NestJs

Primeiro de tudo, antes de começar falando das coisas técnicas primeiro vou apresentar o que é o NestJs
Pra seguir nessa apresentação eu vou assumir que quase todo mundo aqui já tem um conhecimento sobre express, e sobre typescript, mas caso venham a aparecer é só perguntar ok? Perguntem antes de eu passar por cima e virar uma bola de neve de complexidade.

O que é:

O Nest é um framework javascript, backend em que a principal ideia é pegar aquela zona toda, sem padronização do express, e criar um padrão de arquitetura modular em cima do framework, ou seja, o nest está para o express assim como o quasar está para o vue.

Injeção de dependência:

Injeção de dependência é uma técnica para isolar a criação de um objeto da sua classe. A ideia é que quando a gente usa um mecanismo de injeção de dependência a gente não precise se preocupar em criar as instâncias das classes que iremos utilizar, e que a gente só saia dizendo quais queremos, é um componente arquitetônico chamado de **Container de Injeção de Dependência** vai se encarregar desse trabalho pra nós.

Exemplo de como um Container super simples funciona

```
class DependencyContainer {
  constructor() {
    this._dependencies = {};
  }

  register(type) {
    const instance = this.resolve(type);
    this._dependencies[type.name] = instance;
  }

  // Esse é o método responsável por pegar o nome da depedencia e encontrar
  // ela na lista de dependencias disponiveis
  resolve(type) {
    const dependencies = type._dependencies;

    // Caso o tipo não tenha dependencias, simplesmente retorne uma nova instancia
    // do mesmo
    if (!dependencies || dependencies.length == 0) {
      return new type();
    }

    // Vai pegar cada dependencia de dentro do container de acordo com o seu nome e retornar
    // um array com todas as instancias
    const resolvedDependencies = dependencies.map((name) => this.resolveDependencyByName(name));
    // Constroi o tipo passando todas as dependencias que ele precisa
    return new type(...resolvedDependencies);
  }

  resolveDependencyByName(dependencyName) {
    const dep = this._dependencies[dependencyName];

    if (!dep) {
      throw `${dependencyName} was not found in the container`;
    }

    return dep;
  }
}
```

E ele deve ser utilizado dessa forma:

```
class PersonStringValidator {
  isValidEmail(email) {
    return this.isOfTypeString(email) && this.hasAnAtCharacter(email);
  }

  hasAnAtCharacter(email) {
    return email.indexOf('@') !== -1;
  }

  isOfTypeString(email) {
    return typeof email === 'string';
  }
}

class Person {
  static _dependencies = ['PersonStringValidator'];

  constructor(personStringValidator) {
    this._personStringValidator = personStringValidator;
  }

  set email(email) {
    if (!this._personStringValidator.isValidEmail(email)) {
      throw new Error(`${email} is not a valid email`);
    }

    this._email = email;
  }
}

function main() {
  const dependencyContainer = new DependencyContainer();
  dependencyContainer.register(PersonStringValidator);

  const person = dependencyContainer.resolve(Person);
  person.email = 'gustavo@gmail.com';
  console.log(person);
}

main();
```

Componentes básicos:

O nest tem uma arquitetura muito complexa por si, mas pra que a gente consiga entender certinho como ela funciona eu vou mostrar os componentes básicos que são utilizados em uma aplicação nest

O Controller:

O controller em um projeto nest é um pouco diferente de como usamos no admin, aqui, o controller tem como unico propósito, receber dados do protocolo http, mandar eles para serem

processados, e dar o retorno de acordo com o que voltou do processamento.

Exemplo:

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

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

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

Esse é um controller bem simples, com só duas rotas, uma rota Post e um Get.

Um controller é sempre denominado pelo decorator @Controller, que pode receber um namespace para agrupar todas as rotas sem a necessidade de redundância, isso poderia ter sido feito de outra forma, passando a string dentro dos métodos, exemplo @Post(‘users’) indica que o método só vai ser chamado quando um post para uma rota /users acontecer

O Provider:

O provider é um termo simples, apesar de parecer meio complexo, o provider é diretamente relacionado ao modelo de injeção de dependência do framework, basicamente, providers são todas as classes que podem ser injetadas.
Nos providers nós colocaremos todas as nossas regras de negócio, assim como repositórios para poder acessar banco de dados, estratégias para autenticação, factories, etc.
Exemplo:

```
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;
  }
}
```

A anotação de Injectable é o que demarca um provider. Ela é necessária para que uma classe seja injetável dentro da outra.

Aqui, nesse caso, o CatsService está sendo utilizado como um repositório, e fazendo operações de busca e inserção.

Como ter uma classe que depende dessa:

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

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

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

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

Pode se ver que o service está sendo injetado através do construtor, em uma propriedade privada. Como o controller é criado pelo próprio framework, ele vai se encarregar de descobrir onde essa classe está e instanciar ela pra nós(olha que querido);

O Módulo:

Para declarar um novo módulo é só utilizar o decorator @Module

O módulo funciona como se fosse a gavetinha da aplicação, no módulo a gente vai agrupar todas as dependências, controllers e exportações de um domínio da aplicação. Ou seja, a gente vai usar os módulos para poder organizar o nosso app e para fazer com que o injetor de dependência consiga traçar os grafos de dependência para poder injetar nos componentes que precisam

Exemplo:

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

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

O decorator @Module recebe um objeto que permite que a gente possa configurar o escopo de cada um e exportar os que precisarmos, ele aceita 4 opções, são elas:

- **Providers**: Os providers que vão ser utilizados dentro dos componentes do módulo
- **Controllers**: O conjunto de controllers que o domínio possui
- **Imports:** Lista dos módulos que exportam os providers usados neste módulo
- **Exports:** O conjunto de providers deste módulo que devem ficar disponíveis para quem importar(usando imports) esse módulo

O Middleware:

O middleware é um componente que é sempre chamado antes do *Route Handler*, ou seja, antes do método do controller anotado pela rota que ele cairia.

A ideia do middleware é que todo código que deve ser executado antes ou depois do request ficaria em middlewares.

Middlewares são providers(eles têm o decorator @Injectable), porém um middleware é uma classe que implementa interface NestMiddleware

Exemplo de um middleware que printa ‘Request…’ no console a cada requisição:

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

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

Porém, diferente dos providers o middleware não pode ser importado dentro da lista de providers, para poder ter isso, iremos implementar a interface NestModule no nosso module e iremos implementar o método configure para definir quando executar o nosso middleware

Exemplo:

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

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

Os Exception Filters:

É a parte responsável por “pegar” todos os erros que “jogamos” ou são “jogados” durante a execução do nosso request, é a parte encarregada de pegar um erro e retornar um status code de erro ao invés de deixar o request pendente e o socket ocupado

Por padrão o nest já tem um exception filter rodando, que pega todas as exceções do tipo HttpException e as subclasses dela, ele é baseado na biblioteca http-erros, que pega todos os erros que tem as propriedades statusCode e message, e popula o erro de acordo com os valores.

Felizmente o nest já possui um monte de excessões padrão para que não seja necessário implementar filtros para todas

São as que vêm por padrão.

Para criar os nossos próprios filters é necessário criar uma classe que implementa a interface ExceptionFilter e que é anotada por Catch(<tipo-da-excessão>)

Exemplo:

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

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

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

Daí podemos escolher onde vamos bindar o nosso filtro, para fazer o bind no controller, fazemos assim:

```
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

Para aplicar no escopo global faríamos assim:

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

Os Pipes:

Os pipes são providers, que implementam a interface PipeTransform, e o seu principal uso é aplicar transformações no request antes de passar para a execução do método do controller, como por exemplo, converter para inteiro o id de um usuário passado pela url, que normalmente vem como string, e jogar uma exception(para ser pega pelos Exception FIlters) caso a conversão não seja possível.

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

As Guards:

Guard é um provider, que implementa a interface CanActivate, e a sua principal aplicação é validar que o request pode seguir. Guards são análogas às guard clauses, que são, aqueles ifs que a gente coloca para cair fora do comportamento principal de um método caso algo ocorra, para ficar evitando if/elses gigantescos.

Exemplo:

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
```

Aqui uma guard é implementada cujo objetivo é validar que o request está autenticado corretamente, (o método validateRequest é quem faz a validação, porém não vem ao caso como ela foi feita).

Para poder usar uma guard, colocamos ela junto ao decorator que define o tipo do request, ou podemos definir a nível de controller

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