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:

```
$ 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.

Workspaces

Nest has two modes for organizing code:

- **standard mode**: useful for building individual project-focused applications that have their own dependencies and settings, and don't need to optimize for sharing modules, or optimizing complex builds. This is the default mode.
- **monorepo mode**: this mode treats code artifacts as part of a lightweight **monorepo**, and may be more appropriate for teams of developers and/or multi-project environments. It automates parts of the build process to make it easy to create and compose modular components, promotes code re-use, makes integration testing easier, makes it easy to share project-wide artifacts like `eslint` rules and other configuration policies, and is easier to use than alternatives like github submodules. Monorepo mode employs the concept of a **workspace**, represented in the `nest-cli.json` file, to coordinate the relationship between the components of the monorepo.

It's important to note that virtually all of Nest's features are independent of your code organization mode. The **only** effect of this choice is how your projects are composed and how build artifacts are generated. All other functionality, from the CLI to core modules to add-on modules work the same in either mode.

Also, you can easily switch from **standard mode** to **monorepo mode** at any time, so you can delay this decision until the benefits of one or the other approach become more clear.

Standard mode

When you run `nest new`, a new **project** is created for you using a built-in schematic. Nest does the following:

1. Create a new folder, corresponding to the `name` argument you provide to `nest new`
2. Populate that folder with default files corresponding to a minimal base-level Nest application. You can examine these files at the typescript-starter repository.
3. Provide additional files such as `nest-cli.json`, `package.json` and `tsconfig.json` that configure and enable various tools for compiling, testing and serving your application.

From there, you can modify the starter files, add new components, add dependencies (e.g., `npm install`), and otherwise develop your application as covered in the rest of this documentation.

Monorepo mode

To enable monorepo mode, you start with a *standard mode* structure, and add **projects**. A project can be a full **application** (which you add to the workspace with the command `nest generate app`) or a **library** (which you add to the workspace with the command `nest generate library`). We'll discuss the details of these specific types of project components below. The key point to note now is that it is the **act of adding a project** to an existing standard mode structure that **converts it** to monorepo mode. Let's look at an example.

If we run:

```
$ nest new my-project
```

We've constructed a *standard mode* structure, with a folder structure that looks like this:

node_modules
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
nest-cli.json
package.json
tsconfig.json
.eslintrc.js

We can convert this to a monorepo mode structure as follows:

```
$ cd my-project
$ nest generate app my-app
```

At this point, `nest` converts the existing structure to a **monorepo mode** structure. This results in a few important changes. The folder structure now looks like this:

apps
my-app
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
tsconfig.app.json
my-project
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
tsconfig.app.json
nest-cli.json
package.json
tsconfig.json
.eslintrc.js

The `generate app` schematic has reorganized the code - moving each **application** project under the `apps` folder, and adding a project-specific `tsconfig.app.json` file in each project's root folder. Our original `my-project` app has become the **default project** for the monorepo, and is now a peer with the just-added `my-app`, located under the `apps` folder. We'll cover default projects below.

> error **Warning** The conversion of a standard mode structure to monorepo only works for projects that have followed the canonical Nest project structure. Specifically, during conversion, the schematic attempts to relocate the `src` and `test` folders in a project folder beneath the `apps` folder in the root. If a project does not use this structure, the conversion will fail or produce unreliable results.

Workspace projects

A monorepo uses the concept of a workspace to manage its member entities. Workspaces are composed of **projects**. A project may be either:

- an **application**: a full Nest application including a `main.ts` file to bootstrap the application. Aside from compile and build considerations, an application-type project within a workspace is functionally identical to an application within a *standard mode* structure.
- a **library**: a library is a way of packaging a general purpose set of features (modules, providers, controllers, etc.) that can be used within other projects. A library cannot run on its own, and has no `main.ts` file. Read more about libraries here.

All workspaces have a **default project** (which should be an application-type project). This is defined by the top-level `"root"` property in the `nest-cli.json` file, which points at the root of the default project (see CLI properties below for more details). Usually, this is the **standard mode** application you started with, and later converted to a monorepo using `nest generate app`. When you follow these steps, this property is populated automatically.

Default projects are used by `nest` commands like `nest build` and `nest start` when a project name is not supplied.

For example, in the above monorepo structure, running

```
$ nest start
```

will start up the `my-project` app. To start `my-app`, we'd use:

```
$ nest start my-app
```

Applications

Application-type projects, or what we might informally refer to as just "applications", are complete Nest applications that you can run and deploy. You generate an application-type project with `nest generate app`.

This command automatically generates a project skeleton, including the standard `src` and `test` folders from the typescript starter. Unlike standard mode, an application project in a monorepo does not have any of the package dependency (`package.json`) or other project configuration artifacts like `.prettierrc` and `.eslintrc.js`. Instead, the monorepo-wide dependencies and config files are used.

However, the schematic does generate a project-specific `tsconfig.app.json` file in the root folder of the project. This config file automatically sets appropriate build options, including setting the compilation output folder properly. The file extends the top-level (monorepo) `tsconfig.json` file, so you can manage global settings monorepo-wide, but override them if needed at the project level.

Libraries

As mentioned, library-type projects, or simply "libraries", are packages of Nest components that need to be composed into applications in order to run. You generate a library-type project with `nest generate library`. Deciding what belongs in a library is an architectural design decision. We discuss libraries in depth in the libraries chapter.

CLI properties

Nest keeps the metadata needed to organize, build and deploy both standard and monorepo structured projects in the `nest-cli.json` file. Nest automatically adds to and updates this file as you add projects, so you usually do not have to think about it or edit its contents. However, there are some settings you may want to change manually, so it's helpful to have an overview understanding of the file.

After running the steps above to create a monorepo, our `nest-cli.json` file looks like this:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "apps/my-project/src",
  "monorepo": true,
  "root": "apps/my-project",
  "compilerOptions": {
    "webpack": true,
    "tsConfigPath": "apps/my-project/tsconfig.app.json"
  },
  "projects": {
    "my-project": {
      "type": "application",
      "root": "apps/my-project",
      "entryFile": "main",
      "sourceRoot": "apps/my-project/src",
      "compilerOptions": {
        "tsConfigPath": "apps/my-project/tsconfig.app.json"
      }
    },
    "my-app": {
      "type": "application",
      "root": "apps/my-app",
      "entryFile": "main",
      "sourceRoot": "apps/my-app/src",
      "compilerOptions": {
        "tsConfigPath": "apps/my-app/tsconfig.app.json"
      }
    }
  }
}
```

The file is divided into sections:

- a global section with top-level properties controlling standard and monorepo-wide settings
- a top level property (`"projects"`) with metadata about each project. This section is present only for monorepo-mode structures.

The top-level properties are as follows:

- `"collection"`: points at the collection of schematics used to generate components; you generally should not change this value
- `"sourceRoot"`: points at the root of the source code for the single project in standard mode structures, or the *default project* in monorepo mode structures
- `"compilerOptions"`: a map with keys specifying compiler options and values specifying the option setting; see details below
- `"generateOptions"`: a map with keys specifying global generate options and values specifying the option setting; see details below
- `"monorepo"`: (monorepo only) for a monorepo mode structure, this value is always `true`
- `"root"`: (monorepo only) points at the project root of the *default project*

Global compiler options

These properties specify the compiler to use as well as various options that affect **any** compilation step, whether as part of `nest build` or `nest start`, and regardless of the compiler, whether `tsc` or webpack.

Property Name	Property Value Type	Description
`webpack`	boolean	If `true`, use webpack compiler. If `false` or not present, use `tsc`. In monorepo mode, the default is `true` (use webpack), in standard mode, the default is `false` (use `tsc`). See below for details. (deprecated: use `builder` instead)
`tsConfigPath`	string	(**monorepo only**) Points at the file containing the `tsconfig.json` settings that will be used when `nest build` or `nest start` is called without a `project` option (e.g., when the default project is built or started).
`webpackConfigPath`	string	Points at a webpack options file. If not specified, Nest looks for the file `webpack.config.js`. See below for more details.
`deleteOutDir`	boolean	If `true`, whenever the compiler is invoked, it will first remove the compilation output directory (as configured in `tsconfig.json`, where the default is `./dist`).
`assets`	array	Enables automatically distributing non-TypeScript assets whenever a compilation step begins (asset distribution does **not** happen on incremental compiles in `--watch` mode). See below for details.

Property Name	Property Value Type	Description
`watchAssets`	boolean	If `true`, run in watch-mode, watching **all** non-TypeScript assets. (For more fine-grained control of the assets to watch, see Assets section below).
`manualRestart`	boolean	If `true`, enables the shortcut `rs` to manually restart the server. Default value is `false`.
`builder`	string/object	Instructs CLI on what `builder` to use to compile the project (`tsc`, `swc`, or `webpack`). To customize builder's behavior, you can pass an object containing two attributes: `type` (`tsc`, `swc`, or `webpack`) and `options`.
`typeCheck`	boolean	If `true`, enables type checking for SWC-driven projects (when `builder` is `swc`). Default value is `false`.

Global generate options

These properties specify the default generate options to be used by the `nest generate` command.

Property Name	Property Value Type	Description
`spec`	boolean *or* object	If the value is boolean, a value of `true` enables `spec` generation by default and a value of `false` disables it. A flag passed on the CLI command line overrides this setting, as does a project-specific `generateOptions` setting (more below). If the value is an object, each key represents a schematic name, and the boolean value determines whether the default spec generation is enabled / disabled for that specific schematic.
`flat`	boolean	If true, all generate commands will generate a flat structure

The following example uses a boolean value to specify that spec file generation should be disabled by default for all projects:

```
{
  "generateOptions": {
    "spec": false
  },
  ...
}
```

The following example uses a boolean value to specify flat file generation should be the default for all projects:

```
{
  "generateOptions": {
    "flat": true
  },
  ...
}
```

In the following example, `spec` file generation is disabled only for `service` schematics (e.g., `nest generate service...`):

```
{
  "generateOptions": {
    "spec": {
      "service": false
    }
  },
  ...
}
```

warning **Warning** When specifying the `spec` as an object, the key for the generation schematic does not currently support automatic alias handling. This means that specifying a key as for example `service: false` and trying to generate a service via the alias `s`, the spec would still be generated. To make sure both the normal schematic name and the alias work as intended, specify both the normal command name as well as the alias, as seen below.

```
{
  "generateOptions": {
    "spec": {
      "service": false,
      "s": false
    }
  },
  ...
}
```

Project-specific generate options

In addition to providing global generate options, you may also specify project-specific generate options. The project specific generate options follow the exact same format as the global generate options, but are specified directly on each project.

Project-specific generate options override global generate options.

```
{
  "projects": {
```

```
    "cats-project": {
      "generateOptions": {
        "spec": {
          "service": false
        }
      },
      ...
    }
  },
  ...
}
```

> warning **Warning** The order of precedence for generate options is as follows. Options specified on the CLI command line take precedence over project-specific options. Project-specific options override global options.

Specified compiler

The reason for the different default compilers is that for larger projects (e.g., more typical in a monorepo) webpack can have significant advantages in build times and in producing a single file bundling all project components together. If you wish to generate individual files, set `"webpack"` to `false`, which will cause the build process to use `tsc` (or `swc`).

Webpack options

The webpack options file can contain standard webpack configuration options. For example, to tell webpack to bundle `node_modules` (which are excluded by default), add the following to `webpack.config.js`:

```
module.exports = {
  externals: [],
};
```

Since the webpack config file is a JavaScript file, you can even expose a function that takes default options and returns a modified object:

```
module.exports = function (options) {
  return {
    ...options,
    externals: [],
  };
};
```

Assets

TypeScript compilation automatically distributes compiler output (`.js` and `.d.ts` files) to the specified output directory. It can also be convenient to distribute non-TypeScript files, such as `.graphql` files, `images`, `.html` files and other assets. This allows you to treat `nest build` (and any initial compilation step) as a lightweight **development build** step, where you may be editing non-TypeScript files and iteratively compiling and testing. The assets should be located in the `src` folder otherwise they will not be copied.

The value of the `assets` key should be an array of elements specifying the files to be distributed. The elements can be simple strings with `glob`-like file specs, for example:

```
"assets": ["**/*.graphql"],
"watchAssets": true,
```

For finer control, the elements can be objects with the following keys:

- `"include"`: `glob`-like file specifications for the assets to be distributed
- `"exclude"`: `glob`-like file specifications for assets to be **excluded** from the `include` list
- `"outDir"`: a string specifying the path (relative to the root folder) where the assets should be distributed. Defaults to the same output directory configured for compiler output.
- `"watchAssets"`: boolean; if `true`, run in watch mode watching specified assets

For example:

```
"assets": [
  { "include": "**/*.graphql", "exclude": "**/omitted.graphql",
"watchAssets": true },
]
```

> warning **Warning** Setting `watchAssets` in a top-level `compilerOptions` property overrides any `watchAssets` settings within the `assets` property.

Project properties

This element exists only for monorepo-mode structures. You generally should not edit these properties, as they are used by Nest to locate projects and their configuration options within the monorepo.

Libraries

Many applications need to solve the same general problems, or re-use a modular component in several different contexts. Nest has a few ways of addressing this, but each works at a different level to solve the problem in a way that helps meet different architectural and organizational objectives.

Nest modules are useful for providing an execution context that enables sharing components within a single application. Modules can also be packaged with npm to create a reusable library that can be installed in different projects. This can be an effective way to distribute configurable, re-usable libraries that can be used by different, loosely connected or unaffiliated organizations (e.g., by distributing/installing 3rd party libraries).

For sharing code within closely organized groups (e.g., within company/project boundaries), it can be useful to have a more lightweight approach to sharing components. Monorepos have arisen as a construct to enable that, and within a monorepo, a **library** provides a way to share code in an easy, lightweight fashion. In a Nest monorepo, using libraries enables easy assembly of applications that share components. In fact, this encourages decomposition of monolithic applications and development processes to focus on building and composing modular components.

Nest libraries

A Nest library is a Nest project that differs from an application in that it cannot run on its own. A library must be imported into a containing application in order for its code to execute. The built-in support for libraries described in this section is only available for **monorepos** (standard mode projects can achieve similar functionality using npm packages).

For example, an organization may develop an `AuthModule` that manages authentication by implementing company policies that govern all internal applications. Rather than build that module separately for each application, or physically packaging the code with npm and requiring each project to install it, a monorepo can define this module as a library. When organized this way, all consumers of the library module can see an up-to-date version of the `AuthModule` as it is committed. This can have significant benefits for coordinating component development and assembly, and simplifying end-to-end testing.

Creating libraries

Any functionality that is suitable for re-use is a candidate for being managed as a library. Deciding what should be a library, and what should be part of an application, is an architectural design decision. Creating libraries involves more than simply copying code from an existing application to a new library. When packaged as a library, the library code must be decoupled from the application. This may require **more** time up front and force some design decisions that you may not face with more tightly coupled code. But this additional effort can pay off when the library can be used to enable more rapid application assembly across multiple applications.

To get started with creating a library, run the following command:

```
$ nest g library my-library
```

When you run the command, the `library` schematic prompts you for a prefix (AKA alias) for the library:

```
What prefix would you like to use for the library (default: @app)?
```

This creates a new project in your workspace called `my-library`. A library-type project, like an application-type project, is generated into a named folder using a schematic. Libraries are managed under the `libs` folder of the monorepo root. Nest creates the `libs` folder the first time a library is created.

The files generated for a library are slightly different from those generated for an application. Here is the contents of the `libs` folder after executing the command above:

libs
my-library
src
index.ts
my-library.module.ts
my-library.service.ts
tsconfig.lib.json

The `nest-cli.json` file will have a new entry for the library under the `"projects"` key:

```
...
{
    "my-library": {
      "type": "library",
      "root": "libs/my-library",
      "entryFile": "index",
      "sourceRoot": "libs/my-library/src",
      "compilerOptions": {
        "tsConfigPath": "libs/my-library/tsconfig.lib.json"
      }
}
...
```

There are two differences in `nest-cli.json` metadata between libraries and applications:

- the `"type"` property is set to `"library"` instead of `"application"`
- the `"entryFile"` property is set to `"index"` instead of `"main"`

These differences key the build process to handle libraries appropriately. For example, a library exports its functions through the `index.js` file.

As with application-type projects, libraries each have their own `tsconfig.lib.json` file that extends the root (monorepo-wide) `tsconfig.json` file. You can modify this file, if necessary, to provide library-specific compiler options.

You can build the library with the CLI command:

```
$ nest build my-library
```

Using libraries

With the automatically generated configuration files in place, using libraries is straightforward. How would we import `MyLibraryService` from the `my-library` library into the `my-project` application?

First, note that using library modules is the same as using any other Nest module. What the monorepo does is manage paths in a way that importing libraries and generating builds is now transparent. To use `MyLibraryService`, we need to import its declaring module. We can modify `my-project/src/app.module.ts` as follows to import `MyLibraryModule`.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { MyLibraryModule } from '@app/my-library';

@Module({
  imports: [MyLibraryModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Notice above that we've used a path alias of `@app` in the ES module `import` line, which was the `prefix` we supplied with the `nest g library` command above. Under the covers, Nest handles this through tsconfig path mapping. When adding a library, Nest updates the global (monorepo) `tsconfig.json` file's `"paths"` key like this:

```
"paths": {
    "@app/my-library": [
        "libs/my-library/src"
    ],
    "@app/my-library/*": [
        "libs/my-library/src/*"
    ]
}
```

So, in a nutshell, the combination of the monorepo and library features has made it easy and intuitive to include library modules into applications.

This same mechanism enables building and deploying applications that compose libraries. Once you've imported the `MyLibraryModule`, running `nest build` handles all the module resolution automatically and bundles the app along with any library dependencies, for deployment. The default compiler for a

monorepo is **webpack**, so the resulting distribution file is a single file that bundles all of the transpiled JavaScript files into a single file. You can also switch to `tsc` as described here.

CLI command reference

nest new

Creates a new (standard mode) Nest project.

```
$ nest new <name> [options]
$ nest n <name> [options]
```

Description

Creates and initializes a new Nest project. Prompts for package manager.

- Creates a folder with the given `<name>`
- Populates the folder with configuration files
- Creates sub-folders for source code (`/src`) and end-to-end tests (`/test`)
- Populates the sub-folders with default files for app components and tests

Arguments

Argument	Description
`<name>`	The name of the new project

Options

Option	Description
`--dry-run`	Reports changes that would be made, but does not change the filesystem. Alias: `-d`
`--skip-git`	Skip git repository initialization. Alias: `-g`
`--skip-install`	Skip package installation. Alias: `-s`
`--package-manager [package-manager]`	Specify package manager. Use `npm`, `yarn`, or `pnpm`. Package manager must be installed globally. Alias: `-p`
`--language [language]`	Specify programming language (`TS` or `JS`). Alias: `-l`
`--collection [collectionName]`	Specify schematics collection. Use package name of installed npm package containing schematic. Alias: `-c`

Option	Description
`--strict`	Start the project with the following TypeScript compiler flags enabled: `strictNullChecks`, `noImplicitAny`, `strictBindCallApply`, `forceConsistentCasingInFileNames`, `noFallthroughCasesInSwitch`

nest generate

Generates and/or modifies files based on a schematic

```
$ nest generate <schematic> <name> [options]
$ nest g <schematic> <name> [options]
```

Arguments

Argument	Description
`<schematic>`	The `schematic` or `collection:schematic` to generate. See the table below for the available schematics.
`<name>`	The name of the generated component.

Schematics

Name	Alias	Description
`app`		Generate a new application within a monorepo (converting to monorepo if it's a standard structure).
`library`	`lib`	Generate a new library within a monorepo (converting to monorepo if it's a standard structure).
`class`	`cl`	Generate a new class.
`controller`	`co`	Generate a controller declaration.
`decorator`	`d`	Generate a custom decorator.
`filter`	`f`	Generate a filter declaration.
`gateway`	`ga`	Generate a gateway declaration.
`guard`	`gu`	Generate a guard declaration.
`interface`	`itf`	Generate an interface.
`interceptor`	`itc`	Generate an interceptor declaration.
`middleware`	`mi`	Generate a middleware declaration.
`module`	`mo`	Generate a module declaration.

Name	Alias	Description
`pipe`	`pi`	Generate a pipe declaration.
`provider`	`pr`	Generate a provider declaration.
`resolver`	`r`	Generate a resolver declaration.
`resource`	`res`	Generate a new CRUD resource. See the CRUD (resource) generator for more details.
`service`	`s`	Generate a service declaration.

Options

Option	Description
`--dry-run`	Reports changes that would be made, but does not change the filesystem. Alias: `-d`
`--project [project]`	Project that element should be added to. Alias: `-p`
`--flat`	Do not generate a folder for the element.
`--collection [collectionName]`	Specify schematics collection. Use package name of installed npm package containing schematic. Alias: `-c`
`--spec`	Enforce spec files generation (default)
`--no-spec`	Disable spec files generation

nest build

Compiles an application or workspace into an output folder.

Also, the `build` command is responsible for:

- mapping paths (if using path aliases) via `tsconfig-paths`
- annotating DTOs with OpenAPI decorators (if `@nestjs/swagger` CLI plugin is enabled)
- annotating DTOs with GraphQL decorators (if `@nestjs/graphql` CLI plugin is enabled)

```
$ nest build <name> [options]
```

Arguments

Argument	Description
`<name>`	The name of the project to build.

Options

Option	Description
`--path [path]`	Path to `tsconfig` file. Alias `-p`
`--config [path]`	Path to `nest-cli` configuration file. Alias `-c`
`--watch`	Run in watch mode (live-reload). If you're using `tsc` for compilation, you can type `rs` to restart the application (when `manualRestart` option is set to `true`). Alias `-w`
`--builder [name]`	Specify the builder to use for compilation (`tsc`, `swc`, or `webpack`). Alias `-b`
`--webpack`	Use webpack for compilation (deprecated: use `--builder webpack` instead).
`--webpackPath`	Path to webpack configuration.
`--tsc`	Force use `tsc` for compilation.

nest start

Compiles and runs an application (or default project in a workspace).

```
$ nest start <name> [options]
```

Arguments

Argument	Description
`<name>`	The name of the project to run.

Options

Option	Description
`--path [path]`	Path to `tsconfig` file. Alias `-p`
`--config [path]`	Path to `nest-cli` configuration file. Alias `-c`
`--watch`	Run in watch mode (live-reload) Alias `-w`

Option	Description
`--builder [name]`	Specify the builder to use for compilation (`tsc`, `swc`, or `webpack`). Alias `-b`
`--preserveWatchOutput`	Keep outdated console output in watch mode instead of clearing the screen. (`tsc` watch mode only)
`--watchAssets`	Run in watch mode (live-reload), watching non-TS files (assets). See Assets for more details.
`--debug [hostport]`	Run in debug mode (with --inspect flag) Alias `-d`
`--webpack`	Use webpack for compilation. (deprecated: use `--builder webpack` instead)
`--webpackPath`	Path to webpack configuration.
`--tsc`	Force use `tsc` for compilation.
`--exec [binary]`	Binary to run (default: `node`). Alias `-e`

nest add

Imports a library that has been packaged as a **nest library**, running its install schematic.

```
$ nest add <name> [options]
```

Arguments

Argument	Description
`<name>`	The name of the library to import.

nest info

Displays information about installed nest packages and other helpful system info. For example:

```
$ nest info
```

```
 _   _           _      ___  _____  _____  _     _____
| \ | |         | |    |_  |/  ___|/  __ \| |   |_   _|
|  \| |  ___  ___ | |_     | |\ `--. | /  \/| |     | |
| . ` | / _ \/ __|| __|    | | `--. \| |    | |     | |
| |\  ||  __/\__ \| |_ /\__/ //\__/ /| \__/\| |_____| |_
```

```
\_| \_/ \___||___/ \__|\____/ \____/  \____/\_____/\___/

[System Information]
OS Version : macOS High Sierra
NodeJS Version : v16.18.0
[Nest Information]
microservices version : 10.0.0
websockets version : 10.0.0
testing version : 10.0.0
common version : 10.0.0
core version : 10.0.0
```

Nest CLI and scripts

This section provides additional background on how the `nest` command interacts with compilers and scripts to help DevOps personnel manage the development environment.

A Nest application is a **standard** TypeScript application that needs to be compiled to JavaScript before it can be executed. There are various ways to accomplish the compilation step, and developers/teams are free to choose a way that works best for them. With that in mind, Nest provides a set of tools out-of-the-box that seek to do the following:

- Provide a standard build/execute process, available at the command line, that "just works" with reasonable defaults.
- Ensure that the build/execute process is **open**, so developers can directly access the underlying tools to customize them using native features and options.
- Remain a completely standard TypeScript/Node.js framework, so that the entire compile/deploy/execute pipeline can be managed by any external tools that the development team chooses to use.

This goal is accomplished through a combination of the `nest` command, a locally installed TypeScript compiler, and `package.json` scripts. We describe how these technologies work together below. This should help you understand what's happening at each step of the build/execute process, and how to customize that behavior if necessary.

The nest binary

The `nest` command is an OS level binary (i.e., runs from the OS command line). This command actually encompasses 3 distinct areas, described below. We recommend that you run the build (`nest build`) and execution (`nest start`) sub-commands via the `package.json` scripts provided automatically when a project is scaffolded (see typescript starter if you wish to start by cloning a repo, instead of running `nest new`).

Build

`nest build` is a wrapper on top of the standard `tsc` compiler or `swc` compiler (for standard projects) or the webpack bundler using the `ts-loader` (for monorepos). It does not add any other compilation features or steps except for handling `tsconfig-paths` out of the box. The reason it exists is that most developers, especially when starting out with Nest, do not need to adjust compiler options (e.g., `tsconfig.json` file) which can sometimes be tricky.

See the nest build documentation for more details.

Execution

`nest start` simply ensures the project has been built (same as `nest build`), then invokes the `node` command in a portable, easy way to execute the compiled application. As with builds, you are free to customize this process as needed, either using the `nest start` command and its options, or completely replacing it. The entire process is a standard TypeScript application build and execute pipeline, and you are free to manage the process as such.

See the nest start documentation for more details.

Generation

The `nest generate` commands, as the name implies, generate new Nest projects, or components within them.

Package scripts

Running the `nest` commands at the OS command level requires that the `nest` binary be installed globally. This is a standard feature of npm, and outside of Nest's direct control. One consequence of this is that the globally installed `nest` binary is **not** managed as a project dependency in `package.json`. For example, two different developers can be running two different versions of the `nest` binary. The standard solution for this is to use package scripts so that you can treat the tools used in the build and execute steps as development dependencies.

When you run `nest new`, or clone the typescript starter, Nest populates the new project's `package.json` scripts with commands like `build` and `start`. It also installs the underlying compiler tools (such as `typescript`) as **dev dependencies**.

You run the build and execute scripts with commands like:

```
$ npm run build
```

and

```
$ npm run start
```

These commands use npm's script running capabilities to execute `nest build` or `nest start` using the **locally installed** `nest` binary. By using these built-in package scripts, you have full dependency management over the Nest CLI commands*. This means that, by following this **recommended** usage, all members of your organization can be assured of running the same version of the commands.

*This applies to the `build` and `start` commands. The `nest new` and `nest generate` commands aren't part of the build/execute pipeline, so they operate in a different context, and do not come with built-in `package.json` scripts.

For most developers/teams, it is recommended to utilize the package scripts for building and executing their Nest projects. You can fully customize the behavior of these scripts via their options (`--path`, `--webpack`, `--webpackPath`) and/or customize the `tsc` or webpack compiler options files (e.g., `tsconfig.json`) as needed. You are also free to run a completely custom build process to compile the TypeScript (or even to execute TypeScript directly with `ts-node`).

Backward compatibility

Because Nest applications are pure TypeScript applications, previous versions of the Nest build/execute scripts will continue to operate. You are not required to upgrade them. You can choose to take advantage of the new `nest build` and `nest start` commands when you are ready, or continue running previous or customized scripts.

Migration

While you are not required to make any changes, you may want to migrate to using the new CLI commands instead of using tools such as `tsc-watch` or `ts-node`. In this case, simply install the latest version of the `@nestjs/cli`, both globally and locally:

```
$ npm install -g @nestjs/cli
$ cd  /some/project/root/folder
$ npm install -D @nestjs/cli
```

You can then replace the `scripts` defined in `package.json` with the following ones:

```
"build": "nest build",
"start": "nest start",
"start:dev": "nest start --watch",
"start:debug": "nest start --debug --watch",
```