در این فایل نحوه register , login , logout رو داریم .

که سمت بک اند با nestjs انجام شده .

سمت فرانت اند با nextjs انجام شده .

هر دو فلودر front و back یک فایل readme درونشون دارند

که مراحل پروژه رو توضیح میدن

Nestjs :

برنامه nestjs رو با دستور nest new شروع می کنیم و یک برنامه nestjs ایجاد می کنیم .

nest new project-name

اما اگر بار اول هست که می خوایم از nest استفاده کنیم نیاز هست اول cli اونو نصب کنیم .

و بعد دستور nest new رو انجام بدیم .

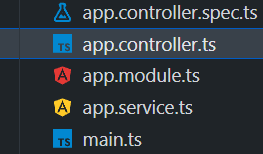
$ npm i -g @nestjs/cli

زمانی که پروژه نصب میشه با دستور nest new یک سری فایل برای ما ایجاد می کنه

App.controller

App.service

App.module



که در واقع در این پروژه نیاز به کد زدن درون فایل های controller , service نداریم .

اما پاک شونم نمی کنیم .

فایل مورد نیاز ما app.module.ts هست که بعد فایل های که ایجاد می کنیم

درون این app.module.ts میاد import میشه و درون کل پروژه اجرا میشن .

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

import { AppController } from './app.controller';

import { AppService } from './app.service';

import { UserModule } from './user/user.module';

import { TodoModule } from './todo/todo.module';

import { PrismaModule } from './prisma/prisma.module';

import { AuthService } from './auth/auth.service';

import { AuthModule } from './auth/auth.module';

@Module({

  imports: [UserModule, TodoModule, PrismaModule, AuthModule],

  controllers: [AppController],

  providers: [AppService, AuthService],

})

export class AppModule {}

همون طور که می بینید فایل های زیادی درون این فایل app.module داریم که import شدن .

که در اینده این فایل ها ایجاد میشن .

یک فایل **main.ts** داریم که هسته اصلی پروژه nestjs ما هستش

که جلو تر در موردش صحبت می کنیم .

بعد از ایجاد پروژه dependencies مورد نیاز خودمون رو نصب می کنیم .

  "dependencies": {

    "@prisma/client": "^5.5.2",

    "@types/bcryptjs": "^2.4.6",

    "@types/jsonwebtoken": "^9.0.5",

    "bcryptjs": "^2.4.3",

    "class-transformer": "^0.5.1",

    "class-validator": "^0.14.0",

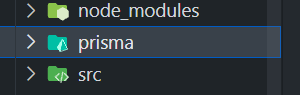
    "jsonwebtoken": "^9.0.2",

    "prisma": "^5.5.2",

  },

در قدم بعدی یک فولدر به اسم prisma ایجاد می کنیم درون خوده دایرکتوری اصلی پروژه که درون این prisma نوع دیتا بیس رو مشخص می کنیم .

در واقع دیتا بیس ما به prisma متصل میشه و همچنین مدل های خودمون رو ایجاد می کنیم .



درون این فولدر یک فایل به اسم schema.prisma داریم .



درون فایل schema.prisma مدل ها و تعریف شدن دیتابیس برای prisma رو داریم .

اتصال دیتا بیس به prisma :

generator client {

    provider = "prisma-client-js"

}

datasource db {

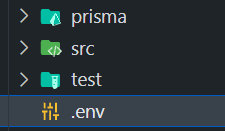
    provider = "mongodb"

    url      = env("DATABASE\_URL")

}

که همونطور می بینید در این پروژه از دیتابیس mongodb استفاده شده و اینجا mongodb متصل شده به prisma

یک فایل env نیاز داریم که در دایرکتوری اصلی بیرون از src ساخته میشه همراه با دات



درون این فایل میام متغییر های خودمون رو که نیاز داریم تعریف می کنیم در حال حاضر

یک DATABASE\_URL رو داریم .

DATABASE\_URL="mongodb+srv://mnazgul:nazgulolij8303893mohsen@cluster0.zutazhf.mongodb.net/NestjsTodo"

که ادرس دیتابیس mongodb ما هستش که از این متغیر در فایل prisma برای اتصال استفاده شده

datasource db {

    provider = "mongodb"

    url      = env("DATABASE\_URL")

}

در ون url امدیم env رو صدا زدیم و متغییر ِDATABSE\_URL رو قرار دادیم البته درون یک string هست .

قدم بعدی ساخت مدل ها و relation ship های که نیاز داریم هستش :

در این پروژه کلا دو مدل داریم . یک مدل user و یک مدل todo .

که در این فایل صرفا با مدل user کار می کنیم در کد ها.

و قصد ما login , register , logout هست و این فایل بخشی از یک پروژه دیگه هست .

مدل todo که داریم به صورت زیر هستش :

model Todo {

    id        String   @id @default(auto()) @map("\_id") @db.ObjectId

    title     String

    text      String

    create\_at DateTime @default(now())

    update\_at DateTime @updatedAt

    // relation to user

    userId    String   @db.ObjectId

    user      User     @relation(fields: [userId], references: [id])

}

همونطور که می بینید در userId و user در اصل اینجا relation زدیم به مدل User درون model های که با schema.prisma ایجاد می کنیم به این شکل relation ship ایجاد می کنیم .

**نکته :** برای id نوع دیتا بیس ها تاثیر گذار هست یعنی اگر دیتابیس دیگه ای استفاده می کردیم id که تعریف شده طوره دیگه تعریف می شد و برای mongodb به صورت زیر id تعریف میشه اگر این مورد برای id ها رعایت نشه به مشکل و ارور بر می خوریم . نوع id زیر برای mongodb هست .

    id        String   @id @default(auto()) @map("\_id") @db.ObjectId

مدل دیگه ای که داریم User هستش که به صورت زیر تعریف میشه :

model User {

    id              String   @id @default(auto()) @map("\_id") @db.ObjectId

    email           String   @unique

    password        String

    confirmPassword String

    create\_at       DateTime @default(now())

    update\_at       DateTime @updatedAt

    role            UserRole @default(USER)

    // relation to todo

    Todo            Todo[]

}

که همونطور که می بینید Todo رو داریم که برابر با Todo[] هست که اشاره می کنه به relation ship که مدل user با todo داره .

نکته : خوده prisma به طور خودکار متوجه هاrelation ship میشه موقع ایجاد و به صورت خودکار کد ها رو ایجاد می کنه .

برای role یک enum تعریف شده که میاد role کاربر های رو مشخص می کنه

enum UserRole {

    ADMIN

    USER

}

درون role یک default تعریف شده که نشون دهنده این هست که می خوایم به طور دیفالت نوع کاربر همیشه USER باشه که بعد درون پنل کاربری در اینده نوع کاربر ها رو عوض کنیم و به USER یا ADMIN تغییر بدیم .

خب بعد از تکمیل فایل schema.prisma نیاز به ساخت فولدر یک فولدر prsima دیگه داریم .

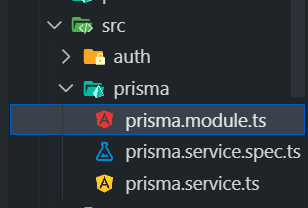
که درون خوده فولدر src ساخته میشه

**Prisma in src folder :**

یک فایل دیگه **prisma** هم نیاز داریم برای اینکه بتونیم در user.service.ts بیام از متد ها دیتا بیس

استفاده کنیم . در اصل این فایل ارتباط prisma با مدل و دیتا بیس رو کامل می کنه .

این فایل درون فولدر src ایجاد میشه .



که حاوی prisma.module.tsو prisma.service.ts هست .

درون prisma.module.ts کد های زیر رو داریم :

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

import { PrismaService } from './prisma.service';

@Module({

  providers: [PrismaService],

  exports: [PrismaService],

})

export class PrismaModule {}

که در اصل PrismaService امده import شده و درون @module قرار داده شده .

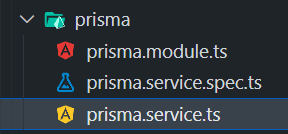
به این صورت می تونیم به مدل های که درون prisma ساختیم دسترسی داشته باشیم .

و اتصال prisma با data base به طور کامل تکمیل شده .

همونطور که می بینید نیاز هست که PrismaService رو exports کردیم .

  exports: [PrismaService],

درون خوده فایل prisma.service.ts میایم اتصال به دیتابیس رو کامل می کنیم .



اول import های که نیاز داریم رو انجام میدیم :

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

import { PrismaClient } from '@prisma/client';

import { OnModuleDestroy, OnModuleInit } from '@nestjs/common';

یدونه کلاس نیاز داریم ایجاد کنیم و export کنیم .

@Injectable()

export class PrismaService

  extends PrismaClient

  implements OnModuleInit, OnModuleDestroy {}

این کلاس یک بار extends میشه از PrismaClient یک بار هم implements میشه از OnmoduleInit و OnModuleDestory .

خب اونی که extends شده در اصل به مدل اتصال شده .

اونی که implements شده در اصل برای connect شدن و disconnect شدن دیتابیس ازشون استفاده میشه .

درون کلاس 2 تا متد داریم برای که برای اتصال از $connect استفاده شده و برای قطع اتصال از

$disconnect استفاده شده .

@Injectable()

export class PrismaService

  extends PrismaClient

  implements OnModuleInit, OnModuleDestroy

{

  async onModuleInit() {

    this.$connect();

  }

  async onModuleDestroy() {

    this.$disconnect();

  }

}

همین دیگه فایل prisma.service.ts ما چیزه خاصی نداره . که این فایل رو درون prisma.module.ts میایم استفاده می کنیم .

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

import { PrismaService } from './prisma.service';

@Module({

  providers: [PrismaService],

  exports: [PrismaService],

})

export class PrismaModule {}

که حالا در ادامه می تونیم هر جا در هر service یا کنترلی که در اینده میسازیم به مدل های مورد نیازمون دسترسی داشته باشیم و از متد های دیتا بیس که در اصل وصل شده به prisma استفاده کنیم

در ادامه User رو میسازیم .

**ساخت فایل های User :**

**درون فولدر user میایم دستورات رو می زنیم برای ساخت service , module , controller**

که درون فولدر user 3 تا فایل module , controller , service , داریم .

که هر سه با دستور زیر می تونیم ایجاد کنیم در واقع generate کنیم .

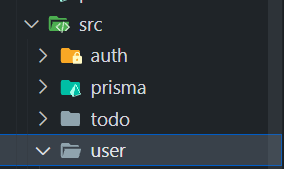
nest –g module user

nest – g controller user

nest –g service user

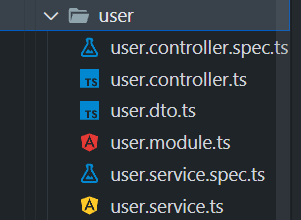
اینجوری به طور خودکار فایل ها generate میشن

الان در پروژه یک فولدر user داریم که درون فایل src ایجاد شده .



که درون فولدر user فایل های که ایجاد کردیم رو می تونیم ببینیم .

همون controller , service , module .



برای فایل service و controller یک فایل تست هم ساخته میشه .

که کاری بهشون نداریم البته با دستوری خاص می تونیم مانع ایجاد فایل تست شیم

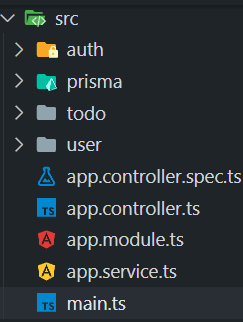
که فعلا کاری بهش نداریم .

**فایل main.ts :**

این فایل هسته اصلی پروژه nestjs ما هستش نیاز هست که یک سری تغییرات درونش انجام بدیم

مثل enabale کردن cors policy مثل ایجاد کردن کوکی اگر نیاز بود و ..

پس اینکارو انجام میدیم :



درون فایل main.ts کد های زیر رو داریم :

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

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

import \* as cookieParser from 'cookie-parser';

async function bootstrap() {

  const app = await NestFactory.create(AppModule);

  app.enableCors({

    origin: 'http://localhost:3000',

    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS',

    credentials: true,

  });

  app.use(cookieParser());

  await app.listen(2023);

}

bootstrap();

اگر دقت کنید در enableCors امدیم origin ست کردیم همینطور credentials رو true کردیم برای

استفاده از کوکی و این مهمه در نهایت method ها مورد استفاده رو قرار دادیم که اون origin می تونه

از چه متد های استفاده کنه :

  app.enableCors({

    origin: 'http://localhost:3000',

    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS',

    credentials: true,

  });

در ادامه از cookie-parser استفاده کردیم اگر بخوایم درون پروژه از cookie استفاده کنیم نیاز داریم که

هم origin و credentials رو تعریف کنیم که در بالا انجام دادیم .

هم نیاز داریم که از coockie-parser که یک پکیج هست اسفتاده کنیم البته اول به وسیله npm اونو

نصب می کنیم و بعد درون فایل main.ts میایم صداش می زنیم .

  app.use(cookieParser());

در ادامه listen رو داریم که نشون میده پروژه ما در چه port داره ران میشه و می تونیم اینجا port رو در صورت نیاز تغییر بدیم و عوض کنیم .

  await app.listen(2023);

یا به اینصورت می تونیم انجام بدیم listen رو :

  const port = 2023 || process.env.PORT;

  await app.listen(port, () => {

    console.log(`server is run at http://localhost:${port}`);

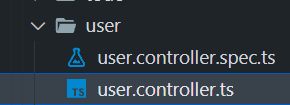
  });

**نکته :** برای انلاین کردن پروژه حتما نیاز به متغیر PORT داریم که درون فایل env تعریف می کنیم .

PORT="2023"

**خب بر گردیم به فولدر user :**

**user.controller.ts :**



خب درون این فایل قرار که بیایم method ها رو تعریف کنیم .

که شامل get , post , update, delete, هستند .

توجه داشته باشید درون nestjs این متد ها همشون decorator هستند .

و تمامی decorator ها با علامت @ تعریف میشن .

در اصل decorator ها کلاس یا فانکشن های هستند که یک کاری رو به عهده دارند .

و حتی می تونیم به طور costum با توجه به نیازمون decorator ایجاد کنیم درون nestjs .

میایم decorator های مورد نیاز خودمون رو import می کنیم از @nestjs/common

import {

  Body,

  Controller,

  Post,

  Get,

  Response,

  Request,

  Req,

} from '@nestjs/common';

یک controller تعریف می کنیم

@Controller('user')

که این decorator میاد route کلی برای تمامی method ها در نظر میگیره

در اینجا route کلی متد های ما شده /user در واقع به این شکل localhost:2023/user

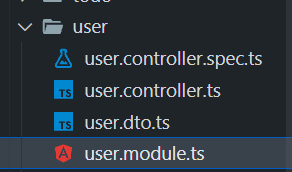
قدم بعدی زیر این controller میایم یک کلاس ایجاد می کنیم و اونو export هم می کنیم :

export class UserController {

}

درون این کلاس قرار هست method های مورد نیاز خودمون رو تعریف کنیم get ,put و ...

Export می کنیم چون قرار درون user.module.ts ازش استفاده کنیم .



اینجا امدیم کلاس که درون user.controller.ts ساختیم رو import کردیم

import { UserController } from './user.controller';

و درون controllers استفاده کردیم و اونو گذاشتیم داخل یک [ ] که سینتکس خوده nestjs هست

@Module({

  imports: [PrismaModule, AuthModule],

  controllers: [UserController],

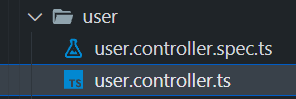
  providers: [UserService],

})

export class UserModule {}

خب بریم ادامه user.controller.ts صرفا می خواستیم بگیم که چرا کلاس که ساختیم رو

امدیم export کردیم .



حلا نوبت به این میرسه که یک constractor تعریف کنیم درون این کلاس که ایجاد کردیم :

export class UserController {

  constructor(private readonly userService: UserService) {}

{

که در اصل در بر گیرنده UserService هست .

که هنوز سراغ user.service.ts نرفتیم اما در کل هر فایل controller.ts معمولا اتصال پیدا می کنه به فایل service .

که درون controller میایم متد ها و ادرس route ها رو تعریف می کنیم .

و درون service میایم logic و اتفاقی که می خوایم درون اون route بیوفته رو کد می زنیم .

اینجور فایل ها از هم تفکیک شده هستند و کد کمتری داریم .

دقت داشته باشید هنوز درون service کدی وجود نداره .

**نکته cookie**

**اگر نیاز بود کوکی ست کنیم نیاز هست درون خوده controller تعریف کنیم و درون service نمیشه همینطور نیاز هست برای ست کردن کوکی متدش get باشه . نکته بعدی در همون ادرس api کوکی ست میشه و قابل مشاهده هست مثلا اگر روت api ما /user باشه درون localhost:2023/user کوکی ست شده ما درون وب قابل مشاهده هست .**

**در این فایل از کوکی استفاده نکردیم ولی این نکته مهمی بود که نیاز بود برای یاداوری گفته شه .**

خب در ادامه میایم اولیل method خودمون رو ایجاد می کنیم :

@Controller('user')

export class UserController {

  constructor(private readonly userService: UserService) {}

  @Post('/login')

  singin(@Body() body: LoginUserDto, @Request() req, @Response() res) {

    return this.userService.login(body, req, res);

    // return "user is here"

  }

{

همونطور که می بینید یک decorator داریم برای post که یک ادرس گرفته به اسم /login در اصل

الان api ما برای لاگین شدن کاربر به این صورت میشه :

localhost:2023/user/login

درون هر decorator ما می تونیم param یا ادرس دلخواه خودمون رو داشته باشیم مثلا متد پست که

در بالا برای لاگین شدن کاربر استفاده میشه .

  @Post('/login')

در ادامه یک فانکشن داریم که وقتی از کلاس ها استفاده می کنیم به اسم متد شناخته میشن به اسم

singin . درون این فانکشن کد زیر رو داریم :

singin(@Body() body: LoginUserDto, @Request() req, @Response() res) {

    return this.userService.login(body, req, res);

    // return "user is here"

  }

که Body decorator به همون بادی که از سمت فرانت اند میاد اشاره داره .

loginUserDto در واقع نوع تایپ body رو مشخص کرده که در یک فایل به اسم user.dto.ts تعریف

شده و در اینجا import شده و داره استفاده میشه .

در ادامه @Requset رو داریم که در واقع همون req هست .

و @Response رو داریم که در واقع همون res هست .

نکته اون res , req که به رنگ نارنجی تعریف شدن نیاز هست که نوشته شن حتما همینطور body .

چون خوده decorator ها رو نمی تونیتم مستقیم به متدی که درون service تعریف شده پاس بدیم

خب درون این فانکشن singin یک متد دیگه صدا زده شده

که در اصل از user.service.ts داره گرفته میشه که درون constractor تعریف کردیم .

اول ایمپورت می کنیم :

import { UserService } from './user.service';

بعد درون constractor صدا می زنیم به صورت زیر

  constructor(private readonly userService: UserService) {}

یک اسم براش گذاشتیم به اسم userService که این اسم رو درون متد ها صدا می زنیم :

مثل زیر که درون فانکشن sigin صدا زدیم :

  singin(@Body() body: LoginUserDto, @Request() req, @Response() res) {

    return this.userService.login(body, req, res);

    // return "user is here"

  }

در کد بالا گفتیم this.userService.login که در اصل متد login درون user.service.ts تعریف شده .

وظیفه اش اجرای لاجیک و کد های هست که برای لاگین شدن نیاز هست .

منتها اینجا پارامتر های body , req ,res رو گرفته . که بهشون نیاز هست .

برای register هم فانکشن زیر تعریف شده در user.controller.ts :

  @Post('/register')

  singup(@Body() body: CreateUserDto) {

    return this.userService.singup(body);

  }

که یک decorator @POST استفاده شده به همراه ادرس /register که همونطور می بینیم فقط

body بهش پاس داده شده که در واقع از سمت فرانت این بادی میاد .

از form که برای ثبت نام در سمت فرانت اند نوشته میشه بادی خودش رو میگیره .

اردس api به این صورت میشه localhost:2023/user/register

در ادامه یک api داریم برای گرفتن profile کاربر که بدونیم چه کاربری الان انلاین هست :

که درون user.controller.ts تعریف شده .

  @Get('profile')

  async getUserProfile(@Req() req) {

    const token = req.headers.authorization.replace('Bearer ', '');

    const userProfile =

      await this.userService.verifyTokenAndGetUserProfile(token);

    return userProfile;

  }

که ادرس رو میشه profile در اصل میشه به اینصورت localhost:2023/user/profile که اینجا از متد Get استفاده شده برای اینکار .

و همینطور که می بینید یک token گرفته شده

و این توکن به یک متد دیگه که درون user.service هست

پاس داده شده به اسم verifyTokenAndGetUserProfile که این متد میاد به وسیله این توکن

مشخص می کنه چه کاربری انلاین شده در اصل توکنی رو که گرفته میاد وریفای می کنه .

که درون user.service.ts تمامی این اتفاقات می افته و درون user.controller.ts صرفا صدا زده شده .

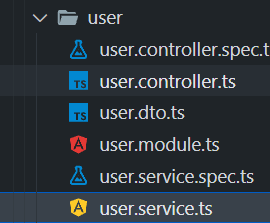
**نکته خیلی مهم :** این **tokenموقعی** که کاربر **login** میشه ایجاد میشه و از سمت **فرانت اند** .

گرفته میشه ذخیره میشه در نهایت در header این api مربوط به **profile**ست میشه

درون **header** ست میشه در نتیجه اینجا بهش دسترسی داریم

**خب حالا میریم سراغ user.service.ts**

**User.service.ts**

****

ایمپورت های که نیاز داریم درون user.service.ts

import { Prisma, UserRole } from '@prisma/client';

import { PrismaService } from './../prisma/prisma.service';

import {

  Injectable,

  NotAcceptableException,

  InternalServerErrorException,

} from '@nestjs/common';

import { AuthService } from 'src/auth/auth.service';

import \* as bcrypt from 'bcryptjs';

import \* as jwt from 'jsonwebtoken';

یک کلاس به اسم UserService داریم :

export class UserService {{

درونش یک constructor داریم :

این prismaService رو بالاتر ساختیم برای اینکه بتونیم به مدل ها مون دسترسی بگیریم .

و اینجا ازش داریم استفاده می کنیم و به مدل مربوط به user دسترسی میگیریم .

اول import کردیم :

import { PrismaService } from './../prisma/prisma.service';

بعد به صورت زیر یک اسم براش تعریف کردیم درون constractor در نهایت در کد برای ارتباط گرفتن به دیتابیس و مدل ها ازش استفاده می کنیم .

export class UserService {

  constructor(

    private readonly prismaService: PrismaService,

    private authService: AuthService,

  ) {}

}

که اینجا AuthService در واقع یدونه token رو میاد decode می کنه

که در اصل در فولدر auth ساخته شده حالا در ادامه بهش می پردازیم .



حالا بعد از consttractor میام متد ها رو تعریف می کنیم که درون user.controller.ts صداشون زدیم .

@Injectable()

export class UserService {

  constructor(

    private readonly prismaService: PrismaService,

    private authService: AuthService,

  ) {}

  async login({ email, password }, req, res) { {

{

که از این متد singup درون user.controller.ts داریم استفاده می کنیم برای لاگین شدن .

و email , passwordدر اصل از body که درون user.controller.ts

نکته که وجود داره برای اینکه می خوایم با متد های دیتابیس کار کنیم نیاز هست که login از نوع

Aysnc باشه که بتونیم در ادامه از await استفاده کنیم .

هست داره گرفته میشه .

یه بار دیگه به کدی که درون user.controller.ts مربوط به loginهست رو نگاه کنیم :

  @Post('/login')

  singin(@Body() body: LoginUserDto, @Request() req, @Response() res) {

    return this.userService.login(body, req, res);

    // return "user is here"

  }

همونطور که در کد بالا مشخص هست یک body داریم که پاس داده شده .

و به وسیله this.userService.login داریم از متد login که درون user.service.ts تعریف شده استفاده می کنیم .

خب بریم ببینم درون **login** چه کدی نوشته شده :

**login method in user.service.ts :**

 async login({ email, password }, req, res) {

    console.log(email, 'email is login');

    console.log(password, 'password is login');

    try {

      const user = await this.prismaService.user.findUnique({

        where: { email },

      });

      console.log('user is login', user);

      if (!user) {

        throw new NotAcceptableException('email or password is wrong');

      }

      const token = await jwt.sign(

        { email: user.email, userId: user.id, role: user.role },

        'secret',

        // { expiresIn: '15s' },

      );

      console.log(token);

      return res.status(200).json({

        message: 'user is login',

        data: { token: token, },

      });

    } catch (err) {

      return res

        .status(404)

        .json({ message: 'error user cant be login', data: err });

    }

  }

خب طبق معمول یک بلاک try / catch برای هندل کردن ارور ها داریم .

قابل توجه اگر این بلاک try / catch نباشه به اررور یا مشکلی بخوریم کل سرور از کار خواهد افتاد .

در اولین بخش پیدا کردن کاربری هست که مشخصات خودش رو ارسال کرده از فرانت اند .

در واقع می خواد لاگین کنه .

      const user = await this.prismaService.user.findUnique({

        where: { email },

      });

اینجا email که درون body ارسال شده رو گرفتیم درون متد finUnique قرار دادیم .

اگر دقت کنید یک prismaService هم داریم که این رو باید ساخته باشیم .

که این prismaService متصل میشه به مدل های که اول کار ایجاد کردیم .

در ادامه متد findUnique رو داریم که درونش یک ابجکت هست و از where استفاده شده .

Where هم خودش یک ابجکت هست و موردی که می خوایم پیدا کنیم رو بهش میدیم .

که در اینجا email رو می خواستیم پیدا کنیم .

خب اگر email باشه که میریم ادامه روند کد . اگر نباشه یک error ایجاد می کنیم مبنی بر اینکه اون

کاربر وجود نداره و این پیام رو به سمت فرانت ارسال می کنیم .

     if (!user) {

        throw new NotAcceptableException('email or password is wrong');}

**نکته** : Exception های زیادی درون nestjs داریم که با توجه به نیازمون می تونیم ازشون

استفاده کنیم . مثل auth بودن یا انتظار نداشتن چیزی مثل کد بالا یا 404 .

اگر کاربر پیدا شد بعد از این if یک توکن ایجاد می کنیم :

      const token = await jwt.sign(

        { email: user.email, userId: user.id, role: user.role },

        'secret',

        // { expiresIn: '15s' },

      );

مقدار های که می خوایم درون توکن داشته باشیم رو قرار میدیم .

در بالا email کاربر و id کاربر همینطور role کاربر رو درون توکن خودمون گذاشتیم .

یدونه secret key می خواد که حتما در پروژه ها واقعی از env استفاده شه .

می تونیم براش زمان expiresIn تعریف کنیم که بعد از چه مدتی توکن از بین بره .

زمانی که توکن رو تعریف کردیم به سمت فرانت اند می فرستیم .

      const token = await jwt.sign(

        { email: user.email, userId: user.id, role: user.role },

        'secret',

        // { expiresIn: '15s' },

      );

      return res.status(200).json({

        message: 'user is login',

        data: { token: token },

      });

این توکن رو نیاز داریم برای اینکه بدونیم چه کاربری لاگین شده و مشخصات شو بگیریم .

که این کار به وسیله یک متد دیگه به اسم verifyTokenAndGetUserProfile انجام میشه .

که میاد همین token رو میگیره .

این token از سمت فرانت اند گرفته میشه در نهایت درون headers مربوط به api که

قرار هست profile کاربری که لاگین شده رو نشون بده ست میشه

    const token = req.headers.authorization.replace('Bearer ', '');

که این کد مربوط درون user.controller.ts هست . و این headers.authrozation در اصل

داره token که موقع login به فرانت اند ارسال شده رو میگیره .

  @Get('profile')

  async getUserProfile(@Req() req) {

    const token = req.headers.authorization.replace('Bearer ', '');

    const userProfile =

      await this.userService.verifyTokenAndGetUserProfile(token);

    return userProfile;}

و در نهایت res رو داریم که چیزی که می خوایم رو ارسال می کنیم سمت فرانت .

که نشون دهنده این موضوع هست که کاربر لاگین شده .

      return res.status(200).json({

        message: 'user is login',

        data: { token: token, user: userProfile },

      });

در بلاک catch هم که امدیم ارور رو هندل کردیم :

 catch (err) {

      return res

        .status(404)

        .json({ message: 'error user cant be login', data: err });

    }

**به طور خلاصه :** یک token در متد login درون user.service.ts ساخته میشه .

این توکن حاوی مشخصات کاربری هست که لاگین شده .

توکن ساخته شده موقع لاگین به شدن به سمت فرانت اند فرستاده میشه .

حالا برای اینکه بدونیم چه کاربری لاگین هست و به اصطلاح پرفایل کاربر رو داشته باشیم .

یک متد دیگه داریم که این متد میاد به token رو verify می کنه و به ما میگه چه کاربری لاگین شده .

اما نیاز داره در سمت فرانت اند موقعی که api مربوط به profile کال میشه به token که موقع لاگین

شدن فرستاده شده دسترسی داشته باشه و درون headers که داره ست شه .

**خب بریم سراغ لاجیکی که برای register درون user.service.ts نوشتیم :**

**register method in user.service.ts :**

یک متد داریم به اسم **singup** که وظیفه ثبت نام کردن رو به عهده داره .

async singup({ email, password, confirmPassword, role }: userType) {}

خب همونطور که میبینید email , password , confrimPassword, role رو داریم که از بادی گرفته میشه .

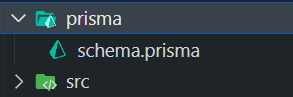
**نکته :** مقدار role به صورت default در مدل تعریف شده و اینجا نیاز هست که همون enum که میخ وایم رو براش تعریف کنیم یعنی اینجوری میشهUserRole.USER

که این UserRole.User در اصل import شده از همون جای که مدل رو ساختیم به این صورت .

import { Prisma, UserRole } from '@prisma/client';

این prisma/clinet در اصل اشاره داره به همون فایل prisma که درون root اصلی پروژه ما ساختیم

بیرون از فایل src .



مقدار بعدی که userType هست در اصل تایپ ورودی های که به فانکشن پاس دادیم

مشخص می کنه که یدونه type هست که تعریف کردیم درون خوده user.service.ts به صورت زیر

type userType = {

  email: string;

  password: string;

  confirmPassword: string;

  role: UserRole;

};

برای هر موردی که به متد هامون پاس میدیم نیاز هست برای وردی ها مون تایپ تعریف کنیم .

چون خوده nestjs به صورت دیفالت از typescript استفاده می کنه .

در ادامه درون متد **singup** کد زیر رو داریم :

اولین مورد hash کردن password هست .

که پسورد که کاربر وارد می کنه به صورت رمزگذاری درون دیتابیس ذخیره شه .

async singup({ email, password, confirmPassword, role }: userType) {

    const salt = await bcrypt.genSalt(10);

    const hashPassword = await bcrypt.hash(password, salt);

    const hashConfirmPassword = await bcrypt.hash(confirmPassword, salt);

}

از bcryptjs استفاده کردیم برای اینکار که قبلا اون نصب و import کردیم .

import \* as bcrypt from 'bcryptjs';

بعد از hash کردن پسورد یک if داریم که داره چک می کنه password و confrimPassword که کاربر در سمت فرانت اند وارد می کنه یکی باشه در غیر این صورت اررور بده :

async singup({ email, password, confirmPassword, role }: userType) {

    const salt = await bcrypt.genSalt(10);

    const hashPassword = await bcrypt.hash(password, salt);

    const hashConfirmPassword = await bcrypt.hash(confirmPassword, salt);

    if (password !== confirmPassword) {

      throw new NotAcceptableException('both password must be same');

    }

}

در ادامه ساخت کاربر جدید رو داریم که از متد create دیتابیس استفاده می کنیم :

    const user = await this.prismaService.user.create({

      data: {

        email,

        password: hashPassword,

        confirmPassword: hashConfirmPassword,

        role: UserRole.USER,

      } as Prisma.UserUncheckedCreateInput,

    });

نیاز به await داریم .

درون create یک ابجکت داریم {} درون ابجکت یک data داریم که اونم ابجکت هست .

حالا درون این دیتا میایم مقادیر جدید که می خوایم درون دیتابیس ایجاد شه رو می فرستیم .

نکته : همونطور که قبلا اشاره شد مقدار role به صورت default روی USER تنظیم شده .

و اینجا هم از UserRole.User استفاده کردیم در غیر اینصورت به ارور مواجه میشیم .

در اخر از تایپ Prisma.UserUncheckedCreateInput استفاده کردیم که نوع تایپ رو مشخص کرده

در نهایت یک return داریم که همین مقدار دیتای جدیدی که درون متغییر user ذخیره کردیم رو به سمت فرانت به همراه یک message ارسال می کنه :

    return { message: 'user is created', data: user };

کل فانکشن register به صورت زیر هستش :

  async singup({ email, password, confirmPassword, role }: userType) {

    const salt = await bcrypt.genSalt(10);

    const hashPassword = await bcrypt.hash(password, salt);

    const hashConfirmPassword = await bcrypt.hash(confirmPassword, salt);

    // or password !== confirmPassword

    if (password !== confirmPassword) {

      throw new NotAcceptableException('both password must be same');

    }

    const user = await this.prismaService.user.create({

      data: {

        email,

        password: hashPassword,

        confirmPassword: hashConfirmPassword,

        role: UserRole.USER,

      } as Prisma.UserUncheckedCreateInput,

    });

    // console.log({ user });

    return { message: 'user is created', data: user };

  }

خب بریم سراغ اینکه پرفایل کاربری که لاگین کرده رو داشته باشیم .

و در سمت فرانت بتونیم مشخصات کاربری که لاگین شده رو نشون بدیم .

**User profile in user.service.ts**

زمانی که کاربر لاگین میشه همزمان یک token ایجاد می کنیم .

که درون این توکن مشخصات کاربری که نیاز داریم داشته باشیم .

درون این توکن sing یا همون امضا می کنیم .

برای اینکه بهتر متوجه شیم کد متد login رو یک بار دیگه با هم نگاه کنیم :

  async login({ email, password }, req, res) {

    console.log(email, 'email is login');

    console.log(password, 'password is login');

    try {

      const user = await this.prismaService.user.findUnique({

        where: { email },

      });

      console.log('user is login', user);

      if (!user) {

        throw new NotAcceptableException('email or password is wrong');

      }

      const token = await jwt.sign(

        { email: user.email, userId: user.id, role: user.role },

        'secret',

        // { expiresIn: '15s' },

      );

      console.log(token);

      return res.status(200).json({

        message: 'user is login',

        data: { token: token, },

      });

    } catch (err) {

      return res

        .status(404)

        .json({ message: 'error user cant be login', data: err });

اول اینکه متدی که درون user.controller.ts داره استفاده میشه :

  @Get('profile')

به صورت get هست .

بریم سراغ **verifyTokenAndGetUserProfile** ببینیم که چه اتفاقی داره براش می افته :

  async verifyTokenAndGetUserProfile(token: string) {  }

اول که یک فانکشن هست و یک parameter به اسم token داریم که در اصل همون token هست که درون متد login بهش پاس دادیم .

در ادامه یک بلاک try / catch داریم .

  async verifyTokenAndGetUserProfile(token: string) {

    try {} catch (error) {

    }}

که کار بلاک try / catch هندل کردن ارور هست مثل همیشه .

درون بلاک try کد های زیر رو داریم :

      const decodeToken: any = await this.authService.verifyToken(token);

      const { email, role, password } = decodeToken;

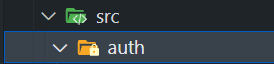
این قسمت از کد داریم در اصل token رو که از قسمت header گرفتیم داریم decode می کنیم .

که از سمت فرانت پاس داده میشه . این توکن رو موقع login کردن سمت فرانت اند می تونیم بگیریم

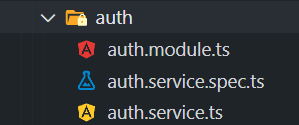
که مقدار های که در login امضا کردیم رو اینجا بکشیم بیرون .

کار این decode کردن رو authService برای ما انجام میده .

یک فولدر auth داریم درون src :



درونش یک auth.service.ts داریم و یک auth.module.ts .



که وظیفه اش در این پروژه decode کردن یا همون verify کردن token هستش .

درون auth.service.ts کد زیر رو داریم

که صرفا token رو به وسیله jsonwebtoken داریم وریفای می کنیم

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

import \* as jwt from 'jsonwebtoken';

@Injectable()

export class AuthService {

  async verifyToken(token: string) {

    console.log('token in verifytoken auth service', token);

    try {

      const decoded = await jwt.verify(token, 'secret');

      return decoded;

    } catch (err) {

      throw new UnauthorizedException('invalid token');

    }

  }

}

در نهایت decoded رو داریم return می کنیم که در اصل همون verify شده توکن هست .

درون auth.module.ts هم امدیم auth.service رو exports کردیم :

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

import { AuthService } from './auth.service';

@Module({

  providers: [AuthService],

  exports: [AuthService],

})

export class AuthModule {}

به این دلیل که می خوایم از AuthService درون user.service.ts در متد مربوط به profile ازش استفاده کنیم .

اما قبل اش نیاز هست که خوده module مربوط به auth رو درون user.module.ts بیایم import کنیم به صورت زیر :

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

import { UserController } from './user.controller';

import { UserService } from './user.service';

import { PrismaModule } from 'src/prisma/prisma.module';

import { AuthModule } from 'src/auth/auth.module';

@Module({

  imports: [PrismaModule, AuthModule],

  controllers: [UserController],

  providers: [UserService],

})

export class UserModule {}

همنطور که می بینید اول import شده AuthModule :

import { AuthModule } from 'src/auth/auth.module';

در ادامه امدیم imports کردیم :

  imports: [PrismaModule, AuthModule],

حالا از کلاس AuthService که درون auth.service.ts قرار داره می تونیم درون user.service.ts استفاده کنیم و مقدار توکنی که decode شده رو داشته باشیم .

اول import می کنیم :

import { AuthService } from 'src/auth/auth.service';

دوم درون constractor یک اسم بهش اختصاص میدیم :

  constructor(

    private readonly prismaService: PrismaService,

    private authService: AuthService,

  ) {}

در ادامه در متد مد نظرمون به صورت زیر استفاده می کنیم :

      const decodeToken: any = await this.authService.verifyToken(token);

که کد بالا در واقع داره token ما رو پاس میده به authService اونجا verify یا همون decode میشه

در اصل مقدار decode شده توکن رو به ما بازگشت میده .

  async verifyTokenAndGetUserProfile(token: string) {

    console.log('this token in verifyTokenAndGetUserProfile', token);

    try {

      // این وریفای توکن یک فاکنشت هست که درون اث سرویس دار اجرا میشه

      const decodeToken: any = await this.authService.verifyToken(token);

      const { email, role, password } = decodeToken;  } {

حالا که مقادیر decode شده رو داریم کاربری که لاگین شده و مشخصاتش درون توکن بود و وریفای کردیم و الان به Email و بقیه موارد دسترسی داریم .

به وسیله email میایم کاربرمون رو پیدا می کنیم :

      const userProfile = await this.prismaService.user.findUnique({

        where: { email },

      });

در نهایت return می کنیم :

      return { email, role, userProfile, password };

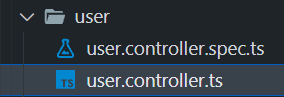
هر چیزی رو که از کاربر می خوایم رو return می کنیم که اینجا برای تست تمامی مقادیر ارسال شده .

اما در اصل مقدار userProfile کفایت می کنه

یا حتی می تونیم دیگه findUnique استفاده نکنیم و مستقیم مقادیر که از توکن گرفتیم ارسال کنیم .

البته این مورد **تست نشده**

یک بار برگردیم به user.controller.ts و برای یاد اوری ببینیم که چه route و چه متدی گذاشته شده :



درون این متد داریم get استفاده می کنیم با ادرس profile و البته توکن رو از header می گیریم .

و باز به متد verifyTokenAndGetUserProfile پاس دادیم .

این توکن در سمت فرانت اند ست میشه موقعی که این api کال میشه .

نیاز هست توکنی که موقع login در سمت فرانت اند داده میشه گرفته شه

ذخیره شه و به این Api درون header پاس داده شه

  @Get('profile')

  async getUserProfile(@Req() req) {

    const token = req.headers.authorization.replace('Bearer ', '');

    const userProfile =

      await this.userService.verifyTokenAndGetUserProfile(token);

    return userProfile;}

خب در این پروژه logout رو سمت کلاینت هندل کردیم .

به این صورت که اون توکن که موقع لاگین کردن داریم درون localstroge ذخیره می کنیم .

و از localstroge می گیریم به api پرفایل در headers ست می کنیم .

حالا زمانی که فانکشن logout در فرانت اند اجرا میشه میاد localstroge که حاوی token هست

رو پاک می کنه اینجوری دیگه token وجود نداره که بخواد پرفایل کاربر رو نشون بده .

در نتیجه کاربر logout میشه همین کار رو می تونیم برای دسترسی login بودن کاربر به خیلی جاها در

سمت کلاینت استفاده شه .

**برای دیدن جزئیات فرانت اند این پروژه به داکیومنت مربوط به فرانت اند پروژه مراجعه کنید**

**پایان داکیومنت بک اند این پروژه .**