CAMPINAS TECH TALENTS

Frameworks

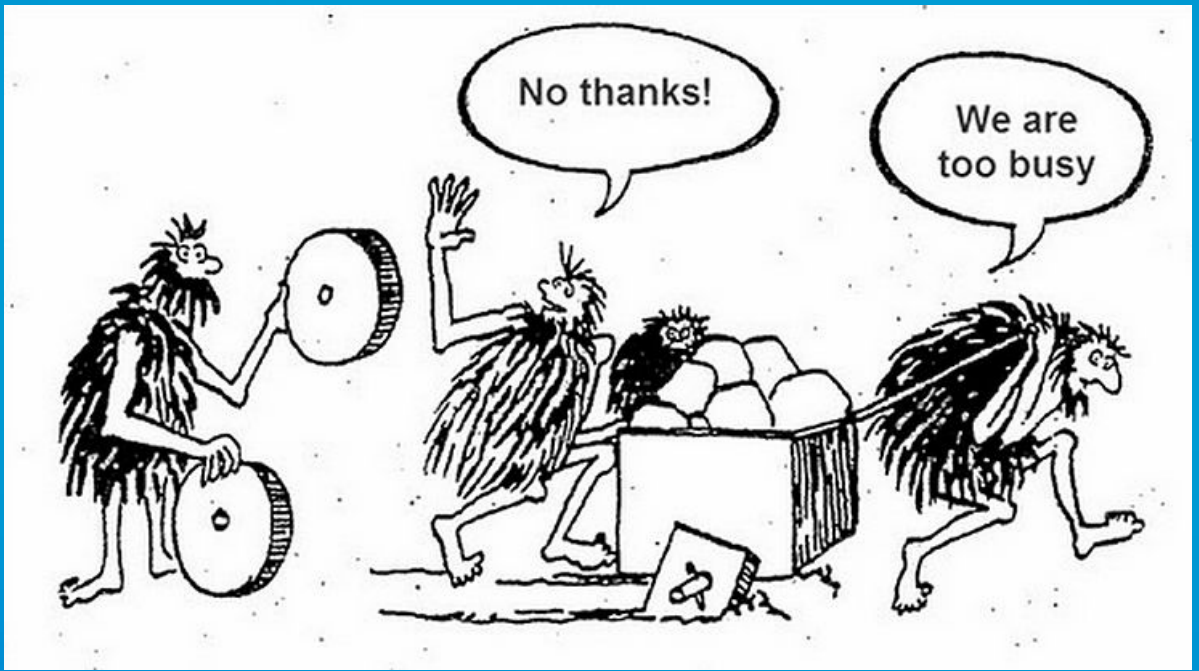
Lucas Roldão
in/lucas-roldão-b6273b149/

O que é framework?

Um dos principais objetivos é resolver problemas recorrentes com uma abordagem genérica, permitindo ao desenvolvedor focar no negócio.

O que é framework?

Em resumo... você não precisa reinventar a roda!



Exemplos framework

Porque utilizar framework?

Introdução ao NestJs

- Construção de aplicações eficientes e escaláveis.
- Suporta TypeScript
- Combina elementos de OOP (Programação Orientada a Objetos), FP (Programação Funcional) e FRP (Programação Reativa Funcional).

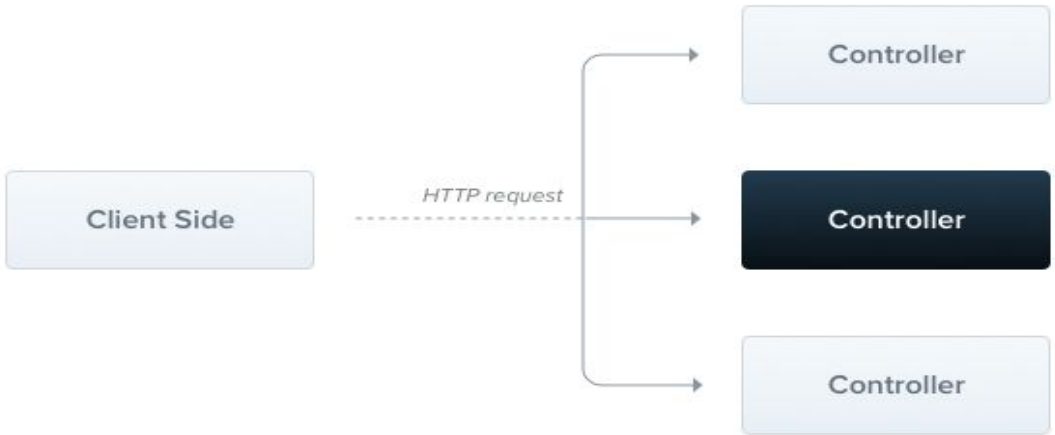
Introdução ao NestJs

Temos conceitos e componentes com diferentes responsabilidades dentro da aplicação utilizando NestJs

- **Controllers**
- **Providers**
- **Modules**
- **Middleware**
- **Exception filters**
- **Pipes**
- **Guards**
- **Interceptors**

Overview - Controllers

São responsáveis por lidar com as solicitações recebidas e retornar as respostas ao client.



Overview - Controllers

Requisição
GET http://localhost:3000/cats

Resposta do servidor >

```
{
    "data": [
        {
            "name": "zeca",
            "age": 2,
            "breed": "vira-lata"
        },
        {
            "name": "tom",
            "age": 3,
            "breed": "persa"
        }
    ]
}
```

Overview - Controllers

Para conseguir esse resultado precisamos adicionar a nossa classe de controle a anotação @Controller. Isso determina dentro do contexto da aplicação que essa classe será responsável pelo endpoint /cats

```
import { Body, Controller, Get, Param, Post, UseGuards } from '@nestjs/common';
import { CatsService } from './cats.service';
import { CreateCatDto } from './dto/create-cat.dto';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Post()
  create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }

  @Get()
  findAll(): Cat[] {
    return this.catsService.findAll();
  }
}
```

Overview - Providers

A ideia principal do provider é fornecer a injeção de dependências. Isso significa que o nestJs fica responsável por gerenciar as instâncias de memória que serão alocadas durante o uso do da aplicação.

```
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  create(cat: Cat) {
    this.cats.push(cat);
  }

  findAll(): Cat[] {
    return this.cats;
  }
}
```

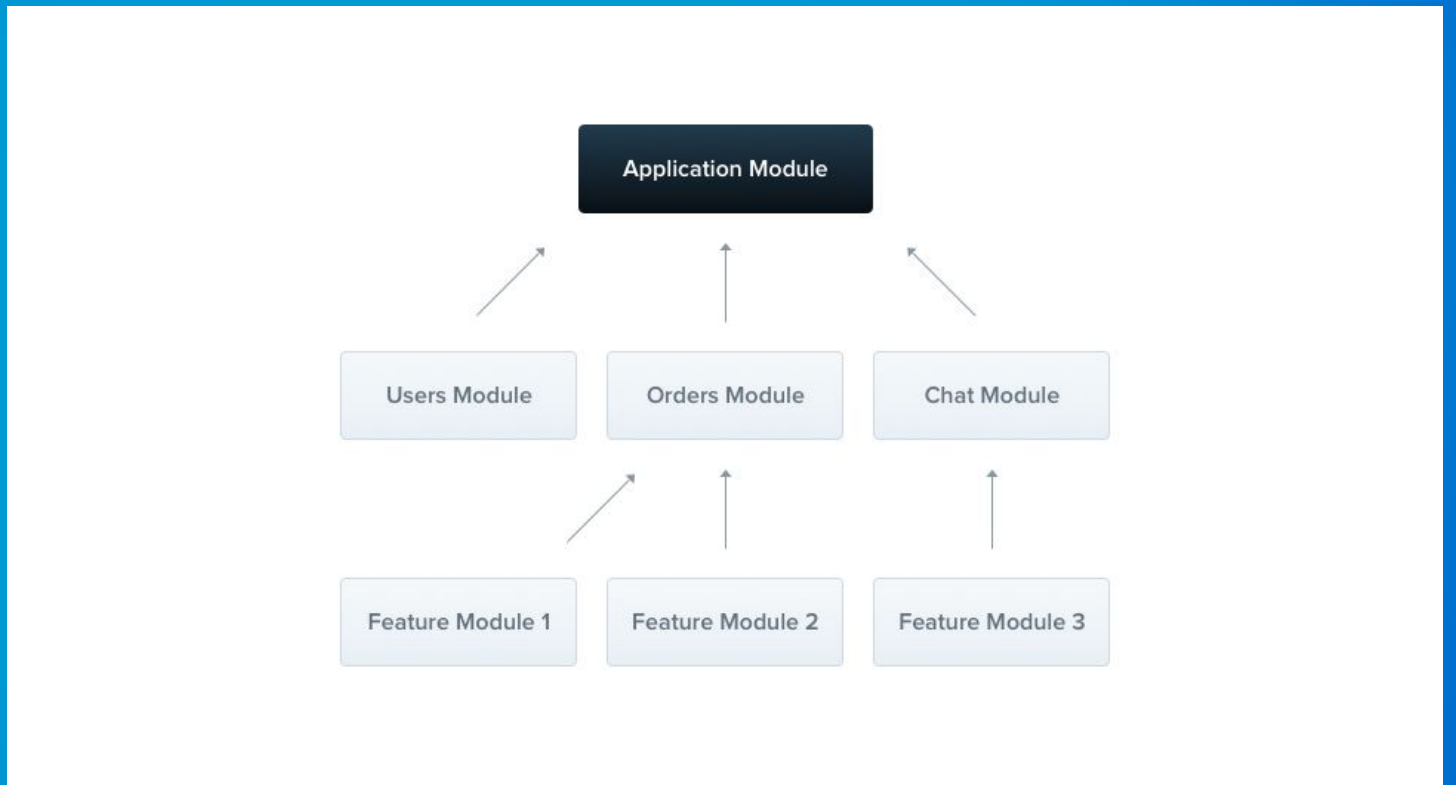
Overview - Modules

Definir um módulo nos ajuda a organizar o nosso código e fica claro quais providers (dependências) aquele módulo precisa no momento da execução

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

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

Overview - Modules

Toda aplicação feita em NestJs vai ter um módulo principal.



Overview - Modules

Nosso módulo principal definido como AppModule

```
import { Module } from '@nestjs/common';
import { CatsModule } from './cats/cats.module';
import { CoreModule } from './core/core.module';

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

Arquivo principal da nossa aplicação que define a porta e chama a função NestFactory.create passando o módulo principal

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

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

  await app.listen(3000);
  console.log(`Application is running on: ${await app.getUrl()}`);
}
bootstrap();
```

Overview - Middleware

Middleware é uma função que é chamada antes do Controller. Essa funções tem acesso nos objetos de request e response.
Exemplo: podemos definir um log da requisição no Middleware.

Overview - Exception Filters

O Nest possui uma camada responsável por tratar os erros do código e devolver uma resposta mais amigável para o consumidor da aplicação.

cats.controller.ts

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

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

Overview - Exception Filters

Podemos também customizar esses retornos através da implementação da interface ExceptionFilter

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

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

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

Overview - Exception Filters

O filtro pode ser aplicado no escopo de método, classe e global.

```
@Controller('cats')
@UseFilters(HttpExceptionFilter)
export class CatsController {
  constructor(private readonly catsService: CatsService) {}
```

```
{
    "statusCode": 403,
    "timestamp": "2021-03-06T19:47:12.679Z",
    "path": "/cats"
}
```

```
1  {
2      "statusCode": 403,
3      "message": "Forbidden"
4  }
```

Overview - Exception Filters

Podemos setar o filtro de maneira global pelo método useGlobalFilters() no arquivo main.ts

```ts
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { HttpExceptionFilter } from './common/filters/http-exception.filter

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  app.useGlobalFilters(new HttpExceptionFilter())
  await app.listen(3000);
  console.log(`Application is running on: ${await app.getUrl()}`);
}
bootstrap();
```

Overview - Pipes

Pipes geralmente são utilizados para:

- Transformação: transformar um dado especifico (Ex: String em integer)
- validação: verifica se o dado informado é valido, caso inválido lança uma Exception

O Nest já vem com alguns Pipes padrão (exportados pelo `@`) como por exemplo:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseBoolPipe`
- `ParseArrayPipe`
- `ParseUUIDPipe`
- `DefaultValuePipe`

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

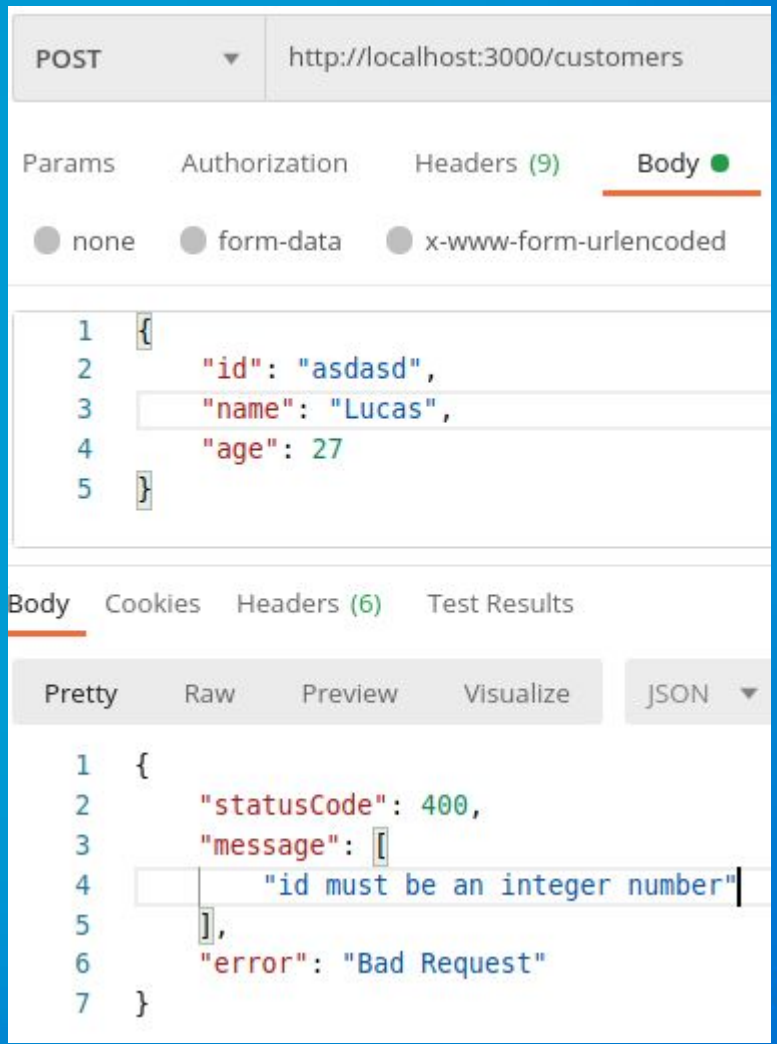
```
{
  "statusCode": 400,
  "message": "Validation failed (numeric string is expected)",
  "error": "Bad Request"
}
```

Overview - Pipes

Podemos fazer validação genérica combinando a biblioteca do Class validator com ValidationPipe

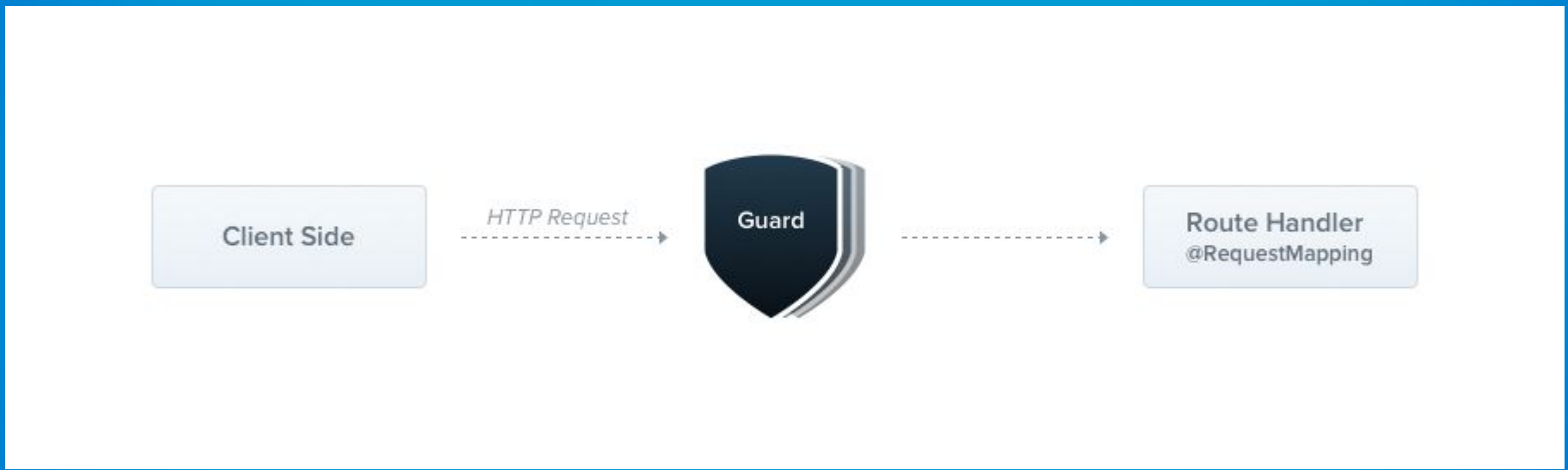
```ts
import { IsInt, IsString } from "class-validator"

export class CustomerRequestDto{

    @IsInt()
    readonly id: number

    @IsString()
    readonly name : string

    @IsInt()
    readonly age: number
}
```

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

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



NestJs - Guards

O Guard lida muito bem com questões de Autorização. Ex: Só usuário com role admin pode excluir um registro.



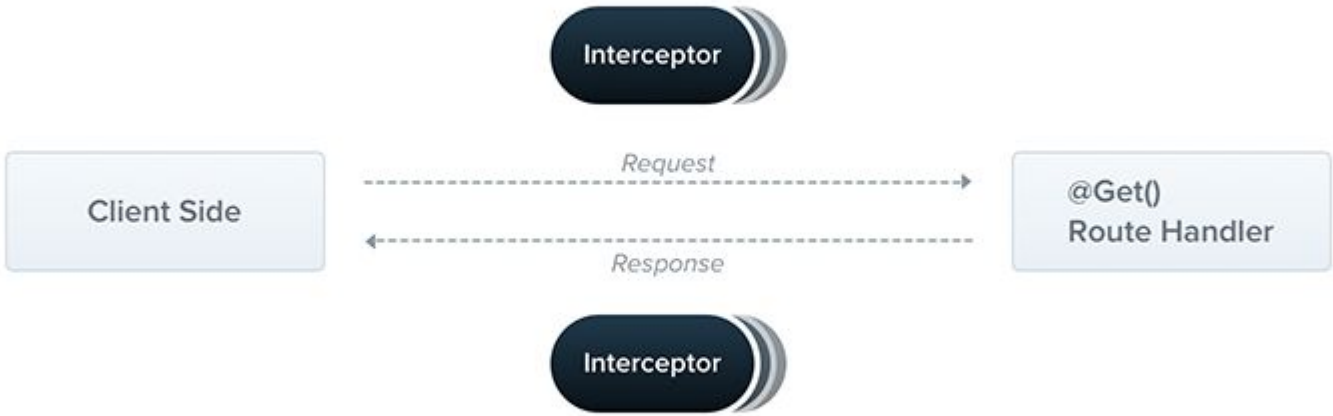
NestJs - Guards

Caso o usuário não tenha autorização para executar determinada operação ele vai receber um erro com status code 403.

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

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

NestJs - Interceptors



NestJs - Interceptors

CAMPINAS TECH TALENTS

Exemplo do código de um interceptor de log.
Ele pode ser utilizado a nível de método, classe ou global.

```
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

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

logging.interceptor.ts

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

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

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

Resultado no log **=>**

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

Dúvidas?

Bora praticar?