**Лабораторная работа № 5**

### Интеграция TypeOrm и подключение к базе данных

### Необходимые инструменты:

* PostgreSQL([установщик](https://www.postgresql.org/download/))
* [Полная документация TypeOrm](https://typeorm.io/)
* [Полная документация NestJs](https://docs.nestjs.com/)

### Начало работы:

Для удобства работы добавим еще несколько библиотек:

**npm install ts-node ts-loader @nestjs/config**

Далее необходимо обеспечить функционирование базы данных. В рамках обучения вами были получены компетенции для работы с базами данных, соответственно, не возбраняется использование наработок из пройденных курсов.

Процесс разворачивания базы данных можно организовать разными путями, в рамках данной лабораторной работы будет рассмотрен метод развертывания при помощи установщика PostgreSQL. По выше приведенной ссылке можно скачать и настроить базу данных, назвать ее **education**, пользователь **education**, пароль **password**.

Итак, база данных готова к использованию, теперь необходимо подключиться к ней из приложения, созданного в Лабораторной №2. Также, для примера работы с реляционными связями будут добавлены сущности

### Контекст базы данных:

В файле **app.module.ts** добавим контекст подключения к базе данных.

Добавим импорт

import { TypeOrmModule } from '@nestjs/typeorm';

Добавим в массив imports контекст подключения

imports: [

AuthorsModule,

DatasourceModule,

TypeOrmModule.forRoot({

type: 'postgres', *//тип подключаемой БД*

port: 5432, *//порт*

username: 'education', *//имя пользователя*

password: 'password', *//пароль*

host: 'localhost', *//хост, в нашем случае БД развернута локально*

synchronize: false, *//отключаем автосинхронизацию(в противном случае при каждом перезапуске наша БД будет создаваться заново)*

logging: 'all', *//включим логирование для удобства отслеживания процессов*

entities: ['dist/\*\*/\*.entity{.ts,.js}'], *//указываем путь к сущностям*

}),

],

Суть происходящего: в модуль приложения мы импортировали модуль библиотеки TypeOrm и задали первичные настройки подключения к БД. Теперь нам необходимо создать сущности в том виде, в котором TypeOrm их корректно воспримет. На данном тапе мы сможем установить реляционные связи для наших сущностей и связать их.

### Сущности

Приведем сущности к типу typeorm. Для этого немного отредактируем нашу сущность author в файле **author.entity.ts** и внимательно изучаем комментарии к каждой строке кода.

import { Affiliation } from 'src/affiliations/entities/affiliation.entity';

import { Article } from 'src/articles/entities/article.entity';

import {

Column,

Entity,

JoinTable,

ManyToMany,

PrimaryGeneratedColumn,

} from 'typeorm';

@Entity('authors') *//указываем что это не просто клаcс, а сущность в рамках TypeOrm, в БД будет храниться как таблица*

export class Author {

@PrimaryGeneratedColumn() *//колонка - идентификатор, значение генерируется автоматически*

id: number;

@Column({}) *//колонка таблицы, сюда можно добавить большое количество параметров для БД, например тип, уникальность, триггер и т.д.*

fullname: string;

@Column()

position: string;

@Column()

grade: string;

@ManyToMany((type) => Article, (article) => article.authors) *//Создадим связь многие ко многим с сущностью article и свяжем с полем authors в статье*

@JoinTable({

*//join таблица с названием author\_article*

name: 'author\_article',

joinColumn: { name: 'author\_id' }, *//для связи с идентификатором автора*

inverseJoinColumn: { name: 'article\_id' }, *//для связи с идентификатором статьи*

})

articles: Article[]; *//объект, в котором будем автоматически получать все статьи автора*

@ManyToMany((type) => Affiliation, (affiliation) => affiliation.authors) *//тоже самое для аффилиаций*

@JoinTable({

name: 'author\_affiliation',

joinColumn: { name: 'author\_id' },

inverseJoinColumn: { name: 'affiliation\_id' },

})

affiliations: Affiliation[];

}

Сущность аффилиация(файл **affiliation.entity.ts**):

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

import {

Column,

Entity,

JoinTable,

ManyToMany,

PrimaryGeneratedColumn,

} from 'typeorm';

@Entity('affiliations')

export class Affiliation {

@PrimaryGeneratedColumn()

id: number;

@Column({ unique: true }) *//поле должно быть уникальным*

companyName: string;

@Column()

address: string;

@Column()

specialization: string;

@ManyToMany((type) => Author, (author) => author.affiliations)

@JoinTable({

name: 'author\_affiliation',

joinColumn: { name: 'affiliation\_id' },

inverseJoinColumn: { name: 'author\_id' },

})

authors: Author[];

}

Сущность статья(файл **article.entity.ts**):

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

import {

Column,

Entity,

JoinTable,

ManyToMany,

PrimaryGeneratedColumn,

} from 'typeorm';

@Entity('articles')

export class Article {

@PrimaryGeneratedColumn()

id: number;

@Column()

name: string;

@Column()

keywords: string;

@Column()

abstract: string;

@ManyToMany((type) => Author, (author) => author.articles)

@JoinTable({

name: 'author\_article',

joinColumn: { name: 'article\_id' },

inverseJoinColumn: { name: 'author\_id' },

})

authors: Author[];

}

Импортируем сущности внутри каждого модуля в модуль typeorm, пример **authors.module.ts**:

@Module({

controllers: [AuthorsController],

providers: [AuthorsService],

imports: [

DatasourceModule,

TypeOrmModule.forFeature([Author, Affiliation, Article]), *// !!! В модуле автор мы используем все три сущности, поэтому все три сущности необходимо импортирвоать!*

],

})

export class AuthorsModule {}

Для остальных модулей будет аналогичный импорт, в зависимости от нужных сущностей.

### Объекты DTO:

Зачастую, во время передачи данных клиенту, возникает потребность в сегрегации отправляемых данных по принципу их назначения. Для этого принято разделять сущность хранения данных в БД и сущность передачи данных клиентам. Для возможности передачи данных (исходя из нужд конечного клиента) используются такие объекты, как Data Transfer Object (DTO).

Например, нам необходимо предоставить урезанную информацию об авторе для не авторизованных пользователей. И мы хотим передать только имя и должность. Конечно же, мы можем в сервисе автора логически выявлять такие параметры, а потом в зависимости от той или иной ситуации отправлять нужные данные, но, такой подход чреват «загрязнением» кода и усложнением процесса редактирования. Каждый метод обработки должен по возможности выполнять одно действие (принцип разделения ответственности). Другое возможное использование DTO – это изменение формата добавления или изменения данных. Например, при создании автора нам не обязательно получать полный объект аффилиаций автора, достаточно лишь получить список ID аффилиаций автора и уже самим в сервисе привязать их к автору.

На данный момент у нас есть три модели, попробуем для каждой из них создать DTO. Для этого создадим папку dto в папке authors. А в ней создадим файл **incomplete-author.dto.ts**

export class IncompleteAuthorDto {

id: number;

fullName: string;

position: string;

}

При запросе от неавторизованного пользователя будет возвращаться список авторов с урезанной информацией об авторах.

Далее, в той же папке создадим DTO для возможности создания нового автора

**AuthorDTO.cs**

export class CreateAuthorDto {

fullname: string;

position: string;

grade: string;

affiliations: number[];

}

Сущность **CreateAuthorDto** имеет небольшие различия с моделью автора, эти различия состоят в отсутствии статей, и типе аффилиаций. Подразумевается, что у нового созданного автора не может быть статей, и он может работать только в тех организациях, что уже были созданы ранее. Поэтому, имея список идентификаторов аффилиаций мы можем в методе создания автора найти все рабочие места исходя из массива их идентификаторов.

### Контроллеры новых сущностей

Создание контроллеров аналогично созданию контроллера Автора из предыдущей лабораторной. Отличие будет лишь в добавлении нового пути для возможности отправки не полной информации по авторам.

Добавим новый путь:

@Get('incomplete')

findIncomplete() {

*this*.authorsService.findIncomplete();

}

Теперь, если в пути, при обращении к контроллеру авторов добавить **incomplete,** то будет вызван метод отправки неполной информации авторов.

### Миграции

В прямом виде делать миграции делать не очень удобно, для удобства процесса TypeOrm позволяет их генерировать на основе заданных сущностей, чем мы и воспользуемся.

Для этого создадим папку в корне проекта с названием **configurations** а в ней файл **typeorm.config.ts**

В нем пропишем параметры подключения к базе данных и путь до наших сущностей.

import { DataSource } from 'typeorm';

const ormConfig: DataSource = new DataSource({

type: 'postgres',

host: 'localhost',

port: 5432,

database: 'education',

username: 'education',

password: 'password',

entities: ['dist/\*\*/\*.entity{.ts,.js}'],

logging: true,

synchronize: false,

migrationsTableName: 'migrations',

migrations: ['dist/src/migrations/\*{.ts,.js}'],

});

export default ormConfig;

Для удобства пропишем команды для генерации, отката и применения миграций в БД в файле **package.json**

Package.json – файл-манифест, который определяет информацию о приложении, версию, подключенные зависимости, подключенные модули и т.д. В том числе там задаются команды запуска и отладки приложения. Зачастую, команда для выполнения того или иного действия может содержать в себе ссылки на конфигурации, дополнительные флаги и параметры, для упрощения вызова рутинных команд их удобно прописывать в этом файле и вызывать короткой командой, в чем мы, в скором времени, и убедимся.

В область **scripts** добавим новый скрипт с названием typeorm.

"typeorm": "node --require ts-node/register ./node\_modules/typeorm/cli.js "

"migration-generate": "npm run build && npm run typeorm -- migration:generate --dataSource configurations/typeorm.config.ts "

Итак, для генерации миграции нам необходимо выполнить следующую команду

**npm run migration-generate src/migrations/initial**

Рассмотрим подробнее эту команду: мы вызываем скрипт migration-generate в котором происходит сборка проекта, далее в скрипте генерации вызывается скрипт typeorm и подставляется команда генерации миграции, а в качестве аргумента **dataSource** подставляется путь до ранее созданного нами файла. Стоит обратить внимание на построение названия миграции, мы указываем путь до папки с миграциями и в конце название самой миграции(в нашем случае initial).

### Доработка сервисов для работы с БД

В прошлой лабораторной работе в качестве источника данных использовалась псевдо-база с массивом объектов. Минусы такого подхода очевидны – наши данные удалялись при перезапуске сервера, мы имели только локальный доступ к хранимой информации.

Для возможности взаимодействия сервиса с репозиторием сущности данных нам необходимо создать его контекст внутри сервиса и указать в качестве аргумента в конструкторе класса.

Для примера будет подробно рассмотрен и описан сервис **AuthorService**. Из прошлой лабораторной мы выяснили назначение сервиса и то, как остальные модули сервера взаимодействуют с сервисами. Также, данный сервис является самым «емким», поскольку содержит связи с сущностями аффилиация и статья.

Создадим контексты репозиториев сущностей **Author, Affiliation и Article** добавим аргумент в конструктор класса **AuthorService**:

constructor(

@InjectRepository(Author)

private readonly authorRepository: Repository<Author>, *// "внедряем" репозиторий Author в сервис*

@InjectRepository(Affiliation)

private readonly affiliationRepository: Repository<Affiliation>, *// "внедряем" репозиторий Affiliation в сервис*

@InjectRepository(Article)

private readonly articleRepository: Repository<Article>, *// "внедряем" репозиторий Artilcle в сервис*

) {}

Теперь при инициализации сервиса ему в качестве аргумента будет отправляться экземпляр контекста репозитория (это внедрение зависимостей). Далее рассмотрим измененный вариант методов для CRUD-операций.

Метод создания автора:

async create(authorDto: CreateAuthorDto): Promise<Author>

{

*//получаем объект CreateAuthorDto*

const author = *this*.authorRepository.create(); *//создаем объект Author из репозитория*

author.fullname = authorDto.fullname; *//заполняем поля объекта Author*

author.position = authorDto.position;

author.grade = authorDto.grade;

const affiliations = await *this*.affiliationRepository.findBy({

*//получаем массив Affiliation по id*

id: In(authorDto.affiliations),

});

author.affiliations = affiliations;

await *this*.authorRepository.save(author); *//сохраняем объект Author в БД*

return author; *//возвращаем объект Author*

}

Метод получения авторов:

async findAll(): Promise<Author[]> {

const authors = await *this*.authorRepository.find({

*//получаем связанные объекты*

relations: {

affiliations: true,

articles: true,

},

}); *//получаем массив Author из БД*

return authors; *//возвращаем массив Author*

}

Результатом выполнения этого метода является список всех авторов в БД. Отдельное внимание стоит обратить на параметр relations, в нем мы указываем элементы из других сущностей, которые хотели бы включить в ответ на запрос. Если не добавить **relations**, то в ответе будут только те поля, что напрямую относятся к автору, без учета реляционного взаимодействия с другими сущностями, простыми словами в ответе не будет аффилиаций и статей.

Метод получения автора по ID:

findOne(id: number): Promise<Author> {

*// Promise<Author> - указывает, что функция возвращает объект Author в виде Promise (c асинхронного потока)*

return *this*.authorRepository.findOne({

*//получаем объект Author по id*

where: { id }, *//указываем условие поиска по id*

relations: { affiliations: true, articles: true }, *//получаем связанные объекты*

});

}

Данный метод схож по выполняемым функциям с предыдущим, за исключением поиска в коллекции автора с конкретным идентификатором.

Метод получения неполной информации об авторе:

async findIncomplete(): Promise<IncompleteAuthorDto[]> {

const authors = await *this*.authorRepository.find(); *//получаем массив Author из БД*

const incompleteAuthors: IncompleteAuthorDto[] = authors.map((author) => {

*//преобразуем массив Author в массив IncompleteAuthorDto*

const incompleteAuthor = new IncompleteAuthorDto();

incompleteAuthor.id = author.id;

incompleteAuthor.position = author.position;

return incompleteAuthor;

});

return incompleteAuthors; *//возвращаем массив IncompleteAuthorDto*

}

Метод обновления информации об авторе:

async update(id: number, updatedAuthor: Author) {

*//получаем объект Author для обновления по id*

const author = await *this*.authorRepository.findOne({ where: { id } }); *//получаем объект Author по id из БД*

author.fullname = updatedAuthor.fullname; *//обновляем поля объекта Author*

author.position = updatedAuthor.position;

author.grade = updatedAuthor.grade;

author.affiliations = updatedAuthor.affiliations;

author.articles = updatedAuthor.articles;

await *this*.authorRepository.save(author); *//сохраняем объект Author в БД*

return author; *//возвращаем объект Author*

}

Метод удаления автора:

remove(id: number) {

*this*.authorRepository.delete({ id }); *//удаляем объект Author из БД*

}

По такому же принципу мы можем изменить сервисы других моделей и привести их к работе с БД.

Учесть всю специфику работы с TypeOrm в рамках лабораторной работы невозможно. После пройденного курса по СУБД вы обладаете навыками проектирования и взаимодействия с базами данных, остается только понять, как реализовывать данную функциональность при помощи TypeOrm!

### Базовая документация

**!!! Копировать весь код из этого блока не нужно! Здесь приведена часть кода соответствующих файлов/классов для понимания куда необходимо добавлять декораторы документации!**

Добавим декоратор в контроллер авторов для структурирования методов

Импорт:

import { ApiTags } from '@nestjs/swagger';

Декоратор **ApiTags**

@Controller('authors')

@ApiTags('Авторы') *// Тег для документации*

export class AuthorsController {

Добавим декоратор **ApiOperation** для post запроса на создание автора

@ApiOperation({ summary: 'Создание автора' }) *// Операция для Swagger*

@Post()

create(@Body() createAuthor: CreateAuthorDto) {

Добавим декоратор **ApiProperty** для полей сущности Author

@Entity('authors') *//указываем что это не просто клас, а сущность в рамках TypeOrm, в БД будет храниться как таблица*

export class Author {

@ApiProperty({ example: '1', description: 'Уникальный идентификатор' })

@PrimaryGeneratedColumn() *//колонка - идентификатор, значение генерируется автоматически*

id: number;

@ApiProperty({ example: 'Иванов Иван Иванович', description: 'ФИО' })

@Column({}) *//колонка таблицы, сюда можно добавить большое количество параметров для БД, например тип, уникальность, триггер и т.д.*

fullname: string;

Добавим декоратор **ApiProperty** для полей **CreateAuthorDto**

import { ApiProperty } from '@nestjs/swagger';

export class CreateAuthorDto {

@ApiProperty({ example: 'Иванов Иван Иванович', description: 'ФИО' })

fullname: string;

@ApiProperty({ example: 'Профессор', description: 'Должность' })

position: string;

@ApiProperty({ example: 'Доцент', description: 'Ученая степень' })

grade: string;

@ApiProperty({

example: [1, 2],

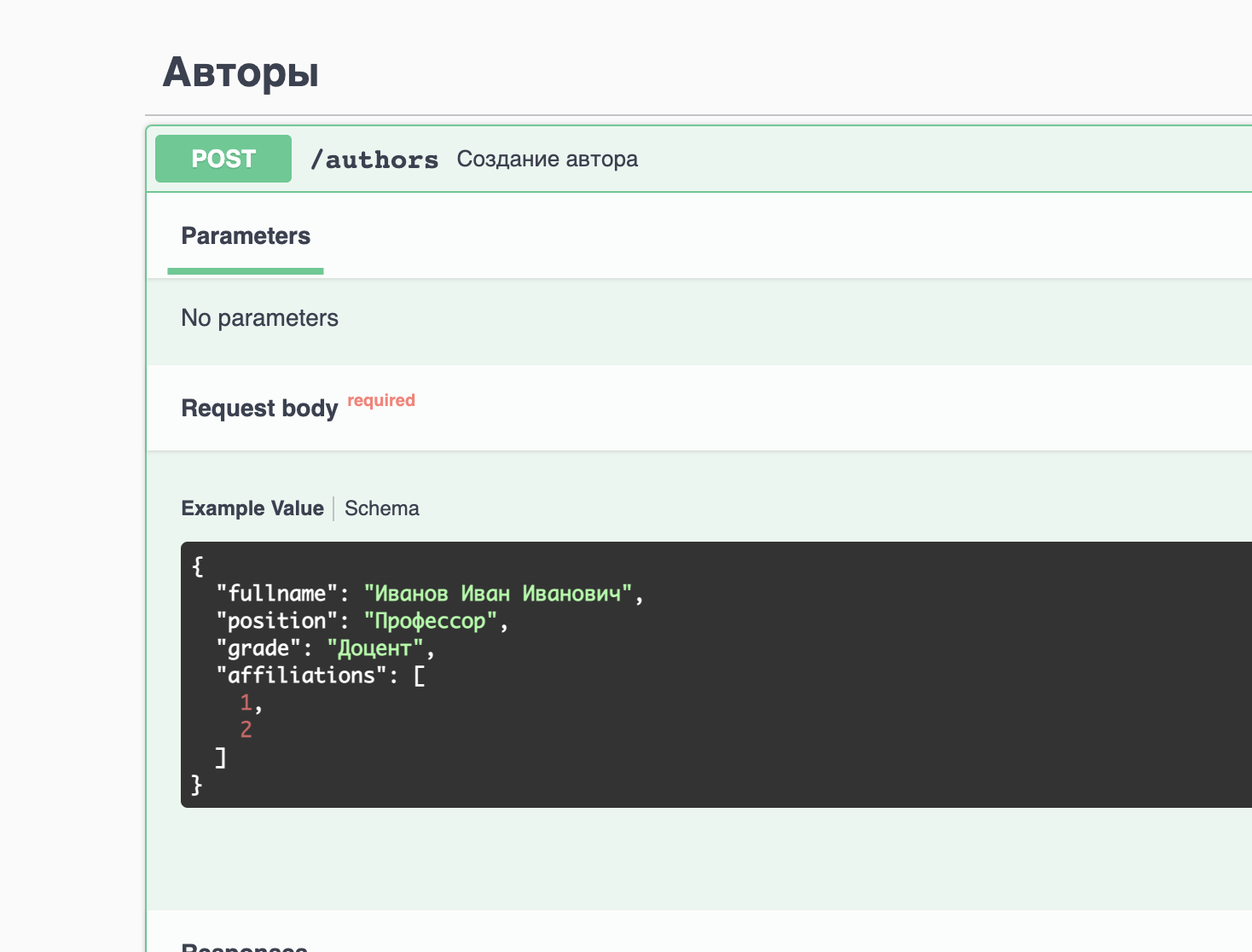
description: 'Список идентификаторов аффилиаций',

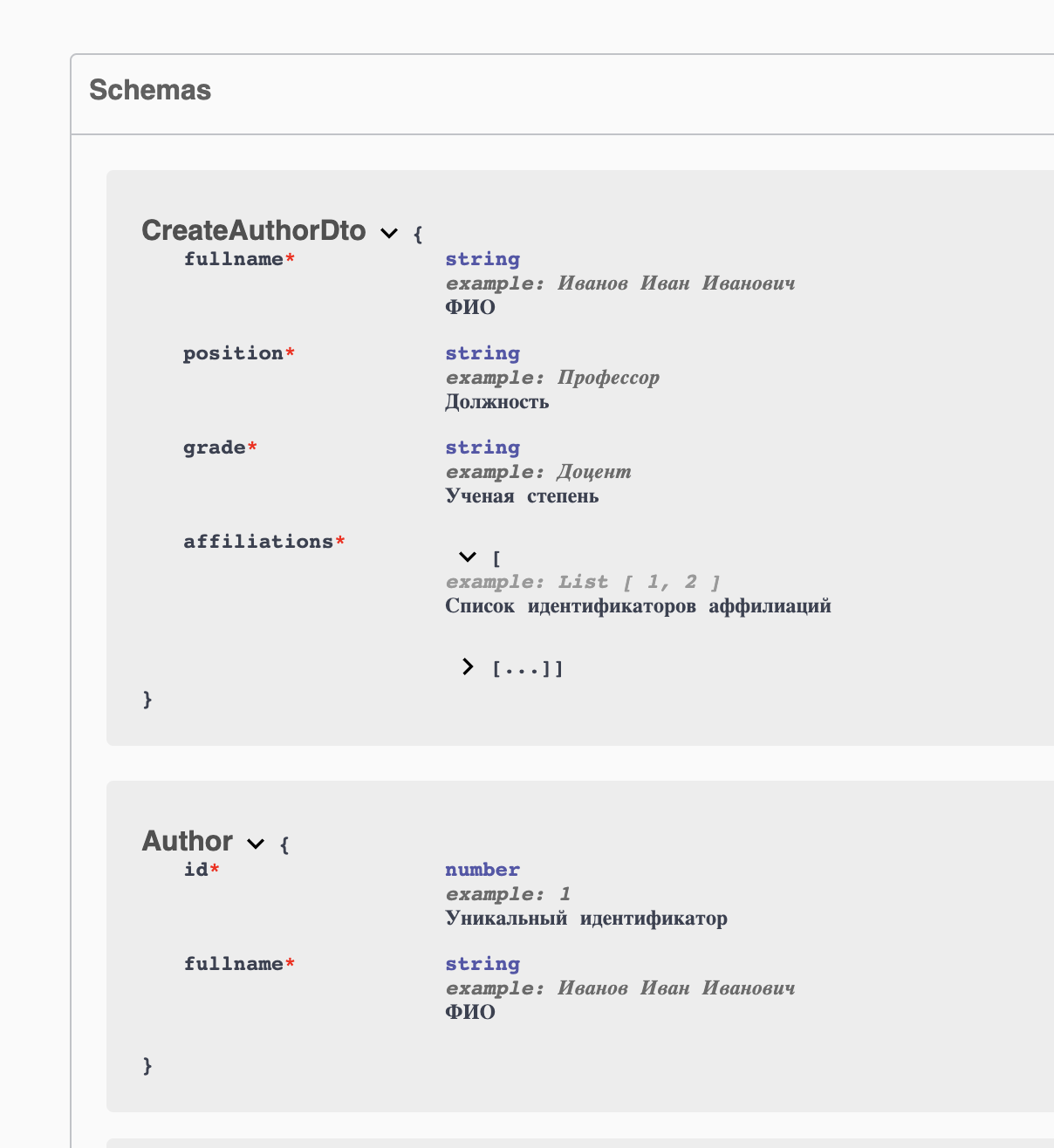
})

affiliations: number[];

}

Как это выглядит в Swagger:





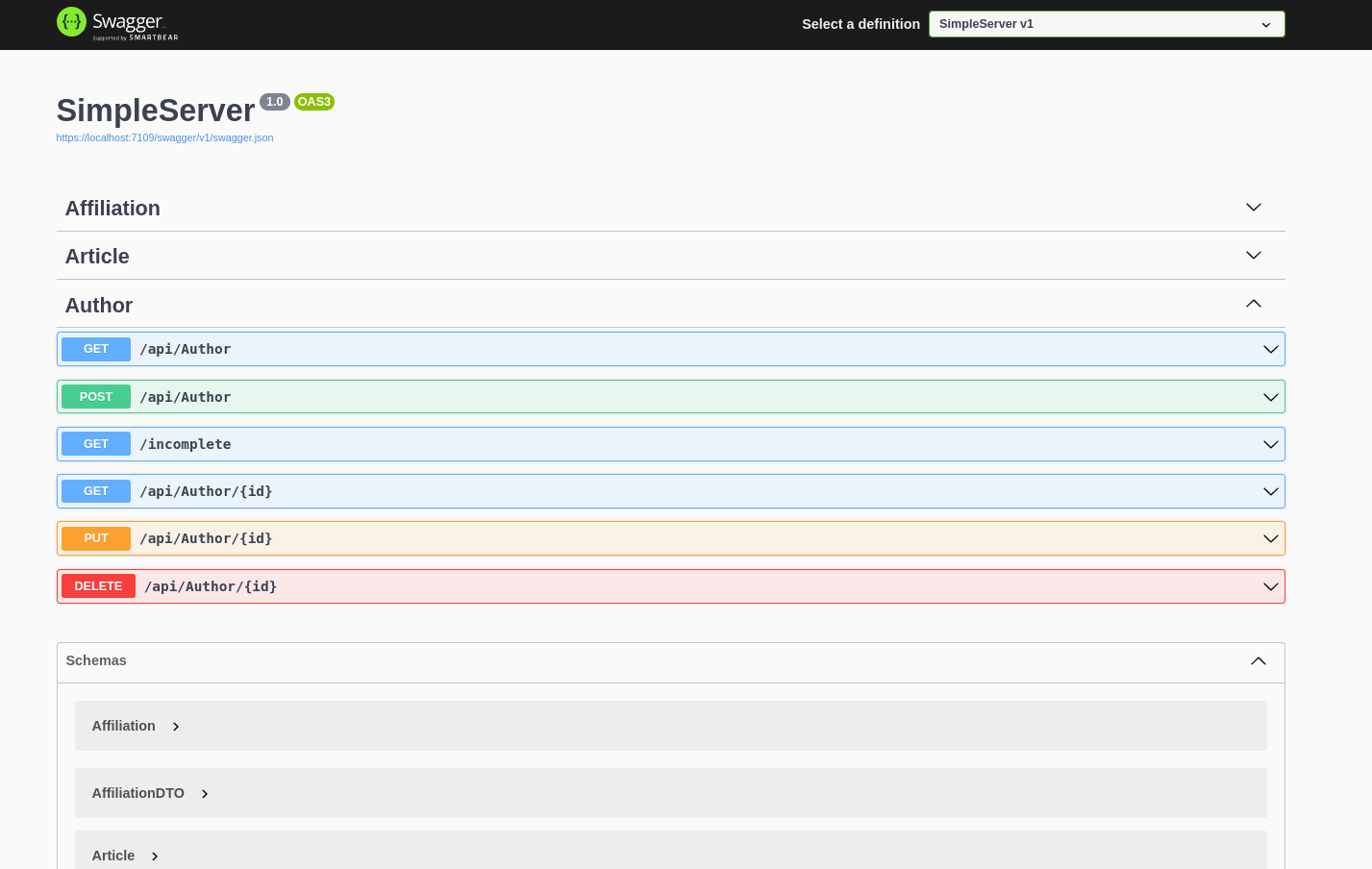
Сервер готов к запуску!

Теперь в командной строке необходимо набрать **npm run start:debug**, и, если вы все сделали правильно, то ваш сервер запустится!

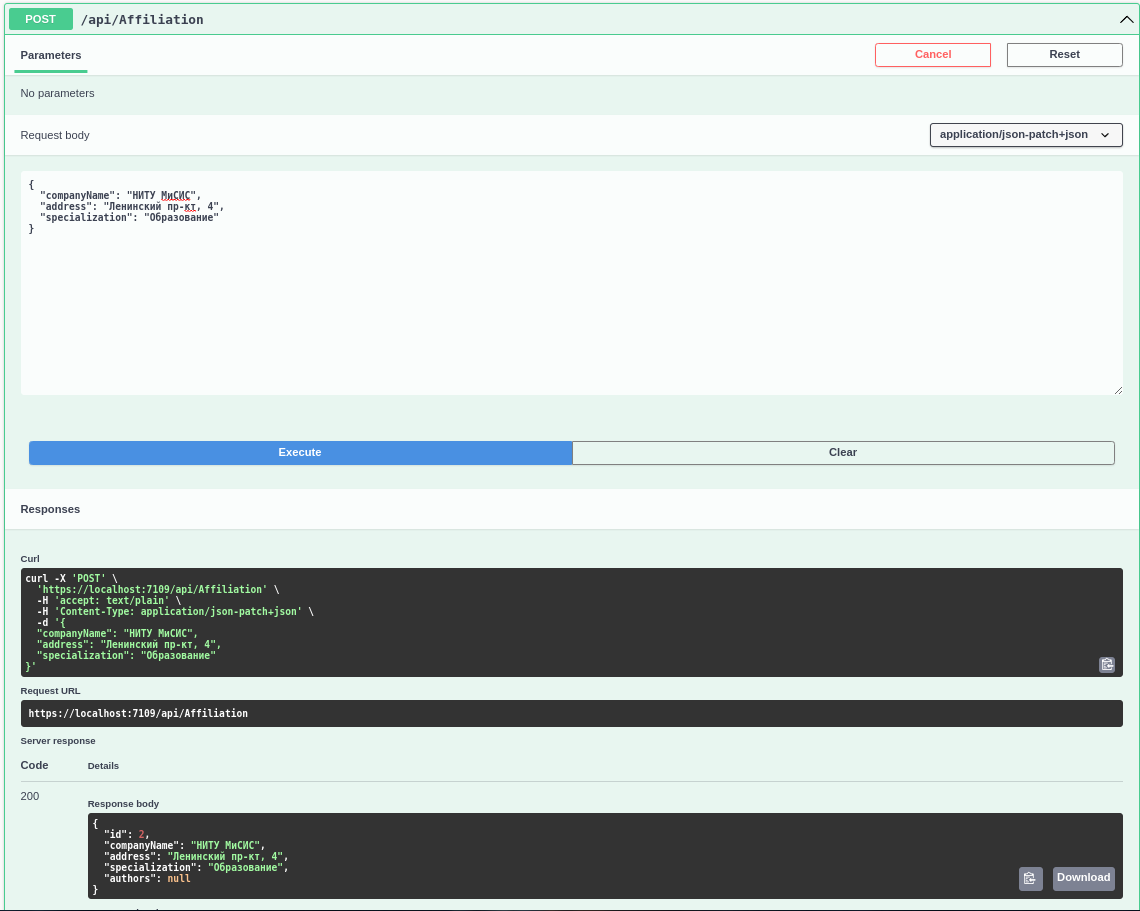
### Проверка в Swagger

Переходим по ссылке **http://localhost:3001/api\_docs#/**

При запуске сервера мы можем наблюдать все ранее созданные контроллеры, пути и схемы моделей.

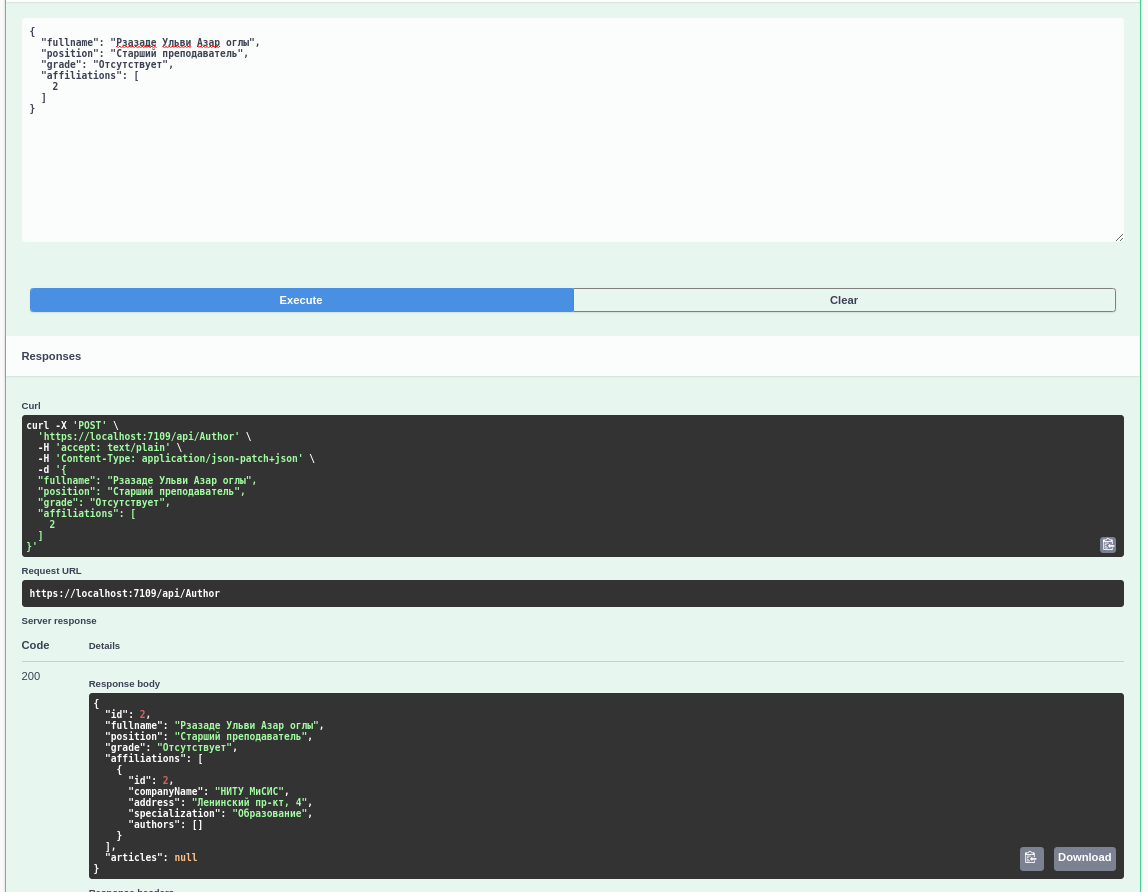


Создадим аффилиацию:



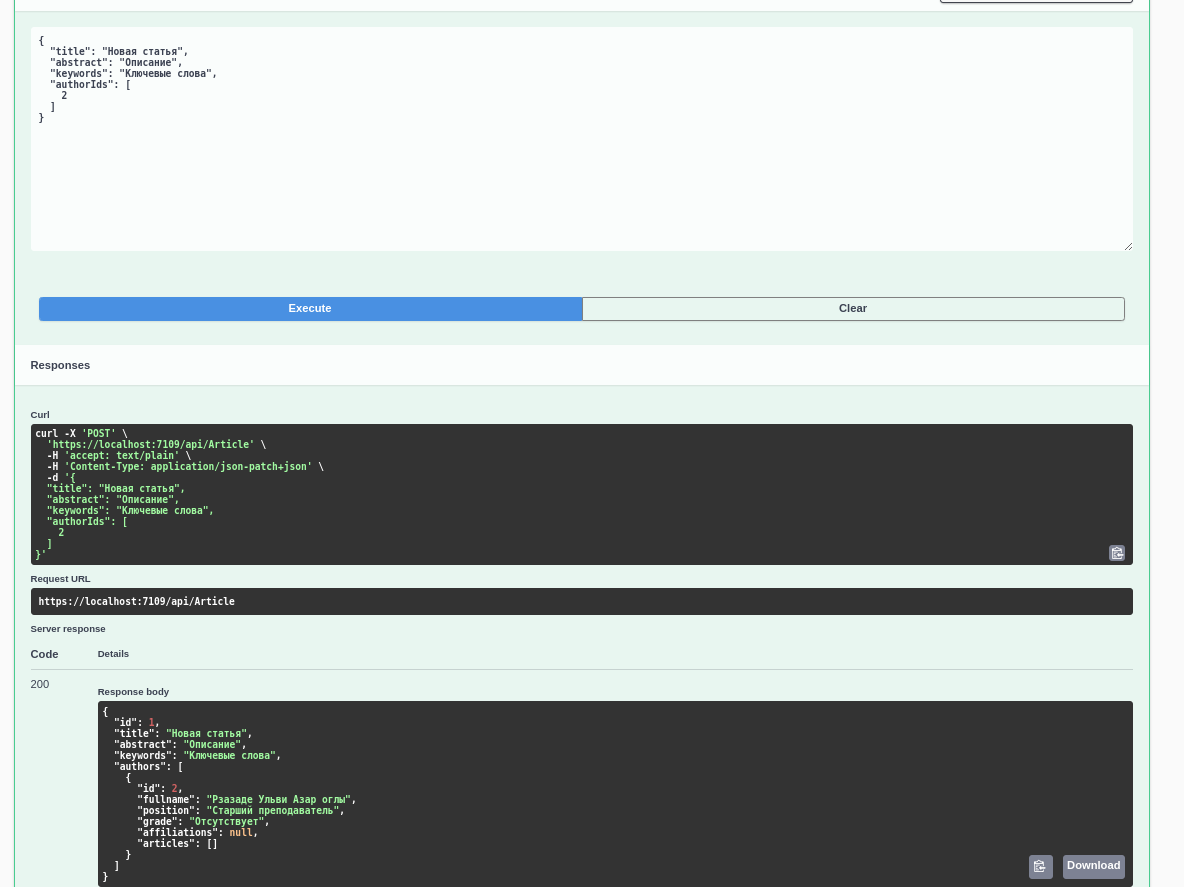
Аффилиация НИТУ МИСиС успешно создана!

Добавим автора и укажем ему аффилиацию:



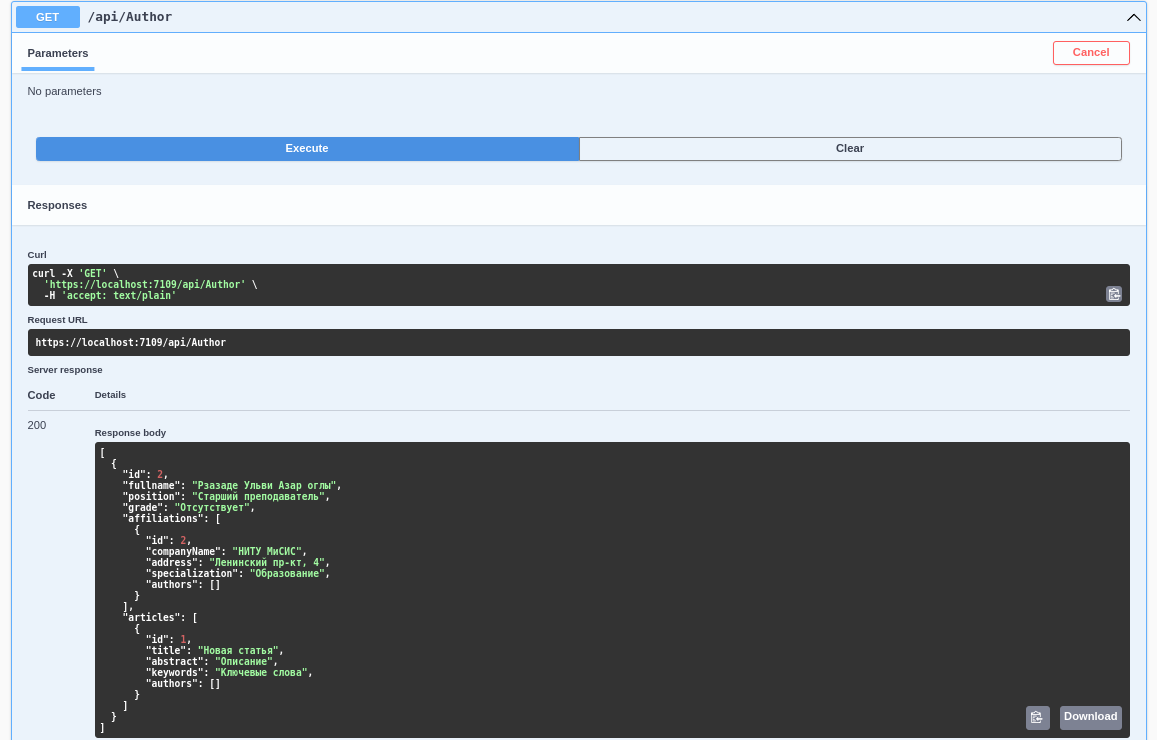
Сервер создал автора, и, поскольку мы указали идентификатор аффилиации, которую создали ранее, в качестве места работы автора, то она к нему привязалась и была возвращена в ответе от сервера.

Создадим статью и укажем автора:



Сервер вернул статью и привязанного к ней автора. Обратите внимание, что аффилиация автора не вернулась в ответе, поскольку в логике сервиса статьи, которая в данной работе не описана в явном виде было решено не возвращать место работы.

Теперь вернемся к контроллеру автора и попробуем получить всех авторов:



Обратите внимание, что в данном ответе мы можем наблюдать как все аффилиации автора, так и его статьи. это возможно благодаря расширению **relations**, что мы использовали в сервисе автора.

### Задание

1. На основе изученного материала инициализировать БД PostgreSQL (или другую реляционную БД).
2. Подготовить все ранее созданные модели для возможности взаимодействия с БД.
3. Обозначить реляционные связи между моделями в соответствии с выбранной вами темой.
4. Создать не менее двух сущностей DTO.
5. Для каждой DTO исходя из логики реализовать метод контроллера.
6. Для каждого метода контроллера реализовать метод сервиса.
7. Переделать сервисы, созданные вами в л.р. №2, под работу с БД.

### Рекомендации

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