

# Node.js interview questions

25 questions · 12 easy · 9 medium · 4 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

## What is Node.js, and how can it be single-threaded yet handle many requests?

Easy



#runtime #libuv

Node.js is a **JavaScript runtime** built on Chrome's V8 engine, plus **libuv**, a C library that provides the event loop, async I/O and a small thread pool.

Your JavaScript runs on **one main thread**. When code starts I/O (a database query, a file read, an HTTP call), Node hands it to the operating system or the libuv thread pool and moves on. When the work finishes, its callback is queued and the event loop runs it.

So one thread can serve thousands of connections, because most of a web request is **waiting**, not computing.

- Great for: APIs, real-time apps, proxies, streaming
- Weak for: CPU-heavy work (image processing, big calculations), which blocks the thread for everyone

? Likely follow-up: Which operations use the libuv thread pool?

2

## Walk me through the phases of the Node.js event loop.

Medium



#event loop

Each turn of the loop goes through phases, each with its own queue of callbacks:

- **timers:** expired `setTimeout` / `setInterval` callbacks
- **pending callbacks:** some deferred system I/O errors
- **poll:** wait for new I/O and run I/O callbacks (most of your code runs here)
- **check:** `setImmediate` callbacks
- **close callbacks:** e.g. `socket.on('close')`

Between every callback, Node drains the microtask queues: first `process.nextTick`, then promise callbacks.

The loop exits when there are no timers, pending I/O or handles left.

#### Example

```
console.log('1 sync')
setTimeout(() => console.log('4 timers phase'), 0)
Promise.resolve().then(() => console.log('3 microtask'))
queueMicrotask(() => console.log('3b microtask'))
console.log('2 sync')
```

 **Likely follow-up:** Why can a long synchronous loop delay a 0 ms timer by seconds?

3

`process.nextTick` vs `setImmediate` vs `setTimeout(fn, 0)` ?

Medium 🔥🔥 #event loop #timers

- `process.nextTick(fn)`: runs **right after the current operation**, before promise callbacks and before the loop continues. Highest priority.
- `setImmediate(fn)`: runs in the **check** phase, right after the current poll phase.
- `setTimeout(fn, 0)`: runs in the **timers** phase, after at least 1 ms.

From the main module, the order of `setTimeout` vs `setImmediate` is not guaranteed. Inside an I/O callback, `setImmediate` always runs first.

Recursive `nextTick` calls can starve the event loop, because I/O never gets a turn. Prefer `setImmediate` or `queueMicrotask` unless you really need `nextTick`.

#### Example

```
const fs = require('node:fs')

fs.readFile(__filename, () => {
  setTimeout(() => console.log('timeout'), 0)
  setImmediate(() => console.log('immediate'))
  process.nextTick(() => console.log('nextTick'))
  Promise.resolve().then(() => console.log('promise'))
})
// nextTick, promise, immediate, timeout
```

4

## What is the difference between blocking and non-blocking code? How do you avoid blocking the event loop?

Easy



#event loop #performance

**Blocking** code makes the single JavaScript thread wait until it finishes, so no other request, timer or callback can run. **Non-blocking** code starts the work and returns immediately, with the result delivered later via a callback or promise.

Common ways to block by accident:

- sync APIs like `fs.readFileSync` or `crypto.pbkdf2Sync` inside a request handler
- huge `JSON.parse` / `JSON.stringify` calls
- heavy loops, sorting large arrays, catastrophic regexes

Fixes: use async APIs, **move CPU work to worker threads** or a separate service, split work into chunks, and stream large data instead of loading it all.



**Likely follow-up:** How would you detect event loop lag in production?

5

## When is it fine to use `fs.readFileSync`?

Easy



#fs #performance

Sync file APIs block the whole process until the disk read finishes, so **never use them while serving requests**.

They are fine when nothing else is running yet or nobody waits on you:

- loading config or certificates at **startup**
- CLI tools and build scripts

Otherwise use `fs/promises` with `await`, or streams for big files. The async versions run on the libuv thread pool, so the main thread stays free.

#### Example

```
const fs = require('node:fs')
const { readFile } = require('node:fs/promises')

const cert = fs.readFileSync('./cert.pem') // OK: once, at startup

app.get('/report', async (req, res) => {
  const data = await readFile('./report.json', 'utf8') // non-blocking
  res.type('json').send(data)
})
```

6

### CommonJS vs ES modules in Node: what are the differences?

Easy



#modules

**CommonJS** (`require` / `module.exports`) is Node's original system. Loading is **synchronous** and happens at runtime, so you can `require` conditionally.

**ES modules** (`import` / `export`) are the JavaScript standard. Imports are **static** and resolved before code runs, which enables tree-shaking, top-level `await` and live bindings.

Node picks the system by file: `.mjs` is ESM, `.cjs` is CommonJS, and `.js` follows `"type": "module"` in `package.json`.

- ESM has no `__dirname` or `require`; use `import.meta.dirname` / `import.meta.url`
- ESM can import CommonJS; recent Node versions can also `require()` synchronous ESM
- New projects should default to ESM

? Likely follow-up: What does Node cache when you require a module twice?

7

### Explain `package.json`, the lockfile and semver ranges like `^1.2.3`.

Easy



#npm

`package.json` describes the project: name, scripts, `dependencies` (needed at runtime) and `devDependencies` (only for build and test).

**Semver** is `MAJOR.MINOR.PATCH`: major = breaking changes, minor = new features, patch = fixes.

- `^1.2.3` allows `>=1.2.3 <2.0.0`
- `~1.2.3` allows `>=1.2.3 <1.3.0`
- `1.2.3` is exact

The **lockfile** (`package-lock.json`) records the exact version of every package in the tree, so every machine installs **the same dependency tree**. Commit it, and use `npm ci` in CI, which installs strictly from the lockfile.

? Likely follow-up: What is the difference between `npm install` and `npm ci`?

8

## What is `EventEmitter`?

Easy



#events

`EventEmitter` is Node's built-in **publish/subscribe** class: you register listeners with `on` (or `once`), and `emit` calls them with arguments. Streams, HTTP servers and sockets all extend it.

Key points:

- listeners run **synchronously**, in registration order, when `emit` is called
- an `'error'` event with no listener **crashes the process**
- forgetting `off` / `removeListener` leaks memory; Node warns after 10 listeners on one event

### Example

```
class Emitter {
  #listeners = new Map()
  on(evt, fn) { (this.#listeners.get(evt) ?? this.#listeners.set(evt, [])).get(evt)).push(fn) }
  emit(evt, ...args) { (this.#listeners.get(evt) ?? []).forEach((fn) => fn(...args)) }
}

const orders = new Emitter()
orders.on('created', (o) => console.log('email for', o.id))
orders.on('created', (o) => console.log('stock update for', o.id))
orders.emit('created', { id: 42 })
console.log('emit is synchronous: this prints last')
```

? Likely follow-up: Implement `once`.

9

## What are streams, and why use them?

Medium



#streams

A stream processes data **piece by piece** (chunks) instead of loading it all into memory. Four types:

- **Readable:** a source (`fs.createReadStream`, an HTTP request)
- **Writable:** a destination (a file, an HTTP response)
- **Duplex:** both (a TCP socket)
- **Transform:** modifies data passing through (gzip, encryption)

Streaming a 2 GB file uses a few KB of memory and the client starts receiving bytes immediately. Use `stream.pipeline` to connect them, because it **forwards errors and cleans up** all streams, which `.pipe()` does not.

### Example

```
const fs = require('node:fs')
const zlib = require('node:zlib')
const { pipeline } = require('node:stream/promises')

app.get('/logs', async (req, res) => {
  res.setHeader('Content-Encoding', 'gzip')
  await pipeline(
    fs.createReadStream('./big.log'),
    zlib.createGzip(),
    res,
  )
})
```

Likely follow-up: What is backpressure?

10

## What is backpressure, and how does Node handle it?

Hard



#streams #memory

Backpressure happens when a readable source produces data **faster** than the writable destination can consume it (fast disk, slow client). Without control, chunks pile up in memory until the process crashes.

Node's mechanism: `writable.write(chunk)` returns `false` when the internal buffer passes its `highWaterMark`. The producer should then stop writing until the 'drain' event.

`pipe()` and `pipeline()` do this automatically, which is the main reason to use them instead of manual `on('data') + write` loops.

#### Example

```
function writeMany(stream, lines) {
  let i = 0
  const writeNext = () => {
    while (i < lines.length) {
      const ok = stream.write(lines[i++] + '\n')
      if (!ok) return stream.once('drain', writeNext) // pause until
buffer empties
    }
    stream.end()
  }
  writeNext()
}
```

11

## What is a `Buffer` ?

Easy



#buffers #binary

A `Buffer` is a fixed-size chunk of **raw binary data**, stored outside the V8 heap. It is a subclass of `Uint8Array`.

You meet buffers whenever Node handles bytes: file reads without an encoding, TCP data, stream chunks, crypto output.

- `Buffer.from('hi', 'utf8')` converts text to bytes
- `buf.toString('base64')` converts bytes to text in an encoding
- `Buffer.alloc(n)` is zero-filled; `Buffer.allocUnsafe(n)` is faster but may contain old memory contents, so fill it before use

#### Example

```
const buf = Buffer.from('hello', 'utf8')
console.log(buf.length)           // 6 bytes, 5 characters
console.log(buf.toString('hex'))  // 68c3a96c6c6f
console.log(buf.toString('base64')) // aMOpbGxv
```

12

## How do you handle errors in Node, including uncaught exceptions and unhandled rejections?

Medium



#errors

Handle errors where you can recover:

- sync code and `await`: `try/catch`
- promises: `.catch()`
- callbacks: check the error-first argument `(err, data)`
- emitters and streams: an `'error'` listener

**Uncaught exceptions** and **unhandled rejections** (the default since Node 15) crash the process. Use `process.on('uncaughtException')` and `'unhandledRejection'` only to **log and exit**, never to keep running, because the app may be in a broken state. Let a process manager (PM2, Kubernetes) restart it.

Separate **operational errors** (network down, bad input: handle them) from **programmer errors** (bugs: crash and fix).

### Example

```
process.on('unhandledRejection', (reason) => {
  logger.error({ reason }, 'unhandled rejection')
  process.exit(1)
})
process.on('uncaughtException', (err) => {
  logger.fatal(err, 'uncaught exception')
  process.exit(1) // state is unknown: restart cleanly
})
```

**Likely follow-up:** Why is it dangerous to keep the process alive after an uncaught exception?

13

## What is an error-first callback, and how do you convert one to a promise?

Easy



#callbacks #async

The classic Node convention: the callback's **first argument is an error** (or `null`), and the result comes after: `fs.readFile(path, (err, data) => ...)`. Always check `err` first.

To use such APIs with `async` / `await`, wrap them with `util.promisify`, or use the built-in promise versions (`fs/promises`, `timers/promises`). Under the hood `promisify` resolves with the data or rejects with the error.

A classic bug with callbacks is calling `cb` twice, or forgetting to `return` after `cb(err)`, so the success path runs too.

#### Example

```
function promisify(fn) {
  return (...args) => new Promise((resolve, reject) =>
    fn(...args, (err, data) => (err ? reject(err) : resolve(data))))
}

function getUser(id, cb) {
  setTimeout(() => (id > 0 ? cb(null, { id }) : cb(new Error('bad id'))),
    10)
}

const getUserAsync = promisify(getUser)
getUserAsync(7).then((u) => console.log('user', u.id))
getUserAsync(-1).catch((e) => console.log('error:', e.message))
```

14

## How do you manage configuration and environment variables in Node?

Easy



#config

Config that changes between environments (DB URLs, API keys, ports) belongs in **environment variables**, read via `process.env`, not in code. This follows the twelve-factor app rule.

Good practice:

- locally, a `.env` file loaded with `node --env-file=.env` (built in since Node 20) or `dotenv`
- never commit .env files; commit a `.env.example` instead
- validate all config at startup (e.g. with zod) and fail fast if something is missing
- every value in `process.env` is a **string**, so parse numbers and booleans
- in production, inject secrets from a secrets manager, not files

#### Example

```
const { z } = require('zod')

const env = z.object({
  PORT: z.coerce.number().default(3000),
  DATABASE_URL: z.string().url(),
  NODE_ENV: z.enum(['development', 'test', 'production']),
}).parse(process.env) // throws at startup if invalid

module.exports = env
```

15

## Worker threads vs cluster vs child\_process: when do you use each?

Medium 🔥🔥 #concurrency #scaling

- **worker\_threads**: extra JavaScript threads **in the same process**. They can share memory (`SharedArrayBuffer`) and pass messages. Use for **CPU-heavy work** (image resizing, parsing, hashing) without blocking the main thread.
- **cluster**: forks **several Node processes** that share one server port. Use to use all CPU cores for an HTTP server. Each worker has its own memory.
- **child\_process**: runs **another program** (`spawn`, `exec`, `fork`), e.g. ffmpeg or a Python script.

Today, many teams skip `cluster` and run one process per container, scaling with Kubernetes or PM2 instead.

#### Example

```
const { Worker, isMainThread, parentPort, workerData } =
  require('node:worker_threads')

if (isMainThread) {
  const worker = new Worker(__filename, { workerData: 40 })
  worker.on('message', (r) => console.log('fib =', r))
} else {
  const fib = (n) => (n < 2 ? n : fib(n - 1) + fib(n - 2))
  parentPort.postMessage(fib(workerData)) // heavy work off the main
  thread
}
```

? Likely follow-up: What is the difference between `spawn` and `exec`?

16

## Is Node really single-threaded? What runs on the libuv thread pool?

Hard 🔥🔥 #libuv #concurrency

Your **JavaScript** runs on one thread, but Node itself uses more. libuv has a **thread pool** (default 4 threads, set with `UV_THREADPOOL_SIZE`) for operations the OS cannot do asynchronously:

- most `fs` operations
- `dns.lookup`
- CPU-heavy `crypto` (`pbkdf2`, `scrypt`, `randomBytes`)
- `zlib` compression

Network I/O (TCP, HTTP) does **not** use the pool; it uses the OS's async mechanisms (epoll, kqueue, IOCP).

So if you fire 100 `bcrypt` hashes at once, only 4 run in parallel and the rest **queue behind them**, which can also delay file reads.

17

## How do you scale a Node.js application?

Medium 🔥🔥 #scaling #pm2

**Vertically on one machine:** one process uses one core, so run several (cluster mode in **PM2**, or several containers).

**Horizontally:** run many instances behind a **load balancer** (Nginx, a cloud LB, Kubernetes service). For this to work the app must be **stateless**:

- sessions in Redis or a database, not in memory
- uploads in object storage (S3), not on local disk
- WebSockets need sticky sessions or a pub/sub adapter (Redis)

Also: cache (Redis, CDN), move slow work to **queues** (BullMQ, SQS) and background workers, and offload CPU-heavy work to worker threads or other services.

? Likely follow-up: What breaks if you keep sessions in memory and add a second instance?

18

## What is middleware in Node web frameworks?

Easy



#middleware #http

Middleware is a function that sits **between the incoming request and the final handler**. Each one can read or change the request and response, end the request, or pass control to the next one.

This forms a **pipeline** where cross-cutting concerns live in one place instead of in every route:

- logging and request IDs
- parsing JSON bodies and cookies
- authentication and authorisation
- CORS, rate limiting, compression
- error handling

In Express the signature is `(req, res, next)`; Koa and Fastify use similar ideas (Koa with `async (ctx, next)`, Fastify with hooks).

### Example

```
function compose(middlewares) {
  return (req) => {
    const run = (i) => middlewares[i]?.(req, () => run(i + 1))
    run(0)
  }
}

const logger = (req, next) => { console.log('log', req.url); next() }
const auth = (req, next) => (req.user ? next() : console.log('401'))
const handler = (req) => console.log('hello', req.user)

compose([logger, auth, handler])({ url: '/me', user: 'asha' })
compose([logger, auth, handler])({ url: '/me' })
```

19

## How do you create an HTTP server without a framework?

Easy



#http

The built-in `node:http` module gives you a server whose handler receives a **request** (a readable stream) and a **response** (a writable stream).

You must do the work frameworks normally hide: match `req.method` and `req.url` yourself, read and parse the body from the stream, set status codes and headers, and call `res.end()`.

Interviewers ask this to check you understand **what Express is built on**: it wraps these same objects with routing, middleware and helpers like `res.json()`.

#### Example

```
const http = require('node:http')

http.createServer(async (req, res) => {
  if (req.method === 'POST' && req.url === '/echo') {
    let body = ''
    for await (const chunk of req) body += chunk
    res.writeHead(200, { 'Content-Type': 'application/json' })
    return res.end(body)
  }
  res.writeHead(404).end('Not found')
}).listen(3000)
```

20

## What causes memory leaks in Node, and how do you find them?

Hard 🔥🔥 #memory #debugging

A leak is memory that stays **reachable** so the garbage collector cannot free it. Common causes:

- global or module-level caches and arrays that only grow (use an LRU with a size limit)
- event listeners added per request and never removed
- `setInterval` timers never cleared
- closures holding large objects
- unbounded in-memory queues

How to find them: watch `process.memoryUsage().heapUsed` growing over time, then take **heap snapshots** (`node --inspect` + Chrome DevTools, or `--heapsnapshot-signal`) a few minutes apart and compare which objects keep growing. Tools like clinic.js help too.

🔗 Likely follow-up: What does the heap snapshot "retainers" view tell you?

21

## How do you implement graceful shutdown?

Medium



#production #process

When the platform stops your app (deploy, scale-down), it sends **SIGTERM** and, after a grace period, SIGKILL. Graceful shutdown means finishing in-flight work instead of dropping it:

1. listen for `SIGTERM` / `SIGINT`
2. stop accepting new connections: `server.close()`
3. let in-flight requests finish (fail health checks so the load balancer stops routing)
4. close DB pools, queues and other resources
5. exit, with a `timeout that forces exit` if something hangs

### Example

```
const server = app.listen(3000)

async function shutdown(signal) {
  logger.info(`${signal} received, shutting down`)
  setTimeout(() => process.exit(1), 10_000).unref() // hard limit
  server.close(async () => {
    await db.end()
    await redis.quit()
    process.exit(0)
  })
}

process.on('SIGTERM', shutdown)
process.on('SIGINT', shutdown)
```

22

## How do you run async operations in parallel vs in sequence in Node?

Medium



#async #promises

`await` inside a loop runs operations **one after another**, which is slow when they are independent.

- `Promise.all`: run in parallel, fail fast on the first rejection
- `Promise.allSettled`: run in parallel, get every result and error
- `Promise.race` / `Promise.any`: first to settle / first to succeed (timeouts, fallbacks)

With thousands of items, unlimited parallelism can overload a DB or API, so limit concurrency (batches, or a library like `p-limit`).

#### Example

```
const fetchUser = (id) => new Promise((r) => setTimeout(() => r({ id }), 50))

async function main() {
  let t = Date.now()
  for (const id of [1, 2, 3]) await fetchUser(id)
  console.log('sequential ~', Math.round((Date.now() - t) / 50) * 50, 'ms')

  t = Date.now()
  const users = await Promise.all([1, 2, 3].map(fetchUser))
  console.log('parallel ~', Math.round((Date.now() - t) / 50) * 50, 'ms', users.length)
}
main()
```

23

### What are basic security practices for a Node.js app?

Medium 🔥👤 #security

- keep dependencies updated and run `npm audit` (or Dependabot/Snyk) to catch known vulnerabilities
- validate all input; use parameterised queries to prevent injection
- set security headers (`helmet` in Express) and configure CORS narrowly
- never run as root in containers; keep secrets out of code and logs
- add rate limiting and request body size limits
- avoid `eval`, `new Function` and `child_process.exec` with user input (command injection)
- watch for **prototype pollution** when merging user JSON into objects
- run a supported LTS version of Node

24

## How is JavaScript in Node different from JavaScript in the browser?

Easy



#runtime

Same language and engine (V8 in Chrome), different **environment**:

- Node has no `window` or `document` (no DOM); it has `process`, `Buffer`, `__dirname` (CommonJS) and `globalThis`
- Node can access the file system, network sockets, OS and child processes; browser code is **sandboxed**
- Node supports CommonJS and ESM; browsers use ESM
- you control the Node version, while browser code must work in every user's browser
- event loop details differ: Node has phases plus `process.nextTick` and `setImmediate`

Many web APIs now exist in both: `fetch`, `URL`, `AbortController`, Web Streams, `structuredClone`.

25

## A Node API has become slow. How do you investigate?

Hard



#performance #debugging

Work from measurement, not guesses:

1. **Find where time goes:** APM or tracing (OpenTelemetry, Datadog) per endpoint; is it the DB, an external API, or Node itself?
2. **Check event loop lag** (`perf_hooks.monitorEventLoopDelay`): high lag means something is **blocking the main thread**
3. **Profile CPU:** `node --cpu-prof` or `--inspect` and a flame graph to find hot functions (sync crypto, big JSON, regex)
4. **Check memory and GC:** growing heap or long GC pauses
5. **Database:** slow query log, missing indexes, N+1 queries, pool size

Then fix the biggest item: cache, add indexes, move CPU work to workers, stream large responses.