

Na Fronteira do Cosmos

Uma viagem rumo ao servidores avançados em NestJS explorando os TypeScript Decorators

TS

I2



Kevin Santana

VERSAO 1.0

AUTHOR



CAPÍTULO 1

TypeScript Decorators

IMPORTANTE

O conteúdo deste EBOOK leva em consideração que você já possui conhecimento prévio em Node.JS e da linguagem TypeScript.



TypeScript Decorators

Decorators em **TypeScript** são como dispositivos alienígenas avançados que podem ser aplicados a classes e métodos, assessores e propriedades alterando suas funcionalidades de maneira poderosa. Eles agem como campos de força, adicionando proteção e funcionalidades especiais sem modificar a estrutura original.

Os **decorators** são como implantes cibernéticos. Eles podem ser usados para atualizar métodos e propriedades de classes existentes, adicionando novas capacidades ou ajustando as existentes para melhorar o desempenho.





TypeScript Decorators

Imagine os **decorators** como escudos de energia, validando dados e controlando acessos. Isso garante que os métodos sejam executados dentro dos parâmetros desejados, protegendo contra falhas e ataques, da mesma forma que um escudo permite o acesso de aliados para dentro do campo ou bloqueia ataque inimigos.





TypeScript Decorators

O **decorator** é uma funcionalidade poderosa fornecida pelo typescript. Veja a sua sintaxe básica abaixo:

Definição em Classe:



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

Definição em Método:



```
class StarshipSystems {

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



TypeScript Decorators

Definição em assensores:



```
class StarshipSystems {  
  
    private _engine: EngineCore  
  
    //Decorator on accessor  
    @CaculateFuel()  
    get Engine(){  
        return this._engine  
    }  
  
    //Note  
}
```

Nota: O TypeScript não permite adicionar decoradores tanto no getter quanto no setter de um único membro. Em vez disso, todos os decoradores do membro devem ser aplicados ao primeiro assessor especificado na ordem do documento. Isso acontece porque os decoradores se aplicam a um Descriptor de Propriedade, que combina tanto o getter quanto o setter, e não a cada declaração separadamente.

Definição em Propriedade:



```
class StarShipFuselage {  
  
    @Registration("LLMS %s")  
    shipName: string;  
  
    ...  
}
```





TypeScript Decorators

Assinatura de um Decorator Básico



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

Assinatura de um Factory Decorator



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

Se você deseja customizar como o decorador é aplicado a declaração, você pode seguir o modelo acima, esse método será chamado em tempo de execução, como se fosse uma arma laser com seus modificadores que alteram a carga de energia liberada pelo carregador no momento em que se aperta o gatilho.





CAPÍTULO 2



NestJS Framework





NestJS Framework

Introdução ao NestJS

Imagine o **NestJS** como uma poderosa nave espacial, projetada para construir aplicativos server-side rápidos e escaláveis.

Principais características:

- Programado em TypeScript, a linguagem do futuro.
- Arquitetura modular, como um conjunto de módulos intercambiáveis.
- Compatível com o universo Node.js.
- Equipado com injeção de dependências para eficiência máxima.





NestJS Framework

Arquitetura e Componentes Básicos do NestJS

Módulos: Estruturas modulares que formam os componentes da nave.

Os módulos basicamente são os principais pontos estruturais de cada função no sistema. Ex: Módulo de usuários, de vendas, de relatórios e etc...





NestJS Framework

Arquitetura e Componentes Básicos do NestJS

Controladores (Controllers): O painel de controle que define os pontos de entrada da nave.

Os controladores são as classes utilizadas para criar os endpoints da sua aplicação REST (em GraphQL outra metodologia é utilizada, ou seja, os controladores não são utilizados, ou pelo menos não deve ser).





NestJS Framework

Arquitetura e Componentes Básicos do NestJS

Provedores (Providers): Motores que encapsulam a lógica de negócios.

Em NestJS os provedores possuem um conceito amplo e fundamental. Basicamente, todas as tarefas complexas ou específicas podem ser delegadas como um provider.

A ideia principal de um provider é que ele pode ser **injetado** como uma dependência em uma controlador ou serviço por exemplo.

Tipo um boost no campo de energia.





NestJS Framework

Arquitetura e Componentes Básicos do NestJS

Pipes, Guards e Interceptors: São sistemas que podem manipular dados, segurança e interceptar as comunicação.

É como um sistema de radar e scanners de uma nave que por sua vez possui uma vasta capacidade de gerenciar visibilidade, proteger de invasões e detectar presença de quem estiver ao redor.





NestJS Framework

Instalação do NestJS

O NestJs é um framework em Node.JS, logo utilizaremos o NPM que é um gerenciador de pacotes para instalá-lo o seu CLI globalmente, é como um menu de configuração em um fighter intergalático, você seleciona o comando e a nave executa.



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

Depois que tudo estiver pronto você pode executar o servidor com o comando abaixo.



```
$ npm run start
```

Pronto! o seu servidor NestJS em node.js e typescript usando o **express** está ligado e executando como uma nave colônia.





CAPÍTULO 3



NestJS Decorators





NestJS Framework

Características do NestJS e Decorators

Na maioria dos servidores, é necessário utilizar classes e métodos específicos implementados pelas bibliotecas para criar seu servidor em REST ou GRAPHQL, tais métodos precisam ser instanciados e seguir uma série de padrões de sintaxe e assinatura para executar. Além de centralizar as rotas em uma classe ou função específica em um arquivo, diminuindo a modularidade.



```
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

Características do NestJS e Decorators

Entretanto, com o NestJS, você aproveita da poderosa capacidade dos **decorators** para realizar esses processos descentralizados, permitindo modularizar todo o seu servidor, criado funções intercambiáveis e independentes.

Pense em uma estação espacial dividida em vários setores, cada um tem as suas funções isoladas, mas ao mesmo tempo podem se comunicar com ou sem dependências.



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

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





NestJS Framework

Características do NestJS e Decorators

Definição

Ferramentas futuristas que adicionam metadados a classes e funções, como upgrades na nave.

Funcionamento

Usam metaprogramação para modificar e aprimorar o comportamento do código.





NestJS Framework

Tipos de Decorators nativos do NestJS

Decorators de Classes

@Module:

Define um módulo no NestJS, como um novo compartimento da nave.

Contém metadados sobre controllers, providers e módulos interligados.

@Injectable:

Marca uma classe como um provedor injetável, como um motor extra da nave.

Usado para serviços e repositórios.





NestJS Framework

Tipos de Decorators nativos do NestJS

Decorators de Métodos

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

Associe métodos a rotas HTTP específicas, como comandos de navegação.

Exemplo: `@Get('/users')` para um endpoint que retorna todos os usuários.

@UseGuards:

Aplica guarda de rota para controle de acesso, como um escudo de segurança.

Exemplo: `@UseGuards(AuthGuard)` para proteger uma rota com autenticação.





NestJS Framework

Tipos de Decorators nativos do NestJS

Decorators de Parâmetros

@Param:

Extrai parâmetros de rota, como coordenadas espaciais.

Exemplo: @Param('id') para capturar o ID da URL.

@Body:

Extrai o corpo da requisição, como a carga da nave.

Exemplo: @Body() createUserDto para acessar os dados enviados em uma requisição POST.





NestJS Framework

Tipos de Decorators nativos do NestJS

Decorators de Parâmetros

@Query:

Captura parâmetros de query string, como dados de escaneamento.

Exemplo: `@Query('page')` para acessar a página de uma listagem.





NestJS Framework

Tipos de Decorators nativos do NestJS

Decorators Especiais

@UsePipes:

Aplica pipes para transformação e validação de dados, como sistemas de filtragem.

Exemplo: `@UsePipes(ValidationPipe)` para validar o corpo da requisição.

@UseInterceptors:

Aplica interceptors para manipular requests/responses, como um sistema de interceptação de comunicações.

Exemplo: `@UseInterceptors(LoggingInterceptor)` para adicionar logs às requisições.





CAPÍTULO 4



NestJS Custom Decorators





NestJS Framework

Decorators Customizados do NestJS

OS decoradores customizados são como novos gadgets que vão sendo criados para a melhoria da nave.

Utilizando o que foi aprendido nas instruções sobre Typescript decorators no capítulo 1, é possível criar decoradores avançados e robustos para serem declarados em seus módulos de serviços ou controladores, aumentando as capacidades de seu servidor.





NestJS Framework

Decorators Customizados do NestJS

Criaremos um exemplo de um decorator que verifica se um usuário tem uma certa permissão antes de acessar um método qualquer.

Vamos para as docas!!

A implementação pode ser feita de várias formas, mas como estamos falando sobre **decoradores**, vamos manter a codificação direcionada a utilizar os decorators.

Estaremos utilizando um package adicional de JWT para exemplificação de acesso...



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





NestJS Framework

Decorators Customizados do NestJS

Passo 1: Criar o decorador



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

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

O que fizemos aqui é muito simples, utilizando uma função nativa do NestJS, **SetMetadata()**, adicionamos um metadado para uma classe ou função invocada, ou seja, adicionamos um objeto com o nome de ‘permissions’ e com os valores fornecidos.

Logo, a classe ou função que utiliza o decorador conterá esse valor na sua instância em tempo de execução.





NestJS Framework

Decorators Customizados do NestJS

Passo 2: Criar um Guard de Permissão

O próximo passo é criar um guard que irá utilizar o decorador que acabamos de criar para verificar as credenciais do usuário antes de permitir a execução do método. Pense nele como uma porta de convés com uma biometria.



```
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

Passo 3: Criar um Guard de Autenticação

Nesta etapa precisamos criar um Guard que autentica o usuário e obtém suas credenciais.

Sendo assim, a título de facilidade, utilizaremos um token JWT com as permissões já no seu 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

Passo 4: Aplicar o Decorator e o Guard ao Controlador

Agora, vamos usar o decorator **@Permissions** para proteger um endpoint específico no controlador e aplicar o **PermissionsGuard** globalmente ou no controlador específico.



```
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

Passo 5: Configurar o AppModule

Por último, mas não menos importante, devemos adicionar o JwtModule, o AuthGuard e o PermissionsGuard ao AppModule. Essa é uma parte importante, porque está dizendo ao NestJS que esse conteúdo criado existe e deve ser carregado e injectado na inicialização do servidor.

Não adianta você instalar um gadget e não plugar os fios não é mesmo.

```
● ● ●

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 {}
```





CONCLUSÃO

Com os decorators, uma série de técnicas avançadas podem ser criadas para deixar o seu servidor cada vez mais futurista e tecnológico, melhorando, a coerência, o desempenho e a redução de boilerplate.

Embarque nesta nave espacial chamada NestJS e descubra o poder dos decorators para construir aplicações intergalácticas!

Caso você queira ver o código em execução, na próxima página tem o link para ao código fonte do servidor.

Execute no seu terminal após clonar o pacote.



```
npm install  
npm run test
```



Agradecimentos



Kevin Santana

KFMS>

<https://github.com/zekdtonik> | decoratorstutorial.com



github.com/zekdtonik



linkedin.com/in/kevin-fms

[Source Code](#)