Published in The Crowdlinker Chronicle

Prateek Kathal
Jul 17, 2020 · 6 min read · Listen

Best Way to Structure Your Directory/Code (NestJS)

I've been building apps using NestJS for quite a while now and I want to share the best way to structure your directory/code with other developers new to this framework.

Before we begin, keep in mind that this structure is quite generic and **might not work** with some techniques like GraphQL. You could however modify it as per your requirements.

Now, let's get straight to the point!

Directory Structure 📂

These are just the directory names. I will give more insights into how the files look inside these directories in their own particular section.

```
src
├── authentication
├── common
│   ├── constants
│   ├── decorators
│   │   ├── metadata
│   │   └── requests
│   ├── exceptions
│   ├── guards
│   ├── helpers
│   │   ├── exceptions
│   │   └── responses
│   ├── interfaces
│   ├── middlewares
│   │   └── models
│   ├── pipes
│   ├── serializers
│   │   ├── exceptions
│   │   └── responses
│   └── validations
├── config
│   ├── api
│   ├── app
│   ├── cache
│   ├── database
│   │   └── postgres
│   ├── queue
│   ├── session
│   └── storage
├── database
```

```
│   ├── factories
│   │   ├── addresses
│   │   └── users
│   ├── migrations
│   └── seeders
│       ├── addresses
│       └── users
├── jobs
│   ├── consumers
│   │   └── verification-mail
│   └── producers
│       └── verification-mail
├── mails
│   └── verification
├── models
│   ├── addresses
│   │   ├── constants
│   │   ├── entities
│   │   ├── interfaces
│   │   └── serializers
│   └── users
│       ├── constants
│       ├── entities
│       ├── interfaces
│       └── serializers
├── providers
│   ├── cache
│   │   └── redis
│   ├── database
│   │   └── postgres
│   └── queue
│       └── redis
├── app.controller.ts
├── app.module.ts
├── app.service.ts
├── main.ts
└── seed.ts
```

Config ⚙

Let's begin our tutorial with the initialization of our environment variables. I am using the package @nestjs/config in this use case.

The `config/` folder consists of different sections of variable types to be loaded into our environment.

```
src/config
├── app
│   ├── config.module.ts
│   ├── config.service.ts
│   └── configuration.ts
├── cache
│   ├── config.module.ts
│   ├── config.service.ts
│   └── configuration.ts
├── database
│   ├── mongo
│   │   ├── config.module.ts
│   │   ├── config.service.ts
│   │   └── configuration.ts
│   ├── mysql
│   │   └── [...]
│   └── postgres
│       └── [...]
├── queue
│   └── [...]
├── session
│   └── [...]
└── storage
    └── [...]
```

Note: Here `[...]` means same as other config folders.

You can see how unique & common the structure is. This makes it easy for other developers that might be new to the project to quickly understand what each file does just by looking at the directory structure.

You can read more about how all the files in config works by looking at my other post: Creating Config Files in NestJS.

Providers 📥

Providers are basically going to be the core modules that initiate a connection between the app & the provider engine (for eg. database).

This I believe is one of the simplest way to structure your providers folder:

```
src/providers
├── cache
│   └── redis
│       └── provider.module.ts
├── database
│   ├── mongo
│   │   └── provider.module.ts
│   ├── mysql
│   │   └── [...]
│   └── postgres
│       └── [...]
├── mail
│   └── smtp
│       └── [...]
└── queue
    └── redis
        └── [...]
```

Note: Here `[...]` means same as other config folders.

Your `provider.module.ts` for each file would look something like this:

```
import { DatabaseType } from 'typeorm';
import { Module } from '@nestjs/common';
import { TypeOrmModule, TypeOrmModuleAsyncOptions } from
'@nestjs/typeorm';
import { MysqlConfigModule } from
'../../../config/database/mysql/config.module';
import { MysqlConfigService } from
'../../../config/database/mysql/config.service';

@Module({
  imports: [
    TypeOrmModule.forRootAsync({
      imports: [MysqlConfigModule],
      useFactory: async (mysqlConfigService: MysqlConfigService) => ({
        type: 'mysql' as DatabaseType,
        host: mysqlConfigService.host,
        port: mysqlConfigService.port,
        username: mysqlConfigService.username,
        password: mysqlConfigService.password,
        database: mysqlConfigService.database,
        entities: [
          // ... All MySQL based schemas/entities
        ],
      }),
      inject: [MysqlConfigService],
    } as TypeOrmModuleAsyncOptions),
  ],
})
export class MysqlDatabaseProviderModule {}
```

Notice how we are importing `MysqlConfigModule` & injecting `MysqlConfigService` from the `src/config/database/mysql` folder.

IMO, this is as clean as it can get. You could try creating classes and directly inject a class by using `useClass` instead of `useFactory` but I feel it's just not needed… especially because you need those environment variable values being passed here.

Models 💾

Models will simply be the parent folder that contains all model related data.

```
src/models
├── addresses
│   ├── entities
│   │   └── address.entity.ts
│   ├── interfaces
│   │   └── address.interface.ts
│   ├── serializers
│   │   └── address.serializer.ts
│   ├── addressess.controller.ts
│   ├── addresses.module.ts
│   ├── addresses.repository.ts
│   └── addresses.service.ts
└── users
    ├── entities
    │   └── user.entity.ts
    ├── interfaces
    │   └── user.interface.ts
    ├── serializers
    │   └── user.serializer.ts
    ├── users.controller.ts
    ├── users.module.ts
    ├── users.repository.ts
    └── users.service.ts
```

To know what the contents of each file are, please read my other post: Best Way to Inject Repositories using TypeORM (NestJS).

Database Migrations/Seeders

The `src/database` folder is as simple as it can get:

```
src/database
├── factories
│   ├── addresses
│   │   └── factory.ts
│   └── users
│       └── factory.ts
├── migrations
│   ├── 1590973586541-CreateAddressesTable.ts
│   └── 1592951122241-CreateUsersTable.ts
└── seeders
    ├── addresses
    │   ├── seeder.module.ts
    │   └── seeder.service.ts
    └── users
        ├── seeder.module.ts
        └── seeder.service.ts
```

I've set my TypeORM config to generate migrations into `src/database/migrations` folder, you can do the same by updating the `migrations` key in your `ormconfig.js` file. You can read more about it here.

To learn more about how `src/database/factories` & `src/database/seeders` work, please read my other article: Seeding Databases Using NestJS.

Authentication

This folder is similar to other folders in `src/models` . The only difference you'd find here is the addition of the authentication strategy involved. You could keep the strategy in its own folder as well.

```
src/authentication
├── dto
│   └── login.dto.ts
├── interfaces
│   ├── jwt-payload.interface.ts
│   ├── login.interface.ts
│   └── token.interface.ts
├── serializers
│   └── token.serializer.ts
├── auth.controller.ts
├── auth.module.ts
└── auth.service.ts
```

```
└── jwt.strategy.ts
```

All files here work according to the authentication tutorial provided in NestJS's documentation itself. Nothing special here! 😄

Common Files 👄

As you can see, `src/common` has the most number of directories here. I use this common folder to literally fill it in with any extra classes/files that might commonly be used by other modules in your app.

There is not much to explain here, except the fact that we are using NestJS's core fundamentals (like guards, pipes, decorators) in this folder & some other common constants, interfaces & helpers.

```
src/common
├── constants
│   └── settings.ts
├── decorators
│   ├── metadata
│   │   └── user-types.decorator.ts
│   ├── requests
│   │   └── logged-in-user.decorator.ts
│   └── validations
│       ├── UserExists.ts
│       └── UniqueUserEmail.ts
├── exceptions
│   └── http-exception.filter.ts
├── guards
│   └── user-types.guard.ts
├── helpers
│   ├── exceptions
│   │   └── validation.helper.ts
│   ├── responses
│   │   ├── error.helper.ts
│   │   └── success.helper.ts
│   ├── number.helper.ts
│   ├── array.helper.ts
│   ├── query.helper.ts
│   ├── request.helper.ts
│   └── string.helper.ts
├── interceptors
│   └── http-cache.interceptor.ts
├── interfaces
│   ├── inputs.interface.ts
│   └── search.interface.ts
├── middleware
│   └── models
│       └── user.middleware.ts
├── pipes
│   ├── models
│   │   └── user-entity.pipe.ts
│   ├── search.pipe.ts
│   └── validation.pipe.ts
└── serializers
    ├── responses
    │   ├── error.serializer.ts
    │   └── success.serializer.ts
    ├── validation
    │   └── validation-error.serializer.ts
    └── model.serializer.ts
```

One extra thing you'd see here is `src/decorator/validations` folder. This folder is based on custom validations built using `class-validator` .

You could also add a `utils` folder for custom classes… in this case, let's say a custom logger class? So you could create a file in `src/common/utils/logger` 😐.

Mails ✉️

The mails folder will just consists of content to be used in your mailers.

```
src/mails
```

```
src/mails
└── verification
    └── content.ts
```

This `data.ts` file would simply be a function returning string content for your mail.

Something like:

```
const content = (firstName: string, verificationLink: string): string
=> {
  return `Hello ${firstName}, <br><br> Please verify your <a
href="${verificationLink}">account</a>. Thanks!`
}
```

Jobs 👷

Finally, the jobs folder will contain your queue consumers/providers.

```
src/jobs
├── consumers
│   └── verification-mail
│       └── verification-mail.job.consumer.ts
└── producers
    └── verification-mail
        └── verification-mail.job.producer.ts
```

This is based on the default queueing technique provided in NestJS documentation using the package `nestjs/bull` .

Since this is literally from the docs, I am hoping you can literally relate the file names with the ones provided in the docs & come up with your own solution.

Conclusion

I hope this helped all the other developers new to this framework in one or the other way.

If you'd like any help, feel free to contact me or my team @Crowdlinker.

Don't forget to share your comments down below. I'd love to hear & see how other developers approaching towards this. 👍 🙂

Prateek Kathal · Jan 30, 2020 ★

Adding Options to your Database Seeder (NestJS)

A little while ago, I published an article on “Seeding Databases in NestJS.” Now, I have created this article basically as a small “icing on the cake” to add some magic to the plain old database seeder. …

Nestjs 3 min read

Prateek Kathal · Jan 9, 2020 ★

Creating Config Files in NestJS

Getting values from an environment file isn’t supposed to be as simple as creating a .env file. In NodeJS, dotenv, is one of the most famous packages that adds values into the node’s process.env and returns them as a string.…

Java Script 4 min read

Prateek Kathal · May 3, 2019

Seeding Databases Using NestJS

NestJS is a highly scalable NodeJS framework revolutionizing the building of server-side apps. Its ability to create Application Contexts allows developers to easily build micro-applications and take advantage of…

Java Script 6 min read

Prateek Kathal · Sep 1, 2018

Create Your Developer Success Story

Some developers code for coffee ☕, some for money 🤑 and some for zen ☮️. Problem solving is one of the skills that makes them a great developer. Is that enough though? Definitely not! For clients, business comes first.…

Programming 6 min read

Read more from The Crowdlinker Chronicle