

Node.js

Runtime, event loop, modules, streams, scaling and npm

1

What Node.js is

Definition

Node.js is a JavaScript runtime built on **V8 + libuv**. It runs JS outside the browser with a single-threaded event loop and non-blocking I/O, so one process can serve many concurrent connections.



- Great for I/O-heavy work: APIs, real-time apps, proxies.
- Bad at long CPU work on the main thread (image processing, big loops) — it blocks every request. Use **worker_threads** or a separate service.

2

The Node event loop phases

1. **timers** – `setTimeout` / `setInterval` callbacks
2. **pending callbacks** – some deferred system callbacks
3. **poll** – wait for and run I/O callbacks
4. **check** – `setImmediate` callbacks
5. **close** – `socket.on("close")` etc.

Ordering (in a Node script)

```
setTimeout(() => console.log('timeout'), 0)
setImmediate(() => console.log('immediate'))
Promise.resolve().then(() => console.log('promise'))
process.nextTick(() => console.log('nextTick'))
console.log('sync')
// sync → nextTick → promise → timeout / immediate
// (timeout vs immediate order varies at top level; inside an I/O callback,
// immediate runs first)
```

`process.nextTick` runs before promise microtasks. Too many nextTicks can starve I/O.

CommonJS	ES Modules
<code>require()</code> / <code>module.exports</code>	<code>import</code> / <code>export</code>
Loaded synchronously at runtime	Static, analysable, top-level <code>await</code>
Default for <code>.js</code> without <code>"type": "module"</code>	<code>.mjs</code> , or <code>"type": "module"</code> in <code>package.json</code>

- `package.json`: scripts, dependencies vs devDependencies, `engines`.
- Semver `MAJOR.MINOR.PATCH`: `^1.2.3` allows minor + patch, `~1.2.3` allows patch only.
- Commit `package-lock.json`; use `npm ci` in CI for exact, repeatable installs.
- `npx` runs a package binary without installing it globally.

Read a big file without loading it all into memory

```
import { createReadStream } from 'node:fs'
import { createInterface } from 'node:readline'

const rl = createInterface({ input: createReadStream('access.log') })
let errors = 0
for await (const line of rl) {
  if (line.includes(' 500 ')) errors++
}
console.log('5xx responses:', errors)
```

- ✎ Core modules: `fs/promises`, `path`, `http`, `crypto`, `events`, `stream`, `os`, `child_process`, `worker_threads`.
- ✎ Streams process data chunk by chunk (Readable, Writable, Duplex, Transform) — use `pipeline()` for errors and back-pressure.
- ✎ `EventEmitter`: `on`, `once`, `emit` — the base of streams and servers.
- ✎ Always handle promise rejections; log `unhandledRejection` / `uncaughtException`, then exit and let the process manager restart.

Common mistake

Sync APIs in request handlers (`readFileSync`, `bcrypt.hashSync`, big `JSON.parse` loops) block every other user. Use the async versions.

<i>Tool</i>	<i>What it does</i>
cluster / PM2	one process per CPU core behind a shared port
worker_threads	run CPU-heavy work off the main thread
Load balancer + stateless app	scale horizontally across machines
Redis	shared cache, sessions, rate limits, queues (BullMQ)
Graceful shutdown	on SIGTERM stop accepting, finish in-flight requests, close DB

- ✎ Config from environment variables (12-factor); never commit secrets.
- ✎ Structured logs (pino), health checks (`/health`), metrics.
- ✎ Cron jobs: node-cron in a single instance or a queue scheduler to avoid duplicates when scaled.

Revision

Single-threaded event loop + libuv for non-blocking I/O. nextTick → promises → timers/check.
Don't block the loop; use streams for big data, worker_threads for CPU, cluster/PM2 + stateless design to scale.

Q. Is Node.js single-threaded?

A. JavaScript runs on one thread (the event loop), but libuv uses the OS and a thread pool for I/O, crypto and DNS, so I/O doesn't block.

Q. What blocks the event loop?

A. Long synchronous work: CPU-heavy loops, sync file/crypto APIs, huge JSON parsing. Every request waits.

Q. `process.nextTick` vs `setImmediate` vs `setTimeout`?

A. `nextTick` runs before other microtasks; `setImmediate` runs in the check phase after I/O; `setTimeout(0)` runs in the timers phase.

Q. What are streams and why use them?

A. Chunk-by-chunk processing of data (files, HTTP bodies) with low memory and back-pressure handling.

Q. CommonJS vs ES Modules?

A. CommonJS uses `require/module.exports` and loads synchronously; ESM uses `import/export`, is statically analysable and supports top-level `await`.

Q. How do you scale a Node app?

A. `cluster`/PM2 to use all cores, stateless servers behind a load balancer, Redis for shared state, `worker_threads` or queues for CPU work.

Q. `dependencies` vs `devDependencies`?

A. `dependencies` are needed at runtime; `devDependencies` only for building and testing.

Q. `^` vs `~` in package versions?

A. `^1.2.3` allows minor and patch updates; `~1.2.3` allows patch updates only.

Q. What is graceful shutdown?

A. On `SIGTERM`: stop accepting connections, finish in-flight requests, close DB connections, then exit.