CLI Plugin

TypeScript's metadata reflection system has several limitations which make it impossible to, for instance, determine what properties a class consists of or recognize whether a given property is optional or required. However, some of these constraints can be addressed at compilation time. Nest provides a plugin that enhances the TypeScript compilation process to reduce the amount of boilerplate code required.

> info **Hint** This plugin is **opt-in**. If you prefer, you can declare all decorators manually, or only specific decorators where you need them.

Overview

The Swagger plugin will automatically:

- annotate all DTO properties with `@ApiProperty` unless `@ApiHideProperty` is used
- set the `required` property depending on the question mark (e.g. `name?: string` will set `required: false`)
- set the `type` or `enum` property depending on the type (supports arrays as well)
- set the `default` property based on the assigned default value
- set several validation rules based on `class-validator` decorators (if `classValidatorShim` set to `true`)
- add a response decorator to every endpoint with a proper status and `type` (response model)
- generate descriptions for properties and endpoints based on comments (if `introspectComments` set to `true`)
- generate example values for properties based on comments (if `introspectComments` set to `true`)

Please, note that your filenames **must have** one of the following suffixes: `['.dto.ts', '.entity.ts']` (e.g., `create-user.dto.ts`) in order to be analysed by the plugin.

If you are using a different suffix, you can adjust the plugin's behavior by specifying the `dtoFileNameSuffix` option (see below).

Previously, if you wanted to provide an interactive experience with the Swagger UI, you had to duplicate a lot of code to let the package know how your models/components should be declared in the specification. For example, you could define a simple `CreateUserDto` class as follows:

```
export class CreateUserDto {
  @ApiProperty()
  email: string;

  @ApiProperty()
  password: string;

  @ApiProperty({ enum: RoleEnum, default: [], isArray: true })
  roles: RoleEnum[] = [];

  @ApiProperty({ required: false, default: true })
  isEnabled?: boolean = true;
}
```

While not a significant issue with medium-sized projects, it becomes verbose & hard to maintain once you have a large set of classes.

By enabling the Swagger plugin, the above class definition can be declared simply:

```
export class CreateUserDto {
  email: string;
  password: string;
  roles: RoleEnum[] = [];
  isEnabled?: boolean = true;
}
```

> info **Note** The Swagger plugin will derive the @ApiProperty() annotations from the TypeScript types and class-validator decorators. This helps in clearly describing your API for the generated Swagger UI documentation. However, the validation at runtime would still be handled by class-validator decorators. So, it is required to continue using validators like `IsEmail()`, `IsNumber()`, etc.

Hence, if you intend to relay on automatic annotations for generating documentations and still wish for runtime validations, then the class-validator decorators are still necessary.

> info **Hint** When using mapped types utilities (like `PartialType`) in DTOs import them from `@nestjs/swagger` instead of `@nestjs/mapped-types` for the plugin to pick up the schema.

The plugin adds appropriate decorators on the fly based on the **Abstract Syntax Tree**. Thus you won't have to struggle with `@ApiProperty` decorators scattered throughout the code.

> info **Hint** The plugin will automatically generate any missing swagger properties, but if you need to override them, you simply set them explicitly via `@ApiProperty()`.

Comments introspection

With the comments introspection feature enabled, CLI plugin will generate descriptions and example values for properties based on comments.

For example, given an example `roles` property:

```
/**
 * A list of user's roles
 * @example ['admin']
 */
@ApiProperty({
  description: `A list of user's roles`,
  example: ['admin'],
})
roles: RoleEnum[] = [];
```

You must duplicate both description and example values. With `introspectComments` enabled, the CLI plugin can extract these comments and automatically provide descriptions (and examples, if defined) for properties. Now, the above property can be declared simply as follows:

```
/**
 * A list of user's roles
 * @example ['admin']
 */
roles: RoleEnum[] = [];
```

There are `dtoKeyOfComment` and `controllerKeyOfComment` plugin options that you can use to customize how the plugin will set the value for `ApiProperty` and `ApiOperation` decorators respectively. Take a look at the following example:

```
export class SomeController {
  /**
   * Create some resource
   */
  @Post()
  create() {}
}
```

By default, these options are set to `"description"`. This means the plugin will assign `"Create some resource"` to `description` key on the `ApiOperation` operator. Like so:

```
@ApiOperation({ description: "Create some resource" })
```

> info **Hint** For models, the same logic applies but to `ApiProperty` decorator instead.

Using the CLI plugin

To enable the plugin, open `nest-cli.json` (if you use Nest CLI) and add the following `plugins` configuration:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src",
  "compilerOptions": {
    "plugins": ["@nestjs/swagger"]
  }
}
```

You can use the `options` property to customize the behavior of the plugin.

```
"plugins": [
  {
    "name": "@nestjs/swagger",
    "options": {
      "classValidatorShim": false,
      "introspectComments": true
    }
  }
]
```

The `options` property has to fulfill the following interface:

```
export interface PluginOptions {
  dtoFileNameSuffix?: string[];
  controllerFileNameSuffix?: string[];
  classValidatorShim?: boolean;
  dtoKeyOfComment?: string;
  controllerKeyOfComment?: string;
  introspectComments?: boolean;
}
```

Option	Default	Description
`dtoFileNameSuffix`	`['.dto.ts', '.entity.ts']`	DTO (Data Transfer Object) files suffix
`controllerFileNameSuffix`	`.controller.ts`	Controller files suffix
`classValidatorShim`	`true`	If set to true, the module will reuse `class-validator` validation decorators (e.g. `@Max(10)` will add `max: 10` to schema definition)
`dtoKeyOfComment`	`'description'`	The property key to set the comment text to on `ApiProperty`.
`controllerKeyOfComment`	`'description'`	The property key to set the comment text to on `ApiOperation`.
`introspectComments`	`false`	If set to true, plugin will generate descriptions and example values for properties based on comments

Make sure to delete the `/dist` folder and rebuild your application whenever plugin options are updated. If you don't use the CLI but instead have a custom `webpack` configuration, you can use this plugin in combination with `ts-loader`:

```
getCustomTransformers: (program: any) => ({
  before: [require('@nestjs/swagger/plugin').before({}, program)]
}),
```

SWC builder

For standard setups (non-monorepo), to use CLI Plugins with the SWC builder, you need to enable type checking, as described here.

```
$ nest start -b swc --type-check
```

For monorepo setups, follow the instructions here.

```
$ npx ts-node src/generate-metadata.ts
# OR npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

Now, the serialized metadata file must be loaded by the `SwaggerModule#loadPluginMetadata` method, as shown below:

```
import metadata from './metadata'; // <-- file auto-generated by the
"PluginMetadataGenerator"

await SwaggerModule.loadPluginMetadata(metadata); // <-- here
const document = SwaggerModule.createDocument(app, config);
```

Integration with `ts-jest` (e2e tests)

To run e2e tests, `ts-jest` compiles your source code files on the fly, in memory. This means, it doesn't use Nest CLI compiler and does not apply any plugins or perform AST transformations.

To enable the plugin, create the following file in your e2e tests directory:

```
const transformer = require('@nestjs/swagger/plugin');

module.exports.name = 'nestjs-swagger-transformer';
// you should change the version number anytime you change the
configuration below - otherwise, jest will not detect changes
module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
      // @nestjs/swagger/plugin options (can be empty)
```

```
    },
    cs.program, // "cs.tsCompiler.program" for older versions of Jest (<=
v27)
  );
};
```

With this in place, import AST transformer within your `jest` configuration file. By default (in the starter application), e2e tests configuration file is located under the `test` folder and is named `jest-e2e.json`.

```
{
  ... // other configuration
  "globals": {
    "ts-jest": {
      "astTransformers": {
        "before": ["<path to the file created above>"]
      }
    }
  }
}
```

If you use `jest@^29`, then use the snippet below, as the previous approach got deprecated.

```
{
  ... // other configuration
  "transform": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "before": ["<path to the file created above>"]
        }
      }
    ]
  }
}
```

Troubleshooting `jest` (e2e tests)

In case `jest` does not seem to pick up your configuration changes, it's possible that Jest has already **cached** the build result. To apply the new configuration, you need to clear Jest's cache directory.

To clear the cache directory, run the following command in your NestJS project folder:

```
$ npx jest --clearCache
```

In case the automatic cache clearance fails, you can still manually remove the cache folder with the following commands:

```
# Find jest cache directory (usually /tmp/jest_rs)
# by running the following command in your NestJS project root
$ npx jest --showConfig | grep cache
# ex result:
#   "cache": true,
#   "cacheDirectory": "/tmp/jest_rs"

# Remove or empty the Jest cache directory
$ rm -rf  <cacheDirectory value>
# ex:
# rm -rf /tmp/jest_rs
```