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",
```