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.