پروژه todo برای تمرین کردم مفاهیم nestjs و nextjs زده شده .

از nestjs برای سمت بک اند پروژه استفاده شده .

و از nextjs برای سمت فرانت پروژه استفاده شده .

دو فولدر داریم که به اسم front و back که درون فولدر back فایل های بک اندی nestjs قرار داره و در فولدر front فایل های فرانت اندی قرار دارند .

**بخش فرانت اند (nextjs) :**

نصب و راه اندازی nextjs با دستور npx create-next-app@latest

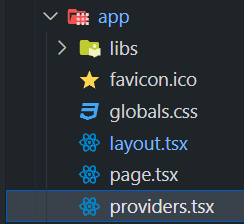
نصب ماژول های مورد نیاز : react-icons , react-query – axios

راه اندازی react-query در nextjs

بعد از نصب react-query با دستور npm i react-query

نوبت به ساخت provider برای react-query هست .

یک فایل در app ایجاد می کنیم به اسم provider.tsx که در این فایل میایم react-query خودمون رو تعریف می کنیم .



تعریف کردن این provider به این دلیل هست که react-query سمت کلاینت قابل اجرا هست و می خوایم در layout اصلی nextjs ازش استفاده کنیم .

برای همین نمی خوایم در layout از use client استفاده کنیم بلکه در provider از use client استفاده می کنیم به همین دلیل provider رو ایجاد کردیم ازش درون layout استفاده می کنیم .

اینجوری layout ما سمت سرور باقی می مونه فایل provider.tsx به صورت زیر هست

"use client";

import { QueryClient, QueryClientProvider } from "react-query";

import { useState } from "react";

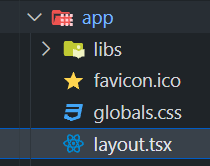
export const Providers = ({ children }: { children: React.ReactNode }) => {

  const [client] = useState(new QueryClient());

  return <QueryClientProvider client={client}>{children}</QueryClientProvider>;

};

درون layout.tsx هم میایم import می کنیم و ازش استفاده می کنیم children رو درون provider ایجاد شده قرار میدیم .



درون layout.tsx کد زیر رو خواهیم داشت البته مقدار های دیفالت خوده layout هم داریم مثل metdata و ... به دلیل اینکه حجم کد کم شه کل کد رو نذاشتیم .

import { Providers } from "./providers";

export default function RootLayout({

  children,

}: {

  children: React.ReactNode;

}) {

  return (

    <html lang="en">

      <body className={inter.className}>

        <Providers>{children}</Providers>

      </body>

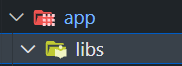
    </html>

  );

}

**فولدر libs :**

درون این فولدر قرار هست که بیایم api مربوط رو صدا بزنیم که در این پروژه 2 api داریم یکی برای todo ها و یکی برای user ها .



**بخش بک اند (nestjs) :**

برای ایجاد یک پروژه جدید nestjs

از دستور nest new project name استفاده می کنیم .

که اسم پروژه ما todo هستش , nest new todo

بعد از نصب پروژه nestjs ما با دستور npm start run:dev

روی لوکال هاست 3000 ران میشه و یک پیام hellow nestjs مشاهده می کنیم .

**نکته :** http://localhost:3000/ رو باید به صورتی دستی در مرورگر خودتون باز کنید بعد از ران کردن پروژه nestjs و لینکی درون ترمینال vscode به ما نمیده که روش کلیک کنیم .

مرحله بعد نصب dependencies های مورد نیاز هست .

در این پروژه dependencies زیر رو به پروژه اضافه کردیم :

"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",

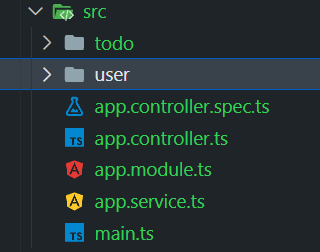
    "jsonwebtoken": "^9.0.2",

    "prisma": "^5.5.2",

  },

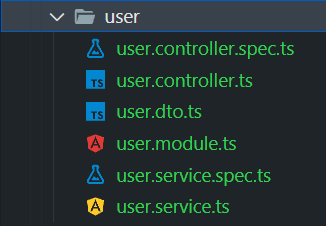
**نکته :** درون nestjs نیاز داریم که @types رو هم نصب کنیم . به این دلیل که nestjs به طور پیش فرض از typescript استفاده می کنه .

فایل اصلی پروژه nestjs فولدر src هست تمامی اتفاقات مهم در این فولدر رخ میده .



در nestjs برای نوشتن هر api نیاز به یک فولدر داریم همونطور که مشاهده میشه در عکس بالا دو فولدر وجود دارند به اسم user و todo

در هر فولدر 3 فایل مورد نیاز داریم . module , service , controller که تمامی با پسوند ts ساخته میشن و در کنارشون یک فایل test هم ایجاد میشه .



این فایل های service , controller , module رو نیاز نیست به طور دستی بسازیم .

کافیه تریمنال vscode رو باز کنیم و از دستور زیر برای ساخت استفاده کنیم .

nest g module user

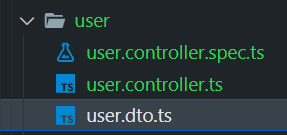
nest g controller user

nest g service user

برای todo هم همینکارو انجام میدیم تا این فایل ها ساخته شن .

برای هر فایل دیگه ای که نیاز داشتیم از همین دستورات استفاده می کنیم .

**نکته :** برای هر فایل یک فایل dto نیاز داریم برای route ها مون که درون این فایل dto میایم نوع تایپ دیتای که ردو بدل میشه رو می نویسیم .



این فایل رو به صورت دستی ایجاد کردیم

به عنوان مثال درون user.dto.ts رو نگاه کنیم :

export class CreateUserDto {

  email: string;

  password: string;

}

در واقع یک کلاس هست که export شده در ادامه تایپ های email و password رو مشخص کردیم که هر دو string هستند .

در همین فایل از class-validator برای ولیدیشن کردن فیلد ها استفاده می کنیم .

و همین طور از class-transformer برای حذف یا تغییر اسم فیلد ها استفاده می کنیم در صورت نیاز .

خوب در ادامه user.controller.ts رو درایم که کلا در controller ها میایم متد ها رو تعریف می کنیم get , post , put , delete و ادرس route ها رو .

controller.user.ts به صورت زیر هستش :

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

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

import { CreateUserDto } from './user.dto';

@Controller('user')

export class UserController {

  constructor(private readonly userService: UserService) {}

  @Get()

  login() {

    return 'user is login';

  }

  @Post()

  singup(@Body() body: CreateUserDto) {

    return this.userService.singup();

  }

}

که همون طور می بینید از دو متد Get و Post استفاده کردیم .

که به اینها در nestjs اصطلاحا decorator می گن که با علامت @ مشخص و تعریف میشن .

در هر controller.ts یک decorator به اسم @Controller داریم که میاد به طور کلی ادرس api ما رو تعریف می کنه .

الان ‘user’ قرار گرفته به این معنی که ادرس api مربوط به user ها میشه localhost:3000/user . و متد های ما در این ادرس کار می کنند .

هر controller برای این که به service مورد نظر خودش اتصال پیدا کنه

نیاز به یک constractor داره که در اینجا مشاهده می کنیم :

  constructor(private readonly userService: UserService) {}

در هر constractor یک اسم برای service اصلی تعریف میشه که اینجا اسم userService تعریف شده که در واقع همون UserService هست .

UserService از فایل user.service.ts ایمپورت شده .

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

در اصل درون controller ها میایم Api و ادرس روت ها و متد ها رو تعریف می کنیم .

و درون service میام logic که قرار هست اتفاق بیوفته مربوط به اون api رو قرار میدیم

به متد post دقت کنید :

  @Post()

  singup(@Body() body: CreateUserDto) {

    return this.userService.singup();

  }

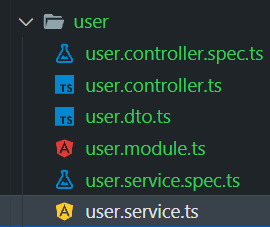
}

گفتیم هر چیزی که @ بگیره در اصل یک decorator هست . در اینجا @post و @Body در واقع decorator درونی خوده nestjs هستند که import شدن .

اگر دقت کنید یک **CreateUserDto** مشاهده می کنید ک این همون dto هست که کمی قبل ایجاد کردیم برای تعریف تایپ های email , password اینجا ازش استفاده کردیم در اصل نیاز هست اینکارو انجام بدیم درغیر اینصورت با ارور مواجه خواهیم شد .

این کار میاد مشخص می کنه نوع body که قرار هست ارسال شه چی هست و نوع تایپ شون قرار هست چی باشه .

اما در ادامه یک method به اسم singup داریم که درونش داره یک return صورت میگیره که در اصل یک method دیگه رو صدا میزنه که این متد اسمش singup هست که درون user.service.ts تعریف شده .



به فایل user.service.ts نگاه کنیم :

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

@Injectable()

export class UserService {

  async singup() {

    return 'user is create';

  }

}

در واقع درون user.controller.ts که یک متد @post هست و یک @body میاد این متدی که درون user.service.ts هست رو اجرا می کنه .

فایل user.controller.ts

  @Post()

  singup(@Body() body: CreateUserDto) {

    return this.userService.singup();

  }