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.