinates all async operations
2. **libuv thread pool** — offloads I/O tasks to threads
3. **Non-blocking APIs** — returns control immediately while tasks run in background
4. **Callback/Microtask queues** — schedules work efficiently

Key Idea:

Node does not create a new thread per request.

Instead, it performs I/O asynchronously, allowing thousands of connections at once.

■ When/Why used

- Ideal for high-traffic APIs
- Works extremely well for I/O-backed operations (DB calls, network requests)

■ Example (conceptual)

```
app.get("/user", async (req, res) => {
  const user = await db.query("SELECT ..."); // handled in thread pool
  res.json(user);
});
```

Node can serve **other requests** while waiting on DB operations.

■ Short summary line

“Node handles concurrency using non-blocking I/O and the event loop, allowing thousands of requests without multiple threads.”
