100 Node.js Coding Snippets

1) Basic “Hello World” HTTP server

```
import http from "node:http";

http.createServer((req, res) => {
  res.writeHead(200, {"Content-Type": "text/plain"});
  res.end("Hello from Node!\n");
}).listen(3000);
```

What it does: Starts a basic HTTP server on port 3000.
Use it for: Quick sanity checks, learning request/response basics.

---

2) Minimal server with routing (no frameworks)

```
import http from "node:http";

http.createServer((req, res) => {
  if (req.url === "/") return res.end("Home");
  if (req.url === "/health") return res.end("OK");
  res.writeHead(404); res.end("Not Found");
}).listen(3000);
```

What it does: Routes by URL with if statements.
Use it for: Tiny services, prototypes, understanding routing fundamentals.

---

3) Read JSON body from a request

```
import http from "node:http";

function readJson(req) {
  return new Promise((resolve, reject) => {
    let data = "";
    req.on("data", chunk => data += chunk);
    req.on("end", () => {
```

```
      try { resolve(JSON.parse(data || "{}")); }
      catch (e) { reject(e); }
    });
  });
}

http.createServer(async (req, res) => {
  if (req.method === "POST" && req.url === "/echo") {
    try {
      const body = await readJson(req);
      res.setHeader("Content-Type", "application/json");
      return res.end(JSON.stringify(body));
    } catch {
      res.writeHead(400); return res.end("Invalid JSON");
    }
  }
  res.writeHead(404); res.end("Not Found");
}).listen(3000);
```

What it does: Buffers incoming chunks, parses JSON safely.
Use it for: Building APIs without Express.

---

4) Parse query string parameters

```
import http from "node:http";
import { URL } from "node:url";

http.createServer((req, res) => {
  const url = new URL(req.url, "http://localhost");
  const name = url.searchParams.get("name") ?? "friend";
  res.end(`Hi ${name}`);
}).listen(3000);
```

What it does: Extracts query params like `?name=Lars`.
Use it for: Search/filter endpoints.

---

5) Serve a static file

```
import http from "node:http";
import fs from "node:fs";
import path from "node:path";

http.createServer((req, res) => {
  if (req.url === "/") {
    const filePath = path.join(process.cwd(), "index.html");
    fs.createReadStream(filePath)
      .on("error", () => { res.writeHead(404); res.end("Missing
file"); })
      .pipe(res);
    return;
  }
  res.writeHead(404); res.end("Not Found");
}).listen(3000);
```

What it does: Streams `index.html` to the browser.
Use it for: Simple demo servers.

---

6) Return JSON response

```
import http from "node:http";

http.createServer((req, res) => {
  const payload = { ok: true, time: new Date().toISOString() };
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify(payload));
}).listen(3000);
```

What it does: Sends JSON with correct header.
Use it for: API endpoints.

---

7) Environment variables + fallback defaults

```
const PORT = Number(process.env.PORT ?? 3000);
const NODE_ENV = process.env.NODE_ENV ?? "development";

console.log({ PORT, NODE_ENV });
```

What it does: Reads env vars safely with defaults.
Use it for: Configuring servers across dev/staging/prod.

---

8) Simple logger middleware pattern (framework-free)

```
function withLogging(handler) {
  return (req, res) => {
    const start = Date.now();
    res.on("finish", () => {
      console.log(req.method, req.url, res.statusCode,
`${Date.now()-start}ms`);
    });
    handler(req, res);
  };
}
```

What it does: Wraps a handler to log timing/status.
Use it for: Observability in any Node server.

---

9) Handle uncaught errors safely

```
process.on("unhandledRejection", (err) => {
  console.error("Unhandled rejection:", err);
});

process.on("uncaughtException", (err) => {
  console.error("Uncaught exception:", err);
  process.exit(1);
});
```

What it does: Catches unexpected errors and logs them.
Use it for: Prevent silent crashes (still fix root cause).

---

10) Read a text file (async/await)

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

const text = await readFile("notes.txt", "utf8");
console.log(text);
```

What it does: Reads file contents asynchronously.
Use it for: Config files, templates, data loading.

---

11) Write a file (atomic-ish pattern)

```
import { writeFile, rename } from "node:fs/promises";

const tmp = "data.json.tmp";
await writeFile(tmp, JSON.stringify({ saved: true }, null, 2),
"utf8");
await rename(tmp, "data.json");
```

What it does: Writes to temp, then renames (reduces partial writes).
Use it for: Saving state safely.

---

12) Append to a log file

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

await appendFile("app.log", `[${new Date().toISOString()}] started\n`,
"utf8");
```

What it does: Adds text to end of file.
Use it for: Lightweight logging.

---

13) List files in a directory

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

const files = await readdir("./src", { withFileTypes: true });
console.log(files.filter(f => f.isFile()).map(f => f.name));
```

What it does: Reads directory entries and filters by type.
Use it for: Scanning folders for builds/tools.

---

14) Ensure a directory exists

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

await mkdir("./output", { recursive: true });
```

What it does: Creates folder(s) if missing.
Use it for: Exports, generated files, cache dirs.

---

15) Join paths safely across OSes

```
import path from "node:path";

const fullPath = path.join(process.cwd(), "data", "users.json");
console.log(fullPath);
```

What it does: Handles `\` vs `/` differences properly.
Use it for: Any file path work.

---

16) Read a JSON file safely

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

```
async function readJsonFile(file) {
  const raw = await readFile(file, "utf8");
  return JSON.parse(raw);
}

const config = await readJsonFile("config.json");
console.log(config);
```

What it does: Reads and parses JSON from disk.
Use it for: App configs, fixtures, data.

---

17) Create a simple CLI argument parser

```
const args = process.argv.slice(2);
const flags = Object.fromEntries(args.map(a => {
  const [k, v="true"] = a.replace(/^--/, "").split("=");
  return [k, v];
}));

console.log(flags);
```

What it does: Parses `node app.js --name=Lars --debug`.
Use it for: Tiny CLI tools without dependencies.

---

18) Prompt the user in terminal

```
import readline from "node:readline";

const rl = readline.createInterface({ input: process.stdin, output:
process.stdout });

const answer = await new Promise(resolve => rl.question("Your name? ",
resolve));
rl.close();
```

```
console.log(`Hello ${answer}`);
```

What it does: Reads user input interactively.
Use it for: CLI wizards, setup scripts.

---

19) Exit with a specific error code

```
console.error("Something went wrong");
process.exit(2);
```

What it does: Stops the process and signals failure.
Use it for: CI scripts, CLIs.

---

20) Measure performance timing

```
console.time("job");
await new Promise(r => setTimeout(r, 250));
console.timeEnd("job");
```

What it does: Prints elapsed time for labeled block.
Use it for: Quick profiling.

---

21) Create an EventEmitter

```
import { EventEmitter } from "node:events";

const bus = new EventEmitter();
bus.on("saved", (id) => console.log("Saved:", id));
bus.emit("saved", 123);
```

What it does: Publishes/subscribes to events.
Use it for: Decoupling modules, internal app events.

---

22) Debounce a function (useful for events)

```
function debounce(fn, ms=200) {
  let t;
  return (...args) => {
    clearTimeout(t);
    t = setTimeout(() => fn(...args), ms);
  };
}
```

What it does: Delays execution until activity stops.
Use it for: Rate-limiting frequent triggers.

---

23) Throttle a function

```
function throttle(fn, ms=200) {
  let last = 0;
  return (...args) => {
    const now = Date.now();
    if (now - last >= ms) { last = now; fn(...args); }
  };
}
```

What it does: Runs at most once every `ms`.
Use it for: Preventing log spam, limiting expensive work.

---

24) Basic cron-like interval job

```
setInterval(() => {
  console.log("Running job:", new Date().toISOString());
}, 60_000);
```

What it does: Runs every minute.
Use it for: Polling, cleanup jobs (simple cases).

---

25) Promise timeout wrapper

```
function withTimeout(promise, ms=2000) {
  return Promise.race([
    promise,
    new Promise((_, rej) => setTimeout(() => rej(new
Error("Timeout")), ms))
  ]);
}
```

What it does: Fails a promise if it takes too long.
Use it for: Network calls, long tasks.

---

26) Retry wrapper with backoff

```
async function retry(fn, tries=3, delay=200) {
  let err;
  for (let i = 0; i < tries; i++) {
    try { return await fn(); }
    catch (e) { err = e; await new Promise(r => setTimeout(r, delay *
(i+1))); }
  }
  throw err;
}
```

What it does: Retries a failing async operation.
Use it for: Flaky network/database calls.

---

27) Make an HTTP GET request (built-in fetch)

```
const res = await fetch("https://api.github.com");
```

```
console.log(res.status, await res.text());
```

What it does: Calls a URL using Node's fetch.
Use it for: APIs, integrations, web scraping (respect terms).

---

28) JSON fetch helper

```
async function fetchJson(url, opts) {
  const res = await fetch(url, opts);
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  return res.json();
}
```

What it does: Fetches and validates response, parses JSON.
Use it for: Cleaner API calls.

---

29) POST JSON with fetch

```
const res = await fetch("https://example.com/api", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name: "Lars" })
});
console.log(res.status);
```

What it does: Sends JSON payload to an endpoint.
Use it for: Writing API clients.

---

30) URL building safely

```
const u = new URL("https://example.com/search");
u.searchParams.set("q", "node js");
u.searchParams.set("page", "2");
console.log(u.toString());
```

What it does: Prevents broken query strings/encoding bugs.
Use it for: API requests, links.

---

31) Basic TCP server

```
import net from "node:net";

net.createServer((socket) => {
  socket.write("Welcome!\n");
  socket.on("data", d => socket.write(`Echo: ${d}`));
}).listen(4000);
```

What it does: Listens for raw TCP connections.
Use it for: Custom protocols, learning networking.

---

32) Create a WebSocket server (requires `ws`)

```
// npm i ws
import { WebSocketServer } from "ws";

const wss = new WebSocketServer({ port: 8080 });
wss.on("connection", (ws) => {
  ws.send("hello");
  ws.on("message", (msg) => ws.send(`echo: ${msg}`));
});
```

What it does: Real-time push messaging.
Use it for: Live dashboards, chat, multiplayer games.

---

33) Hash a password (requires `bcrypt`)

```
// npm i bcrypt
```

```
import bcrypt from "bcrypt";

const hash = await bcrypt.hash("secret", 12);
const ok = await bcrypt.compare("secret", hash);
console.log({ hash, ok });
```

What it does: Securely hashes and verifies passwords.
Use it for: Authentication systems.

---

34) Create a JWT (requires `jsonwebtoken`)

```
// npm i jsonwebtoken
import jwt from "jsonwebtoken";

const token = jwt.sign({ userId: 123 }, process.env.JWT_SECRET, {
expiresIn: "1h" });
console.log(token);
```

What it does: Creates a signed token for auth.
Use it for: Session-less APIs.

---

35) Verify a JWT

```
import jwt from "jsonwebtoken";

const payload = jwt.verify(token, process.env.JWT_SECRET);
console.log(payload);
```

What it does: Validates token signature + expiration.
Use it for: Protecting routes.

---

36) Simple Express server

```
// npm i express
import express from "express";

const app = express();
app.get("/health", (_, res) => res.json({ ok: true }));
app.listen(3000);
```

What it does: Minimal Express app with a route.
Use it for: Most Node web APIs.

---

37) Express JSON body parsing

```
import express from "express";
const app = express();

app.use(express.json());
app.post("/echo", (req, res) => res.json(req.body));
app.listen(3000);
```

What it does: Parses incoming JSON into `req.body`.
Use it for: REST APIs.

---

38) Express error handler

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

What it does: Centralizes error responses.
Use it for: Consistent API behavior.

---

39) Serve static files in Express

```
import express from "express";
const app = express();

app.use(express.static("public"));
app.listen(3000);
```

What it does: Serves `public/*` as web assets.
Use it for: Hosting frontend builds.

---

40) CORS in Express (requires `cors`)

```
// npm i cors
import cors from "cors";
app.use(cors({ origin: "https://your-site.com" }));
```

What it does: Allows cross-site requests.
Use it for: Frontend calling your API.

---

41) Rate limiting (requires `express-rate-limit`)

```
// npm i express-rate-limit
import rateLimit from "express-rate-limit";

app.use(rateLimit({ windowMs: 60_000, max: 100 }));
```

What it does: Limits requests per IP.
Use it for: Basic abuse protection.

---

42) Validate input with Zod

```
// npm i zod
import { z } from "zod";
```

```
const schema = z.object({ email: z.string().email() });
const parsed = schema.parse({ email: "test@example.com" });
console.log(parsed);
```

What it does: Validates and parses data with good errors.
Use it for: Request validation, config validation.

---

43) Read config from `.env` (dotenv)

```
// npm i dotenv
import "dotenv/config";

console.log(process.env.DB_URL);
```

What it does: Loads environment variables from `.env`.
Use it for: Local dev configuration.

---

44) Connect to MongoDB (mongodb driver)

```
// npm i mongodb
import { MongoClient } from "mongodb";

const client = new MongoClient(process.env.MONGO_URL);
await client.connect();
const db = client.db("app");
const users = db.collection("users");
console.log(await users.findOne({}));
await client.close();
```

What it does: Connects and queries MongoDB.
Use it for: Document databases, flexible schemas.

---

45) Postgres query (pg)

```
// npm i pg
import pg from "pg";
const { Pool } = pg;

const pool = new Pool({ connectionString: process.env.DATABASE_URL });
const { rows } = await pool.query("SELECT now() AS time");
console.log(rows[0]);
```

What it does: Runs SQL in Postgres.
Use it for: Relational data and transactions.

---

46) SQLite quick DB (better-sqlite3)

```
// npm i better-sqlite3
import Database from "better-sqlite3";

const db = new Database("app.db");
db.exec("CREATE TABLE IF NOT EXISTS notes(id INTEGER PRIMARY KEY, text TEXT)");
db.prepare("INSERT INTO notes(text) VALUES (?)").run("hello");
console.log(db.prepare("SELECT * FROM notes").all());
```

What it does: Embedded database, no server needed.
Use it for: Desktop tools, prototypes, local caching.

---

47) Create a simple in-memory cache

```
const cache = new Map();

function getCached(key, compute) {
  if (cache.has(key)) return cache.get(key);
  const val = compute();
  cache.set(key, val);
  return val;
}
```

What it does: Stores computed values by key.
Use it for: Expensive computations, repeated lookups.

---

48) TTL cache (expires entries)

```
const cache = new Map();

function setTTL(key, value, ms) {
  cache.set(key, value);
  setTimeout(() => cache.delete(key), ms).unref?.();
}
```

What it does: Deletes cached entries after time.
Use it for: Token caching, short-lived data.

---

49) Stream a large file download

```
import fs from "node:fs";
import http from "node:http";

http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "application/octet-stream" });
  fs.createReadStream("big.zip").pipe(res);
}).listen(3000);
```

What it does: Streams file without loading into memory.
Use it for: Large downloads, media serving.

---

50) Compress response with gzip (built-in)

```
import http from "node:http";
import zlib from "node:zlib";
```

```
http.createServer((req, res) => {
  const gz = zlib.createGzip();
  res.writeHead(200, { "Content-Encoding": "gzip" });
  gz.pipe(res);
  gz.end("Compressed response!");
}).listen(3000);
```

What it does: Sends gzip-compressed text.
Use it for: Bandwidth savings.

---

51) Create a readable stream from text

```
import { Readable } from "node:stream";

const stream = Readable.from(["one\n", "two\n", "three\n"]);
stream.pipe(process.stdout);
```

What it does: Streams chunks as if it were a file.
Use it for: Streaming generated content.

---

52) Pipeline streams with error handling

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

await pipeline(
  fs.createReadStream("input.txt"),
  zlib.createGzip(),
  fs.createWriteStream("input.txt.gz")
);
```

What it does: Safely connects streams and propagates errors.
Use it for: File transforms, compression, ETL.

---

53) CSV parsing (csv-parse)

```
// npm i csv-parse
import { parse } from "csv-parse/sync";

const records = parse("a,b\n1,2\n3,4\n", { columns: true });
console.log(records);
```

What it does: Parses CSV into objects.
Use it for: Import tools, data pipelines.

---

54) Generate UUIDs (built-in)

```
import { randomUUID } from "node:crypto";
console.log(randomUUID());
```

What it does: Creates a unique identifier.
Use it for: IDs, tokens, correlation IDs.

---

55) Hash content with SHA-256

```
import { createHash } from "node:crypto";

const hash = createHash("sha256").update("hello").digest("hex");
console.log(hash);
```

What it does: Creates a checksum/hash.
Use it for: Integrity checks, deduping, fingerprints.

---

56) Encrypt/decrypt (AES-GCM) (concept demo)

```
import { randomBytes, createCipheriv, createDecipheriv } from
"node:crypto";

const key = randomBytes(32);
const iv = randomBytes(12);

function encrypt(text) {
  const cipher = createCipheriv("aes-256-gcm", key, iv);
  const enc = Buffer.concat([cipher.update(text, "utf8"),
cipher.final()]);
  return { enc, tag: cipher.getAuthTag() };
}
function decrypt(enc, tag) {
  const decipher = createDecipheriv("aes-256-gcm", key, iv);
  decipher.setAuthTag(tag);
  return Buffer.concat([decipher.update(enc),
decipher.final()]).toString("utf8");
}
```

What it does: Demonstrates authenticated encryption.
Use it for: Protecting secrets at rest (be careful with key/iv management).

---

57) Generate a secure random token

```
import { randomBytes } from "node:crypto";
const token = randomBytes(32).toString("hex");
console.log(token);
```

What it does: Creates a cryptographically strong token.
Use it for: Password resets, API keys (store hashed).

---

58) Validate password strength (simple)

```
function isStrong(pw) {
```

```
  return pw.length >= 12 && /[A-Z]/.test(pw) && /[a-z]/.test(pw) &&
/\d/.test(pw);
}
```

What it does: Checks basic complexity rules.
Use it for: Signup validation (prefer zxcvbn for better).

---

59) Simple file upload server (raw multipart is hard—use multer)

```
// npm i express multer
import express from "express";
import multer from "multer";

const upload = multer({ dest: "uploads/" });
const app = express();

app.post("/upload", upload.single("file"), (req, res) => {
  res.json({ savedAs: req.file.filename, original:
req.file.originalname });
});
app.listen(3000);
```

What it does: Handles multipart uploads easily.
Use it for: Upload endpoints.

---

60) Basic unit test with node:test

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

test("adds numbers", () => {
  assert.equal(2 + 3, 5);
});
```

What it does: Runs a test without extra libraries.
Use it for: Simple testing built-in.

---

61) Mock timers quickly (simple approach)

```
await new Promise(r => setTimeout(r, 50)); // test waits
```

What it does: Delays in async tests.
Use it for: Timing-sensitive code (better: fake timers in a test runner).

---

62) Read process memory usage

```
console.log(process.memoryUsage());
```

What it does: Shows heap/ RSS usage.
Use it for: Investigating leaks/perf.

---

63) Read CPU info

```
import os from "node:os";
console.log(os.cpus().length);
```

What it does: Gets number of CPU cores.
Use it for: Choosing concurrency limits.

---

64) Worker threads (CPU-heavy tasks)

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

const worker = new Worker(new URL("./worker.js", import.meta.url), {
type: "module" });
worker.postMessage({ n: 40 });
worker.on("message", msg => console.log("Result:", msg));
```

What it does: Offloads heavy CPU work off main thread.
Use it for: Image processing, big computations.

---

65) Cluster mode (multi-process web server)

```
import cluster from "node:cluster";
import os from "node:os";
import http from "node:http";

if (cluster.isPrimary) {
  for (let i = 0; i < os.cpus().length; i++) cluster.fork();
} else {
  http.createServer((_, res) => res.end("ok")).listen(3000);
}
```

What it does: Spawns workers to use multiple cores.
Use it for: Scaling CPU-bound or high traffic servers.

---

66) Graceful shutdown pattern

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

process.on("SIGTERM", () => {
  console.log("Shutting down...");
  server.close(() => process.exit(0));
});
```

What it does: Stops accepting new connections, then exits.
Use it for: Clean deploys in containers.

---

67) Simple healthcheck endpoint

```
app.get("/health", (req, res) => res.json({ ok: true, uptime:
process.uptime() }));
```

What it does: Returns service status.
Use it for: Load balancers, uptime monitors.

---

68) Structured logging (JSON)

```
function log(level, msg, extra={}) {
  console.log(JSON.stringify({ level, msg, ...extra, ts: new
Date().toISOString() }));
}
log("info", "server_started", { port: 3000 });
```

What it does: Emits machine-readable logs.
Use it for: Cloud logs, better search/filtering.

---

69) Generate an OpenAPI-ish response shape (simple)

```
res.json({ data: result, error: null, meta: { requestId } });
```

What it does: Consistent API response format.
Use it for: Cleaner frontend integration.

---

70) Pagination helper

```
function paginate(page=1, limit=20) {
  const p = Math.max(1, Number(page));
  const l = Math.min(100, Math.max(1, Number(limit)));
  return { skip: (p-1)*l, take: l };
}
```

What it does: Computes skip/take values safely.
Use it for: DB queries, list endpoints.

---

71) Validate/normalize an email

```
function normalizeEmail(email) {
  return String(email).trim().toLowerCase();
}
```

What it does: Normalizes user input.
Use it for: Logins, deduping.

---

72) Safe JSON stringify (handles circular refs)

```
function safeStringify(obj) {
  const seen = new WeakSet();
  return JSON.stringify(obj, (k, v) => {
    if (typeof v === "object" && v) {
      if (seen.has(v)) return "[Circular]";
      seen.add(v);
    }
    return v;
  });
}
```

What it does: Prevents crashes when logging complex objects.
Use it for: Debug logging.

---

73) Parse a date safely

```
function parseDate(value) {
  const d = new Date(value);
  return Number.isNaN(d.getTime()) ? null : d;
}
```

What it does: Avoids invalid date bugs.
Use it for: Filters, scheduling inputs.

---

74) Simple in-memory job queue

```
const queue = [];
let running = false;

async function runQueue() {
  if (running) return;
  running = true;
  while (queue.length) await queue.shift()();
  running = false;
}

function enqueue(job) {
  queue.push(job);
  runQueue();
}
```

What it does: Runs async jobs sequentially.
Use it for: Rate-limited processing (simple cases).

---

75) Limit concurrency for async tasks

```
async function mapLimit(items, limit, fn) {
  const results = [];
  const executing = new Set();
  for (const item of items) {
    const p = Promise.resolve().then(() => fn(item));
    results.push(p);
    executing.add(p);
    p.finally(() => executing.delete(p));
    if (executing.size >= limit) await Promise.race(executing);
```

```
  }
  return Promise.all(results);
}
```

What it does: Controls how many promises run at once.
Use it for: Batch API calls without overload.

---

76) Basic file watcher

```
import fs from "node:fs";

fs.watch("./src", { recursive: true }, (eventType, filename) => {
  console.log(eventType, filename);
});
```

What it does: Watches changes in a directory.
Use it for: Dev tooling, rebuild triggers.

---

77) Spawn a child process

```
import { spawn } from "node:child_process";

const p = spawn("node", ["-v"]);
p.stdout.on("data", d => process.stdout.write(d));
p.on("close", code => console.log("exit", code));
```

What it does: Runs another command and reads output.
Use it for: Build pipelines, CLIs.

---

78) Exec a shell command (careful!)

```
import { exec } from "node:child_process";

exec("ls -la", (err, stdout, stderr) => {
```

```
  if (err) return console.error(err);
  console.log(stdout);
});
```

What it does: Runs shell with buffered output.
Use it for: Quick scripts (avoid with untrusted input).

---

79) Serve SSE (Server-Sent Events)

```
import http from "node:http";

http.createServer((req, res) => {
  if (req.url === "/events") {
    res.writeHead(200, {
      "Content-Type": "text/event-stream",
      "Cache-Control": "no-cache",
      Connection: "keep-alive",
    });
    const t = setInterval(() => {
      res.write(`data: ${JSON.stringify({ time: Date.now() })}\n\n`);
    }, 1000);
    req.on("close", () => clearInterval(t));
    return;
  }
  res.end("ok");
}).listen(3000);
```

What it does: Pushes updates over HTTP without WebSockets.
Use it for: Live feeds, dashboards.

---

80) Basic cookie parsing (no deps)

```
function parseCookies(cookieHeader="") {
  return Object.fromEntries(cookieHeader.split(";").map(v =>
v.trim().split("=")).filter(a => a[0]));
```

```
}
```

What it does: Parses `Cookie:` header into object.
Use it for: Simple session identifiers.

---

81) Set a cookie header

```
res.setHeader("Set-Cookie", "session=abc123; HttpOnly; Path=/;
SameSite=Lax");
```

What it does: Sends a cookie to the browser.
Use it for: Sessions (prefer signed/secure cookies).

---

82) Simple CSRF token idea (concept)

```
import { randomBytes } from "node:crypto";
const csrf = randomBytes(16).toString("hex");
```

What it does: Generates token you store + verify.
Use it for: Protecting form submissions (when using cookies).

---

83) Base64 encode/decode

```
const b64 = Buffer.from("hello", "utf8").toString("base64");
const txt = Buffer.from(b64, "base64").toString("utf8");
```

What it does: Converts data to/from base64.
Use it for: Tokens, safe transport (not encryption).

---

84) File checksum

```
import fs from "node:fs";
```

```
import { createHash } from "node:crypto";

function sha256File(file) {
  return new Promise((resolve, reject) => {
    const h = createHash("sha256");
    fs.createReadStream(file)
      .on("data", d => h.update(d))
      .on("error", reject)
      .on("end", () => resolve(h.digest("hex")));
  });
}
```

What it does: Hashes a file without loading it all.
Use it for: Integrity checks.

---

85) Validate JSON schema quickly (Ajv)

```
// npm i ajv
import Ajv from "ajv";
const ajv = new Ajv();

const validate = ajv.compile({ type: "object", properties: { id: {
type: "number" } }, required: ["id"] });
console.log(validate({ id: 1 }), validate.errors);
```

What it does: Validates data against JSON Schema.
Use it for: APIs with formal schemas.

---

86) Create an LRU cache (quick)

```
class LRU {
  constructor(limit=100) { this.limit=limit; this.map=new Map(); }
  get(k){ if(!this.map.has(k)) return; const v=this.map.get(k);
this.map.delete(k); this.map.set(k,v); return v; }
```

```
  set(k,v){ if(this.map.has(k)) this.map.delete(k); this.map.set(k,v);
if(this.map.size>this.limit)
this.map.delete(this.map.keys().next().value); }
}
```

What it does: Keeps recent items, evicts oldest.
Use it for: Bounded caching.

---

87) Simple “once” helper

```
function once(fn) {
  let done = false, val;
  return (...args) => (done ? val : (done=true, val=fn(...args)));
}
```

What it does: Ensures a function runs only once.
Use it for: Init logic, singletons.

---

88) Validate required env vars

```
function requireEnv(name) {
  const v = process.env[name];
  if (!v) throw new Error(`Missing env var: ${name}`);
  return v;
}
```

What it does: Fails fast on missing config.
Use it for: Production safety.

---

89) Simple config object with types

```
const config = {
  port: Number(process.env.PORT ?? 3000),
  debug: process.env.DEBUG === "true",
```

```
};
```

What it does: Normalizes config values.
Use it for: Avoiding “stringly typed” bugs.

---

90) Basic request ID middleware (Express)

```
import { randomUUID } from "node:crypto";

app.use((req, res, next) => {
  req.id = randomUUID();
  res.setHeader("X-Request-Id", req.id);
  next();
});
```

What it does: Adds trace ID per request.
Use it for: Debugging logs across services.

---

91) Input sanitization (basic)

```
function stripControlChars(s) {
  return String(s).replace(/[\x00-\x1F\x7F]/g, "");
}
```

What it does: Removes control characters.
Use it for: Logging safety, basic cleanup (not full security).

---

92) Validate a URL safely

```
function isValidUrl(value) {
  try { new URL(value); return true; } catch { return false; }
}
```

What it does: Checks if a string is a valid URL format.
Use it for: Input validation.

---

93) Basic file download with fetch + stream to disk

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

const res = await fetch("https://example.com/file.zip");
await pipeline(res.body, fs.createWriteStream("file.zip"));
```

What it does: Streams HTTP response to a file.
Use it for: Download tools, installers.

---

94) Parse a JSON Lines (NDJSON) file

```
import fs from "node:fs";
import readline from "node:readline";

const rl = readline.createInterface({ input:
fs.createReadStream("data.ndjson") });
for await (const line of rl) {
  const obj = JSON.parse(line);
  // process obj
}
```

What it does: Reads line-by-line JSON objects.
Use it for: Large logs, data exports.

---

95) Simple metrics counter

```
const metrics = { requests: 0 };

app.use((req, res, next) => {
```

```
  metrics.requests++;
  next();
});

app.get("/metrics", (req, res) => res.json(metrics));
```

What it does: Tracks simple runtime metrics.
Use it for: Debugging traffic patterns.

---

96) Memory leak guard: max listeners

```
import { EventEmitter } from "node:events";
EventEmitter.defaultMaxListeners = 50;
```

What it does: Raises listener warning threshold (or set lower to catch leaks).
Use it for: Avoid “MaxListenersExceededWarning” surprises.

---

97) ESM __dirname equivalent

```
import { fileURLToPath } from "node:url";
import path from "node:path";

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
```

What it does: Recreates `__dirname` in ES modules.
Use it for: Relative file reads in modern Node.

---

98) Simple HTML escaping for output safety

```
function escapeHtml(s="") {
  return s.replace(/[&<>"']/g, c => ({
"&":"&amp;","<":"&lt;",">":"&gt;",'"':"&quot;","'":"&#39;" }[c]));
}
```

What it does: Prevents HTML injection in rendered strings.
Use it for: Server-side templates, logs shown in HTML.

---

99) Validate port + start server safely

```
function normalizePort(v) {
  const p = Number(v);
  return Number.isInteger(p) && p > 0 && p < 65536 ? p : 3000;
}
```

What it does: Prevents invalid port values.
Use it for: Robust server startup.

---

100) One-file “API + static” server combo

```
import express from "express";

const app = express();
app.use(express.json());
app.use(express.static("public"));

app.get("/api/time", (req, res) => res.json({ time: new
Date().toISOString() }));
app.listen(3000, () => console.log("http://localhost:3000"));
```

What it does: Serves frontend + API from one Node app.
Use it for: Small apps, demos, internal tools.