

Задание №3

Стажировка

«Веб-приложение для публикации постов - боилерплейт»

Выполнил: Ноздряков Богдан Валериевич

Проверил: Пармузин Александр Игоревич

Санкт-Петербург

2024

Условие:

Оформить боилерплейт проекта на TypeScript, Express.js, Sequelize + SQLite/PostgreSQL(на выбор)

В последствии будет приложение, где каждый пользователь может писать свои посты, как личный дневник

По итогу ожидается:

- Оформленная структура распределения файлов
- Инициализирована ORM Sequelize, описаны модели, есть миграции, при запуске проекта создается БД, имя которой указано в конфиге
- Предоставлены пара API по которым можно получить сохраненные посты и имеющихся пользователей

Примечание:

Данный отчет содержит решение, в котором предоставлен код и структура базы данных с комментариями. Также данный отчет содержит тесты, в которых предоставлены результаты различных тестов написанного api.

Запуск приложения – команда `npm run start:dev`. После этого можно перейти на сайт - нужно в браузере ввести <http://localhost:port>. Чтобы перейти к документации Swagger – нужно ввести в браузере <http://localhost:port/api>.

!Примечание: в адресе вместо port нужно указать порт, на котором слушает приложение. Приложение определяет порт в специальном конфиге (`./src/app.config.ts`), который затем передает вычисленное значение порта в первый исполняемый файл приложения (`./src/main.ts`), где этот порт уже будет использован для слушания:

```
export default () => ({
  port: parseInt(process.env.PORT, 10) || 3000
});
```

То есть в качестве порта приложения – будет указан либо тот, который записан в конфигурационном файле `.env`, либо 3000 (если нет файла `.env`, или в нем нет переменной `PORT`, или переменная `PORT` есть, но значение указано такое, которое не может являться портом приложения).

У меня есть файл `.env`, в котором переменная `PORT` определена так:

```
PORT=5000
```

Поэтому мое приложение слушает на порту 5000 и в браузере я ввожу <http://localhost:5000> или <http://localhost:5000/api>.

Если же, как я написал раньше - нет файла `.env`, или в нем нет переменной `PORT`, или переменная `PORT` есть, но значение указано такое, которое не может являться портом приложения, то нужно вводить `http://localhost:3000` или `http://localhost:3000/api`.

Решение:

Для решения поставленной задачи используем фреймворк NodeJS – NestJS. Данный фреймворк предназначен для создания масштабируемых серверных приложений. NestJS позволяет использовать принципы ООП, различные паттерны проектирования и установленные решения, основанные на модульности, легкой поддержки, безопасности типов и внедрении зависимостей.

Стек приложения: TypeScript, Express.js, Sequelize и PostgreSQL (может быть расширен в последствии разработки).

Структура приложения:

Приложение будет поддерживать архитектуру, основанную на принципах MVC, Solid и ООП, благодаря чему получится достигнуть следующих преимуществ – понятность, легкость поддержки, расширяемость и безопасность.

В корне проекта будут храниться:

- различные конфигурационные файлы (package.json, nest-cli.json, tsconfig.json, .env и т.д.)
- тесты (директория ./test)
- статические ресурсы (директория ./public)
- источник кода (директория ./src)
- информация о проекте (Боилерплейт.pdf, db.pgerd, README.md)

Остановимся подробно на источнике кода (все, что находится по директории ./src). Мы воспользуемся подходом, при котором весь код разделяется на модули и их подмодули. При этом все файлы будут именовать по следующей логике:

модуль, к которому принадлежит файл.назначение файла.ts

Пример: app.controller.ts

app – модуль, к которому относится файл app.controller.ts

controller – назначение файла

На данный момент в ./src есть главный модуль – app. Его не принято выносить явно в отдельный модуль, поэтому все файлы app (app.module.ts, app.service.ts и т.д.) будут просто лежать в ./src. Но уже все остальные файлы – придется выносить в отдельные модули с их подмодулями.

Также в ./src будут находиться модули domain и database. Здесь модуль domain будет отвечать за все модули, которые создают и взаимодействуют с API, а database будет отвечать за все, что связано с работой с базой данных: схема, миграции и т.д. В последствии разработки приложения – будут появляться новые модули, например, filter, auth, validator и т.д.

Сейчас можно точно сказать, что потребуется создать модуль filter, который будет содержать фильтры, перехватывающие определенные исключения, которые возникают во время работы приложения. Добавим в ./src этот модуль. Добавим в него следующие классы:

- 1) AllExceptionsFilter – предназначенный для перехвата любой ошибки:

```
@Catch()
class AllExceptionsFilter implements ExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    const logger = new Logger(AllExceptionsFilter.name);
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();

    let status = 500;
    let errorMessage = null;

    if (exception instanceof HttpException) {
      status = exception.getStatus();
      errorMessage = exception.message;
    } else if (exception instanceof Error) {
      errorMessage = exception.message;
    }

    const errorResponse = {
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      message: errorMessage,
    };

    if (exception instanceof Error) {
      logger.error(
        `Error occurred: ${JSON.stringify(errorResponse)}`,
        exception.stack,
      );
    } else {
      logger.error(
        `Error occurred: ${JSON.stringify(errorResponse)}`,
        'Exception is not an instance of Error, stack trace is not available.',
      );
    }

    response.status(status).json(errorResponse);
  }
}
```

- 2) NotFoundExceptionFilter – предназначенный для перехвата всех NotFoundException исключений в приложении:

```
@Catch(NotFoundException)
class NotFoundExceptionFilter implements ExceptionFilter {
    private readonly logger = new Logger(NotFoundExceptionFilter.name);

    catch(exception: NotFoundException, host: ArgumentsHost) {
        const ctx = host.switchToHttp();
        const response = ctx.getResponse<Response>();
        const request = ctx.getRequest<Request>();

        const errorResponse = {
            statusCode: HttpStatus.NOT_FOUND,
            timestamp: new Date().toISOString(),
            path: request.url,
            message: exception.message,
        };

        this.logger.error(
            `Error occurred: ${JSON.stringify(errorResponse)}`,
            exception.stack,
        );

        response.status(HttpStatus.NOT_FOUND).json(errorResponse);
    }
}
```

- 3) ValidationExceptionFilter - предназначенный для перехвата всех HttpException, связанных с ошибками валидации, в приложении:

```
@Catch(HttpException)
class ValidationExceptionFilter implements ExceptionFilter {
    catch(exception: HttpException, host: ArgumentsHost) {
        const ctx = host.switchToHttp();
        const response = ctx.getResponse();
        const excMessage = exception.getResponse() as any;

        let formattedErrors = [];
        if (Array.isArray(excMessage.message)) {
            formattedErrors = excMessage.message.map((error) => ({
                field: error.field,
                message: error.message,
            }));
        } else {
            formattedErrors = [
                {
                    field: 'general',
                }
            ];
        }
    }
}
```

```

        message: excMessage.message,
      },
    ];
  }

  response.status(HttpStatus.UNPROCESSABLE_ENTITY).json({
    statusCode: HttpStatus.UNPROCESSABLE_ENTITY,
    message: 'Validation failed',
    errors: formattedErrors,
  });
}
}

```

Не забудем подключить все эти фильтры глобально в приложении, подключив их в главном модуле приложения AppModule (./src/app.module.ts) и настроив в main.ts. Данные фильтры нужны, чтобы перехватывать определенные исключения в приложении, централизованно управлять ими и возвращать их в качестве ответа.

Подключение TypeScript:

При создании приложения NestJS — происходит автоматическое подключение TypeScript, как основного языка приложения. Все зависимости TypeScript можно увидеть в файле package.json:

```
44 "devDependencies": {
45   "@nestjs/cli": "^10.0.0",
46   "@nestjs/schematics": "^10.0.0",
47   "@nestjs/testing": "^10.0.0",
48   "@types/express": "^4.17.17",
49   "@types/jest": "^29.5.2",
50   "@types/node": "^20.14.0",
51   "@types/sequelize": "^4.28.20",
52   "@types/supertest": "^6.0.0",
53   "@typescript-eslint/eslint-plugin": "^6.0.0",
54   "@typescript-eslint/parser": "^6.0.0",
55   "cross-env": "^7.0.3",
56   "eslint": "^8.42.0",
57   "eslint-config-prettier": "^9.0.0",
58   "eslint-plugin-prettier": "^5.0.0",
59   "jest": "^29.5.0",
60   "prettier": "^3.0.0",
61   "source-map-support": "^0.5.21",
62   "supertest": "^6.3.3",
63   "ts-jest": "^29.1.0",
64   "ts-loader": "^9.4.3",
65   "ts-node": "^10.9.2",
66   "tsconfig-paths": "^4.2.0",
67   "typescript": "^5.1.3"
68 },
```

Также можно посмотреть конфигурацию TypeScript в tsconfig.json:

TS tsconfig.json > {} compilerOptions

```
1  {
2    "compilerOptions": {
3      "module": "commonjs",
4      "declaration": true,
5      "removeComments": true,
6      "emitDecoratorMetadata": true,
7      "experimentalDecorators": true,
8      "allowSyntheticDefaultImports": true,
9      "target": "ES2021",
10     "esModuleInterop": true,
11     "sourceMap": true,
12     "outDir": "./dist",
13     "baseUrl": "./",
14     "incremental": true,
15     "skipLibCheck": true,
16     "strictNullChecks": false,
17     "noImplicitAny": false,
18     "strictBindCallApply": false,
19     "forceConsistentCasingInFileNames": false,
20     "noFallthroughCasesInSwitch": false
21   },
22   "include": [
23     "src/**/*.ts"
24   ]
25 }
```


Подключение приложения к Express.js:

Фреймворк NestJS позволяет интегрировать приложение с веб-сервером Express.js, передав в качестве типа экземпляра приложения специальный тип – NestExpressApplication при создании экземпляра приложения (./src/main.ts):

```
import { NestExpressApplication } from '@nestjs/platform-express';
import { ConfigService } from '@nestjs/config';
import { join } from 'path';
import AppModule from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(AppModule);
  const configService = app.get(ConfigService);
  const port = configService.get('PORT');
  app.useStaticAssets(join(__dirname, '..', 'public'));

  await app.listen(port);
}

bootstrap();
```

Подключение приложения к Sequelize и локальному серверу PostgreSQL:

Для подключения ORM Sequelize к приложению, достаточно установить необходимые зависимости в package.json приложения:

```
"sequelize": "^6.37.3",
"sequelize-cli": "^6.6.2",
"sequelize-typescript": "^2.1.6",
```

Здесь:

- sequelize – для работы с реляционными базами данных через ORM Sequelize
- sequelize-cli – командная строка для Sequelize, предоставляющая набор утилит для упрощения работы с Sequelize, позволяя, например, генерировать миграции
- sequelize-typescript – пакет, предоставляющий интеграцию между Sequelize и TypeScript

Определение всех моделей для приложения:

Всего будет 4 модели – User, UserContact, Post, Subscription (./src/database/schema/):

- 1) Модель User – представляет пользователя приложения:

```
@Table
class UserModel extends Model {
  @Column({
    type: DataType.UUID,
    primaryKey: true,
    defaultValue: () => uuid(),
    unique: true,
    allowNull: false
  })
  id!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  name!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  role!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  email!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  password!: string;

  @Column({
    type: DataType.DATE,
    defaultValue: new Date(),
    allowNull: false
  })
  createdAt!: Date;

  @UpdatedAt
  @Column({
    type: DataType.DATE,
```

```

        allowNull: false
    })
    updatedAt!: Date;

    @Column({
        type: DataType.INTEGER,
        defaultValue: 0,
        allowNull: false
    })
    rating!: number;

    @HasMany(() => PostModel)
    posts!: PostModel[];

    @HasMany(() => UserContactModel)
    contacts!: UserContactModel[];

    @BelongsToMany(() => UserModel, {
        through: () => SubscriptionModel,
        foreignKey: 'subscriberId',
        otherKey: 'userId'
    })
    subscribers!: UserModel[];
}

```

Здесь:

- **id** – обязательное поле, которое не может быть null или undefined. Представляет из себя уникальный идентификатор пользователя, который является первичным ключом. Заполняется сервером автоматически, выполняя функцию `uuid`, чтобы получить строковый уникальный идентификатор
- **name** - обязательное поле, которое не может быть null или undefined. Представляет из себя строковое имя пользователя в приложении. Должно передаваться в объекте запроса
- **role** - обязательное поле, которое не может быть null или undefined. Представляет из себя роль пользователя в приложении. Роли может быть две – админ и обычный пользователь. В зависимости от роли – у пользователя будет разный функционал. Должно передаваться в объекте запроса
- **email** - обязательное поле, которое не может быть null или undefined. Представляет из себя email пользователя. Должно передаваться в объекте запроса
- **password** - обязательное поле, которое не может быть null или undefined. Представляет из себя пароль пользователя. Должно передаваться в объекте запроса

- `createdAt` - обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя дату создания пользователя. Заполняется сервером автоматически за счет создания экземпляра `Date` в качестве значения по умолчанию
- `updatedAt` - обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя дату изменения пользователя. Заполняется сервером автоматически за счет декоратора `@UpdatedAt`
- `rating` - обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя рейтинг пользователя в приложении. Заполняется сервером автоматически, если в объекте запроса нет данного параметра (пользователь только создан, по умолчанию рейтинг равен 0). Если же передавать в объекте запроса данный параметр, то он будет записан
- `posts` – представляет из себя ассоциацию, которая будет определена либо как пустой массив `Post`, либо как массив `Post`. У пользователя могут либо быть посты, либо их может не быть. Если они есть – их может быть много. У поста же обязательно должен быть пользователь, причем только один. Это отношение один ко многим. Чтобы его реализовать – создаем данную ассоциацию с помощью декоратора `@HasMany`
- `contacts` - представляет из себя ассоциацию, которая будет определена либо как пустой массив `UserContact`, либо как массив `UserContact`. У пользователя могут либо быть дополнительные контакты, либо их может не быть. Если они есть – их может быть много. У контакта же обязательно должен быть пользователь, причем только один. Это отношение один ко многим. Чтобы его реализовать – создаем данную ассоциацию с помощью декоратора `@HasMany`
- `subscribers` - представляет из себя ассоциацию, которая будет определена либо как пустой массив `User`, либо как массив `User`. Пользователь может подписаться на другого пользователя, если ему интересно читать его посты. У пользователя может быть много подписчиков. Также у его подписчиков может быть тоже много подписчиков. Это отношение многие ко многим. Чтобы его реализовать – воспользуемся декоратором `@BelongsToMany` через вспомогательную модель `Subscription`

2) Модель UserContact:

```
@Table
class UserContactModel extends Model {
  @Column({
    type: DataType.UUID,
    primaryKey: true,
    defaultValue: () => uuid(),
    unique: true,
    allowNull: false
  })
  id!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  type!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  value!: string;

  @ForeignKey(() => UserModel)
  @Column({
    type: DataType.UUID,
    allowNull: false
  })
  userId!: string;

  @BelongsTo(() => UserModel)
  user!: UserModel;
}
```

Здесь:

- id – обязательное поле, которое не может быть null или undefined. Представляет из себя уникальный идентификатор контакта пользователя, который является первичным ключом. Заполняется сервером автоматически, выполняя функцию `uuid`, чтобы получить строковый уникальный идентификатор
- type - обязательное поле, которое не может быть null или undefined. Представляет из себя тип контакта пользователя. Должно передаваться в объекте запроса

- value - обязательное поле, которое не может быть null или undefined.
Представляет из себя значение контакта пользователя. Должно передаваться в объекте запроса
- userId - обязательное поле, которое не может быть null или undefined.
Представляет из себя id пользователя, которому принадлежит данный контакт. Является внешним ключом, который ссылается на столбец id в модели User. Реализуется за счет декоратора @ForeignKey
- user - представляет из себя ассоциацию, которая представляет из себя пользователя. Если у пользователя дополнительные контакты могут либо быть, либо не быть, а если они есть, то их может быть много, то у контакта пользователь должен обязательно быть, причем один. Это отношение один ко многим. Чтобы это показать – используется декоратор @BelongsTo

3) Модель Subscription:

```
@Table
class SubscriptionModel extends Model {
  @Column({
    type: DataType.UUID,
    primaryKey: true,
    defaultValue: () => uuid(),
    unique: true,
    allowNull: false
  })
  id!: string;

  @Column({
    type: DataType.STRING,
    allowNull: false
  })
  type!: string;

  @Column({
    type: DataType.DATE,
    allowNull: true
  })
  period?: Date;

  @ForeignKey(() => UserModel)
  @Column({
    type: DataType.UUID
  })
  userId!: string;

  @ForeignKey(() => UserModel)
```

```
@Column({
  type: DataType.UUID
})
subscriberId!: string;
}
```

Здесь:

- `id` – обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя уникальный идентификатор подписки, который является первичным ключом. Заполняется сервером автоматически, выполняя функцию `uuid`, чтобы получить строковый уникальный идентификатор
- `type` - обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя тип подписки. Должно передаваться в объекте запроса
- `period` - необязательное поле, которое может быть `null` или `undefined`. В таком случае подписка – бессрочная. Если же оно определено, то это тип `Date`, который соответствует периоду подписки. Необязательно передавать в объекте запроса
- `userId` - обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя `id` пользователя, на которого подписываются. Является внешним ключом, который ссылается на столбец `id` в модели `User`. Реализуется за счет декоратора `@ForeignKey`
- `subscriberId` - обязательное поле, которое не может быть `null` или `undefined`. Представляет из себя `id` пользователя, который подписывается. Является внешним ключом, который ссылается на столбец `id` в модели `User`. Реализуется за счет декоратора `@ForeignKey`

4) Модель Post:

```
@Table
class PostModel extends Model {
  @Column({
    type: DataType.UUID,
    primaryKey: true,
    defaultValue: () => uuid(),
    unique: true,
    allowNull: false
  })
  id!: string;

  @Column({
    type: DataType.STRING,
```

```

        allowNull: false
    })
    title!: string;

    @Column({
        type: DataType.DATE,
        defaultValue: new Date(),
        allowNull: false
    })
    createdAt!: Date;

    @UpdatedAt
    @Column({
        type: DataType.DATE,
        allowNull: false
    })
    updatedAt!: Date;

    @Column({
        type: DataType.TEXT,
        allowNull: false
    })
    content!: string;

    @Column({
        type: DataType.INTEGER,
        defaultValue: 0,
        allowNull: false
    })
    rating!: number;

    @Column({
        type: DataType.ARRAY(DataType.STRING),
        allowNull: true
    })
    tags?: string[] | null | undefined;

    @ForeignKey(() => UserModel)
    @Column({
        type: DataType.UUID
    })
    authorId!: string;

    @BelongsTo(() => UserModel)
    author!: UserModel;
}

```

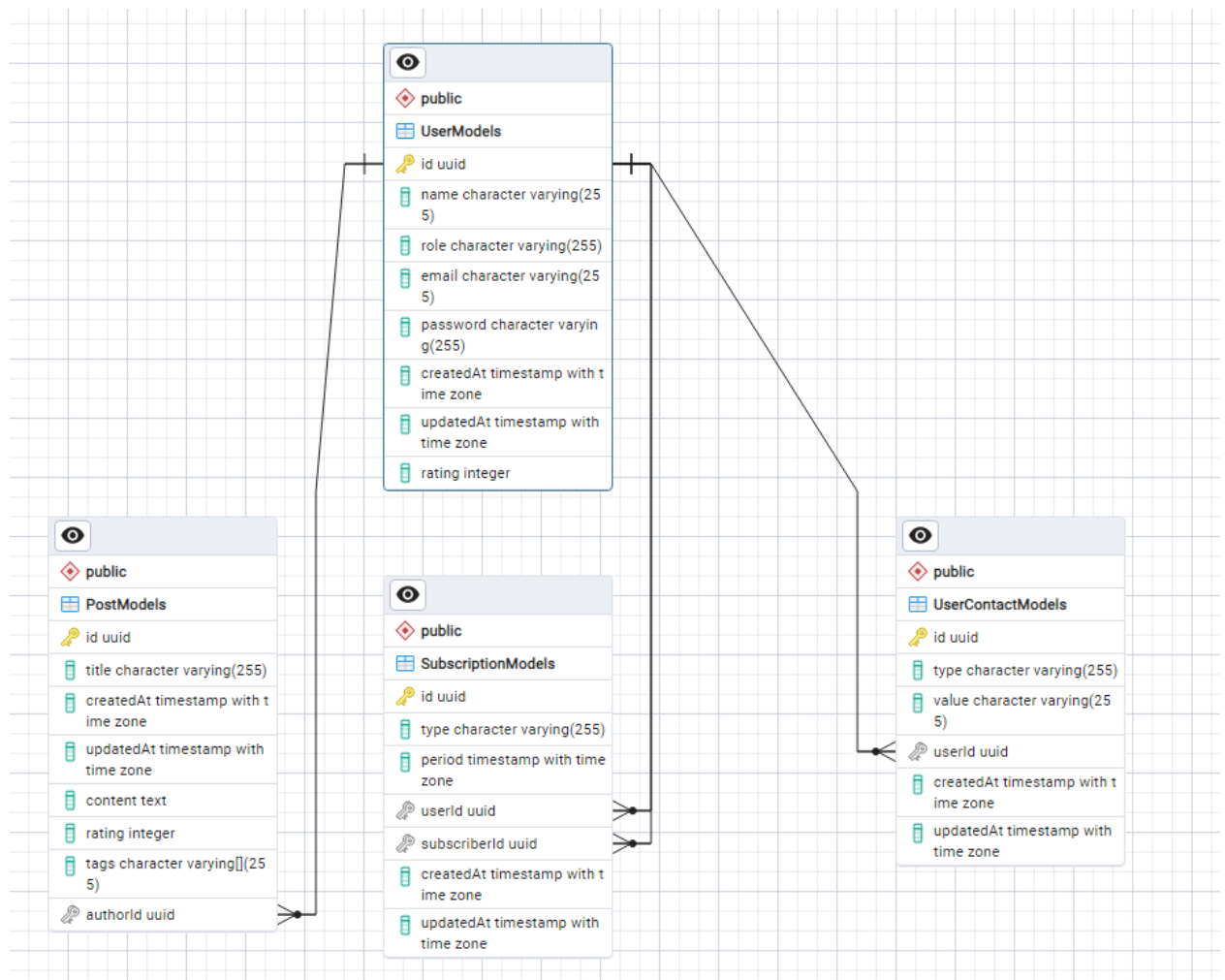

Здесь:

- `id` – обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя уникальный идентификатор поста пользователя, который является первичным ключом. Заполняется сервером автоматически, выполняя функцию `uuid`, чтобы получить строковый уникальный идентификатор
- `title` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя заголовок поста. Должно передаваться в объекте запроса
- `createdAt` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя дату создания поста. Заполняется сервером автоматически за счет создания экземпляра `Date` в качестве значения по умолчанию
- `updatedAt` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя дату изменения поста. Заполняется сервером автоматически за счет декоратора `@UpdatedAt`
- `content` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя текст поста. Должно передаваться в объекте запроса
- `rating` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя рейтинг поста. Заполняется сервером автоматически, если в объекте запроса нет данного параметра (пост только создан, по умолчанию рейтинг равен 0). Если же передавать в объекте запроса данный параметр, то он будет записан
- `tags` – необязательное поле, которое может быть `null` или `undefined`. Если же оно определено, то это массив строк, представляющий из себя теги данного поста. То есть у поста как могут быть указаны теги, так и могут быть не указаны. Необязательно передавать в объекте запроса
- `authorId` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя `id` пользователя, которому принадлежит данный пост. Является внешним ключом, который ссылается на столбец `id` в модели `User`. Реализуется за счет декоратора `@ForeignKey`
- `author` - представляет из себя ассоциацию, которая представляет из себя пользователя. Если у пользователя посты могут либо быть, либо не быть, а если они есть, то их может быть много, то у поста пользователь должен обязательно быть, причем один. Это отношение один ко многим. Чтобы это показать – используется декоратор `@BelongsTo`

Получившиеся таблицы:

- ▼ Tables (4)
- > PostModels
- > SubscriptionModels
- > UserContactModels
- > UserModels

Схема в виде ERD-диаграммы (./db.pgerd):



Подключение к локальному серверу PostgreSQL:

Теперь для возможности подключения приложения к локальному серверу PostgreSQL – нужно определить модуль базы данных. В контексте ORM Sequelize – это можно выполнить несколькими способами:

- Создать пакет на основе Sequelize с нуля, используя пользовательские компоненты
- Использовать специальный @nestjs/sequelize пакет

Мы будем использовать модуль из @nestjs/sequelize, так как первый подход сопряжен с большими накладными расходами, которых можно избежать, как-раз используя модуль из @nestjs/sequelize. Для этого просто нужно импортировать в главный модуль приложения AppModule (/src/app.module.ts) модуль Sequelize и передать в него конфигурацию:

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { SequelizeModule } from '@nestjs/sequelize';
import AppController from './app.controller';
import AppService from './app.service';
import config from './app.config';
import UserModule from './domain/user/user.module';
import PostModule from './domain/post/post.module';
import UserModel from './database/schema/user/user.model';
import UserContactModel from './database/schema/user/user.contact.model';
import PostModel from './database/schema/post/post.model';
import SubscriptionModel from './database/schema/subscription/subscription.model';

@Module({
  imports: [
    ConfigModule.forRoot({
      load: [config]
    }),
    SequelizeModule.forRoot({
      dialect: 'postgres',
      host: process.env.DATABASE_HOST,
      port: Number(process.env.DATABASE_PORT),
      username: process.env.DATABASE_USERNAME,
      password: process.env.DATABASE_PASSWORD,
      database: process.env.DATABASE_NAME,
      models: [UserModel, UserContactModel, PostModel, SubscriptionModel],
      autoLoadModels: true
    }),
    UserModule,
    PostModule
  ],
  controllers: [AppController],
  providers: [AppService],
})
```

```
})  
export default class AppModule {}
```

Модуль Sequelize (SequelizeModule) имеет статический метод `forRoot`, который позволяет настроить подключение к базе данных на уровне приложения. Чтобы настроить подключение, нужно передать в данный метод объект, содержащий определенные опции:

- `dialect` – тип базы данных, к которой мы подключаемся. В нашем случае – это `postgres`, так как мы работаем с локальным сервером PostgreSQL
- `host` – имя или IP-адрес, на котором работает сервер базы данных
- `port` – номер порта, на котором слушает сервер базы данных
- `username` – имя пользователя, которое используется для аутентификации на сервере базы данных
- `password` – пароль, который используется для аутентификации на сервере базы данных
- `database` – имя базы данных, к которой мы хотим подключиться
- `models` – массив всех созданных моделей, на основе которых построится схема в базе данных
- `autoLoadModels` – специальный параметр, который при значении `true` позволяет автоматически загружать все модели

Здесь в файле конфигурации `.env` были определены большинство данных параметров, необходимых для подключения к базе данных. В результате – в статический метод `forRoot` передаются данные для подключения к базе данных из соответствующих переменных окружения, определенных в файле `.env` (все эти переменные окружения содержат соответствующие им значения из моей локальной базы данных).

Важно:

ORM Sequelize позволяет работать с базой данных двумя способами:

- 1) Использовать автоматическую синхронизацию и загрузку с базой данных от Sequelize
- 2) Полностью самостоятельно контролировать процесс миграций

Первый подход – Sequelize будет автоматически писать миграции под все изменения в схеме. Он будет это делать скрыто. То есть явная директория миграций не появится. Этот подход очень опасен, так как при его использовании есть риск потерять данные, при изменении схемы. Однако этот подход опасно применять только для production-версии

проекта. Данный подход рекомендуется применять именно для development-версии проекта, так как в такой версии – нам важно именно протестировать наше приложение, чтобы убедиться, что все работает правильно, а потерять данные из базы данных при миграции от Sequelize не страшно, так как эти данные не настоящие, а тестовые для какой-то тестовой базы данных.

Именно по этой причине я воспользуюсь именно этим подходом. Сейчас мне важно просто протестировать приложение на стадии development. Однако я установил пакет sequelize-cli, о чем я писал раньше. Данный пакет позволит мне создавать свои миграции и самостоятельно ими управлять. Это как-раз позволяет использовать второй подход. Я специально его установил на будущее, чтобы начать полностью контролировать процесс миграций после того, как все будет протестировано. То есть, когда можно думать о production-версии.

Чтобы начать самостоятельно контролировать миграции при помощи Sequelize – нужно просто выполнить команду `npm sequelize-cli init`. Данная команда создаст явную директорию с миграциями, директорию с seed-файлами, директорию для моделей базы данных с файлом `index.js`, который будет добавлять все модели в Sequelize, конфигурационный файл `json` для подключения к базе данных и настроек окружения. Дальше я смогу создавать и контролировать миграции явно и самостоятельно с помощью команд `sequelize-cli`.

Написание api:

На данном этапе достаточно написать api только для пользователей и их постов. Также на данном этапе достаточно реализовать только метод `Get`. Однако, чтобы что-то получить, нужно сначала что-то создать. Поэтому также опишем метод `Post`:

1) Пользователи (`./src/domain/user`):

- a. Опишем метод `Post` для создания пользователя. Для начала – нужно решить, что можно отправлять в теле запроса. Это будет объект, содержащий следующие поля:

1. `name` – тип `string`, не может быть `undefined` или `null`. Представляет из себя имя пользователя
2. `role` – тип `string`, не может быть `undefined` или `null`. Представляет из себя роль пользователя
3. `email` – тип `string`, не может быть `undefined` или `null`. Представляет из себя почту пользователя
4. `password` – тип `string`, не может быть `undefined` или `null`. Представляет из себя пароль пользователя
5. `rating` – может быть `undefined` или `null`. Однако если это свойство задано – оно должно быть числом типа `Number`. Представляет из себя рейтинг пользователя

6. `contacts` - может быть `undefined` или `null`. Однако если это свойство задано – оно должно быть массивом, где каждый элемент – это JavaScript-объект, имеющий два поля: `type` и `value`. Оба не могут быть `undefined` или `null`. У обоих тип – `string`. Представляет из себя дополнительные контакты пользователя

Чтобы это реализовать – нужно подключить специальный класс DTO, который будет представлять из себя структуру отправляемого объекта в теле запроса. В последствии – мы сможем проводить валидацию этого объекта благодаря DTO.

Создадим в `./src/domain/user` новый модуль `dto`, который будет отвечать за все `dto`-классы в модуле `user`. Добавим туда два DTO класса:

1. `UserDTO` – будет отвечать за проверку свойств `name`, `role`, `email`, `password`, `rating`, `contacts`:

```
class UserDTO {
  @IsString()
  @NotEmpty()
  name: string;

  @IsString()
  @NotEmpty()
  role: string;

  @IsString()
  @NotEmpty()
  email: string;

  @IsString()
  @NotEmpty()
  password: string;

  @IsNumber()
  @Optional()
  rating?: number;

  @isArray()
  @ValidateNested({ each: true })
  @Type(() => UserContactDTO)
  @Optional()
  contacts?: UserContactDTO[];
}
```

2. `UserContactDTO` – будет отвечать за проверку свойств `type` и `value` в каждом объекте массива `contacts`:

```
class UserContactDTO {  
  @IsString()  
  @NotEmpty()  
  type: string;  
  
  @IsString()  
  @NotEmpty()  
  value: string;  
}
```

Данные DTO классы реализуем именно такую логику.

Теперь можно написать метод `Post` в контроллере пользователей (`./src/domain/user/user.controller.ts`), который будет принимать все `Post`-запросы для `/api/users`:

```
@Post()  
async createUser(@Body() userData: UserDTO, @Res() res: Response): Promise<Response | never> {  
  const newUser = await this.userService.createUser(userData);  
  res.status(201).location(`/api/users/${newUser.id}`).send(newUser);  
  return res;  
}
```

Данный метод будет получать объект из тела запроса. После будет проведена валидация полученного объекта за счет DTO-классов и за счет подключенного `ValidationPipe` к методу:

```
@UsePipes(new ValidationPipe({  
  transform: true,  
  exceptionFactory: (errors) => {  
    const formattedErrors = formatValidationErrors(errors);  
  
    return new HttpException(  
      {  
        statusCode: 422,  
        message: formattedErrors,  
      },  
      422,  
    );  
  },  
}))
```

Данный `ValidationPipe` проводит валидацию полученного объекта на основе классов DTO. Если валидация не прошла, то в таком случае будет вызван метод

formatValidationErrors из ./src/domain/validator/format.validator.ts:

```
interface ValidationError {
  field: string;
  message: string;
  children?: ValidationError[];
}

function formatContactErrors(
  contactErrors: any[],
  contactIndex: number,
): ValidationError[] {
  return contactErrors.reduce((acc: ValidationError[], error) => {
    if (error && error.constraints) {
      acc.push({
        field: `contacts[${contactIndex}] - ${error.property}`,
        message: Object.values(error.constraints).join(', '),
      });
    }
    return acc;
  }, []);
}

function formatValidationErrors(errors: any[]): ValidationError[] {
  return errors.reduce((acc: ValidationError[], error) => {
    if (error.constraints) {
      acc.push({
        field: error.property,
        message: Object.values(error.constraints).join(', '),
      });
    } else if (error.property === 'contacts') {
      error.children.forEach((childError, index) => {
        acc.push(...formatContactErrors(childError, index));
      });
    }

    if (error.children && error.children.length > 0) {
      acc.push(...formatValidationErrors(error.children));
    }

    return acc;
  }, []);
}

export default formatValidationErrors;
```

Данный метод возвращает массив с ошибками, где каждый элемент – это объект, содержащий два поля – первое показывает, где была совершена ошибка валидации, а второе – показывает саму ошибку валидации. ValidationPipe получает этот массив и бросает `HttpException` со статусом 422. Эту ошибку уже перехватит фильтр валидации `ValidationExceptionFilter` и вернет ее в качестве ответа на запрос.

Однако если же ошибок валидации нет, то контроллер передаст прошедший валидацию объект сервису UserService, который уже создаст пользователя в базе данных (для сделаем класс UserService зависимостью и инyectируем его в конструктор класса UserController, не забыв указать в UserModule, что UserService – это провайдер для создания синглтона UserService). После того, как контроллер получит от сервиса созданного пользователя – он вернет ответ, в заголовке Location которого будет содержаться ссылка на Get-метод получения созданного пользователя.

Так будет работать контроллер. Теперь рассмотрим, как сервис создает пользователя в базе данных:

```
async createUser(userData: UserDTO): Promise<UserModel | never> {
  const newUser = await this.userModel.create({
    name: userData.name,
    role: userData.role,
    email: userData.email,
    password: userData.password,
    rating: userData.rating
  });

  if (userData.contacts) {
    const contactsData = userData.contacts.map(contact => ({
      type: contact.type,
      value: contact.value,
      userId: newUser.id
    }));

    await UserContactModel.bulkCreate(contactsData);
  }

  return newUser;
}
```

Сервис создает нового пользователя в newUser, не учитывая возможность наличия дополнительных контактов в объекте запроса. Однако после создания нового пользователя – происходит проверка наличия дополнительных контактов. Если они есть, то создаются соответствующие записи в модели UserContact и все эти записи связываются со значением из модели User. Потом просто возвращается созданный пользователь

- b. Метод Get для получения всех существующих пользователей:

Напишем данный метод в контроллере пользователей:

```
@Get ()
  async getAllUsers(@Query('search') searchSubstring: string = ''): Promise<UserModel[] | never> {
    return this.userService.getAllUsers(searchSubstring);
  }
```

Данный метод будет обрабатывать все Get-запросы на /api/users. Также данный метод будет иметь возможность не только получать всех пользователей, но и фильтровать их, то есть искать каких-то определенных пользователей по какому-то критерию. Для это реализуем логику, что в параметры URL данного Get-метода можно передавать какую-нибудь подстроку. Реализуется это с помощью декоратора Query – мы принимаем эту подстроку searchSubstring и устанавливаем ей значение по умолчанию – пустую строку. В таком случае, если при отправке запроса на этот Get-метод был передан параметр с подстрокой, то он будет использоваться для фильтрации. А если такой параметр не был передан, то подстроке присвоится пустое значение и фильтрации не будет.

Контроллер просто вызывает метод сервиса, передавая ему подстроку для фильтрации и возвращает результат работы сервиса.

Теперь рассмотрим эту логику в сервисе:

```
async getAllUsers(searchSubstring: string): Promise<UserModel[] | never> {
  let users = [];
  if (!searchSubstring) {
    users = await this.userModel.findAll();
  }
  else {
    users = await this.userModel.findAll({
      where: {
        [Op.or]: [
          { name: { [Op.like]: `>${searchSubstring}<` } },
          { email: { [Op.like]: `>${searchSubstring}<` } }
        ]
      }
    });

    if (users.length === 0) {
      throw new NotFoundException(`Users by search substring: ${searchSubstring} - not found`);
    }
  }
}
```

```
    return users;
}
```

Сервис просто проверяет – задана ли фильтрация. Если не задана, то сервис через Sequelize ищет всех пользователей и возвращает их (если пользователей нет, то вернется пустой массив). Если задана, то сервис сначала ищет все совпадения подстроки в столбце name пользователей. Если хоть что-то найдет, то вернет их. Если ничего не найдет, то будет искать все совпадения подстроки в столбце email пользователей. Если и после этого ничего не получится найти, то сервис выкинет исключение `NotFoundException`, которое будет говорить, что пользователи по такому фильтру не найдены, а уже глобальный фильтр `NotFoundExceptionFilter` перехватит это исключение и вернет его как ответ.

- с. Метод `Get` для получения одного определенного пользователя по переданному параметру `id` в запросе:

Напишем этот метод в контроллере пользователей:

```
@Get('/:id')
async getUserById(@Param('id') id: string): Promise<UserModel | never> {
    return this.userService.getUserById(id);
}
```

Данный метод будет получать переданный в параметры `id` пользователя и передавать ему сервису.

Рассмотрим сервис:

```
async getUserById(id: string): Promise<UserModel | never> {
    const user = await this.userModel.findByPk(id);
    if (!user) {
        throw new NotFoundException(`User with id ${id} - not found`);
    }

    return user;
}
```

Сервис просто ищет определенного пользователя по переданному `id` через Sequelize. Если найдет, то вернет этого пользователя, а если не найдет, то выкинет исключение `NotFoundException`, которое будет говорить, что пользователь по такому `id` не найден, а уже глобальный фильтр `NotFoundExceptionFilter` перехватит это исключение и вернет его как ответ.

2) Посты (./src/domain/post):

Для постов абсолютно вся логика будет аналогична пользователям, но настроенная под посты:

Решим, что можно отправлять в теле запроса. Это будет объект, содержащий следующие поля:

1. title – тип string, не может быть undefined или null. Представляет из себя заголовок поста
2. content – тип string, не может быть undefined или null. Представляет из себя текст поста
3. rating – может быть undefined или null. Однако если это свойство задано – оно должно быть числом типа Number. Представляет из себя рейтинг поста
4. tags - может быть undefined или null. Однако если это свойство задано – оно должно быть массивом, где каждый элемент – это строка. Представляет из себя теги поста
5. authorId – тип string, не может быть undefined или null. Представляет из себя id пользователя, который создает пост

Создадим в ./src/domain/post новый модуль dto, который будет отвечать за все dto-классы в модуле post. Добавим туда DTO класс:

```
class PostDTO {
  @ApiProperty({ description: 'Post title', example: 'Выходные на
природе!', type: String })
  @IsString()
  @NotEmpty()
  title: string;

  @ApiProperty({
    description: 'Post content',
    example: 'Недавно я решил выбраться на природу...',
    type: String
  })
  @IsString()
  @NotEmpty()
  content: string;

  @ApiProperty({ description: 'Post rating', example: 12, type: Number, required: false
})
  @IsNumber()
  @Optional()
```

```

rating?: number;

@ApiProperty({
  description: 'Post tags',
  example: ['природа', 'выходные'],
  type: [String],
  required: false
})
@isArray()
@isOptional()
tags?: string[];

@ApiProperty({
  description: 'Author id of the post',
  example: '123e4567-e89b-12d3-a456-426614174000',
  type: String
})
@isString()
@isNotEmpty()
authorId: string;
}

```

Данный DTO класс реализует такую логику.

Теперь можно переходить к api:

а. Метод Post для создания поста пользователя:

-Контроллер (./src/domain/post/post.controller.ts):

```

@UsePipes(new ValidationPipe({
  transform: true,
  exceptionFactory: (errors) => {
    const formattedErrors = formatValidationErrors(errors);

    return new HttpException(
      {
        statusCode: 422,
        message: formattedErrors,
      },
      422,
    );
  },
}))
@Post()
async createPost(@Body() postData: PostDTO, @Res() res: Response): Promise<Response | never> {
  const newPost = await this.postService.createPost(postData);
  res.status(201).location(`/api/posts/${newPost.id}`).send(newPost);
  return res;
}

```

```
}
```

-Сервис (./src/domain/post/post.service.ts):

```
async createPost(postData: PostDT0): Promise<PostModel | never> {  
    await this.userService.getUserById(postData.authorId);  
  
    return (  
        await this.postModel.create({  
            ...postData  
        })  
    );  
}
```

- b. Метод Get для получения всех постов. Можно фильтровать по заголовку и контенту поста:

-Контроллер:

```
@Get()  
async getAllPosts(@Query('search') searchSubstring: string = ''): Promise<PostModel[] | never> {  
    return this.postService.getAllPosts(searchSubstring);  
}
```

-Сервис:

```
async getAllPosts(searchSubstring: string): Promise<PostModel[] | never> {  
    let posts = [];  
    if (!searchSubstring) {  
        posts = await this.postModel.findAll();  
    }  
    else {  
        posts = await this.postModel.findAll({  
            where: {  
                [Op.or]: [  
                    { title: { [Op.like]: `%${searchSubstring}%` } },  
                    { content: { [Op.like]: `%${searchSubstring}%` } }  
                ]  
            }  
        });  
  
        if (posts.length === 0) {  
            throw new NotFoundException(`Posts by search substring: ${searchSubstring} - not found`);  
        }  
    }  
}
```

```
    return posts;
}
```

с. Метод Get для получения поста пользователя через его id:

-Контроллер:

```
@Get('/:id')
async getPostById(@Param('id') id: string): Promise<PostModel | never> {
    return this.postService.getPostById(id);
}
```

-Сервис:

```
async getPostById(id: string): Promise<PostModel | never> {
    const post = await this.postModel.findByPk(id);
    if (!post) {
        throw new NotFoundException(`User with id ${id} – not found`);
    }

    return post;
}
```

Тесты:

Для того, чтобы протестировать написанный api – воспользуемся OpenApi спецификацией – модулем Swagger. Для начала – установим необходимые зависимости для Swagger:

```
"@nestjs/swagger": "^7.3.1",
```

И

```
"swagger-ui-express": "^5.0.1"
```

Эти зависимости можно посмотреть в package.json. Теперь необходимо инициализировать и настроить SwaggerModule в главном исполняемом файле (./src/main.ts):

```
import { NestFactory } from '@nestjs/core';
import { NestExpressApplication } from '@nestjs/platform-express';
import { ConfigService } from '@nestjs/config';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { join } from 'path';
import AppModule from '../app.module';

async function bootstrap() {
    const app = await NestFactory.create<NestExpressApplication>(AppModule);
```

```

const configService = app.get(ConfigService);
const port = configService.get('PORT');

if (process.env.SERVER_DESCR === 'Local') {
  process.env.SERVER_URL += `:${port}`;
}

const swaggerConfig = new DocumentBuilder()
  .setTitle('REST API для базы данных пользователей и их постов')
  .setDescription('API для управления пользователями и их постами')
  .setVersion('1.0')
  .addTag('users')
  .addTag('posts')
  .addServer(process.env.SERVER_URL, process.env.SERVER_DESCR)
  .build()

const swaggerDocument = SwaggerModule.createDocument(app, swaggerConfig);

SwaggerModule.setup('api', app, swaggerDocument);
app.useStaticAssets(join(__dirname, '..', 'public'));

await app.listen(port);
}

bootstrap();

```

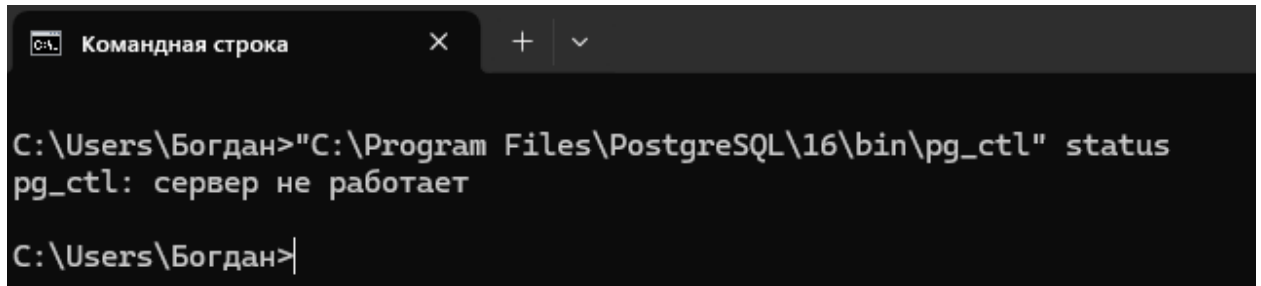
Здесь:

- `import { SwaggerModule, DocumentBuilder }` – импортировали `SwaggerModule` (для настройки и запуска `Swagger UI` в приложении, делая документацию API доступной через браузер) и `DocumentBuilder` (для построения конфигурационного объекта `Swagger`, который описывает мой API) из `@nestjs/swagger`
- `swaggerConfig` – строится конфиг для `Swagger` через `DocumentBuilder`. Там я вызываю методы, позволяющие описать мой API
- `swaggerDocument` – строится сама документация `Swagger` через `SwaggerModule`
- `SwaggerModule.setup` – настраивается документация `Swagger` для запуска `Swagger UI`, делая документацию API доступной по URL `/api`

Теперь можно настроить саму документацию в различных сущностях приложения (например, контроллеры, `dto` и т.д.) с помощью декораторов, начинающихся на `Api` (например, `ApiResponse`, `ApiOperation` и т.д.).

После настройки документации, можно после запуска приложения командой `npm run start:dev` – перейти в документацию API по адресу <http://localhost:port/api> и там протестировать написанный API.

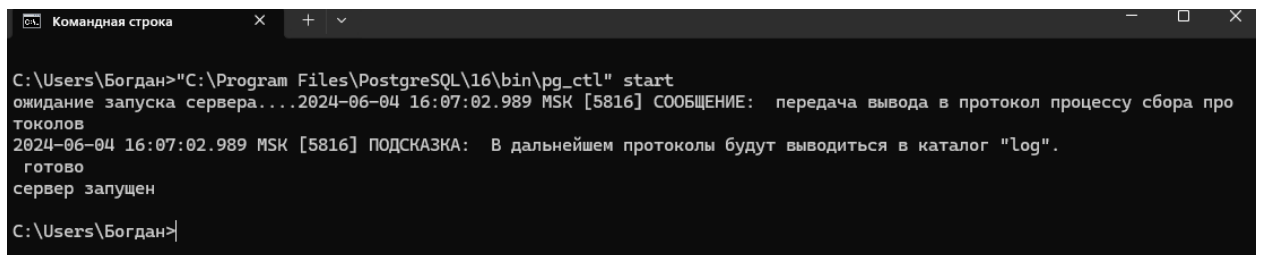
Но сначала убедимся, что локальный сервер PostgreSQL работает:



```
C:\Users\Богдан>"C:\Program Files\PostgreSQL\16\bin\pg_ctl" status
pg_ctl: сервер не работает

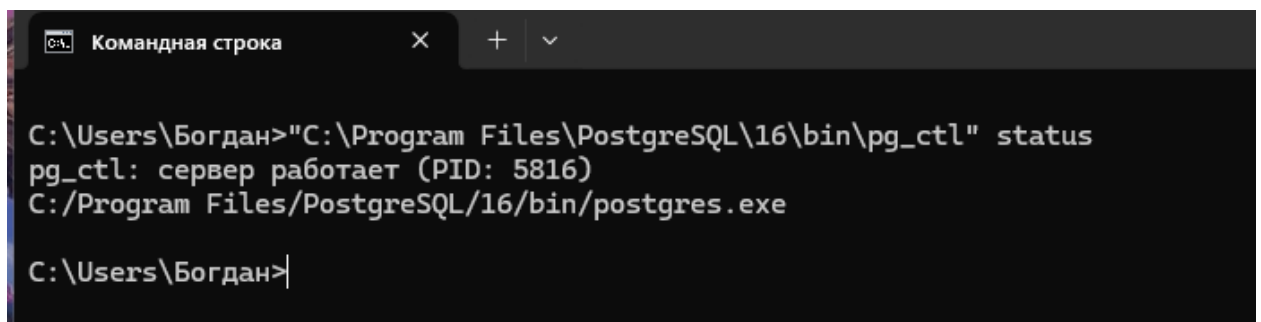
C:\Users\Богдан>
```

Как можно заметить, локальный сервер PostgreSQL не работает. Запустим его:



```
C:\Users\Богдан>"C:\Program Files\PostgreSQL\16\bin\pg_ctl" start
ожидание запуска сервера...2024-06-04 16:07:02.989 MSK [5816] СООБЩЕНИЕ:  передача вывода в протокол процессу сбора про
токолов
2024-06-04 16:07:02.989 MSK [5816] ПОДСКАЗКА:  В дальнейшем протоколы будут выводиться в каталог "log".
готово
сервер запущен

C:\Users\Богдан>
```



```
C:\Users\Богдан>"C:\Program Files\PostgreSQL\16\bin\pg_ctl" status
pg_ctl: сервер работает (PID: 5816)
C:/Program Files/PostgreSQL/16/bin/postgres.exe

C:\Users\Богдан>
```

Теперь локальный сервер PostgreSQL работает и можно переходить к тестированию api.

Выполним тесты:

1) Пользователи:

- а. Метод Post (Создание пользователя):

POST

/api/users

Create a new user

Cancel

Reset

Parameters

No parameters

Request body required

application/json

Examples:

[Modified value]

```
{
  "name": "bogdan_super",
  "role": "user",
  "email": "voyagerbv@gmail.com",
  "password": "nWkJK88?!",
  "contacts": [
    {
      "type": "phone",
      "value": "8 982 408 31 75"
    },
    {
      "type": "telegram",
      "value": "@zira839"
    }
  ]
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/users' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "bogdan_super",
    "role": "user",
    "email": "voyagerbv@gmail.com",
    "password": "nWkJK88?!",
    "contacts": [
      {
        "type": "phone",
        "value": "8 982 408 31 75"
      },
      {
        "type": "telegram",
        "value": "@zira839"
      }
    ]
  }'
```

Request URL

http://localhost:5000/api/users

Server response

Code

Details

201

Response body

```
{
  "createdAt": "2024-06-04T13:25:18.775Z",
  "id": "6f5e8bf2-5b3f-410a-9b63-03253b8083c6",
  "rating": 0,
  "name": "bogdan_super",
  "role": "user",
  "email": "voyagerbv@gmail.com",
  "password": "nWkJK88?!",
  "updatedAt": "2024-06-04T13:29:04.800Z"
}
```

Download

Response headers

```
connection: keep-alive
content-length: 226
content-type: application/json; charset=utf-8
date: Tue, 04 Jun 2024 13:29:04 GMT
etag: W/"e2-vfyUP5RIvfj218QkdkpLzg3Y"
keep-alive: timeout=5
location: /api/users/6f5e8bf2-5b3f-410a-9b63-03253b8083c6
x-powered-by: Express
```

b. Метод Get (Получение всех пользователей):

users

GET

/api/users

Get all users

Cancel

Parameters

Name	Description
search	Search substring that looks for all matches in either the user's name or email

(query)

search

Execute

Clear

Responses

Curl

curl -X 'GET' \n'http://localhost:5000/api/users' \n-H 'accept: application/json'

Request URL

http://localhost:5000/api/users

Server response

Code

Details

200

Response body

```
{\n  \"id\": \"6f5c8bf2-5b3f-410a-9b63-03253b0083c6\", \n  \"name\": \"bogdan_super\", \n  \"role\": \"user\", \n  \"email\": \"voyagerbvb@gmail.com\", \n  \"password\": \"aWkJK88?!!\", \n  \"createdAt\": \"2024-06-04T13:25:18.775Z\", \n  \"updatedAt\": \"2024-06-04T13:29:04.800Z\", \n  \"rating\": 0 \n}
```

Download

Response headers

```
connection: keep-alive\ncontent-length: 228\ncontent-type: application/json; charset=utf-8\ndate: Tue, 04 Jun 2024 13:29:41 GMT\netag: W/\"e4-z5i+9odEVI1xwg7481ADLoplyc\"\nkeep-alive: timeout=5\nx-powered-by: Express
```

users

GET

/api/users

Get all users

Parameters

Cancel

Name	Description
search	Search substring that looks for all matches in either the user's name or email
(query)	<input type="text" value="bogdan"/>

Execute

Clear

Responses

Curl

curl -X 'GET' \n'http://localhost:5000/api/users/search-bogdan' \n-H 'accept: application/json'

Request URL

http://localhost:5000/api/users/search-bogdan

Server response

Code

Details

200

Response body

```
{\n  \"id\": \"6f5c8bf2-5b3f-410a-9b63-03253b0083c6\", \n  \"name\": \"bogdan_super\", \n  \"role\": \"user\", \n  \"email\": \"voyagerbvb@gmail.com\", \n  \"password\": \"aWkJK88?!!\", \n  \"createdAt\": \"2024-06-04T13:25:18.775Z\", \n  \"updatedAt\": \"2024-06-04T13:29:04.800Z\", \n  \"rating\": 0 \n}
```

Download

Response headers

```
connection: keep-alive\ncontent-length: 228\ncontent-type: application/json; charset=utf-8\ndate: Tue, 04 Jun 2024 13:34:49 GMT\netag: W/\"e4-z5i+9odEVI1xwg7481ADLoplyc\"\nkeep-alive: timeout=5\nx-powered-by: Express
```

с. Метод Get (Получение определенного пользователя по id):

GET

/api/users/{id} Get a user by id

Cancel

Parameters

Name	Description
id * required	User id
string (path)	6f5e8bf2-5b3f-410a-9b63-03253b8083c6

ExecuteClear

Responses

Curl

curl -X 'GET' \n'http://localhost:5000/api/users/6f5e8bf2-5b3f-410a-9b63-03253b8083c6' \n-H 'accept: application/json'

Request URL

http://localhost:5000/api/users/6f5e8bf2-5b3f-410a-9b63-03253b8083c6

Server response

Code

Details

200

Response body

```
{  "id": "6f5e8bf2-5b3f-410a-9b63-03253b8083c6",  "name": "bogdan_super",  "role": "user",  "email": "voyagerbvd@gmail.com",  "password": "w!k30K88?!!",  "createdAt": "2024-06-04T13:25:18.775Z",  "updatedAt": "2024-06-04T13:29:04.800Z",  "rating": 0}
```

Download

Response headers

```
connection: keep-alive
content-length: 226
content-type: application/json; charset=utf-8
date: Tue, 04 Jun 2024 13:36:49 GMT
etag: W/"62-b5mc1fyE1BoC7HECd2l0/XbPpg"
keep-alive: timeout=5
x-powered-by: Express
```

2) Посты:

а. Метод Post (Создание поста):

POST

/api/posts Create a new post

Cancel

Reset

Parameters

No parameters

Request body * required

application/json

Examples:

[Modified value]

```
{  "title": "Первый проект на NextJS",  "content": "Сегодня разберемся с популярным фреймворком NextJS...",  "tags": [    "web",    "NextJS",    "TypeScript"  ],  "authorId": "6f5e8bf2-5b3f-410a-9b63-03253b8083c6"}
```

ExecuteClear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/posts' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "Первый проект на NestJS",
    "content": "Сегодня разберемся с популярным фреймворком NestJS...",
    "tags": [
      "web",
      "NestJS",
      "TypeScript"
    ],
    "authorId": "6f5e8bf2-5b3f-410a-9b63-03253b8083c6"
  }'
```

Request URL

http://localhost:5000/api/posts

Server response

Code	Details
201	<div><div>Response body</div><pre>{ "createdAt": "2024-06-04T13:50:53.880Z", "id": "6d56eec9-876b-4fde-a432-46d2d22515cd", "rating": 0, "title": "Первый проект на NestJS", "content": "Сегодня разберемся с популярным фреймворком NestJS...", "tags": ["web", "NestJS", "TypeScript"], "authorId": "6f5e8bf2-5b3f-410a-9b63-03253b8083c6", "updatedAt": "2024-06-04T13:52:11.413Z" }</pre><div>Download</div></div> <div><div>Response headers</div><pre>connection: keep-alive content-length: 374 content-type: application/json; charset=utf-8 date: Tue, 04 Jun 2024 13:52:11 GMT etag: W/"176-wBrEK0d0/+gKUXbrGCFqcVb4s" keep-alive: timeout=5 location: /api/posts/6d56eec9-876b-4fde-a432-46d2d22515cd x-powered-by: Express</pre></div>

b. Метод Get (Получение всех постов):

posts

GET /api/posts

Get all posts

Parameters

Cancel

Name	Description
search	Search substring that looks for all matches in either the post's title or content
(query)	<input type="text" value="search"/>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/posts

Server response

Code	Details
200	<div><div>Response body</div><pre>[{ "id": "6d56eec9-876b-4fde-a432-46d2d22515cd", "title": "Первый проект на NestJS", "createdAt": "2024-06-04T13:50:53.880Z", "updatedAt": "2024-06-04T13:52:11.413Z", "content": "Сегодня разберемся с популярным фреймворком NestJS...", "rating": 0, "tags": ["web", "NestJS", "TypeScript"], "authorId": "6f5e8bf2-5b3f-410a-9b63-03253b8083c6" }]</pre><div>Download</div></div> <div><div>Response headers</div><pre>connection: keep-alive content-length: 376 content-type: application/json; charset=utf-8 date: Tue, 04 Jun 2024 13:54:08 GMT etag: W/"178-HTruqAPW0juzl0VfakVKicXNa0Q" keep-alive: timeout=5 x-powered-by: Express</pre></div>

posts

GET /api/posts Get all posts

Parameters

Cancel

Name	Description
search <small>(query)</small>	Search substring that looks for all matches in either the post's title or content <div>NestJS</div>

Execute

Clear

Responses

Curl

curl -X 'GET' \n'http://localhost:5000/api/posts/search=NestJS' \n-H 'accept: application/json'

Request URL

http://localhost:5000/api/posts?search=NestJS

Server response

Code

Details

200

Response body

```
{
  "id": "6d56eec9-876b-4fde-a432-46d2d22515cd",
  "title": "Первый проект на NestJS",
  "createdAt": "2024-06-04T13:58:53.888Z",
  "updatedAt": "2024-06-04T13:52:11.413Z",
  "content": "Сегодня разберемся с популярным фреймворком NestJS...",
  "rating": 0,
  "tags": [
    "web",
    "NestJS",
    "TypeScript"
  ],
  "authorId": "6f5e8bf2-5b3f-418a-9b63-03253b8083c6"
}
```

Download

Response headers

connection: keep-alive
content-length: 376
content-type: application/json; charset=utf-8
date: Tue, 04 Jun 2024 13:55:38 GMT
etag: W/"178-HtrugUPN0juz10VfakVKIcXNcmQ"
keep-alive: timeout=5
x-powered-by: Express

с. Метод Get (Получение определенного поста по id):

GET /api/posts/{id} Get a post by id

Parameters

Cancel

Name	Description
id <small>required string (path)</small>	Post id <div>6d56eec9-876b-4fde-a432-46d2d22515cd</div>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/posts/6d56eec9-876b-4fde-a432-46d2d22515cd' \
-H 'accept: application/json'
```

Request URL

http://localhost:5000/api/posts/6d56eec9-876b-4fde-a432-46d2d22515cd

Server response

Code

Details

200

Response body

```
{
  "id": "6d56eec9-876b-4fde-a432-46d2d22515cd",
  "title": "Первый проект на NestJS",
  "createdAt": "2024-06-04T13:50:53.880Z",
  "updatedAt": "2024-06-04T13:52:11.413Z",
  "content": "Сегодня разберемся с популярным фреймворком NestJS...",
  "rating": 0,
  "tags": [
    "Web",
    "NestJS",
    "TypeScript"
  ],
  "authorId": "6f5e8bf2-5b3f-410a-9b63-03253b0803c6"
}
```

Response headers

```
connection: keep-alive
content-length: 374
content-type: application/json; charset=utf-8
date: Tue, 04 Jun 2024 13:56:39 GMT
etag: W/"176-a23b04bc3352pc50kuQWf0bT6dJE"
keep-alive: timeout=5
x-powered-by: Express
```