Nest e o uso do Inversion of Control

Na aula anterior, nós vimos como o **pattern injeção** de **dependência** é utilizado no **Nest**, onde injetamos uma **instância** do nosso **Service Provider Jogadores** em nosso **Controller**, fazendo uso da **Injeção de dependência baseada no Construtor**

O que é Injeção de Dependência?

É uma **técnica** de **Inversão** de **Controle** (IoC), na qual **delegamos** a **instanciação** de **dependências** para o **IoC Container** (em nosso caso, o **runtime** do **NestJS**), ao invés de **fazermos** em nosso próprio código de **forma imperativa**

Recapitulando...

Primeiro, nós definimos um Provider. O decorator @Injectable tornou nossa classe JogadoresService um Provider.

```
import { Injectable, Logger, NotFoundException } from '@nestjs/common';
import { CriarJogadorDto } from './dtos/criar-jogador.dto'
import { Jogador } from './interfaces/jogador.interface'
import * as uuid from 'uuid/v1'

@Injectable()
export class JogadoresService {

    private jogadores: Jogador[] = [];

    private readonly logger = new Logger(JogadoresService.name)

    async criarAtualizarJogador(criaJogadorDto: CriarJogadorDto): Promise<void> {
        await this.criar(criaJogadorDto)
    }
```

Em seguida, pedimos para o Nest injetar o provider dentro de nossa classe Controller.

```
import { Controller, Post, Body, Get, Query, Delete } from '@nestjs/common';
import { CriarJogadorDto } from './dtos/criar-jogador.dto'
import { JogadoresService } from './jogadores.service'
import { Jogador } from './interfaces/jogador.interface'


@Controller('api/v1/jogadores')
export class JogadoresController {

    constructor(private readonly jogadoresService: JogadoresService) {}

    @Post()
    async criarAtualizarJogador(
        @Body() criaJogadorDto: CriarJogadorDto) {
        await this.jogadoresService.criarAtualizarJogador(criaJogadorDto)
    }
```

Finalmente, registramos nosso provider no Nest IoC Container

jogadores.module.ts 275 Bytes

```ts
import { Module } from '@nestjs/common';
import { JogadoresController } from './jogadores.controller';
import { JogadoresService } from './jogadores.service';

@Module({
  controllers: [JogadoresController],
  providers: [JogadoresService]
})
export class JogadoresModule {}
```

Mas o que exatamente está acontecendo por debaixo dos panos, para tudo isso funcionar?

Existem três etapas principais nesse processo:

1- Em jogadores.service.ts, o decorator @Injectable **declara** a classe **JogadoresService** como uma classe que pode ser **gerenciada** pelo **Nest IoC Container**

2- Em jogadores.controller.ts, JogadoresController **declara** uma **dependência** do **token JogadoresService** com a injeção de dependência baseada no construtor:

```
constructor(private readonly jogadoresService: JogadoresService) {}
```

3- Em app.module.ts, nós **associamos** o **token JogadoresService** com a **classe JogadoresService** do arquivo jogadores.service.ts

```
@Module({
  controllers: [JogadoresController],
  providers: [JogadoresService]
})
```

E como essa associação, também chamada de registro acontece?

1- Quando o **Nest IoC Container instancia** um **JogadoresController**, ele primeiro **procura** por quaisquer **dependências**

2- Quando o **container encontra** a **dependência** de **JogadoresService**, realiza uma **pesquisa** pelo **token** JogadoresService, que **retorna** a classe **JogadoresService**, utilizando a etapa de **registro** mencionada anteriormente

3- Assumindo o **escopo SINGLETON** (comportamento padrão), o **Nest criará** uma **instância** de **JogadoresService**, armazenará em cache e retornará, ou, se já estiver em cache, retornará a instância existente!