**Лабораторная работа № 2**

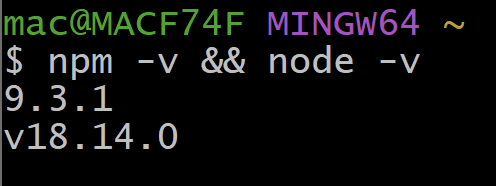
## Разработка серверного приложения

### Необходимые инструменты:

* Среда разработки Visual Studio Code
* Фреймворк NestJs 9.0
* Коммандная строка(если у вас windows, то рекомендуется установить Git Bash, ссылка: [Git for Windows](https://gitforwindows.org/))
* Node Js - [Download | Node.js (nodejs.org)](https://nodejs.org/en/download/)

### Начало работы:

Запускаем Git Bash и убеждаемся что пакет npm и node корректно установлены(если у вас ОС Linux или MacOS, то необходимости в установке git bash нет)

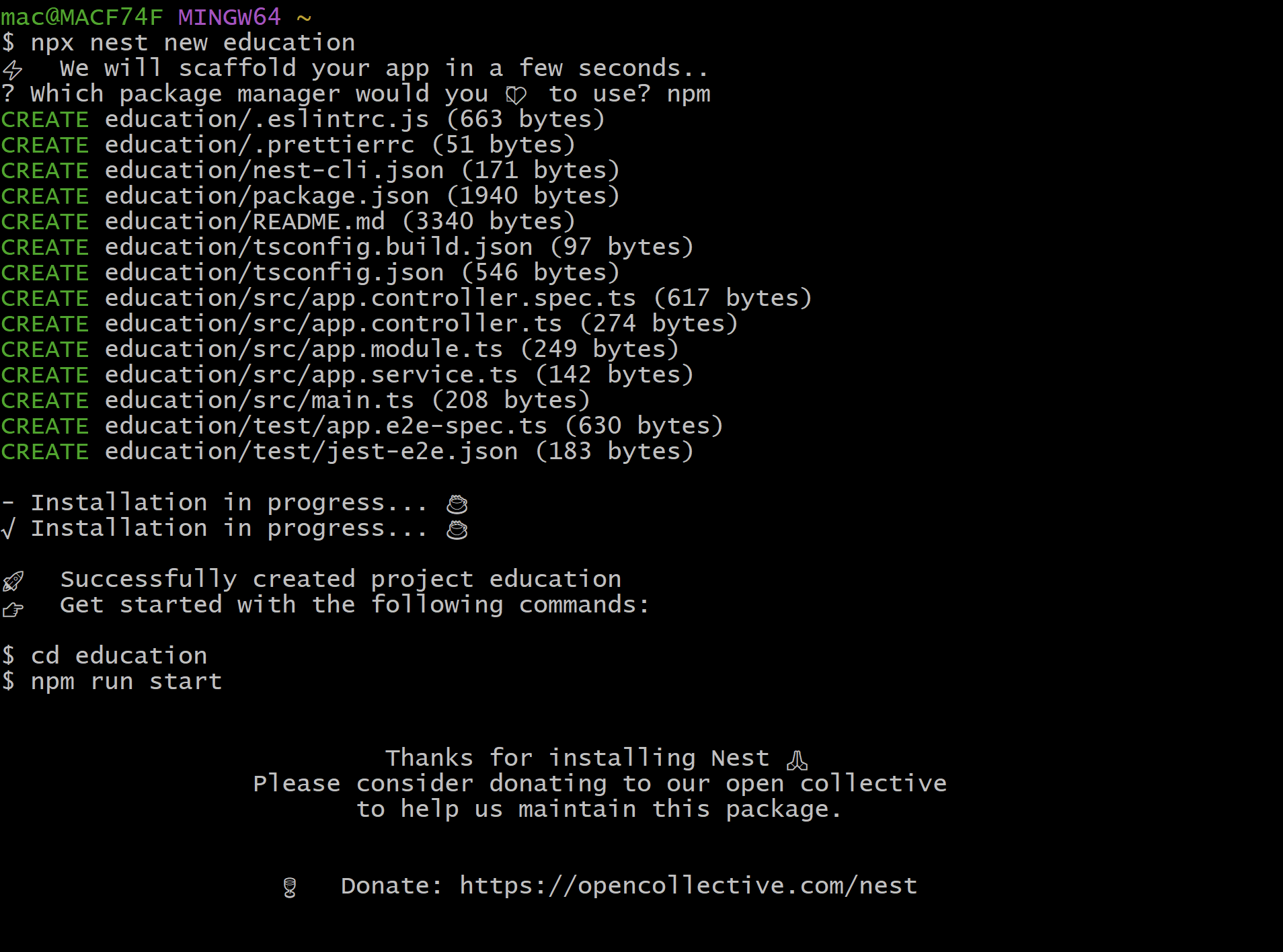


Изображение 1 Проверка корректной установки пакетов

После чего нам необходимо установить cli утилиту для возможности работы с фреймворком NestJs. Вводим в git bash

**npm -G install @nestjs/cli**

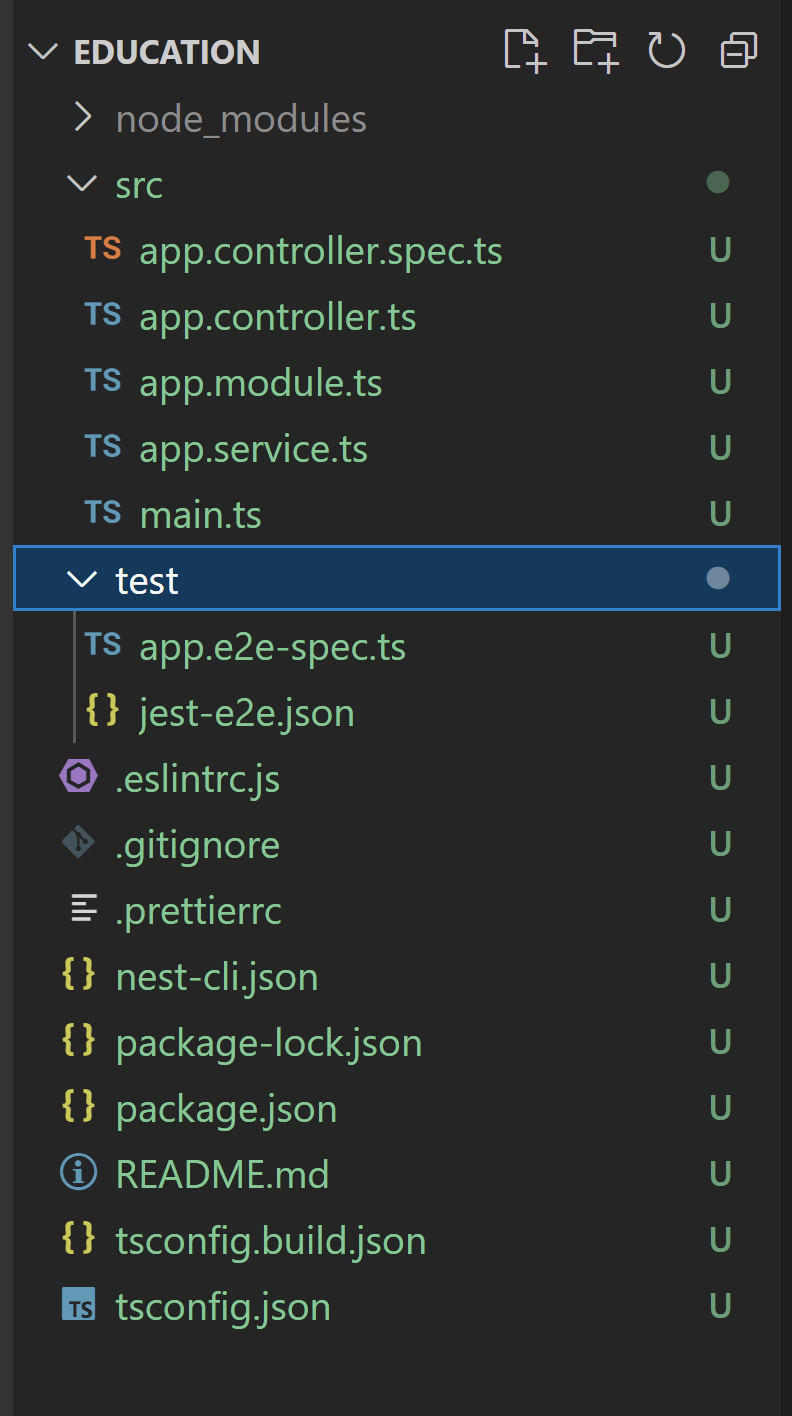
Далее необходимо инициировать новый проект NestJs, для этого вводим команду **npx nest new education**, выбираем пакетный менеджер **npm**, через несколько секунд базовый проект будет создан!



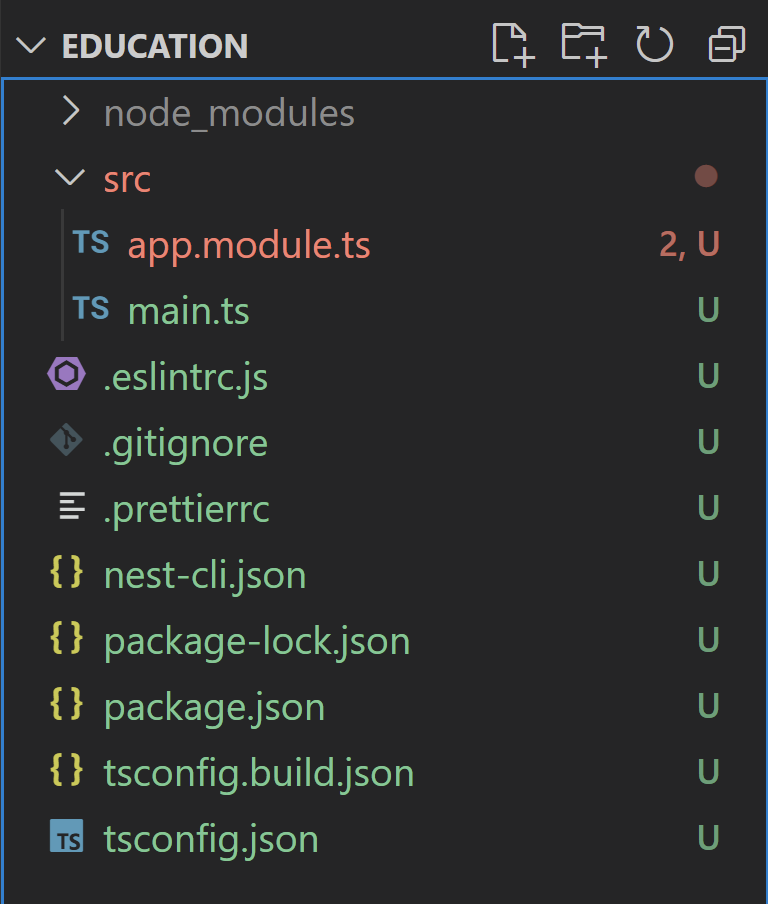
Изображение 2 Созданный проект NestJs

При помощи командной строки необходимо установить нужные пакеты для возможности работы с фреймворком.

Для macos и linux можно ввести команду **cd /education** (название вашего проекта) и далее **code .** . На windows придется запустить VS Code и из него открыть папку с созданным приложением в качестве проекта.

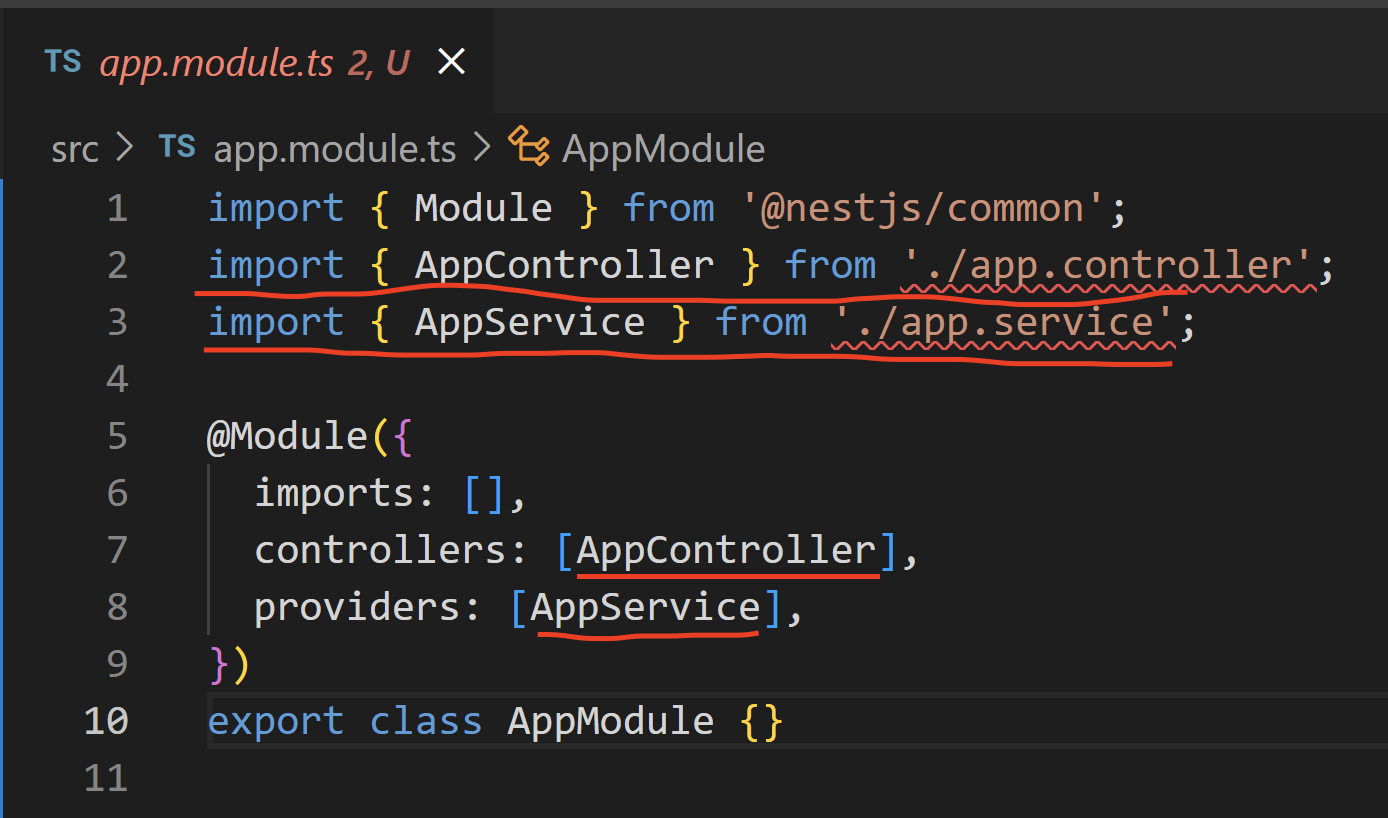


Изображение 3 Общая структура пустого приложения



Изображение 4 Структура проекта после удаления ненужных файлов

В файле app.module.ts необходимо удалить ссылки на удаленные файлы



Изображение 5 Удаление строк со ссылкой на удаленные файлы

Установим дополнительные пакеты командой **npm install @nestjs/typeorm @nestjs/swagger typeorm nestjs-form-data dotenv cross-env swagger-ui-express tsconfig-paths pg pg-hstore rimraf**

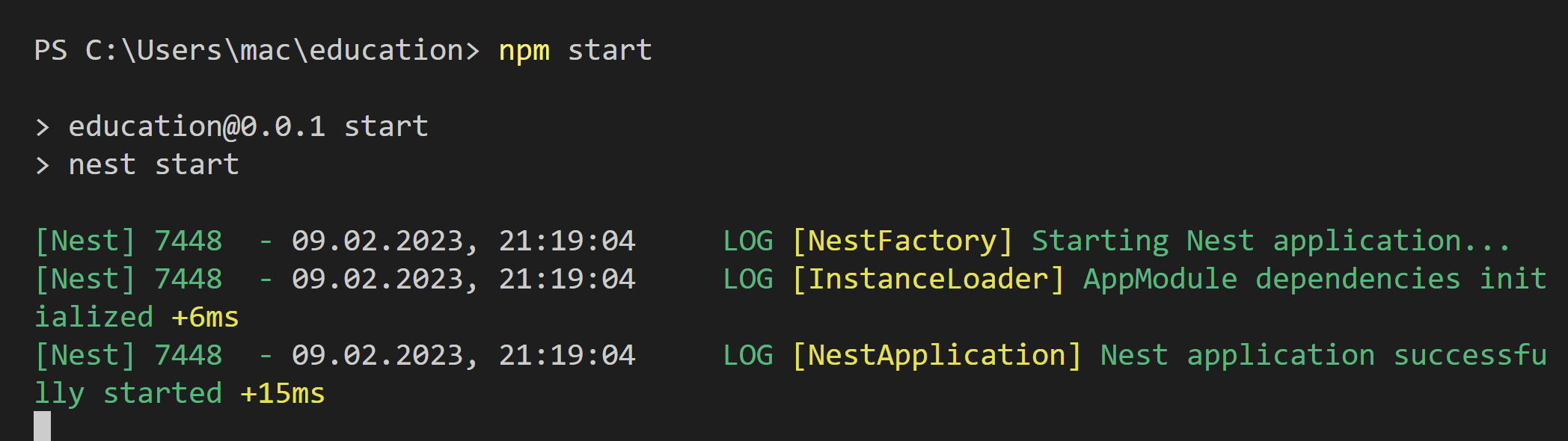
Описание дополнительных библиотек из репозитория npm:

1. TypeORM – необходима для работы с базами данных и создания схемы БД
2. Nestjs-form-data – формат передачи данных между клиентским и серверным приложением
3. Dotenv – переменные окружения
4. Crossenv – переменные окружения
5. Swgger – утилита для тестирования и автодокументирования api запросов
6. Pg – бибилиотека для работы с БД postgresql
7. Tsconfig-paths – конфигурация typescript

### Начало работы:

Итак, мы полностью настроили наше серверное приложение и оно готово к первому запуску!

Вводим команду npm start и радуемся успешному запуску приложения!



**Ремарка: Далее, на примере сущности Автор будут созданы модель, сервис и контроллер. Данная сущность является примером, в рамках выполнения лабораторной работы вам необходимо придумать и использовать сущности из ваших тем.**

### Создание модуля:

В папке src создадим папку authors, а в ней файл authors.module.ts . Этот файл будет отвечать за управление всеми компонентами в рамках сущности авторы, включая модель, контроллер и сервис, о которых мы поговорим далее. В конце необходимо включить в список модулей файла app.module.ts

Наполнение файла authors.module.ts

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

@Module({

controllers: [],

providers: [],

})

export class AuthorsModule {}

Измененный файл app.module.ts

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

import { AuthorsModule } from './authors/authors.module';

@Module({

imports: [AuthorsModule],

controllers: [],

providers: [],

})

export class AppModule {}

### Создание модели:

В папке authors создадим файл author.entity.ts. Это наша первая модель, теперь необходимо прописать для нее свойства.

export class Author {

id: number;

fullname: string;

position: string;

grade: string;

}

У каждого автора должен быть уникальный идентификатор(ID), фио(FullName), Должность(Position) и звание(Grade).

Данная модель предусматривает, что она обладает конечным набором свойств, описывающих экземпляр сущности.

И также импортируем в модуль авторов только что созданную модель, чтобы модуль идентифицировал наш класс Author как модель.

Измененный файл authors.module.ts

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

import { Author } from './entities/author.entity';

@Module({

controllers: [],

providers: [],

imports: [Author],

})

export class AuthorsModule {}

### Создание источника данных:

В рамках лабораторной работы №2 будем использовать псевдобазу данных на основе класса со списком авторов.

В других языках это можно было бы реализовать при помощи антипаттерна Singleton. Однако в nestjs мы создадим для этого отдельный модуль с сервисом, в котором и будем хранить наш список с авторами.

Для этого в папке src создадим папку datasource и в ней два файла: datasource.module.ts и datasource.service.ts

В файле модуля мы создадим класс, в котором будет храниться пустой массив авторов, который будем использовать в других сервисах. Важно то, что нам всегда будет возвращаться не новый экземпляр массива, а каждый раз один и тот же массив авторов.

В сервисе, помимо массива, необходимо реализовать метод доступа к классу авторов.

Наполнение файла datasource.service.ts

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

import { Author } from 'src/authors/entities/author.entity';

@Injectable()

export class DatasourceService {

private authors: Author[] = [];

getAuthors(): Author[] {

return *this*.authors;

}

}

Теперь нам необходимо сообщить модулю, что у него есть сервис, которым мы можем делиться с другими модулями для доступа к импровизированной базе данных.

Наполнение файла datasource.module.ts

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

import { DatasourceService } from './datasource.service';

@Module({

providers: [DatasourceService], *// тут мы указали что у нас есть сервис внутри модуля*

exports: [DatasourceService], *// здесь мы разрешаем экспортировать сервис в другие модули*

})

export class DatasourceModule {}

### Создание сервиса:

Сервис - это класс - обработчик высоконагруженных операций. В сервис обычно выносится асинхронная логика взаимодействия с базой данных и любыми операциями с данными.

Создадим сервис для сущности Автор. В папке authors создадим файл authros.service.ts

@Injectable()

export class AuthorsService {}

Это пустой класс, который пока не умеет ничего обрабатывать. Над ним указана аннотация, которая говорит нам о возможности применения такого подхода, как Dependency Injection. Но, подробнее об этом понятии мы поговорим чуть позже.

Теперь, нам необходимо создать конструктор класса сервиса и в нем объявить сервис псевдо-базы данных. Внутри класса объявим конструктор и пропишем параметром сервис псевдо-базы данных.

constructor(private readonly datasourceService: DatasourceService) {}

Создадим методы обработки сущности автор внутри сервиса. При обращении к методам будет возвращаться только тот результат, что декларирован методами обработки и преобразования списка авторов.

Создадим метод добавления автора:

create(author: Author) {

*this*.datasourceService.getAuthors().push(author);

return author;

}

Данный метод принимает в качестве аргумента экземпляр автора и добавляет его в коллекцию существующих авторов.

Метод для получения автора по ID:

findOne(id: number) {

return *this*.datasourceService

.getAuthors()

.find((author) => author.id === id);

}

Метод для получения всех авторов:

findAll(): Author[] {

return *this*.datasourceService.getAuthors();

}

Метод изменения автора:

update(id: number, updatedAuthor: Author) {

const index = *this*.datasourceService

.getAuthors()

.findIndex((author) => author.id === id);

*this*.datasourceService.getAuthors()[index] = updatedAuthor;

return *this*.datasourceService.getAuthors()[index];

}

Метод удаления автора:

remove(id: number) {

const index = *this*.datasourceService

.getAuthors()

.findIndex((author) => author.id === id);

*this*.datasourceService.getAuthors().splice(index, 1);

return HttpStatus.OK;

}

На данном этапе у нас написана базовая реализация основных операций с данными. Данные операции называются CRUD(Create, Read, Update, Delete).Исходя из этого, у нас есть готовый инструментарий декларированного доступа к коллекции авторов для внешних источников. Таких, как контроллер.

### Создание контроллера:

Контроллер - это точка обращения на сервер. В методах происходит обработка начального запроса в зависимости от типа.

Создадим контроллер для сущности Автор. В папке authors создадим файл authors.controller.ts

import { AuthorsService } from './authors.service';

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

@Controller('authors')

export class AuthorsController {

constructor(private readonly authorsService: AuthorsService) {}

}

На данный момент это пустой контроллер с указанным маршрутом для доступа. В качестве зависимости(параметра конструктора) выступает созданный ранее сервис авторов. Теперь создадим первый обработчик запроса.

Самый простой и распространенный тип запроса - это Get запрос. Благодаря Get запросу мы можем запросить данные от сервера по пути контроллера. В данном случае - это путь localhost:3001/api/author

Get запрос на получение авторов:

@Get()

findAll() {

return *this*.authorsService.findAll();

}

При Get запросе по пути localhost:3001/api/author сервер вернет всех доступных авторов.

Get запрос на получение автора по ID

@Get(':id')

findOne(@Param('id') id: string) {

return *this*.authorsService.findOne(+id);

}

При Get запросе по пути localhost:3001/api/author/{числовое значение - идентификатор} сервер вернет автора с указанным ID.

PUT запрос на изменение автора

@Put(':id')

update(@Param('id') id: string, @Body() updateAuthor: Author) {

return *this*.authorsService.update(+id, updateAuthor);

}

При PUT запросе по пути localhost:3001/api/author/{числовое значение - идентификатор} и с объектом типа Author в теле запроса сервер применит новые значения для автора, если найдёт его в источнике данных и вернет обновленный экземпляр сущности.

POST запрос на добавление автора

@Post()

create(@Body() createAuthor: Author) {

return *this*.authorsService.create(createAuthor);

}

При POST запросе по пути localhost:3001/api/author с объектом типа Author в теле запроса сервер добавит в источник данных новый экземпляр сущности автор.

DELETE запрос на удаление автора

@Delete(':id')

remove(@Param('id') id: string) {

return *this*.authorsService.remove(+id);

}

При DELETE запросе по пути localhost:3001/api/author/{числовое значение - идентификатор} сервер удалит автора с указанным в запросе ID, если такой будет в источнике данных.

### Конечная настройка и подготовка к запуску

Добавим компоненты модуля авторов непосредственно в сам модуль

Файл authors.module.ts:

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

import { AuthorsService } from './authors.service';

import { AuthorsController } from './authors.controller';

import { DatasourceModule } from 'src/datasource/datasource.module';

@Module({

controllers: [AuthorsController],

providers: [AuthorsService],

imports: [DatasourceModule],

})

export class AuthorsModule {}

Импортируем в модуле приложения созданные ранее модули псевдо-базы и авторов

Файл app.module.ts:

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

import { AuthorsModule } from './authors/authors.module';

import { DatasourceModule } from './datasource/datasource.module';

@Module({

imports: [AuthorsModule, DatasourceModule],

controllers: [],

providers: [],

})

export class AppModule {}

Изменим порт для запуска, зададим глобальный путь и включим систем автодокументации апи Swagger.

Файл main.ts

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

import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

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

async function bootstrap() {

const app = await NestFactory.create(AppModule);

const config = new DocumentBuilder()

.setTitle('Education API')

.setVersion('1.0')

.build(); *// Конфигурируем сборщик документации*

const document = SwaggerModule.createDocument(app, config); *// создаем апи документацию*

SwaggerModule.setup('api\_docs', app, document); *//включаем документацию Swagger по пути localhost:3001/api\_docs*

await app.listen(3001); *//устанавливаем порт прослушивания 3001*

await app.setGlobalPrefix('/api'); *//глобальный префикс для роутов контроллера*

}

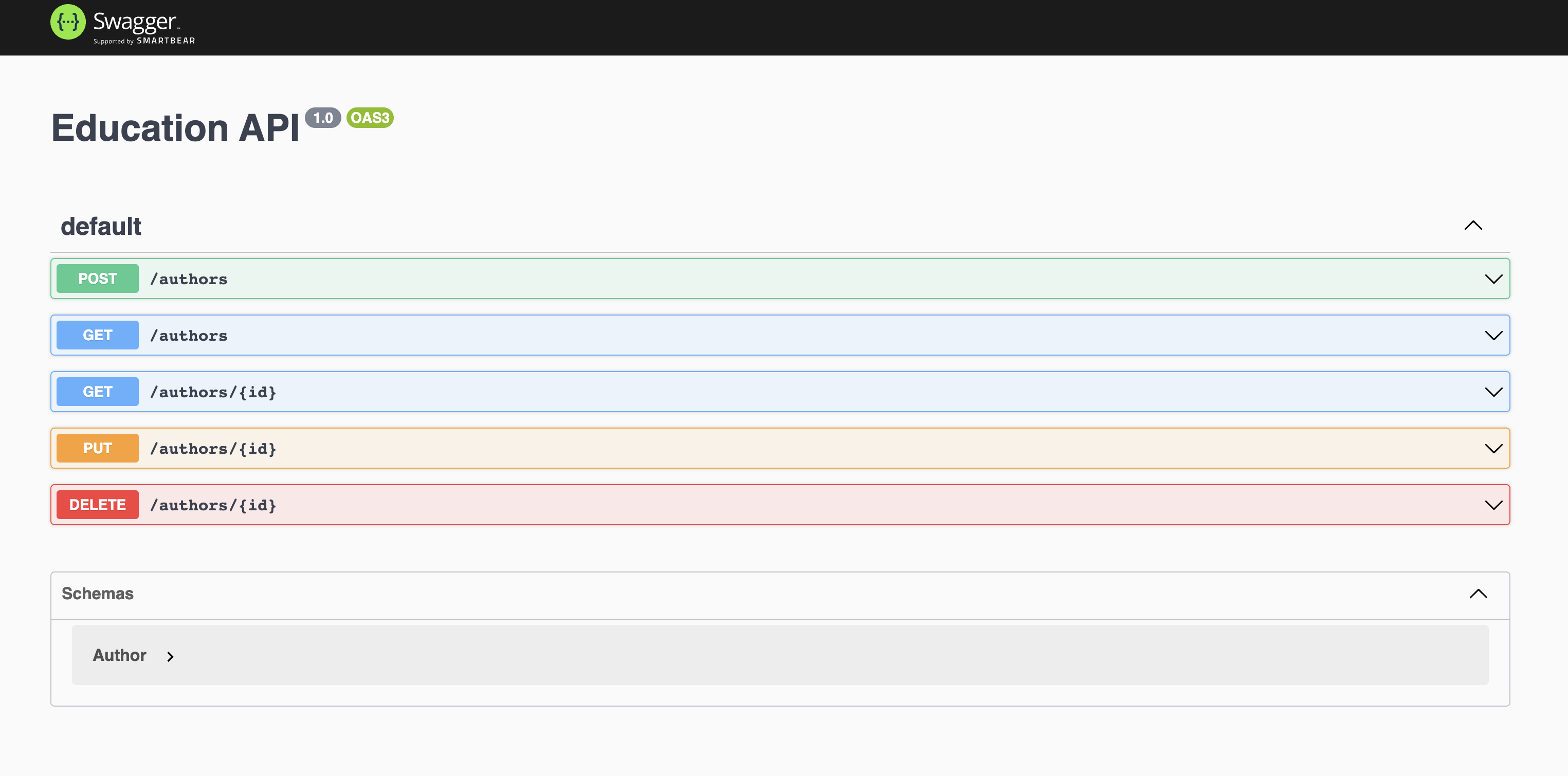
bootstrap();

Сервер готов к запуску!

Вводим в командной строке npm run start

### Тестирование:

Для тестирования необходимо перейти по адресу - localhost:7165/swagger/index.html. На данной странице находится система автодокументирования Swagger.



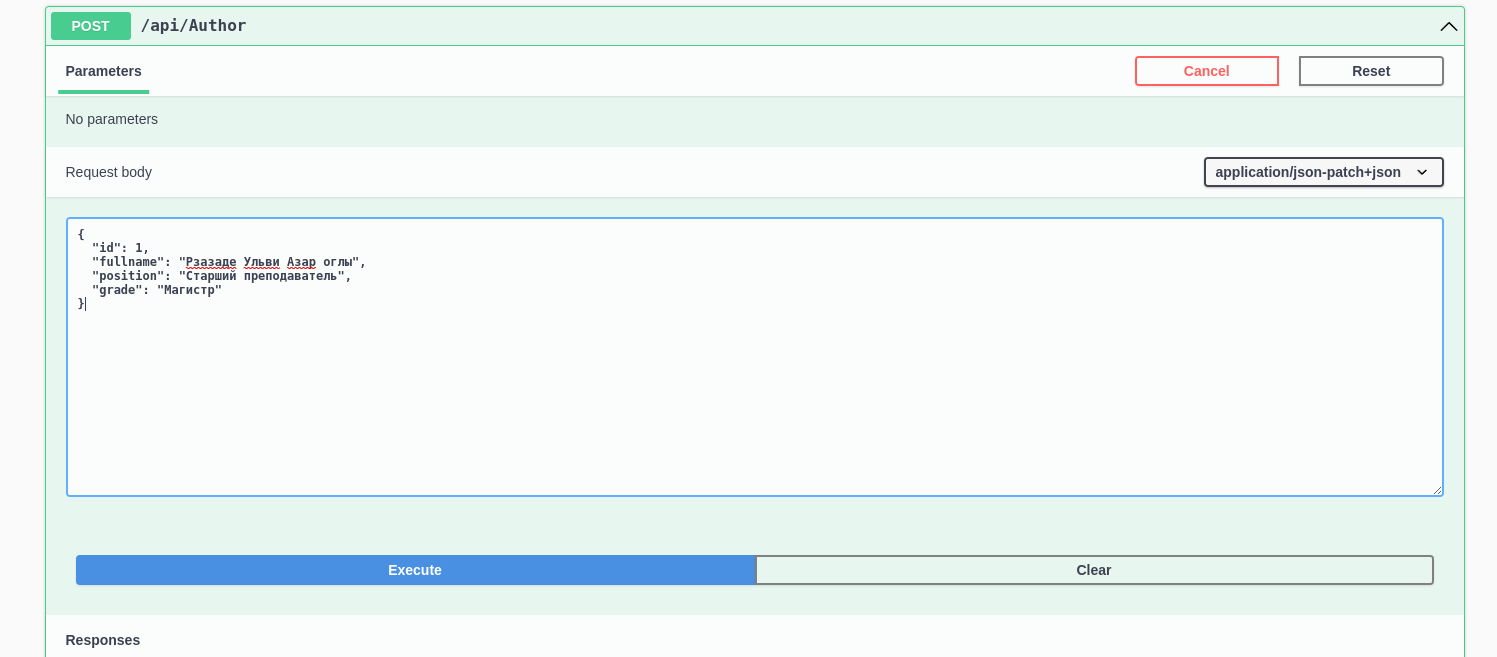
Изображение 6 Система автодокументирования Swagger

В соответствии с рисунком 3 изображено окно системы Swagger. При нажатии на любой из элементов контроллера Author откроется информации о запросе и появится возможность отправки соответствующего запроса на сервер.

Создадим автора:

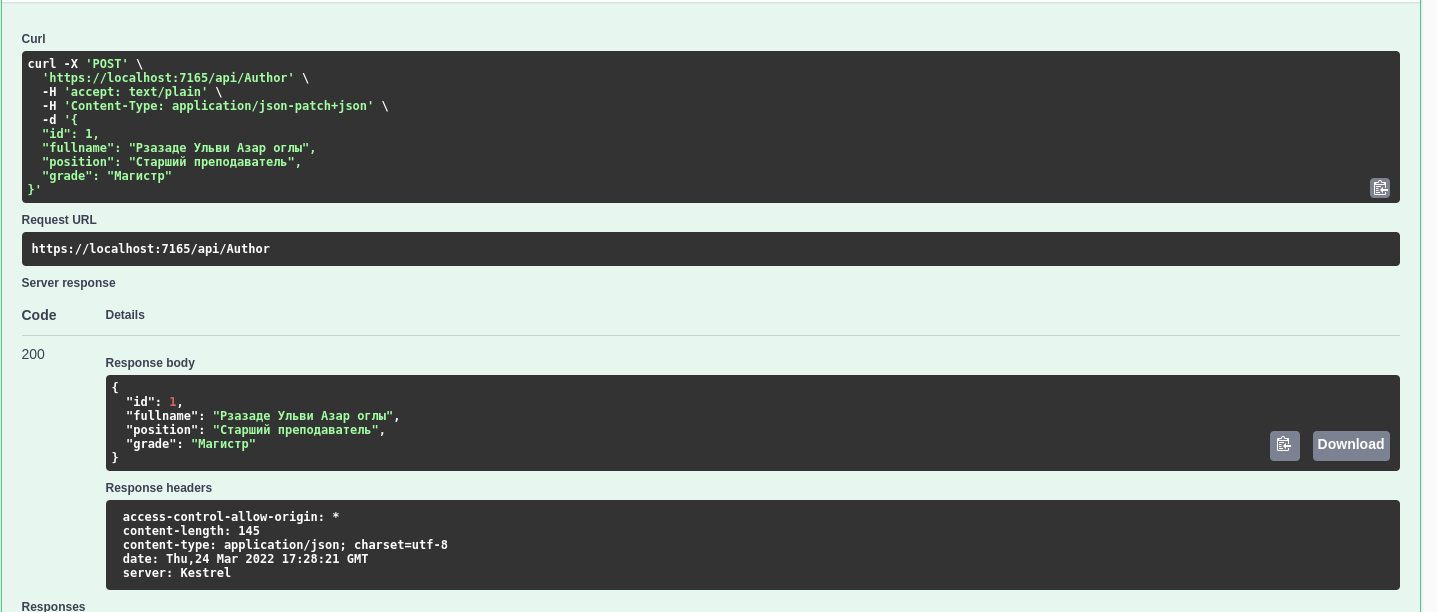
Для этого необходимо выбрать POST запрос и нажать Try it out!

После чего заполнить значения свойств автора в формате JSON и нажать Execute.



Изображение 7 Окно заполнения свойств автора

После чего Swagger вернёт ответ от сервера:



Изображение 8 Ответ сервера в Swagger

Видно, что сервер вернул код 200, то-есть запрос прошел успешно и был выполнен. В теле ответа(Responce body) сервер вернул объект типа Author, который ранее был создан.

### Задание:

1. На основе изученного материала реализовать сервер.
2. Реализовать не менее трёх моделей в рамках вашей темы
3. Для каждой модели реализовать в классе DataSource объект для хранения коллекции экземпляров сущностей
4. Для каждой модели реализовать сервис выполнения CRUD операций
5. Для каждой модели реализовать свой контроллер(который будет связан со своим сервисом) с методами GET, POST, UPDATE, DELETE.

### Рекомендации:

1. При возникновении ошибок внимательно изучайте причину, она обычно появляется в консоли и учитесь искать причину в google
2. 99,9% всех ошибок в коде по вине программиста