## Configuration

Applications often run in different **environments**. Depending on the environment, different configuration settings should be used. For example, usually the local environment relies on specific database credentials, valid only for the local DB instance. The production environment would use a separate set of DB credentials. Since configuration variables change, best practice is to store configuration variables in the environment.

Externally defined environment variables are visible inside Node.js through the `process.env` global. We could try to solve the problem of multiple environments by setting the environment variables separately in each environment. This can quickly get unwieldy, especially in the development and testing environments where these values need to be easily mocked and/or changed.

In Node.js applications, it's common to use `.env` files, holding key-value pairs where each key represents a particular value, to represent each environment. Running an app in different environments is then just a matter of swapping in the correct `.env` file.

A good approach for using this technique in Nest is to create a `ConfigModule` that exposes a `ConfigService` which loads the appropriate `.env` file. While you may choose to write such a module yourself, for convenience Nest provides the `@nestjs/config` package out-of-the box. We'll cover this package in the current chapter.

### Installation

To begin using it, we first install the required dependency.

```
$ npm i --save @nestjs/config
```

> info **Hint** The `@nestjs/config` package internally uses dotenv.

> warning **Note** `@nestjs/config` requires TypeScript 4.1 or later.

### Getting started

Once the installation process is complete, we can import the `ConfigModule`. Typically, we'll import it into the root `AppModule` and control its behavior using the `.forRoot()` static method. During this step, environment variable key/value pairs are parsed and resolved. Later, we'll see several options for accessing the `ConfigService` class of the `ConfigModule` in our other feature modules.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';

@Module({
  imports: [ConfigModule.forRoot()],
})
export class AppModule {}
```

The above code will load and parse a `.env` file from the default location (the project root directory), merge key/value pairs from the `.env` file with environment variables assigned to `process.env`, and store the result in a private structure that you can access through the `ConfigService`. The `forRoot()` method registers the `ConfigService` provider, which provides a `get()` method for reading these parsed/merged configuration variables. Since `@nestjs/config` relies on dotenv, it uses that package's rules for resolving conflicts in environment variable names. When a key exists both in the runtime environment as an environment variable (e.g., via OS shell exports like `export DATABASE_USER=test`) and in a `.env` file, the runtime environment variable takes precedence.

A sample `.env` file looks something like this:

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

#### Custom env file path

By default, the package looks for a `.env` file in the root directory of the application. To specify another path for the `.env` file, set the `envFilePath` property of an (optional) options object you pass to `forRoot()`, as follows:

```
ConfigModule.forRoot({
  envFilePath: '.development.env',
});
```

You can also specify multiple paths for `.env` files like this:

```
ConfigModule.forRoot({
  envFilePath: ['.env.development.local', '.env.development'],
});
```

If a variable is found in multiple files, the first one takes precedence.

#### Disable env variables loading

If you don't want to load the `.env` file, but instead would like to simply access environment variables from the runtime environment (as with OS shell exports like `export DATABASE_USER=test`), set the options object's `ignoreEnvFile` property to `true`, as follows:

```
ConfigModule.forRoot({
  ignoreEnvFile: true,
});
```

### Use module globally

When you want to use `ConfigModule` in other modules, you'll need to import it (as is standard with any Nest module). Alternatively, declare it as a global module by setting the options object's `isGlobal` property to `true`, as shown below. In that case, you will not need to import `ConfigModule` in other modules once it's been loaded in the root module (e.g., `AppModule`).

```
ConfigModule.forRoot({
  isGlobal: true,
});
```

### Custom configuration files

For more complex projects, you may utilize custom configuration files to return nested configuration objects. This allows you to group related configuration settings by function (e.g., database-related settings), and to store related settings in individual files to help manage them independently.

A custom configuration file exports a factory function that returns a configuration object. The configuration object can be any arbitrarily nested plain JavaScript object. The `process.env` object will contain the fully resolved environment variable key/value pairs (with `.env` file and externally defined variables resolved and merged as described above). Since you control the returned configuration object, you can add any required logic to cast values to an appropriate type, set default values, etc. For example:

```
@@filename(config/configuration)
export default () => ({
  port: parseInt(process.env.PORT, 10) || 3000,
  database: {
    host: process.env.DATABASE_HOST,
    port: parseInt(process.env.DATABASE_PORT, 10) || 5432
  }
});
```

We load this file using the `load` property of the options object we pass to the `ConfigModule.forRoot()` method:

```
import configuration from './config/configuration';

@Module({
  imports: [
    ConfigModule.forRoot({
      load: [configuration],
    }),
  ],
})
export class AppModule {}
```

> info **Notice** The value assigned to the `load` property is an array, allowing you to load multiple configuration files (e.g. `load: [databaseConfig, authConfig]`)

With custom configuration files, we can also manage custom files such as YAML files. Here is an example of a configuration using YAML format:

```
http:
  host: 'localhost'
  port: 8080

db:
  postgres:
    url: 'localhost'
    port: 5432
    database: 'yaml-db'

  sqlite:
    database: 'sqlite.db'
```

To read and parse YAML files, we can leverage the `js-yaml` package.

```
$ npm i js-yaml
$ npm i -D @types/js-yaml
```

Once the package is installed, we use `yaml#load` function to load YAML file we just created above.

```
@@filename(config/configuration)
import { readFileSync } from 'fs';
import * as yaml from 'js-yaml';
import { join } from 'path';

const YAML_CONFIG_FILENAME = 'config.yaml';

export default () => {
  return yaml.load(
    readFileSync(join(__dirname, YAML_CONFIG_FILENAME), 'utf8'),
  ) as Record<string, any>;
};
```

> warning **Note** Nest CLI does not automatically move your "assets" (non-TS files) to the `dist` folder during the build process. To make sure that your YAML files are copied, you have to specify this in the `compilerOptions#assets` object in the `nest-cli.json` file. As an example, if the `config` folder is at the same level as the `src` folder, add `compilerOptions#assets` with the value `"assets": [{{ '{' }}"include": "../config/*.yaml", "outDir": "./dist/config"{{ '}' }}]`. Read more here.

### Using the `ConfigService`

To access configuration values from our `ConfigService`, we first need to inject `ConfigService`. As with any provider, we need to import its containing module - the `ConfigModule` - into the module that will use it (unless you set the `isGlobal` property in the options object passed to the `ConfigModule.forRoot()` method to `true`). Import it into a feature module as shown below.

```
@@filename(feature.module)
@Module({
  imports: [ConfigModule],
  // ...
})
```

Then we can inject it using standard constructor injection:

```
constructor(private configService: ConfigService) {}
```

> info **Hint** The `ConfigService` is imported from the `@nestjs/config` package.

And use it in our class:

```
// get an environment variable
const dbUser = this.configService.get<string>('DATABASE_USER');

// get a custom configuration value
const dbHost = this.configService.get<string>('database.host');
```

As shown above, use the `configService.get()` method to get a simple environment variable by passing the variable name. You can do TypeScript type hinting by passing the type, as shown above (e.g., `get<string>(...)`). The `get()` method can also traverse a nested custom configuration object (created via a Custom configuration file), as shown in the second example above.

You can also get the whole nested custom configuration object using an interface as the type hint:

```
interface DatabaseConfig {
  host: string;
  port: number;
}

const dbConfig = this.configService.get<DatabaseConfig>('database');

// you can now use `dbConfig.port` and `dbConfig.host`
const port = dbConfig.port;
```

The `get()` method also takes an optional second argument defining a default value, which will be returned when the key doesn't exist, as shown below:

```
// use "localhost" when "database.host" is not defined
const dbHost = this.configService.get<string>('database.host',
'localhost');
```

`ConfigService` has two optional generics (type arguments). The first one is to help prevent accessing a config property that does not exist. Use it as shown below:

```
interface EnvironmentVariables {
  PORT: number;
  TIMEOUT: string;
}

// somewhere in the code
constructor(private configService: ConfigService<EnvironmentVariables>) {
  const port = this.configService.get('PORT', { infer: true });

  // TypeScript Error: this is invalid as the URL property is not defined
in EnvironmentVariables
  const url = this.configService.get('URL', { infer: true });
}
```

With the `infer` property set to `true`, the `ConfigService#get` method will automatically infer the property type based on the interface, so for example, `typeof port === "number"` (if you're not using `strictNullChecks` flag from TypeScript) since `PORT` has a `number` type in the `EnvironmentVariables` interface.

Also, with the `infer` feature, you can infer the type of a nested custom configuration object's property, even when using dot notation, as follows:

```
constructor(private configService: ConfigService<{ database: { host:
string } }>) {
  const dbHost = this.configService.get('database.host', { infer: true
})!;
  // typeof dbHost === "string"                                    |
  //
+--> non-null assertion operator
}
```

The second generic relies on the first one, acting as a type assertion to get rid of all `undefined` types that `ConfigService`'s methods can return when `strictNullChecks` is on. For instance:

```
// ...
constructor(private configService: ConfigService<{ PORT: number }, true>)
{
  //                                                          ^^^^
  const port = this.configService.get('PORT', { infer: true });
  //    ^^^ The type of port will be 'number' thus you don't need TS type
assertions anymore
}
```

#### Configuration namespaces

The `ConfigModule` allows you to define and load multiple custom configuration files, as shown in Custom configuration files above. You can manage complex configuration object hierarchies with nested configuration objects as shown in that section. Alternatively, you can return a "namespaced" configuration object with the `registerAs()` function as follows:

```
@@filename(config/database.config)
export default registerAs('database', () => ({
  host: process.env.DATABASE_HOST,
  port: process.env.DATABASE_PORT || 5432
}));
```

As with custom configuration files, inside your `registerAs()` factory function, the `process.env` object will contain the fully resolved environment variable key/value pairs (with `.env` file and externally defined variables resolved and merged as described above).

> info **Hint** The `registerAs` function is exported from the `@nestjs/config` package.

Load a namespaced configuration with the `load` property of the `forRoot()` method's options object, in the same way you load a custom configuration file:

```
import databaseConfig from './config/database.config';

@Module({
  imports: [
    ConfigModule.forRoot({
      load: [databaseConfig],
    }),
  ],
})
export class AppModule {}
```

Now, to get the `host` value from the `database` namespace, use dot notation. Use `'database'` as the prefix to the property name, corresponding to the name of the namespace (passed as the first argument to the `registerAs()` function):

```
const dbHost = this.configService.get<string>('database.host');
```

A reasonable alternative is to inject the `database` namespace directly. This allows us to benefit from strong typing:

```
constructor(
  @Inject(databaseConfig.KEY)
  private dbConfig: ConfigType<typeof databaseConfig>,
) {}
```

> info **Hint** The `ConfigType` is exported from the `@nestjs/config` package.

#### Cache environment variables

As accessing `process.env` can be slow, you can set the `cache` property of the options object passed to `ConfigModule.forRoot()` to increase the performance of `ConfigService#get` method when it comes to variables stored in `process.env`.

```
ConfigModule.forRoot({
  cache: true,
});
```

#### Partial registration

Thus far, we've processed configuration files in our root module (e.g., `AppModule`), with the `forRoot()` method. Perhaps you have a more complex project structure, with feature-specific configuration files located in multiple different directories. Rather than load all these files in the root module, the `@nestjs/config` package provides a feature called **partial registration**, which references only the configuration files associated with each feature module. Use the `forFeature()` static method within a feature module to perform this partial registration, as follows:

```
import databaseConfig from './config/database.config';

@Module({
  imports: [ConfigModule.forFeature(databaseConfig)],
})
export class DatabaseModule {}
```

> info **Warning** In some circumstances, you may need to access properties loaded via partial registration using the `onModuleInit()` hook, rather than in a constructor. This is because the `forFeature()` method is run during module initialization, and the order of module initialization is indeterminate. If you access values loaded this way by another module, in a constructor, the module

> that the configuration depends upon may not yet have initialized. The `onModuleInit()` method runs only after all modules it depends upon have been initialized, so this technique is safe.

**Schema validation**

It is standard practice to throw an exception during application startup if required environment variables haven't been provided or if they don't meet certain validation rules. The `@nestjs/config` package enables two different ways to do this:

- Joi built-in validator. With Joi, you define an object schema and validate JavaScript objects against it.
- A custom `validate()` function which takes environment variables as an input.

To use Joi, we must install Joi package:

```
$ npm install --save joi
```

Now we can define a Joi validation schema and pass it via the `validationSchema` property of the `forRoot()` method's options object, as shown below:

```
@@filename(app.module)
import * as Joi from 'joi';

@Module({
  imports: [
    ConfigModule.forRoot({
      validationSchema: Joi.object({
        NODE_ENV: Joi.string()
          .valid('development', 'production', 'test', 'provision')
          .default('development'),
        PORT: Joi.number().default(3000),
      }),
    }),
  ],
})
export class AppModule {}
```

By default, all schema keys are considered optional. Here, we set default values for `NODE_ENV` and `PORT` which will be used if we don't provide these variables in the environment (`.env` file or process environment). Alternatively, we can use the `required()` validation method to require that a value must be defined in the environment (`.env` file or process environment). In this case, the validation step will throw an exception if we don't provide the variable in the environment. See Joi validation methods for more on how to construct validation schemas.

By default, unknown environment variables (environment variables whose keys are not present in the schema) are allowed and do not trigger a validation exception. By default, all validation errors are reported. You can alter these behaviors by passing an options object via the `validationOptions` key of the `forRoot()` options object. This options object can contain any of the standard validation options

properties provided by Joi validation options. For example, to reverse the two settings above, pass options like this:

```
@@filename(app.module)
import * as Joi from 'joi';

@Module({
  imports: [
    ConfigModule.forRoot({
      validationSchema: Joi.object({
        NODE_ENV: Joi.string()
          .valid('development', 'production', 'test', 'provision')
          .default('development'),
        PORT: Joi.number().default(3000),
      }),
      validationOptions: {
        allowUnknown: false,
        abortEarly: true,
      },
    }),
  ],
})
export class AppModule {}
```

The `@nestjs/config` package uses default settings of:

- `allowUnknown`: controls whether or not to allow unknown keys in the environment variables. Default is `true`
- `abortEarly`: if true, stops validation on the first error; if false, returns all errors. Defaults to `false`.

Note that once you decide to pass a `validationOptions` object, any settings you do not explicitly pass will default to `Joi` standard defaults (not the `@nestjs/config` defaults). For example, if you leave `allowUnknowns` unspecified in your custom `validationOptions` object, it will have the `Joi` default value of `false`. Hence, it is probably safest to specify **both** of these settings in your custom object.

#### Custom validate function

Alternatively, you can specify a **synchronous** `validate` function that takes an object containing the environment variables (from env file and process) and returns an object containing validated environment variables so that you can convert/mutate them if needed. If the function throws an error, it will prevent the application from bootstrapping.

In this example, we'll proceed with the `class-transformer` and `class-validator` packages. First, we have to define:

- a class with validation constraints,
- a validate function that makes use of the `plainToInstance` and `validateSync` functions.

```
@@filename(env.validation)
import { plainToInstance } from 'class-transformer';
import { IsEnum, IsNumber, validateSync } from 'class-validator';

enum Environment {
  Development = "development",
  Production = "production",
  Test = "test",
  Provision = "provision",
}

class EnvironmentVariables {
  @IsEnum(Environment)
  NODE_ENV: Environment;

  @IsNumber()
  PORT: number;
}

export function validate(config: Record<string, unknown>) {
  const validatedConfig = plainToInstance(
    EnvironmentVariables,
    config,
    { enableImplicitConversion: true },
  );
  const errors = validateSync(validatedConfig, { skipMissingProperties:
false });

  if (errors.length > 0) {
    throw new Error(errors.toString());
  }
  return validatedConfig;
}
```

With this in place, use the `validate` function as a configuration option of the `ConfigModule`, as follows:

```
@@filename(app.module)
import { validate } from './env.validation';

@Module({
  imports: [
    ConfigModule.forRoot({
      validate,
    }),
  ],
})
export class AppModule {}
```

### Custom getter functions

`ConfigService` defines a generic `get()` method to retrieve a configuration value by key. We may also add `getter` functions to enable a little more natural coding style:

```
@@filename()
@Injectable()
export class ApiConfigService {
  constructor(private configService: ConfigService) {}

  get isAuthEnabled(): boolean {
    return this.configService.get('AUTH_ENABLED') === 'true';
  }
}
@@switch
@Dependencies(ConfigService)
@Injectable()
export class ApiConfigService {
  constructor(configService) {
    this.configService = configService;
  }

  get isAuthEnabled() {
    return this.configService.get('AUTH_ENABLED') === 'true';
  }
}
```

Now we can use the getter function as follows:

```
@@filename(app.service)
@Injectable()
export class AppService {
  constructor(apiConfigService: ApiConfigService) {
    if (apiConfigService.isAuthEnabled) {
      // Authentication is enabled
    }
  }
}
@@switch
@Dependencies(ApiConfigService)
@Injectable()
export class AppService {
  constructor(apiConfigService) {
    if (apiConfigService.isAuthEnabled) {
      // Authentication is enabled
    }
  }
}
```

**Environment variables loaded hook**

If a module configuration depends on the environment variables, and these variables are loaded from the `.env` file, you can use the `ConfigModule.envVariablesLoaded` hook to ensure that the file was loaded before interacting with the `process.env` object, see the following example:

```
export async function getStorageModule() {
  await ConfigModule.envVariablesLoaded;
  return process.env.STORAGE === 'S3' ? S3StorageModule :
DefaultStorageModule;
}
```

This construction guarantees that after the `ConfigModule.envVariablesLoaded` Promise resolves, all configuration variables are loaded up.

#### Expandable variables

The `@nestjs/config` package supports environment variable expansion. With this technique, you can create nested environment variables, where one variable is referred to within the definition of another. For example:

```
APP_URL=mywebsite.com
SUPPORT_EMAIL=support@${APP_URL}
```

With this construction, the variable `SUPPORT_EMAIL` resolves to `'support@mywebsite.com'`. Note the use of the `${{ '{' }}...{{ '}' }}` syntax to trigger resolving the value of the variable `APP_URL` inside the definition of `SUPPORT_EMAIL`.

> info **Hint** For this feature, `@nestjs/config` package internally uses dotenv-expand.

Enable environment variable expansion using the `expandVariables` property in the options object passed to the `forRoot()` method of the `ConfigModule`, as shown below:

```
@@filename(app.module)
@Module({
  imports: [
    ConfigModule.forRoot({
      // ...
      expandVariables: true,
    }),
  ],
})
export class AppModule {}
```

#### Using in the `main.ts`

While our config is a stored in a service, it can still be used in the `main.ts` file. This way, you can use it to store variables such as the application port or the CORS host.

To access it, you must use the `app.get()` method, followed by the service reference:

```
const configService = app.get(ConfigService);
```

You can then use it as usual, by calling the `get` method with the configuration key:

```
const port = configService.get('PORT');
```