Configuration

The applications used to run in different **environments**. Depending on an environment, various sets of configuration variables should be used. For example, that's very likely that local environment relies on specific database credentials, valid solely for local db instance. In order to solve this issue, we used to take advantage of `.env` files, that hold key-value pairs, where each key represents a particular value since this approach is very convenient.

But when we use a `process` global object, it's difficult to keep our tests clean since tested class may directly use it. Another way is to create an abstraction layer, a `ConfigModule` that exposes a `ConfigService` with loaded configuration variables.

Installation

Certain platforms automatically attach our environment variables to the `process.env` global. However, in the local environment, we have to manually take care of it. In order to parse our environment files, we'll use a **dotenv** package.

```
$ npm i --save dotenv
$ npm i --save-dev @types/dotenv
```

Service

Firstly, let's create a `ConfigService` class.

JS

```
import * as dotenv from 'dotenv';
import * as fs from 'fs';

export class ConfigService {
  private readonly envConfig: { [key: string]: string };

  constructor(filePath: string) {
    this.envConfig = dotenv.parse(fs.readFileSync(filePath))
  }

  get(key: string): string {
    return this.envConfig[key];
  }
}
```

This class takes a single argument, a `filePath`, which is a path to your `.env` file. The `get()` method is provided to enable access to a private `envConfig` object that holds each property defined inside an environment file.

The last step is to create a `ConfigModule`.

JS

```
import { Module } from '@nestjs/common';
import { ConfigService } from './config.service';

@Module({
  providers: [
    {
      provide: ConfigService,
      useValue: new ConfigService(`${process.env.NODE_ENV}.env`),
    },
  ],
  exports: [ConfigService],
})
export class ConfigModule {}
```

The `ConfigModule` registers a `ConfigService` and exports it as well. Additionally, we passed a path to the `.env` file. This path will be different depending on actual execution environment. Now you can simply inject `ConfigService` anywhere, and pull out a particular value based on a passed key. Sample `.env` file could look like below:

```
DATABASE_USER = test;
DATABASE_PASSWORD = test;
```

Using the ConfigService

To access **environment variables** from our `ConfigService` we need to inject it. Therefore we firstly need to import the module.

app.module.ts JS

```
@Module({
  imports: [ConfigModule],
  ...
})
```

Afterward, you can inject it using an injection token. By default, the token is equal to the class name (in our example `ConfigService`).

app.service.ts JS

```
@Injectable()
export class AppService {
  private isAuthEnabled: boolean;
  constructor(config: ConfigService) {
    // Please take note that this check is case sensitive!
    this.isAuthEnabled = config.get('IS_AUTH_ENABLED') === 'true' ? true : false;
  }
}
```

> **HINT**
> Instead of importing `ConfigModule` in all your modules, you can also declare `ConfigModule` as a global module.

Advanced configuration

We just implemented a basic `ConfigService`. However, this approach has a couple of disadvantages, which we'll address now:

- missing names & types for the environment variables (no IntelliSense)
- a lack of **validation** of the provided `.env` file
- the env file provides booleans as string (`'true'`), and thus have to cast them to a `boolean` every time

Validation

We'll start with the validation of the provided environment variables. You can throw an error if required environment variables haven't been provided or if they don't meet your predefined requirements. For this purpose, we are going to use the npm package **Joi**. With Joi, you define an object schema and validate JavaScript objects against it.

Install Joi and it's types (for **TypeScript** users):

```
$ npm install --save joi
$ npm install --save-dev @types/joi
```

Once the packages are installed, we can move to our `ConfigService`.

config.service.ts JS

```
import * as dotenv from 'dotenv';
import * as Joi from 'joi';
import * as fs from 'fs';

export interface EnvConfig {
  [key: string]: string;
}

export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor(filePath: string) {
    const config = dotenv.parse(fs.readFileSync(filePath));
    this.envConfig = this.validateInput(config);
  }

  /**
   * Ensures all needed variables are set, and returns the validated JavaScript object
   * including the applied default values.
   */
  private validateInput(envConfig: EnvConfig): EnvConfig {
    const envVarsSchema: Joi.ObjectSchema = Joi.object({
      NODE_ENV: Joi.string()
        .valid(['development', 'production', 'test', 'provision'])
        .default('development'),
      PORT: Joi.number().default(3000),
      API_AUTH_ENABLED: Joi.boolean().required(),
    });

    const { error, value: validatedEnvConfig } = Joi.validate(
      envConfig,
      envVarsSchema,
    );
    if (error) {
      throw new Error(`Config validation error: ${error.message}`);
    }
    return validatedEnvConfig;
  }
}
```

Since we set default values for `NODE_ENV` and `PORT` the validation will not fail if we don't provide these variables in the environment file. Nevertheless, we need to explicitly provide `API_AUTH_ENABLED`. The validation will also throw an error if we have variables in our .env file which aren't part of the schema. Additionally, Joi tries to convert the env strings into the right type.

Class properties

For each config property, we have to add a getter function.

config.service.ts JS

```
get isApiAuthEnabled(): boolean {
  return Boolean(this.envConfig.API_AUTH_ENABLED);
}
```

Usage example

Now we can directly access the class properties.

app.service.ts JS

```
@Injectable()
export class AppService {
  constructor(config: ConfigService) {
    if (config.isApiAuthEnabled) {
      // Authorization is enabled
    }
  }
}
```

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Sponsors / Partners

blueanchor.io

NOVOLOGIC Inc.

Designed by Jakub Staroń, hosted by Netlify