Overview

The Nest CLI is a command-line interface tool that helps you to initialize, develop, and maintain your Nest applications. It assists in multiple ways, including scaffolding the project, serving it in development mode, and building and bundling the application for production distribution. It embodies best-practice architectural patterns to encourage well-structured apps.

Installation

Note: In this guide we describe using npm to install packages, including the Nest CLI. Other package managers may be used at your discretion. With npm, you have several options available for managing how your OS command line resolves the location of the `nest` CLI binary file. Here, we describe installing the `nest` binary globally using the `-g` option. This provides a measure of convenience, and is the approach we assume throughout the documentation. Note that installing **any** `npm` package globally leaves the responsibility of ensuring they're running the correct version up to the user. It also means that if you have different projects, each will run the **same** version of the CLI. A reasonable alternative is to use the npx program, built into the `npm` cli (or similar features with other package managers) to ensure that you run a **managed version** of the Nest CLI. We recommend you consult the npx documentation and/or your DevOps support staff for more information.

Install the CLI globally using the `npm install -g` command (see the **Note** above for details about global installs).

```
$ npm install -g @nestjs/cli
```

> info **Hint** Alternatively, you can use this command `npx @nestjs/cli@latest` without installing the cli globally.

Basic workflow

Once installed, you can invoke CLI commands directly from your OS command line through the `nest` executable. See the available `nest` commands by entering the following:

```
$ nest --help
```

Get help on an individual command using the following construct. Substitute any command, like `new`, `add`, etc., where you see `generate` in the example below to get detailed help on that command:

```
$ nest generate --help
```

To create, build and run a new basic Nest project in development mode, go to the folder that should be the parent of your new project, and run the following commands:

```bash
$ nest new my-nest-project
$ cd my-nest-project
$ npm run start:dev
```

In your browser, open http://localhost:3000 to see the new application running. The app will automatically recompile and reload when you change any of the source files.

> info **Hint** We recommend using the SWC builder for faster builds (10x more performant than the default TypeScript compiler).

Project structure

When you run `nest new`, Nest generates a boilerplate application structure by creating a new folder and populating an initial set of files. You can continue working in this default structure, adding new components, as described throughout this documentation. We refer to the project structure generated by `nest new` as **standard mode**. Nest also supports an alternate structure for managing multiple projects and libraries called **monorepo mode**.

Aside from a few specific considerations around how the **build** process works (essentially, monorepo mode simplifies build complexities that can sometimes arise from monorepo-style project structures), and built-in library support, the rest of the Nest features, and this documentation, apply equally to both standard and monorepo mode project structures. In fact, you can easily switch from standard mode to monorepo mode at any time in the future, so you can safely defer this decision while you're still learning about Nest.

You can use either mode to manage multiple projects. Here's a quick summary of the differences:

Feature	Standard Mode	Monorepo Mode
Multiple projects	Separate file system structure	Single file system structure
`node_modules` & `package.json`	Separate instances	Shared across monorepo
Default compiler	`tsc`	webpack
Compiler settings	Specified separately	Monorepo defaults that can be overridden per project
Config files like `.eslintrc.js`, `.prettierrc`, etc.	Specified separately	Shared across monorepo
`nest build` and `nest start` commands	Target defaults automatically to the (only) project in the context	Target defaults to the **default project** in the monorepo
Libraries	Managed manually, usually via npm packaging	Built-in support, including path management and bundling

Read the sections on Workspaces and Libraries for more detailed information to help you decide which mode is most suitable for you.

CLI command syntax

All `nest` commands follow the same format:

```
nest commandOrAlias requiredArg [optionalArg] [options]
```

For example:

```
$ nest new my-nest-project --dry-run
```

Here, `new` is the *commandOrAlias*. The `new` command has an alias of `n`. `my-nest-project` is the *requiredArg*. If a *requiredArg* is not supplied on the command line, `nest` will prompt for it. Also, `--dry-run` has an equivalent short-hand form `-d`. With this in mind, the following command is the equivalent of the above:

```
$ nest n my-nest-project -d
```

Most commands, and some options, have aliases. Try running `nest new --help` to see these options and aliases, and to confirm your understanding of the above constructs.

Command overview

Run `nest <command> --help` for any of the following commands to see command-specific options.

See usage for detailed descriptions for each command.

Command	Alias	Description
`new`	`n`	Scaffolds a new *standard mode* application with all boilerplate files needed to run.
`generate`	`g`	Generates and/or modifies files based on a schematic.
`build`		Compiles an application or workspace into an output folder.
`start`		Compiles and runs an application (or default project in a workspace).
`add`		Imports a library that has been packaged as a **nest library**, running its install schematic.
`info`	`i`	Displays information about installed nest packages and other helpful system info.

Requirements

Nest CLI requires a Node.js binary built with internationalization support (ICU), such as the official binaries from the Node.js project page. If you encounter errors related to ICU, check that your binary meets this requirement.

```
node -p process.versions.icu
```

If the command prints `undefined`, your Node.js binary has no internationalization support.