Introduction

Nest (NestJS) is a framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript, is built with and fully supports TypeScript (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).

Under the hood, Nest makes use of robust HTTP Server frameworks like Express (the default) and optionally can be configured to use Fastify as well!

Nest provides a level of abstraction above these common Node.js frameworks (Express/Fastify), but also exposes their APIs directly to the developer. This gives developers the freedom to use the myriad of third-party modules which are available for the underlying platform.

Philosophy

In recent years, thanks to Node.js, JavaScript has become the “lingua franca” of the web for both front and backend applications. This has given rise to awesome projects like Angular, React and Vue, which improve developer productivity and enable the creation of fast, testable, and extensible frontend applications. However, while plenty of superb libraries, helpers, and tools exist for Node (and server-side JavaScript), none of them effectively solve the main problem of - **Architecture**.

Nest provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled, and easily maintainable applications. The architecture is heavily inspired by Angular.

Installation

To get started, you can either scaffold the project with the Nest CLI, or clone a starter project (both will produce the same outcome).

To scaffold the project with the Nest CLI, run the following commands. This will create a new project directory, and populate the directory with the initial core Nest files and supporting modules, creating a conventional base structure for your project. Creating a new project with the **Nest CLI** is recommended for first-time users. We'll continue with this approach in First Steps.

```
$ npm i -g @nestjs/cli
$ nest new project-name
```

> info **Hint** To create a new TypeScript project with stricter feature set, pass the `--strict` flag to the `nest new` command.

Alternatives

Alternatively, to install the TypeScript starter project with **Git**:

```bash
$ git clone https://github.com/nestjs/typescript-starter.git project
$ cd project
$ npm install
$ npm run start
```

> info **Hint** If you'd like to clone the repository without the git history, you can use degit.

Open your browser and navigate to `http://localhost:3000/`.

To install the JavaScript flavor of the starter project, use `javascript-starter.git` in the command sequence above.

You can also manually create a new project from scratch by installing the core and supporting files with **npm** (or **yarn**). In this case, of course, you'll be responsible for creating the project boilerplate files yourself.

```bash
$ npm i --save @nestjs/core @nestjs/common rxjs reflect-metadata
```

First steps

In this set of articles, you'll learn the **core fundamentals** of Nest. To get familiar with the essential building blocks of Nest applications, we'll build a basic CRUD application with features that cover a lot of ground at an introductory level.

Language

We're in love with TypeScript, but above all - we love Node.js. That's why Nest is compatible with both TypeScript and **pure JavaScript**. Nest takes advantage of the latest language features, so to use it with vanilla JavaScript we need a Babel compiler.

We'll mostly use TypeScript in the examples we provide, but you can always **switch the code snippets** to vanilla JavaScript syntax (simply click to toggle the language button in the upper right hand corner of each snippet).

Prerequisites

Please make sure that Node.js (version >= 16) is installed on your operating system.

Setup

Setting up a new project is quite simple with the Nest CLI. With npm installed, you can create a new Nest project with the following commands in your OS terminal:

```
$ npm i -g @nestjs/cli
$ nest new project-name
```

> info **Hint** To create a new project with TypeScript's stricter feature set, pass the `--strict` flag to the `nest new` command.

The `project-name` directory will be created, node modules and a few other boilerplate files will be installed, and a `src/` directory will be created and populated with several core files.

src
app.controller.spec.ts
app.controller.ts
app.module.ts
app.service.ts
main.ts

Here's a brief overview of those core files:

`app.controller.ts`	A basic controller with a single route.
`app.controller.spec.ts`	The unit tests for the controller.

`app.module.ts`	The root module of the application.
`app.service.ts`	A basic service with a single method.
`main.ts`	The entry file of the application which uses the core function `NestFactory` to create a Nest application instance.

The `main.ts` includes an async function, which will **bootstrap** our application:

```
@@filename(main)

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

To create a Nest application instance, we use the core `NestFactory` class. `NestFactory` exposes a few static methods that allow creating an application instance. The `create()` method returns an application object, which fulfills the `INestApplication` interface. This object provides a set of methods which are described in the coming chapters. In the `main.ts` example above, we simply start up our HTTP listener, which lets the application await inbound HTTP requests.

Note that a project scaffolded with the Nest CLI creates an initial project structure that encourages developers to follow the convention of keeping each module in its own dedicated directory.

> info **Hint** By default, if any error happens while creating the application your app will exit with the code `1`. If you want to make it throw an error instead disable the option `abortOnError` (e.g., `NestFactory.create(AppModule, {{ '{' }} abortOnError: false {{ '}' }})`).

Platform

Nest aims to be a platform-agnostic framework. Platform independence makes it possible to create reusable logical parts that developers can take advantage of across several different types of applications. Technically, Nest is able to work with any Node HTTP framework once an adapter is created. There are two HTTP platforms supported out-of-the-box: express and fastify. You can choose the one that best suits your needs.

`platform-express`	Express is a well-known minimalist web framework for node. It's a battle tested, production-ready library with lots of resources implemented by the community. The `@nestjs/platform-express` package is used by default. Many users are well served with Express, and need take no action to enable it.
`platform-fastify`	Fastify is a high performance and low overhead framework highly focused on providing maximum efficiency and speed. Read how to use it here.

Whichever platform is used, it exposes its own application interface. These are seen respectively as `NestExpressApplication` and `NestFastifyApplication`.

When you pass a type to the `NestFactory.create()` method, as in the example below, the `app` object will have methods available exclusively for that specific platform. Note, however, you don't **need** to specify a type **unless** you actually want to access the underlying platform API.

```
const app = await NestFactory.create<NestExpressApplication>(AppModule);
```

Running the application

Once the installation process is complete, you can run the following command at your OS command prompt to start the application listening for inbound HTTP requests:

```
$ npm run start
```

> info **Hint** To speed up the development process (x20 times faster builds), you can use the SWC builder by passing the `-b swc` flag to the `start` script, as follows `npm run start -- -b swc`.

This command starts the app with the HTTP server listening on the port defined in the `src/main.ts` file. Once the application is running, open your browser and navigate to `http://localhost:3000/`. You should see the `Hello World!` message.

To watch for changes in your files, you can run the following command to start the application:

```
$ npm run start:dev
```

This command will watch your files, automatically recompiling and reloading the server.

Linting and formatting

CLI provides best effort to scaffold a reliable development workflow at scale. Thus, a generated Nest project comes with both a code **linter** and **formatter** preinstalled (respectively eslint and prettier).

> info **Hint** Not sure about the role of formatters vs linters? Learn the difference here.

To ensure maximum stability and extensibility, we use the base `eslint` and `prettier` cli packages. This setup allows neat IDE integration with official extensions by design.

For headless environments where an IDE is not relevant (Continuous Integration, Git hooks, etc.) a Nest project comes with ready-to-use `npm` scripts.

```
# Lint and autofix with eslint
$ npm run lint

# Format with prettier
$ npm run format
```

Controllers

Controllers are responsible for handling incoming **requests** and returning **responses** to the client.

A controller's purpose is to receive specific requests for the application. The **routing** mechanism controls which controller receives which requests. Frequently, each controller has more than one route, and different routes can perform different actions.

In order to create a basic controller, we use classes and **decorators**. Decorators associate classes with required metadata and enable Nest to create a routing map (tie requests to the corresponding controllers).

> info **Hint** For quickly creating a CRUD controller with the validation built-in, you may use the CLI's CRUD generator: `nest g resource [name]`.

Routing

In the following example we'll use the `@Controller()` decorator, which is **required** to define a basic controller. We'll specify an optional route path prefix of `cats`. Using a path prefix in a `@Controller()` decorator allows us to easily group a set of related routes, and minimize repetitive code. For example, we may choose to group a set of routes that manage interactions with a cat entity under the route `/cats`. In that case, we could specify the path prefix `cats` in the `@Controller()` decorator so that we don't have to repeat that portion of the path for each route in the file.

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

@Controller('cats')
export class CatsController {
  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
@@switch
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll() {
    return 'This action returns all cats';
  }
}
```

> info **Hint** To create a controller using the CLI, simply execute the `$ nest g controller [name]` command.

The `@Get()` HTTP request method decorator before the `findAll()` method tells Nest to create a handler for a specific endpoint for HTTP requests. The endpoint corresponds to the HTTP request method (GET in this case) and the route path. What is the route path? The route path for a handler is determined by concatenating the (optional) prefix declared for the controller, and any path specified in the method's decorator. Since we've declared a prefix for every route (`cats`), and haven't added any path information in the decorator, Nest will map `GET /cats` requests to this handler. As mentioned, the path includes both the optional controller path prefix **and** any path string declared in the request method decorator. For example, a path prefix of `cats` combined with the decorator `@Get('breed')` would produce a route mapping for requests like `GET /cats/breed`.

In our example above, when a GET request is made to this endpoint, Nest routes the request to our user-defined `findAll()` method. Note that the method name we choose here is completely arbitrary. We obviously must declare a method to bind the route to, but Nest doesn't attach any significance to the method name chosen.

This method will return a 200 status code and the associated response, which in this case is just a string. Why does that happen? To explain, we'll first introduce the concept that Nest employs two **different** options for manipulating responses:

Standard (recommended)	Using this built-in method, when a request handler returns a JavaScript object or array, it will **automatically** be serialized to JSON. When it returns a JavaScript primitive type (e.g., `string`, `number`, `boolean`), however, Nest will send just the value without attempting to serialize it. This makes response handling simple: just return the value, and Nest takes care of the rest. Furthermore, the response's **status code** is always 200 by default, except for POST requests which use 201. We can easily change this behavior by adding the `@HttpCode(...)` decorator at a handler-level (see Status codes).
Library-specific	We can use the library-specific (e.g., Express) response object, which can be injected using the `@Res()` decorator in the method handler signature (e.g., `findAll(@Res() response)`). With this approach, you have the ability to use the native response handling methods exposed by that object. For example, with Express, you can construct responses using code like `response.status(200).send()`.

> warning **Warning** Nest detects when the handler is using either `@Res()` or `@Next()`, indicating you have chosen the library-specific option. If both approaches are used at the same time, the Standard approach is **automatically disabled** for this single route and will no longer work as expected. To use both approaches at the same time (for example, by injecting the response object to only set cookies/headers but still leave the rest to the framework), you must set the `passthrough` option to `true` in the `@Res({{ '{' }} passthrough: true {{ '}' }})` decorator.

Request object

Handlers often need access to the client **request** details. Nest provides access to the request object of the underlying platform (Express by default). We can access the request object by instructing Nest to inject it by adding the `@Req()` decorator to the handler's signature.

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

@Controller('cats')
export class CatsController {
  @Get()
  findAll(@Req() request: Request): string {
    return 'This action returns all cats';
  }
}
@@switch
import { Controller, Bind, Get, Req } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  @Bind(Req())
  findAll(request) {
    return 'This action returns all cats';
  }
}
```

> info **Hint** In order to take advantage of `express` typings (as in the `request: Request` parameter example above), install `@types/express` package.

The request object represents the HTTP request and has properties for the request query string, parameters, HTTP headers, and body (read more here). In most cases, it's not necessary to grab these properties manually. We can use dedicated decorators instead, such as `@Body()` or `@Query()`, which are available out of the box. Below is a list of the provided decorators and the plain platform-specific objects they represent.

`@Request(), @Req()`	`req`
`@Response(), @Res()`*	`res`
`@Next()`	`next`
`@Session()`	`req.session`
`@Param(key?: string)`	`req.params` / `req.params[key]`
`@Body(key?: string)`	`req.body` / `req.body[key]`
`@Query(key?: string)`	`req.query` / `req.query[key]`
`@Headers(name?: string)`	`req.headers` / `req.headers[name]`
`@Ip()`	`req.ip`
`@HostParam()`	`req.hosts`

* For compatibility with typings across underlying HTTP platforms (e.g., Express and Fastify), Nest provides `@Res()` and `@Response()` decorators. `@Res()` is simply an alias for `@Response()`. Both directly expose the underlying native platform `response` object interface. When using them, you should also import the typings for the underlying library (e.g., `@types/express`) to take full advantage. Note that when you inject either `@Res()` or `@Response()` in a method handler, you put Nest into **Library-specific mode** for that handler, and you become responsible for managing the response. When doing so, you must issue some kind of response by making a call on the `response` object (e.g., `res.json(...)` or `res.send(...)`), or the HTTP server will hang.

> info **Hint** To learn how to create your own custom decorators, visit this chapter.

Resources

Earlier, we defined an endpoint to fetch the cats resource (**GET** route). We'll typically also want to provide an endpoint that creates new records. For this, let's create the **POST** handler:

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

@Controller('cats')
export class CatsController {
  @Post()
  create(): string {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
@@switch
import { Controller, Get, Post } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  create() {
    return 'This action adds a new cat';
  }

  @Get()
  findAll() {
    return 'This action returns all cats';
  }
}
```

It's that simple. Nest provides decorators for all of the standard HTTP methods: `@Get()`, `@Post()`, `@Put()`, `@Delete()`, `@Patch()`, `@Options()`, and `@Head()`. In addition, `@All()` defines an endpoint that handles all of them.

Route wildcards

Pattern based routes are supported as well. For instance, the asterisk is used as a wildcard, and will match any combination of characters.

```
@Get('ab*cd')
findAll() {
  return 'This route uses a wildcard';
}
```

The `'ab*cd'` route path will match `abcd`, `ab_cd`, `abecd`, and so on. The characters `?`, `+`, `*`, and `()` may be used in a route path, and are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

> warning **Warning** A wildcard in the middle of the route is only supported by express.

Status code

As mentioned, the response **status code** is always **200** by default, except for POST requests which are **201**. We can easily change this behavior by adding the `@HttpCode(...)` decorator at a handler level.

```
@Post()
@HttpCode(204)
create() {
  return 'This action adds a new cat';
}
```

> info **Hint** Import `HttpCode` from the `@nestjs/common` package.

Often, your status code isn't static but depends on various factors. In that case, you can use a library-specific **response** (inject using `@Res()`) object (or, in case of an error, throw an exception).

Headers

To specify a custom response header, you can either use a `@Header()` decorator or a library-specific response object (and call `res.header()` directly).

```
@Post()
@Header('Cache-Control', 'none')
create() {
  return 'This action adds a new cat';
}
```

> info **Hint** Import `Header` from the `@nestjs/common` package.

Redirection

To redirect a response to a specific URL, you can either use a `@Redirect()` decorator or a library-specific response object (and call `res.redirect()` directly).

`@Redirect()` takes two arguments, `url` and `statusCode`, both are optional. The default value of `statusCode` is `302` (`Found`) if omitted.

```
@Get()
@Redirect('https://nestjs.com', 301)
```

Sometimes you may want to determine the HTTP status code or the redirect URL dynamically. Do this by returning an object from the route handler method with the shape:

```
{
  "url": string,
  "statusCode": number
}
```

Returned values will override any arguments passed to the `@Redirect()` decorator. For example:

```
@Get('docs')
@Redirect('https://docs.nestjs.com', 302)
getDocs(@Query('version') version) {
  if (version && version === '5') {
    return { url: 'https://docs.nestjs.com/v5/' };
  }
}
```

Route parameters

Routes with static paths won't work when you need to accept **dynamic data** as part of the request (e.g., `GET /cats/1` to get cat with id `1`). In order to define routes with parameters, we can add route parameter **tokens** in the path of the route to capture the dynamic value at that position in the request URL. The route parameter token in the `@Get()` decorator example below demonstrates this usage. Route parameters declared in this way can be accessed using the `@Param()` decorator, which should be added to the method signature.

> info **Hint** Routes with parameters should be declared after any static paths. This prevents the parameterized paths from intercepting traffic destined for the static paths.

```
@@filename()
@Get(':id')
findOne(@Param() params: any): string {
  console.log(params.id);
  return `This action returns a #${params.id} cat`;
}
@@switch
@Get(':id')
@Bind(Param())
findOne(params) {
  console.log(params.id);
  return `This action returns a #${params.id} cat`;
}
```

`@Param()` is used to decorate a method parameter (`params` in the example above), and makes the **route** parameters available as properties of that decorated method parameter inside the body of the method. As seen in the code above, we can access the `id` parameter by referencing `params.id`. You can also pass in a particular parameter token to the decorator, and then reference the route parameter directly by name in the method body.

> info **Hint** Import `Param` from the `@nestjs/common` package.

```
@@filename()
@Get(':id')
findOne(@Param('id') id: string): string {
  return `This action returns a #${id} cat`;
}
@@switch
@Get(':id')
@Bind(Param('id'))
findOne(id) {
  return `This action returns a #${id} cat`;
}
```

Sub-Domain Routing

The `@Controller` decorator can take a `host` option to require that the HTTP host of the incoming requests matches some specific value.

```
@Controller({ host: 'admin.example.com' })
export class AdminController {
  @Get()
  index(): string {
    return 'Admin page';
  }
}
```

> **Warning** Since **Fastify** lacks support for nested routers, when using sub-domain routing, the (default) Express adapter should be used instead.

Similar to a route `path`, the `hosts` option can use tokens to capture the dynamic value at that position in the host name. The host parameter token in the `@Controller()` decorator example below demonstrates this usage. Host parameters declared in this way can be accessed using the `@HostParam()` decorator, which should be added to the method signature.

```
@Controller({ host: ':account.example.com' })
export class AccountController {
  @Get()
  getInfo(@HostParam('account') account: string) {
    return account;
  }
}
```

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 of the controller may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking or multi-tenancy. Learn how to control scopes here.

Asynchronicity

We love modern JavaScript and we know that data extraction is mostly **asynchronous**. That's why Nest supports and works well with `async` functions.

> info **Hint** Learn more about `async / await` feature here

Every async function has to return a `Promise`. This means that you can return a deferred value that Nest will be able to resolve by itself. Let's see an example of this:

```
@@filename(cats.controller)
@Get()
async findAll(): Promise<any[]> {
  return [];
}
@@switch
@Get()
async findAll() {
  return [];
}
```

The above code is fully valid. Furthermore, Nest route handlers are even more powerful by being able to return RxJS observable streams. Nest will automatically subscribe to the source underneath and take the last emitted value (once the stream is completed).

```
@@filename(cats.controller)
@Get()
findAll(): Observable<any[]> {
  return of([]);
}
@@switch
@Get()
findAll() {
  return of([]);
}
```

Both of the above approaches work and you can use whatever fits your requirements.

Request payloads

Our previous example of the POST route handler didn't accept any client params. Let's fix this by adding the `@Body()` decorator here.

But first (if you use TypeScript), we need to determine the **DTO** (Data Transfer Object) schema. A DTO is an object that defines how the data will be sent over the network. We could determine the DTO schema by using **TypeScript** interfaces, or by simple classes. Interestingly, we recommend using **classes** here. Why? Classes are part of the JavaScript ES6 standard, and therefore they are preserved as real entities in the compiled JavaScript. On the other hand, since TypeScript interfaces are removed during the transpilation, Nest can't refer to them at runtime. This is important because features such as **Pipes** enable additional possibilities when they have access to the metatype of the variable at runtime.

Let's create the `CreateCatDto` class:

```
@@filename(create-cat.dto)
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

It has only three basic properties. Thereafter we can use the newly created DTO inside the `CatsController`:

```
@@filename(cats.controller)
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  return 'This action adds a new cat';
}
@@switch
@Post()
@Bind(Body())
async create(createCatDto) {
  return 'This action adds a new cat';
}
```

> info **Hint** Our `ValidationPipe` can filter out properties that should not be received by the method handler. In this case, we can whitelist the acceptable properties, and any property not included in the whitelist is automatically stripped from the resulting object. In the `CreateCatDto` example, our whitelist is the `name`, `age`, and `breed` properties. Learn more here.

Handling errors

There's a separate chapter about handling errors (i.e., working with exceptions) here.

Full resource sample

Below is an example that makes use of several of the available decorators to create a basic controller. This controller exposes a couple of methods to access and manipulate internal data.

```
@@filename(cats.controller)
import { Controller, Get, Query, Post, Body, Put, Param, Delete } from
'@nestjs/common';
import { CreateCatDto, UpdateCatDto, ListAllEntities } from './dto';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Body() createCatDto: CreateCatDto) {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(@Query() query: ListAllEntities) {
    return `This action returns all cats (limit: ${query.limit} items)`;
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return `This action returns a #${id} cat`;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateCatDto: UpdateCatDto) {
    return `This action updates a #${id} cat`;
  }
  @Delete(':id')
  remove(@Param('id') id: string) {
    return `This action removes a #${id} cat`;
  }
}
@@switch
import { Controller, Get, Query, Post, Body, Put, Param, Delete, Bind }
from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  @Bind(Body())
  create(createCatDto) {
    return 'This action adds a new cat';
  }

  @Get()
  @Bind(Query())
  findAll(query) {
    console.log(query);
    return `This action returns all cats (limit: ${query.limit} items)`;
  }

  @Get(':id')
  @Bind(Param('id'))
  findOne(id) {
    return `This action returns a #${id} cat`;
  }

  @Put(':id')
  @Bind(Param('id'), Body())
  update(id, updateCatDto) {
    return `This action updates a #${id} cat`;
  }

  @Delete(':id')
  @Bind(Param('id'))
  remove(id) {
    return `This action removes a #${id} cat`;
  }
}
```

> info **Hint** Nest CLI provides a generator (schematic) that automatically generates **all the boilerplate code** to help us avoid doing all of this, and make the developer experience much simpler. Read more about this feature here.

Getting up and running

With the above controller fully defined, Nest still doesn't know that `CatsController` exists and as a result won't create an instance of this class.

Controllers always belong to a module, which is why we include the `controllers` array within the `@Module()` decorator. Since we haven't yet defined any other modules except the root `AppModule`, we'll use that to introduce the `CatsController`:

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

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

We attached the metadata to the module class using the `@Module()` decorator, and Nest can now easily reflect which controllers have to be mounted.

Library-specific approach

So far we've discussed the Nest standard way of manipulating responses. The second way of manipulating the response is to use a library-specific response object. In order to inject a particular response object, we need to use the `@Res()` decorator. To show the differences, let's rewrite the `CatsController` to the following:

```
@@filename()
import { Controller, Get, Post, Res, HttpStatus } from '@nestjs/common';
import { Response } from 'express';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Res() res: Response) {
    res.status(HttpStatus.CREATED).send();
  }

  @Get()
  findAll(@Res() res: Response) {
     res.status(HttpStatus.OK).json([]);
  }
}
@@switch
import { Controller, Get, Post, Bind, Res, Body, HttpStatus } from
'@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  @Bind(Res(), Body())
  create(res, createCatDto) {
    res.status(HttpStatus.CREATED).send();
  }

  @Get()
  @Bind(Res())
  findAll(res) {
     res.status(HttpStatus.OK).json([]);
  }
}
```

Though this approach works, and does in fact allow for more flexibility in some ways by providing full control of the response object (headers manipulation, library-specific features, and so on), it should be used with care. In general, the approach is much less clear and does have some disadvantages. The main disadvantage is that your code becomes platform-dependent (as underlying libraries may have different APIs on the response object), and harder to test (you'll have to mock the response object, etc.).

Also, in the example above, you lose compatibility with Nest features that depend on Nest standard response handling, such as Interceptors and `@HttpCode()` / `@Header()` decorators. To fix this, you can set the `passthrough` option to `true`, as follows:

```
@@filename()
@Get()
findAll(@Res({ passthrough: true }) res: Response) {
  res.status(HttpStatus.OK);
  return [];
}
@@switch
@Get()
@Bind(Res({ passthrough: true }))
findAll(res) {
  res.status(HttpStatus.OK);
  return [];
}
```

Now you can interact with the native response object (for example, set cookies or headers depending on certain conditions), but leave the rest to the framework.

Modules

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

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

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

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

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

Feature modules

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

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

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

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

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

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

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

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

Here is how our directory structure looks now:

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

Shared modules

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

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

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

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

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

Module re-exporting

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

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

Dependency injection

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

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

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {
  constructor(private catsService: CatsService) {}
}
@@switch
import { Module, Dependencies } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

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

However, module classes themselves cannot be injected as providers due to circular dependency .

Global modules

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

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

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

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

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

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

Dynamic modules

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

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

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

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options?): DynamicModule {
    const providers = createDatabaseProviders(options, entities);
    return {
      module: DatabaseModule,
      providers: providers,
      exports: providers,
    };
  }
}
@@switch
import { Module } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

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

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

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

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

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

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

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

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

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

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

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

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

The Dynamic modules chapter covers this topic in greater detail, and includes a working example.

> info **Hint** Learn how to build highly customizable dynamic modules with the use of `ConfigurableModuleBuilder` here in this chapter.

Middleware

Middleware is a function which is called **before** the route handler. Middleware functions have access to the request and response objects, and the `next()` middleware function in the application's request-response cycle. The **next** middleware function is commonly denoted by a variable named `next`.

Nest middleware are, by default, equivalent to express middleware. The following description from the official express documentation describes the capabilities of middleware:

> Middleware functions can perform the following tasks:
>
> - execute any code.
> - make changes to the request and the response objects.
> - end the request-response cycle.
> - call the next middleware function in the stack.
> - if the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

You implement custom Nest middleware in either a function, or in a class with an `@Injectable()` decorator. The class should implement the `NestMiddleware` interface, while the function does not have any special requirements. Let's start by implementing a simple middleware feature using the class method.

> warning **Warning** `Express` and `fastify` handle middleware differently and provide different method signatures, read more here.

```
@@filename(logger.middleware)
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('Request...');
    next();
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class LoggerMiddleware {
  use(req, res, next) {
    console.log('Request...');
    next();
  }
}
```

Dependency injection

Nest middleware fully supports Dependency Injection. Just as with providers and controllers, they are able to **inject dependencies** that are available within the same module. As usual, this is done through the `constructor`.

Applying middleware

There is no place for middleware in the `@Module()` decorator. Instead, we set them up using the `configure()` method of the module class. Modules that include middleware have to implement the `NestModule` interface. Let's set up the `LoggerMiddleware` at the `AppModule` level.

```
@@filename(app.module)
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
@@switch
import { Module } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
```

In the above example we have set up the `LoggerMiddleware` for the `/cats` route handlers that were previously defined inside the `CatsController`. We may also further restrict a middleware to a particular request method by passing an object containing the route `path` and request `method` to the `forRoutes()` method when configuring the middleware. In the example below, notice that we import the `RequestMethod` enum to reference the desired request method type.

```
@@filename(app.module)
import { Module, NestModule, RequestMethod, MiddlewareConsumer } from
'@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET });
  }
}
@@switch
import { Module, RequestMethod } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET });
  }
}
```

> info **Hint** The `configure()` method can be made asynchronous using `async/await` (e.g., you can `await` completion of an asynchronous operation inside the `configure()` method body).

> warning **Warning** When using the `express` adapter, the NestJS app will register `json` and `urlencoded` from the package `body-parser` by default. This means if you want to customize that middleware via the `MiddlewareConsumer`, you need to turn off the global middleware by setting the `bodyParser` flag to `false` when creating the application with `NestFactory.create()`.

Route wildcards

Pattern based routes are supported as well. For instance, the asterisk is used as a **wildcard**, and will match any combination of characters:

```
forRoutes({ path: 'ab*cd', method: RequestMethod.ALL });
```

The `'ab*cd'` route path will match `abcd`, `ab_cd`, `abecd`, and so on. The characters `?`, `+`, `*`, and `()` may be used in a route path, and are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

> warning **Warning** The `fastify` package uses the latest version of the `path-to-regexp` package, which no longer supports wildcard asterisks `*`. Instead, you must use parameters (e.g., `(.*)`, `:splat*`).

Middleware consumer

The `MiddlewareConsumer` is a helper class. It provides several built-in methods to manage middleware. All of them can be simply **chained** in the fluent style. The `forRoutes()` method can take a single string, multiple strings, a `RouteInfo` object, a controller class and even multiple controller classes. In most cases you'll probably just pass a list of **controllers** separated by commas. Below is an example with a single controller:

```
@@filename(app.module)
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
@@switch
import { Module } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
```

> info **Hint** The `apply()` method may either take a single middleware, or multiple arguments to specify multiple middlewares.

Excluding routes

At times we want to **exclude** certain routes from having the middleware applied. We can easily exclude certain routes with the `exclude()` method. This method can take a single string, multiple strings, or a `RouteInfo` object identifying routes to be excluded, as shown below:

```
consumer
  .apply(LoggerMiddleware)
  .exclude(
    { path: 'cats', method: RequestMethod.GET },
    { path: 'cats', method: RequestMethod.POST },
    'cats/(.*)',
  )
  .forRoutes(CatsController);
```

> info **Hint** The `exclude()` method supports wildcard parameters using the path-to-regexp package.

With the example above, `LoggerMiddleware` will be bound to all routes defined inside `CatsController` **except** the three passed to the `exclude()` method.

Functional middleware

The `LoggerMiddleware` class we've been using is quite simple. It has no members, no additional methods, and no dependencies. Why can't we just define it in a simple function instead of a class? In fact, we can. This type of middleware is called **functional middleware**. Let's transform the logger middleware from class-based into functional middleware to illustrate the difference:

```
@@filename(logger.middleware)
import { Request, Response, NextFunction } from 'express';

export function logger(req: Request, res: Response, next: NextFunction) {
  console.log(`Request...`);
  next();
};
@@switch
export function logger(req, res, next) {
  console.log(`Request...`);
  next();
};
```

And use it within the `AppModule`:

```
@@filename(app.module)
consumer
```

```
    .apply(logger)
    .forRoutes(CatsController);
```

> info **Hint** Consider using the simpler **functional middleware** alternative any time your middleware doesn't need any dependencies.

Multiple middleware

As mentioned above, in order to bind multiple middleware that are executed sequentially, simply provide a comma separated list inside the `apply()` method:

```
consumer.apply(cors(), helmet(), logger).forRoutes(CatsController);
```

Global middleware

If we want to bind middleware to every registered route at once, we can use the `use()` method that is supplied by the `INestApplication` instance:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.use(logger);
await app.listen(3000);
```

> info **Hint** Accessing the DI container in a global middleware is not possible. You can use a functional middleware instead when using `app.use()`. Alternatively, you can use a class middleware and consume it with `.forRoutes('*')` within the `AppModule` (or any other module).

Exception filters

Nest comes with a built-in **exceptions layer** which is responsible for processing all unhandled exceptions across an application. When an exception is not handled by your application code, it is caught by this layer, which then automatically sends an appropriate user-friendly response.

Out of the box, this action is performed by a built-in **global exception filter**, which handles exceptions of type `HttpException` (and subclasses of it). When an exception is **unrecognized** (is neither `HttpException` nor a class that inherits from `HttpException`), the built-in exception filter generates the following default JSON response:

```
{
  "statusCode": 500,
  "message": "Internal server error"
}
```

> info **Hint** The global exception filter partially supports the `http-errors` library. Basically, any thrown exception containing the `statusCode` and `message` properties will be properly populated and sent back as a response (instead of the default `InternalServerErrorException` for unrecognized exceptions).

Throwing standard exceptions

Nest provides a built-in `HttpException` class, exposed from the `@nestjs/common` package. For typical HTTP REST/GraphQL API based applications, it's best practice to send standard HTTP response objects when certain error conditions occur.

For example, in the `CatsController`, we have a `findAll()` method (a `GET` route handler). Let's assume that this route handler throws an exception for some reason. To demonstrate this, we'll hard-code it as follows:

```
@@filename(cats.controller)
@Get()
async findAll() {
  throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
}
```

> info **Hint** We used the `HttpStatus` here. This is a helper enum imported from the `@nestjs/common` package.

When the client calls this endpoint, the response looks like this:

```
{
  "statusCode": 403,
  "message": "Forbidden"
}
```

The `HttpException` constructor takes two required arguments which determine the response:

- The `response` argument defines the JSON response body. It can be a `string` or an `object` as described below.
- The `status` argument defines the HTTP status code.

By default, the JSON response body contains two properties:

- `statusCode`: defaults to the HTTP status code provided in the `status` argument
- `message`: a short description of the HTTP error based on the `status`

To override just the message portion of the JSON response body, supply a string in the `response` argument. To override the entire JSON response body, pass an object in the `response` argument. Nest will serialize the object and return it as the JSON response body.

The second constructor argument - `status` - should be a valid HTTP status code. Best practice is to use the `HttpStatus` enum imported from `@nestjs/common`.

There is a **third** constructor argument (optional) - `options` - that can be used to provide an error cause. This `cause` object is not serialized into the response object, but it can be useful for logging purposes, providing valuable information about the inner error that caused the `HttpException` to be thrown.

Here's an example overriding the entire response body and providing an error cause:

```
@@filename(cats.controller)
@Get()
async findAll() {
  try {
    await this.service.findAll()
  } catch (error) {
    throw new HttpException({
      status: HttpStatus.FORBIDDEN,
      error: 'This is a custom message',
    }, HttpStatus.FORBIDDEN, {
      cause: error
    });
  }
}
```

Using the above, this is how the response would look:

```
{
  "status": 403,
  "error": "This is a custom message"
}
```

Custom exceptions

In many cases, you will not need to write custom exceptions, and can use the built-in Nest HTTP exception, as described in the next section. If you do need to create customized exceptions, it's good practice to create your own **exceptions hierarchy**, where your custom exceptions inherit from the base `HttpException` class. With this approach, Nest will recognize your exceptions, and automatically take care of the error responses. Let's implement such a custom exception:

```
@@filename(forbidden.exception)
export class ForbiddenException extends HttpException {
  constructor() {
    super('Forbidden', HttpStatus.FORBIDDEN);
  }
}
```

Since `ForbiddenException` extends the base `HttpException`, it will work seamlessly with the built-in exception handler, and therefore we can use it inside the `findAll()` method.

```
@@filename(cats.controller)
@Get()
async findAll() {
  throw new ForbiddenException();
}
```

Built-in HTTP exceptions

Nest provides a set of standard exceptions that inherit from the base `HttpException`. These are exposed from the `@nestjs/common` package, and represent many of the most common HTTP exceptions:

- `BadRequestException`
- `UnauthorizedException`
- `NotFoundException`
- `ForbiddenException`
- `NotAcceptableException`
- `RequestTimeoutException`
- `ConflictException`
- `GoneException`
- `HttpVersionNotSupportedException`
- `PayloadTooLargeException`
- `UnsupportedMediaTypeException`
- `UnprocessableEntityException`
- `InternalServerErrorException`

- `NotImplementedException`
- `ImATeapotException`
- `MethodNotAllowedException`
- `BadGatewayException`
- `ServiceUnavailableException`
- `GatewayTimeoutException`
- `PreconditionFailedException`

All the built-in exceptions can also provide both an error `cause` and an error description using the `options` parameter:

```
throw new BadRequestException('Something bad happened', { cause: new
Error(), description: 'Some error description' })
```

Using the above, this is how the response would look:

```
{
  "message": "Something bad happened",
  "error": "Some error description",
  "statusCode": 400,
}
```

Exception filters

While the base (built-in) exception filter can automatically handle many cases for you, you may want **full control** over the exceptions layer. For example, you may want to add logging or use a different JSON schema based on some dynamic factors. **Exception filters** are designed for exactly this purpose. They let you control the exact flow of control and the content of the response sent back to the client.

Let's create an exception filter that is responsible for catching exceptions which are an instance of the `HttpException` class, and implementing custom response logic for them. To do this, we'll need to access the underlying platform `Request` and `Response` objects. We'll access the `Request` object so we can pull out the original `url` and include that in the logging information. We'll use the `Response` object to take direct control of the response that is sent, using the `response.json()` method.

```
@@filename(http-exception.filter)
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from
'@nestjs/common';
import { Request, Response } from 'express';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const status = exception.getStatus();

    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        path: request.url,
      });
  }
}
@@switch
import { Catch, HttpException } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter {
  catch(exception, host) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const request = ctx.getRequest();
    const status = exception.getStatus();

    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        path: request.url,
      });
  }
}
```

> info **Hint** All exception filters should implement the generic `ExceptionFilter<T>` interface. This requires you to provide the `catch(exception: T, host: ArgumentsHost)` method with its indicated signature. `T` indicates the type of the exception.

> warning **Warning** If you are using `@nestjs/platform-fastify` you can use `response.send()` instead of `response.json()`. Don't forget to import the correct types from `fastify`.

The `@Catch(HttpException)` decorator binds the required metadata to the exception filter, telling Nest that this particular filter is looking for exceptions of type `HttpException` and nothing else. The `@Catch()` decorator may take a single parameter, or a comma-separated list. This lets you set up the filter for several types of exceptions at once.

Arguments host

Let's look at the parameters of the `catch()` method. The `exception` parameter is the exception object currently being processed. The `host` parameter is an `ArgumentsHost` object. `ArgumentsHost` is a powerful utility object that we'll examine further in the execution context chapter*. In this code sample, we use it to obtain a reference to the `Request` and `Response` objects that are being passed to the original request handler (in the controller where the exception originates). In this code sample, we've used some helper methods on `ArgumentsHost` to get the desired `Request` and `Response` objects. Learn more about `ArgumentsHost` here.

*The reason for this level of abstraction is that `ArgumentsHost` functions in all contexts (e.g., the HTTP server context we're working with now, but also Microservices and WebSockets). In the execution context chapter we'll see how we can access the appropriate underlying arguments for **any** execution context with the power of `ArgumentsHost` and its helper functions. This will allow us to write generic exception filters that operate across all contexts.

Binding filters

Let's tie our new `HttpExceptionFilter` to the `CatsController`'s `create()` method.

```
@@filename(cats.controller)
@Post()
@UseFilters(new HttpExceptionFilter())
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
@@switch
@Post()
@UseFilters(new HttpExceptionFilter())
@Bind(Body())
async create(createCatDto) {
  throw new ForbiddenException();
}
```

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

We have used the `@UseFilters()` decorator here. Similar to the `@Catch()` decorator, it can take a single filter instance, or a comma-separated list of filter instances. Here, we created the instance of `HttpExceptionFilter` in place. Alternatively, you may pass the class (instead of an instance), leaving responsibility for instantiation to the framework, and enabling **dependency injection**.

```
@@filename(cats.controller)
@Post()
@UseFilters(HttpExceptionFilter)
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
@@switch
@Post()
@UseFilters(HttpExceptionFilter)
@Bind(Body())
async create(createCatDto) {
  throw new ForbiddenException();
}
```

> info **Hint** Prefer applying filters by using classes instead of instances when possible. It reduces **memory usage** since Nest can easily reuse instances of the same class across your entire module.

In the example above, the `HttpExceptionFilter` is applied only to the single `create()` route handler, making it method-scoped. Exception filters can be scoped at different levels: method-scoped of the controller/resolver/gateway, controller-scoped, or global-scoped.
For example, to set up a filter as controller-scoped, you would do the following:

```
@@filename(cats.controller)
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

This construction sets up the `HttpExceptionFilter` for every route handler defined inside the `CatsController`.

To create a global-scoped filter, you would do the following:

```
@@filename(main)
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new HttpExceptionFilter());
  await app.listen(3000);
}
bootstrap();
```

> warning **Warning** The `useGlobalFilters()` method does not set up filters for gateways or hybrid applications.

Global-scoped filters are used across the whole application, for every controller and every route handler. In terms of dependency injection, global filters registered from outside of any module (with `useGlobalFilters()` as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can register a global-scoped filter **directly from any module** using the following construction:

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

@Module({
  providers: [
    {
      provide: APP_FILTER,
      useClass: HttpExceptionFilter,
    },
  ],
})
export class AppModule {}
```

> info **Hint** When using this approach to perform dependency injection for the filter, note that regardless of the module where this construction is employed, the filter is, in fact, global. Where should this be done? Choose the module where the filter (`HttpExceptionFilter` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more here.

You can add as many filters with this technique as needed; simply add each to the providers array.

Catch everything

In order to catch **every** unhandled exception (regardless of the exception type), leave the `@Catch()` decorator's parameter list empty, e.g., `@Catch()`.

In the example below we have a code that is platform-agnostic because it uses the HTTP adapter to deliver the response, and doesn't use any of the platform-specific objects (`Request` and `Response`) directly:

```
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
  HttpStatus,
} from '@nestjs/common';
import { HttpAdapterHost } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  constructor(private readonly httpAdapterHost: HttpAdapterHost) {}

  catch(exception: unknown, host: ArgumentsHost): void {
    // In certain situations `httpAdapter` might not be available in the
    // constructor method, thus we should resolve it here.
    const { httpAdapter } = this.httpAdapterHost;

    const ctx = host.switchToHttp();

    const httpStatus =
      exception instanceof HttpException
        ? exception.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    const responseBody = {
      statusCode: httpStatus,
      timestamp: new Date().toISOString(),
      path: httpAdapter.getRequestUrl(ctx.getRequest()),
    };

    httpAdapter.reply(ctx.getResponse(), responseBody, httpStatus);
  }
}
```

> warning **Warning** When combining an exception filter that catches everything with a filter that is bound to a specific type, the "Catch anything" filter should be declared first to allow the specific filter to correctly handle the bound type.

Inheritance

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the built-in default **global exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend `BaseExceptionFilter` and call the inherited `catch()` method.

```
@@filename(all-exceptions.filter)
import { Catch, ArgumentsHost } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(exception, host);
  }
}
@@switch
import { Catch } from '@nestjs/common';
import { BaseExceptionFilter } from '@nestjs/core';

@Catch()
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception, host) {
    super.catch(exception, host);
  }
}
```

> warning **Warning** Method-scoped and Controller-scoped filters that extend the `BaseExceptionFilter` should not be instantiated with `new`. Instead, let the framework instantiate them automatically.

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business** logic (e.g., handling various conditions).

Global filters **can** extend the base filter. This can be done in either of two ways.

The first method is to inject the `HttpAdapter` reference when instantiating the custom global filter:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const { httpAdapter } = app.get(HttpAdapterHost);
  app.useGlobalFilters(new AllExceptionsFilter(httpAdapter));

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

The second method is to use the `APP_FILTER` token as shown here.

Pipes

A pipe is a class annotated with the `@Injectable()` decorator, which implements the `PipeTransform` interface.

Pipes have two typical use cases:

- **transformation**: transform input data to the desired form (e.g., from string to integer)
- **validation**: evaluate input data and if valid, simply pass it through unchanged; otherwise, throw an exception

In both cases, pipes operate on the `arguments` being processed by a controller route handler. Nest interposes a pipe just before a method is invoked, and the pipe receives the arguments destined for the method and operates on them. Any transformation or validation operation takes place at that time, after which the route handler is invoked with any (potentially) transformed arguments.

Nest comes with a number of built-in pipes that you can use out-of-the-box. You can also build your own custom pipes. In this chapter, we'll introduce the built-in pipes and show how to bind them to route handlers. We'll then examine several custom-built pipes to show how you can build one from scratch.

> info **Hint** Pipes run inside the exceptions zone. This means that when a Pipe throws an exception it is handled by the exceptions layer (global exceptions filter and any exceptions filters that are applied to the current context). Given the above, it should be clear that when an exception is thrown in a Pipe, no controller method is subsequently executed. This gives you a best-practice technique for validating data coming into the application from external sources at the system boundary.

Built-in pipes

Nest comes with nine pipes available out-of-the-box:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseFloatPipe`
- `ParseBoolPipe`
- `ParseArrayPipe`
- `ParseUUIDPipe`
- `ParseEnumPipe`
- `DefaultValuePipe`
- `ParseFilePipe`

They're exported from the `@nestjs/common` package.

Let's take a quick look at using `ParseIntPipe`. This is an example of the **transformation** use case, where the pipe ensures that a method handler parameter is converted to a JavaScript integer (or throws an exception if the conversion fails). Later in this chapter, we'll show a simple custom implementation for a `ParseIntPipe`. The example techniques below also apply to the other built-in transformation pipes

(`ParseBoolPipe`, `ParseFloatPipe`, `ParseEnumPipe`, `ParseArrayPipe` and `ParseUUIDPipe`, which we'll refer to as the `Parse*` pipes in this chapter).

Binding pipes

To use a pipe, we need to bind an instance of the pipe class to the appropriate context. In our `ParseIntPipe` example, we want to associate the pipe with a particular route handler method, and make sure it runs before the method is called. We do so with the following construct, which we'll refer to as binding the pipe at the method parameter level:

```
@Get(':id')
async findOne(@Param('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

This ensures that one of the following two conditions is true: either the parameter we receive in the `findOne()` method is a number (as expected in our call to `this.catsService.findOne()`), or an exception is thrown before the route handler is called.

For example, assume the route is called like:

```
GET localhost:3000/abc
```

Nest will throw an exception like this:

```
{
  "statusCode": 400,
  "message": "Validation failed (numeric string is expected)",
  "error": "Bad Request"
}
```

The exception will prevent the body of the `findOne()` method from executing.

In the example above, we pass a class (`ParseIntPipe`), not an instance, leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes and guards, we can instead pass an in-place instance. Passing an in-place instance is useful if we want to customize the built-in pipe's behavior by passing options:

```
@Get(':id')
async findOne(
  @Param('id', new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.NOT_ACCEPTABLE }))
  id: number,
) {
  return this.catsService.findOne(id);
}
```

Binding the other transformation pipes (all of the **Parse*** pipes) works similarly. These pipes all work in the context of validating route parameters, query string parameters and request body values.

For example with a query string parameter:

```
@Get()
async findOne(@Query('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

Here's an example of using the `ParseUUIDPipe` to parse a string parameter and validate if it is a UUID.

```
@@filename()
@Get(':uuid')
async findOne(@Param('uuid', new ParseUUIDPipe()) uuid: string) {
  return this.catsService.findOne(uuid);
}
@@switch
@Get(':uuid')
@Bind(Param('uuid', new ParseUUIDPipe()))
async findOne(uuid) {
  return this.catsService.findOne(uuid);
}
```

> info **Hint** When using `ParseUUIDPipe()` you are parsing UUID in version 3, 4 or 5, if you only require a specific version of UUID you can pass a version in the pipe options.

Above we've seen examples of binding the various `Parse*` family of built-in pipes. Binding validation pipes is a little bit different; we'll discuss that in the following section.

> info **Hint** Also, see Validation techniques for extensive examples of validation pipes.

Custom pipes

As mentioned, you can build your own custom pipes. While Nest provides a robust built-in `ParseIntPipe` and `ValidationPipe`, let's build simple custom versions of each from scratch to see how custom pipes are constructed.

We start with a simple `ValidationPipe`. Initially, we'll have it simply take an input value and immediately return the same value, behaving like an identity function.

```
@@filename(validation.pipe)
import { PipeTransform, Injectable, ArgumentMetadata } from
```

```
'@nestjs/common';

@Injectable()
export class ValidationPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    return value;
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class ValidationPipe {
  transform(value, metadata) {
    return value;
  }
}
```

> info **Hint** `PipeTransform<T, R>` is a generic interface that must be implemented by any pipe. The generic interface uses `T` to indicate the type of the input `value`, and `R` to indicate the return type of the `transform()` method.

Every pipe must implement the `transform()` method to fulfill the `PipeTransform` interface contract. This method has two parameters:

- `value`
- `metadata`

The `value` parameter is the currently processed method argument (before it is received by the route handling method), and `metadata` is the currently processed method argument's metadata. The metadata object has these properties:

```
export interface ArgumentMetadata {
  type: 'body' | 'query' | 'param' | 'custom';
  metatype?: Type<unknown>;
  data?: string;
}
```

These properties describe the currently processed argument.

`type`	Indicates whether the argument is a body `@Body()`, query `@Query()`, param `@Param()`, or a custom parameter (read more here).
`metatype`	Provides the metatype of the argument, for example, `String`. Note: the value is `undefined` if you either omit a type declaration in the route handler method signature, or use vanilla JavaScript.
`data`	The string passed to the decorator, for example `@Body('string')`. It's `undefined` if you leave the decorator parenthesis empty.

> warning **Warning** TypeScript interfaces disappear during transpilation. Thus, if a method parameter's type is declared as an interface instead of a class, the `metatype` value will be `Object`.

Schema based validation

Let's make our validation pipe a little more useful. Take a closer look at the `create()` method of the `CatsController`, where we probably would like to ensure that the post body object is valid before attempting to run our service method.

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

Let's focus in on the `createCatDto` body parameter. Its type is `CreateCatDto`:

```
@@filename(create-cat.dto)
export class CreateCatDto {
  name: string;
  age: number;
  breed: string;
}
```

We want to ensure that any incoming request to the create method contains a valid body. So we have to validate the three members of the `createCatDto` object. We could do this inside the route handler method, but doing so is not ideal as it would break the **single responsibility rule** (SRP).

Another approach could be to create a **validator class** and delegate the task there. This has the disadvantage that we would have to remember to call this validator at the beginning of each method.

How about creating validation middleware? This could work, but unfortunately, it's not possible to create **generic middleware** which can be used across all contexts across the whole application. This is because middleware is unaware of the **execution context**, including the handler that will be called and any of its parameters.

This is, of course, exactly the use case for which pipes are designed. So let's go ahead and refine our validation pipe.

Object schema validation

There are several approaches available for doing object validation in a clean, DRY way. One common approach is to use **schema-based** validation. Let's go ahead and try that approach.

The Zod library allows you to create schemas in a straightforward way, with a readable API. Let's build a validation pipe that makes use of Zod-based schemas.

Start by installing the required package:

```
$ npm install --save zod
```

In the code sample below, we create a simple class that takes a schema as a `constructor` argument. We then apply the `schema.parse()` method, which validates our incoming argument against the provided schema.

As noted earlier, a **validation pipe** either returns the value unchanged or throws an exception.

In the next section, you'll see how we supply the appropriate schema for a given controller method using the `@UsePipes()` decorator. Doing so makes our validation pipe reusable across contexts, just as we set out to do.

```
@@filename()
import { PipeTransform, ArgumentMetadata, BadRequestException } from
'@nestjs/common';
import { ZodObject } from 'zod';

export class ZodValidationPipe implements PipeTransform {
  constructor(private schema: ZodObject<any>) {}

  transform(value: unknown, metadata: ArgumentMetadata) {
    try {
      this.schema.parse(value);
    } catch (error) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }
}
@@switch
import { BadRequestException } from '@nestjs/common';
import { ZodObject } from 'zod';

export class ZodValidationPip {
  constructor(private schema) {}

  transform(value, metadata) {
    try {
      this.schema.parse(value);
    } catch (error) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }
}
```

Binding validation pipes

Earlier, we saw how to bind transformation pipes (like `ParseIntPipe` and the rest of the `Parse*` pipes).

Binding validation pipes is also very straightforward.

In this case, we want to bind the pipe at the method call level. In our current example, we need to do the following to use the `ZodValidationPipe`:

1. Create an instance of the `ZodValidationPipe`
2. Pass the context-specific Zod schema in the class constructor of the pipe
3. Bind the pipe to the method

Zod schema example:

```
import { z } from 'zod';

export const createCatSchema = z
  .object({
    name: z.string(),
    age: z.number(),
    breed: z.string(),
  })
  .required();

export type CreateCatDto = z.infer<typeof createCatSchema>;
```

We do that using the `@UsePipes()` decorator as shown below:

```
@@filename(cats.controller)
@Post()
@UsePipes(new ZodValidationPipe(createCatSchema))
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
@@switch
@Post()
@Bind(Body())
@UsePipes(new ZodValidationPipe(createCatSchema))
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

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

> warning **Warning** `zod` library requires the `strictNullChecks` configuration to be enabled in your `tsconfig.json` file.

Class validator

> warning **Warning** The techniques in this section require TypeScript and are not available if your app is written using vanilla JavaScript.

Let's look at an alternate implementation for our validation technique.

Nest works well with the class-validator library. This powerful library allows you to use decorator-based validation. Decorator-based validation is extremely powerful, especially when combined with Nest's **Pipe** capabilities since we have access to the `metatype` of the processed property. Before we start, we need to install the required packages:

```
$ npm i --save class-validator class-transformer
```

Once these are installed, we can add a few decorators to the `CreateCatDto` class. Here we see a significant advantage of this technique: the `CreateCatDto` class remains the single source of truth for our Post body object (rather than having to create a separate validation class).

```
@@filename(create-cat.dto)
import { IsString, IsInt } from 'class-validator';

export class CreateCatDto {
  @IsString()
  name: string;

  @IsInt()
  age: number;

  @IsString()
  breed: string;
}
```

> info **Hint** Read more about the class-validator decorators here.

Now we can create a `ValidationPipe` class that uses these annotations.

```
@@filename(validation.pipe)
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException
} from '@nestjs/common';
import { validate } from 'class-validator';
import { plainToInstance } from 'class-transformer';
```

```
@Injectable()
export class ValidationPipe implements PipeTransform<any> {
  async transform(value: any, { metatype }: ArgumentMetadata) {
    if (!metatype || !this.toValidate(metatype)) {
      return value;
    }
    const object = plainToInstance(metatype, value);
    const errors = await validate(object);
    if (errors.length > 0) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }

  private toValidate(metatype: Function): boolean {
    const types: Function[] = [String, Boolean, Number, Array, Object];
    return !types.includes(metatype);
  }
}
```

> info **Hint** As a reminder, you don't have to build a generic validation pipe on your own since the `ValidationPipe` is provided by Nest out-of-the-box. The built-in `ValidationPipe` offers more options than the sample we built in this chapter, which has been kept basic for the sake of illustrating the mechanics of a custom-built pipe. You can find full details, along with lots of examples here.

> warning **Notice** We used the class-transformer library above which is made by the same author as the **class-validator** library, and as a result, they play very well together.

Let's go through this code. First, note that the `transform()` method is marked as `async`. This is possible because Nest supports both synchronous and **asynchronous** pipes. We make this method `async` because some of the class-validator validations can be async (utilize Promises).

Next note that we are using destructuring to extract the metatype field (extracting just this member from an `ArgumentMetadata`) into our `metatype` parameter. This is just shorthand for getting the full `ArgumentMetadata` and then having an additional statement to assign the metatype variable.

Next, note the helper function `toValidate()`. It's responsible for bypassing the validation step when the current argument being processed is a native JavaScript type (these can't have validation decorators attached, so there's no reason to run them through the validation step).

Next, we use the class-transformer function `plainToInstance()` to transform our plain JavaScript argument object into a typed object so that we can apply validation. The reason we must do this is that the incoming post body object, when deserialized from the network request, does **not have any type information** (this is the way the underlying platform, such as Express, works). Class-validator needs to use the validation decorators we defined for our DTO earlier, so we need to perform this transformation to treat the incoming body as an appropriately decorated object, not just a plain vanilla object.

Finally, as noted earlier, since this is a **validation pipe** it either returns the value unchanged, or throws an exception.

The last step is to bind the `ValidationPipe`. Pipes can be parameter-scoped, method-scoped, controller-scoped, or global-scoped. Earlier, with our Joi-based validation pipe, we saw an example of binding the pipe at the method level. In the example below, we'll bind the pipe instance to the route handler `@Body()` decorator so that our pipe is called to validate the post body.

```
@@filename(cats.controller)
@Post()
async create(
  @Body(new ValidationPipe()) createCatDto: CreateCatDto,
) {
  this.catsService.create(createCatDto);
}
```

Parameter-scoped pipes are useful when the validation logic concerns only one specified parameter.

Global scoped pipes

Since the `ValidationPipe` was created to be as generic as possible, we can realize it's full utility by setting it up as a **global-scoped** pipe so that it is applied to every route handler across the entire application.

```
@@filename(main)
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

> warning **Notice** In the case of hybrid apps the `useGlobalPipes()` method doesn't set up pipes for gateways and micro services. For "standard" (non-hybrid) microservice apps, `useGlobalPipes()` does mount pipes globally.

Global pipes are used across the whole application, for every controller and every route handler.

Note that in terms of dependency injection, global pipes registered from outside of any module (with `useGlobalPipes()` as in the example above) cannot inject dependencies since the binding has been done outside the context of any module. In order to solve this issue, you can set up a global pipe **directly from any module** using the following construction:

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

@Module({
  providers: [
    {
      provide: APP_PIPE,
      useClass: ValidationPipe,
    },
  ],
})
export class AppModule {}
```

> info **Hint** When using this approach to perform dependency injection for the pipe, note that regardless of the module where this construction is employed, the pipe is, in fact, global. Where should this be done? Choose the module where the pipe (`ValidationPipe` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more here.

The built-in ValidationPipe

As a reminder, you don't have to build a generic validation pipe on your own since the `ValidationPipe` is provided by Nest out-of-the-box. The built-in `ValidationPipe` offers more options than the sample we built in this chapter, which has been kept basic for the sake of illustrating the mechanics of a custom-built pipe. You can find full details, along with lots of examples here.

Transformation use case

Validation isn't the only use case for custom pipes. At the beginning of this chapter, we mentioned that a pipe can also **transform** the input data to the desired format. This is possible because the value returned from the `transform` function completely overrides the previous value of the argument.

When is this useful? Consider that sometimes the data passed from the client needs to undergo some change - for example converting a string to an integer - before it can be properly handled by the route handler method. Furthermore, some required data fields may be missing, and we would like to apply default values. **Transformation pipes** can perform these functions by interposing a processing function between the client request and the request handler.

Here's a simple `ParseIntPipe` which is responsible for parsing a string into an integer value. (As noted above, Nest has a built-in `ParseIntPipe` that is more sophisticated; we include this as a simple example of a custom transformation pipe).

```
@@filename(parse-int.pipe)
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException
} from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform<string, number> {
  transform(value: string, metadata: ArgumentMetadata): number {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}
@@switch
import { Injectable, BadRequestException } from '@nestjs/common';

@Injectable()
export class ParseIntPipe {
  transform(value, metadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}
```

We can then bind this pipe to the selected param as shown below:

```
@@filename()
@Get(':id')
async findOne(@Param('id', new ParseIntPipe()) id) {
  return this.catsService.findOne(id);
}
@@switch
@Get(':id')
@Bind(Param('id', new ParseIntPipe()))
async findOne(id) {
  return this.catsService.findOne(id);
}
```

Another useful transformation case would be to select an **existing user** entity from the database using an id supplied in the request:

```
@@filename()
@Get(':id')
findOne(@Param('id', UserByIdPipe) userEntity: UserEntity) {
  return userEntity;
}
@@switch
@Get(':id')
@Bind(Param('id', UserByIdPipe))
findOne(userEntity) {
  return userEntity;
}
```

We leave the implementation of this pipe to the reader, but note that like all other transformation pipes, it receives an input value (an `id`) and returns an output value (a `UserEntity` object). This can make your code more declarative and DRY by abstracting boilerplate code out of your handler and into a common pipe.

Providing defaults

Parse* pipes expect a parameter's value to be defined. They throw an exception upon receiving null or undefined values. To allow an endpoint to handle missing querystring parameter values, we have to provide a default value to be injected before the Parse* pipes operate on these values. The DefaultValuePipe serves that purpose. Simply instantiate a DefaultValuePipe in the @Query() decorator before the relevant Parse* pipe, as shown below:

```
@@filename()
@Get()
async findAll(
  @Query('activeOnly', new DefaultValuePipe(false), ParseBoolPipe)
activeOnly: boolean,
  @Query('page', new DefaultValuePipe(0), ParseIntPipe) page: number,
) {
  return this.catsService.findAll({ activeOnly, page });
}
```

Guards

A guard is a class annotated with the `@Injectable()` decorator, which implements the `CanActivate` interface.

Guards have a **single responsibility**. They determine whether a given request will be handled by the route handler or not, depending on certain conditions (like permissions, roles, ACLs, etc.) present at run-time. This is often referred to as **authorization**. Authorization (and its cousin, **authentication**, with which it usually collaborates) has typically been handled by middleware in traditional Express applications. Middleware is a fine choice for authentication, since things like token validation and attaching properties to the `request` object are not strongly connected with a particular route context (and its metadata).

But middleware, by its nature, is dumb. It doesn't know which handler will be executed after calling the `next()` function. On the other hand, **Guards** have access to the `ExecutionContext` instance, and thus know exactly what's going to be executed next. They're designed, much like exception filters, pipes, and interceptors, to let you interpose processing logic at exactly the right point in the request/response cycle, and to do so declaratively. This helps keep your code DRY and declarative.

> info **Hint** Guards are executed **after** all middleware, but **before** any interceptor or pipe.

Authorization guard

As mentioned, **authorization** is a great use case for Guards because specific routes should be available only when the caller (usually a specific authenticated user) has sufficient permissions. The `AuthGuard` that we'll build now assumes an authenticated user (and that, therefore, a token is attached to the request headers). It will extract and validate the token, and use the extracted information to determine whether the request can proceed or not.

```
@@filename(auth.guard)
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class AuthGuard {
  async canActivate(context) {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
```

> info **Hint** If you are looking for a real-world example on how to implement an authentication mechanism in your application, visit this chapter. Likewise, for more sophisticated authorization example, check this page.

The logic inside the `validateRequest()` function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.

Every guard must implement a `canActivate()` function. This function should return a boolean, indicating whether the current request is allowed or not. It can return the response either synchronously or asynchronously (via a `Promise` or `Observable`). Nest uses the return value to control the next action:

- if it returns `true`, the request will be processed.
- if it returns `false`, Nest will deny the request.

Execution context

The `canActivate()` function takes a single argument, the `ExecutionContext` instance. The `ExecutionContext` inherits from `ArgumentsHost`. We saw `ArgumentsHost` previously in the exception filters chapter. In the sample above, we are just using the same helper methods defined on `ArgumentsHost` that we used earlier, to get a reference to the `Request` object. You can refer back to the **Arguments host** section of the exception filters chapter for more on this topic.

By extending `ArgumentsHost`, `ExecutionContext` also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic guards that can work across a broad set of controllers, methods, and execution contexts. Learn more about `ExecutionContext` here.

Role-based authentication

Let's build a more functional guard that permits access only to users with a specific role. We'll start with a basic guard template, and build on it in the coming sections. For now, it allows all requests to proceed:

```
@@filename(roles.guard)
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class RolesGuard {
  canActivate(context) {
    return true;
  }
}
```

Binding guards

Like pipes and exception filters, guards can be **controller-scoped**, method-scoped, or global-scoped. Below, we set up a controller-scoped guard using the `@UseGuards()` decorator. This decorator may take a single argument, or a comma-separated list of arguments. This lets you easily apply the appropriate set of guards with one declaration.

```
@@filename()
@Controller('cats')
@UseGuards(RolesGuard)
export class CatsController {}
```

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

Above, we passed the `RolesGuard` class (instead of an instance), leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes and exception filters, we can also pass an in-place instance:

```
@@filename()
@Controller('cats')
@UseGuards(new RolesGuard())
export class CatsController {}
```

The construction above attaches the guard to every handler declared by this controller. If we wish the guard to apply only to a single method, we apply the `@UseGuards()` decorator at the **method level**.

In order to set up a global guard, use the `useGlobalGuards()` method of the Nest application instance:

```
@@filename()
const app = await NestFactory.create(AppModule);
app.useGlobalGuards(new RolesGuard());
```

> warning **Notice** In the case of hybrid apps the `useGlobalGuards()` method doesn't set up guards for gateways and micro services by default (see Hybrid application for information on how to change

> this behavior). For "standard" (non-hybrid) microservice apps, `useGlobalGuards()` does mount the guards globally.

Global guards are used across the whole application, for every controller and every route handler. In terms of dependency injection, global guards registered from outside of any module (with `useGlobalGuards()` as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up a guard directly from any module using the following construction:

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

@Module({
  providers: [
    {
      provide: APP_GUARD,
      useClass: RolesGuard,
    },
  ],
})
export class AppModule {}
```

> info **Hint** When using this approach to perform dependency injection for the guard, note that regardless of the module where this construction is employed, the guard is, in fact, global. Where should this be done? Choose the module where the guard (`RolesGuard` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more here.

Setting roles per handler

Our `RolesGuard` is working, but it's not very smart yet. We're not yet taking advantage of the most important guard feature - the execution context. It doesn't yet know about roles, or which roles are allowed for each handler. The `CatsController`, for example, could have different permission schemes for different routes. Some might be available only for an admin user, and others could be open for everyone. How can we match roles to routes in a flexible and reusable way?

This is where **custom metadata** comes into play (learn more here). Nest provides the ability to attach custom **metadata** to route handlers through either decorators created via `Reflector#createDecorator` static method, or the built-in `@SetMetadata()` decorator.

For example, let's create a `@Roles()` decorator using the `Reflector#createDecorator` method that will attach the metadata to the handler. `Reflector` is provided out of the box by the framework and exposed from the `@nestjs/core` package.

```
@@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.

Alernatively, instead of using the `Reflector#createDecorator` method, we could use the built-in `@SetMetadata()` decorator. Learn more about here.

Putting it all together

Let's now go back and tie this together with our `RolesGuard`. Currently, it simply returns `true` in all cases, allowing every request to proceed. We want to make the return value conditional based on the comparing the **roles assigned to the current user** to the actual roles required by the current route being processed. In order to access the route's role(s) (custom metadata), we'll use the `Reflector` helper class again, as follows:

```
@@filename(roles.guard)
import { Injectable, CanActivate, ExecutionContext } from
'@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Roles } from './roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return matchRoles(roles, user.roles);
  }
}
@@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Roles } from './roles.decorator';

@Injectable()
@Dependencies(Reflector)
export class RolesGuard {
  constructor(reflector) {
    this.reflector = reflector;
  }

  canActivate(context) {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return matchRoles(roles, user.roles);
  }
}
```

> info **Hint** In the node.js world, it's common practice to attach the authorized user to the `request` object. Thus, in our sample code above, we are assuming that `request.user` contains the user instance and allowed roles. In your app, you will probably make that association in your custom **authentication guard** (or middleware). Check this chapter for more information on this topic.

> warning **Warning** The logic inside the `matchRoles()` function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.

Refer to the Reflection and metadata section of the **Execution context** chapter for more details on utilizing `Reflector` in a context-sensitive way.

When a user with insufficient privileges requests an endpoint, Nest automatically returns the following response:

```
{
  "statusCode": 403,
  "message": "Forbidden resource",
  "error": "Forbidden"
}
```

Note that behind the scenes, when a guard returns `false`, the framework throws a `ForbiddenException`. If you want to return a different error response, you should throw your own specific exception. For example:

```
throw new UnauthorizedException();
```

Any exception thrown by a guard will be handled by the exceptions layer (global exceptions filter and any exceptions filters that are applied to the current context).

> info **Hint** If you are looking for a real-world example on how to implement authorization, check this chapter.

Interceptors

An interceptor is a class annotated with the `@Injectable()` decorator and implements the `NestInterceptor` interface.

Interceptors have a set of useful capabilities which are inspired by the Aspect Oriented Programming (AOP) technique. They make it possible to:

- bind extra logic before / after method execution
- transform the result returned from a function
- transform the exception thrown from a function
- extend the basic function behavior
- completely override a function depending on specific conditions (e.g., for caching purposes)

Basics

Each interceptor implements the `intercept()` method, which takes two arguments. The first one is the `ExecutionContext` instance (exactly the same object as for guards). The `ExecutionContext` inherits from `ArgumentsHost`. We saw `ArgumentsHost` before in the exception filters chapter. There, we saw that it's a wrapper around arguments that have been passed to the original handler, and contains different arguments arrays based on the type of the application. You can refer back to the exception filters for more on this topic.

Execution context

By extending `ArgumentsHost`, `ExecutionContext` also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic interceptors that can work across a broad set of controllers, methods, and execution contexts. Learn more about `ExecutionContext` here.

Call handler

The second argument is a `CallHandler`. The `CallHandler` interface implements the `handle()` method, which you can use to invoke the route handler method at some point in your interceptor. If you don't call the `handle()` method in your implementation of the `intercept()` method, the route handler method won't be executed at all.

This approach means that the `intercept()` method effectively **wraps** the request/response stream. As a result, you may implement custom logic **both before and after** the execution of the final route handler. It's clear that you can write code in your `intercept()` method that executes **before** calling `handle()`, but how do you affect what happens afterward? Because the `handle()` method returns an `Observable`, we can use powerful RxJS operators to further manipulate the response. Using Aspect Oriented Programming terminology, the invocation of the route handler (i.e., calling `handle()`) is called a Pointcut, indicating that it's the point at which our additional logic is inserted.

Consider, for example, an incoming `POST /cats` request. This request is destined for the `create()` handler defined inside the `CatsController`. If an interceptor which does not call the `handle()` method is called anywhere along the way, the `create()` method won't be executed. Once `handle()` is called (and its `Observable` has been returned), the `create()` handler will be triggered. And once the response stream is received via the `Observable`, additional operations can be performed on the stream, and a final result returned to the caller.

Aspect interception

The first use case we'll look at is to use an interceptor to log user interaction (e.g., storing user calls, asynchronously dispatching events or calculating a timestamp). We show a simple `LoggingInterceptor` below:

```
@@filename(logging.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
@@switch
import { Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor {
  intercept(context, next) {
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
```

> info **Hint** The `NestInterceptor<T, R>` is a generic interface in which `T` indicates the type of an `Observable<T>` (supporting the response stream), and `R` is the type of the value wrapped by `Observable<R>`.

> warning **Notice** Interceptors, like controllers, providers, guards, and so on, can **inject dependencies** through their `constructor`.

Since `handle()` returns an RxJS `Observable`, we have a wide choice of operators we can use to manipulate the stream. In the example above, we used the `tap()` operator, which invokes our anonymous logging function upon graceful or exceptional termination of the observable stream, but doesn't otherwise interfere with the response cycle.

Binding interceptors

In order to set up the interceptor, we use the `@UseInterceptors()` decorator imported from the `@nestjs/common` package. Like pipes and guards, interceptors can be controller-scoped, method-scoped, or global-scoped.

```
@@filename(cats.controller)
@UseInterceptors(LoggingInterceptor)
export class CatsController {}
```

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

Using the above construction, each route handler defined in `CatsController` will use `LoggingInterceptor`. When someone calls the `GET /cats` endpoint, you'll see the following output in your standard output:

```
Before...
After... 1ms
```

Note that we passed the `LoggingInterceptor` type (instead of an instance), leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes, guards, and exception filters, we can also pass an in-place instance:

```
@@filename(cats.controller)
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

As mentioned, the construction above attaches the interceptor to every handler declared by this controller. If we want to restrict the interceptor's scope to a single method, we simply apply the decorator at the **method level**.

In order to set up a global interceptor, we use the `useGlobalInterceptors()` method of the Nest application instance:

```
const app = await NestFactory.create(AppModule);
app.useGlobalInterceptors(new LoggingInterceptor());
```

Global interceptors are used across the whole application, for every controller and every route handler. In terms of dependency injection, global interceptors registered from outside of any module (with `useGlobalInterceptors()`, as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up an interceptor **directly from any module** using the following construction:

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

@Module({
  providers: [
    {
      provide: APP_INTERCEPTOR,
      useClass: LoggingInterceptor,
    },
  ],
})
export class AppModule {}
```

> info **Hint** When using this approach to perform dependency injection for the interceptor, note that regardless of the module where this construction is employed, the interceptor is, in fact, global. Where should this be done? Choose the module where the interceptor (`LoggingInterceptor` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more here.

Response mapping

We already know that `handle()` returns an `Observable`. The stream contains the value **returned** from the route handler, and thus we can easily mutate it using RxJS's `map()` operator.

> warning **Warning** The response mapping feature doesn't work with the library-specific response strategy (using the `@Res()` object directly is forbidden).

Let's create the `TransformInterceptor`, which will modify each response in a trivial way to demonstrate the process. It will use RxJS's `map()` operator to assign the response object to the `data` property of a newly created object, returning the new object to the client.

```
@@filename(transform.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
```

```
export interface Response<T> {
  data: T;
}

@Injectable()
export class TransformInterceptor<T> implements NestInterceptor<T,
Response<T>> {
  intercept(context: ExecutionContext, next: CallHandler):
Observable<Response<T>> {
    return next.handle().pipe(map(data => ({ data })));
  }
}
@@switch
import { Injectable } from '@nestjs/common';
import { map } from 'rxjs/operators';

@Injectable()
export class TransformInterceptor {
  intercept(context, next) {
    return next.handle().pipe(map(data => ({ data })));
  }
}
```

> info **Hint** Nest interceptors work with both synchronous and asynchronous `intercept()` methods. You can simply switch the method to `async` if necessary.

With the above construction, when someone calls the `GET /cats` endpoint, the response would look like the following (assuming that route handler returns an empty array `[]`):

```
{
  "data": []
}
```

Interceptors have great value in creating re-usable solutions to requirements that occur across an entire application. For example, imagine we need to transform each occurrence of a `null` value to an empty string `''`. We can do it using one line of code and bind the interceptor globally so that it will automatically be used by each registered handler.

```
@@filename()
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    return next
      .handle()
      .pipe(map(value => value === null ? '' : value ));
  }
}
@@switch
import { Injectable } from '@nestjs/common';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor {
  intercept(context, next) {
    return next
      .handle()
      .pipe(map(value => value === null ? '' : value ));
  }
}
```

Exception mapping

Another interesting use-case is to take advantage of RxJS's `catchError()` operator to override thrown exceptions:

```
@@filename(errors.interceptor)
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  BadGatewayException,
  CallHandler,
} from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
export class ErrorsInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    return next
      .handle()
      .pipe(
        catchError(err => throwError(() => new BadGatewayException())),
      );
  }
}
@@switch
import { Injectable, BadGatewayException } from '@nestjs/common';
import { throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
```

```
export class ErrorsInterceptor {
  intercept(context, next) {
    return next
      .handle()
      .pipe(
        catchError(err => throwError(() => new BadGatewayException())),
      );
  }
}
```

Stream overriding

There are several reasons why we may sometimes want to completely prevent calling the handler and return a different value instead. An obvious example is to implement a cache to improve response time. Let's take a look at a simple **cache interceptor** that returns its response from a cache. In a realistic example, we'd want to consider other factors like TTL, cache invalidation, cache size, etc., but that's beyond the scope of this discussion. Here we'll provide a basic example that demonstrates the main concept.

```
@@filename(cache.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
'@nestjs/common';
import { Observable, of } from 'rxjs';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
@@switch
import { Injectable } from '@nestjs/common';
import { of } from 'rxjs';

@Injectable()
export class CacheInterceptor {
  intercept(context, next) {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
```

Our `CacheInterceptor` has a hardcoded `isCached` variable and a hardcoded response `[]` as well. The key point to note is that we return a new stream here, created by the RxJS `of()` operator, therefore the route handler **won't be called** at all. When someone calls an endpoint that makes use of `CacheInterceptor`, the response (a hardcoded, empty array) will be returned immediately. In order to create a generic solution, you can take advantage of `Reflector` and create a custom decorator. The `Reflector` is well described in the guards chapter.

More operators

The possibility of manipulating the stream using RxJS operators gives us many capabilities. Let's consider another common use case. Imagine you would like to handle **timeouts** on route requests. When your endpoint doesn't return anything after a period of time, you want to terminate with an error response. The following construction enables this:

```
@@filename(timeout.interceptor)
import { Injectable, NestInterceptor, ExecutionContext, CallHandler,
RequestTimeoutException } from '@nestjs/common';
import { Observable, throwError, TimeoutError } from 'rxjs';
import { catchError, timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    return next.handle().pipe(
      timeout(5000),
      catchError(err => {
        if (err instanceof TimeoutError) {
          return throwError(() => new RequestTimeoutException());
        }
        return throwError(() => err);
      }),
    );
  };
};
@@switch
import { Injectable, RequestTimeoutException } from '@nestjs/common';
import { Observable, throwError, TimeoutError } from 'rxjs';
import { catchError, timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor {
  intercept(context, next) {
    return next.handle().pipe(
      timeout(5000),
      catchError(err => {
        if (err instanceof TimeoutError) {
          return throwError(() => new RequestTimeoutException());
        }
        return throwError(() => err);
      }),
      );
    };
  };
```

After 5 seconds, request processing will be canceled. You can also add custom logic before throwing `RequestTimeoutException` (e.g. release resources).

Custom route decorators

Nest is built around a language feature called **decorators**. Decorators are a well-known concept in a lot of commonly used programming languages, but in the JavaScript world, they're still relatively new. In order to better understand how decorators work, we recommend reading this article. Here's a simple definition:

> An ES2016 decorator is an expression which returns a function and can take a target, name and property descriptor as arguments. You apply it by prefixing the decorator with an `@` character and placing this at the very top of what you are trying to decorate. Decorators can be defined for either a class, a method or a property.

Param decorators

Nest provides a set of useful **param decorators** that you can use together with the HTTP route handlers. Below is a list of the provided decorators and the plain Express (or Fastify) objects they represent

`@Request(), @Req()`	`req`
`@Response(), @Res()`	`res`
`@Next()`	`next`
`@Session()`	`req.session`
`@Param(param?: string)`	`req.params` / `req.params[param]`
`@Body(param?: string)`	`req.body` / `req.body[param]`
`@Query(param?: string)`	`req.query` / `req.query[param]`
`@Headers(param?: string)`	`req.headers` / `req.headers[param]`
`@Ip()`	`req.ip`
`@HostParam()`	`req.hosts`

Additionally, you can create your own **custom decorators**. Why is this useful?

In the node.js world, it's common practice to attach properties to the **request** object. Then you manually extract them in each route handler, using code like the following:

```
const user = req.user;
```

In order to make your code more readable and transparent, you can create a `@User()` decorator and reuse it across all of your controllers.

```
@@filename(user.decorator)
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return request.user;
  },
);
```

Then, you can simply use it wherever it fits your requirements.

```
@@filename()
@Get()
async findOne(@User() user: UserEntity) {
  console.log(user);
}
@@switch
@Get()
@Bind(User())
async findOne(user) {
  console.log(user);
}
```

Passing data

When the behavior of your decorator depends on some conditions, you can use the `data` parameter to pass an argument to the decorator's factory function. One use case for this is a custom decorator that extracts properties from the request object by key. Let's assume, for example, that our authentication layer validates requests and attaches a user entity to the request object. The user entity for an authenticated request might look like:

```
{
  "id": 101,
  "firstName": "Alan",
  "lastName": "Turing",
  "email": "alan@email.com",
  "roles": ["admin"]
}
```

Let's define a decorator that takes a property name as key, and returns the associated value if it exists (or undefined if it doesn't exist, or if the `user` object has not been created).

```
@@filename(user.decorator)
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    const user = request.user;
    return data ? user?.[data] : user;
  },
);
@@switch
import { createParamDecorator } from '@nestjs/common';

export const User = createParamDecorator((data, ctx) => {
  const request = ctx.switchToHttp().getRequest();
  const user = request.user;

  return data ? user && user[data] : user;
});
```

Here's how you could then access a particular property via the `@User()` decorator in the controller:

```
@@filename()
@Get()
async findOne(@User('firstName') firstName: string) {
  console.log(`Hello ${firstName}`);
}
@@switch
@Get()
@Bind(User('firstName'))
async findOne(firstName) {
  console.log(`Hello ${firstName}`);
}
```

You can use this same decorator with different keys to access different properties. If the `user` object is deep or complex, this can make for easier and more readable request handler implementations.

> info **Hint** For TypeScript users, note that `createParamDecorator<T>()` is a generic. This means you can explicitly enforce type safety, for example `createParamDecorator<string>((data, ctx) => ...)`. Alternatively, specify a parameter type in the factory function, for example `createParamDecorator((data: string, ctx) => ...)`. If you omit both, the type for `data` will be `any`.

Working with pipes

Nest treats custom param decorators in the same fashion as the built-in ones (`@Body()`, `@Param()` and `@Query()`). This means that pipes are executed for the custom annotated parameters as well (in our examples, the `user` argument). Moreover, you can apply the pipe directly to the custom decorator:

```
@@filename()
@Get()
async findOne(
  @User(new ValidationPipe({ validateCustomDecorators: true }))
  user: UserEntity,
) {
  console.log(user);
}
@@switch
@Get()
@Bind(User(new ValidationPipe({ validateCustomDecorators: true })))
async findOne(user) {
  console.log(user);
}
```

> info **Hint** Note that `validateCustomDecorators` option must be set to true. `ValidationPipe` does not validate arguments annotated with the custom decorators by default.

Decorator composition

Nest provides a helper method to compose multiple decorators. For example, suppose you want to combine all decorators related to authentication into a single decorator. This could be done with the following construction:

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

export function Auth(...roles: Role[]) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}
@@switch
import { applyDecorators } from '@nestjs/common';

export function Auth(...roles) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}
```

You can then use this custom `@Auth()` decorator as follows:

```
@Get('users')
@Auth('admin')
findAllUsers() {}
```

This has the effect of applying all four decorators with a single declaration.

> warning **Warning** The `@ApiHideProperty()` decorator from the `@nestjs/swagger` package is not composable and won't work properly with the `applyDecorators` function.