

Modules

A module is a class annotated with a `@Module()` decorator. The `@Module()` decorator provides metadata that **Nest** makes use of to organize the application structure.



Each application has at least one module, a **root module**. The root module is the starting point Nest uses to build the **application graph** - the internal data structure Nest uses to resolve module and provider relationships and dependencies. While very small applications may theoretically have just the root module, this is not the typical case. We want to emphasize that modules are **strongly** recommended as an effective way to organize your components. Thus, for most applications, the resulting architecture will employ multiple modules, each encapsulating a closely related set of **capabilities**.

The `@Module()` decorator takes a single object whose properties describe the module:

<code>providers</code>	the providers that will be instantiated by the Nest injector and that may be shared at least across this module
<code>controllers</code>	the set of controllers defined in this module which have to be instantiated
<code>imports</code>	the list of imported modules that export the providers which are required in this module
<code>exports</code>	the subset of <code>providers</code> that are provided by this module and should be available in other modules which import this module. You can use either the provider itself or just its token (<code>provide</code> value)

The module **encapsulates** providers by default. This means that it's impossible to inject providers that are neither directly part of the current module nor exported from the imported modules. Thus, you may consider the exported providers from a module as the module's public interface, or API.

Feature modules

The `CatsController` and `CatsService` belong to the same application domain. As they are closely related, it makes sense to move them into a feature module. A feature module simply organizes code relevant for a specific feature, keeping code organized and establishing clear boundaries. This helps us manage complexity and develop with **SOLID** principles, especially as the size of the application and/or team grow.

To demonstrate this, we'll create the `CatsModule`.

```
@filename(cats/cats.module)
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
```

```
    providers: [CatsService],
  })
  export class CatsModule {}
```

info **Hint** To create a module using the CLI, simply execute the `$ nest g module cats` command.

Above, we defined the `CatsModule` in the `cats.module.ts` file, and moved everything related to this module into the `cats` directory. The last thing we need to do is import this module into the root module (the `AppModule`, defined in the `app.module.ts` file).

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

@Module({
  imports: [CatsModule],
})
export class AppModule {}
```

Here is how our directory structure looks now:

```
src
cats
dto
create-cat.dto.ts
interfaces
cat.interface.ts
cats.controller.ts
cats.module.ts
cats.service.ts
app.module.ts
main.ts
```

Shared modules

In Nest, modules are **singletons** by default, and thus you can share the same instance of any provider between multiple modules effortlessly.



Every module is automatically a **shared module**. Once created it can be reused by any module. Let's imagine that we want to share an instance of the `CatsService` between several other modules. In order to do that, we first need to **export** the `CatsService` provider by adding it to the module's `exports` array, as shown below:

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

```
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
export class CatsModule {}
```

Now any module that imports the `CatsModule` has access to the `CatsService` and will share the same instance with all other modules that import it as well.

Module re-exporting

As seen above, Modules can export their internal providers. In addition, they can re-export modules that they import. In the example below, the `CommonModule` is both imported into **and** exported from the `CoreModule`, making it available for other modules which import this one.

```
@Module({
  imports: [CommonModule],
  exports: [CommonModule],
})
export class CoreModule {}
```

Dependency injection

A module class can **inject** providers as well (e.g., for configuration purposes):

```
@@filename(cats.module)
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {
  constructor(private catsService: CatsService) {}
}

@@switch
import { Module, Dependencies } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
```

```
    providers: [CatsService],
  })
  @Dependencies(CatsService)
  export class CatsModule {
    constructor(catsService) {
      this.catsService = catsService;
    }
  }
}
```

However, module classes themselves cannot be injected as providers due to [circular dependency](#) .

Global modules

If you have to import the same set of modules everywhere, it can get tedious. Unlike in Nest, [Angular providers](#) are registered in the global scope. Once defined, they're available everywhere. Nest, however, encapsulates providers inside the module scope. You aren't able to use a module's providers elsewhere without first importing the encapsulating module.

When you want to provide a set of providers which should be available everywhere out-of-the-box (e.g., helpers, database connections, etc.), make the module **global** with the `@Global()` decorator.

```
import { Module, Global } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Global()
@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService],
})
export class CatsModule {}
```

The `@Global()` decorator makes the module global-scoped. Global modules should be registered **only once**, generally by the root or core module. In the above example, the `CatsService` provider will be ubiquitous, and modules that wish to inject the service will not need to import the `CatsModule` in their imports array.

info Hint Making everything global is not a good design decision. Global modules are available to reduce the amount of necessary boilerplate. The `imports` array is generally the preferred way to make the module's API available to consumers.

Dynamic modules

The Nest module system includes a powerful feature called **dynamic modules**. This feature enables you to easily create customizable modules that can register and configure providers dynamically. Dynamic modules are covered extensively [here](#). In this chapter, we'll give a brief overview to complete the introduction to modules.

Following is an example of a dynamic module definition for a `DatabaseModule`:

```

@@filename()
import { Module, DynamicModule } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options?): DynamicModule {
    const providers = createDatabaseProviders(options, entities);
    return {
      module: DatabaseModule,
      providers: providers,
      exports: providers,
    };
  }
}

@@switch
import { Module } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options) {
    const providers = createDatabaseProviders(options, entities);
    return {
      module: DatabaseModule,
      providers: providers,
      exports: providers,
    };
  }
}

```

info **Hint** The `forRoot()` method may return a dynamic module either synchronously or asynchronously (i.e., via a `Promise`).

This module defines the `Connection` provider by default (in the `@Module()` decorator metadata), but additionally - depending on the `entities` and `options` objects passed into the `forRoot()` method - exposes a collection of providers, for example, repositories. Note that the properties returned by the dynamic module **extend** (rather than override) the base module metadata defined in the `@Module()` decorator. That's how both the statically declared `Connection` provider **and** the dynamically generated repository providers are exported from the module.

If you want to register a dynamic module in the global scope, set the `global` property to `true`.

```
{
  global: true,
  module: DatabaseModule,
  providers: providers,
  exports: providers,
}
```

warning **Warning** As mentioned above, making everything global **is not a good design decision**.

The `DatabaseModule` can be imported and configured in the following manner:

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { User } from '../users/entities/user.entity';

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

If you want to in turn re-export a dynamic module, you can omit the `forRoot()` method call in the exports array:

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { User } from '../users/entities/user.entity';

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

The [Dynamic modules](#) chapter covers this topic in greater detail, and includes a [working example](#).

info **Hint** Learn how to build highly customizable dynamic modules with the use of `ConfigurableModuleBuilder` here in [this chapter](#).