

Node.js Interview Questions and Answers

Intern to senior level

Node.js interview questions and answers from beginner to senior: single-threaded event loop, require vs import, Express middleware, process.nextTick vs setImmediate, streams, error handling, JWT vs sessions, API security and scaling.

9 questions Read online: <https://www.freedevtemplates.com/interview-questions/nodejs>

1. What is Node.js and why is it called single-threaded?

Level: intern Node.js

Node.js is a JavaScript runtime built on Chrome's V8 engine that runs your JavaScript on a single main thread and uses non-blocking I/O, so one thread can handle many concurrent connections.

Your JavaScript code runs on one thread, so there is no shared-memory race condition between your own functions. But Node.js is not single-threaded overall: the libuv library handles network I/O through the operating system's asynchronous mechanisms and uses a thread pool (four threads by default) for file system access, DNS lookups, compression and some crypto functions.

When a request arrives and needs a database result, Node starts the operation, registers a callback and moves on to other work instead of waiting. This model is excellent for I/O-heavy applications such as APIs, chat and streaming, and poor for long CPU-bound computation on the main thread, which blocks every other request.

Key points

Your JavaScript runs on one thread

libuv uses the OS and a thread pool for I/O

Great for I/O-bound work, risky for CPU-bound work

2. What is the difference between package.json and package-lock.json?

Level: intern Node.js

package.json lists your project's dependencies with allowed version ranges, while package-lock.json records the exact versions that were installed so every machine gets an identical dependency tree.

In package.json, a range such as ^4.18.0 allows any compatible 4.x update. The lock file pins the exact resolved version and the integrity hash of every package, including nested dependencies. Commit both files, and use npm ci in CI and production builds: it installs exactly what the lock file says and fails if the two files disagree, while npm install may update the lock file.

Also know the difference between dependencies (needed at runtime) and devDependencies (build and test tools), and run npm audit regularly to check for known vulnerabilities.

3. What is the difference between require and import in Node.js?

Level: junior Node.js

require is the CommonJS module system, which loads modules synchronously at runtime, while import is the ES module standard, which is statically analysed, asynchronous and supports top-level await and tree shaking.

CommonJS uses require() and module.exports; it is the default in .js files unless your package.json says otherwise. ES modules use import and export, enabled by "type": "module" in package.json or by using .mjs files. Imports are hoisted and resolved before the code runs, which allows tools to remove unused exports.

Differences to remember: ES modules require file extensions in relative imports, have no `__dirname` (use `import.meta.url`), and can only be loaded from CommonJS with `dynamic import()`. `Dynamic import()` works in both systems.

```
// CommonJS
const express = require("express");
module.exports = { createApp };

// ES modules
import express from "express";
export function createApp() {}
```

4. What is middleware in Express and how does it work?

Level: junior Node.js

Middleware are functions that receive the request, response and a next callback; they run in the order they are registered and can modify the request, end the response or pass control on with `next()`.

Each middleware can do work such as logging, parsing JSON bodies, checking authentication or setting headers, and then either sends a response or calls `next()` to continue to the next function. Order matters: a body parser must be registered before the routes that read `req.body`.

Error-handling middleware has four parameters (`err`, `req`, `res`, `next`) and is registered last. In Express 4, an error thrown inside an async handler is not caught automatically; you must catch it and call `next(err)` or use a small wrapper (Express 5 handles rejected promises for you).

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

function requireAuth(req, res, next) {
  if (!req.headers.authorization) return res.status(401).json({ error: "Unauthorized" });
  next();
}

app.get("/profile", requireAuth, (req, res) => res.json({ ok: true }));

app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ error: "Something went wrong" });
});
```

5. What is the difference between `process.nextTick`, `setImmediate` and `setTimeout` in Node.js?

Level: mid Node.js

`process.nextTick` runs its callback right after the current operation, before the event loop continues; `setTimeout` runs in the timers phase after a minimum delay; and `setImmediate` runs in the check phase right after the poll (I/O) phase.

The Node.js event loop has phases: timers, pending callbacks, poll (I/O), check (`setImmediate`) and close callbacks. The `nextTick` queue and the promise microtask queue are drained between phases, and the `nextTick` queue goes first. Recursively calling `process.nextTick` can therefore starve I/O.

From the main module, the order of `setTimeout(fn, 0)` and `setImmediate(fn)` depends on process timing and is not guaranteed. Inside an I/O callback, `setImmediate` always runs first. Prefer `setImmediate` or `queueMicrotask` for deferring work, and reserve `nextTick` for cases such as emitting an event after a constructor returns.

```
setTimeout(() => console.log("timeout"), 0);
setImmediate(() => console.log("immediate"));
Promise.resolve().then(() => console.log("promise"));
process.nextTick(() => console.log("nextTick"));
// nextTick, promise, then timeout/immediate (order of last two varies)
```

6. What are streams in Node.js and why should you use them?

Level: mid Node.js

Streams process data piece by piece as it arrives instead of loading it all into memory, which lets you handle large files and network data efficiently; they come in Readable, Writable, Duplex and Transform types.

Reading a 2 GB file with `fs.readFile` loads all 2 GB into memory and can crash the process. A readable stream emits small chunks, so memory use stays flat. Streams can be chained: read, compress, then write.

Backpressure is the key concept: when the destination is slower than the source, the stream signals the source to pause. Use `stream.pipeline()` (or `pipeline` from `stream/promises`) instead of manual `.pipe()` chains, because it handles errors and cleans up all streams properly.

```
import { pipeline } from "node:stream/promises";
import { createReadStream, createWriteStream } from "node:fs";
import { createGzip } from "node:zlib";

await pipeline(
  createReadStream("access.log"),
  createGzip(),
  createWriteStream("access.log.gz")
);
```

7. What is the difference between JWT and session-based authentication?

Level: mid Node.js

Session authentication stores the login state on the server and gives the browser a session ID cookie, while JWT authentication gives the client a signed token that contains the user's claims so the server can verify it without storing state.

Sessions are simple and easy to revoke: delete the server-side record and the user is logged out immediately, but you need shared storage such as Redis once you run several servers. A JWT is self-contained and works well across services, but you cannot invalidate a single token before it expires unless you keep a deny list or use short lifetimes with refresh tokens.

Whichever you pick, store tokens in `HttpOnly`, `Secure`, `SameSite` cookies rather than `localStorage` to limit XSS risk, hash passwords with `bcrypt` or `Argon2`, use short access-token lifetimes, and verify the signature algorithm explicitly. Authentication confirms who you are; authorisation decides what you may do.

Key points

Session: state on the server, easy to revoke

JWT: stateless, hard to revoke before expiry

Use `HttpOnly` `Secure` cookies and short token lifetimes

8. How do you secure a Node.js Express API?

Level: mid Node.js

Validate all input, use parameterised queries, add security headers with `helmet`, rate-limit requests, restrict CORS, hash passwords, keep secrets in environment variables, and keep dependencies updated

and audited.

Input: validate and sanitise every body, query and parameter with a schema library such as Zod or Joi, and never build SQL or shell commands by string concatenation, which prevents injection. Output: do not send stack traces or internal error messages to clients.

Transport and headers: use HTTPS, set helmet() for secure headers, configure CORS to allow only your own origins, and set cookies as HttpOnly, Secure and SameSite. Abuse: add rate limiting (for example express-rate-limit), limit request body size and add stricter limits on login and password reset endpoints.

Operations: store secrets in environment variables or a secret manager, run npm audit, pin dependency versions with a lock file, run the process as a non-root user and log security events.

```
import helmet from "helmet";
import rateLimit from "express-rate-limit";
import cors from "cors";

app.use(helmet());
app.use(cors({ origin: ["https://www.example.com"] }));
app.use(express.json({ limit: "100kb" }));
app.use("/api/login", rateLimit({ windowMs: 15 * 60 * 1000, max: 10 }));
```

9. How do you scale a Node.js application and avoid blocking the event loop?

Level: senior Node.js

Keep the main thread free by moving CPU-heavy work to worker threads or a job queue, and scale across cores and machines by running multiple processes (cluster or PM2) behind a load balancer with shared, external state.

One Node.js process uses one core for your JavaScript. Run one process per core with the cluster module, PM2 or by running several containers behind a load balancer. Because the processes do not share memory, session data, caches and rate-limit counters must move to an external store such as Redis.

Blocking the event loop stalls every request in that process. Common causes are synchronous file or crypto calls (fs.readFileSync, bcrypt sync versions), parsing huge JSON documents, heavy loops, and regular expressions with catastrophic backtracking. Fix them with async APIs, worker_threads for CPU work, queues such as BullMQ for background jobs, streaming instead of buffering, and input size limits. Measure event loop delay with perf_hooks.monitorEventLoopDelay() and alert on it, and make the app stateless so instances can be added or removed freely.

Key points

One process per core, behind a load balancer

Move CPU work to workers or queues

Keep state in Redis or a database, not in process memory

```
import { Worker } from "node:worker_threads";

function runInWorker(data) {
  return new Promise((resolve, reject) => {
    const worker = new Worker("./hash-worker.js", { workerData: data });
    worker.once("message", resolve);
    worker.once("error", reject);
  });
}
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/nodejs>