

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа 1

Выполнил:

Никитин Павел

Группа

K33402

Проверил:

Добряков Д. И.

Санкт-Петербург

2024 г.

Задача

Нужно написать свой boilerplate на express + sequelize / TypeORM + typescript.

Должно быть явное разделение на:

- модели
- контроллеры
- роуты
- сервисы для работы с моделями (реализуем паттерн “репозиторий”)

Ход работы

В рамках рабочих задач я уже работал с Express, поэтому выбрал для первой лабораторной работы фреймворк Nest js, чтобы изучить что то новое.

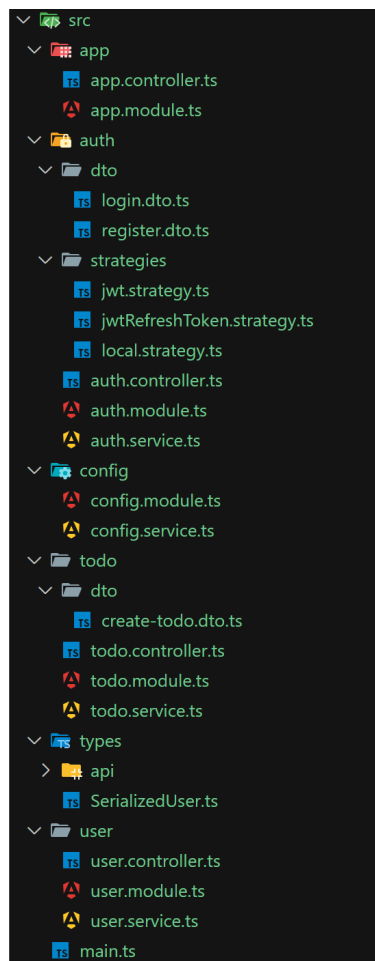


Рисунок 1 - структура проекта

Несмотря на то, что первоначально структура проекта пугает обилием папок и файлов, она задает четкие правила написания кода.

Весь проект разделен на модули - строительные блоки, которые инкапсулируют в себя логику определенной части приложения.

Для boilerplate примера я выбрал todo-list, поэтому в качестве модулей выступают:

app module - собирает воедино все остальные модули

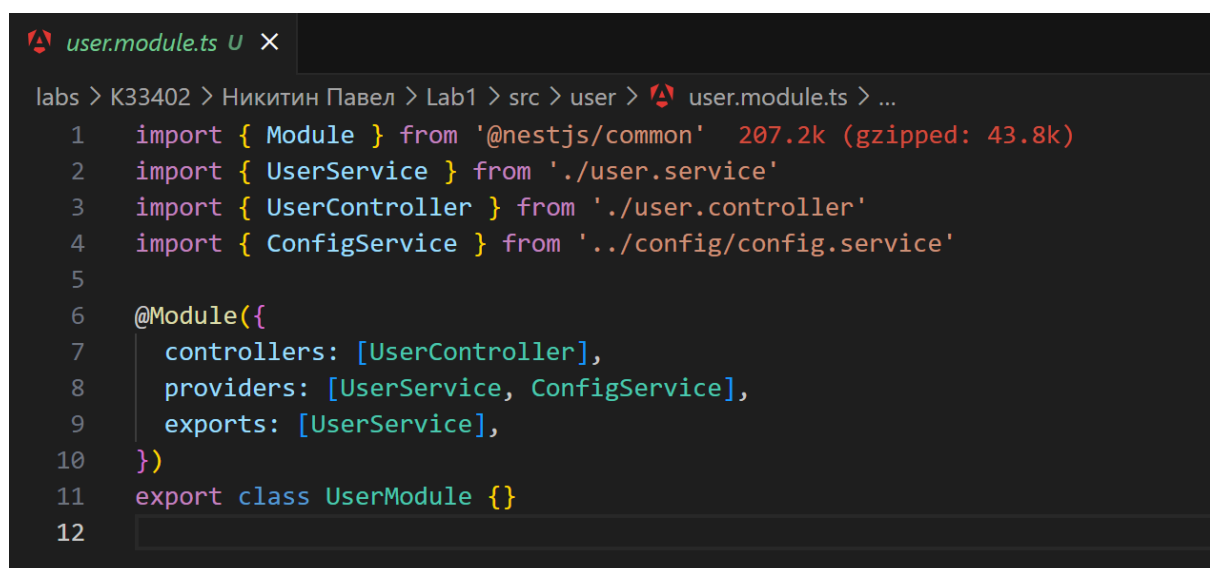
auth module - отвечает за авторизацию в приложении

config module - конфигурирует настройки приложения

todo module - является ответственным за все манипуляции со списком дел

user module - инкапсулирует логику взаимодействия с пользователями

В качестве примера давайте разберем UserModule:



```
user.module.ts U ×
labs > K33402 > Никитин Павел > Lab1 > src > user > user.module.ts > ...
1  import { Module } from '@nestjs/common'  207.2k (gzipped: 43.8k)
2  import { UserService } from './user.service'
3  import { UserController } from './user.controller'
4  import { ConfigService } from '../config/config.service'
5
6  @Module({
7    controllers: [UserController],
8    providers: [UserService, ConfigService],
9    exports: [UserService],
10 })
11 export class UserModule {}
12
```

Рисунок 2 - UserModule

Модуль инициализируется при помощи декоратора “@Module”. Nest.js реализует паттерн инъекцию зависимостей, мы нигде не создаем сами классы они автоматически создаются при помощи декораторов при инициализации проекта. Модуль же является связующим звеном он может содержать:

Контроллеры - классы реализующие API методы

Провайдеры - классы утилиты необходимые например для запросов к бд или логирования

Также модуль может экспортировать, какие нибудь провайдеры, чтобы они были доступны в других модулях.

```
import { Logger, Module } from '@nestjs/common' 207.2k (gzipped)
import { AppController } from './app.controller'
import { UserModule } from 'src/user/user.module'
import { AuthModule } from 'src/auth/auth.module'
import { ConfigModule } from 'src/config/config.module'
import { TodoModule } from 'src/todo/todo.module'
import { PrismaModule, loggingMiddleware } from 'nestjs-prisma'

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
    }),
    PrismaModule.forRoot({
      isGlobal: true,
      prismaServiceOptions: {
        middlewares: [
          loggingMiddleware({
            logger: new Logger('PrismaMiddleware'),
            logLevel: 'log',
          }),
        ],
      },
    ),
    UserModule,
    TodoModule,
    AuthModule,
  ],
  controllers: [AppController],
})
export class AppModule {}
```

Рисунок 3 - импорт второстепенных модулей в App Module

Следующий важный компонент Nest.js - контроллеры

В контроллере Todo реализованы все основные методы CRUD для работы со списком дел. При помощи декораторов мы можем задавать роуты, изменять тип запроса.

Также рассмотрим пример сервиса на примере UserService

```
@Injectable()
export class UserService {
  private readonly logger = new Logger('UserService')

  constructor(private prisma: PrismaService) {}

  serializeUser(user: User): SerializedUser {
    return {
      id: user.id,
      email: user.email,
    }
  }

  async user(
    userWhereUniqueInput: Prisma.UserWhereUniqueInput
  ): Promise<User | null> {
    return await this.prisma.user.findUnique({
      where: userWhereUniqueInput,
    })
  }

  async users(params: {
    skip?: number
    take?: number
    cursor?: Prisma.UserWhereUniqueInput
    where?: Prisma.UserWhereInput
    orderBy?: Prisma.UserOrderByWithRelationInput
  }): Promise<User[]> {
    const { skip, take, cursor, where, orderBy } = params
    return await this.prisma.user.findMany({
      skip,
      take,
      cursor,
      where,
      orderBy,
    })
  }

  async createUser(data: Prisma.UserCreateInput): Promise<User> {
    return await this.prisma.user.create({
      data,
    })
  }

  async updateUser(params: {
    where: Prisma.UserWhereUniqueInput
    data: Prisma.UserUpdateInput
  }): Promise<User> {
    const { where, data } = params
    return await this.prisma.user.update({
      data,
      where,
    })
  }

  async deleteUser(where: Prisma.UserWhereUniqueInput): Promise<User> {
    return await this.prisma.user.delete({
      where,
    })
  }

  async login(email: string, password: string): Promise<User> {
    const user = await this.user({ email })

    if (!user) {
      throw new Error('User not found')
    }

    const passwordMatch = await compare(password, user.password)

    if (!passwordMatch) {
      throw new Error('Invalid credentials')
    }

    return user
  }
}
```

Рисунок 5 - сервис для работы с базой данных User

Здесь реализован паттерн репозиторий - и в рамках одного класса реализована логика для всех возможных запросов к базе данных.

Работоспособность можно проверить при помощи swagger

auth ^	
POST	/api/auth/login
GET	/api/auth/logout
POST	/api/auth/register
GET	/api/auth/refresh
user ^	
GET	/api/user/me
todo ^	
GET	/api/todo/all
GET	/api/todo/{id}
DELETE	/api/todo/{id}
GET	/api/todo/{id}/toggleCompletedState
POST	/api/todo/create
POST	/api/todo/{id}/update

Вывод

В ходе работы освоил фреймворк Nest Js, научился работать с Orm библиотекой Prisma.