Progressive Coder

NESTJS FUNDAMENTALS

Beginner-Friendly Guide

SAURABH DASHORA

NestJS Fundamentals

Chapter 1 - Setup and Platform Selection

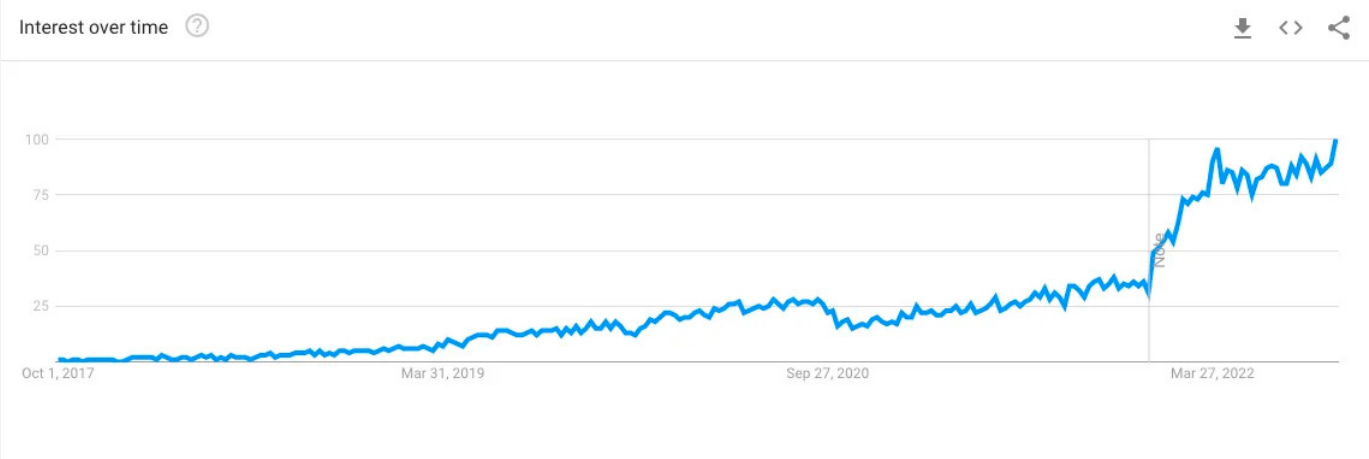
Introducing NestJS

Over the years, NestJS has become a very popular framework.

If Github is any indication of a framework's popularity within the developer community, NestJS is in second place with almost 53K stars (at the time of writing this).

For reference, **ExpressJS** has around 59K stars and it has been around far longer.

Even if you check Google Trends, the search term NestJS has seen a massive worldwide jump in search interest during the last couple of years.



What is the reason for the growing popularity of NestJS?

Actually, there are multiple reasons:

- NestJS adds discipline and structure to your Node.js applications. This is unlike anything you've seen before in the Node.js ecosystem. NestJS does for Node.js what Spring Boot did for the Java ecosystem a decade ago.

- NestJS encourages the use of **TypeScript**. This means you get additional features like static typing and class-based object-oriented programming.
- Being built on top of already-popular web frameworks and libraries such as Express.js and Angular makes it easy for developers to adopt NestJS.
- NestJS follows a **modular and scalable architecture** supported by an ecosystem of multiple pre-built modules. This frees up developer time to focus on the actual business problem rather than worrying about low-level details.

Generating your first NestJS Project

It's always exciting to start learning a new framework.

However, before you go ahead with any of this, you would need Node.js and NPM installed on your system. To install Node.js, refer to the official Node.js website.

Once you have got that nailed down, NestJS makes it extremely easy to get started.

Broadly, there are two main options to start with NestJS:

- Nest CLI
- Manual Setup

👉 Nest CLI

I strongly recommend using the Nest CLI.

The CLI makes it easy to scaffold a new project. It also provides many other features such as generating modules and controllers.

You can install the `@nestjs/cli` package using NPM. See the below command:

```
$ npm i -g @nestjs/cli
```

The `-g` flag over here stands for global. It means that the Nest CLI is available globally across your local system.

Once the installation is complete, you can generate a brand-new project using the `nest new` command. Refer to the example.

```
$ nest new nestjs-course-demo-app
⚡  We will scaffold your app in a few seconds..

CREATE nestjs-course-demo-app/.eslintrc.js (665 bytes)
CREATE nestjs-course-demo-app/.prettierrc (51 bytes)
CREATE nestjs-course-demo-app/README.md (3340 bytes)
CREATE nestjs-course-demo-app/nest-cli.json (118 bytes)
CREATE nestjs-course-demo-app/package.json (2007 bytes)
CREATE nestjs-course-demo-app/tsconfig.build.json (97 bytes)
CREATE nestjs-course-demo-app/tsconfig.json (546 bytes)
CREATE nestjs-course-demo-app/src/app.controller.spec.ts (617 bytes)
CREATE nestjs-course-demo-app/src/app.controller.ts (274 bytes)
CREATE nestjs-course-demo-app/src/app.module.ts (249 bytes)
CREATE nestjs-course-demo-app/src/app.service.ts (142 bytes)
CREATE nestjs-course-demo-app/src/main.ts (208 bytes)
CREATE nestjs-course-demo-app/test/app.e2e-spec.ts (630 bytes)
CREATE nestjs-course-demo-app/test/jest-e2e.json (183 bytes)

? Which package manager would you ❤️  to use? npm
▹▹▸▹▹ Installation in progress... ☕
```

When you execute the command, it will prompt you for a choice of package manager. You can choose between `npm` or `yarn`. In this book, I will go with `npm`.

This command creates a new project directory `nestjs-course-demo-app` with a basic project structure. The project directory contains some initial NestJS files and other supporting modules.

👉 Manual Setup

Nest CLI generates a project based on recommended best practices.

Sometimes, you might not want to use the best practices and need a high degree of customization. In such rare situations, you can install the core NestJS package along with the supporting libraries.

See the below command.

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

Once the packages are installed, you can create your own project structure according to your specific requirements.

What's inside a NestJS Project?

When you scaffold a project using the Nest CLI, you get a project directory with a bunch of sub-folders and files.

The core files are present in the `src` directory.

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

Each file has a specific use.

- **app.controller.ts** - This is a basic controller with a single example route.
- **app.controller.spec.ts** - This file contains the unit tests for the controller.
- **app.module.ts** - The root module of our NestJS application.
- **app.service.ts** - A demo service class with a single method.
- **main.ts** - The application entry file that creates the NestJS application instance.

As you can notice, all of these are TypeScript files.

NestJS supports both Typescript and vanilla Javascript. However, **the creators of NestJS officially recommend using Typescript**. The use of Typescript allows us to use the features of NestJS in a much better way.

Here's what the `main.ts` file of our project looks like.

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

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

bootstrap();
```

So, what is going on over here?

As discussed earlier, the `main.ts` file is the entry point to the NestJS Application. It contains an async function `bootstrap()` that will bootstrap the entire application.

> *Bootstrapping means loading a particular program into the system's memory for execution.*
>
> *Essentially, it is the first piece of code that runs when you try to start the application.*

Here are some points that explain the whole process:

- On a high level, the `bootstrap()` function creates a Nest application instance by using the `NestFactory` class.
- The `NestFactory` class is part of the `@nestjs/core` package. It exposes a few **static methods** to create an application instance.
- The `create()` method is one such static method. It takes the application root module as input and returns an application or the `app` object. This `app` object uses the `INestApplication` interface.
- The `app` object has several methods you can use to control the application. For example, the above code snippet calls the `listen()` method to start up a basic

HTTP listener.
- The `listen()` method takes the port number as input and makes the application listen to incoming HTTP requests.

Running the Application

Time to run our starter project.

And yes - it's possible to do so without making any changes to the starter project. This is what makes the CLI so useful. It gives you access to a working project straight out of the gate.

You can run the project by executing the below command:

```
$ npm run start
```

This command starts up the application and makes the HTTP server listen on port 3000 as defined in the `main.ts` file.

You can open a browser window and navigate to `http://localhost:3000` to see the default `Hello, World!` message printed on the screen.

Also, you can start the application in watch mode using the below command.

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

In this mode, the local server will watch for code changes to automatically recompile and reload the server. This is a great feature during development.

In case you are wondering, these commands are defined in the `scripts` section of the `package.json` file.

```
"scripts": {
    "start": "nest start",
    "start:dev": "nest start --watch",
}
```

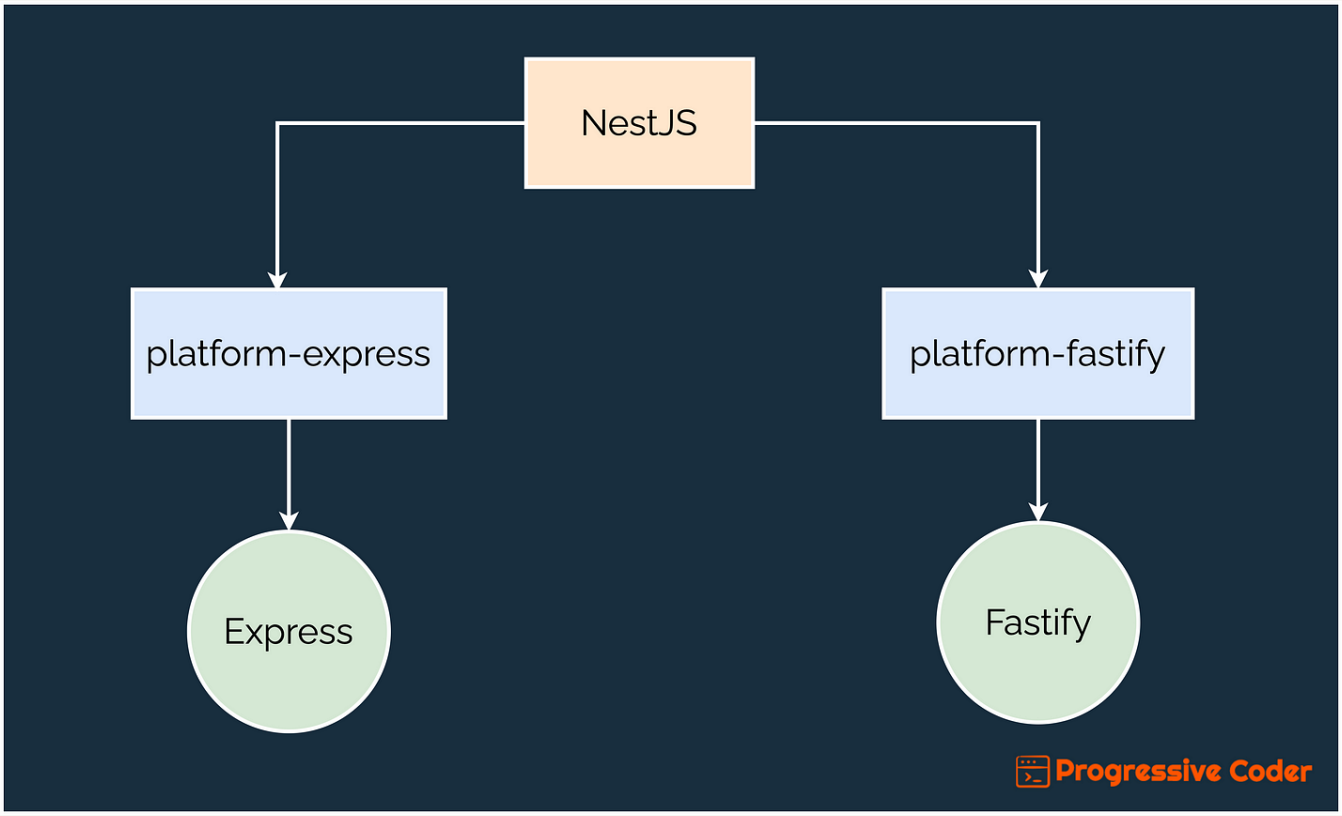
Choosing the Right Platform

NestJS is a **platform-agnostic** framework.

This means you can use NestJS with any **Node-based HTTP framework** after creating an adapter for the same. The primary function of such an adapter is to proxy middleware and request handlers to the chosen platform.

Out of the box, NestJS supports two platforms:

- `platform-express` - As the name suggests, this platform uses the **Express** framework. This is also the default setting and you don't have to make any changes to the starter project if you intend to go with Express.
- `platform-fastify` - This is the second option and it uses Fastify as the underlying HTTP framework. To activate Fastify, we have to explicitly specify the same in the `main.ts` file.



But what does it exactly mean by supporting these platforms?

NestJS exposes the chosen platform's interface. For example, the `platform-express` package exposes the `NestExpressApplication` interface.

You can also instruct the `NestFactory.create()` method to return the app object of a specific platform type.

👉 For example, to specify Express explicitly, you can tweak the `main.ts` file and import `NestExpressApplication` from `@nestjs/platform-express`.

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

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

bootstrap();
```

Of course, in the case of Express, this is unnecessary. As mentioned earlier, the starter project uses Express by default. If nothing is specified in the `create()` method, our application will automatically use Express.

👉 However, in order to work with **Fastify**, you need to install the `platform-fastify` package using the below command.

```
$ npm install --save @nestjs/platform-fastify
```

Once the installation is done, you need to enable `FastifyAdapter` within the `main.ts` file at the time of bootstrapping the NestJS application.

Check the below code:

```
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
```

```
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter()
  );
  await app.listen(3000, '0.0.0.0');
}

bootstrap();
```

Basically, you need to set the `NestFastifyApplication` type to the `NestFactory.create()` method. The type `NestFastifyApplication` is imported from the newly installed package. Also, the `create()` method takes an additional argument as input - an instance of the `FastifyAdapter` class.

By default, Fastify listens on `127.0.0.1` only. Therefore, to accept connections on other hosts, we specify `0.0.0.0` in the call to the `listen()` method.

Now, at this point, I won't fret too much about the choice of platform. Which platform to use depends on what you are trying to accomplish with your application.

🚀 The **advantage of using Express** is the large number of third-party libraries that are available in the Express ecosystem. It is a **very mature framework with great community support and documentation**. Since it has been around for so long, it is also easy to find developers with knowledge about Express. Moreover, the `platform-express` doesn't require you to perform any additional setups.

🚀 **Fastify, on the other hand, is geared towards high performance**. According to the developers of Fastify, the framework can support up to 30,000 requests per second. In essence, Fastify promises high performance with low overhead.

Ultimately, I feel that if you don't have any crazy performance requirements, it might be better to stick to NestJS with Express as the underlying platform.

In this book, I will be using the **Express platform**. However, wherever applicable, I will point out any specific differences between the two platforms.

Chapter 2 - Building Interfaces with Controllers

Congratulations on reaching the next level.

Here's a recap of what you have learnt in the previous chapter:

- Bootstrapping a NestJS project using the Nest CLI
- Learning the use of NestJS core files and the `main.ts` file.
- Understanding the platform independence concept of NestJS
- Choosing the right platform

Time to take the next step.

In this chapter, we will cover how to **build interfaces using controllers in NestJS**.

Controllers help us group route handlers based on functionality. This makes them one of the most important concepts to grasp in NestJS.

Here's the plan for this chapter:

- Create a simple NestJS controller
- Get a first look at classes and decorators
- Say hello to the **Request** and **Response** objects
- Use the `@Param()` and `@Body()` decorators

I recommend that you follow along and build the project as you go through this book. It will give you continuous hands-on practice and clarify doubts.

To make things easy, you can utilize the Github repository as a starting point so that there are fewer chances of discrepancies between the book and your code because of version mismatch.

What is a Controller?

In the world of web development, the controller is a component that acts as the **entry point for incoming HTTP requests**.

Think of it as the bouncer checking IDs at the club's entrance, making sure only authorized guests are allowed in.

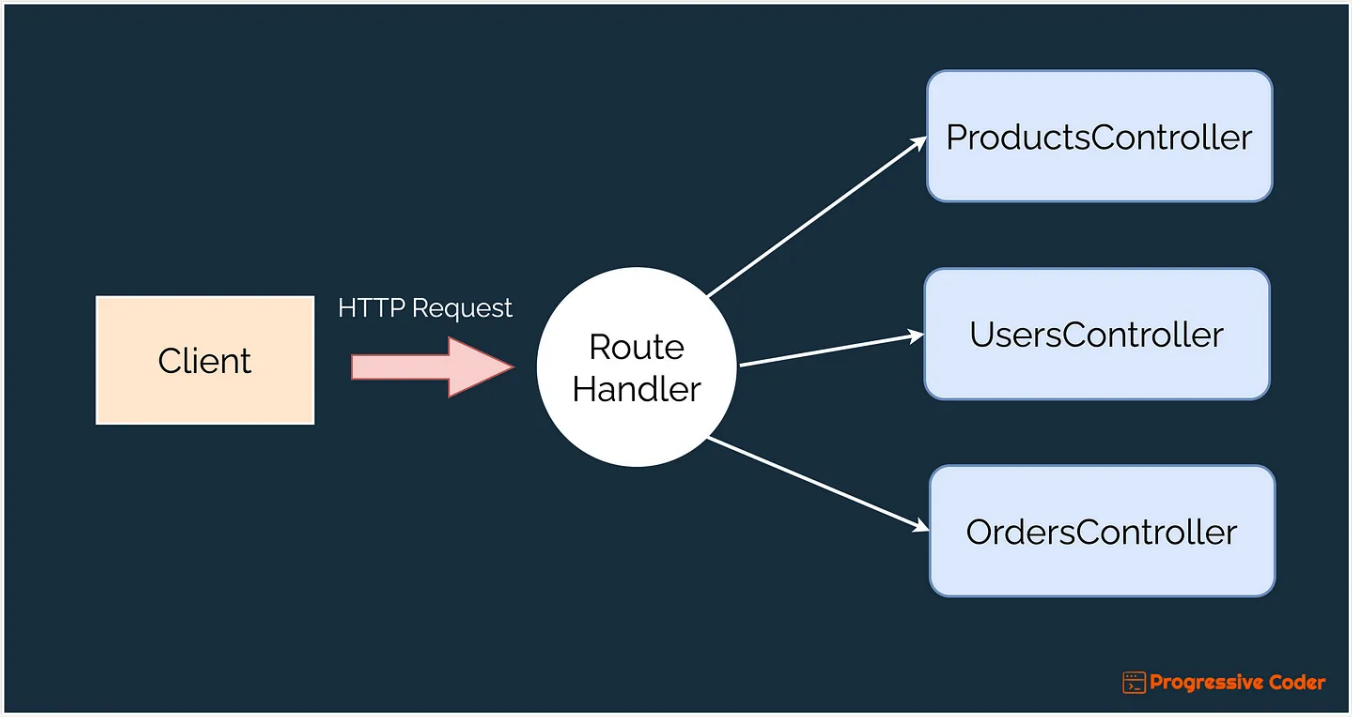
😞 OK. It was a lame comparison! I take it back.

In a nutshell, the job of a controller is to **handle incoming requests** from clients and **return appropriate responses** to the clients.

However, each request can differ from the other one. It can have different parameters or use another HTTP operation.

This is where the **routing mechanism** comes into the picture. This mechanism controls which request should go to which controller.

Here’s an illustration that shows the role of a **routing mechanism** in a typical web application. Based on the incoming request, the routing mechanism directs the request to the appropriate controller based on the properties of the incoming request.



To get a better idea, let's see a NestJS controller in action.

In the `src` directory of your starter project, create a new file named `products.controller.ts`. Don't worry about putting it in a sub-folder just yet as we will take care of code organization when we reach the topic of modules in NestJS.

Within the file, paste the below code.

```
import { Controller, Delete, Get, Post, Put } from "@nestjs/common";

@Controller('products')
export class ProductsController {

    @Get()
    getAllProducts(): string {
        return 'This endpoint returns a list of all products';
    }

    @Post()
    createProduct(): string {
        return 'This endpoint creates a new product';
    }

    @Put()
    updateProduct(): string {
        return 'This endpoint updates an existing product';
    }

    @Delete()
    deleteProduct(): string {
        return 'This endpoint deletes an existing product';
    }
}
```

In NestJS, we use **classes and decorators** to create a controller.

In the context of a controller, you could think of classes as a group of routes that perform some common functionality.

👉 The `@Controller` decorator in NestJS marks a class as a controller and defines a high-level route for it, which maps incoming HTTP requests to its methods (handlers).

Internally, the decorator also ensures that the class is registered as a component in the **NestJS Dependency Injection (DI) system**.

> *Dependency Injection (DI) is a software design pattern where an object's dependencies are passed to it by an external entity, rather than being created within the object itself.*
>
> *Think of it like a superhero (the object) who's got superpowers (its own behaviour), but needs gadgets (dependencies) to use those powers effectively. Rather than going on a shopping spree to get all the gadgets, the superhero's butler (DI system) brings everything to the superhero as and when needed.*
>
> *The superhero can now focus on saving the world (performing its own behaviour) without worrying about fetching its own gadgets!*

👉 The `@Controller` decorator takes in a path that acts as the route prefix for the methods within the controller.

In our code example, the `@Controller()` decorator also has the path prefix "products" as input. This tells the NestJS routing mechanism that any incoming requests to the route `/products` should be sent to `ProductsController` class.

> *The path prefix is an optional parameter for the decorator. However, using it can help us group a set of related routes. Hence, I recommend using it wherever possible.*

👉 There are 4 methods in the `ProductsController` class. Each method handles a particular HTTP request method.

For example, `getAllProducts()` handles any `GET` requests to `/products` route. Similarly, other methods handle the `POST`, `PUT` and `DELETE` request methods to the same route path.

👉 A route path is constructed by concatenating the optional prefix path in the `@Controller()` decorator and any path specified in the individual request handler.

For example, if the `getAllProducts()` method has a decorator `@Get('/fetch')`, the route path for the GET request will be `/products/fetch`.

> *The method name for the request handler such as `getAllProducts()` is arbitrary and totally up to the choice of the developer.*
>
> *NestJS does not attach any significance to the method name. It's only concerned with the decorators in order to map the correct request handler for a particular request.*

Registering the Controller

Even though we have declared the `ProductsController` class, NestJS won't pick up the class automatically during application startup.

You need to register the controller so that the dependency injection system can kick in and create an instance of the class when the application starts up.

In NestJS, controllers belong to a module. Therefore, you need to include the `ProductsController` in the `controllers` array of the `AppModule`. Think of it as a way of registering the controller.

Here's the change you need to do in the `app.module.ts` file of your project.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsController } from './products.controller';

@Module({
  imports: [],
  controllers: [AppController, ProductsController],
  providers: [AppService],
})
export class AppModule {}
```

You can now start the application using the `npm run start` command and test out the controller by accessing the various endpoints.

For example, you can access the `getAllProducts` endpoint by making a request to `http://localhost:3000/products`.

Of course, at this point, all controllers will just return a text message. So things won't feel particularly exciting.

Understanding the Request Object

While the request handlers in our above example perform the job of handling requests, they are not that useful.

To perform anything meaningful, request handlers need access to the request details sent by the client. NestJS provides access to the request object based on the underlying platform.

To access the request object, you can instruct NestJS to inject it by adding the `@Req()` decorator to the request handler's signature. Check out the below example:

```
import { Controller, Delete, Get, Post, Put, Req } from
"@nestjs/common";
import { Request } from "express";

@Controller('products')
export class ProductsController {

    @Get()
    getAllProducts(@Req() request: Request): string {
        console.log(request.headers)
        return 'This endpoint returns a list of all products';
    }
}
```

Here, the `Request` class is imported from `express`.

The `@Req()` decorator binds the incoming request object to the `request` variable. You can then access the `request` variable within the handler.

The request object represents the HTTP request. It has properties for the request query string, parameters, HTTP headers and also the request body.

While you can extract these values manually from the request object, NestJS also provides dedicated decorators to make your life easy.

The below table shows some of the important decorators you can use to deal with the request object.

DECORATOR	PLATFORM-SPECIFIC OBJECT (DEFAULT EXPRESS)
`@Request()` or `@Req()`	`req`
`@Response()` or `@Res()`	`res`
`@Next()`	`next`
`@Param()`	`req.params`
`@Param(key?: string)`	`req.params.key`
`@Body()`	`req.body`
`@Query()`	`req.query`
`@Headers()`	`req.headers`

While there are other decorators as well, these are the ones that are used most frequently.

Let us look at a few of them in more detail.

👉 The `@Param()` Decorator

It is a very common requirement to accept dynamic data as part of the incoming request.

For example, if you want to fetch the product information for product id 1, your request's path would be something like `GET /products/1`. Here, “1” is a route parameter token.

In this case, the route handler must be able to deal with dynamic parameters in the route. Ideally, you want to get access to the dynamic id and do something with it inside the request handler.

This is where the `@Param()` decorator comes in handy.

See the below example:

```
import { Controller, Delete, Get, Param, Post, Put, Req } from
"@nestjs/common";

@Controller('products')
export class ProductsController {

    @Get(':id')
    getProduct(@Param() params): string {
        console.log(params.id);
        return `This action returns the product details for product
id #${params.id}`;
    }
}
```

As you can see, `@Param()` decorates the method parameter known as `params`. Internally, this binds the route parameters as properties of the `params` object.

You can then easily access the `params` object within the request handler. For example, accessing the `params.id` property.

`Param` itself is imported from the `@nestjs/common` package.

There is another alternative way of extracting the route parameters.

Check out the snippet below:

```
import { Controller, Delete, Get, Param, Post, Put, Req } from
"@nestjs/common";

@Controller('products')
export class ProductsController {

    @Get(':id')
    getProduct(@Param('id') id: string): string {
        console.log(id);
        return `This action returns the product details for product
id #${id}`;
    }
}
```

In the above example, you pass the parameter's token name i.e. 'id' to the `@Param()` decorator to directly get access by name inside the method body.

In my opinion, this approach has better readability when compared to the previous one.

👉 The `Body()` Decorator

Path parameters are good for single fields. You don't want to be using them to transfer multiple fields. It will make the route path difficult to manage.

To deal with multiple fields, you must use the request body.

NestJS provides the `@Body()` decorator to extract the request body from the HTTP request object.

Check out the below example:

```
import { Body, Controller, Delete, Get, Param, Post, Put, Req } from
"@nestjs/common";
import { Request } from "express";

@Controller('products')
export class ProductsController {

    @Post()
    createProduct(@Body() productData: any): string {
        console.log(productData);
        return 'This endpoint creates a new product';
    }
}
```

The `@Body()` decorator is used for the `productData` variable.

At this point, you can simply define the type of `productData` variable as `any`. In a later section of the book, I will show you how you can define Typescript classes to map the request body to a specific schema.

Understanding the Response Object

NestJS automatically handles the response part depending on the underlying platform.

For example, when you return a JavaScript object, NestJS will automatically serialize it to JSON format. This is the default approach.

Also, in case of a request handler returning primitive types such as `string`, `boolean` or `number`, NestJS will just send the value without serializing it.

This makes it quite easy to handle responses. All you have to do is return the value and NestJS will take care of the rest.

However, you can also tap into the library-specific response object if needed. This can be done by using the `@Res()` decorator to inject the response object in the method handler.

See the below example:

```
import { Body, Controller, Delete, Get, Param, Post, Put, Req, Res }
from "@nestjs/common";
import { Request } from "express";

@Controller('products')
export class ProductsController {

    @Get()
    getAllProducts(@Req() request: Request, @Res() response): any {
        console.log(request.headers)
        response.status(200).send('This endpoint returns a list of
all products')
    }
}
```

In the above snippet, you are using the native response handling approach of Express.

> *When we use `@Res()` decorator to activate the library-specific option, the standard NestJS approach is automatically disabled for that particular route.*
>
> *However, we can still use both approaches.*
>
> *To do so, we have to set the passthrough option to true in the `@Res({ passthrough: true })` decorator.*
>
> *Using both approaches is useful when we want to use the platform-specific response object for setting a few special fields such as cookies or headers and still delegate the rest of response handling to NestJS.*

👉 Response Status Code

At first glance, it might seem that NestJS does not provide any way to set a specific HTTP Status Code for the response.

By default, `GET` methods return an HTTP Status Code of 200 and `POST` methods return 201.

However, despite automatically handling responses, NestJS also provides a special decorator `@HttpCode()` to provide a specific status code.

Check out the below example:

```
@Post()
@HttpCode(204)
createProduct(@Body() productData: any): string {
    console.log(productData);
    return 'This endpoint creates a new flight';
}
```

The `@HttpCode()` decorator is part of the `@nestjs/common` package.

An important point to note here is that the status code of a particular method is not usually static. It can depend on various factors. In such cases, you can use a library-specific response object using `@Res()` or throw an exception of some sort. More on that in later sections of the book.

👉 Response Headers

Similar to HTTP Status Code, you can also specify a custom response header.

You can either use the library-specific response object using `res.header()`. Alternatively, NestJS also provides a special decorator `@Header()`.

```
@Get()
@Header('Content-Type', 'application/json')
getAllFlights(@Req() request: Request, @Res() response): any {
    response.status(200).send('This endpoint returns a list of all
flights');
}
```

The `@Header()` decorator is imported from the `@nestjs/common` package.

What's Next?

The code discussed in this chapter is available on GitHub. You can download it and play around with it to practice more stuff.

To summarize things, you have now covered an extremely important topic of Controllers in NestJS.

With this topic under your belt, you can create **interfaces for your application** and expose useful functionality for the customers. In short, you have managed to build a working RESTful API even though it might not be considered very useful at this point in time.

In the next chapter, you are going to take another big step forward and learn how to handle business logic in your application using NestJS Providers.

So continue reading!

Chapter 3 - Handling Business Logic with Providers

Here's a recap of what you have learnt in the previous chapter:

- The role of a controller in NestJS
- Creating a NestJS Controller
- Understanding the request and response objects
- Request and response related decorators

Using this knowledge, you were able to put together an interface for your application. However, creating an interface is only half the journey. You need something meaningful to happen when someone accesses your application's interface.

This is where providers play a critical role in NestJS.

In this chapter, I'm going to show you how to handle business logic with NestJS providers.

Here's the plan for this chapter:

- Dependency injection with providers
- Creating your first NestJS provider
- Injecting the provider into a controller
- Registering a provider
- Testing the GET, POST, PUT & DELETE operations
- Understanding how the Nest IOC container works

What is a NestJS Provider?

Providers are a fundamental concept in NestJS. In my view, they are far more important than even controllers.

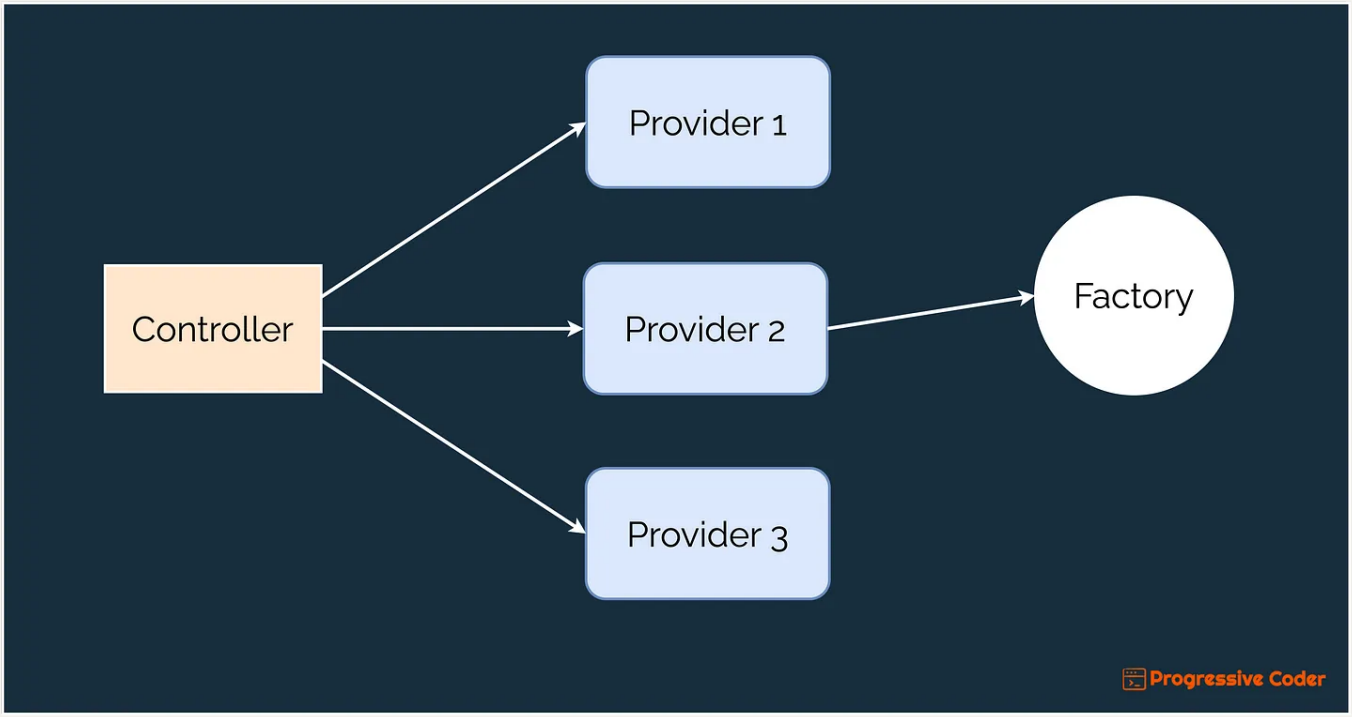
Think of **providers** as ingredients in a kitchen. Each ingredient has a different purpose. When a chef wants to cook a delicious meal, he/she will use a combination of ingredients.

In NestJS, providers play a similar role.

The idea behind a provider is that you inject it as a dependency within a class to establish a relationship. The job of creating the actual instances and wiring them up falls on the NestJS runtime system.

In the context of our demo example, we can have a **provider** that acts as the service for the `ProductsController`.

In fact, a single controller can utilize multiple providers as shown in the below illustration.



Creating a Service using a NestJS Provider

Let us understand the concept of providers with a code example.

Create a file named `products.service.ts` in the `src` directory with the below contents.

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

@Injectable()
export class ProductsService {
  private products = [];

  getAllProducts() {
```

```
    return this.products;
  }

  getProduct(id: number) {
    return this.products.find(product => product.id == id);
  }

  createProduct(product) {
    this.products.push(product);
    return product;
  }

  updateProduct(id: number, product) {
    const index = this.products.findIndex(p => p.id == id);
    if (index === -1) {
      return null;
    }
    this.products[index] = { ...this.products[index], ...product, id
};
    console.log(this.products)
    return this.products[index];
  }

  deleteProduct(id: number) {
    this.products = this.products.filter(product => product.id !==
id);
  }
}
```

In this example, the `ProductService` class is decorated with the `@Injectable()` decorator. This turns the class into a NestJS provider that can be injected into other components.

Basically, the `@Injectable()` decorator signals to NestJS that this class must be managed by the **Nest IOC container**. IOC stands for **Inversion of Control**.

But what's going on inside this class?

Think of this class as the backend for the `ProductsController` we saw in the previous chapter.

Within the class, we declare an array called `products` that acts as an extremely simple **in-memory** storage for the product details.

Then, we implement 5 operations for manipulating the products array: `getAllProducts`, `getProduct`, `createProduct`, `updateProduct`, and `deleteProduct`.

These methods perform the corresponding operations on the `products` array and return the result. Nothing too serious about them except for the array manipulation operations.

A short note on Inversion of Control

Let's say you're having a party and you want to make some delicious pizzas for your guests.

You start off by trying to make the pizzas yourself - you roll out the dough, spread the sauce, add the toppings and pop it in the oven. But before you know it, you're overwhelmed with guests, music, drinks and everything else that comes with hosting a party.

Making pizzas is the last thing on your mind now!

This is where **Inversion of Control** comes in. Instead of making the pizzas yourself, you hand over control to someone else - in this case, a pizza chef. The chef takes over and makes the pizzas for you, freeing you up to enjoy the party and take care of your guests.

In software development, **Inversion of Control works in much the same way**. Instead of your code being responsible for performing certain tasks, it hands over control to another component - a service, a library or a framework. This allows you to focus on the bigger picture and not get bogged down with the details of implementation.

So, in a nutshell, **Inversion of Control is like having a party and letting someone else make the pizzas so you can focus on having a good time!**

Injecting a Provider

Declaring a provider is one part of the puzzle. But as discussed earlier, you want to use it for something.

In our case, the provider takes care of the basic backend operations of our RESTful API interface. If you can use this provider within the controller, you can make an application that works on a superficial level.

Here's how you can use the `ProductsService` provider in the `ProductsController` class.

```
import { Body, Controller, Delete, Get, Param, Post, Put, Req, Res }
from "@nestjs/common";
import { ProductsService } from "./products.service";

@Controller('products')
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get()
    getAllProducts() {
        return this.productsService.getAllProducts();
    }

    @Get(':id')
    getProduct(@Param('id') id: number) {
        return this.productsService.getProduct(id);
    }

    @Post()
    createProduct(@Body() productData) {
        return this.productsService.createProduct(productData)
    }

    @Put(':id')
    updateProduct(@Param('id') id: number, @Body() productData) {
        return this.productsService.updateProduct(id, productData);
    }
```

```
    @Delete(':id')
    deleteProduct(@Param('id') id: number) {
        return this.productsService.deleteProduct(id);
    }
}
```

The `ProductsService` is injected within the controller using a constructor. Just by including it in the constructor, you are declaring and initializing the `productsService` member in the controller class.

The `productsService` member is basically an instance of the `ProductsService` class and has access to all the methods exposed by the class.

After this step, it is a simple matter of calling the appropriate method of the `ProductsService` in the various operations such as `createProduct()`, `updateProduct()` and so on.

Registering the Provider

The final piece of the puzzle is actually **registering the provider** in the module just like we did with the controller.

Head over to `app.module.ts` and add `ProductsService` to the `providers` array.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsController } from './products.controller';
import { ProductsService } from './products.service';

@Module({
  imports: [],
  controllers: [AppController, ProductsController],
  providers: [AppService, ProductsService],
})
export class AppModule {}
```

Without this step, the application would have no idea of how to create the `ProductsService` instance.

You are now in a position to test the code.

Execute the `npm run start` command and once the application starts up, you can use the below commands to test the API endpoints:

Creating a Product

```
$ curl -X POST \\
  <http://localhost:3000/products> \\
  -H 'Content-Type: application/json' \\
  -d '{
  "id": 1,
  "name": "New Product",
  "description": "A product description",
  "price": 50
}'
```

Fetching All Products

```
$ curl -X GET <http://localhost:3000/products>
```

Fetching a Single Product by Id

```
curl -X GET <http://localhost:3000/products/1>
```

Updating a Product

```
curl -X PUT <http://localhost:3000/products/1> -H "Content-Type:
application/json" -d '{"name": "Product 1 - Updated", "description":
"A updated description of product 1", "price": 200.0}'
```

Deleting a Product

```
curl -X DELETE <http://localhost:3000/products/1>
```

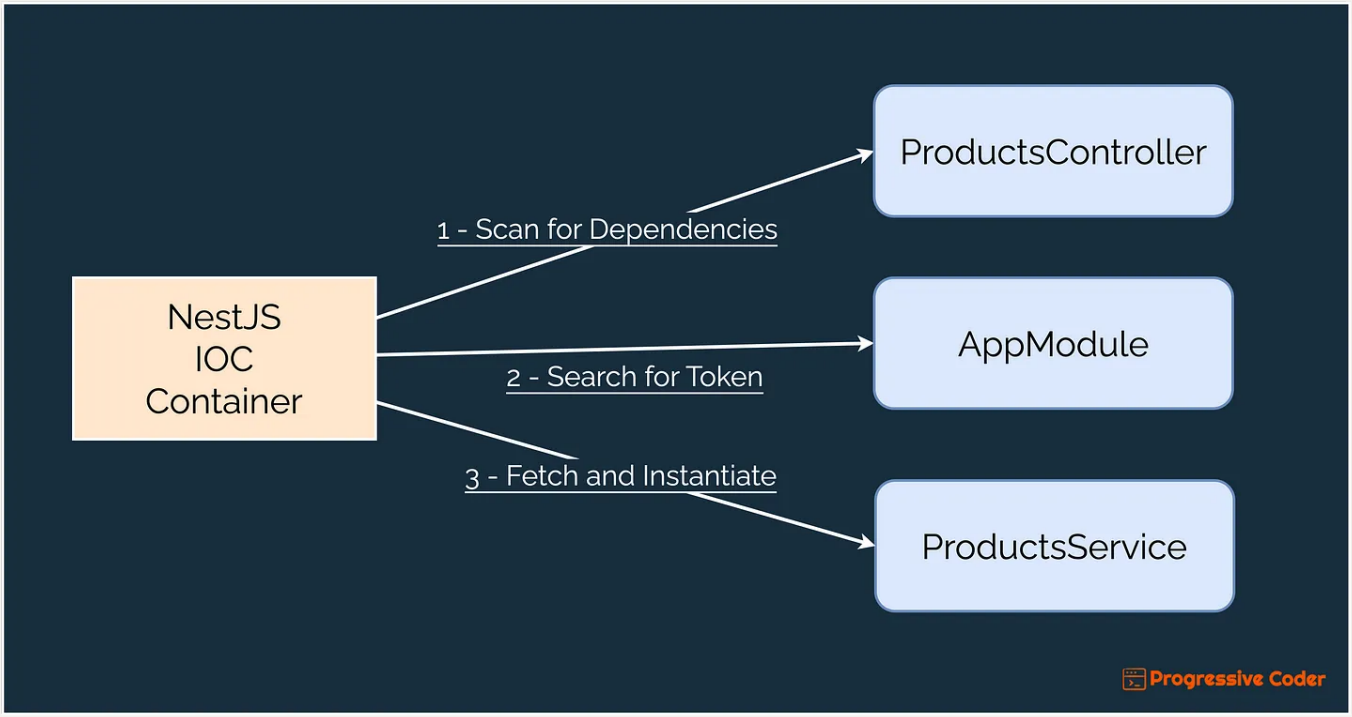
Nest IOC Container and Providers

What is happening under the hood when we are using NestJS Providers?

Let's look at each step in more detail:

- When the **Nest IOC Container** instantiates the `ProductsController` class, it looks at the dependencies of this class. Think of the IoC Container as a central registry for all the components in your application. The IoC Container creates instances of the components and manages their entire lifecycle.
- When the IOC Container finds that the `ProductsController` has declared the `ProductsService` class as a dependency using the constructor, it does a lookup for the `ProductsService` token.
- The IOC Container finds that the token points to the `ProductsService` class as per the registration within the module. At this point, NestJS creates an instance of the `ProductsService` class, caches it and returns it.
- In case the instance is already cached (if the class was already discovered), then NestJS returns the cached instance.

Here's an illustration showing the steps performed by the IOC Container.



Note that the dependency analysis performed by NestJS is **transitive** in nature.

For example, if `ProductsService` had multiple dependencies, NestJS will discover and resolve those as well. The entire scanning operation follows a bottom-up approach.

What's Next?

The code for this post is available on GitHub. You can download it and play around with it to practice the concepts discussed in this post.

To summarize this chapter, you have now covered an extremely important topic of **Providers in NestJS**.

After covering this topic, you have managed to separate the interface of your application from the business logic of storing and manipulating data. There are more things to discuss when it comes to providers, but I will be discussing them in a later chapter so as to have the maximum understanding.

In the next chapter, you are going to start organizing the code base by **structuring your application into modules**

So continue reading!

Chapter 4 - Structuring your Application with NestJS Modules

Here's a recap of what you have learnt in the previous chapter.

- Dependency injection with providers
- Creating your first NestJS provider
- Injecting the provider into a controller
- Registering a provider
- Testing the GET, POST, PUT & DELETE operations
- Understanding how the Nest IOC container works

With the help of providers, you were able to make your fledgling Product API a little bit more useful. Of course, it is still far from being a real application.

But we are moving closer with each chapter.

In this chapter, I'm going to show you how to organize your source code with NestJS modules

Here's the plan for this chapter:

- What is a NestJS Module?
- The role of `@Module()` decorator
- Creating a feature module for Products API
- Sharing functionalities between modules
- Re-exporting a module
- The use of global modules

What is a NestJS Module?

Literally, every programming language or framework out there has some concept of modules.

But why is there such a fascination with modules?

👉 Firstly, modules help **structure our code** into boxes by grouping related functionality.

This is a huge benefit when you have to build large applications. You can add more features to the code without making it confusing and unmanageable.

👉 Second, modules improve the **reusability** of your code.

If you have built a general-purpose module to do some common stuff like logging, you can simply reuse that module everywhere you want to log something. No need to reinvent the wheel every single time.

Ultimately, you can't do much without modules. They are like the essential blocks required to build a super-structure.

Talking about NestJS, a module is a way to organize related components, controllers and services into a single unit.

Each application has at least one module - **the root module**.

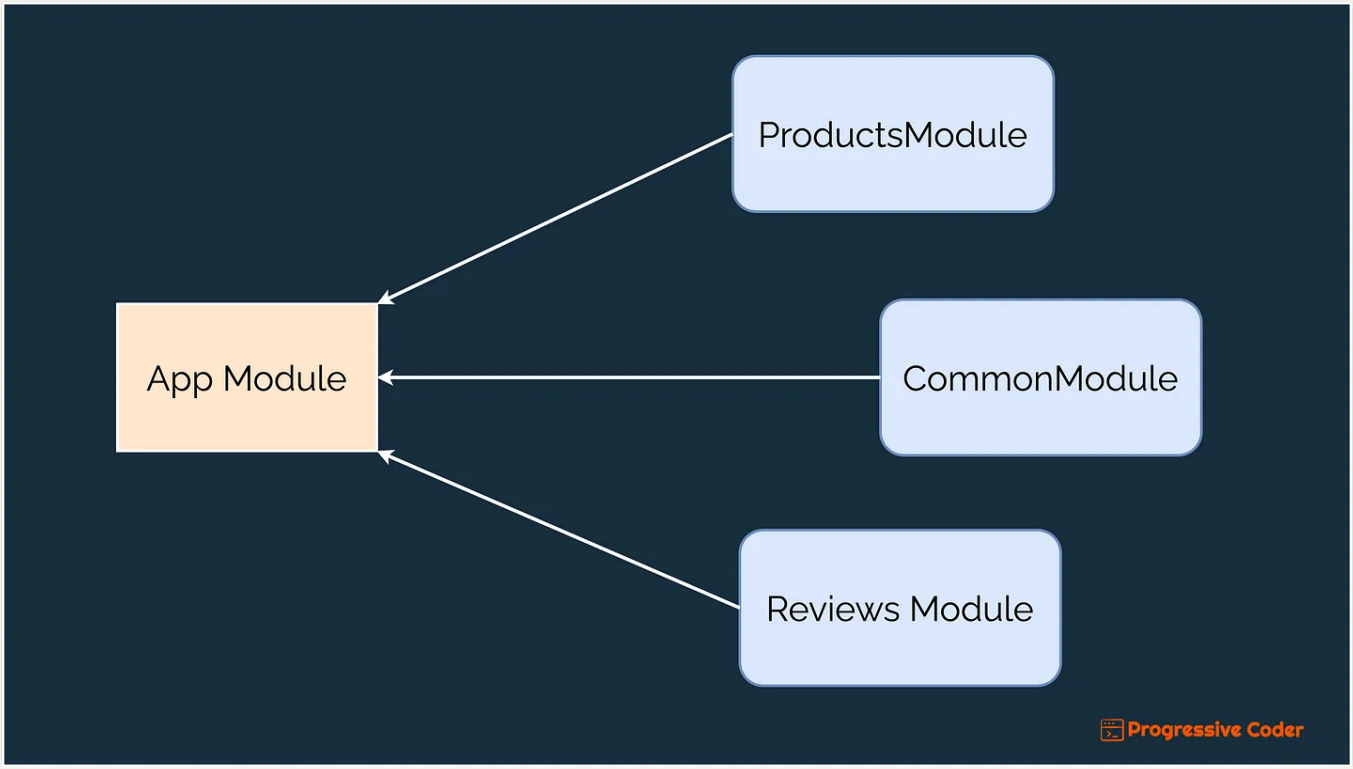
Here's the code for the root module (also known as the `AppModule`) from our demo application:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsController } from './products.controller';
import { ProductsService } from './products.service';

@Module({
  imports: [],
  controllers: [AppController, ProductsController],
  providers: [AppService, ProductsService],
})
export class AppModule {}
```

Think of the root module as the starting point of a forest trail. NestJS uses the root module to create an application graph to resolve relationships between modules and their dependencies.

A good way to determine the scope of a module is to map them to the key domain objects in your overall design.

Check the below illustration.



For example, in a service about **Products**, you might have domain objects for basic product information, details about its features, reviews pricing and so on. Each of these objects and their functionalities could be encapsulated into an individual module.

Again, there is **no one-size-fits-all** answer when determining the scope of a module. You can make them as big or as small as you want. Ultimately, it boils down to the kind of structure you are comfortable with.

The `@Module()` decorator

To define a module in NestJS, you need to use a special decorator `@Module()`.

This decorator tells NestJS whether a particular class is a module and makes it available for injection.

You can configure the module by passing an object to the `@Module()` decorator.

Here are the properties of this object:

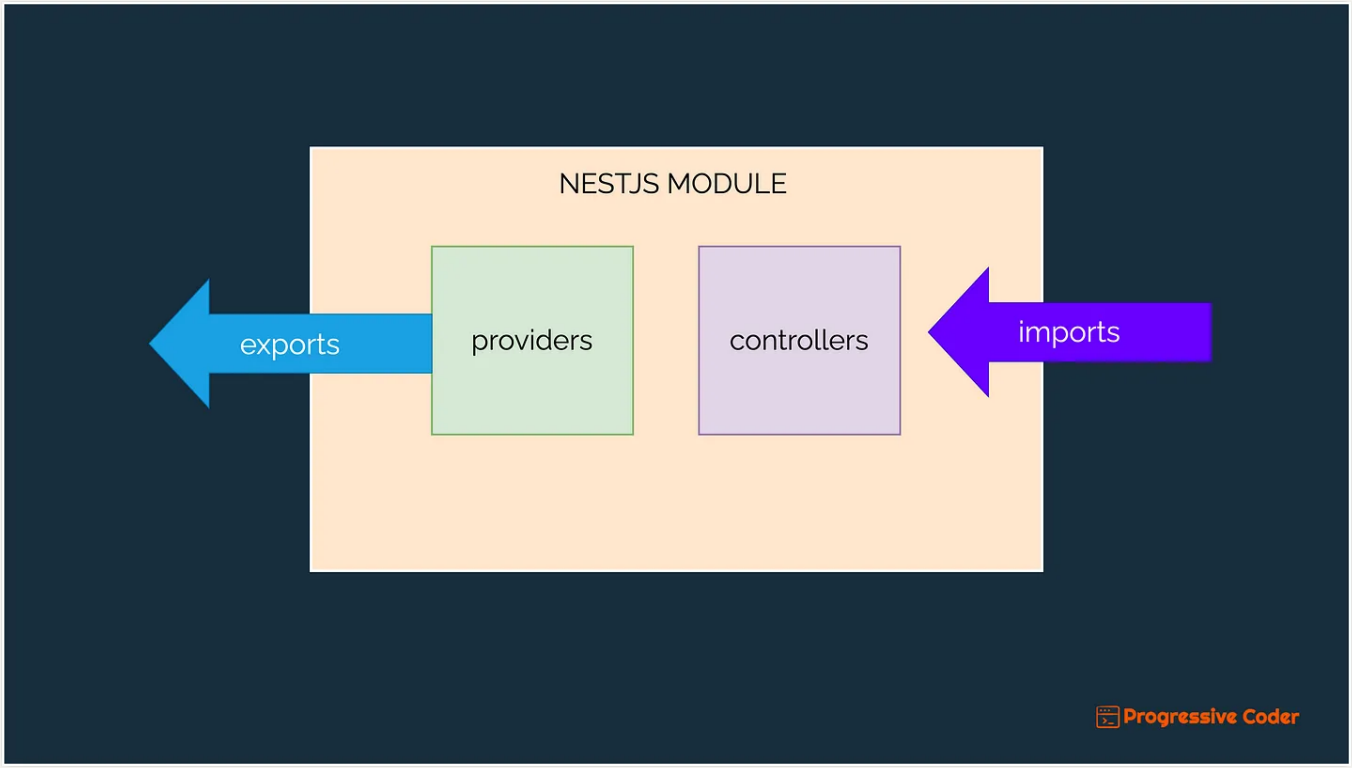
👉 `providers` - List of providers declared within the module. Basically, the idea is that these providers can be shared within the context of the module and NestJS has to instantiate them.

👉 `controllers` - List of controllers declared within the module. Again, NestJS makes sure to instantiate these controllers at startup time.

👉 `imports` - A module can use the functionality exposed by another module. This array contains the list of other modules that are required by this particular module.

👉 `exports` - Think of it as the interface or API of the module. Here, we set the list of `providers` that are exported by this module. Other modules can use these exported providers after importing the module.

Check out the below illustration that tries to depict the structure of a module.



Don't worry if the above definitions still sound confusing. You will be seeing them in action in the upcoming sections of this post.

Creating a feature module

Time to create your first module.

Within the demo application's source directory, you need to create a new folder named `products`. Within this folder, create a file named `products.module.ts`.

```
import { Module } from "@nestjs/common";
import { ProductsController } from "./products.controller";
import { ProductsService } from "./products.service";

@Module({
    controllers: [ProductsController],
    providers: [ProductsService]
})
export class ProductsModule {}
```

Notice the object within the `@Module()` decorator. It contains the list of `controllers` and `providers` within the scope of the module.

Since you want this module to be concerned with product-related functionality, you must move the `ProductsController` and `ProductsService` within the `products` folder. This helps with better code organization.

After declaring the module, you also need to import this module into the root module i.e. the **AppModule** (`app.module.ts`)

Check the `imports` array in the `AppModule`. Based on the list of imports, NestJS discovers the `ProductsModule` and instantiates the necessary classes.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';

@Module({
  imports: [ProductsModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

At this point, you can start the application using the `npm run start` command.

If the module import was successful, you should see that NestJS has successfully mapped the GET, POST, PUT and DELETE endpoints for `/products`. This means that NestJS was able to successfully discover the route handlers declared within the `ProductsModule`.

```
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [NestFactory] Starting
Nest application...
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [InstanceLoader]
AppModule dependencies initialized +37ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [InstanceLoader]
ProductsModule dependencies initialized +0ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RoutesResolver]
AppController {/}: +3ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RouterExplorer] Mapped
{/, GET} route +1ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RoutesResolver]
ProductsController {/products}: +0ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RouterExplorer] Mapped
{/products, GET} route +1ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RouterExplorer] Mapped
{/products/:id, GET} route +0ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RouterExplorer] Mapped
{/products, POST} route +0ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RouterExplorer] Mapped
{/products/:id, PUT} route +1ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [RouterExplorer] Mapped
{/products/:id, DELETE} route +0ms
[Nest] 1600  - 07/02/2023, 08:12:55     LOG [NestApplication] Nest
application successfully started +1ms
```

Sharing module functionalities

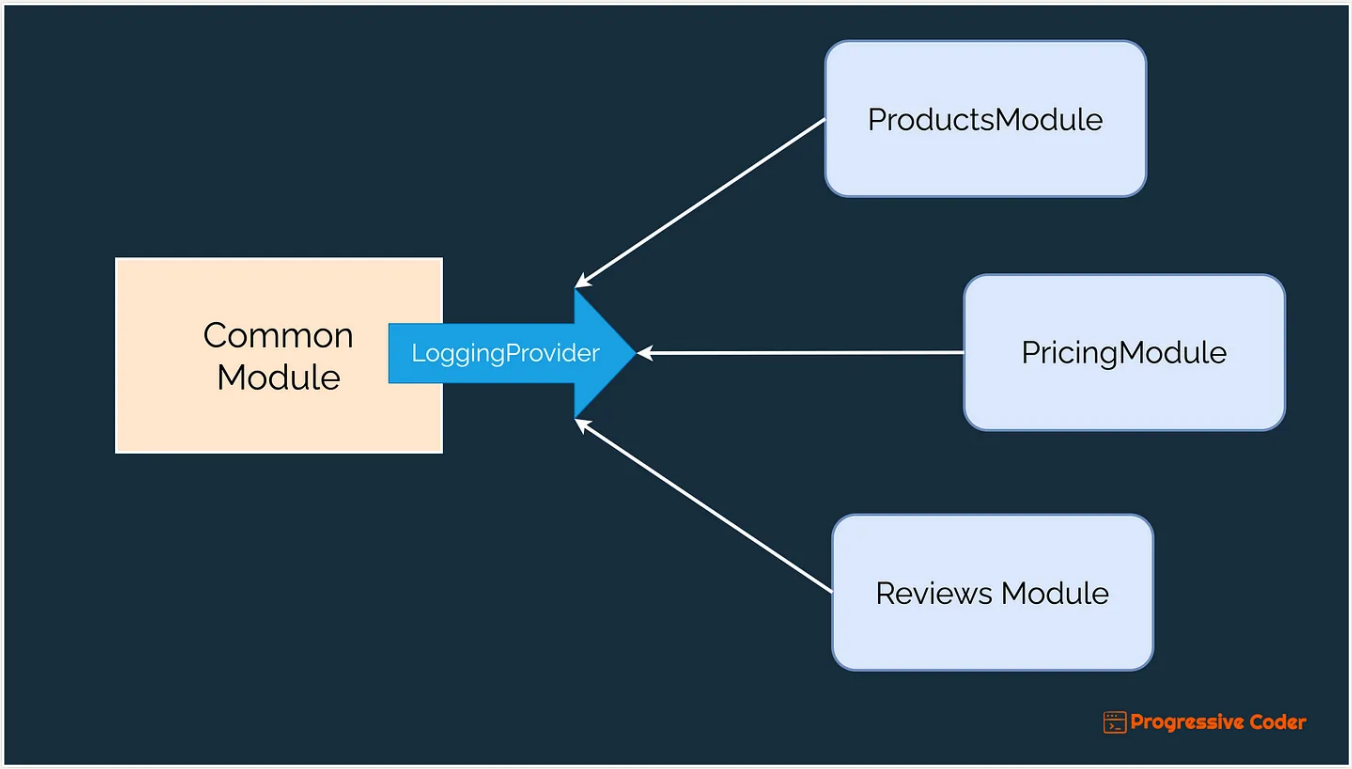
NestJS modules are singletons by default.

What does it mean?

It means you can share the same instance of any provider belonging to a particular module between multiple other modules.

This is a very useful property when you want to share reusable functionality (such as logging, configuration and so on) within the scope of an entire application.

See the below illustration where a common module can expose a logging service that can be consumed by other modules in the application.



However, though the module is automatically shared by default, you need to specifically export the provider that has to be exposed outside a module's scope.

For example, if you wanted that the `ProductsService` was available to other modules within the application, you need to export the `ProductsService` provider from the `ProductsModule`.

Here's how you can do it:

```
import { Module } from "@nestjs/common";
import { ProductsController } from "./products.controller";
import { ProductsService } from "./products.service";

@Module({
    exports: [ProductsService],
    controllers: [ProductsController],
    providers: [ProductsService]
})
export class ProductsModule {}
```

Just by adding the `ProductsService` to the `exports` array, you are making this service available to the entire application. It's like exposing an interface from the `ProductsModule`.

Now, any module within the application that imports the `ProductsModule` can access `ProductsService`.

Re-exporting Modules in NestJS

Another interesting feature provided by the NestJS module system is re-exporting imported modules.

As your application gets more complicated, you can build wrapper modules to encompass lower-level modules. For example, you might have a `ProductsWrapperModule` module that wraps the `ProductsModule`.

In this case, you can import `ProductsModule` into the `ProductsWrapperModule` and then, re-export it.

See the below example:

```
@Module({
  imports: [ProductsModule],
  exports: [ProductsModule],
})
export class ProductsWrapperModule {}
```

Making a Module Global

This one is not my favourite approach. But it might come in handy sometimes when you are tired of importing a particular module or a set of modules everywhere.

You can declare a module as global by using the `@Global()` decorator.

```
import { Module, Global } from "@nestjs/common";
import { ProductsController } from "./products.controller";
import { ProductsService } from "./products.service";

@Global()
@Module({
    exports: [ProductsService],
    controllers: [ProductsController],
    providers: [ProductsService]
})
export class ProductsModule {}
```

In this case, the `ProductsService` will be available to all the modules that need to inject the service without importing the `ProductsModule` in their `imports` array.

Global modules should be registered only once by the root module.

What's Next?

The code for this post is available on GitHub. You can download it and play around with it to practice the concepts discussed in this post.

To summarize this chapter, you have now covered a fundamental topic of NestJS modules.

Modules are an important way to structure and organize our applications based on functionality. In my view, you should always try to carve out appropriate modules based on the scope of the application.

In the next chapter, I'm going to talk about dynamic modules and how they can make your application more flexible.

So continue reading!

Chapter 5 - Creating Dynamic Modules

Here's a recap of what you have learnt in the previous chapter:

- What is a NestJS Module?
- The role of `@Module()` decorator
- Creating a feature module for Products API
- Sharing functionalities between modules
- Re-exporting a module
- The use of global modules

With the help of modules, you isolated the Products-related functionality into its own module and build a better structure for your NestJS application.

In this chapter, I'm going to talk about **dynamic modules in NestJS**.

Here are the things you will learn:

- Why dynamic modules?
- How to use a dynamic module in NestJS?
- How to create a dynamic module and customize the provider?

Why dynamic modules?

Static modules are sufficient for most use cases.

But they have a limitation.

Static modules stay the same, no matter which module is consuming them. The consumer module cannot influence the behaviour of the static module's providers.

But why do you need to influence the behaviour of a static module?

A simple use case could be the need to have different database parameters for different deployments.

You can also think of this as a **plugin-based approach** to building software. A customizable generic plugin can be used by many systems just by tweaking the configuration parameters.

In a typical NestJS application, you can use dynamic modules to handle stuff like configurations. These configurations can be anything from database settings to file system details and so on.

The consuming module can simply call an API provided by the dynamic module to tweak the configuration according to its requirement.

How to use a dynamic module in NestJS?

Let's create a sample dynamic module that can help understand the concept in more detail.

However, before actually creating a module, you must understand how to use a dynamic module.

Below is an example of how to use a **simple static module**.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';

@Module({
  imports: [ProductsModule, ConfigurationModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

As you can see, you are importing the `ConfigurationModule` into the `AppModule`. This approach is also known as static module binding.

Let's look at how to use a dynamic module.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';

@Module({
  imports: [ProductsModule,
    ConfigurationModule.register({db_url:
'http://localhost:27017'})],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Can you spot the difference between the static and dynamic approaches?

In the dynamic approach, the `ConfigurationModule` is a normal typescript class with a method named `register()`.

Basically, `register()` is a static method attached to the `ConfigurationModule` class. By convention, it is good practice to call this method `register()` or `forRoot()`. Technically, it can have any arbitrary name.

You accept some arguments in the `register()` method. You can pass these arguments as a configuration object. In the above example, the configuration object contains the property `db_url`.

🚀 Important Point

Every item within the `imports` array needs to be a module in order to make it available. Therefore, we can conclude that the `register()` method also returns a module.

In fact, the `register()` method actually returns a **dynamic module** that we can configure based on the input configuration object.

> *Dynamic Modules basically provide an API for importing one module to another. This API helps customize the properties and behaviour of the module while importing. This is different from static bindings where no such customization is possible.*

Creating a dynamic module

Within our project, you need to create a separate folder named `configuration` for the `ConfigurationModule`.

Within this folder, create a file named `configuration.module.ts` that contains the module definition along with the code for the `register()` method.

Check out the below snippet.

```
import { DynamicModule, Module } from "@nestjs/common";
import { ConfigurationService } from "./configuration.service";

@Module({})
export class ConfigurationModule {
    static register(options): DynamicModule {
        return {
            module: ConfigurationModule,
            providers: [
                {
                    provide: 'CONFIGURATION_OPTIONS',
                    useValue: options
                },
                ConfigurationService
            ],
            exports: [ConfigurationService]
        }
    }
}
```

Basically, you are defining the static `register()` method.

This method takes the `options` object as input and returns an instance of `DynamicModule`. The `DynamicModule` has the same properties you might have noticed within the `@Module()` decorator. However, you pass a blank object to the `@Module()` decorator.

Also, notice the `providers` array.

Apart from the `ConfigurationService`, you also have something known as `CONFIGURATION_OPTIONS`. Basically, you are assigning the `options` object to the `useValue` attribute. We will be using this in the `ConfigurationService`.

Below is the code for the `ConfigurationService` in the `configuration.service.ts` file.

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

@Injectable()
export class ConfigurationService {
    private readonly db_url;

    constructor(@Inject('CONFIGURATION_OPTIONS') private options) {
        this.db_url = options.db_url;
    }

    getDBUrl(): string {
        return this.db_url;
    }
}
```

The first thing to note here is the constructor. You inject the `CONFIGURATION_OPTIONS` within the constructor. This helps us follow the principle of **Dependency Injection**.

Next, you assign the `db_url` from the options object to the private `db_url` of the service class.

For demonstration, you can implement a method named `getDBUrl()`. This method simply returns the current value of the `db_url`.

A real application might use the `db_url` value for database configuration. But we are not doing that right now. We are only trying to understand the anatomy of a dynamic module.

You can now use the `ConfigurationService` within the `AppModule`. More precisely, you use it in the `AppController`.

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';
import { ConfigurationService } from
'./configuration/configuration.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService,
    private readonly configService: ConfigurationService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }

  @Get("/config")
  getConfiguration(): string {
    return this.configService.getDBUrl();
  }
}
```

Here, you inject the `ConfigurationService` within the `AppController` class. Next, you make a call to the `getDBUrl()` method within the `getConfiguration` route handler.

When you start the application and make a request to `http://localhost:3000/config`, the output will be `http://localhost:27017`.

Basically, the dynamic `ConfigurationModule` returns the injected value. You can tweak the dynamic module from the calling module based on your requirements.

What's Next?

The code for this chapter is available on GitHub. You can download it and play around with it to practice the concepts discussed in this post.

To summarize this chapter, you have now learnt how to use dynamic modules in NestJS.

While we saw a very rudimentary example, it was more of an attempt to understand the concept of dynamic modules. These types of modules will come in extremely useful in some of the more complicated requirements when you need to deal with real database configurations.

Chapter 6 - Exception Handling in NestJS

Here's a recap of what you have learnt in the previous chapter:

- The need for dynamic modules
- How to use a dynamic module?
- How to create a dynamic module and customize the provider?

Time to move ahead with the next important fundamental concept in NestJS. And that is exception handling.

Exceptions are an incredibly important part of any real-world application. Proper exception handling is often the difference between a professionally built and an amateurish application.

In this chapter, I'm going to cover the **in-built exception handling in NestJS**.

Here's the plan for this chapter:

- Understanding the global exception filter
- How to throw standard exceptions?
- How to customize the exceptions?

The Global Exception Filter in NestJS

NestJS comes pre-packaged with an in-built exceptions layer.

The job of this exceptions layer is to process any exceptions that are unhandled within the application.

In a way, this layer acts as a safety net. If you don’t handle an exception in our application code, the exceptions layer catches it and sends an appropriate user-friendly response to the client.

All of this catching and handling exceptions is done by the **global exception filter**.

The filter is programmed to handle exceptions of type `HttpException` and all of its subclasses. The `HttpException` is a class provided by NestJS as part of the `@nestjs/common` package.

How to throw standard exceptions?

Here's an example demonstrating how to throw an exception from our NestJS application.

```
import { Controller, Get, HttpException, HttpStatus } from
"@nestjs/common";
import { ProductsService } from "./products.service";

@Controller('products')
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get('admin')
    getAdminDetails() {
        throw new HttpException('Forbidden endpoint',
HttpStatus.FORBIDDEN);
    }
}
```

The `getAdminDetails()` is the route handler for the admin route. Within the function `getAdminDetails()`, you throw an `HttpException`.

When you call the endpoint `http://localhost:3000/products/admin`, you get the below response.

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

The response body consists of the `statusCode` value and the `message` value that you provided in the constructor of the `HttpException` class.

As you may have guess, the `HttpException` constructor takes two **required** arguments as input:

- `response` - This defines the JSON response body and can contain a string or object value. In our example, we hard-code the response body to the string value 'Forbidden endpoint'.
- `status` - This argument defines the HTTP Status Code. In the code example, you set the status code to 403 using the enum `HttpStatus.FORBIDDEN`.

Both the `HttpException` class and `HttpStatus` enum are provided by the `@nestjs/common` package.

However, the constructor of the `HttpException` class also takes a third optional argument. This argument is known as `options` and you can use it to provide an error cause.

The cause object is not serialized into the response object, but it can be used for logging purposes.

See the below example:

```
import { Controller, Get, HttpException, HttpStatus } from
"@nestjs/common";
import { ProductsService } from "./products.service";

@Controller('products')
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get('admin')
    getAdminDetails() {
        try {
            this.productsService.getAllProducts()
        } catch (err) {
```

```
            throw new HttpException({
                status: HttpStatus.FORBIDDEN,
                error: 'You are forbidden to access the endpoint',
              }, HttpStatus.FORBIDDEN, {
                cause: err
              });
        }
    }
}
```

Notice that you are passing an object instead of a string in the `response` attribute of the `HttpException` constructor. Also, you are adding the optional property `cause` and assigning it the value of the `err` object generated by the `try-catch` block.

If you access the endpoints now, you will get the below response in case of an error.

```
{
   "status": 403,
   "message": "You are forbidden to access the endpoint"
}
```

How to customize the exceptions?

You don't need to write custom exceptions frequently because the **built-in Nest HTTP exception** is quite sufficient for most requirements.

However, you may need to create customized exceptions sometimes for some specialized requirements. In such cases, it's still a good practice to use the existing exception hierarchy.

In other words, any custom exceptions you create should inherit from the base `HttpException` class. This allows NestJS to recognize your custom exceptions and handle the error responses automatically.

Check the below example:

```
export class ForbiddenException extends HttpException {
  constructor() {
    super('You are forbidden to access the endpoint',
HttpStatus.FORBIDDEN);
  }
}
```

The `ForbiddenException` class extends the base `HttpException` class.

Now, you can directly use this exception class in your application code.

```
@Controller('products')
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get('admin')
    getAdminDetails() {
        throw new ForbiddenException();
    }
}
```

What's Next?

To summarize the chapter, you have learnt how to use the in-built exception handling in NestJS.

You looked at basic exception handling using the `HttpException` class.

However, NestJS also provides **exception filters** that can allow you to gain complete control over exception handling in your application.

I will be talking about them in the next chapter.

So continue reading!

Chapter 7 - Exception Filters in NestJS

Here's a recap of what you have learnt in the previous chapter:

- Understanding the global exception filter
- How to throw standard exceptions?
- How to customize the exception for better control?

In-built exception handling is a pretty robust way of handling exceptions in NestJS.

However, if you want more control over the exceptions layer, you need to use exception filters.

I will be covering them in this chapter.

Here's the plan:

- What is an Exception Filter?
    - The role of generic interface
    - The `catch()` method
- How to create a Custom Exception Filter?
- Understanding the `ArgumentsHost`
- How to use the Custom Exception Filter?
    - method-level
    - controller-level
    - global

What is an Exception Filter?

NestJS has a pretty robust in-built exception handler.

Then, why do you need the exception filter?

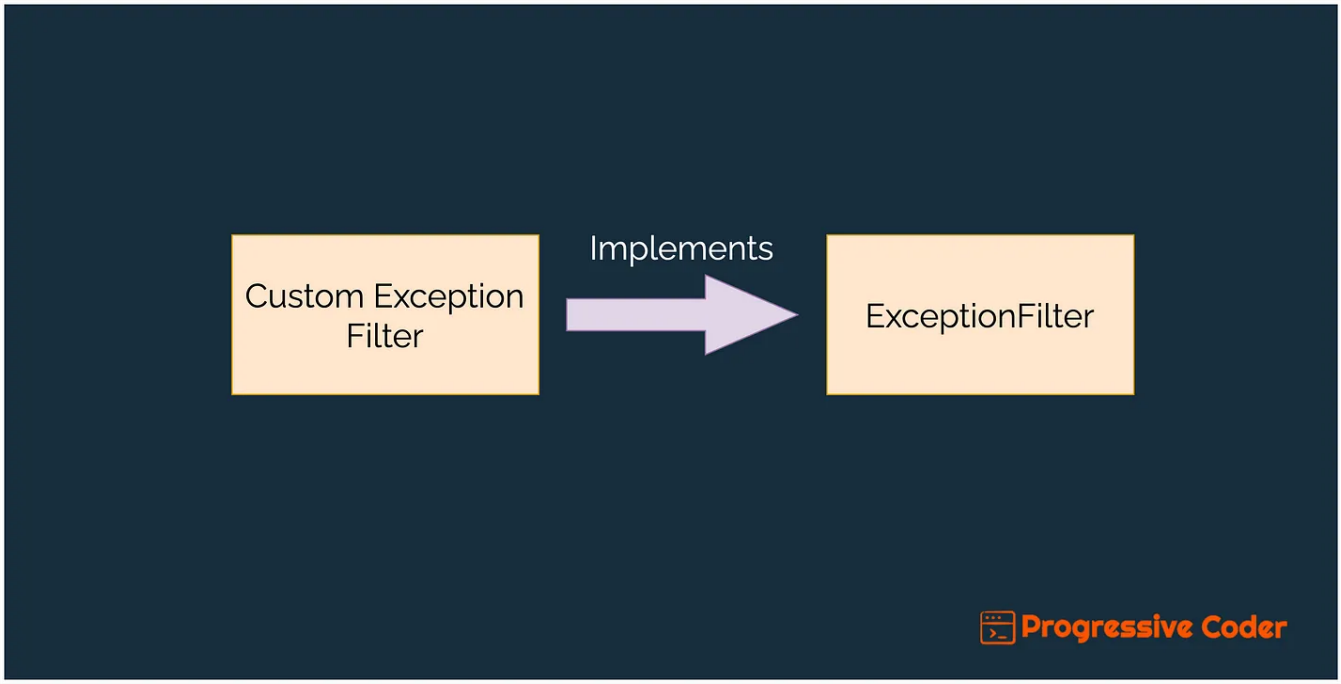
Well, the exception filter gives us complete control over the exceptions thrown by our application.

What kind of control are we talking about?

Imagine you want your exception response to use a specific JSON schema that contains more details such as the timestamp or the request URL. Maybe, you want to perform some sort of logging before sending the exception response.

The exception filter lets you control the complete exception flow and also the response sent back to the client.

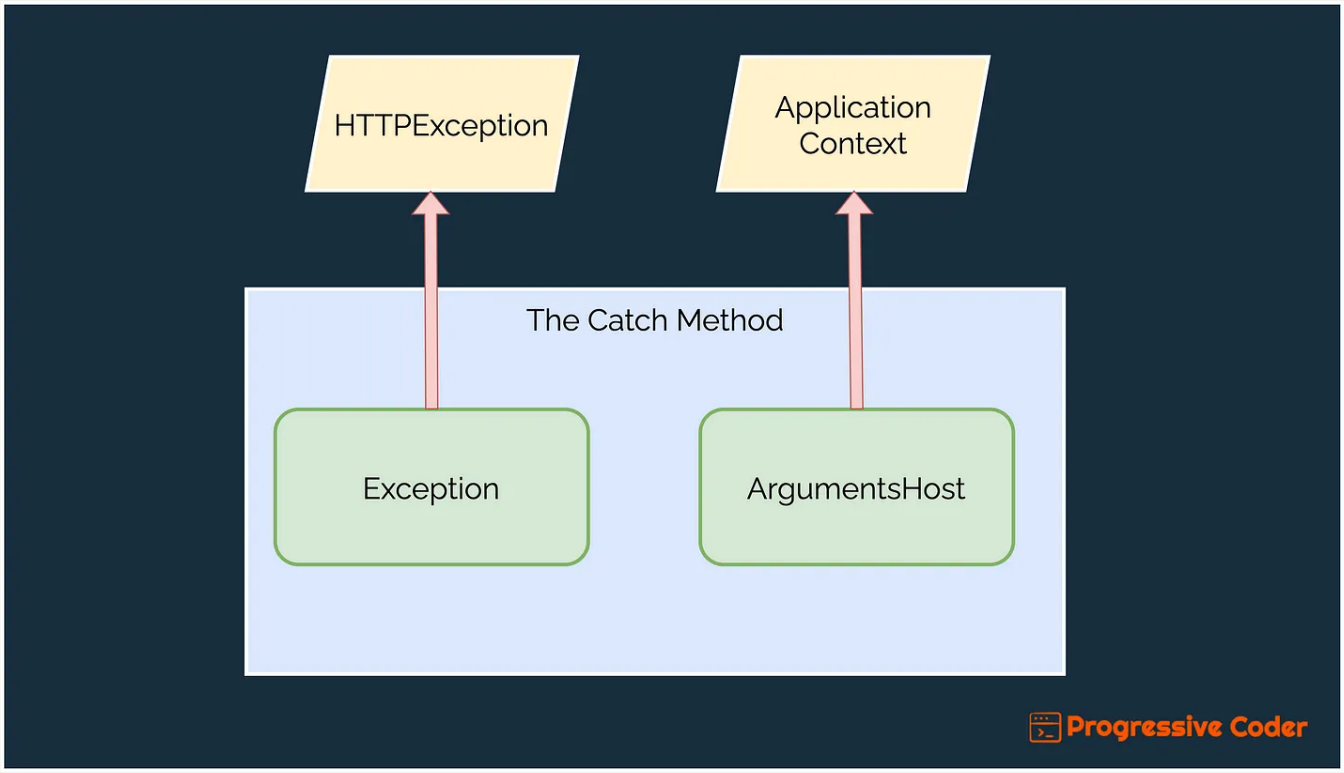
In order to create a **custom exception filter**, we need to implement a generic interface called `ExceptionFilter`.



This basically means implementing the `catch()` method provided by the `ExceptionFilter` interface.

The `catch()` method takes two arguments as input:

- The `Exception` object - This points to the exception object that is being processed
- The `ArgumentsHost` object - This is a utility object that allows us to access the arguments passed to a handler and obtain references to the underlying request and response objects



How to create a custom exception filter?

To understand the role of exception filters better, you need to create a custom exception filter from scratch.

Within your project's `src` directory, create a folder named `exceptions`.

Inside this folder, create a file named `custom-exception.filter.ts` with the below contents.

```
import { ArgumentsHost, Catch, ExceptionFilter, HttpException } from
"@nestjs/common";
import { Request, Response } from "express";

@Catch(HttpException)
export class CustomExceptionFilter 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
            })
    }
}
```

Let's understand what's going on here:

- The class `CustomExceptionFilter` implements the `ExceptionFilter` interface.
- The class has a decorator `@Catch()`. This decorator takes the type of exception being handled by this exception filter as input. In this case, you are trying to handle an exception of type `HttpException`.
- To satisfy the interface requirement, you need to implement the `catch()` method. As discussed earlier, this method takes two arguments as input - `exception` and `host`.
- You get access to the current context using the `switchToHttp()` method available with the `host` object. More on that in the next section of this chapter.
- With the `ctx` object, you get the reference to the `request` and `response` objects of the underlying platform. Since you are using Express as the platform in this book, these objects are nothing but express request and response objects.
- Finally, you use the `response` object to set a custom JSON schema.
- Apart from the status code, you also add a new field for the timestamp and the `request.url`. With this change, the response object gives us a better idea of when and where the exception was generated.

Understanding the `ArgumentsHost` Object

NestJS supports different types of applications such as:

- HTTP server-based
- Microservices

- GraphQL
- WebSockets

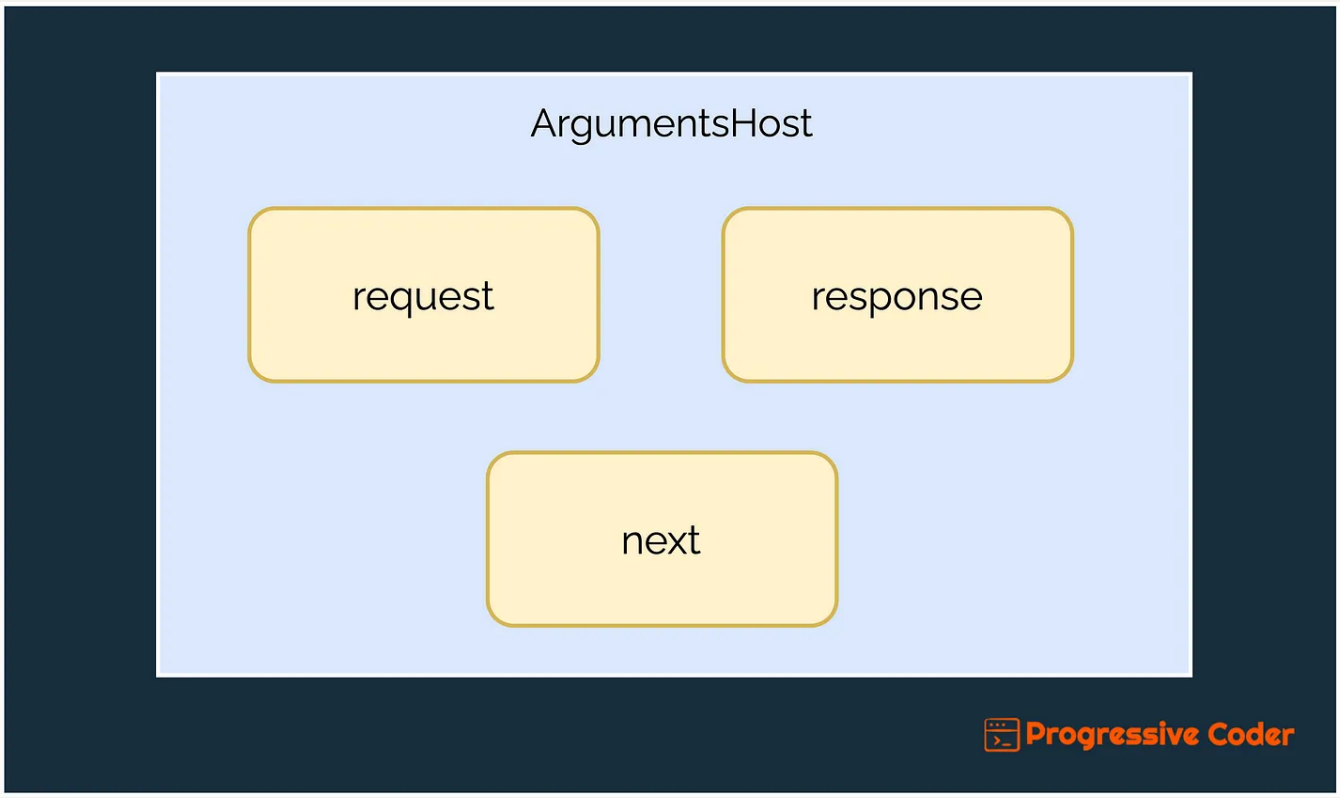
Each type of application has its own application context.

The `ArgumentsHost` class provides methods for retrieving the arguments being passed to a handler in a particular context. In other words, this class acts as a container for the context information associated with a particular request.

For example, when a request is made to a controller method, NestJS creates an instance of the `ArgumentsHost` object.

For an HTTP server-based application, this object contains information about the current request's context, including the request object, response object and any errors that may have occurred.

Check the below illustration that shows the composition of the `ArgumentsHost` object for an HTTP application using the Express platform.



At this point, you might have one particular question.

👉 Why the need for `const ctx = host.switchToHttp()`?

The `switchToHttp()` is a helper method available with the `host` object.

This method returns an `HttpArgumentsHost` object that is appropriate for the HTTP application context. It provides two helper methods `getResponse()` and `getRequest()` and makes it easy for you to get access to the underlying express objects.

The alternative to **not** using `switchToHttp()` is to obtain the request and response objects as below:

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

This ties our implementation to the HTTP application context and makes our code less reusable.

Hence, it is always considered good practice to use the appropriate switch method to obtain a reference to the required application context.

How to use the Custom Exception Filter?

Now that you are clear about how to declare a custom exception filter, it's time to understand how to use it.

First, create an exception class in a file named `forbidden.exception.ts`.

```
import { HttpException, HttpStatus } from "@nestjs/common";

export class ForbiddenException extends HttpException {
    constructor() {
        super('You are forbidden to access the endpoint',
HttpStatus.FORBIDDEN);
    }
}
```

Just like in the previous chapter, `ForbiddenException` extends `HttpException`. This will be your new exception class.

There are three ways you can use the exception filter in your NestJS application:

👉 Method-Level

In this approach, you bind the exception filter to a particular method within the controller.

See the below example:

```
@Controller('products')
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get('admin')
    @UseFilters(new CustomExceptionFilter())
    getAdminDetails() {
        throw new ForbiddenException();
    }
}
```

The `@UseFilters()` decorator (available as part of the `@nestjs/common` package) lets you attach the `CustomExceptionFilter` to the `getAdminDetails()` request handler.

Alternatively, you can also pass the name of the class instead of an instance to the `@UseFilters()` decorator. See below:

```
@Controller('products')
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get('admin')
    @UseFilters(CustomExceptionFilter)
    getAdminDetails() {
        throw new ForbiddenException();
    }
}
```

It is always better to pass a class instead of an instance. NestJS automatically takes care of instantiating the class based on the principles of Dependency Injection.

If you access `http://localhost:3000/products/admin`, you get the below JSON response.

```
{
  "statusCode": 403,
  "timestamp": "2023-02-28T02:46:22.780Z",
  "path": "/products/admin"
}
```

As you can see, the `timestamp` and `path` fields are available in the response object. This is because of the `CustomExceptionFilter()` you configured in the previous section.

👉 Controller-Level

Alternatively, you can also apply the `CustomExceptionFilter()` at the controller level.

```
@Controller('products')
@UseFilters(CustomExceptionFilter)
export class ProductsController {

    constructor(private readonly productsService: ProductsService) {}

    @Get('admin')
    getAdminDetails() {
        throw new ForbiddenException();
    }
}
```

This will apply the exception filter for every route within the `ProductsController`.

👉 Global-Level

Lastly, you can also apply the exception filter at a global level for the entire application

See the below code from the `main.ts` file.

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { CustomExceptionFilter } from './exceptions/custom-
exception.filter';

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

What's Next?

The code for this chapter is available on GitHub. You can download it and play around with it to practice the concepts discussed in this chapter.

To summarize this chapter, you have looked at **exception filters in NestJS** and how you can use them to control the exceptions layer.

In the next chapter, you are going to learn how to generate **API Documentation in NestJS using Swagger**.

So continue reading!

Chapter 8 - API Documentation in NestJS with Swagger

Here's a recap of what you have learnt in the previous chapter:

- What is an Exception Filter?
- How to create a Custom Exception Filter?
- Understanding the `ArgumentsHost`
- How to use the Custom Exception Filter?

In this chapter, I'm going to cover the important but often ignored topic of API documentation with Swagger in your NestJS application.

Here's the plan for this chapter:

- What is Swagger?
- Initializing the Swagger module
- Setting the Proper API Tags

The idea is to start with Swagger integration for our NestJS application. This can help you document your API endpoints in a much better way from the beginning.

What is Swagger?

An API is like a waiter taking your order at a restaurant.

It's the interface between your app or website and the server it's communicating with. And just like a waiter needs to be clear and precise with their order-taking, an API needs to be clear and precise with its communication.

That's where Swagger comes in.

Basically, Swagger is a tool that helps you define and document your API in a standardized way.

It follows the **OpenAPI specification** which is an industry standard for RESTful API design. This makes it widely adopted in the industry as the default API documentation approach.

> *The **OpenAPI** specification is a language-agnostic definition format used to describe RESTful APIs. You can read more about it at this link.*

Swagger provides a clear and easy-to-read format for describing the different endpoints of your API, the data they accept and return and how to interact with them.

For example, let's say you are building an API for a weather app.

With Swagger, you can define the endpoint for getting the current weather in a particular location. You can specify the expected input, such as the latitude and longitude of the location and the expected output, such as the temperature and weather conditions.

This makes it easy for other developers to understand how to interact with your API and build their own apps that use it.

Initializing Swagger for a NestJS application

NestJS provides a special module that lets you generate Swagger documentation for your application.

To use the module, you need to install the below package.

```
$ npm install --save @nestjs/swagger
```

Next, you need to configure Swagger for the application using the `SwaggerModule` class that was installed as part of the package.

To do so, open the `main.ts` file of your application and make the below changes:

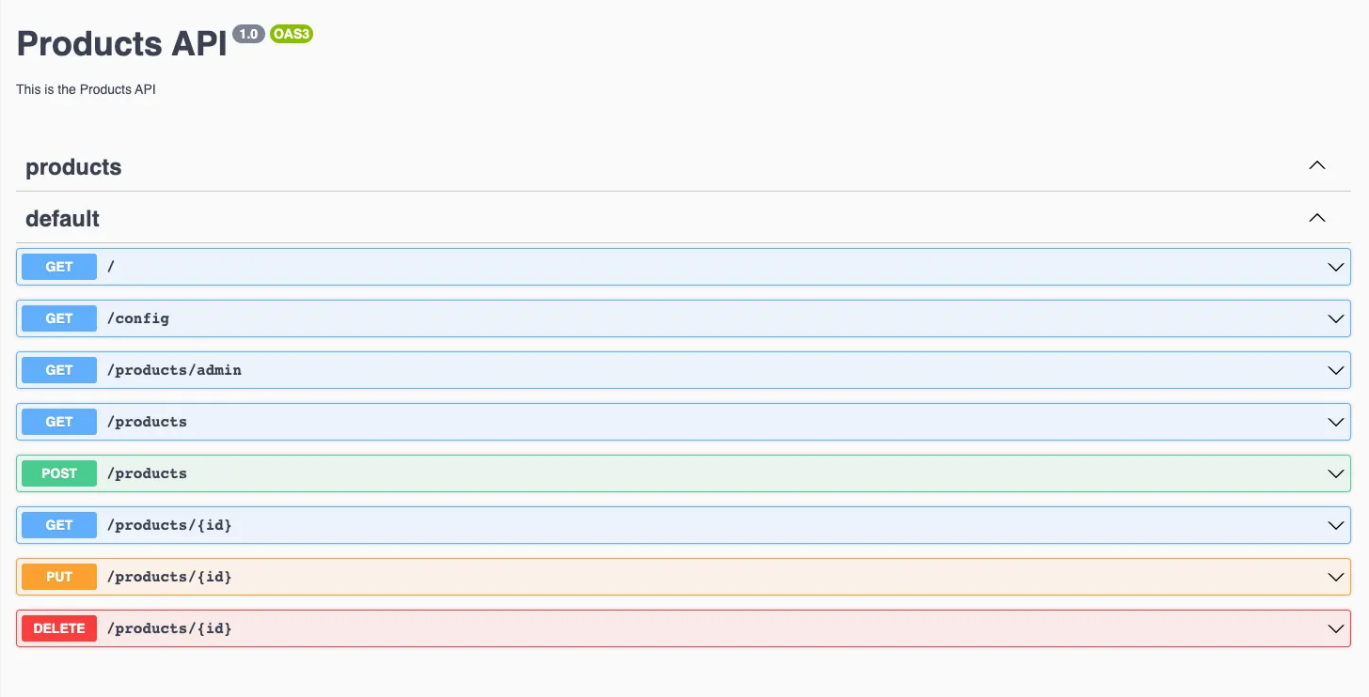
```
import { NestFactory } from '@nestjs/core';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { AppModule } from './app.module';
```

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

  const swaggerConfig = new DocumentBuilder()
    .setTitle('Products API')
    .setDescription('This is the Products API')
    .setVersion('1.0')
    .addTag('products')
    .build();

  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('api-docs', app, document);

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

Here's a breakdown of the above code with regard to bootstrapping the Swagger configuration:

- You first create a new `DocumentBuilder` object that will be used to define the API documentation.
- The `setTitle()`, `setDescription()`, and `setVersion()` methods on the `DocumentBuilder` object set some basic information about the API such as the title, description and version.
- Next, you call the `addTag()` method that is used to add a tag to the API. Tags are a way of organizing the endpoints of the application. In this case, the tag is "products". You will see it in action shortly.
- The `build()` method is called to create a complete Swagger configuration object based on the information defined by the `DocumentBuilder`.
- The `SwaggerModule.createDocument()` method is used to generate a Swagger document in line with Open API specifications.
- Finally, the `SwaggerModule.setup()` method is called to serve the Swagger document as HTML at a specific route of the application. The setup method takes the path, application instance and the Swagger document as input.
- Based on the path property in our example, the documentation will be available at the `/api-docs` endpoint.

You can now run the application using `npm run start` command and visit `http://localhost:3000/api-docs` to view the Swagger UI.



All the endpoints you have created so far in our demo application are displayed over here. You can also expand each endpoint and see more details.

With Swagger, you also get the option of triggering the API call and testing the output.

Grouping the Endpoints

At this point, you can see that even though all the endpoints are included, they are not grouped properly.

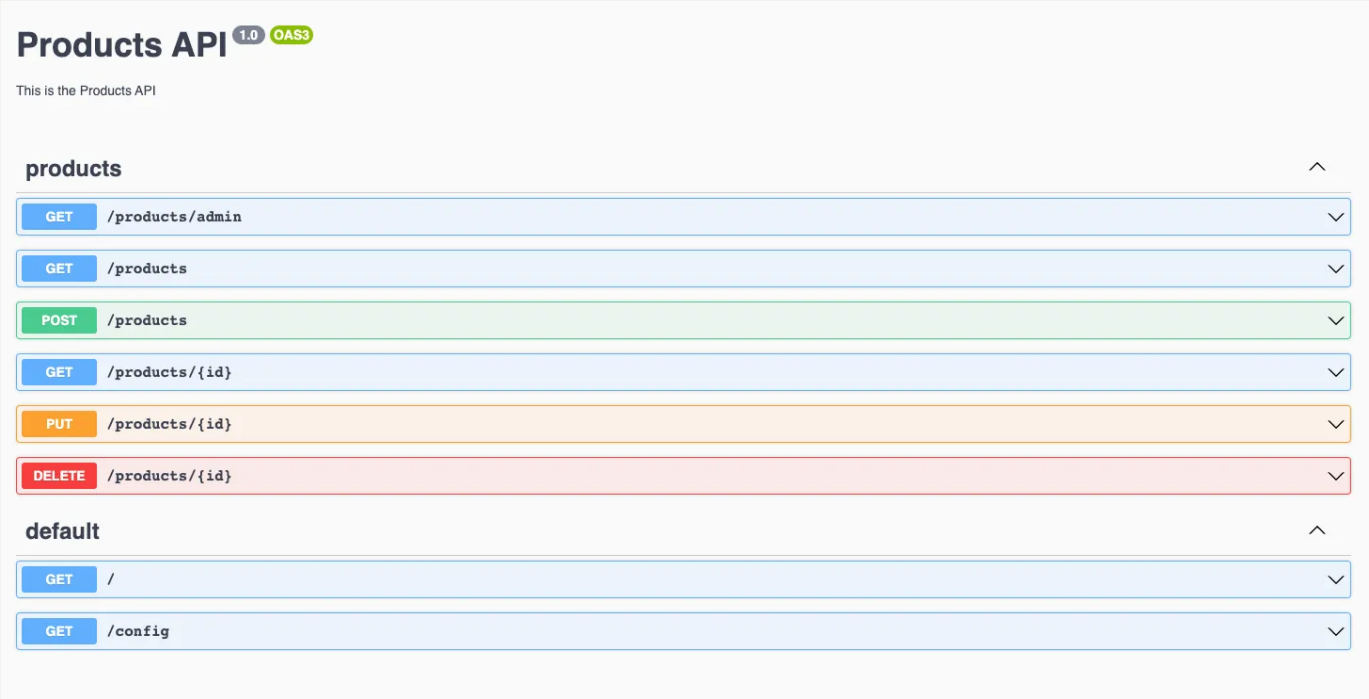
All the endpoints are listed under the “default” tag even though the “products” tag is present.

To fix this, you have to attach the product endpoints to the “product” tag by using the `@ApiTags()` decorator in the `ProductsController`. This tag is available as part of the `@nestjs/swagger` package.

See the below example:

```
@Controller('products')
@ApiTags('products')
export class ProductsController {}
```

If you run the application after making this change, you will find that the product-related endpoints are now grouped properly.



What's Next?

The code for this post is available on GitHub. You can download it and play around with it to practice the concepts discussed in this post.

To summarize this chapter, you have learned how to generate API documentation in NestJS with the Swagger module.

In the next chapter, you are going to learn about middleware functions in NestJS.

So continue reading!

Chapter 9 - Middleware Functions in NestJS

Here's a recap of what you have learnt in the previous chapter:

- The role of a controller in NestJS
- Creating a NestJS Controller
- Understanding the request and response objects
- Request and response related decorators

Using this knowledge, you were able to build **OpenAPI-compliant documentation** for your application.

It's time to look at another fundamental NestJS concept - **middleware functions**.

Here's the plan for this chapter:

- What is a middleware function?
- How to apply a middleware function in NestJS?
    - method-level
    - route wildcards
    - excluding routes
    - multiple middleware functions
- Functional middleware
- Global middleware

What is a Middleware?

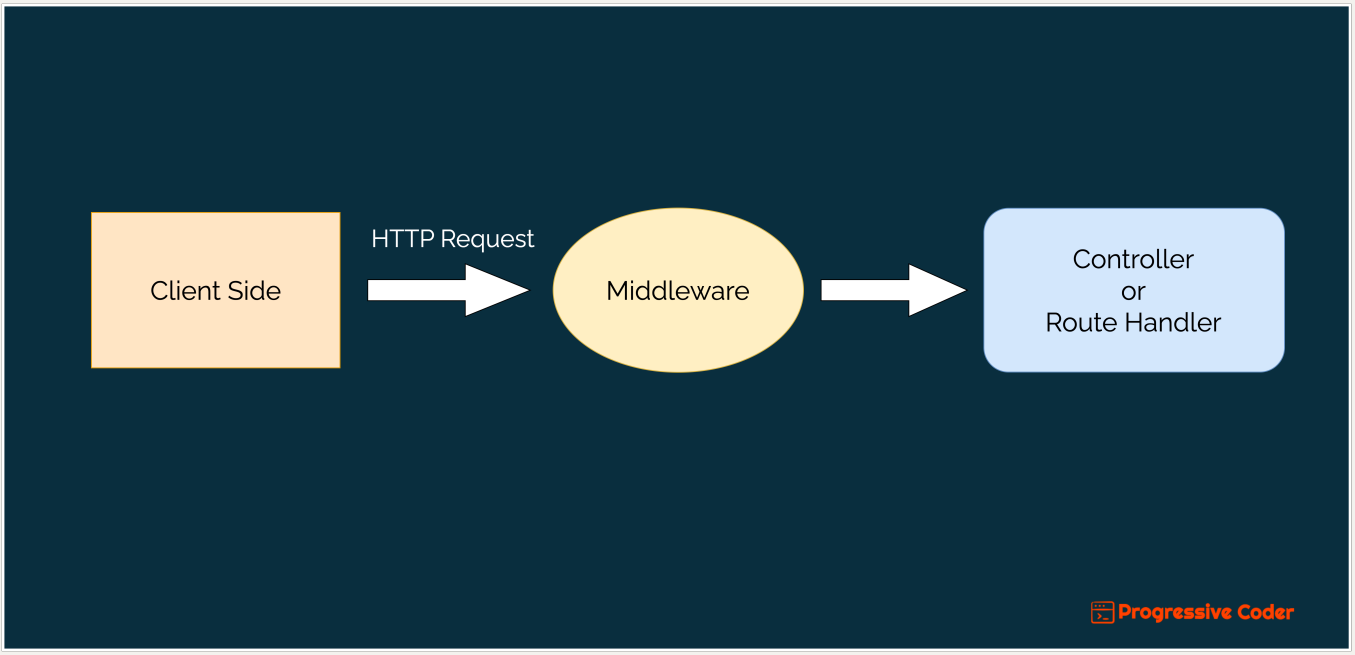
The term “middleware” might sound intimidating. But it’s just a fancy term for a function that’s called before the route handler.

Middleware functions get access to both **request** and **response** objects along with the `next()` middleware function.

You can use **middleware functions** to do all sorts of things such as:

- logging requests

- authentication & authorization
- rate limiting
- transform the request and response objects
- and many more...



When it comes to NestJS, the middleware functions are equivalent to **express** middleware.

Here's how you implement one in NestJS.

```
import { Injectable, NestMiddleware } from "@nestjs/common";
import { NextFunction, Request, Response } from "express";

@Injectable()
export class LoggingMiddleware implements NestMiddleware {
    use(req: Request, res: Response, next: NextFunction) {
        console.log(`Request object:
${JSON.stringify(req.headers)}`);
        next();
    }
}
```

This is known as the **class-based middleware implementation**. You can also implement using a simple function. We will see it in a later section of this post.

A couple of things to keep in mind:

- The class should implement the `NestMiddleware` interface. This means you need to implement the `use()` method that comes with the interface
- The class should be decorated with `@Injectable()` decorator. This is the same decorator you used while creating a NestJS Provider.

As you can see in the example, the `use()` function gets access to the request & response objects along with the `next()` function.

Within the function, you log the request headers. You can also log other parts of the request object depending on the requirement.

In case you have some other use case for the middleware, you can implement the desired logic within the `use()` function.

> *Nest middleware also supports dependency injection. Just like providers and controllers, you can inject dependencies that are available within the same module.*

How to apply a Middleware in NestJS?

In the previous section, you simply declared by a middleware function. It is still not active or applied to your application.

You activate a middleware function using the `configure()` method of a module class.

See the below example:

```
import { MiddlewareConsumer, Module, NestModule } from
'@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';
import { LoggingMiddleware } from './middlewares/logging.middleware';

@Module({
```

```
  imports: [ProductsModule,
    ConfigurationModule.register({db_url:
'<http://localhost:27017>'})],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggingMiddleware)
      .forRoutes('products')
  }
}
```

The `configure()` method takes an instance of the `MiddlewareConsumer` class. This is a helper class that has several in-built methods to manage the middleware.

You can chain the methods in a descriptive manner. In the above example, you `apply()` the middleware by chaining the `forRoutes('products')` method. This tells NestJS to apply the `LoggingMiddleware` to all the routes starting with `/products`.

There are also other ways you can apply a middleware function:

Method-Based Selection

You can apply the middleware function to specific routes by passing an object containing the route path and the method type as input.

See the below example:

```
import { MiddlewareConsumer, Module, NestModule, RequestMethod } from
'@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';
```

```
import { LoggingMiddleware } from './middlewares/logging.middleware';

@Module({
  imports: [ProductsModule,
    ConfigurationModule.register({db_url:
'<http://localhost:27017>'})],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggingMiddleware)
      .forRoutes({ path: 'products', method: RequestMethod.GET })
  }
}
```

Controller-Based Selection

Instead of the route path, you can also set a list of controllers in the `forRoutes()` method.

See the below example where the middleware function is applied to all the route handlers in the `ProductsController`.

```
import { MiddlewareConsumer, Module, NestModule, RequestMethod } from
'@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';
import { LoggingMiddleware } from './middlewares/logging.middleware';
import { ProductsController } from './products/products.controller';

@Module({
  imports: [ProductsModule,
```

```
    ConfigurationModule.register({db_url:
'<http://localhost:27017>'})],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggingMiddleware)
      .forRoutes(ProductsController)
  }
}
```

Pattern-Based Route Selection

The `forRoutes()` method also supports pattern-based route selection.

See the below example:

```
import { MiddlewareConsumer, Module, NestModule, RequestMethod } from
'@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';
import { LoggingMiddleware } from './middlewares/logging.middleware';

@Module({
  imports: [ProductsModule,
    ConfigurationModule.register({db_url:
'<http://localhost:27017>'})],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
```

```
      .apply(LoggingMiddleware)
      .forRoutes({ path: 'x*z', method: RequestMethod.ALL})
  }
}
```

Here, the `x*z` path contains the asterisk as a wildcard that will match any combination of characters.

Excluding Routes

Sometimes, you might need to exclude certain routes. You can do so by chaining the `exclude()` method to the `MiddlewareConsumer` instance.

Check the below example:

```
import { MiddlewareConsumer, Module, NestModule, RequestMethod } from
'@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsModule } from './products/products.module';
import { ConfigurationModule } from
'./configuration/configuration.module';
import { LoggingMiddleware } from './middlewares/logging.middleware';
import { ProductsController } from './products/products.controller';

@Module({
  imports: [ProductsModule,
    ConfigurationModule.register({db_url:
'<http://localhost:27017>'})],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggingMiddleware)
      .exclude(
```

```
        { path: 'products', method: RequestMethod.DELETE }
      )
      .forRoutes(ProductsController)
  }
}
```

Here, the middleware function does not execute for the `DELETE` method in the `ProductsController`.

Functional Middleware

In the above examples, you have used class-based middleware.

However, it is overkill considering that our class does not have dependencies, member variables or other methods.

In such cases, you can instead opt for **functional middleware**.

Check the below example:

```
import { NextFunction, Request, Response } from "express";

export function logger(req: Request, res: Response, next:
NextFunction) {
    console.log(`Request object: ${JSON.stringify(req.headers)}`);
    next();
}
```

The `logger()` function does the same thing as before but without the class syntax.

Within the `app.module.ts`, you can register the function.

```
import { logger } from './middlewares/logging.middleware';

export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(logger)
      .forRoutes(ProductsController)
  }
}
```

Global Middleware

There are cases when you want to apply a middleware function to the entire application scope. Logging can be one such requirement.

In such situations, you can apply a middleware function globally by attaching it to the application instance.

See the below snippet from the `main.ts` file.

```
import { NestFactory } from '@nestjs/core';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { AppModule } from './app.module';
import { logger } from './middlewares/logging.middleware';

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

  const swaggerConfig = new DocumentBuilder()
    .setTitle('Products API')
    .setDescription('This is the Products API')
    .setVersion('1.0')
    .addTag('products')
    .build();

  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('api-docs', app, document);
```

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

Here, the `logger()` middleware function is attached to the application instance using the `app.use()` function.

What's Next?

The code for this chapter is available on GitHub. You can download it and play around with it to practice the concepts discussed in this chapter.

To summarize this chapter, you have looked at middleware functions in NestJS and how you can use them.

In the next chapter, you are going to learn how to perform data validations in NestJS using pipes.

So continue reading!

Chapter 10 - Interceptors in NestJS

Here's a recap of what you have learnt in the previous chapter:

- What is a middleware function?
- How to apply a NestJS middleware function?
    - method-level
    - route wildcards
    - excluding routes
    - multiple middleware functions
- Class-based middleware to functional middleware
- Global middleware

Time to explore another NestJS fundamental feature. And that is Interceptors.

Here's the plan for this chapter:

- What is a NestJS interceptor?
- Understanding the concept of Aspect-Oriented Programming
- Creating a NestJS Interceptor
- Binding the Interceptor
    - controller-scoped
    - method-scoped
    - global-scoped

What is a NestJS Interceptor?

The term “interceptor” might sound like the name of a Jason Statham movie, but it is simply another TypeScript class with a few specific features.

Here are those specific features:

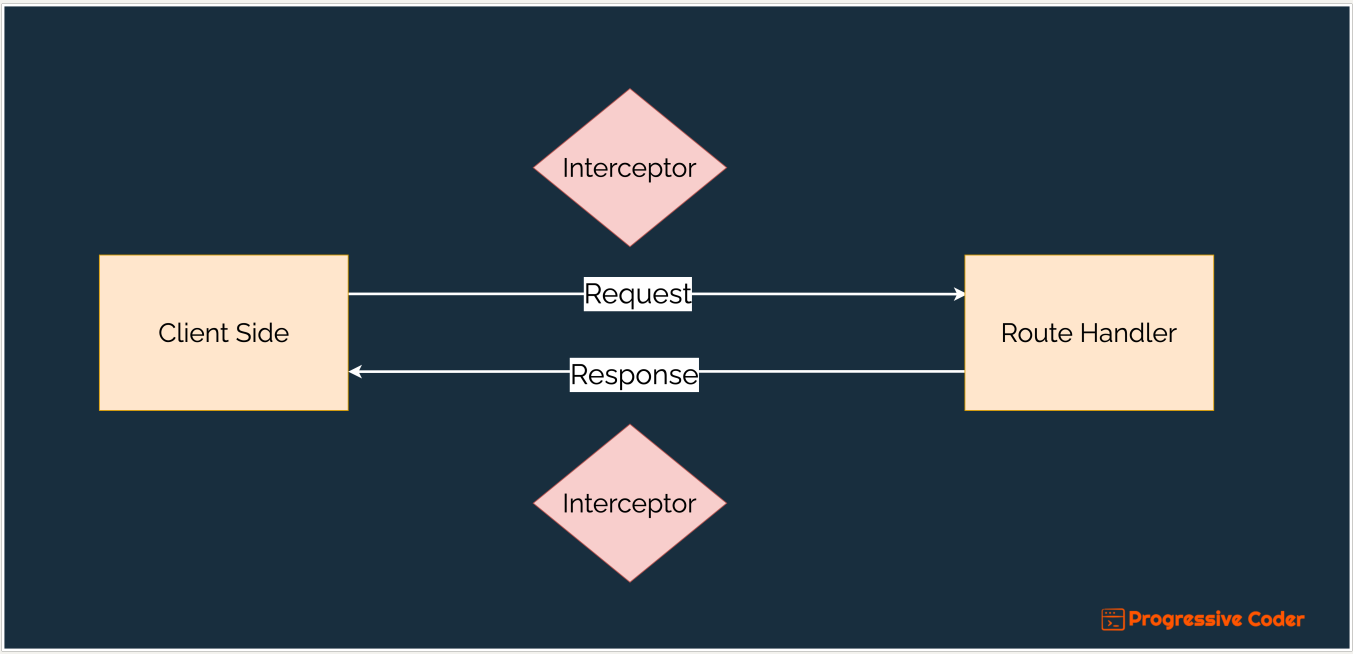
- The class should be annotated with the `@Injectable()` decorator.
- It should implement the `NestInterceptor` interface.

But what is the use of an interceptor?

The NestJS interceptors are embodiments of Aspect-Oriented Programming or the AOP technique. This means you can do a bunch of cool stuff with them such as:

- executing extra logic before & after a method's execution
- transform the result or exception returned from a function
- extend the behaviour of a particular function

Here's what the interceptor looks like in a big-picture view.



Understanding the concept of AOP

To understand Aspect-Oriented Programming or AOP, you need to get the hang of a few important terms.

🚀 Aspect

An aspect is a piece of code (like a module) that encapsulates a specific behaviour or concern.

You can think of it as a cross-cutting feature or functionality that can be used in different parts of a program.

For example, logging is a cross-cutting concern that can be used in a wide variety of situations in a program.

🚀 Pointcut

A pointcut is a specification that defines the place in the program flow where a particular aspect should be applied. It signifies a particular point or set of points in the program's execution.

For example, let's say you have a web application that handles user authentication.

You might define an aspect that handles logging and you want this aspect to log all successful and unsuccessful login attempts. In this case, you would define a pointcut that attaches the logging aspect to all authentication-related endpoints.

⭐ Ultimately, **Aspect-Oriented Programming** is a combination of aspects and pointcuts that allow you to write more modular code.

In the context of NestJS, interceptors allow you to implement the concepts of AOP.

Creating a NestJS Interceptor

Let us now create a NestJS interceptor.

In your project's `src` directory, create a folder named `interceptors` with a file named `logging.interceptor.ts`.

See the below code for this file:

```
import { CallHandler, ExecutionContext, NestInterceptor } from
"@nestjs/common";
import { Observable, tap } from "rxjs";

export class LoggingInterceptor implements NestInterceptor {
    intercept(context: ExecutionContext, next: CallHandler):
Observable<any> {
        console.log("Before reaching the request handler");
```

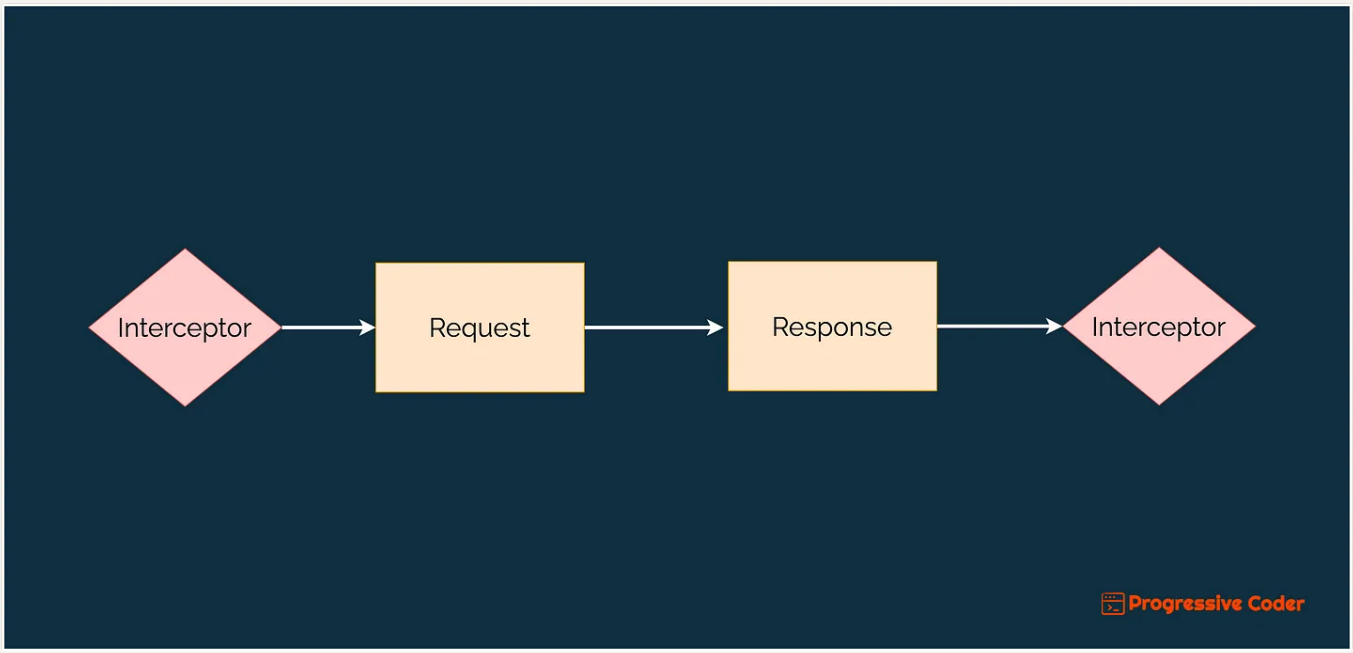
```
        const startTime = Date.now();

        const methodName = context.getHandler().name;
        const className = context.getClass().name;

        return next
            .handle()
            .pipe(
                tap(() =>
                    console.log(`Response Lag for ${methodName} in
class ${className}: ${Date.now() - startTime}ms`)
                )
            )
    }
}
```

So, what's going on in this piece of code?

- The `LoggingInterceptor` class implements the `NestInterceptor` interface. This was discussed earlier as a fundamental requirement for creating an interceptor.
- To satisfy the requirement of the `NestInterceptor`, you need to implement the `intercept()` method.
- The `intercept()` method takes two arguments - the `ExecutionContext` object and the `CallHandler` object.
- The `ExecutionContext` class is inherited from the `ArgumentsHost` class. We talked about it in the chapter on exception filters in NestJS.
- In the above example, we are using the `context` object to extract the `methodName` and `className` of the route handler.
- The `CallHandler` class implements the `handle()` method. Basically, this method sends the control to the route handler. If we don't call the `handle()` method within our interceptor, the incoming request will not reach the route handler.
- Also, the `handle()` method returns an RxJS observable. This provides a wide range of operators to manipulate the stream.
- In the above example, we use the `tap()` operator. This operator invokes an anonymous function that logs a message upon graceful termination of the

stream. In other words, at the end of the response cycle.

In a nutshell, the interceptor function wraps the request and response cycles. Here's what the flow of control looks like:



Binding the Interceptor

Declaring the interceptor class is one part of the equation. However, you still need to bind the interceptor to a particular set of functionality. Remember the concept of pointcut in aspect-oriented programming.

You can bind the interceptor at various scopes.

👉 Controller Scoped

As the name suggests, binding an interceptor at the controller scope activates the `intercept()` method for every request for all the route handlers within that particular controller.

See the below example:

```
@Controller('products')
@ApiTags('products')
@UseInterceptors(LoggingInterceptor)
export class ProductsController { }
```

The `@UseInterceptors()` decorator takes a list of interceptors as input.

You can directly pass the class definition as in the above case. NestJS will automatically take care of instantiating the class and wiring up the instance as part of its dependency injection process.

If you run the application and visit the `/products` route, you should the interceptor response in the console.

```
Before reaching the request handler
Response Lag for getAllProducts in class ProductsController: 3ms
```

👉 Method Scoped

You can also make an interceptor method-scoped.

See the below example:

```
@Get()
@UseInterceptors(LoggingInterceptor)
getAllProducts() {
    return this.productsService.getAllProducts();
}
```

👉 Global Scoped

Lastly, you can also go for a global-scoped interceptor.

For this, you need to make the below changes to the `main.ts` file of your application.

```
import { NestFactory } from '@nestjs/core';
```

```
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { AppModule } from './app.module';
import { LoggingInterceptor } from
'./interceptors/logging.interceptor';

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

  const swaggerConfig = new DocumentBuilder()
    .setTitle('Products API')
    .setDescription('This is the Products API')
    .setVersion('1.0')
    .addTag('products')
    .build();

  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('api-docs', app, document);

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

Here, we call the `useGlobalInterceptors()` method available on the `app` object.

What's Next?

The code for this chapter is available on GitHub. You can download it and play around with it to practice the concepts discussed in this chapter.

To summarize this chapter, you have looked at interceptors in NestJS and their utility in a typical application flow.

In the next chapter, you are going to learn how to use NestJS built-in pipes in your application controllers.

So continue reading!

Chapter 11 - NestJS Built-In Pipes

Here's a recap of what you have learnt in the previous chapter:

- What is a NestJS interceptor?
- Understanding the concept of Aspect-Oriented Programming
- Creating a NestJS Interceptor
- Binding the Interceptor
    - controller-scoped
    - method-scoped
    - global-scoped

It's time to look at the last major fundamental concept in NestJS known as Pipes.

Here's the plan for this chapter:

- What is a Pipe?
- How to use a built-in pipe?
- Customizing the built-in pipe

What is a Pipe?

A pipe is a class that has two distinguishing features:

- It is annotated with the `@Injectable()` decorator.
- It also implements the `PipeTransform` interface.

But what is the use of a pipe in NestJS?

There are two main use cases:

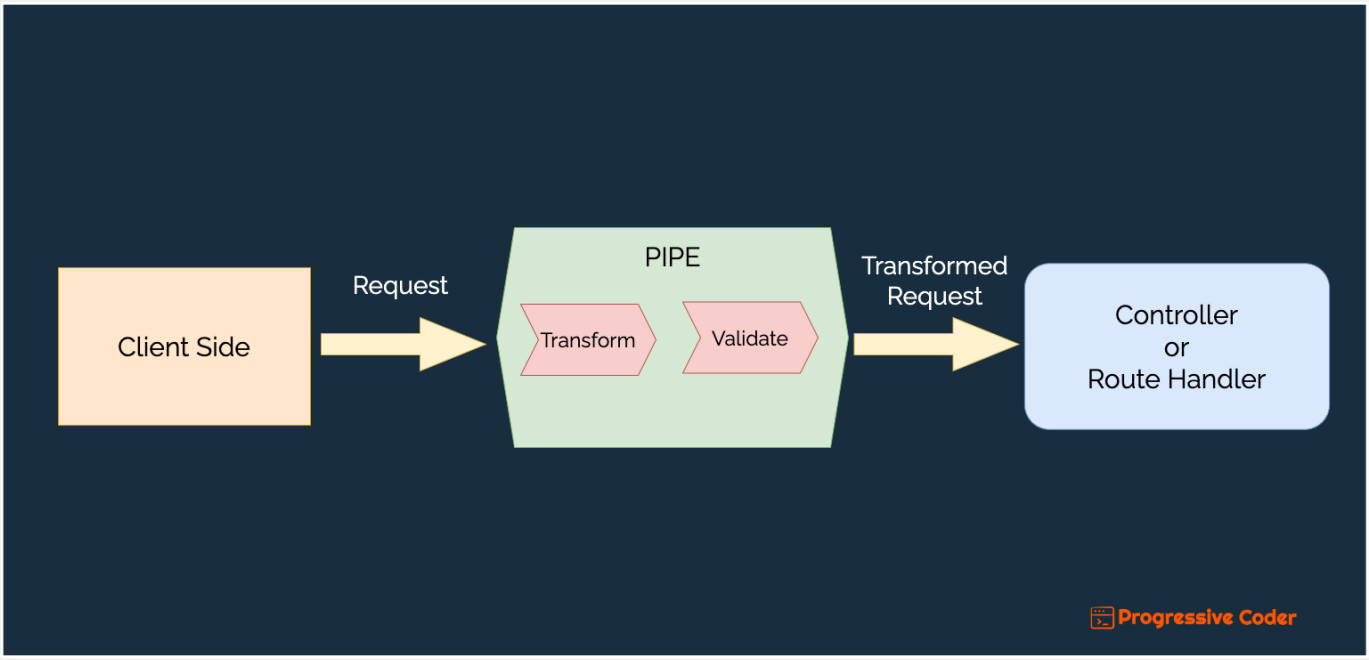
- **Transformation** of input data to the desired form. For example, you can use pipes to convert an input string to an integer.
- **Validation** of input data and throwing exceptions in case it is not valid.

Whatever the use case, NestJS first passes the arguments going to a route handler to the pipe.

The pipe receives the arguments and operates on them. Here, operating means transforming or validating.

Once done, the route handler is invoked with the transformed and validated arguments.

Here's an illustration that depicts this flow:



How to use the Built-In Pipes?

NestJS comes with several built-in pipes as part of the `@nestjs/common` package.

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

To use a pipe, you need to bind the pipe to a particular controller method.

For example, the `ParseIntPipe` transforms the method handler's parameter to a JavaScript integer. If the transformation is not successful, it throws an exception.

Here's the example code for the same.

```
@Get(':id')
getProduct(@Param('id', ParseIntPipe) id: number) {
    return this.productsService.getProduct(id);
}
```

Basically, you are binding the `ParseIntPipe` to the `getProduct()` route handler.

Notice that we are passing only the class name `ParseIntPipe` and leaving the responsibility for instantiation to the framework. This is using the concept of **Dependency Injection** once again.

NestJS automatically makes sure that the pipe runs before the method is called.

If you call the endpoint with a string as input i.e. `http://localhost:3000/products/abc`, you get an exception as below:

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

The exception prevents the `getProduct()` method to execute with invalid data. The response status code is 400.

Customizing the Built-In Pipe

In case you want to customize the built-in behaviour of the pipe such as changing the status code, you need to pass an in-place instance.

See the below example:

```
@Get(':id')
getProduct(@Param('id', new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.I_AM_A_TEAPOT })) id: number) {
      return this.productsService.getProduct(id);
}
```

With this change, you get the below response:

```
{
  "statusCode": 418,
  "message": "Validation failed (numeric string is expected)",
  "error": "I'm a teapot"
}
```

Wrapping Up

That's all for the **NestJS fundamentals**.

Here's a **high-level summary & recap** of all the things we covered in this book as part of learning the fundamentals.

Setting up NestJS and choosing the right platform

- Understanding the NestJS ecosystem and creating your first NestJS application with the CLI.
- Platform-agnostic nature of NestJS and the choice of Express or Fastify as the base platforms.

Controllers, Providers and Modules

- Handling incoming requests and returning responses with **NestJS Controllers**.
- Sharing common functionality and data between different parts of the application using **NestJS Providers**.
- Structuring your application with **NestJS Modules**.
- Learning about **Dynamic NestJS Modules** to make things a little more flexible.

Exception Handling in NestJS

- In-built exception handling process in NestJS
- Implementing custom exception filter in NestJS to customize the exception flow

Middleware & Interceptors in NestJS

- Creating and applying **middleware functions** in a typical NestJS request flow.
- Aspect-oriented programming using **interceptors in NestJS**.

API Documentation using Swagger

- Generating OpenAPI compliant documentation using Swagger in NestJS application.

Pipes in NestJS

- Using in-built pipes in NestJS to transform incoming request data.