Providers

Providers are a fundamental concept in Nest. Many of the basic Nest classes may be treated as a provider – services, repositories, factories, helpers, and so on. The main idea of a provider is that it can be **injected** as a dependency; this means objects can create various relationships with each other, and the function of "wiring up" instances of objects can largely be delegated to the Nest runtime system.

In the previous chapter, we built a simple `CatsController`. Controllers should handle HTTP requests and delegate more complex tasks to **providers**. Providers are plain JavaScript classes that are declared as `providers` in a module.

> info **Hint** Since Nest enables the possibility to design and organize dependencies in a more OO way, we strongly recommend following the SOLID principles.

Services

Let's start by creating a simple `CatsService`. This service will be responsible for data storage and retrieval, and is designed to be used by the `CatsController`, so it's a good candidate to be defined 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[] = [];

  create(cat: Cat) {
    this.cats.push(cat);
  }

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

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

  create(cat) {
    this.cats.push(cat);
  }
  findAll() {
    return this.cats;
  }
}
```

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

Our `CatsService` is a basic class with one property and two methods. The only new feature is that it uses the `@Injectable()` decorator. The `@Injectable()` decorator attaches metadata, which declares that `CatsService` is a class that can be managed by the Nest IoC container. By the way, this example also uses a `Cat` interface, which probably looks something like this:

```
@@filename(interfaces/cat.interface)
export interface Cat {
  name: string;
  age: number;
  breed: string;
}
```

Now that we have a service class to retrieve cats, let's use it inside the `CatsController`:

```
@@filename(cats.controller)
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

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

  @Post()
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }

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

@Controller('cats')
```

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

  @Post()
  @Bind(Body())
  async create(createCatDto) {
    this.catsService.create(createCatDto);
  }

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

The `CatsService` is **injected** through the class constructor. Notice the use of the `private` syntax. This shorthand allows us to both declare and initialize the `catsService` member immediately in the same location.

Dependency injection

Nest is built around the strong design pattern commonly known as **Dependency injection**. We recommend reading a great article about this concept in the official Angular documentation.

In Nest, thanks to TypeScript capabilities, it's extremely easy to manage dependencies because they are resolved just by type. In the example below, Nest will resolve the `catsService` by creating and returning an instance of `CatsService` (or, in the normal case of a singleton, returning the existing instance if it has already been requested elsewhere). This dependency is resolved and passed to your controller's constructor (or assigned to the indicated property):

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

Scopes

Providers normally have a lifetime ("scope") synchronized with the application lifecycle. When the application is bootstrapped, every dependency must be resolved, and therefore every provider has to be instantiated. Similarly, when the application shuts down, each provider will be destroyed. However, there are ways to make your provider lifetime **request-scoped** as well. You can read more about these techniques here.

Custom providers

Nest has a built-in inversion of control ("IoC") container that resolves relationships between providers. This feature underlies the dependency injection feature described above, but is in fact far more powerful than what we've described so far. There are several ways to define a provider: you can use plain values, classes, and either asynchronous or synchronous factories. More examples are provided here.

Optional providers

Occasionally, you might have dependencies which do not necessarily have to be resolved. For instance, your class may depend on a **configuration object**, but if none is passed, the default values should be used. In such a case, the dependency becomes optional, because lack of the configuration provider wouldn't lead to errors.

To indicate a provider is optional, use the `@Optional()` decorator in the constructor's signature.

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

@Injectable()
export class HttpService<T> {
  constructor(@Optional() @Inject('HTTP_OPTIONS') private httpClient: T)
{}
}
```

Note that in the example above we are using a custom provider, which is the reason we include the `HTTP_OPTIONS` custom **token**. Previous examples showed constructor-based injection indicating a dependency through a class in the constructor. Read more about custom providers and their associated tokens here.

Property-based injection

The technique we've used so far is called constructor-based injection, as providers are injected via the constructor method. In some very specific cases, **property-based injection** might be useful. For instance, if your top-level class depends on either one or multiple providers, passing them all the way up by calling `super()` in sub-classes from the constructor can be very tedious. In order to avoid this, you can use the `@Inject()` decorator at the property level.

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

@Injectable()
export class HttpService<T> {
  @Inject('HTTP_OPTIONS')
  private readonly httpClient: T;
}
```

> warning **Warning** If your class doesn't extend another class, you should always prefer using **constructor-based** injection.

Provider registration

Now that we have defined a provider (`CatsService`), and we have a consumer of that service (`CatsController`), we need to register the service with Nest so that it can perform the injection. We do this by editing our module file (`app.module.ts`) and adding the service to the `providers` array of the `@Module()` decorator.

```
@@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 {}
```

Nest will now be able to resolve the dependencies of the `CatsController` class.

This is how our directory structure should look now:

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

Manual instantiation

Thus far, we've discussed how Nest automatically handles most of the details of resolving dependencies. In certain circumstances, you may need to step outside of the built-in Dependency Injection system and manually retrieve or instantiate providers. We briefly discuss two such topics below.

To get existing instances, or instantiate providers dynamically, you can use Module reference.

To get providers within the `bootstrap()` function (for example for standalone applications without controllers, or to utilize a configuration service during bootstrapping) see Standalone applications.