



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	2
Modules: ESM & CommonJS	2
Running code: flags, env & TypeScript	3
Filesystem & paths	5
HTTP: server & fetch client	6
Streams & backpressure	8
Errors & the process lifecycle	10
Testing with node:test	11
Offloading CPU work	13
Further reading	14

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
<code>"type": "module"</code>	<code>.js</code> files are ESM
<code>import x from "y"</code>	ESM, statically resolved
<code>require("y")</code>	CJS – now loads ESM too
<code>node:fs</code>	always the builtin, never a package
<code>import.meta.dirname</code>	ESM's <code>__dirname</code> , 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

Flag	Effect
<code>--env-file=.env</code>	load env vars, 20.6+
<code>--test</code>	run the built-in test runner
<code>node app.ts</code>	strip types and run, no build
<code>--no-strip-types</code>	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
<code>node:fs/promises</code>	async fs you can <code>await</code>
<code>readFile(p, "utf8")</code>	returns a string, not a Buffer
<code>join(a, b)</code>	build paths, OS-correct
<code>import.meta.dirname</code>	directory of the current file
<code>fs.glob(pat)</code>	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
<code>createServer(fn)</code>	handle requests, <code>node:http</code>
<code>res.setHeader(k, v)</code>	set before writing the body

API	Use
<code>res.end(body)</code>	required – or the request hangs
<code>fetch(url)</code>	global client, no dependency
<code>res.ok</code>	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
<code>Readable.from(iterable)</code>	turn data into a stream
<code>pipeline(a, b)</code>	connect and propagate errors
<code>node:stream/promises</code>	the awaitable pipeline
<code>for await (const c of s)</code>	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
<code>process.exitCode = 1</code>	fail without exiting now
<code>process.on("SIGTERM")</code>	drain before shutdown
<code>new Error(m, { cause })</code>	keep the original error
<code>AbortSignal.timeout(ms)</code>	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
<code>node --test</code>	discover and run test files
<code>test(name, fn)</code>	a single test
<code>describe / it</code>	grouped style

API	Use
<code>node:assert/strict</code>	strict-equality assertions
<code>mock.fn()</code>	spy or stub a function
<code>--test --watch</code>	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
<code>node:worker_threads</code>	CPU work off the main loop
<code>new Worker(file)</code>	spawn a thread
<code>parentPort.postMessage(v)</code>	send a result back
<code>node:cluster</code>	fork one process per core
<code>os.availableParallelism()</code>	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