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 {}
```