Comandos iniciais:  
 **node -v** (verificar versão do node instalado)  
 **npm -v** (verificar versão do npm instalado)  
 **npm i -g** @nest/cli (instalar o NestJS na máquina)  
 **nest -v** (verifica a versão do NestJS instalada)

NestJS – aplicações escaláveis em NodeJS – arquitetura orientada a eventos

Comando:  
**nest new nome-do-projeto**  
*package manager escolhido: npm  
foram deletados: controller.spec, controller e service + dependências (serão recriados manualmente)  
diretório de tests também foi deletado   
module e main ficaram*

Main:

import { NestFactory } from '@nestjs/core';

import { AppModule } from './app.module';

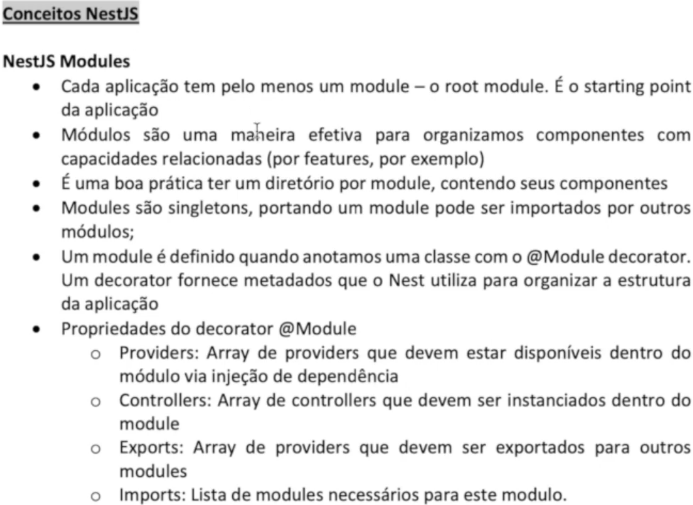
async function bootstrap() {

  const app = await NestFactory.create(AppModule);

  await app.listen(8080);

}

bootstrap();



Module principal da aplicação:

import { Module } from '@nestjs/common';

@Module({

  imports: [],

  controllers: [],

  providers: [],

})

export class AppModule {}

Comando para rodar o projeto (precisa estar rodando pra conferir via navegador):  
 **npm run start:dev**

**Criação do module de jogadores**

Comando para criar o module jogadores  
 **nest g module jogadores**

import { Module } from '@nestjs/common';

import { JogadoresController } from './jogadores.controller';

import { JogadoresService } from './jogadores.service';

@Module({

  controllers: [JogadoresController],

  providers: [JogadoresService]

})

export class JogadoresModule {}

Aqui ele já está com o Controller e o Service adicionados. “Pelado” ele estaria sem as linhas vermelhas. Como o module foi criado pelo comando ali em cima, ele já é injetado no módulo principal da aplicação (nos imports).

**Interface:** no caso deste projeto, é onde é feita a definição dos atributos, como se fosse uma Classe. Ela não é criada por comando.

export interface Jogador {

  readonly \_id: string;

  readonly celular: string;

  readonly email: string;

  nome: string;

  ranking: string;

  posicaoRanking: number;

  urlFotoJogador: string;

}

**DTO:** é o que vai ser enviado para a View. É o que vai ser recebido no POST, pois, neste caso, os outros atributos são obtidos através de métodos. “O que o cliente vai mandar no contexto da criação.”

export class CriarJogadorDto{

  readonly celular: string;

  readonly email: string;

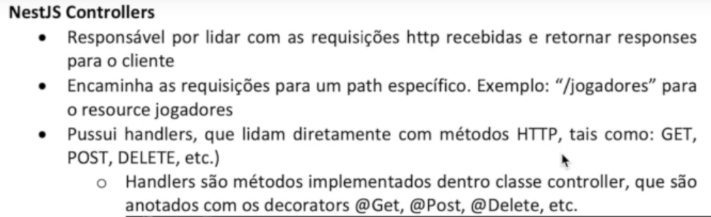
  readonly nome: string;

}

**Controller** (inicialmente)

Comando para criar:  
**nest g controller jogadores**

Como o controller foi criado pelo comando, ele já é injetado no módulo jogadores.



1 import { Body, Controller, Post } from '@nestjs/common';

2 import { CriarJogadorDto } from './dtos/criar-jogador.dto';

3 import { JogadoresService } from './jogadores.service';

4

5 @Controller('api/v1/jogadores')

6 export class JogadoresController {

7

8   constructor(private readonly jogadoresService: JogadoresService){}

9

10   @Post()

11   async criarAtualizarJogador(

12

13     @Body() criarJogadorDto: CriarJogadorDto){

14

15      await this.jogadoresService.criarAtualizarJogador(criarJogadorDto)

16     }

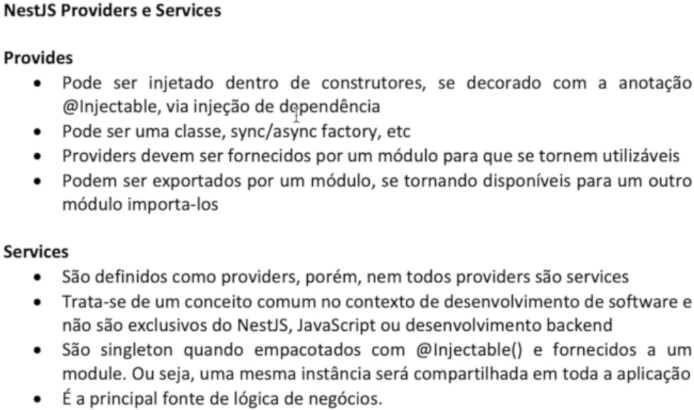
17 }

Os @ são “decorators”. Controller é responsável por lidar com as requisições http (e com as responses).

A ideia aqui foi receber um Post e a partir dele já retornar um texto para o cliente (criar E atualizar um jogador)

O **@Body** ali é pq ta vindo na requisição http um body do tipo jogador. Ele então é tipado como jogadorDto

Aí entra o conceito de Services e Providers -> as requisições são direcionadas



- Pode ser classe, sync, async, factory (não sei o que é este último)

**Service**: todo Service é um Provider (nem todo Provider é um Service)

Comando para criar:  
**nest g service jogadores**

- vai fazer a persistência em memória

Obs.: foi necessário instalar um biblioteca pra criar o id

Comando pra instalar:  
 **npm install uuid**

Método POST criarAtualizarJogador

No controller (obs.: alguns desses imports vieram depois):

import { Body, Controller, Post, 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() criarJogadorDto: CriarJogadorDto){

    await this.jogadoresService.criarAtualizarJogador(criarJogadorDto)

  }

O Controller encaminha a requisição para o Service

import { Injectable, Logger } from '@nestjs/common';

import { CriarJogadorDto } from './dtos/criar-jogador.dto';

import { Jogador } from './interfaces/jogador.interface';

import { v4 as uuidv4 } from 'uuid';

@Injectable()

export class JogadoresService {

  private jogadores: Jogador[] = [];

  private readonly logger = new Logger(JogadoresService.name);

  async criarAtualizarJogador(criaJogadorDto: CriarJogadorDto): Promise<void>{

    await this.criar(criaJogadorDto)

  }

  private criar(criaJogadorDto: CriarJogadorDto): void{

    const { nome, email, celular} = criaJogadorDto;

    const jogador: Jogador = {

      \_id: uuidv4(),

      nome,

      celular,

      email,

      ranking: 'A',

      posicaoRanking: 1,

      urlFotoJogador: 'www.foto.com.br/foto.jpg'

    };

    this.logger.log(`criaJogadorDto: ${JSON.stringify(jogador)}`);

    this.jogadores.push(jogador);

  }

}

Neste Service, o logger está simulando uma persistência na memória (só pra conferir se tá tudo dando certo)

Promise<void> não sei o que isso faz [se entendi, o Promise é usado por causa do async e o void é pq o método não devolve nada

**OBS.: A AULA 15 É UMA AULA TEÓRICA IMPORTANTE SOBRE INJEÇÃO DE DEPENDÊNCIA EM NESTJS. REVÊ-LA QUANDO RETORNAR A ESSE ARQUIVO.**

Implementações feitas no **GET**:

Retornar todos os jogadores

Controller:

  @Get()

async consultarJogadores(): Promise<Jogador[]> {

    return this.jogadoresService.consultarTodosJogadores();

  }

Service:

async consultarTodosJogadores(): Promise<Jogador[]>{

    return await this.jogadores;

  }

Retornar um jogador pesquisando pelo e-mail (é o método acima com mais o outro)

Controller (importante observar que é um método só pra getAll e getById (email no lugar do id):

@Get()

  async consultarJogadores(

    @Query('email') email: string): Promise<Jogador[] | Jogador> {

      if(email){

        return await this.jogadoresService.consultarJogadorPeloEmail(email);

      } else {

        return await this.jogadoresService.consultarTodosJogadores();

      }

  }

Service (aqui são dois métodos diferentes pra getAll e getById):

async consultarJogadorPeloEmail(email: string): Promise<Jogador>{

    const jogadorEncontrado = this.jogadores.find(jogador => jogador.email === email);

    if(!jogadorEncontrado){

      throw new NotFoundException(`Jogador com e-mail ${email} não encontrado.`);

    }

    return jogadorEncontrado;

  }

Delete:

controller

  @Delete()

  async deletarJogador(

    @Query('email') email: string): Promise<void>{

      this.jogadoresService.deletarJogador(email);

    }

Service

  async deletarJogador(email): Promise<void>{

    const jogadorEncontrado = this.jogadores.find(jogador => jogador.email === email);

    if(!jogadorEncontrado){

      throw new NotFoundException(`Jogador com e-mail ${email} não encontrado.`);

    }

    this.jogadores = this.jogadores.filter(jogador => jogador.email !== jogadorEncontrado.email);

    //vai filtrar/retornar todo mundo que tiver email diferente do jogadorEncontrado

  }

Conexão com o MongoDB

1. foi criada conta em mongodb.com

Com essa conta, criou-se um cluster (o procedimento é bem basicão)

Configuração de conexão:

Obs. A string de conexão é o **const uri**

const { MongoClient } = require('mongodb');

const uri = "mongodb+srv://admin:Pp159753@cluster0.9nues.mongodb.net/smartranking?retryWrites=true&w=majority";

const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true });

client.connect(err => {

const collection = client.db("test").collection("devices");

// perform actions on the collection object

client.close();

});

Foi usado o mongoDB Compass para fazer a conexão com o bd externo

A partir daí, será feita a comunicação entre a aplicação e o banco, usando o mongoose

1. Refatorar a Interface e criar o Schema

Antes disso, instalação de pacotes:  
**npm install @nestjs/mongoose mongoose  
npm install –save-dev @types/mongoose**   
(esse último comando não fez nenhuma instalação, pq “mongoose publishes its own types”. Provavelmente é um comando obsoleto)

Dependências instaladas, hora de voltar ao **app.module.ts.** Ele usa bibliotecas pra fazer conexão com o mongodb. Pra isso, é necessário importar o MongooseModule.

import { Module } from '@nestjs/common';

import { MongooseModule } from '@nestjs/mongoose';

import { JogadoresModule } from './jogadores/jogadores.module';

@Module({

  imports: [

    MongooseModule.forRoot('mongodb+srv://admin:Pp159753@cluster0.9nues.mongodb.net/smartranking?retryWrites=true&w=majority'),

    JogadoresModule],

  controllers: [],

  providers: [],

})

export class AppModule {}

**JogadorSchema**:

import \* as mongoose from 'mongoose';

export const JogadorSchema = new mongoose.Schema({

     telefoneCelular: { type: String, unique: true },

     emaail: { type: String, unique: true },

     nome: String,

     ranking: String,

     posicaoRanking: Number,

     urlFotoJogador: String

}, {timestamps: true, collection: 'jogadores'});

O schema é modelo a ser utilizado quando vai fazer a persistência no mongodb. Todas as interfaces precisam ter um schema.

a **interface** passa por mudanças menores, apenas passa a utilizar a importação de Document do mongoose. Além disso, o id foi removido, pois passa a ser responsabilidade do mongodb

import { Document } from "mongoose";

export interface Jogador extends Document {

  readonly celular: string;

  readonly email: string;

  nome: string;

  ranking: string;

  posicaoRanking: number;

  urlFotoJogador: string;

}

JogadoresModule também precisa importar mongoose e vai ficar assim

import { Module } from '@nestjs/common';

import { JogadoresController } from './jogadores.controller';

import { JogadoresService } from './jogadores.service';

import { MongooseModule } from '@nestjs/mongoose';

import { JogadorSchema } from './interfaces/jogador.schema';

@Module({

  imports: [MongooseModule.forFeature([{ name: 'Jogador', schema: JogadorSchema }])],

  controllers: [JogadoresController],

  providers: [JogadoresService]

})

export class JogadoresModule {}

a partir daí, vamos para JogadoresService, que vai ser praticamente refeito

dentro do construtor, será injetado um JogadorModel, que vai trazer métodos para viabilizar a comunicação com o BD

devido à conexão com o mongoose, os métodos criados precisarão ser refatorados.

import { Injectable, Logger, NotFoundException } from '@nestjs/common';

import { CriarJogadorDto } from './dtos/criar-jogador.dto';

import { Jogador } from './interfaces/jogador.interface';

import { v4 as uuidv4 } from 'uuid';

import { InjectModel } from '@nestjs/mongoose';

import { Model } from 'mongoose';

/\*##############################################################################

####### AS LINHAS DE CÓDIGO COMENTADAS ABAIXO SÃO CÓDIGO FEITO PARA      #######

####### FUNCIONAR APENAS EM MEMÓRIA, SEM O BD E SEM MÉTODOS DO MONGOOSE. #######

##############################################################################\*/

@Injectable()

export class JogadoresService {

  constructor(@InjectModel('Jogador') private readonly jogadorModel: Model<Jogador>){}

  //private jogadores: Jogador[] = [];

  //private readonly logger = new Logger(JogadoresService.name);

  async criarAtualizarJogador(criarJogadorDto: CriarJogadorDto): Promise<void>{

    const {email} = criarJogadorDto;

    const jogadorEncontrado = await this.jogadorModel.findOne({email}).exec();

    //const jogadorEncontrado = this.jogadores.find(jogador => jogador.email === email);

    if(jogadorEncontrado){

      await this.atualizar(criarJogadorDto);

    } else {

      await this.criar(criarJogadorDto);

    }

  }

  async consultarTodosJogadores(): Promise<Jogador[]>{

    return await this.jogadorModel.find().exec();

    //return this.jogadores;

  }

  async consultarJogadorPeloEmail(email: string): Promise<Jogador>{

    const jogadorEncontrado = await this.jogadorModel.findOne({email}).exec();

    if(!jogadorEncontrado){

      throw new NotFoundException(`Jogador com e-mail ${email} não encontrado.`);

    }

    return jogadorEncontrado;

  }

  async deletarJogador(email): Promise<any>{

    return await this.jogadorModel.deleteOne({email}).exec();

    /\*

    const jogadorEncontrado = this.jogadores.find(jogador => jogador.email === email);

    if(!jogadorEncontrado){

      throw new NotFoundException(`Jogador com e-mail ${email} não encontrado.`);

    }

    this.jogadores = this.jogadores.filter(jogador => jogador.email !== jogadorEncontrado.email);

    //vai filtrar/retornar todo mundo que tiver email diferente do jogadorEncontrado

    \*/

  }

  private async criar(criarJogadorDto: CriarJogadorDto): Promise<Jogador>{

    const jogadorCriado = new this.jogadorModel(criarJogadorDto);

    return await jogadorCriado.save();

    /\*

    const { nome, email, celular} = criarJogadorDto;

    const jogador: Jogador = {

      \_id: uuidv4(),

      nome,

      celular,

      email,

      ranking: 'A',

      posicaoRanking: 1,

      urlFotoJogador: 'www.foto.com.br/foto.jpg'

    };

    this.logger.log(`criarJogadorDto: ${JSON.stringify(jogador)}`);

    this.jogadores.push(jogador);

    \*/

  }

  private async atualizar(criarJogadorDto: CriarJogadorDto): Promise<Jogador>{

    return await this.jogadorModel.findOneAndUpdate({email: criarJogadorDto.email}, {$set: criarJogadorDto}).exec();

    //os parametros de findOneAndUpdate são: 1. o que vai ser buscado pra achar o objeto no BD (no caso, o email) e

    //2. o objeto que será usado para que seja feito o update

    /\*

    const { nome } = criarJogadorDto;

    jogadorEncontrado.nome = nome;

    \*/

  }

}