

Sidebar

Programming · Node.js and npm Cheat Sheet

Running JavaScript with Node.js and managing packages with npm: command-line options, modules, built-ins, package.json, installing, scripts, security, configuration and debugging. Written for Node 24 and npm 11, the versions installed here; the commands and options were run in a scratch project to check they work, and a card records what was found on this machine.

YOUR SETUP (FOUND ON THIS MACHINE)

- node -v / npm -v** Node **v24.18.0** (the 24 line is Long Term Support, nicknamed Krypton) and npm **11.16.0**; **npm** is the same version
- npm doctor** Reports that newer versions exist (npm 12.2.0 and Node 24.21.0 at the time of writing); everything else checked was fine
- npm root -g** Global packages live in `C:\Users\emuba\AppData\Roaming\npm`; the only global package is `@anthropic-ai/claude-code`
- npm config get registry** `https://registry.npmjs.org/` (the default)
- corepack --version** 0.35.0 is included with Node; **corepack enable** sets up pnpm and Yarn when you need them
- Not installed** pnpm, Yarn, nvm, fnm and Volta (no Node version manager); Bun and Deno are absent too
- astro-site** Your site's `package.json` asks for Node 22.12 or newer (`engines`), which 24.18 satisfies

RUNNING NODE

- node file.js a b** Run a file; the extra words arrive in `process.argv.slice(2)`
- node** With no file: the interactive REPL
- node -e "code" node -p "expression"** Run code from the command line / run an expression and print the result
- node --watch file.js** Restart whenever the file (or what it imports) changes; `--watch-path=src` chooses folders. Not combinable with `-e`
- node --env-file=.env file.js** Load variables from a file into `process.env`; `--env-file-if-exists` does not fail if the file is missing
- node --run <script>** Run a script from `package.json` without starting npm (faster)
- node file.ts** Runs TypeScript directly on this version, but only syntax that can be erased (no enums or other generated code)
- node --check file.js** Check syntax without running
- node --test** Run test files with the built-in runner; `--test-reporter=spec` and `--experimental-test-coverage` work
- node -r ./setup.js node --import ./setup.mjs** Load a module before your program (CommonJS / ES module)
- node --max-old-space-size=2048** Raise the memory limit (megabytes)
- node --trace-warnings --no-warnings --disable-warning=Code** Show where warnings come from / hide all / hide one kind
- node --inspect --inspect-brk** Allow a debugger to connect / also stop on the first line
- node --cpu-prof --heap-prof** Write CPU or memory profiles for later analysis
- node --permission --allow-fs-read=*** Run with the permission model: file, network and process access is denied unless allowed
- node --help** All options

THE REPL

<code>.help</code>	List the REPL commands
<code>.editor</code>	Enter several lines; finish with Ctrl+D
<code>.load file.js .save file.js</code>	Run a file in the session / save what you typed
<code>.break .clear</code>	Abandon a multi-line entry / reset the session
<code>_</code>	The result of the last expression
<code>Tab</code>	Complete names; press twice to list the options
<code>.exit Ctrl+D Ctrl+C twice</code>	Leave the REPL

MODULES: ESM AND COMMONJS

<code>import fs from 'node:fs'</code>	ES module syntax; the <code>node:</code> prefix marks a built-in
<code>const fs = require('node:fs')</code>	CommonJS syntax
<code>"type": "module"</code>	In <code>package.json</code> : treat <code>.js</code> files as ES modules (otherwise as CommonJS)
<code>.mjs .cjs</code>	File extensions that force ES module / CommonJS whatever <code>type</code> says
<code>import.meta.dirname import.meta.filename</code>	This file's folder and path in an ES module (works on this version)
<code>__dirname __filename</code>	The CommonJS equivalents; not defined in ES modules
<code>await import('./x.js')</code>	Load a module while the program runs; top-level <code>await</code> works in ES modules
<code>module.exports = ... export default ...</code>	Share a value from a file
<code>import data from './x.json' with { type: 'json' }</code>	Import a JSON file in an ES module (my knowledge)

BUILT-IN MODULES AND GLOBALS WORTH KNOWING

<code>node:fs / node:fs/promises</code>	Read and write files (the promise version works with <code>await</code>)
<code>node:path node:os node:url</code>	Paths, system information, URLs
<code>node:http node:https</code>	Servers and requests
<code>node:crypto</code>	Hashes, random bytes, encryption
<code>node:child_process</code>	Run other programs
<code>node:readline node:events node:stream node:util</code>	Read input a line at a time / events / streams / helpers
<code>node:test node:assert</code>	The built-in test runner and assertions
<code>node:worker_threads node:zlib node:timers/promises</code>	Threads / compression / <code>await</code> -able timers
<code>node:sqlite</code>	A built-in SQLite database (present on this version)
<code>fetch WebSocket structuredClone</code>	Global functions available without importing (confirmed on this version)
<code>process console Buffer URL AbortController setTimeout</code>	More globals
<code>72 built-in modules</code>	That is how many this version lists in <code>require('module').builtinModules</code>

THE PROCESS OBJECT

<code>process.argv</code>	Command-line words: <code>[node, script, ...your arguments]</code>
<code>process.env.NAME</code>	An environment variable (a string or <code>undefined</code>)
<code>process.cwd() process.platform</code>	Current folder / <code>win32</code> here
<code>process.exit(1) process.exitCode = 1</code>	Stop now with a code / set the code and finish normally (0 means success)
<code>process.on('SIGINT', fn)</code>	Run code when the user presses Ctrl+C
<code>process.memoryUsage() process.uptime()</code>	Memory and running time
<code>process.versions</code>	Versions of Node and its parts (V8, OpenSSL, SQLite...)
<code>process.stdin process.stdout</code>	Standard input and output streams

PACKAGE.JSON ESSENTIALS

<code>npm init -y</code>	Create a <code>package.json</code> with defaults
<code>name version description</code>	The package's identity (<code>version</code> uses semantic versioning: major.minor.patch)
<code>scripts</code>	Named commands, run with <code>npm run <name></code>
<code>dependencies devDependencies</code>	Packages needed to run / only to build and test
<code>peerDependencies optionalDependencies</code>	Packages the user must supply / nice to have
<code>main exports bin</code>	Entry file / public entry points / commands the package installs
<code>type</code>	<code>module</code> or <code>commonjs</code>
<code>engines</code>	Versions of Node (and npm) that the project supports, e.g. <code>{ "node": ">=22.12.0" }</code>
<code>private</code>	<code>true</code> prevents publishing by accident
<code>files</code>	What goes into the package when published
<code>workspaces</code>	Several packages in one repository
<code>overrides</code>	Force a version of a dependency anywhere in the tree
<code>npm pkg get name npm pkg set scripts.dev="..."</code>	Read and edit <code>package.json</code> from the command line
<code>^1.2.3 ~1.2.3 1.2.3 >=1.2 *</code>	Version ranges: any 1.x.y at least 1.2.3 / only 1.2.x / exactly that / at least / anything

RUNNING SCRIPTS & TOOLS

<code>npm run <name></code>	Run a script from <code>package.json</code> ; plain <code>npm run</code> lists them
<code>npm run <name> -- --flag value</code>	Everything after <code>--</code> is passed to the script
<code>npm start npm test</code>	Shortcuts for the <code>start</code> and <code>test</code> scripts (<code>npm start</code> defaults to <code>node server.js</code> if that file exists)
<code>npm run --if-present <name></code>	Do nothing instead of failing when the script does not exist
<code>pre<name> post<name></code>	Scripts that run before / after another (such as <code>predev</code> , <code>prebuild</code> ; your Astro site uses these)
<code>npx <tool> npm exec -- <tool></code>	Run a package's command, installing it temporarily if needed; <code>--yes</code> skips the prompt
<code>npm create <name></code>	Run a project generator, such as <code>npm create astro@latest</code>
<code>node_modules/.bin</code>	Where project tools live; npm scripts find them automatically

INSTALLING & REMOVING PACKAGES

<code>npm install</code>	Install everything listed in <code>package.json</code> (<code>npm i</code>)
<code>npm install lodash</code>	Add a package to <code>dependencies</code>
<code>npm install -D prettier</code>	Add to <code>devDependencies</code> (<code>--save-dev</code>)
<code>npm install lodash@4.17.21 @latest @^4</code>	A specific version / the newest / a range or tag
<code>npm install --save-exact lodash</code>	Record the exact version, not a <code>^</code> range
<code>npm install -g <pkg></code>	Install globally (a command-line tool). Add <code>--dry-run</code> to see what would happen
<code>npm install --omit=dev</code>	Skip dev dependencies (for a production install)
<code>npm uninstall lodash</code>	Remove a package
<code>npm update</code>	Move packages up within the ranges in <code>package.json</code>
<code>npm ci</code>	A clean, exact install from <code>package-lock.json</code> ; deletes <code>node_modules</code> first; used in automated builds
<code>npm prune npm dedupe npm rebuild</code>	Remove extras / flatten duplicates / rebuild native add-ons

LOOKING AROUND

<code>npm ls --depth=0</code>	Top-level installed packages; <code>npm ls -g --depth=0</code> for global ones
<code>npm outdated</code>	Which packages have newer versions (current, wanted, latest)
<code>npm explain <pkg></code>	Why a package is installed: the chain that needs it
<code>npm view <pkg> version versions --json</code>	Facts about a package on the registry
<code>npm search <words></code>	Search the registry
<code>npm docs <pkg> npm repo <pkg></code>	Open its home page / repository
<code>npm fund</code>	Which packages ask for funding
<code>npm pack --dry-run</code>	List what would be in the published package
<code>npm query ':root'</code>	Search the dependency tree with selectors
<code>npm root npm root -g npm prefix</code>	Where local / global packages live; the project root
<code>npm ping npm whoami npm doctor</code>	Check the registry connection / who you are logged in as / the health of the setup

SECURITY & THE LOCK FILE

`package-lock.json` Records the exact version of every package; commit it so everyone gets the same tree

`npm audit` Report known vulnerabilities in your dependencies

`npm audit fix` Apply safe fixes; add `--dry-run` first, and treat `--force` as risky because it can jump major versions

`npm install` Never edit `node_modules` by hand: it is rebuilt from the lock file

`ignore-scripts` A setting (`false` here) that stops packages running their install scripts; `npm config set ignore-scripts true` turns it on, and some packages then need `npm rebuild`

`Choose packages with care` Check the publisher, download counts, last update and the exact spelling: look-alike names are used to spread malware

VERSIONS & UPDATING

`npm install -g npm@latest` Update npm itself (my knowledge)

`winget install OpenJS.NodeJS.LTS` Install or update the Long Term Support Node with winget (the package id is from my knowledge)

`nvm-windows` `fnm` `Volta` Tools that install several Node versions side by side; none is installed here

`engines .nvmrc` Record which Node version a project wants

`LTS` or `Current?` Use an LTS (even-numbered) release for real work; Current releases are for trying new features

`corepack enable` Turn on pnpm and Yarn shims that ship with Node

HABITS WORTH HAVING (MY SUGGESTIONS)

`Ignore node_modules` Add `node_modules/` to `.gitignore`; never commit it

`Commit package-lock.json` And use `npm ci` when you want an exact copy

`Script the routine` Put `dev`, `build`, `test` and `lint` in `scripts` so everyone uses the same commands

`Pin the Node version` Use `engines` (and `.nvmrc` if you add a version manager)

`Update regularly` Run `npm outdated` now and then; small updates are easier than big ones

NPM CONFIGURATION

`npm config list` `npm config list -l` Your settings / every setting with its default

`npm config get <key>` `set` `delete` Read, change or remove a setting

`.npmrc` A settings file: project (next to `package.json`), user (`~/.npmrc`), global, and the built-in one; project wins over user

`registry @scope:registry` Where to download packages; a different registry just for one scope

`save-exact=true` In `.npmrc`: always record exact versions

`fund=false` `audit=false` Switch off the funding message / the audit on install

`npm cache verify` Check and tidy the cache (here under `C:\Users\emuba\AppData\Local\npm-cache`)

`npm help <command>` The manual for a command

DEBUGGING & PROFILING

`JavaScript Debug Terminal` VS Code command: a terminal where any `node` you start is debugged automatically

`node --inspect-brk app.js` Pause on the first line; attach from VS Code or `chrome://inspect`

`launch.json` `type node` See the VS Code Debugging sheet (`program`, `skipFiles`)

`console.table(x)` `console.dir(x, {depth: null})` Readable output of arrays and deeply nested objects

`console.time('x')` `console.timeEnd('x')` Time a block

`node --cpu-prof app.js` Produces a `.cpuprofile` file you can open in VS Code or Chrome DevTools

"NPM.PS1 CANNOT BE LOADED BECAUSE RUNNING SCRIPTS IS DISABLED"

In Windows PowerShell, typing `npm` can run the `npm.ps1` script, which the execution policy may block. Fix it for your account with `Set-ExecutionPolicy RemoteSigned -Scope CurrentUser` (see the Python Environments and PowerShell sheets), or type `npm.cmd` instead, or use a Command Prompt or WSL terminal.

COMMON NPM ERRORS AND WHAT TO TRY

ERESOLVE (conflicting peer dependencies): read which two packages disagree, update one, or as a last resort add `--legacy-peer-deps`. **EACCES** (permission denied): avoid `sudo` `npm`; fix the folder permissions or use a user-level prefix. **Cannot find module**: run `npm install`, then check the path, the file extension and whether you used `require` in an ES module (or the reverse). **EADDRINUSE**: something already uses the port; on Windows find it with `Get-NetTCPConnection -State Listen`. Strange behaviour after changing branches: delete `node_modules` and run `npm ci`.

NODE_MODULES INSIDE ONEDRIVE

Your projects live in a OneDrive folder, and `node_modules` can hold tens of thousands of small files that OneDrive will try to sync, which can be slow. Consider keeping heavy projects outside OneDrive, or telling OneDrive to skip the `node_modules` folder. (This is a general observation, not something I tested on your machine.)