TECHNICAL REFERENCE · NODE.JS FOR JAVASCRIPT DEVELOPERS

Node.js

A Developer Reference for the Browser-to-Server Transition

Version 1.0 · November 2025 · Prepared by Avalynn Circe

This reference accompanies JavaScript Grew Up. Here's What Changed. It is organized by topic and intended as a lookup resource while you build. Each section covers the core API surface, common patterns, and the gotchas most likely to cost you time.

Contents

1. Environment & Process

The process object is the Node equivalent of the browser's window — a global that's always available without importing anything.

Property / Method	What It Does
`process.env`	Object containing all environment variables. Access with process.env.MY_VAR.
`process.argv`	Array of command-line arguments. Index 0 is node, index 1 is the script path, index 2+ are your args.
`process.cwd()`	Returns the current working directory as a string.
`process.exit(code)`	Terminates the process. Code 0 = success. Any other number = error.
`process.stdout.write()`	Writes to stdout without a trailing newline (unlike console.log).
`process.stderr.write()`	Writes to stderr. Useful for separating error output from normal output.
`process.on('exit', fn)`	Registers a callback that runs just before the process exits.
`process.on('uncaughtException', fn)`	Last-resort handler for uncaught errors. Log and exit — do not try to resume.
`process.platform`	Returns 'linux', 'darwin', or 'win32'. Useful for platform-specific logic.

`process.version`	The Node.js version string, e.g. 'v20.11.0'.

> *process is a global — you never require() or import it. It is always available.*

2. The File System (fs)

Node's fs module provides everything you need to read, write, copy, and delete files. Always use the promises API (fs/promises or fs.promises) — the callback-based versions are older and harder to work with.

Import

```
// Recommended import pattern
import { readFile, writeFile, readdir, stat, mkdir, unlink } from 'fs/promises';

// CommonJS equivalent
const { readFile, writeFile, readdir, stat, mkdir, unlink } = require('fs').promises;
```

Core Operations

Method	Signature	Notes
`readFile()`	readFile(path, encoding?)	Returns file contents as a string (if encoding given) or Buffer. 'utf8' is the most common encoding.
`writeFile()`	writeFile(path, data, options?)	Writes data to path. Creates the file if it doesn't exist. Overwrites by default.
`appendFile()`	appendFile(path, data)	Like writeFile but adds to the end instead of overwriting.
`readdir()`	readdir(path, options?)	Returns an array of filenames in a directory. Pass { withFileTypes: true } for file type info.
`stat()`	stat(path)	Returns metadata: size, creation date, modification date, isFile(), isDirectory().
`mkdir()`	mkdir(path, options?)	Creates a directory. Pass { recursive: true } to create nested directories without error.
`unlink()`	unlink(path)	Deletes a file. Does not work on directories — use rmdir() or rm() for those.
`rename()`	rename(oldPath, newPath)	Moves or renames a file or directory.
`copyFile()`	copyFile(src, dest, flags?)	Copies a file. Pass fs.constants.COPYFILE_EXCL as flags to fail if dest already exists.
`access()`	access(path, mode?)	Checks whether a file exists and is accessible. Throws if not. Use instead of checking existence before opening.

Common Patterns

fs patterns

```
import { readFile, writeFile, mkdir } from 'fs/promises';
import { join } from 'path';

// Read a JSON config file
const config = JSON.parse(await readFile('config.json', 'utf8'));

// Write an object to a JSON file
await writeFile('output.json', JSON.stringify(data, null, 2));

// Read all .js files in a directory
const files = (await readdir('./src')).filter(f => f.endsWith('.js'));

// Create a nested directory safely
await mkdir(join('output', 'reports', '2025'), { recursive: true });
```

> *path.join() and path.resolve() are your friends. Never concatenate file paths with string addition — it breaks on Windows. Always use the path module.*

3. HTTP Servers

The built-in http module is the foundation. Most real applications add Express or Fastify on top, but understanding the base layer helps when things go wrong.

Anatomy of an HTTP Request Handler

http.createServer

```
import { createServer } from 'http';

const server = createServer((req, res) => {
  // req: IncomingMessage — what came in
  // res: ServerResponse — what you send back

  const { method, url, headers } = req;

  // Read request body (it comes in as a stream)
  let body = '';
  req.on('data', chunk => { body += chunk; });
  req.on('end', () => {
    const parsed = body ? JSON.parse(body) : null;

    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify({ received: parsed }));
  });
});

server.listen(3000);
```

req property	What It Contains
`req.method`	HTTP method as uppercase string: 'GET', 'POST', 'PUT', 'DELETE', etc.
`req.url`	The path and query string: '/users?page=2'. Does not include host or protocol.
`req.headers`	Object of request headers, all keys lowercased: req.headers['content-type'].

`req.socket.remoteAddress`	The IP address of the connecting client.

res method	What It Does
`res.writeHead(code, headers)`	Sets the status code and response headers. Must be called before res.write() or res.end().
`res.setHeader(name, value)`	Sets a single response header. Can be called multiple times before writeHead.
`res.write(chunk)`	Sends a chunk of the response body. Can be called multiple times for streaming.
`res.end(data?)`	Finalizes and sends the response. Required — every request must end.

With Express

Express basics

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

app.use(express.json());  // Parse JSON request bodies automatically

// GET route
app.get('/users/:id', async (req, res) => {
  const { id } = req.params;       // Route parameters
  const { include } = req.query;   // Query string: ?include=posts
  // req.body is available for POST/PUT
  res.json({ userId: id });
});

// Error handling middleware — must have 4 parameters
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Internal server error' });
});

app.listen(3000, () => console.log('Running on :3000'));
```

4. Modules: CommonJS vs. ES Modules

Node supports both module systems. Which one you use is determined by your package.json or file extension.

	CommonJS (CJS)	ES Modules (ESM)
`Import syntax`	const x = require('./mod')	import x from './mod.js'
`Export syntax`	module.exports = x	export default x / export { x }
`File extension`	.js or .cjs	.js (with type:module) or .mjs
`package.json`	No type field (or "type": "commonjs")	"type": "module"
`__dirname`	Available as a global	Not available — use import.meta.url instead
`__filename`	Available as a global	Not available — use import.meta.url instead
`Top-level await`	Not supported	Supported

`Dynamic import`	require() is synchronous	import() returns a Promise

Getting __dirname in ESM

ESM __dirname equivalent

```
import { fileURLToPath } from 'url';
import { dirname, join } from 'path';

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

// Now use __dirname as normal
const configPath = join(__dirname, 'config.json');
```

> *When mixing CJS and ESM packages — which happens constantly with npm packages — ESM files can import CJS modules, but CJS files cannot require() ESM modules. If you hit a 'require() of ES Module' error, you need to use dynamic import() instead.*

5. npm & package.json

Essential npm Commands

Command	What It Does
`npm init -y`	Creates a package.json with defaults. Start every Node project with this.
`npm install <package>`	Installs a package and adds it to dependencies.
`npm install -D <package>`	Installs a dev dependency (testing tools, build tools, nodemon, etc.).
`npm install`	Installs all dependencies listed in package.json. Run this after cloning a repo.
`npm uninstall <package>`	Removes a package and its entry from package.json.
`npm run <script>`	Runs a script defined in the scripts section of package.json.
`npm outdated`	Lists installed packages that have newer versions available.
`npm audit`	Checks installed packages for known security vulnerabilities.
`npm ci`	Clean install for CI environments. Faster and stricter than npm install.
`npx <command>`	Runs a package without installing it globally. Use for one-off tools.

package.json Key Fields

package.json

```
{
  "name": "my-project",
  "version": "1.0.0",
  "type": "module",          // Enables ES modules for .js files
  "main": "src/index.js",    // Entry point for the package
  "scripts": {
```

```
    "start": "node src/index.js",
    "dev": "nodemon src/index.js",
    "test": "jest",
    "build": "tsc"
  },
  "dependencies": {
    "express": "^4.18.2"      // ^ = accept minor/patch updates
  },
  "devDependencies": {
    "nodemon": "^3.0.1",      // Only needed during development
    "jest": "^29.0.0"
  },
  "engines": {
    "node": ">=18.0.0"        // Minimum Node version required
  }
}
```

> *Always commit package-lock.json (or yarn.lock / pnpm-lock.yaml). Never commit node_modules. Add node_modules to your .gitignore before your first commit.*

6. Environment Variables

Never hardcode secrets — API keys, database credentials, tokens — in source code. Use environment variables.

.env

```
# .env file — never commit this
DATABASE_URL=postgres://user:pass@localhost:5432/mydb
API_KEY=sk-abc123
PORT=3000
NODE_ENV=development
```

Using dotenv

```
// Load .env into process.env (do this once, at app entry point)
import 'dotenv/config';

// Access values
const port = process.env.PORT ?? 3000;  // Use ?? for defaults
const dbUrl = process.env.DATABASE_URL;

if (!dbUrl) {
  throw new Error('DATABASE_URL is required');  // Fail fast
}
```

Variable	Convention / Notes
`NODE_ENV`	Set to 'development', 'test', or 'production'. Many libraries change behavior based on this.
`PORT`	The port to listen on. Hosting platforms (Heroku, Render, Railway) set this automatically.
`DATABASE_URL`	Connection string for the database. Standard name across most platforms and ORMs.
`LOG_LEVEL`	Controls logging verbosity. Common values: 'debug', 'info', 'warn', 'error'.

7. Error Handling Patterns

async/await with try/catch

try/catch

```
async function getUser(id) {
  try {
    const response = await fetch(`/api/users/${id}`);
    if (!response.ok) {
      throw new Error(`HTTP ${response.status}: ${response.statusText}`);
    }
    return await response.json();
  } catch (err) {
    // Handle or re-throw
    console.error('Failed to fetch user:', err.message);
    throw err;  // Re-throw if the caller needs to handle it
  }
}
```

Custom Error Classes

Custom errors

```
class AppError extends Error {
  constructor(message, statusCode = 500) {
    super(message);
    this.name = this.constructor.name;
    this.statusCode = statusCode;
    Error.captureStackTrace(this, this.constructor);
  }
}

class NotFoundError extends AppError {
  constructor(resource) {
    super(`${resource} not found`, 404);
  }
}

// Usage
throw new NotFoundError('User');

// In Express error handler
app.use((err, req, res, next) => {
  const status = err.statusCode ?? 500;
  res.status(status).json({ error: err.message });
});
```

Unhandled Rejections

Process-level error handlers

```
// Catch promises that were rejected without a .catch()
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled rejection:', reason);
  process.exit(1);  // Exit — unknown state is dangerous
});
```

```
// Catch synchronous exceptions that escaped all try/catch
process.on('uncaughtException', (err) => {
  console.error('Uncaught exception:', err);
  process.exit(1);
});
```

> *These handlers are a safety net, not a strategy. If you're hitting unhandledRejection regularly, find the missing await or .catch() and fix it.*

8. Streams

Streams let you process data in chunks rather than loading it all into memory at once. Essential for large files, HTTP responses, and real-time data.

Stream Type	Direction	Example
`Readable`	Data flows out	fs.createReadStream(), HTTP request body, process.stdin
`Writable`	Data flows in	fs.createWriteStream(), HTTP response, process.stdout
`Duplex`	Both directions	TCP sockets, net.Socket
`Transform`	In, modify, out	zlib.createGzip(), crypto streams, CSV parsers

Stream patterns

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

// Compress a large file without loading it into memory
await pipeline(
  createReadStream('large-file.csv'),
  createGzip(),
  createWriteStream('large-file.csv.gz')
);

// Read a stream line by line
import { createInterface } from 'readline';

const rl = createInterface({
  input: createReadStream('data.txt'),
  crlfDelay: Infinity
});

for await (const line of rl) {
  console.log(line);
}
```

> *Use stream/promises pipeline() instead of pipe(). It properly propagates errors and cleans up streams — pipe() silently swallows errors from intermediate streams.*

9. Child Processes

Node can spawn other processes — shell commands, Python scripts, other Node processes — using the child_process module.

Method	Use When
`exec()`	You need the full output as a string and the command is simple. Buffers all output.
`execFile()`	Like exec() but runs a file directly without a shell. Slightly more secure.
`spawn()`	You need streaming output or the command produces large amounts of data.
`fork()`	You want to run another Node.js script as a child with IPC (message passing).
`execSync()`	Blocking version of exec(). Useful in scripts, dangerous in servers.

child_process patterns

```
import { exec, spawn } from 'child_process';
import { promisify } from 'util';

const execAsync = promisify(exec);

// Simple command — capture output
const { stdout } = await execAsync('git log --oneline -10');
console.log(stdout);

// Long-running command — stream output
const child = spawn('npm', ['run', 'build'], { stdio: 'inherit' });

child.on('close', (code) => {
  if (code !== 0) throw new Error(`Build failed with code ${code}`);
});
```

10. Useful Built-in Globals

These are available everywhere in Node without importing anything.

Global	What It Is
`console`	Same as browser: log, error, warn, table, time/timeEnd, assert.
`process`	Process info and control. See Section 1.
`Buffer`	For working with binary data. Buffer.from(), Buffer.alloc(), buf.toString().
`setTimeout / clearTimeout`	Same as browser. Schedules a one-time callback. Returns a Timeout object.
`setInterval / clearInterval`	Same as browser. Schedules a repeating callback.
`setImmediate / clearImmediate`	Schedules a callback after I/O events in the current event loop turn. Faster than setTimeout(fn, 0).
`queueMicrotask()`	Schedules a microtask — runs before the next macrotask. Same as Promise.resolve().then().
`globalThis`	The global object. In browsers it's window, in Node it's global. globalThis works in both.

`structuredClone()`	Deep clones an object. Available since Node 17. No more JSON.parse(JSON.stringify(x)).
`fetch()`	The same fetch() you know from the browser. Available natively since Node 18.
`crypto`	Web Crypto API, available globally since Node 19. For Node-specific crypto, use require('crypto').
`URL / URLSearchParams`	Same as browser. Parse and construct URLs without a library.

11. Common npm Packages

HTTP & APIs

Package	Use Case
`express`	The standard HTTP framework. Routing, middleware, request parsing.
`fastify`	Faster alternative to Express with built-in schema validation.
`hono`	Lightweight, edge-ready framework. Works in Node, Deno, Bun, and Cloudflare Workers.
`axios`	HTTP client with interceptors, automatic JSON parsing, and better error handling than fetch().
`zod`	Schema validation and parsing for request bodies, env vars, and config objects.

Databases

Package	Use Case
`prisma`	Full-featured ORM with migrations, type safety, and a query builder. Works with Postgres, MySQL, SQLite.
`drizzle-orm`	Lightweight ORM. Type-safe, closer to raw SQL, smaller bundle.
`pg`	Raw Postgres client. Use when you want full SQL control without an ORM.
`better-sqlite3`	Synchronous SQLite driver. Great for local-first apps, scripts, and testing.
`ioredis`	Redis client with promise support, clustering, and pub/sub.
`mongoose`	MongoDB ODM. Schema-based modeling for MongoDB documents.

Auth & Security

Package	Use Case
`jsonwebtoken`	Sign and verify JWTs. The standard for token-based auth.
`bcrypt`	Hash passwords. Never store plain text — always hash with bcrypt or argon2.
`argon2`	More modern password hashing than bcrypt. Recommended for new projects.
`helmet`	Express middleware that sets security-related HTTP headers automatically.
`cors`	Express middleware for configuring CORS headers.

Developer Tools

Package	Use Case
`nodemon`	Restarts the server automatically on file save during development.
`dotenv`	Loads .env files into process.env. See Section 6.
`winston`	Structured logging with multiple transports (console, file, external service).
`pino`	Extremely fast JSON logger. Better choice than winston for high-throughput servers.
`jest`	Testing framework. Works for unit, integration, and snapshot tests.
`vitest`	Faster alternative to Jest. Native ESM support and Vite-compatible.
`tsx`	Runs TypeScript files directly with Node. No build step needed for development.

12. HTTP Status Code Reference

When you're building an API, use status codes consistently. They are the API's primary signaling mechanism.

2xx — Success

Code	Name	Use When
`200`	OK	Standard success. The response body contains the requested data.
`201`	Created	A new resource was created. Return the new resource in the body.
`204`	No Content	Success with no body. Use for DELETE or PUT when nothing needs to be returned.
`206`	Partial Content	Used for range requests (streaming, pagination at the byte level).

3xx — Redirection

Code	Name	Use When
`301`	Moved Permanently	URL has changed. Browsers and crawlers cache this. Use for permanent moves.
`302`	Found	Temporary redirect. The original URL may be used again.
`304`	Not Modified	Response to a conditional GET. Client can use its cached version.

4xx — Client Errors

Code	Name	Use When
`400`	Bad Request	Malformed syntax, invalid JSON, or missing required fields.
`401`	Unauthorized	No valid authentication provided. The client needs to log in.
`403`	Forbidden	Authenticated but not allowed. The client is logged in but lacks permission.
`404`	Not Found	Resource does not exist. Also use to avoid leaking existence of private resources.

405	Method Not Allowed	The HTTP method is not supported for this route (e.g. DELETE on a read-only endpoint).
409	Conflict	Request conflicts with current state (e.g. duplicate unique value, edit conflict).
410	Gone	Resource existed but has been permanently deleted.
422	Unprocessable Entity	Syntax is valid but semantic errors exist. Use for validation failures.
429	Too Many Requests	Rate limit exceeded. Include Retry-After header with wait time.

5xx — Server Errors

Code	Name	Use When
500	Internal Server Error	Something unexpected went wrong on your end. Log the details, return a safe message.
501	Not Implemented	The endpoint exists but the feature isn't built yet.
502	Bad Gateway	Your server received an invalid response from an upstream service.
503	Service Unavailable	Server is down for maintenance or overloaded. Include Retry-After if known.
504	Gateway Timeout	Upstream service didn't respond in time.

> *The most common mistake in API design is returning 200 OK with an error message in the body. Use the status code to signal success or failure — that's what it's for.*

Avalynn Circe is a technical writer and software developer based in St. Paul, MN. She is the author of JavaScript: The Parts, available at leanpub.com/jsparts.