HOW-TO GUIDE

How to Get Started with Node.js

Set up Node.js correctly, verify your runtime, run a first script, and understand the checks that matter before you use it in production.

Programming - July 29, 2026

Quick start

If you need to get Node.js running on a workstation or build host, the shortest safe path is to install a current supported release, verify that both node and npm are available, run a simple script, and confirm that your package workflow works as expected. That gives you a working baseline before you add frameworks, services, or deployment tooling.

A minimal first check looks like this:

Expected output is a version string from each command. If either command fails, fix the installation or update your shell PATH before continuing.

For many technical teams, the practical goal is not just "install Node.js". It is to establish a reproducible runtime that can support local development, CI jobs, and eventually production workloads. After reading this guide, you will be able to install Node.js, validate that the runtime works, create and run a first program, and decide what to verify before treating the environment as production-ready.

When this approach applies

This guide is for the common case where you want a local or build-time Node.js environment for command-line tools, API development, or automation scripts. It assumes you want a simple first working setup rather than a full application framework.

Use this approach if you need to:

- validate that Node.js is installed correctly on a new machine
- create a repeatable starting point for JavaScript-based tooling
- prepare a developer or CI environment before adding dependencies
- confirm that your shell can find the runtime you expect

If your goal is to build an API, you can start with the setup here and then move into a structured service pattern such as Build a REST API in Node.js with Express and JWT Auth. If your concern is supply-chain risk, plan a dependency review before production use and follow Hardening Node.js Apps with Secure Dependency Management.

Prerequisites

Before you begin, make sure you have:

- administrative rights to install software on the target machine, or an approved package manager workflow
- a shell that can run standard command-line tools
- internet access if you plan to install Node.js from an online package source
- a clear decision on which Node.js version you want to use

Version choice matters operationally. If you are setting up a workstation, a current supported release is usually the right starting point. If you are preparing a build pipeline or application runtime, confirm the version required by your application and any native modules before installation. Do not assume the newest release is automatically safe for every workload.

Install Node.js

The exact installation method depends on your operating system and internal standards, but the operational requirement is the same: install Node.js in a way that makes the node and npm commands available in your shell.

On many systems, Node.js can be installed through the native package manager, an enterprise software distribution method, or a version manager. A version manager is useful when you need to test multiple releases without replacing the system runtime.

After installation, open a new terminal session and verify the commands:

A successful installation returns two version numbers. If node works but npm does not, the install is incomplete or your environment is pointing at a partial runtime package. If neither command works, the shell is not resolving the installation path.

Validate the executable path

On shared systems or machines with multiple runtimes installed, confirm which binary your shell is using:

On Windows, use:

This is useful when the output version is unexpected. A different path than the one you intended usually means an older installation is earlier in PATH or your shell session has not picked up the change.

Run a first script

Once the runtime is installed, confirm that it can execute a JavaScript file. Create a simple file named hello.js with this content:

Then run it:

Expected output:

This step validates the essential execution path: the runtime can read a file, interpret JavaScript, and print to standard output. If this fails, check the file name, the working directory, and whether your shell is invoking the correct node binary.

Confirm a basic module workflow

Node.js projects usually rely on packages and a local project directory. To confirm that your environment supports the standard workflow, create a directory, initialize a package manifest, and inspect the generated file:

Expected output is a new package.json file in the directory. You do not need dependencies yet. The purpose of this step is to prove that the package manager can create a project skeleton and that the runtime can support a normal Node.js working directory.

Understand the minimal project structure

A clean starting point helps you avoid confusion later when logs, scripts, and dependencies accumulate. For a small first project, the structure can be as simple as:

The package.json file defines metadata and scripts. Your JavaScript files contain the executable logic. As projects grow, you may add directories such as src/, test/, or config/, but do not introduce that complexity until you need it.

If you want to automate the first script, you can add a start command to package.json:

Then run:

That verifies the package script path, which is the same mechanism you will later use for application startup, tests, and maintenance tasks.

Validate the runtime before you rely on it

A working install is not the same as a safe or stable operational baseline. Before you use Node.js in development or production-adjacent work, validate the environment in a few practical ways.

First, confirm that the version matches what you expect:

Second, confirm that the platform details are what you intended to run on:

This matters when native modules are involved, because mismatched architecture or platform assumptions can break installs later.

Third, confirm that package installation works:

You do not need this package specifically; the point is to verify that dependency resolution, download, and lockfile generation all succeed in your environment. If your organization requires dependency controls, this is also the moment to verify registry access, proxy settings, and approved package sources.

If you are preparing a security-sensitive environment, make sure the runtime can support whatever dependency governance process your organization uses. A clean install today does not guarantee trusted packages tomorrow.

Common issues and safe fixes

The most frequent startup problems are operational, not conceptual.

If node is not found, check whether the install completed and whether the binary directory is in PATH. If a version manager is in use, confirm that the selected version is active in the current shell.

If npm fails while node works, reinstall the package or confirm that the package manager component is included in your distribution.

If commands resolve to the wrong version, inspect the executable path and remove or deactivate conflicting installations. On systems with multiple versions, avoid guessing which runtime is active.

If package installation fails behind a proxy, verify proxy environment variables and corporate certificate requirements before retrying. Do not bypass TLS verification as a convenience fix in a managed environment unless your security policy explicitly allows it and you understand the consequences.

For teams building network-facing services, it is also wise to think ahead about operational controls such as request throttling. Once you move from a first script to an API, patterns like Node.js API Rate Limiting with Redis and Express Middleware become relevant for protecting service capacity.

Rollback and cleanup

If a new installation causes conflicts, the safest rollback is to remove the newly added runtime and restore the previous version or package source. The exact method depends on how Node.js was installed, so use the same tool that installed it.

Before uninstalling, capture the current state:

Keep that information if you need to restore the environment later or document the change for operations review.

For a temporary test environment, cleanup is straightforward:

Remove only the test project directory unless you intentionally want to uninstall the runtime. On shared systems, avoid deleting any system-wide package directories by hand.

What to verify before using Node.js in production workflows

A first successful script is only the beginning. Before you rely on Node.js for production automation, build jobs, or service startup, verify the following:

- the installed version is approved for your workload
- the shell resolves the intended node and npm binaries
- package installation works through your approved network path
- any proxy, certificate, or registry settings are documented
- native modules, if used, are compatible with your platform and architecture
- your dependency review process is defined before packages are added
- startup commands are repeatable in a clean environment

These checks reduce surprises when the runtime moves from a personal workstation to CI or production hosts. They also make troubleshooting easier because you know the baseline was valid before application-specific changes were introduced.

Final takeaway

Getting started with Node.js is mostly about establishing a clean, verifiable runtime: install the correct version, confirm the commands resolve properly, run one script, and prove that package management works. Once that baseline is in place, you can move on to project structure, dependencies, and application design with much less risk of chasing environment problems later.

Use this guidance together with Node.js production readiness checklist and ASP.NET programming to connect the workflow with related operational context already available on the site.