Dynamic modules

The Modules chapter covers the basics of Nest modules, and includes a brief introduction to dynamic modules. This chapter expands on the subject of dynamic modules. Upon completion, you should have a good grasp of what they are and how and when to use them.

Introduction

Most application code examples in the **Overview** section of the documentation make use of regular, or static, modules. Modules define groups of components like providers and controllers that fit together as a modular part of an overall application. They provide an execution context, or scope, for these components. For example, providers defined in a module are visible to other members of the module without the need to export them. When a provider needs to be visible outside of a module, it is first exported from its host module, and then imported into its consuming module.

Let's walk through a familiar example.

First, we'll define a `UsersModule` to provide and export a `UsersService`. `UsersModule` is the **host** module for `UsersService`.

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';

@Module({
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}
```

Next, we'll define an `AuthModule`, which imports `UsersModule`, making `UsersModule`'s exported providers available inside `AuthModule`:

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [UsersModule],
  providers: [AuthService],
  exports: [AuthService],
})
export class AuthModule {}
```

These constructs allow us to inject `UsersService` in, for example, the `AuthService` that is hosted in `AuthModule`:

```
import { Injectable } from '@nestjs/common';
import { UsersService } from '../users/users.service';

@Injectable()
export class AuthService {
  constructor(private usersService: UsersService) {}
  /*
    Implementation that makes use of this.usersService
  */
}
```

We'll refer to this as **static** module binding. All the information Nest needs to wire together the modules has already been declared in the host and consuming modules. Let's unpack what's happening during this process. Nest makes `UsersService` available inside `AuthModule` by:

1. Instantiating `UsersModule`, including transitively importing other modules that `UsersModule` itself consumes, and transitively resolving any dependencies (see Custom providers).
2. Instantiating `AuthModule`, and making `UsersModule`'s exported providers available to components in `AuthModule` (just as if they had been declared in `AuthModule`).
3. Injecting an instance of `UsersService` in `AuthService`.

Dynamic module use case

With static module binding, there's no opportunity for the consuming module to **influence** how providers from the host module are configured. Why does this matter? Consider the case where we have a general purpose module that needs to behave differently in different use cases. This is analogous to the concept of a "plugin" in many systems, where a generic facility requires some configuration before it can be used by a consumer.

A good example with Nest is a **configuration module**. Many applications find it useful to externalize configuration details by using a configuration module. This makes it easy to dynamically change the application settings in different deployments: e.g., a development database for developers, a staging database for the staging/testing environment, etc. By delegating the management of configuration parameters to a configuration module, the application source code remains independent of configuration parameters.

The challenge is that the configuration module itself, since it's generic (similar to a "plugin"), needs to be customized by its consuming module. This is where *dynamic modules* come into play. Using dynamic module features, we can make our configuration module **dynamic** so that the consuming module can use an API to control how the configuration module is customized at the time it is imported.

In other words, dynamic modules provide an API for importing one module into another, and customizing the properties and behavior of that module when it is imported, as opposed to using the static bindings we've seen so far.

Config module example

We'll be using the basic version of the example code from the configuration chapter for this section. The completed version as of the end of this chapter is available as a working example here.

Our requirement is to make `ConfigModule` accept an `options` object to customize it. Here's the feature we want to support. The basic sample hard-codes the location of the `.env` file to be in the project root folder. Let's suppose we want to make that configurable, such that you can manage your `.env` files in any folder of your choosing. For example, imagine you want to store your various `.env` files in a folder under the project root called `config` (i.e., a sibling folder to `src`). You'd like to be able to choose different folders when using the `ConfigModule` in different projects.

Dynamic modules give us the ability to pass parameters into the module being imported so we can change its behavior. Let's see how this works. It's helpful if we start from the end-goal of how this might look from the consuming module's perspective, and then work backwards. First, let's quickly review the example of *statically* importing the `ConfigModule` (i.e., an approach which has no ability to influence the behavior of the imported module). Pay close attention to the `imports` array in the `@Module()` decorator:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Let's consider what a *dynamic module* import, where we're passing in a configuration object, might look like. Compare the difference in the `imports` array between these two examples:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule.register({ folder: './config' })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Let's see what's happening in the dynamic example above. What are the moving parts?

1. `ConfigModule` is a normal class, so we can infer that it must have a **static method** called `register()`. We know it's static because we're calling it on the `ConfigModule` class, not on an

instance of the class. Note: this method, which we will create soon, can have any arbitrary name, but by convention we should call it either `forRoot()` or `register()`.
2. The `register()` method is defined by us, so we can accept any input arguments we like. In this case, we're going to accept a simple `options` object with suitable properties, which is the typical case.
3. We can infer that the `register()` method must return something like a `module` since its return value appears in the familiar `imports` list, which we've seen so far includes a list of modules.

In fact, what our `register()` method will return is a `DynamicModule`. A dynamic module is nothing more than a module created at run-time, with the same exact properties as a static module, plus one additional property called `module`. Let's quickly review a sample static module declaration, paying close attention to the module options passed in to the decorator:

```
@Module({
  imports: [DogsModule],
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
```

Dynamic modules must return an object with the exact same interface, plus one additional property called `module`. The `module` property serves as the name of the module, and should be the same as the class name of the module, as shown in the example below.

> info **Hint** For a dynamic module, all properties of the module options object are optional **except** `module`.

What about the static `register()` method? We can now see that its job is to return an object that has the `DynamicModule` interface. When we call it, we are effectively providing a module to the `imports` list, similar to the way we would do so in the static case by listing a module class name. In other words, the dynamic module API simply returns a module, but rather than fix the properties in the `@Module` decorator, we specify them programmatically.

There are still a couple of details to cover to help make the picture complete:

1. We can now state that the `@Module()` decorator's `imports` property can take not only a module class name (e.g., `imports: [UsersModule]`), but also a function **returning** a dynamic module (e.g., `imports: [ConfigModule.register(...)]`).
2. A dynamic module can itself import other modules. We won't do so in this example, but if the dynamic module depends on providers from other modules, you would import them using the optional `imports` property. Again, this is exactly analogous to the way you'd declare metadata for a static module using the `@Module()` decorator.

Armed with this understanding, we can now look at what our dynamic `ConfigModule` declaration must look like. Let's take a crack at it.

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

@Module({})
export class ConfigModule {
  static register(): DynamicModule {
    return {
      module: ConfigModule,
      providers: [ConfigService],
      exports: [ConfigService],
    };
  }
}
```

It should now be clear how the pieces tie together. Calling `ConfigModule.register(...)` returns a `DynamicModule` object with properties which are essentially the same as those that, until now, we've provided as metadata via the `@Module()` decorator.

> info **Hint** Import `DynamicModule` from `@nestjs/common`.

Our dynamic module isn't very interesting yet, however, as we haven't introduced any capability to **configure** it as we said we would like to do. Let's address that next.

Module configuration

The obvious solution for customizing the behavior of the `ConfigModule` is to pass it an `options` object in the static `register()` method, as we guessed above. Let's look once again at our consuming module's `imports` property:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule.register({ folder: './config' })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

That nicely handles passing an `options` object to our dynamic module. How do we then use that `options` object in the `ConfigModule`? Let's consider that for a minute. We know that our `ConfigModule` is basically a host for providing and exporting an injectable service - the `ConfigService` - for use by other providers. It's actually our `ConfigService` that needs to read the `options` object to customize its behavior. Let's assume for the moment that we know how to somehow get the `options` from the `register()` method into the `ConfigService`. With that assumption, we can make a few changes to the service to customize its behavior based on the properties from the `options` object. (**Note**: for the time

being, since we *haven't* actually determined how to pass it in, we'll just hard-code `options`. We'll fix this in a minute).

```
import { Injectable } from '@nestjs/common';
import * as dotenv from 'dotenv';
import * as fs from 'fs';
import * as path from 'path';
import { EnvConfig } from './interfaces';

@Injectable()
export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor() {
    const options = { folder: './config' };

    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../../', options.folder,
filePath);
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

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

Now our `ConfigService` knows how to find the `.env` file in the folder we've specified in `options`.

Our remaining task is to somehow inject the `options` object from the `register()` step into our `ConfigService`. And of course, we'll use *dependency injection* to do it. This is a key point, so make sure you understand it. Our `ConfigModule` is providing `ConfigService`. `ConfigService` in turn depends on the `options` object that is only supplied at run-time. So, at run-time, we'll need to first bind the `options` object to the Nest IoC container, and then have Nest inject it into our `ConfigService`. Remember from the **Custom providers** chapter that providers can include any value not just services, so we're fine using dependency injection to handle a simple `options` object.

Let's tackle binding the options object to the IoC container first. We do this in our static `register()` method. Remember that we are dynamically constructing a module, and one of the properties of a module is its list of providers. So what we need to do is define our options object as a provider. This will make it injectable into the `ConfigService`, which we'll take advantage of in the next step. In the code below, pay attention to the `providers` array:

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

@Module({})
export class ConfigModule {
  static register(options: Record<string, any>): DynamicModule {
```

```
      return {
        module: ConfigModule,
        providers: [
          {
            provide: 'CONFIG_OPTIONS',
            useValue: options,
          },
          ConfigService,
        ],
        exports: [ConfigService],
      };
    }
  }
```

Now we can complete the process by injecting the `'CONFIG_OPTIONS'` provider into the `ConfigService`. Recall that when we define a provider using a non-class token we need to use the `@Inject()` decorator as described here.

```
import * as dotenv from 'dotenv';
import * as fs from 'fs';
import * as path from 'path';
import { Injectable, Inject } from '@nestjs/common';
import { EnvConfig } from './interfaces';

@Injectable()
export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor(@Inject('CONFIG_OPTIONS') private options: Record<string,
any>) {
    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../../', options.folder,
filePath);
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

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

One final note: for simplicity we used a string-based injection token (`'CONFIG_OPTIONS'`) above, but best practice is to define it as a constant (or `Symbol`) in a separate file, and import that file. For example:

```
export const CONFIG_OPTIONS = 'CONFIG_OPTIONS';
```

Example

A full example of the code in this chapter can be found here.

Community guidelines

You may have seen the use for methods like `forRoot`, `register`, and `forFeature` around some of the `@nestjs/` packages and may be wondering what the difference for all of these methods are. There is no hard rule about this, but the `@nestjs/` packages try to follow these guidelines:

When creating a module with:

- `register`, you are expecting to configure a dynamic module with a specific configuration for use only by the calling module. For example, with Nest's `@nestjs/axios`: `HttpModule.register({{ '{' }} baseUrl: 'someUrl' {{ '}' }})`. If, in another module you use `HttpModule.register({{ '{' }} baseUrl: 'somewhere else' {{ '}' }})`, it will have the different configuration. You can do this for as many modules as you want.
- `forRoot`, you are expecting to configure a dynamic module once and reuse that configuration in multiple places (though possibly unknowingly as it's abstracted away). This is why you have one `GraphQLModule.forRoot()`, one `TypeOrmModule.forRoot()`, etc.
- `forFeature`, you are expecting to use the configuration of a dynamic module's `forRoot` but need to modify some configuration specific to the calling module's needs (i.e. which repository this module should have access to, or the context that a logger should use.)

All of these, usually, have their `async` counterparts as well, `registerAsync`, `forRootAsync`, and `forFeatureAsync`, that mean the same thing, but use Nest's Dependency Injection for the configuration as well.

Configurable module builder

As manually creating highly configurable, dynamic modules that expose `async` methods (`registerAsync`, `forRootAsync`, etc.) is quite complicated, especially for newcomers, Nest exposes the `ConfigurableModuleBuilder` class that facilitates this process and lets you construct a module "blueprint" in just a few lines of code.

For example, let's take the example we used above (`ConfigModule`) and convert it to use the `ConfigurableModuleBuilder`. Before we start, let's make sure we create a dedicated interface that represents what options our `ConfigModule` takes in.

```
export interface ConfigModuleOptions {
  folder: string;
}
```

With this in place, create a new dedicated file (alongside the existing `config.module.ts` file) and name it `config.module-definition.ts`. In this file, let's utilize the `ConfigurableModuleBuilder` to construct `ConfigModule` definition.

```

```

```
@@filename(config.module-definition)
import { ConfigurableModuleBuilder } from '@nestjs/common';
import { ConfigModuleOptions } from './interfaces/config-module-
options.interface';

export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>().build();
@@switch
import { ConfigurableModuleBuilder } from '@nestjs/common';

export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder().build();
```

Now let's open up the `config.module.ts` file and modify its implementation to leverage the auto-generated `ConfigurableModuleClass`:

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

@Module({
  providers: [ConfigService],
  exports: [ConfigService],
})
export class ConfigModule extends ConfigurableModuleClass {}
```

Extending the `ConfigurableModuleClass` means that `ConfigModule` provides now not only the `register` method (as previously with the custom implementation), but also the `registerAsync` method which allows consumers asynchronously configure that module, for example, by supplying async factories:

```
@Module({
  imports: [
    ConfigModule.register({ folder: './config' }),
    // or alternatively:
    // ConfigModule.registerAsync({
    //   useFactory: () => {
    //     return {
    //       folder: './config',
    //     }
    //   },
    //   inject: [...any extra dependencies...]
    // }),
  ],
})
export class AppModule {}
```

Lastly, let's update the `ConfigService` class to inject the generated module options' provider instead of the `'CONFIG_OPTIONS'` that we used so far.

```
@Injectable()
export class ConfigService {
  constructor(@Inject(MODULE_OPTIONS_TOKEN) private options:
ConfigModuleOptions) { ... }
}
```

Custom method key

`ConfigurableModuleClass` by default provides the `register` and its counterpart `registerAsync` methods. To use a different method name, use the `ConfigurableModuleBuilder#setClassMethodName` method, as follows:

```
@@filename(config.module-definition)
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>
().setClassMethodName('forRoot').build();
@@switch
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder().setClassMethodName('forRoot').build();
```

This construction will instruct `ConfigurableModuleBuilder` to generate a class that exposes `forRoot` and `forRootAsync` instead. Example:

```
@Module({
  imports: [
    ConfigModule.forRoot({ folder: './config' }), // <-- note the use of
"forRoot" instead of "register"
    // or alternatively:
    // ConfigModule.forRootAsync({
    //   useFactory: () => {
    //     return {
    //       folder: './config',
    //     }
    //   },
    //   inject: [...any extra dependencies...]
    // }),
  ],
})
export class AppModule {}
```

Custom options factory class

Since the `registerAsync` method (or `forRootAsync` or any other name, depending on the configuration) lets consumer pass a provider definition that resolves to the module configuration, a library consumer could potentially supply a class to be used to construct the configuration object.

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

This class, by default, must provide the `create()` method that returns a module configuration object. However, if your library follows a different naming convention, you can change that behavior and instruct `ConfigurableModuleBuilder` to expect a different method, for example, `createConfigOptions`, using the `ConfigurableModuleBuilder#setFactoryMethodName` method:

```
@@filename(config.module-definition)
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>
().setFactoryMethodName('createConfigOptions').build();
@@switch
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new
ConfigurableModuleBuilder().setFactoryMethodName('createConfigOptions').bu
ild();
```

Now, `ConfigModuleOptionsFactory` class must expose the `createConfigOptions` method (instead of `create`):

```
@Module({
  imports: [
    ConfigModule.registerAsync({
      useClass: ConfigModuleOptionsFactory, // <-- this class must provide
the "createConfigOptions" method
    }),
  ],
})
export class AppModule {}
```

Extra options

There are edge-cases when your module may need to take extra options that determine how it is supposed to behave (a nice example of such an option is the `isGlobal` flag - or just `global`) that at the same time,

shouldn't be included in the `MODULE_OPTIONS_TOKEN` provider (as they are irrelevant to services/providers registered within that module, for example, `ConfigService` does not need to know whether its host module is registered as a global module).

In such cases, the `ConfigurableModuleBuilder#setExtras` method can be used. See the following example:

```
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } = new
ConfigurableModuleBuilder<ConfigModuleOptions>()
  .setExtras(
    {
      isGlobal: true,
    },
    (definition, extras) => ({
      ...definition,
      global: extras.isGlobal,
    }),
  )
  .build();
```

In the example above, the first argument passed into the `setExtras` method is an object containing default values for the "extra" properties. The second argument is a function that takes an auto-generated module definitions (with `provider`, `exports`, etc.) and `extras` object which represents extra properties (either specified by the consumer or defaults). The returned value of this function is a modified module definition. In this specific example, we're taking the `extras.isGlobal` property and assigning it to the `global` property of the module definition (which in turn determines whether a module is global or not, read more here).

Now when consuming this module, the additional `isGlobal` flag can be passed in, as follows:

```
@Module({
  imports: [
    ConfigModule.register({
      isGlobal: true,
      folder: './config',
    }),
  ],
})
export class AppModule {}
```

However, since `isGlobal` is declared as an "extra" property, it won't be available in the `MODULE_OPTIONS_TOKEN` provider:

```
@Injectable()
export class ConfigService {
  constructor(@Inject(MODULE_OPTIONS_TOKEN) private options:
ConfigModuleOptions) {
```

```
    // "options" object will not have the "isGlobal" property
    // ...
  }
}
```

Extending auto-generated methods

The auto-generated static methods (`register`, `registerAsync`, etc.) can be extended if needed, as follows:

```
import { Module } from '@nestjs/common';
import { ConfigService } from './config.service';
import { ConfigurableModuleClass, ASYNC_OPTIONS_TYPE, OPTIONS_TYPE } from
'./config.module-definition';

@Module({
  providers: [ConfigService],
  exports: [ConfigService],
})
export class ConfigModule extends ConfigurableModuleClass {
  static register(options: typeof OPTIONS_TYPE): DynamicModule {
    return {
      // your custom logic here
      ...super.register(options),
    };
  }

  static registerAsync(options: typeof ASYNC_OPTIONS_TYPE): DynamicModule
{
    return {
      // your custom logic here
      ...super.registerAsync(options),
    };
  }
}
```

Note the use of `OPTIONS_TYPE` and `ASYNC_OPTIONS_TYPE` types that must be exported from the module definition file:

```
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN,
OPTIONS_TYPE, ASYNC_OPTIONS_TYPE } = new
ConfigurableModuleBuilder<ConfigModuleOptions>().build();
```