On the edge of the Cosmos

A journey towards advanced servers in NestJS exploring TypeScript Decorators

Kevin Santana
VERSÃO 1.0 AUTHOR

CHAPTER 1

Typescript Decorators

WARNING

The content of this EBOOK assumes that you already have prior knowledge of Node.JS and the TypeScript language.

Typescript Decorators

Decorators in **TypeScript** are like advanced alien devices that can be applied to classes, methods, accessors, and properties, altering their functionalities in a powerful way. They act like force fields, adding protection and special functionalities without modifying the original structure.

Decorators are like cybernetic implants. They can be used to upgrade methods and properties of existing classes, adding new capabilities or adjusting existing ones to improve performance.

Typescript Decorators

Imagine **decorators** as energy shields, validating data and controlling access. This ensures that methods are executed within the desired parameters, protecting against failures and attacks, just like a shield allows allies to enter the field or blocks enemy attacks.

Typescript Decorators

The **decorator** is a powerful feature provided by TypeScript. See its basic syntax below:

Syntax in Class:

```
//Decorator on class declaration
@NavigationMode(FlightMode.ALLIED_SECTORS_ONLY)
class StarShipCore {
  ...
}
```

Syntax in Method:

```
class StarshipSystems {

  //Decorator on Method
  @CheckHealth(CheckLevel.NOMINAL)
  public executeSubSystem( ){
    ...
  }
}
```

Typescript Decorators

Syntax in assessor:

```
class StarshipSystems {

  private _engine: EngineCore

  //Decorator on acessor
  @CaculateFuel()
  get Engine(){
    return this._engine
  }

  //Note
}
```

Note: TypeScript does not allow adding decorators to both the getter and the setter of a single member. Instead, all member decorators must be applied to the first accessor specified in the order of the document. This is because decorators apply to a Property Descriptor, which combines both the getter and the setter, and not to each declaration separately.

Syntax in Property:

```
class StarShipFuselage {

  @Registration("LLMS %s")
  shipName: string;

  ...
}
```

Typescript Decorators

Signature of a Basic Decorator

```
function decorator(target) {
  // do something with 'target' ...
  // target is a class, method, accessor or property, depending on which place you are
  // defining
}
```

Signature of a Factory Decorator

```
function decorator() {

  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    ...
  };
}
```

If you want to customize how the decorator is applied to the declaration, you can follow the model above. This method will be called at runtime, like a laser weapon with its modifiers that alter the energy charge released by the charger at the moment the trigger is pulled.


CAPÍTULO 2
NestJS Framework

NestJS Framework

NestJS Introduction

Imagine **NestJS** as a powerful spaceship, designed to build fast and scalable server-side applications.

Key features::

- Programmed in TypeScript, the language of the future;
- Modular architecture, like a set of interchangeable modules;
- Compatible with the Node.js universe;
- Equipped with dependency injection for maximum efficiency.

NestJS Framework

Architecture and Basic Components of NestJS

Modules: Modular structures that form the components of the spaceship.

Modules are essentially the main structural points of each function in the system. For example: Users module, Sales module, Reports module, and so on.

NestJS Framework

Architecture and Basic Components of NestJS

Controllers: The control panel that defines the entry points of the spaceship.

Controllers are classes used to create the endpoints of your REST application (in GraphQL, another methodology is used, meaning controllers are not used, or at least should not be).

NestJS Framework

Architecture and Basic Components of NestJS

Providers: Engines that encapsulate business logic.

In NestJS, providers have a broad and fundamental concept. Basically, all complex or specific tasks can be delegated as a provider.

The main idea of a provider is that it can be injected as a dependency into a controller or service, for example.

Like a boost to the energy field.

NestJS Framework

Architecture and Basic Components of NestJS

Pipes, Guards e Interceptors: They are systems that can manipulate data, security, and intercept communications.

It's like a radar and scanner system on a spaceship that has extensive capabilities to manage visibility, protect against intrusions, and detect the presence of others nearby.

NestJS Framework

NestJS Installation

NestJS is a framework in Node.js, so we'll use NPM, which is a package manager, to install its CLI globally. It's like a configuration menu on an intergalactic fighter—you select the command, and the spaceship executes it.

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

After everything is set up, you can run the server with the command below.

```
$ npm run start
```

Ready! Your NestJS server in Node.js and TypeScript using Express is up and running like a colony ship.


CHAPTER 3
NestJS Decorators

NestJS Framework

Features of NestJS and Decorators

In most servers, it's necessary to use specific classes and methods provided by libraries to create your REST or GraphQL server. These methods need to be instantiated and follow a series of syntax and signature patterns to execute. Additionally, routes are typically centralized in a specific class or function within a file, which can reduce modularity.

```
import { Router } from 'express';
import userController from '../controllers/userController';

const router = Router();

// route GET for users
router.get('/', userController.getUsers);

export default router;
```

NestJS Framework

Features of NestJS and Decorators

However, with NestJS, you harness the powerful capability of decorators to decentralize these processes, allowing you to modularize your entire server by creating interchangeable and independent functions.

Think of a space station divided into multiple sectors, each with its isolated functions, but at the same time able to communicate with or without dependencies.

```
import { Controller, Get } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get()
  findAll(): string {
    return 'This action returns all users';
  }
}
```

NestJS Framework

Features of NestJS and Decorators

Definition

Futuristic tools that add metadata to classes and functions, like upgrades to the spaceship.

Operation

They use metaprogramming to modify and enhance the behavior of the code.

NestJS Framework

Types of native decorators in NestJS

Decorators of Classes

@Module:

Defines a module in NestJS, like a new compartment of the spaceship.

It contains metadata about controllers, providers, and interconnected modules.

@Injectable:

Marks a class as an injectable provider, like an extra engine of the spaceship.

Used for services and repositories.

NestJS Framework

Types of native decorators in NestJS

Decorators of Methods

@Get, @Post, @Put, @Delete:

Associates methods with specific HTTP routes, like navigation commands.

Example: @Get('/users') for an endpoint that returns all users.

@UseGuards:

Applies route guards for access control, like a security shield.

Example: @UseGuards(AuthGuard) to protect a route with authentication.

NestJS Framework

Types of native decorators in NestJS

Decorators of Parameters

@Param:

Extracts route parameters, like spatial coordinates.

Example: @Param('id') to capture the ID from the URL.

@Body:

Extracts the request body, like the payload of the spaceship.

Example: @Body() createUserDto to access the data sent in a POST request.

NestJS Framework

Types of native decorators in NestJS

Decorators of Parameters

@Query:

Captures query string parameters, like scanning data.

Example: @Query('page') to access the page of a listing.

NestJS Framework

Types of native decorators in NestJS

Special Decorators

@UsePipes:

Applies pipes for data transformation and validation, like filtering systems.

Example: @UsePipes(ValidationPipe) to validate the request body.

@UseInterceptors:

Applies interceptors to handle requests/responses, like a communication interception system.

Example: @UseInterceptors(LoggingInterceptor) to add logs to requests.


CHAPTER 4
NestJS Custom Decorators

NestJS Framework

Custom decorators in NestJS

Custom decorators in NestJS are like new gadgets being created to enhance the spaceship.

Using what was learned about TypeScript decorators in Chapter 1, it's possible to create advanced and robust decorators to be declared in your service or controller modules, thereby enhancing your server's capabilities.

NestJS Framework

Custom decorators in NestJS

Let's create an example of a decorator that checks if a user has a certain permission before accessing any method.

Let's go to the docks!!

The implementation can be done in various ways, but since we're talking about decorators, let's keep the coding focused on using decorators.

We will be using an additional JWT package for access exemplification...

```
npm i @nestjs/jwt
```

NestJS Framework

Decorators Customizados do NestJS

Step 1: Create the decorador

```ts
import { SetMetadata } from '@nestjs/common';

export const PERMISSIONS_KEY = 'permissions';
export const Permissions = (...permissions: string[]) =>
    SetMetadata(PERMISSIONS_KEY, permissions);
```

What we did here is very simple. By using a native function of NestJS, SetMetadata(), we added metadata to an invoked class or function. In other words, we added an object with the name 'permissions' and the provided values.

Thus, the class or function that uses the decorator will contain this value in its instance at runtime.

NestJS Framework

Decorators Customizados do NestJS

Step 2: Create a Guard for Permission

The next step is to create a guard that will use the decorator we just created to check the user's credentials before allowing the method to execute. Think of it like a deck door with biometrics.

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { PERMISSIONS_KEY } from './permissions.decorator';

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

  canActivate(context: ExecutionContext): boolean {
    const requiredPermissions = this.reflector.getAllAndOverride<string[]>(PERMISSIONS_KEY,
[
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredPermissions) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredPermissions.every(permission => user.permissions.includes(permission));
  }
}
```

NestJS Framework

Decorators Customizados do NestJS

Step 3: Create a Guard for Authentication

In this step, we need to create a Guard that authenticates the user and obtains their credentials.

For simplicity, we will use a JWT token with the permissions already included in its payload.

```
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException } from
'@nestjs/common';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}

  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);

    if (!token) {
      throw new UnauthorizedException('No token provided');
    }

    try {
      const payload = await this.jwtService.verifyAsync(token);
      request.user = {
        id: payload.sub,
        username: payload.username,
        permissions: payload.permissions || [],
      };
    } catch (e) {
      throw new UnauthorizedException('Invalid token');
    }

    return true;
  }

  private extractTokenFromHeader(request: any): string | null {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : null;
  }
}
```

NestJS Framework

Decorators Customizados do NestJS

Step 4: Apply the decorator and the guard to the controller.

Now, let's use the @Permissions decorator to protect a specific endpoint in the controller and apply the **PermissionsGuard** globally or to the specific controller.

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { Permissions } from './permissions.decorator';
import { PermissionsGuard } from './permissions.guard';

@Controller('items')
@UseGuards(PermissionsGuard)
export class ItemsController {
  @Get()
  @Permissions('view_items')
  findAll() {
    return 'This action returns all items';
  }
}
```

NestJS Framework

Decorators Customizados do NestJS

Step 5: Configure the AppModule

Lastly, but not least, we need to add JwtModule, AuthGuard, and PermissionsGuard to the AppModule. This is an important step because it informs NestJS that this created content exists and should be loaded and injected during server initialization.

It's like installing a gadget but not connecting the wires, right?

```
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';
import { JwtModule } from '@nestjs/jwt';
import { AuthGuard } from './Auth.guard';
import { PermissionsGuard } from './Permissions.guard';
import { ItemsController } from './items.controller';

@Module({
  imports: [
    JwtModule.register({
      secret: '12345', // Secret Key
      signOptions: { expiresIn: '60s' },
    }),
  ],
  controllers: [ItemsController],
  providers: [
    {
      provide: APP_GUARD,
      useClass: AuthGuard,
    },
    {
      provide: APP_GUARD,
      useClass: PermissionsGuard,
    },
  ],
})
export class AppModule { }
```

CONCLUSION

With decorators, a series of advanced techniques can be created to make your server increasingly futuristic and technologically advanced, improving coherence, performance, and reducing boilerplate code.

Embark on this spaceship called NestJS and discover the power of decorators to build intergalactic applications!

If you want to see the code in action, the next page contains the link to the server's source code.

Run it in your terminal after cloning the repository.

```
npm install
npm run test
```

THANK YOU!!!

Kevin Santana

KFMS

github.com/zekdtonik

linkedin.com/in/kevin-fms

Source Code