Injection scopes

For people coming from different programming language backgrounds, it might be unexpected to learn that in Nest, almost everything is shared across incoming requests. We have a connection pool to the database, singleton services with global state, etc. Remember that Node.js doesn't follow the request/response Multi-Threaded Stateless Model in which every request is processed by a separate thread. Hence, using singleton instances is fully **safe** for our applications.

However, there are edge-cases when request-based lifetime may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking, and multi-tenancy. Injection scopes provide a mechanism to obtain the desired provider lifetime behavior.

Provider scope

A provider can have any of the following scopes:

`DEFAULT`	A single instance of the provider is shared across the entire application. The instance lifetime is tied directly to the application lifecycle. Once the application has bootstrapped, all singleton providers have been instantiated. Singleton scope is used by default.
`REQUEST`	A new instance of the provider is created exclusively for each incoming **request**. The instance is garbage-collected after the request has completed processing.
`TRANSIENT`	Transient providers are not shared across consumers. Each consumer that injects a transient provider will receive a new, dedicated instance.

> info **Hint** Using singleton scope is **recommended** for most use cases. Sharing providers across consumers and across requests means that an instance can be cached and its initialization occurs only once, during application startup.

Usage

Specify injection scope by passing the `scope` property to the `@Injectable()` decorator options object:

```
import { Injectable, Scope } from '@nestjs/common';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {}
```

Similarly, for custom providers, set the `scope` property in the long-hand form for a provider registration:

```
{
  provide: 'CACHE_MANAGER',
  useClass: CacheManager,
  scope: Scope.TRANSIENT,
}
```

> info **Hint** Import the `Scope` enum from `@nestjs/common`

Singleton scope is used by default, and need not be declared. If you do want to declare a provider as singleton scoped, use the `Scope.DEFAULT` value for the `scope` property.

> warning **Notice** Websocket Gateways should not use request-scoped providers because they must act as singletons. Each gateway encapsulates a real socket and cannot be instantiated multiple times. The limitation also applies to some other providers, like *Passport strategies* or *Cron controllers*.

Controller scope

Controllers can also have scope, which applies to all request method handlers declared in that controller. Like provider scope, the scope of a controller declares its lifetime. For a request-scoped controller, a new instance is created for each inbound request, and garbage-collected when the request has completed processing.

Declare controller scope with the `scope` property of the `ControllerOptions` object:

```
@Controller({
  path: 'cats',
  scope: Scope.REQUEST,
})
export class CatsController {}
```

Scope hierarchy

The `REQUEST` scope bubbles up the injection chain. A controller that depends on a request-scoped provider will, itself, be request-scoped.

Imagine the following dependency graph: `CatsController <- CatsService <- CatsRepository`. If `CatsService` is request-scoped (and the others are default singletons), the `CatsController` will become request-scoped as it is dependent on the injected service. The `CatsRepository`, which is not dependent, would remain singleton-scoped.

Transient-scoped dependencies don't follow that pattern. If a singleton-scoped `DogsService` injects a transient `LoggerService` provider, it will receive a fresh instance of it. However, `DogsService` will stay singleton-scoped, so injecting it anywhere would *not* resolve to a new instance of `DogsService`. In case it's desired behavior, `DogsService` must be explicitly marked as `TRANSIENT` as well.

Request provider

In an HTTP server-based application (e.g., using `@nestjs/platform-express` or `@nestjs/platform-fastify`), you may want to access a reference to the original request object when using request-scoped providers. You can do this by injecting the `REQUEST` object.

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { REQUEST } from '@nestjs/core';
import { Request } from 'express';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(REQUEST) private request: Request) {}
}
```

Because of underlying platform/protocol differences, you access the inbound request slightly differently for Microservice or GraphQL applications. In GraphQL applications, you inject `CONTEXT` instead of `REQUEST`:

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { CONTEXT } from '@nestjs/graphql';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(CONTEXT) private context) {}
}
```

You then configure your `context` value (in the `GraphQLModule`) to contain `request` as its property.

Inquirer provider

If you want to get the class where a provider was constructed, for instance in logging or metrics providers, you can inject the `INQUIRER` token.

```
import { Inject, Injectable, Scope } from '@nestjs/common';
import { INQUIRER } from '@nestjs/core';

@Injectable({ scope: Scope.TRANSIENT })
export class HelloService {
  constructor(@Inject(INQUIRER) private parentClass: object) {}

  sayHello(message: string) {
    console.log(`${this.parentClass?.constructor?.name}: ${message}`);
  }
}
```

And then use it as follows:

```
import { Injectable } from '@nestjs/common';
import { HelloService } from './hello.service';

@Injectable()
export class AppService {
```

```
    constructor(private helloService: HelloService) {}

    getRoot(): string {
      this.helloService.sayHello('My name is getRoot');

      return 'Hello world!';
    }
  }
```

In the example above when `AppService#getRoot` is called, `"AppService: My name is getRoot"` will be logged to the console.

Performance

Using request-scoped providers will have an impact on application performance. While Nest tries to cache as much metadata as possible, it will still have to create an instance of your class on each request. Hence, it will slow down your average response time and overall benchmarking result. Unless a provider must be request-scoped, it is strongly recommended that you use the default singleton scope.

> info **Hint** Although it all sounds quite intimidating, a properly designed application that leverages request-scoped providers should not slow down by more than ~5% latency-wise.

Durable providers

Request-scoped providers, as mentioned in the section above, may lead to increased latency since having at least 1 request-scoped provider (injected into the controller instance, or deeper - injected into one of its providers) makes the controller request-scoped as well. That means, it must be recreated (instantiated) per each individual request (and garbage collected afterwards). Now, that also means, that for let's say 30k requests in parallel, there will be 30k ephemeral instances of the controller (and its request-scoped providers).

Having a common provider that most providers depend on (think of a database connection, or a logger service), automatically converts all those providers to request-scoped providers as well. This can pose a challenge in **multi-tenant applications**, especially for those that have a central request-scoped "data source" provider that grabs headers/token from the request object and based on its values, retrieves the corresponding database connection/schema (specific to that tenant).

For instance, let's say you have an application alternately used by 10 different customers. Each customer has its **own dedicated data source**, and you want to make sure customer A will never be able to reach customer B's database. One way to achieve this could be to declare a request-scoped "data source" provider that - based on the request object - determines what's the "current customer" and retrieves its corresponding database. With this approach, you can turn your application into a multi-tenant application in just a few minutes. But, a major downside to this approach is that since most likely a large chunk of your application' components rely on the "data source" provider, they will implicitly become "request-scoped", and therefore you will undoubtedly see an impact in your apps performance.

But what if we had a better solution? Since we only have 10 customers, couldn't we have 10 individual DI sub-trees per customer (instead of recreating each tree per request)? If your providers don't rely on any property that's truly unique for each consecutive request (e.g., request UUID) but instead there're some

specific attributes that let us aggregate (classify) them, there's no reason to *recreate DI sub-tree* on every incoming request.

And that's exactly when the **durable providers** come in handy.

Before we start flagging providers as durable, we must first register a **strategy** that instructs Nest what are those "common request attributes", provide logic that groups requests - associates them with their corresponding DI sub-trees.

```
import {
  HostComponentInfo,
  ContextId,
  ContextIdFactory,
  ContextIdStrategy,
} from '@nestjs/core';
import { Request } from 'express';

const tenants = new Map<string, ContextId>();

export class AggregateByTenantContextIdStrategy implements
ContextIdStrategy {
  attach(contextId: ContextId, request: Request) {
    const tenantId = request.headers['x-tenant-id'] as string;
    let tenantSubTreeId: ContextId;

    if (tenants.has(tenantId)) {
      tenantSubTreeId = tenants.get(tenantId);
    } else {
      tenantSubTreeId = ContextIdFactory.create();
      tenants.set(tenantId, tenantSubTreeId);
    }

    // If tree is not durable, return the original "contextId" object
    return (info: HostComponentInfo) =>
      info.isTreeDurable ? tenantSubTreeId : contextId;
  }
}
```

> info **Hint** Similar to the request scope, durability bubbles up the injection chain. That means if A depends on B which is flagged as `durable`, A implicitly becomes durable too (unless `durable` is explicitly set to `false` for A provider).

> warning **Warning** Note this strategy is not ideal for applications operating with a large number of tenants.

The value returned from the `attach` method instructs Nest what context identifier should be used for a given host. In this case, we specified that the `tenantSubTreeId` should be used instead of the original, auto-generated `contextId` object, when the host component (e.g., request-scoped controller) is flagged as durable (you can learn how to mark providers as durable below). Also, in the above example, **no payload**

would be registered (where payload = REQUEST/CONTEXT provider that represents the "root" - parent of the sub-tree).

If you want to register the payload for a durable tree, use the following construction instead:

```
// The return of `AggregateByTenantContextIdStrategy#attach` method:
return {
  resolve: (info: HostComponentInfo) =>
    info.isTreeDurable ? tenantSubTreeId : contextId,
  payload: { tenantId },
  }
```

Now whenever you inject the REQUEST provider (or CONTEXT for GraphQL applications) using the @Inject(REQUEST)/@Inject(CONTEXT), the payload object would be injected (consisting of a single property - tenantId in this case).

Alright so with this strategy in place, you can register it somewhere in your code (as it applies globally anyway), so for example, you could place it in the main.ts file:

```
ContextIdFactory.apply(new AggregateByTenantContextIdStrategy());
```

> info **Hint** The ContextIdFactory class is imported from the @nestjs/core package.

As long as the registration occurs before any request hits your application, everything will work as intended.

Lastly, to turn a regular provider into a durable provider, simply set the durable flag to true and change its scope to Scope.REQUEST (not needed if the REQUEST scope is in the injection chain already):

```
import { Injectable, Scope } from '@nestjs/common';

@Injectable({ scope: Scope.REQUEST, durable: true })
export class CatsService {}
```

Similarly, for custom providers, set the durable property in the long-hand form for a provider registration:

```
{
  provide: 'foobar',
  useFactory: () => { ... },
  scope: Scope.REQUEST,
  durable: true,
}
```

Asynchronous providers

At times, the application start should be delayed until one or more **asynchronous tasks** are completed. For example, you may not want to start accepting requests until the connection with the database has been established. You can achieve this using asynchronous providers.

The syntax for this is to use `async/await` with the `useFactory` syntax. The factory returns a `Promise`, and the factory function can `await` asynchronous tasks. Nest will await resolution of the promise before instantiating any class that depends on (injects) such a provider.

```
{
  provide: 'ASYNC_CONNECTION',
  useFactory: async () => {
    const connection = await createConnection(options);
    return connection;
  },
}
```

> info **Hint** Learn more about custom provider syntax here.

Injection

Asynchronous providers are injected to other components by their tokens, like any other provider. In the example above, you would use the construct `@Inject('ASYNC_CONNECTION')`.

Example

The TypeORM recipe has a more substantial example of an asynchronous provider.

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();
```

Custom providers

In earlier chapters, we touched on various aspects of **Dependency Injection (DI)** and how it is used in Nest. One example of this is the constructor based dependency injection used to inject instances (often service providers) into classes. You won't be surprised to learn that Dependency Injection is built into the Nest core in a fundamental way. So far, we've only explored one main pattern. As your application grows more complex, you may need to take advantage of the full features of the DI system, so let's explore them in more detail.

DI fundamentals

Dependency injection is an inversion of control (IoC) technique wherein you delegate instantiation of dependencies to the IoC container (in our case, the NestJS runtime system), instead of doing it in your own code imperatively. Let's examine what's happening in this example from the Providers chapter.

First, we define a provider. The `@Injectable()` decorator marks the `CatsService` class as a provider.

```
@@filename(cats.service)
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  findAll(): Cat[] {
    return this.cats;
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class CatsService {
  constructor() {
    this.cats = [];
  }

  findAll() {
    return this.cats;
  }
}
```

Then we request that Nest inject the provider into our controller class:

```
@@filename(cats.controller)
import { Controller, Get } from '@nestjs/common';
import { CatsService } from './cats.service';
```

```
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
  }
}
@@switch
import { Controller, Get, Bind, Dependencies } from '@nestjs/common';
import { CatsService } from './cats.service';

@Controller('cats')
@Dependencies(CatsService)
export class CatsController {
  constructor(catsService) {
    this.catsService = catsService;
  }

  @Get()
  async findAll() {
    return this.catsService.findAll();
  }
}
```

Finally, we register the provider with the Nest IoC container:

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

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

What exactly is happening under the covers to make this work? There are three key steps in the process:

1. In `cats.service.ts`, the `@Injectable()` decorator declares the `CatsService` class as a class that can be managed by the Nest IoC container.
2. In `cats.controller.ts`, `CatsController` declares a dependency on the `CatsService` token with constructor injection:

```
  constructor(private catsService: CatsService)
```

3. In `app.module.ts`, we associate the token `CatsService` with the class `CatsService` from the `cats.service.ts` file. We'll see below exactly how this association (also called *registration*) occurs.

When the Nest IoC container instantiates a `CatsController`, it first looks for any dependencies*. When it finds the `CatsService` dependency, it performs a lookup on the `CatsService` token, which returns the `CatsService` class, per the registration step (#3 above). Assuming `SINGLETON` scope (the default behavior), Nest will then either create an instance of `CatsService`, cache it, and return it, or if one is already cached, return the existing instance.

*This explanation is a bit simplified to illustrate the point. One important area we glossed over is that the process of analyzing the code for dependencies is very sophisticated, and happens during application bootstrapping. One key feature is that dependency analysis (or "creating the dependency graph"), is **transitive**. In the above example, if the `CatsService` itself had dependencies, those too would be resolved. The dependency graph ensures that dependencies are resolved in the correct order - essentially "bottom up". This mechanism relieves the developer from having to manage such complex dependency graphs.

Standard providers

Let's take a closer look at the `@Module()` decorator. In `app.module`, we declare:

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

The `providers` property takes an array of `providers`. So far, we've supplied those providers via a list of class names. In fact, the syntax `providers: [CatsService]` is short-hand for the more complete syntax:

```
providers: [
  {
    provide: CatsService,
    useClass: CatsService,
  },
];
```

Now that we see this explicit construction, we can understand the registration process. Here, we are clearly associating the token `CatsService` with the class `CatsService`. The short-hand notation is merely a convenience to simplify the most common use-case, where the token is used to request an instance of a class by the same name.

Custom providers

What happens when your requirements go beyond those offered by *Standard providers*? Here are a few examples:

- You want to create a custom instance instead of having Nest instantiate (or return a cached instance of) a class
- You want to re-use an existing class in a second dependency
- You want to override a class with a mock version for testing

Nest allows you to define Custom providers to handle these cases. It provides several ways to define custom providers. Let's walk through them.

> info **Hint** If you are having problems with dependency resolution you can set the `NEST_DEBUG` environment variable and get extra dependency resolution logs during startup.

Value providers: `useValue`

The `useValue` syntax is useful for injecting a constant value, putting an external library into the Nest container, or replacing a real implementation with a mock object. Let's say you'd like to force Nest to use a mock `CatsService` for testing purposes.

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

const mockCatsService = {
  /* mock implementation
  ...
  */
};

@Module({
  imports: [CatsModule],
  providers: [
    {
      provide: CatsService,
      useValue: mockCatsService,
    },
  ],
})
export class AppModule {}
```

In this example, the `CatsService` token will resolve to the `mockCatsService` mock object. `useValue` requires a value - in this case a literal object that has the same interface as the `CatsService` class it is replacing. Because of TypeScript's structural typing, you can use any object that has a compatible interface, including a literal object or a class instance instantiated with `new`.

Non-class-based provider tokens

So far, we've used class names as our provider tokens (the value of the `provide` property in a provider listed in the `providers` array). This is matched by the standard pattern used with constructor based injection, where the token is also a class name. (Refer back to DI Fundamentals for a refresher on tokens if

this concept isn't entirely clear). Sometimes, we may want the flexibility to use strings or symbols as the DI token. For example:

```
import { connection } from './connection';

@Module({
  providers: [
    {
      provide: 'CONNECTION',
      useValue: connection,
    },
  ],
})
export class AppModule {}
```

In this example, we are associating a string-valued token (`'CONNECTION'`) with a pre-existing `connection` object we've imported from an external file.

> warning **Notice** In addition to using strings as token values, you can also use JavaScript symbols or TypeScript enums.

We've previously seen how to inject a provider using the standard constructor based injection pattern. This pattern **requires** that the dependency be declared with a class name. The `'CONNECTION'` custom provider uses a string-valued token. Let's see how to inject such a provider. To do so, we use the `@Inject()` decorator. This decorator takes a single argument - the token.

```
@@filename()
@Injectable()
export class CatsRepository {
  constructor(@Inject('CONNECTION') connection: Connection) {}
}
@@switch
@Injectable()
@Dependencies('CONNECTION')
export class CatsRepository {
  constructor(connection) {}
}
```

> info **Hint** The `@Inject()` decorator is imported from `@nestjs/common` package.

While we directly use the string `'CONNECTION'` in the above examples for illustration purposes, for clean code organization, it's best practice to define tokens in a separate file, such as `constants.ts`. Treat them much as you would symbols or enums that are defined in their own file and imported where needed.

Class providers: `useClass`

The `useClass` syntax allows you to dynamically determine a class that a token should resolve to. For example, suppose we have an abstract (or default) `ConfigService` class. Depending on the current

environment, we want Nest to provide a different implementation of the configuration service. The following code implements such a strategy.

```
const configServiceProvider = {
  provide: ConfigService,
  useClass:
    process.env.NODE_ENV === 'development'
      ? DevelopmentConfigService
      : ProductionConfigService,
};

@Module({
  providers: [configServiceProvider],
})
export class AppModule {}
```

Let's look at a couple of details in this code sample. You'll notice that we define `configServiceProvider` with a literal object first, then pass it in the module decorator's `providers` property. This is just a bit of code organization, but is functionally equivalent to the examples we've used thus far in this chapter.

Also, we have used the `ConfigService` class name as our token. For any class that depends on `ConfigService`, Nest will inject an instance of the provided class (`DevelopmentConfigService` or `ProductionConfigService`) overriding any default implementation that may have been declared elsewhere (e.g., a `ConfigService` declared with an `@Injectable()` decorator).

Factory providers: `useFactory`

The `useFactory` syntax allows for creating providers **dynamically**. The actual provider will be supplied by the value returned from a factory function. The factory function can be as simple or complex as needed. A simple factory may not depend on any other providers. A more complex factory can itself inject other providers it needs to compute its result. For the latter case, the factory provider syntax has a pair of related mechanisms:

1. The factory function can accept (optional) arguments.
2. The (optional) `inject` property accepts an array of providers that Nest will resolve and pass as arguments to the factory function during the instantiation process. Also, these providers can be marked as optional. The two lists should be correlated: Nest will pass instances from the `inject` list as arguments to the factory function in the same order. The example below demonstrates this.

```
@@filename()
const connectionProvider = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider, optionalProvider?:
string) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider, { token: 'SomeOptionalProvider', optional:
```

```
  true }],
    //       \_____________/           \__________________/
    //        This provider             The provider with this
    //        is mandatory.             token can resolve to `undefined`.
  };

  @Module({
    providers: [
      connectionProvider,
      OptionsProvider,
      // { provide: 'SomeOptionalProvider', useValue: 'anything' },
    ],
  })
  export class AppModule {}
  @@switch
  const connectionProvider = {
    provide: 'CONNECTION',
    useFactory: (optionsProvider, optionalProvider) => {
      const options = optionsProvider.get();
      return new DatabaseConnection(options);
    },
    inject: [OptionsProvider, { token: 'SomeOptionalProvider', optional:
  true }],
    //       \_____________/           \__________________/
    //        This provider             The provider with this
    //        is mandatory.             token can resolve to `undefined`.
  };

  @Module({
    providers: [
      connectionProvider,
      OptionsProvider,
      // { provide: 'SomeOptionalProvider', useValue: 'anything' },
    ],
  })
  export class AppModule {}
```

Alias providers: `useExisting`

The `useExisting` syntax allows you to create aliases for existing providers. This creates two ways to access the same provider. In the example below, the (string-based) token `'AliasedLoggerService'` is an alias for the (class-based) token `LoggerService`. Assume we have two different dependencies, one for `'AliasedLoggerService'` and one for `LoggerService`. If both dependencies are specified with `SINGLETON` scope, they'll both resolve to the same instance.

```
  @Injectable()
  class LoggerService {
    /* implementation details */
  }
```

```
const loggerAliasProvider = {
  provide: 'AliasedLoggerService',
  useExisting: LoggerService,
};

@Module({
  providers: [LoggerService, loggerAliasProvider],
})
export class AppModule {}
```

Non-service based providers

While providers often supply services, they are not limited to that usage. A provider can supply **any** value. For example, a provider may supply an array of configuration objects based on the current environment, as shown below:

```
const configFactory = {
  provide: 'CONFIG',
  useFactory: () => {
    return process.env.NODE_ENV === 'development' ? devConfig :
prodConfig;
  },
};

@Module({
  providers: [configFactory],
})
export class AppModule {}
```

Export custom provider

Like any provider, a custom provider is scoped to its declaring module. To make it visible to other modules, it must be exported. To export a custom provider, we can either use its token or the full provider object.

The following example shows exporting using the token:

```
@@filename()
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
```

```
  exports: ['CONNECTION'],
})
export class AppModule {}
@@switch
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: ['CONNECTION'],
})
export class AppModule {}
```

Alternatively, export with the full provider object:

```
@@filename()
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: [connectionFactory],
})
export class AppModule {}
@@switch
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: [connectionFactory],
})
export class AppModule {}
```

Circular dependency

A circular dependency occurs when two classes depend on each other. For example, class A needs class B, and class B also needs class A. Circular dependencies can arise in Nest between modules and between providers.

While circular dependencies should be avoided where possible, you can't always do so. In such cases, Nest enables resolving circular dependencies between providers in two ways. In this chapter, we describe using **forward referencing** as one technique, and using the **ModuleRef** class to retrieve a provider instance from the DI container as another.

We also describe resolving circular dependencies between modules.

> warning **Warning** A circular dependency might also be caused when using "barrel files"/index.ts files to group imports. Barrel files should be omitted when it comes to module/provider classes. For example, barrel files should not be used when importing files within the same directory as the barrel file, i.e. `cats/cats.controller` should not import `cats` to import the `cats/cats.service` file. For more details please also see this github issue.

Forward reference

A **forward reference** allows Nest to reference classes which aren't yet defined using the `forwardRef()` utility function. For example, if `CatsService` and `CommonService` depend on each other, both sides of the relationship can use `@Inject()` and the `forwardRef()` utility to resolve the circular dependency. Otherwise Nest won't instantiate them because all of the essential metadata won't be available. Here's an example:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(
    @Inject(forwardRef(() => CommonService))
    private commonService: CommonService,
  ) {}
}
@@switch
@Injectable()
@Dependencies(forwardRef(() => CommonService))
export class CatsService {
  constructor(commonService) {
    this.commonService = commonService;
  }
}
```

> info **Hint** The `forwardRef()` function is imported from the `@nestjs/common` package.

That covers one side of the relationship. Now let's do the same with `CommonService`:

```
@@filename(common.service)
@Injectable()
export class CommonService {
  constructor(
    @Inject(forwardRef(() => CatsService))
    private catsService: CatsService,
  ) {}
}
@@switch
@Injectable()
@Dependencies(forwardRef(() => CatsService))
export class CommonService {
  constructor(catsService) {
    this.catsService = catsService;
  }
}
```

> warning **Warning** The order of instantiation is indeterminate. Make sure your code does not depend on which constructor is called first. Having circular dependencies depend on providers with `Scope.REQUEST` can lead to undefined dependencies. More information available here

ModuleRef class alternative

An alternative to using `forwardRef()` is to refactor your code and use the `ModuleRef` class to retrieve a provider on one side of the (otherwise) circular relationship. Learn more about the `ModuleRef` utility class here.

Module forward reference

In order to resolve circular dependencies between modules, use the same `forwardRef()` utility function on both sides of the modules association. For example:

```
@@filename(common.module)
@Module({
  imports: [forwardRef(() => CatsModule)],
})
export class CommonModule {}
```

That covers one side of the relationship. Now let's do the same with `CatsModule`:

```
@@filename(cats.module)
@Module({
  imports: [forwardRef(() => CommonModule)],
})
export class CatsModule {}
```

Module reference

Nest provides the `ModuleRef` class to navigate the internal list of providers and obtain a reference to any provider using its injection token as a lookup key. The `ModuleRef` class also provides a way to dynamically instantiate both static and scoped providers. `ModuleRef` can be injected into a class in the normal way:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(private moduleRef: ModuleRef) {}
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }
}
```

> info **Hint** The `ModuleRef` class is imported from the `@nestjs/core` package.

Retrieving instances

The `ModuleRef` instance (hereafter we'll refer to it as the **module reference**) has a `get()` method. This method retrieves a provider, controller, or injectable (e.g., guard, interceptor, etc.) that exists (has been instantiated) in the **current** module using its injection token/class name.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private service: Service;
  constructor(private moduleRef: ModuleRef) {}

  onModuleInit() {
    this.service = this.moduleRef.get(Service);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  onModuleInit() {
    this.service = this.moduleRef.get(Service);
```

```
  }
}
```

> warning **Warning** You can't retrieve scoped providers (transient or request-scoped) with the `get()` method. Instead, use the technique described below. Learn how to control scopes here.

To retrieve a provider from the global context (for example, if the provider has been injected in a different module), pass the `{{ '{' }} strict: false {{ '}' }}` option as a second argument to `get()`.

```
this.moduleRef.get(Service, { strict: false });
```

Resolving scoped providers

To dynamically resolve a scoped provider (transient or request-scoped), use the `resolve()` method, passing the provider's injection token as an argument.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private transientService: TransientService;
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    this.transientService = await
this.moduleRef.resolve(TransientService);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    this.transientService = await
this.moduleRef.resolve(TransientService);
  }
}
```

The `resolve()` method returns a unique instance of the provider, from its own **DI container sub-tree**. Each sub-tree has a unique **context identifier**. Thus, if you call this method more than once and compare instance references, you will see that they are not equal.

```

```

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService),
      this.moduleRef.resolve(TransientService),
    ]);
    console.log(transientServices[0] === transientServices[1]); // false
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService),
      this.moduleRef.resolve(TransientService),
    ]);
    console.log(transientServices[0] === transientServices[1]); // false
  }
}
```

To generate a single instance across multiple `resolve()` calls, and ensure they share the same generated DI container sub-tree, you can pass a context identifier to the `resolve()` method. Use the `ContextIdFactory` class to generate a context identifier. This class provides a `create()` method that returns an appropriate unique identifier.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService, contextId),
      this.moduleRef.resolve(TransientService, contextId),
    ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}
@@switch
```

```
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService, contextId),
      this.moduleRef.resolve(TransientService, contextId),
    ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}
```

info **Hint** The `ContextIdFactory` class is imported from the `@nestjs/core` package.

Registering `REQUEST` provider

Manually generated context identifiers (with `ContextIdFactory.create()`) represent DI sub-trees in which `REQUEST` provider is `undefined` as they are not instantiated and managed by the Nest dependency injection system.

To register a custom `REQUEST` object for a manually created DI sub-tree, use the `ModuleRef#registerRequestByContextId()` method, as follows:

```
const contextId = ContextIdFactory.create();
this.moduleRef.registerRequestByContextId(/* YOUR_REQUEST_OBJECT */,
contextId);
```

Getting current sub-tree

Occasionally, you may want to resolve an instance of a request-scoped provider within a **request context**. Let's say that `CatsService` is request-scoped and you want to resolve the `CatsRepository` instance which is also marked as a request-scoped provider. In order to share the same DI container sub-tree, you must obtain the current context identifier instead of generating a new one (e.g., with the `ContextIdFactory.create()` function, as shown above). To obtain the current context identifier, start by injecting the request object using `@Inject()` decorator.

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(
    @Inject(REQUEST) private request: Record<string, unknown>,
  ) {}
}
```

```
@@switch
@Injectable()
@Dependencies(REQUEST)
export class CatsService {
  constructor(request) {
    this.request = request;
  }
}
```

> info **Hint** Learn more about the request provider here.

Now, use the `getByRequest()` method of the `ContextIdFactory` class to create a context id based on the request object, and pass this to the `resolve()` call:

```
const contextId = ContextIdFactory.getByRequest(this.request);
const catsRepository = await this.moduleRef.resolve(CatsRepository,
contextId);
```

Instantiating custom classes dynamically

To dynamically instantiate a class that **wasn't previously registered** as a **provider**, use the module reference's `create()` method.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private catsFactory: CatsFactory;
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}
```

This technique enables you to conditionally instantiate different classes outside of the framework container.

Lazy-loading modules

By default, modules are eagerly loaded, which means that as soon as the application loads, so do all the modules, whether or not they are immediately necessary. While this is fine for most applications, it may become a bottleneck for apps/workers running in the **serverless environment**, where the startup latency ("cold start") is crucial.

Lazy loading can help decrease bootstrap time by loading only modules required by the specific serverless function invocation. In addition, you could also load other modules asynchronously once the serverless function is "warm" to speed-up the bootstrap time for subsequent calls even further (deferred modules registration).

> info **Hint** If you're familiar with the **Angular** framework, you might have seen the "lazy-loading modules" term before. Be aware that this technique is **functionally different** in Nest and so think about this as an entirely different feature that shares similar naming conventions.

> warning **Warning** Do note that lifecycle hooks methods are not invoked in lazy loaded modules and services.

Getting started

To load modules on-demand, Nest provides the `LazyModuleLoader` class that can be injected into a class in the normal way:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(private lazyModuleLoader: LazyModuleLoader) {}
}
@@switch
@Injectable()
@Dependencies(LazyModuleLoader)
export class CatsService {
  constructor(lazyModuleLoader) {
    this.lazyModuleLoader = lazyModuleLoader;
  }
}
```

> info **Hint** The `LazyModuleLoader` class is imported from the `@nestjs/core` package.

Alternatively, you can obtain a reference to the `LazyModuleLoader` provider from within your application bootstrap file (`main.ts`), as follows:

```
// "app" represents a Nest application instance
const lazyModuleLoader = app.get(LazyModuleLoader);
```

With this in place, you can now load any module using the following construction:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);
```

info **Hint** "Lazy-loaded" modules are **cached** upon the first `LazyModuleLoader#load` method invocation. That means, each consecutive attempt to load `LazyModule` will be **very fast** and will return a cached instance, instead of loading the module again.

```
Load "LazyModule" attempt: 1
time: 2.379ms
Load "LazyModule" attempt: 2
time: 0.294ms
Load "LazyModule" attempt: 3
time: 0.303ms
```

Also, "lazy-loaded" modules share the same modules graph as those eagerly loaded on the application bootstrap as well as any other lazy modules registered later in your app.

Where `lazy.module.ts` is a TypeScript file that exports a **regular Nest module** (no extra changes are required).

The `LazyModuleLoader#load` method returns the module reference (of `LazyModule`) that lets you navigate the internal list of providers and obtain a reference to any provider using its injection token as a lookup key.

For example, let's say we have a `LazyModule` with the following definition:

```
@Module({
  providers: [LazyService],
  exports: [LazyService],
})
export class LazyModule {}
```

info **Hint** Lazy-loaded modules cannot be registered as **global modules** as it simply makes no sense (since they are registered lazily, on-demand when all the statically registered modules have been already instantiated). Likewise, registered **global enhancers** (guards/interceptors/etc.) **will not work** properly either.

With this, we could obtain a reference to the `LazyService` provider, as follows:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);

const { LazyService } = await import('./lazy.service');
const lazyService = moduleRef.get(LazyService);
```

> warning **Warning** If you use **Webpack**, make sure to update your `tsconfig.json` file - setting `compilerOptions.module` to `"esnext"` and adding `compilerOptions.moduleResolution` property with `"node"` as a value:
>
> ```
> {
>   "compilerOptions": {
>     "module": "esnext",
>     "moduleResolution": "node",
>     ...
>   }
> }
> ```
>
> With these options set up, you'll be able to leverage the code splitting feature.

Lazy-loading controllers, gateways, and resolvers

Since controllers (or resolvers in GraphQL applications) in Nest represent sets of routes/paths/topics (or queries/mutations), you **cannot lazy load them** using the `LazyModuleLoader` class.

> error **Warning** Controllers, resolvers, and gateways registered inside lazy-loaded modules will not behave as expected. Similarly, you cannot register middleware functions (by implementing the `MiddlewareConsumer` interface) on-demand.

For example, let's say you're building a REST API (HTTP application) with a Fastify driver under the hood (using the `@nestjs/platform-fastify` package). Fastify does not let you register routes after the application is ready/successfully listening to messages. That means even if we analyzed route mappings registered in the module's controllers, all lazy-loaded routes wouldn't be accessible since there is no way to register them at runtime.

Likewise, some transport strategies we provide as part of the `@nestjs/microservices` package (including Kafka, gRPC, or RabbitMQ) require to subscribe/listen to specific topics/channels before the connection is established. Once your application starts listening to messages, the framework would not be able to subscribe/listen to new topics.

Lastly, the `@nestjs/graphql` package with the code first approach enabled automatically generates the GraphQL schema on-the-fly based on the metadata. That means, it requires all classes to be loaded beforehand. Otherwise, it would not be doable to create the appropriate, valid schema.

Common use-cases

Most commonly, you will see lazy loaded modules in situations when your worker/cron job/lambda & serverless function/webhook must trigger different services (different logic) based on the input arguments (route path/date/query parameters, etc.). On the other hand, lazy-loading modules may not make too much sense for monolithic applications, where the startup time is rather irrelevant.

Execution context

Nest provides several utility classes that help make it easy to write applications that function across multiple application contexts (e.g., Nest HTTP server-based, microservices and WebSockets application contexts). These utilities provide information about the current execution context which can be used to build generic guards, filters, and interceptors that can work across a broad set of controllers, methods, and execution contexts.

We cover two such classes in this chapter: `ArgumentsHost` and `ExecutionContext`.

ArgumentsHost class

The `ArgumentsHost` class provides methods for retrieving the arguments being passed to a handler. It allows choosing the appropriate context (e.g., HTTP, RPC (microservice), or WebSockets) to retrieve the arguments from. The framework provides an instance of `ArgumentsHost`, typically referenced as a `host` parameter, in places where you may want to access it. For example, the `catch()` method of an exception filter is called with an `ArgumentsHost`instance.

`ArgumentsHost` simply acts as an abstraction over a handler's arguments. For example, for HTTP server applications (when `@nestjs/platform-express` is being used), the `host` object encapsulates Express's `[request, response, next]` array, where `request` is the request object, `response` is the response object, and `next` is a function that controls the application's request-response cycle. On the other hand, for GraphQL applications, the `host` object contains the `[root, args, context, info]` array.

Current application context

When building generic guards, filters, and interceptors which are meant to run across multiple application contexts, we need a way to determine the type of application that our method is currently running in. Do this with the `getType()` method of `ArgumentsHost`:

```
if (host.getType() === 'http') {
  // do something that is only important in the context of regular HTTP
requests (REST)
} else if (host.getType() === 'rpc') {
  // do something that is only important in the context of Microservice
requests
} else if (host.getType<GqlContextType>() === 'graphql') {
  // do something that is only important in the context of GraphQL
requests
}
```

> info **Hint** The `GqlContextType` is imported from the `@nestjs/graphql` package.

With the application type available, we can write more generic components, as shown below.

Host handler arguments

To retrieve the array of arguments being passed to the handler, one approach is to use the host object's `getArgs()` method.

```
const [req, res, next] = host.getArgs();
```

You can pluck a particular argument by index using the `getArgByIndex()` method:

```
const request = host.getArgByIndex(0);
const response = host.getArgByIndex(1);
```

In these examples we retrieved the request and response objects by index, which is not typically recommended as it couples the application to a particular execution context. Instead, you can make your code more robust and reusable by using one of the `host` object's utility methods to switch to the appropriate application context for your application. The context switch utility methods are shown below.

```
/**
 * Switch context to RPC.
 */
switchToRpc(): RpcArgumentsHost;
/**
 * Switch context to HTTP.
 */
switchToHttp(): HttpArgumentsHost;
/**
 * Switch context to WebSockets.
 */
switchToWs(): WsArgumentsHost;
```

Let's rewrite the previous example using the `switchToHttp()` method. The `host.switchToHttp()` helper call returns an `HttpArgumentsHost` object that is appropriate for the HTTP application context. The `HttpArgumentsHost` object has two useful methods we can use to extract the desired objects. We also use the Express type assertions in this case to return native Express typed objects:

```
const ctx = host.switchToHttp();
const request = ctx.getRequest<Request>();
const response = ctx.getResponse<Response>();
```

Similarly `WsArgumentsHost` and `RpcArgumentsHost` have methods to return appropriate objects in the microservices and WebSockets contexts. Here are the methods for `WsArgumentsHost`:

```
export interface WsArgumentsHost {
  /**
   * Returns the data object.
```

```
   */
  getData<T>(): T;
  /**
   * Returns the client object.
   */
  getClient<T>(): T;
}
```

Following are the methods for `RpcArgumentsHost`:

```
export interface RpcArgumentsHost {
  /**
   * Returns the data object.
   */
  getData<T>(): T;

  /**
   * Returns the context object.
   */
  getContext<T>(): T;
}
```

ExecutionContext class

`ExecutionContext` extends `ArgumentsHost`, providing additional details about the current execution process. Like `ArgumentsHost`, Nest provides an instance of `ExecutionContext` in places you may need it, such as in the `canActivate()` method of a guard and the `intercept()` method of an interceptor. It provides the following methods:

```
export interface ExecutionContext extends ArgumentsHost {
  /**
   * Returns the type of the controller class which the current handler
belongs to.
   */
  getClass<T>(): Type<T>;
  /**
   * Returns a reference to the handler (method) that will be invoked next
in the
   * request pipeline.
   */
  getHandler(): Function;
}
```

The `getHandler()` method returns a reference to the handler about to be invoked. The `getClass()` method returns the type of the `Controller` class which this particular handler belongs to. For example, in an HTTP context, if the currently processed request is a `POST` request, bound to the `create()` method on

the `CatsController`, `getHandler()` returns a reference to the `create()` method and `getClass()` returns the `CatsController` **type** (not instance).

```
const methodKey = ctx.getHandler().name; // "create"
const className = ctx.getClass().name; // "CatsController"
```

The ability to access references to both the current class and handler method provides great flexibility. Most importantly, it gives us the opportunity to access the metadata set through either decorators created via `Reflector#createDecorator` or the built-in `@SetMetadata()` decorator from within guards or interceptors. We cover this use case below.

Reflection and metadata

Nest provides the ability to attach **custom metadata** to route handlers through decorators created via `Reflector#createDecorator` method, and the built-in `@SetMetadata()` decorator. In this section, let's compare the two approaches and see how to access the metadata from within a guard or interceptor.

To create strongly-typed decorators using `Reflector#createDecorator`, we need to specify the type argument. For example, let's create a `Roles` decorator that takes an array of strings as an argument.

```
@@filename(roles.decorator)
import { Reflector } from '@nestjs/core';

export const Roles = Reflector.createDecorator<string[]>();
```

The `Roles` decorator here is a function that takes a single argument of type `string[]`.

Now, to use this decorator, we simply annotate the handler with it:

```
@@filename(cats.controller)
@Post()
@Roles(['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@Roles(['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

Here we've attached the `Roles` decorator metadata to the `create()` method, indicating that only users with the `admin` role should be allowed to access this route.

To access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again. `Reflector` can be injected into a class in the normal way:

```
@@filename(roles.guard)
@Injectable()
export class RolesGuard {
  constructor(private reflector: Reflector) {}
}
@@switch
@Injectable()
@Dependencies(Reflector)
export class CatsService {
  constructor(reflector) {
    this.reflector = reflector;
  }
}
```

> info **Hint** The `Reflector` class is imported from the `@nestjs/core` package.

Now, to read the handler metadata, use the `get()` method:

```
const roles = this.reflector.get(Roles, context.getHandler());
```

The `Reflector#get` method allows us to easily access the metadata by passing in two arguments: a decorator reference and a **context** (decorator target) to retrieve the metadata from. In this example, the specified **decorator** is `Roles` (refer back to the `roles.decorator.ts` file above). The context is provided by the call to `context.getHandler()`, which results in extracting the metadata for the currently processed route handler. Remember, `getHandler()` gives us a **reference** to the route handler function.

Alternatively, we may organize our controller by applying metadata at the controller level, applying to all routes in the controller class.

```
@@filename(cats.controller)
@Roles(['admin'])
@Controller('cats')
export class CatsController {}
@@switch
@Roles(['admin'])
@Controller('cats')
export class CatsController {}
```

In this case, to extract controller metadata, we pass `context.getClass()` as the second argument (to provide the controller class as the context for metadata extraction) instead of `context.getHandler()`:

```
@@filename(roles.guard)
const roles = this.reflector.get(Roles, context.getClass());
```

Given the ability to provide metadata at multiple levels, you may need to extract and merge metadata from several contexts. The `Reflector` class provides two utility methods used to help with this. These methods extract **both** controller and method metadata at once, and combine them in different ways.

Consider the following scenario, where you've supplied `Roles` metadata at both levels.

```
@@filename(cats.controller)
@Roles(['user'])
@Controller('cats')
export class CatsController {
  @Post()
  @Roles(['admin'])
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }
}
@@switch
@Roles(['user'])
@Controller('cats')
export class CatsController {}
  @Post()
  @Roles(['admin'])
  @Bind(Body())
  async create(createCatDto) {
    this.catsService.create(createCatDto);
  }
}
```

If your intent is to specify `'user'` as the default role, and override it selectively for certain methods, you would probably use the `getAllAndOverride()` method.

```
const roles = this.reflector.getAllAndOverride(Roles,
[context.getHandler(), context.getClass()]);
```

A guard with this code, running in the context of the `create()` method, with the above metadata, would result in `roles` containing `['admin']`.

To get metadata for both and merge it (this method merges both arrays and objects), use the `getAllAndMerge()` method:

```
const roles = this.reflector.getAllAndMerge(Roles, [context.getHandler(),
context.getClass()]);
```

This would result in `roles` containing `['user', 'admin']`.

For both of these merge methods, you pass the metadata key as the first argument, and an array of metadata target contexts (i.e., calls to the `getHandler()` and/or `getClass())` methods) as the second argument.

Low-level approach

As mentioned earlier, instead of using `Reflector#createDecorator`, you can also use the built-in `@SetMetadata()` decorator to attach metadata to a handler.

```
@@filename(cats.controller)
@Post()
@SetMetadata('roles', ['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@SetMetadata('roles', ['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

info **Hint** The `@SetMetadata()` decorator is imported from the `@nestjs/common` package.

With the construction above, we attached the `roles` metadata (`roles` is a metadata key and `['admin']` is the associated value) to the `create()` method. While this works, it's not good practice to use `@SetMetadata()` directly in your routes. Instead, you can create your own decorators, as shown below:

```
@@filename(roles.decorator)
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles: string[]) => SetMetadata('roles', roles);
@@switch
import { SetMetadata } from '@nestjs/common';

export const Roles = (...roles) => SetMetadata('roles', roles);
```

This approach is much cleaner and more readable, and somewhat resembles the `Reflector#createDecorator` approach. The difference is that with `@SetMetadata` you have more control over the metadata key and value, and also can create decorators that take more than one argument.

Now that we have a custom `@Roles()` decorator, we can use it to decorate the `create()` method.

```
@@filename(cats.controller)
@Post()
@Roles('admin')
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@Roles('admin')
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

To access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again:

```
@@filename(roles.guard)
@Injectable()
export class RolesGuard {
  constructor(private reflector: Reflector) {}
}
@@switch
@Injectable()
@Dependencies(Reflector)
export class CatsService {
  constructor(reflector) {
    this.reflector = reflector;
  }
}
```

> info **Hint** The `Reflector` class is imported from the `@nestjs/core` package.

Now, to read the handler metadata, use the `get()` method.

```
const roles = this.reflector.get<string[]>('roles', context.getHandler());
```

Here instead of passing a decorator reference, we pass the metadata **key** as the first argument (which in our case is `'roles'`). Everything else remains the same as in the `Reflector#createDecorator` example.

Lifecycle Events

A Nest application, as well as every application element, has a lifecycle managed by Nest. Nest provides **lifecycle hooks** that give visibility into key lifecycle events, and the ability to act (run registered code on your modules, providers or controllers) when they occur.

Lifecycle sequence

The following diagram depicts the sequence of key application lifecycle events, from the time the application is bootstrapped until the node process exits. We can divide the overall lifecycle into three phases: **initializing**, **running** and **terminating**. Using this lifecycle, you can plan for appropriate initialization of modules and services, manage active connections, and gracefully shutdown your application when it receives a termination signal.

Lifecycle events

Lifecycle events happen during application bootstrapping and shutdown. Nest calls registered lifecycle hook methods on modules, providers and controllers at each of the following lifecycle events (**shutdown hooks** need to be enabled first, as described below). As shown in the diagram above, Nest also calls the appropriate underlying methods to begin listening for connections, and to stop listening for connections.

In the following table, `onModuleDestroy`, `beforeApplicationShutdown` and `onApplicationShutdown` are only triggered if you explicitly call `app.close()` or if the process receives a special system signal (such as SIGTERM) and you have correctly called `enableShutdownHooks` at application bootstrap (see below **Application shutdown** part).

Lifecycle hook method	Lifecycle event triggering the hook method call
`onModuleInit()`	Called once the host module's dependencies have been resolved.
`onApplicationBootstrap()`	Called once all modules have been initialized, but before listening for connections.
`onModuleDestroy()*`	Called after a termination signal (e.g., `SIGTERM`) has been received.
`beforeApplicationShutdown()*`	Called after all `onModuleDestroy()` handlers have completed (Promises resolved or rejected); once complete (Promises resolved or rejected), all existing connections will be closed (`app.close()` called).
`onApplicationShutdown()*`	Called after connections close (`app.close()` resolves).

* For these events, if you're not calling `app.close()` explicitly, you must opt-in to make them work with system signals such as `SIGTERM`. See Application shutdown below.

> warning **Warning** The lifecycle hooks listed above are not triggered for **request-scoped** classes. Request-scoped classes are not tied to the application lifecycle and their lifespan is unpredictable. They are exclusively created for each request and automatically garbage-collected after the response is sent.

> info **Hint** Execution order of `onModuleInit()` and `onApplicationBootstrap()` directly depends on the order of module imports, awaiting the previous hook.

Usage

Each lifecycle hook is represented by an interface. Interfaces are technically optional because they do not exist after TypeScript compilation. Nonetheless, it's good practice to use them in order to benefit from strong typing and editor tooling. To register a lifecycle hook, implement the appropriate interface. For example, to register a method to be called during module initialization on a particular class (e.g., Controller, Provider or Module), implement the `OnModuleInit` interface by supplying an `onModuleInit()` method, as shown below:

```
@@filename()
import { Injectable, OnModuleInit } from '@nestjs/common';

@Injectable()
export class UsersService implements OnModuleInit {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class UsersService {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}
```

Asynchronous initialization

Both the `OnModuleInit` and `OnApplicationBootstrap` hooks allow you to defer the application initialization process (return a `Promise` or mark the method as `async` and `await` an asynchronous method completion in the method body).

```
@@filename()
async onModuleInit(): Promise<void> {
  await this.fetch();
}
@@switch
async onModuleInit() {
```

```
    await this.fetch();
  }
```

Application shutdown

The `onModuleDestroy()`, `beforeApplicationShutdown()` and `onApplicationShutdown()` hooks are called in the terminating phase (in response to an explicit call to `app.close()` or upon receipt of system signals such as SIGTERM if opted-in). This feature is often used with Kubernetes to manage containers' lifecycles, by Heroku for dynos or similar services.

Shutdown hook listeners consume system resources, so they are disabled by default. To use shutdown hooks, you **must enable listeners** by calling `enableShutdownHooks()`:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Starts listening for shutdown hooks
  app.enableShutdownHooks();

  await app.listen(3000);
}
bootstrap();
```

> warning **warning** Due to inherent platform limitations, NestJS has limited support for application shutdown hooks on Windows. You can expect `SIGINT` to work, as well as `SIGBREAK` and to some extent `SIGHUP` - read more. However `SIGTERM` will never work on Windows because killing a process in the task manager is unconditional, "i.e., there's no way for an application to detect or prevent it". Here's some relevant documentation from libuv to learn more about how `SIGINT`, `SIGBREAK` and others are handled on Windows. Also, see Node.js documentation of Process Signal Events

> info **Info** `enableShutdownHooks` consumes memory by starting listeners. In cases where you are running multiple Nest apps in a single Node process (e.g., when running parallel tests with Jest), Node may complain about excessive listener processes. For this reason, `enableShutdownHooks` is not enabled by default. Be aware of this condition when you are running multiple instances in a single Node process.

When the application receives a termination signal it will call any registered `onModuleDestroy()`, `beforeApplicationShutdown()`, then `onApplicationShutdown()` methods (in the sequence described above) with the corresponding signal as the first parameter. If a registered function awaits an asynchronous call (returns a promise), Nest will not continue in the sequence until the promise is resolved or rejected.

```
@@filename()
@Injectable()
```

```
class UsersService implements OnApplicationShutdown {
  onApplicationShutdown(signal: string) {
    console.log(signal); // e.g. "SIGINT"
  }
}
@@switch
@Injectable()
class UsersService implements OnApplicationShutdown {
  onApplicationShutdown(signal) {
    console.log(signal); // e.g. "SIGINT"
  }
}
```

> info **Info** Calling `app.close()` doesn't terminate the Node process but only triggers the `onModuleDestroy()` and `onApplicationShutdown()` hooks, so if there are some intervals, long-running background tasks, etc. the process won't be automatically terminated.

Platform agnosticism

Nest is a platform-agnostic framework. This means you can develop **reusable logical parts** that can be used across different types of applications. For example, most components can be re-used without change across different underlying HTTP server frameworks (e.g., Express and Fastify), and even across different *types* of applications (e.g., HTTP server frameworks, Microservices with different transport layers, and Web Sockets).

Build once, use everywhere

The **Overview** section of the documentation primarily shows coding techniques using HTTP server frameworks (e.g., apps providing a REST API or providing an MVC-style server-side rendered app). However, all those building blocks can be used on top of different transport layers (microservices or websockets).

Furthermore, Nest comes with a dedicated GraphQL module. You can use GraphQL as your API layer interchangeably with providing a REST API.

In addition, the application context feature helps to create any kind of Node.js application - including things like CRON jobs and CLI apps - on top of Nest.

Nest aspires to be a full-fledged platform for Node.js apps that brings a higher-level of modularity and reusability to your applications. Build once, use everywhere!

Testing

Automated testing is considered an essential part of any serious software development effort. Automation makes it easy to repeat individual tests or test suites quickly and easily during development. This helps ensure that releases meet quality and performance goals. Automation helps increase coverage and provides a faster feedback loop to developers. Automation both increases the productivity of individual developers and ensures that tests are run at critical development lifecycle junctures, such as source code control check-in, feature integration, and version release.

Such tests often span a variety of types, including unit tests, end-to-end (e2e) tests, integration tests, and so on. While the benefits are unquestionable, it can be tedious to set them up. Nest strives to promote development best practices, including effective testing, so it includes features such as the following to help developers and teams build and automate tests. Nest:

- automatically scaffolds default unit tests for components and e2e tests for applications
- provides default tooling (such as a test runner that builds an isolated module/application loader)
- provides integration with Jest and Supertest out-of-the-box, while remaining agnostic to testing tools
- makes the Nest dependency injection system available in the testing environment for easily mocking components

As mentioned, you can use any **testing framework** that you like, as Nest doesn't force any specific tooling. Simply replace the elements needed (such as the test runner), and you will still enjoy the benefits of Nest's ready-made testing facilities.

Installation

To get started, first install the required package:

```
$ npm i --save-dev @nestjs/testing
```

Unit testing

In the following example, we test two classes: `CatsController` and `CatsService`. As mentioned, Jest is provided as the default testing framework. It serves as a test-runner and also provides assert functions and test-double utilities that help with mocking, spying, etc. In the following basic test, we manually instantiate these classes, and ensure that the controller and service fulfill their API contract.

```
@@filename(cats.controller.spec)
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(() => {
```

```
    catsService = new CatsService();
    catsController = new CatsController(catsService);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});
@@switch
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController;
  let catsService;

  beforeEach(() => {
    catsService = new CatsService();
    catsController = new CatsController(catsService);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});
```

> info **Hint** Keep your test files located near the classes they test. Testing files should have a `.spec` or `.test` suffix.

Because the above sample is trivial, we aren't really testing anything Nest-specific. Indeed, we aren't even using dependency injection (notice that we pass an instance of `CatsService` to our `catsController`). This form of testing - where we manually instantiate the classes being tested - is often called **isolated testing** as it is independent from the framework. Let's introduce some more advanced capabilities that help you test applications that make more extensive use of Nest features.

Testing utilities

The `@nestjs/testing` package provides a set of utilities that enable a more robust testing process. Let's rewrite the previous example using the built-in `Test` class:

```
@@filename(cats.controller.spec)
import { Test } from '@nestjs/testing';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
        controllers: [CatsController],
        providers: [CatsService],
      }).compile();

    catsService = moduleRef.get<CatsService>(CatsService);
    catsController = moduleRef.get<CatsController>(CatsController);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});
@@switch
import { Test } from '@nestjs/testing';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController;
  let catsService;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
        controllers: [CatsController],
        providers: [CatsService],
      }).compile();

    catsService = moduleRef.get(CatsService);
    catsController = moduleRef.get(CatsController);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
```

```
    });
  });
});
```

The `Test` class is useful for providing an application execution context that essentially mocks the full Nest runtime, but gives you hooks that make it easy to manage class instances, including mocking and overriding. The `Test` class has a `createTestingModule()` method that takes a module metadata object as its argument (the same object you pass to the `@Module()` decorator). This method returns a `TestingModule` instance which in turn provides a few methods. For unit tests, the important one is the `compile()` method. This method bootstraps a module with its dependencies (similar to the way an application is bootstrapped in the conventional `main.ts` file using `NestFactory.create()`), and returns a module that is ready for testing.

> info **Hint** The `compile()` method is **asynchronous** and therefore has to be awaited. Once the module is compiled you can retrieve any **static** instance it declares (controllers and providers) using the `get()` method.

`TestingModule` inherits from the module reference class, and therefore its ability to dynamically resolve scoped providers (transient or request-scoped). Do this with the `resolve()` method (the `get()` method can only retrieve static instances).

```
const moduleRef = await Test.createTestingModule({
  controllers: [CatsController],
  providers: [CatsService],
}).compile();

catsService = await moduleRef.resolve(CatsService);
```

> warning **Warning** The `resolve()` method returns a unique instance of the provider, from its own **DI container sub-tree**. Each sub-tree has a unique context identifier. Thus, if you call this method more than once and compare instance references, you will see that they are not equal.

> info **Hint** Learn more about the module reference features here.

Instead of using the production version of any provider, you can override it with a custom provider for testing purposes. For example, you can mock a database service instead of connecting to a live database. We'll cover overrides in the next section, but they're available for unit tests as well.

Auto mocking

Nest also allows you to define a mock factory to apply to all of your missing dependencies. This is useful for cases where you have a large number of dependencies in a class and mocking all of them will take a long time and a lot of setup. To make use of this feature, the `createTestingModule()` will need to be chained up with the `useMocker()` method, passing a factory for your dependency mocks. This factory can take in an optional token, which is an instance token, any token which is valid for a Nest provider, and returns a mock implementation. The below is an example of creating a generic mocker using `jest-mock` and a specific mock for `CatsService` using `jest.fn()`.

```
// ...
import { ModuleMocker, MockFunctionMetadata } from 'jest-mock';

const moduleMocker = new ModuleMocker(global);

describe('CatsController', () => {
  let controller: CatsController;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
    })
      .useMocker((token) => {
        const results = ['test1', 'test2'];
        if (token === CatsService) {
          return { findAll: jest.fn().mockResolvedValue(results) };
        }
        if (typeof token === 'function') {
          const mockMetadata = moduleMocker.getMetadata(token) as
MockFunctionMetadata<any, any>;
          const Mock = moduleMocker.generateFromMetadata(mockMetadata);
          return new Mock();
        }
      })
      .compile();

    controller = moduleRef.get(CatsController);
  });
});
```

You can also retrieve these mocks out of the testing container as you normally would custom providers, `moduleRef.get(CatsService)`.

> info **Hint** A general mock factory, like `createMock` from `@golevelup/ts-jest` can also be passed directly.

> info **Hint** `REQUEST` and `INQUIRER` providers cannot be auto-mocked because they're already pre-defined in the context. However, they can be *overwritten* using the custom provider syntax or by utilizing the `.overrideProvider` method.

End-to-end testing

Unlike unit testing, which focuses on individual modules and classes, end-to-end (e2e) testing covers the interaction of classes and modules at a more aggregate level -- closer to the kind of interaction that end-users will have with the production system. As an application grows, it becomes hard to manually test the end-to-end behavior of each API endpoint. Automated end-to-end tests help us ensure that the overall behavior of the system is correct and meets project requirements. To perform e2e tests we use a similar configuration to the one we just covered in **unit testing**. In addition, Nest makes it easy to use the Supertest library to simulate HTTP requests.

```
@@filename(cats.e2e-spec)
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { CatsModule } from '../../src/cats/cats.module';
import { CatsService } from '../../src/cats/cats.service';
import { INestApplication } from '@nestjs/common';

describe('Cats', () => {
  let app: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
      .compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });

  it(`/GET cats`, () => {
    return request(app.getHttpServer())
      .get('/cats')
      .expect(200)
      .expect({
        data: catsService.findAll(),
      });
  });

  afterAll(async () => {
    await app.close();
  });
});
@@switch
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { CatsModule } from '../../src/cats/cats.module';
import { CatsService } from '../../src/cats/cats.service';
import { INestApplication } from '@nestjs/common';

describe('Cats', () => {
  let app: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
```

```
      .compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });

  it(`/GET cats`, () => {
    return request(app.getHttpServer())
      .get('/cats')
      .expect(200)
      .expect({
        data: catsService.findAll(),
      });
  });

  afterAll(async () => {
    await app.close();
  });
});
```

info **Hint** If you're using Fastify as your HTTP adapter, it requires a slightly different configuration, and has built-in testing capabilities:

```
let app: NestFastifyApplication;

beforeAll(async () => {
  app = moduleRef.createNestApplication<NestFastifyApplication>(new
FastifyAdapter());

  await app.init();
  await app.getHttpAdapter().getInstance().ready();
});

it(`/GET cats`, () => {
  return app
    .inject({
      method: 'GET',
      url: '/cats',
    })
    .then((result) => {
      expect(result.statusCode).toEqual(200);
      expect(result.payload).toEqual(/* expectedPayload */);
    });
});

afterAll(async () => {
  await app.close();
});
```

In this example, we build on some of the concepts described earlier. In addition to the `compile()` method we used earlier, we now use the `createNestApplication()` method to instantiate a full Nest runtime environment. We save a reference to the running app in our `app` variable so we can use it to simulate HTTP requests.

We simulate HTTP tests using the `request()` function from Supertest. We want these HTTP requests to route to our running Nest app, so we pass the `request()` function a reference to the HTTP listener that underlies Nest (which, in turn, may be provided by the Express platform). Hence the construction `request(app.getHttpServer())`. The call to `request()` hands us a wrapped HTTP Server, now connected to the Nest app, which exposes methods to simulate an actual HTTP request. For example, using `request(...).get('/cats')` will initiate a request to the Nest app that is identical to an **actual** HTTP request like `get '/cats'` coming in over the network.

In this example, we also provide an alternate (test-double) implementation of the `CatsService` which simply returns a hard-coded value that we can test for. Use `overrideProvider()` to provide such an alternate implementation. Similarly, Nest provides methods to override modules, guards, interceptors, filters and pipes with the `overrideModule()`, `overrideGuard()`, `overrideInterceptor()`, `overrideFilter()`, and `overridePipe()` methods respectively.

Each of the override methods (except for `overrideModule()`) returns an object with 3 different methods that mirror those described for custom providers:

- `useClass`: you supply a class that will be instantiated to provide the instance to override the object (provider, guard, etc.).
- `useValue`: you supply an instance that will override the object.
- `useFactory`: you supply a function that returns an instance that will override the object.

On the other hand, `overrideModule()` returns an object with the `useModule()` method, which you can use to supply a module that will override the original module, as follows:

```
const moduleRef = await Test.createTestingModule({
  imports: [AppModule],
})
  .overrideModule(CatsModule)
  .useModule(AlternateCatsModule)
  .compile();
```

Each of the override method types, in turn, returns the `TestingModule` instance, and can thus be chained with other methods in the fluent style. You should use `compile()` at the end of such a chain to cause Nest to instantiate and initialize the module.

Also, sometimes you may want to provide a custom logger e.g. when the tests are run (for example, on a CI server). Use the `setLogger()` method and pass an object that fulfills the `LoggerService` interface to instruct the `TestModuleBuilder` how to log during tests (by default, only "error" logs will be logged to the console).

> warning **Warning** The `@nestjs/core` package exposes unique provider tokens with the `APP_` prefix to help on define global enhancers. Those tokens cannot be overridden since they can represent

> multiple providers. Thus you can't use `.overrideProvider(APP_GUARD)` (and so on). If you want to override some global enhancer, follow this workaround.

The compiled module has several useful methods, as described in the following table:

`createNestApplication()`	Creates and returns a Nest application (`INestApplication` instance) based on the given module. Note that you must manually initialize the application using the `init()` method.
`createNestMicroservice()`	Creates and returns a Nest microservice (`INestMicroservice` instance) based on the given module.
`get()`	Retrieves a static instance of a controller or provider (including guards, filters, etc.) available in the application context. Inherited from the module reference class.
`resolve()`	Retrieves a dynamically created scoped instance (request or transient) of a controller or provider (including guards, filters, etc.) available in the application context. Inherited from the module reference class.
`select()`	Navigates through the module's dependency graph; can be used to retrieve a specific instance from the selected module (used along with strict mode (`strict: true`) in `get()` method).

> info **Hint** Keep your e2e test files inside the `test` directory. The testing files should have a `.e2e-spec` suffix.

Overriding globally registered enhancers

If you have a globally registered guard (or pipe, interceptor, or filter), you need to take a few more steps to override that enhancer. To recap the original registration looks like this:

```
providers: [
  {
    provide: APP_GUARD,
    useClass: JwtAuthGuard,
  },
],
```

This is registering the guard as a "multi"-provider through the `APP_*` token. To be able to replace the `JwtAuthGuard` here, the registration needs to use an existing provider in this slot:

```
providers: [
  {
    provide: APP_GUARD,
    useExisting: JwtAuthGuard,
    // ^^^^^^^^ notice the use of 'useExisting' instead of 'useClass'
  },
```

```
    JwtAuthGuard,
  ],
```

> info **Hint** Change the `useClass` to `useExisting` to reference a registered provider instead of having Nest instantiate it behind the token.

Now the `JwtAuthGuard` is visible to Nest as a regular provider that can be overridden when creating the `TestingModule`:

```
const moduleRef = await Test.createTestingModule({
  imports: [AppModule],
})
  .overrideProvider(JwtAuthGuard)
  .useClass(MockAuthGuard)
  .compile();
```

Now all your tests will use the `MockAuthGuard` on every request.

Testing request-scoped instances

Request-scoped providers are created uniquely for each incoming **request**. The instance is garbage-collected after the request has completed processing. This poses a problem, because we can't access a dependency injection sub-tree generated specifically for a tested request.

We know (based on the sections above) that the `resolve()` method can be used to retrieve a dynamically instantiated class. Also, as described here, we know we can pass a unique context identifier to control the lifecycle of a DI container sub-tree. How do we leverage this in a testing context?

The strategy is to generate a context identifier beforehand and force Nest to use this particular ID to create a sub-tree for all incoming requests. In this way we'll be able to retrieve instances created for a tested request.

To accomplish this, use `jest.spyOn()` on the `ContextIdFactory`:

```
const contextId = ContextIdFactory.create();
jest.spyOn(ContextIdFactory, 'getByRequest').mockImplementation(() =>
contextId);
```

Now we can use the `contextId` to access a single generated DI container sub-tree for any subsequent request.

```
catsService = await moduleRef.resolve(CatsService, contextId);
```