Node.js

Version 24 LTS · verified 2026-07-28

A gradual, compact walk through modern Node.js — modules, HTTP, streams, errors, testing, and worker threads.

Contents

Mental model

Node runs your JavaScript on one thread driven by an event loop. I/O is handed to the operating system and libuv, so one process serves thousands of concurrent connections happily — but any synchronous CPU work freezes every one of them until it finishes. Modern Node ships as batteries-included: an HTTP client, a test runner, `.env` loading, watch mode, and TypeScript stripping are all built in, so reach for a dependency only after checking whether the runtime already has it.

Modules: ESM & CommonJS

Syntax	Meaning
`"type": "module"`	`.js` files are ESM
`import x from "y"`	ESM, statically resolved
`require("y")`	CJS — now loads ESM too
`node:fs`	always the builtin, never a package
`import.meta.dirname`	ESM's `__dirname`, 20.11+

ESM is the default for new projects: set `"type": "module"` in `package.json`. Always prefix builtins with `node:` — it resolves faster and can never be shadowed by a package of the same name.

```
import { readFile } from "node:fs/promises";


console.log(typeof readFile);
// function
```

```
console.log(import.meta.dirname !== undefined);
// true
```

> **Gotcha:** `__dirname` and `__filename` do not exist in ESM — reading one throws `ReferenceError`. Use `import.meta.dirname` and `import.meta.filename`.

Running code: flags, env & TypeScript

Flag	Effect
`--watch`	restart on file change
`--env-file=.env`	load env vars, 20.6+
`--test`	run the built-in test runner
`node app.ts`	strip types and run, no build
`--no-strip-types`	opt out of TypeScript handling

Node reads `.env` and restarts on change without `dotenv` or `nodemon`. It also runs `.ts` files directly by erasing type annotations — stable since 24.12 — though it never type-checks, so keep `tsc --noEmit` in CI.

```
// node --env-file=.env --watch app.js
const port = process.env.PORT ?? 3000;
const [, , cmd = "serve"] = process.argv;
console.log(cmd, port);
// serve 3000
```

> **Gotcha:** type stripping only erases; it cannot compile. `enum`, parameter properties, and decorators throw `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX`.

Filesystem & paths

API	Use
`node:fs/promises`	async fs you can `await`
`readFile(p, "utf8")`	returns a string, not a Buffer
`join(a, b)`	build paths, OS-correct
`import.meta.dirname`	directory of the current file
`fs.glob(pat)`	match files by pattern, 22+

Use the promises API and `await` it; the callback forms exist for legacy code. Never concatenate path strings — `join` handles separators and normalization.

```
import * as fs from "node:fs/promises";
import { join } from "node:path";

const dir = import.meta.dirname;
const p = join(dir, "data.json");

await fs.writeFile(p, '{"ok":true}');
const raw = await fs.readFile(p, "utf8");
```

```
console.log(JSON.parse(raw).ok);
// true
```

> **Gotcha:** omit the encoding and `readFile` resolves to a `Buffer`, so `JSON.parse` still works but string methods behave unexpectedly. Pass `"utf8"` explicitly.

HTTP: server & fetch client

API	Use
`createServer(fn)`	handle requests, `node:http`
`res.setHeader(k, v)`	set before writing the body
`res.end(body)`	required — or the request hangs
`fetch(url)`	global client, no dependency
`res.ok`	`false` on any 4xx or 5xx

`node:http` is the foundation every framework builds on. `fetch` is global and needs no package. Listening on port `0` picks a free port, which is what tests should do.

```
import { createServer } from "node:http";

const srv = createServer((req, res) => {
  res.setHeader("content-type", "text/plain");
  res.end(`you asked for ${req.url}`);
});
```

```
srv.listen(0, async () => {
  const { port } = srv.address();
  const url = `http://localhost:${port}/hi`;
  const body = await (await fetch(url)).text();
  console.log(body);  // you asked for /hi
  srv.close();
});
```

> **Gotcha:** `fetch` rejects only on network failure. A 404 or 500 resolves normally — check `res.ok` yourself or you will parse an error page as your payload.

Streams & backpressure

API	Use
`Readable.from(iterable)`	turn data into a stream
`pipeline(a, b)`	connect and propagate errors
`node:stream/promises`	the awaitable `pipeline`
`for await (const c of s)`	consume a readable

Streams process data in chunks so memory stays flat regardless of payload size. `pipeline` wires stages together, applies backpressure, and destroys every stage if one fails.

```
import { pipeline } from "node:stream/promises";
import { Readable } from "node:stream";
import { createWriteStream } from "node:fs";

await pipeline(
  Readable.from(["a", "b", "c"]),
  createWriteStream("out.txt"),
);
console.log("written");
```

> **Gotcha:** `a.pipe(b)` does not forward errors — a failure in `b` goes unhandled and leaks `a`. Use `pipeline`, which cleans up both ends.

Errors & the process lifecycle

API	Use
unhandled rejection	crashes the process, exit code 1
`process.exitCode = 1`	fail without exiting now
`process.on("SIGTERM")`	drain before shutdown
`new Error(m, { cause })`	keep the original error
`AbortSignal.timeout(ms)`	cancel a slow operation

An unhandled promise rejection terminates the process — it is a crash, not a warning. Containers send `SIGTERM` before `SIGKILL`, so handle it to stop accepting connections and finish in-flight requests.

```
process.on("SIGTERM", () => {
  console.log("draining");
  process.exitCode = 0;
});

try {
  try {
    JSON.parse("{oops");
  } catch (cause) {
    throw new Error("bad config", { cause });
  }
} catch (err) {
  console.log(err.message, err.cause.name);
  // bad config SyntaxError
}
```

> **Gotcha:** `process.exit()` discards pending writes, so logs and responses can vanish mid-flight. Set `process.exitCode` and let the loop drain instead.

Testing with node:test

API	Use
`node --test`	discover and run test files
`test(name, fn)`	a single test

API	Use
`describe` / `it`	grouped style
`node:assert/strict`	strict-equality assertions
`mock.fn()`	spy or stub a function
`--test --watch`	rerun on change

The built-in runner has been stable since Node 20 and covers most needs with no dependency. It discovers `*.test.js` and files under `test/`, and exits non-zero if anything fails.

```
import test from "node:test";
import assert from "node:assert/strict";

test("adds", () => {
  assert.equal(1 + 1, 2);
});

test("async work", async () => {
  const v = await Promise.resolve(7);
  assert.equal(v, 7);
});
```

> **Tip:** import `node:assert/strict`, not `node:assert` — the default export's `equal` uses `==`, so `assert.equal(1, "1")` quietly passes.

Offloading CPU work

API	Use
`node:worker_threads`	CPU work off the main loop
`new Worker(file)`	spawn a thread
`parentPort.postMessage(v)`	send a result back
`node:cluster`	fork one process per core
`os.availableParallelism()`	how many to spawn

Threads are for CPU-bound work only — I/O is already concurrent without them. Workers share no variables; messages are structured-cloned between them.

```
import * as wt from "node:worker_threads";


if (wt.isMainThread) {
  const w = new wt.Worker(import.meta.filename);
  w.on("message", (m) => console.log("got", m));
} else {
  let total = 0;
  for (let i = 0; i < 1e6; i++) total += i;
  wt.parentPort.postMessage(total);
}
// got 499999500000
```

> **Gotcha:** a tight synchronous loop on the main thread delays every timer and request behind it — a 300 ms block makes a 10 ms timer fire ~300 ms late.

Further reading

- Node.js API documentation
- Node.js — TypeScript support
- Node.js — test runner
- Node.js — stream
- Node.js release schedule