Nest Commander

Expanding on the standalone application docs there's also the nest-commander package for writing command line applications in a structure similar to your typical Nest application.

> info **info** `nest-commander` is a third party package and is not managed by the entirety of the NestJS core team. Please, report any issues found with the library in the appropriate repository

Installation

Just like any other package, you've got to install it before you can use it.

```
$ npm i nest-commander
```

A Command file

`nest-commander` makes it easy to write new command-line applications with decorators via the `@Command()` decorator for classes and the `@Option()` decorator for methods of that class. Every command file should implement the `CommandRunner` abstract class and should be decorated with a `@Command()` decorator.

Every command is seen as an `@Injectable()` by Nest, so your normal Dependency Injection still works as you would expect it to. The only thing to take note of is the abstract class `CommandRunner`, which should be implemented by each command. The `CommandRunner` abstract class ensures that all commands have a `run` method that returns a `Promise<void>` and takes in the parameters `string[], Record<string, any>`. The `run` command is where you can kick all of your logic off from, it will take in whatever parameters did not match option flags and pass them in as an array, just in case you are really meaning to work with multiple parameters. As for the options, the `Record<string, any>`, the names of these properties match the `name` property given to the `@Option()` decorators, while their value matches the return of the option handler. If you'd like better type safety, you are welcome to create an interface for your options as well.

Running the Command

Similar to how in a NestJS application we can use the `NestFactory` to create a server for us, and run it using `listen`, the `nest-commander` package exposes a simple to use API to run your server. Import the `CommandFactory` and use the `static` method `run` and pass in the root module of your application. This would probably look like below

```
import { CommandFactory } from 'nest-commander';
import { AppModule } from './app.module';

async function bootstrap() {
  await CommandFactory.run(AppModule);
}

bootstrap();
```

By default, Nest's logger is disabled when using the `CommandFactory`. It's possible to provide it though, as the second argument to the `run` function. You can either provide a custom NestJS logger, or an array of log levels you want to keep - it might be useful to at least provide `['error']` here, if you only want to print out Nest's error logs.

```
import { CommandFactory } from 'nest-commander';
import { AppModule } from './app.module';
import { LogService } './log.service';

async function bootstrap() {
  await CommandFactory.run(AppModule, new LogService());

  // or, if you only want to print Nest's warnings and errors
  await CommandFactory.run(AppModule, ['warn', 'error']);
}

bootstrap();
```

And that's it. Under the hood, `CommandFactory` will worry about calling `NestFactory` for you and calling `app.close()` when necessary, so you shouldn't need to worry about memory leaks there. If you need to add in some error handling, there's always `try/catch` wrapping the `run` command, or you can chain on some `.catch()` method to the `bootstrap()` call.

Testing

So what's the use of writing a super awesome command line script if you can't test it super easily, right? Fortunately, `nest-commander` has some utilities you can make use of that fits in perfectly with the NestJS ecosystem, it'll feel right at home to any Nestlings out there. Instead of using the `CommandFactory` for building the command in test mode, you can use `CommandTestFactory` and pass in your metadata, very similarly to how `Test.createTestingModule` from `@nestjs/testing` works. In fact, it uses this package under the hood. You're also still able to chain on the `overrideProvider` methods before calling `compile()` so you can swap out DI pieces right in the test.

Putting it all together

The following class would equate to having a CLI command that can take in the subcommand `basic` or be called directly, with `-n`, `-s`, and `-b` (along with their long flags) all being supported and with custom parsers for each option. The `--help` flag is also supported, as is customary with commander.

```
import { Command, CommandRunner, Option } from 'nest-commander';
import { LogService } from './log.service';

interface BasicCommandOptions {
  string?: string;
  boolean?: boolean;
  number?: number;
```

```
}

@Command({ name: 'basic', description: 'A parameter parse' })
export class BasicCommand extends CommandRunner {
  constructor(private readonly logService: LogService) {
    super()
  }

  async run(
    passedParam: string[],
    options?: BasicCommandOptions,
  ): Promise<void> {
    if (options?.boolean !== undefined && options?.boolean !== null) {
      this.runWithBoolean(passedParam, options.boolean);
    } else if (options?.number) {
      this.runWithNumber(passedParam, options.number);
    } else if (options?.string) {
      this.runWithString(passedParam, options.string);
    } else {
      this.runWithNone(passedParam);
    }
  }

  @Option({
    flags: '-n, --number [number]',
    description: 'A basic number parser',
  })
  parseNumber(val: string): number {
    return Number(val);
  }

  @Option({
    flags: '-s, --string [string]',
    description: 'A string return',
  })
  parseString(val: string): string {
    return val;
  }

  @Option({
    flags: '-b, --boolean [boolean]',
    description: 'A boolean parser',
  })
  parseBoolean(val: string): boolean {
    return JSON.parse(val);
  }

  runWithString(param: string[], option: string): void {
    this.logService.log({ param, string: option });
  }

  runWithNumber(param: string[], option: number): void {
    this.logService.log({ param, number: option });
  }
  runWithBoolean(param: string[], option: boolean): void {
    this.logService.log({ param, boolean: option });
  }

  runWithNone(param: string[]): void {
    this.logService.log({ param });
  }
}
```

Make sure the command class is added to a module

```
@Module({
  providers: [LogService, BasicCommand],
})
export class AppModule {}
```

And now to be able to run the CLI in your main.ts you can do the following

```
async function bootstrap() {
  await CommandFactory.run(AppModule);
}

bootstrap();
```

And just like that, you've got a command line application.

More Information

Visit the nest-commander docs site for more information, examples, and API documentation.