

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:

<http://localhost:port/api> - документация api

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:

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, безопасность, легкость тестирования.

В корне проекта будут храниться:

- различные конфигурационные файлы (package.json, tsconfig.json, .env и т.д.)
- 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.

Подключение TypeScript:

Для подключения TypeScript – просто установим необходимые зависимости командой:

```
npm install --save-dev typescript @types/node @types/express.
```

После мы можем убедиться, что TypeScript установился, увидев следующую зависимость в package.json:

```

"devDependencies": {
  "@types/cors": "^2.8.17",
  "@types/swagger-jsdoc": "^6.0.4",
  "@types/uuid": "^9.0.8",
  "@types/yamljs": "^0.2.34",
  "cross-env": "^7.0.3",
  "ts-node": "^10.9.2",
  "ts-node-dev": "^2.0.0",
  "typescript": "^5.4.5"
}

```

Также можно посмотреть конфигурацию TypeScript в появившемся tsconfig.json:

```

{
  "compilerOptions": {
    "target": "ES6",
    "module": "CommonJS",
    "outDir": "./dist",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": [
    "src/**/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}

```

Подключение приложения к Express.js:

Для того, чтобы подключиться к Express.js — нужно просто в главном файле приложения main.ts — импортировать express:

```
import express from 'express';
```

И вызвать его:

```
const app = express();
```

Внедрение Dependency Injection:

Приложение будет иметь следующую логику:

Есть три группы самых главных файлов приложения:

- 1) main.ts – первая группа
- 2) app.module.ts – вторая группа
- 3) все.module.ts – третья группа

В каждом модуле приложения будет находиться файл, который будет представлять этот модуль. У данных файлов будет специальное назначение в названии – module, например:

app.module.ts, user.module.ts, post.module.ts и т.д.

Здесь:

- app.module.ts – представляет модуль app
- user.module.ts – представляет модуль user
- post.module.ts – представляет модуль post
- и т.д.

В каждом модуле (файле file.module.ts) – будут создаваться экземпляры всех классов, определенных в данном модуле. Например, если в модуле user есть user.module.ts, user.controller.ts, user.service.ts и user.routes.ts, то в user.module.ts будут создаваться экземпляры:

- Класса UserController из user.controller.ts
- Класса UserService из user.service.ts
- Класса UserRouter из user.routes.ts

Однако экземпляры классов самих модулей – экземпляр класса UserModule из user.module.ts, создаваться пока не будут. Каждый модуль будет импортирован в самый главный модуль приложения – app.module.ts. И в этом главном модуле – будут по очереди создаваться экземпляры классов модулей каждого из импортированных модулей. То есть класс UserModule импортируется в класс AppModule (app.module.ts) и там создастся его экземпляр. Соответственно, когда создастся экземпляр класса UserModule – также и создадутся экземпляры классов из модуля user: UserController, UserService, UserRouter.

И там с каждым модулем приложения. Сам же класс AppModule мы импортируем в точку входа в наше приложения – в файл main.ts. Там мы в главной функции bootstrap приложения создадим экземпляр класса AppModule и загрузим все определенные зависимости, которые в нем определены. Причем строго в том порядке, в котором зависимости создаются в классе AppModule.

Такой подход – даст нам следующие преимущества:

- Мы определяем строго в каком порядке будут создаваться зависимости. Это значит, что если нам необходимо что-то установить в приложении, чтобы этим мог пользоваться любой класс в приложении, например, нам нужно установить

dotenv.config(), чтобы потом воспользоваться переменной из .env-файла в классе UserController, то мы просто сначала в AppModule устанавливаем зависимость, которой нужно потом воспользоваться (dotenv.config()), а после установки этой зависимости – устанавливаем другую зависимость, которая будет пользоваться предыдущей (new UserController). Таким образом, гарантируется, что к моменту создания UserController - dotenv.config() уже будет определен

- Мы выносим логику создания экземпляров классов конкретного модуля в специальные места – файлы file.module.ts соответствующего модуля, а потом в нужном нам порядке загружаем их через основной модуль app.module.ts. Это централизует логику создания классов. Благодаря этому мы упрощаем разработку, делаем ее более предсказуемой, более доступной для тестирования и поддержки, а также более модульной
- Становится возможным запомнить экземпляр абсолютно любого класса в приложении, и, в последствии, внедрить экземпляр этого класса абсолютно в любое место приложения (учитывая порядок создания зависимостей). То есть при создании экземпляра класса – он будет запоминаться и дальше нам не нужно создавать новый экземпляр этого же класса, если мы его будем использовать в каком-то другом классе. Мы будем постоянно работать с одним и тем же экземпляром класса – синглтоном, без необходимости каждый раз создавать новый экземпляр. А если вдруг в каком-либо месте понадобится новый экземпляр класса – можно с легкостью внедрить фабрику

Теперь нам нужно сделать так, чтобы вся эта логика реализовывалась. Для этого создадим в директории ./src директорию dependencyInjection, где создадим два файла:

- dependency.type.ts:

```
type DependencyContainerType = {  
  [key: string]: any;  
};  
export default DependencyContainerType;
```

- dependency.container.ts:

```
class DependencyContainer {  
  private depContainer: DependencyContainerType = {};  
  
  public registerInstance(key: string, instance: any): void {  
    this.depContainer[key] = instance;  
  }  
  
  public getInstance<T>(key: string): T {  
    return this.depContainer[key] as T;  
  }  
}
```

```
}  
const dependencyContainer = new DependencyContainer();  
export default dependencyContainer;
```

В классе `DependencyContainer` создается специальный объект, который будет запоминать экземпляры классов. Мы можем получить нужный экземпляр, выполнив функцию `getInstance`, а также мы можем зарегистрировать экземпляр, выполнив функцию `registerInstance`.

Теперь мы можем добавлять все зависимости в этот контейнер, что позволит нам получать эти зависимости в любом месте, учитывая порядок их создания.

Добавление нужных модулей:

В `./src` помимо модуля `app` и `dependencyInjection` - будут находиться другие модули. Например, нам понадобятся модули `domain`, `sequelize`, `models`, `middleware` и `swagger`.
Здесь:

- Модуль `domain` будет отвечать за все модули, которые создают и взаимодействуют с API по маршрутам `/api/`
- Модуль `sequelize` будет отвечать за инициализацию ORM Sequelize в приложении, а также за подключение к базе данных через Sequelize
- Модуль `models` будет отвечать за все модели, на основе которых будут строиться таблицы в базе данных
- Модуль `middleware` будет отвечать за внедрение промежуточного ПО
- Модуль `swagger` будет отвечать за внедрение OpenApi Swagger в приложение для последующего тестирования написанного api в браузере

Модуль sequelize (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

Дальше необходимо, чтобы sequelize синхронизировал приложение к заданной базе данных. Для этого создадим:

- `sequelize.interface.config.ts` – представляет из себя интерфейс конфига подключения к базе данных, который необходимо передать sequelize для синхронизации:

```
interface ISequelizeConfig {
  dialect: Dialect;
  host: string;
  port: number;
  username: string;
  password: string;
  database: string;
}
```

То есть для того, чтобы sequelize подключил приложение к базе данных, нужно передать объект, содержащий диалект базы данных, хост, порт, имя пользователя, пароль пользователя и имя базы данных, к которой подключаемся. Все эти параметры позволят подключиться к серверу, на котором работает база данных, а также к самой базе данных. В этом интерфейсе – мы указываем как должен выглядеть конфиг для подключения к базе данных

- `sequelize.interface.service.ts` – представляет из себя интерфейс для каждого сервиса sequelize (чтобы подключение к базе данных через sequelize зависело не от конкретной реализации сервиса sequelize, а от абстракции – буква D из Solid):

```
interface ISequelizeService {
  sync(): Promise<void>;
}
```

- `sequelize.service.ts` – сервис для инициализации ORM Sequelize в приложении, а также для подключения к базе данных через Sequelize:

```
class SequelizeService implements ISequelizeService {
  private sequelize: Sequelize;

  constructor(config: ISequelizeConfig, models: ModelCtor[]) {
```



```

    this.sequelize = new Sequelize(config);
    this.sequelize.addModels(models);
  }

  public async sync(): Promise<void> {
    await this.sequelize.sync();
  }
}

```

- sequelize.module.ts – модуль, запускающий процесс инициализации ORM Sequelize и процесс подключения к базе данных + не забывать про назначение всех модулей (Dependency Injection):

```

class SequelizeModule {
  private seqService: ISequelizeService;

  constructor(
    private readonly dbConfig: ISequelizeConfig,
    private readonly models: ModelCtor[]
  ) {
    this.seqService = new SequelizeService(this.dbConfig, this.models);
    dependencyContainer.registerInstance('seqService', this.seqService);
  }

  public async onModuleInit() {
    await this.seqService.sync();
  }
}

```

Модуль models:

Всего будет 4 модели (4 таблицы в базе данных) – User, UserContact, Post, Subscription:

- 1) Модель User – представляет пользователя приложения:

```

@Table
class User 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(() => Post)  
posts!: Post[];  
  
@HasMany(() => UserContact)  
contacts!: UserContact[];  
  
@BelongsToMany(() => Subscription, {  
    through: () => Subscription,  
    foreignKey: 'userId',  
})
```

```
    otherKey: 'subscriberId'  
  })  
  subscribers!: Subscription[];  
}
```

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 (admin, user). 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 Subscription[], либо как массив Subscription[]. 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 данного контакта. 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;
}

```

```

    })
    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 Post 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.STRING,
    allowNull: false
  })
  access!: 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(() => User)
@Column({
  type: DataType.UUID
})
authorId!: string;

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

```

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
- `access` - обязательное поле, которое не может быть `null` или `undefined`.
Представляет из себя тип поста (пост может быть приватным, публичным или ограниченным (могут читать только определенные пользователи)). 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:

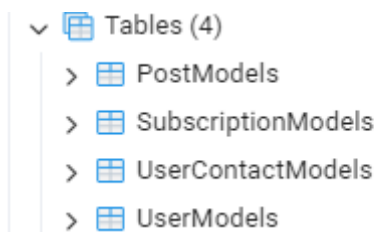
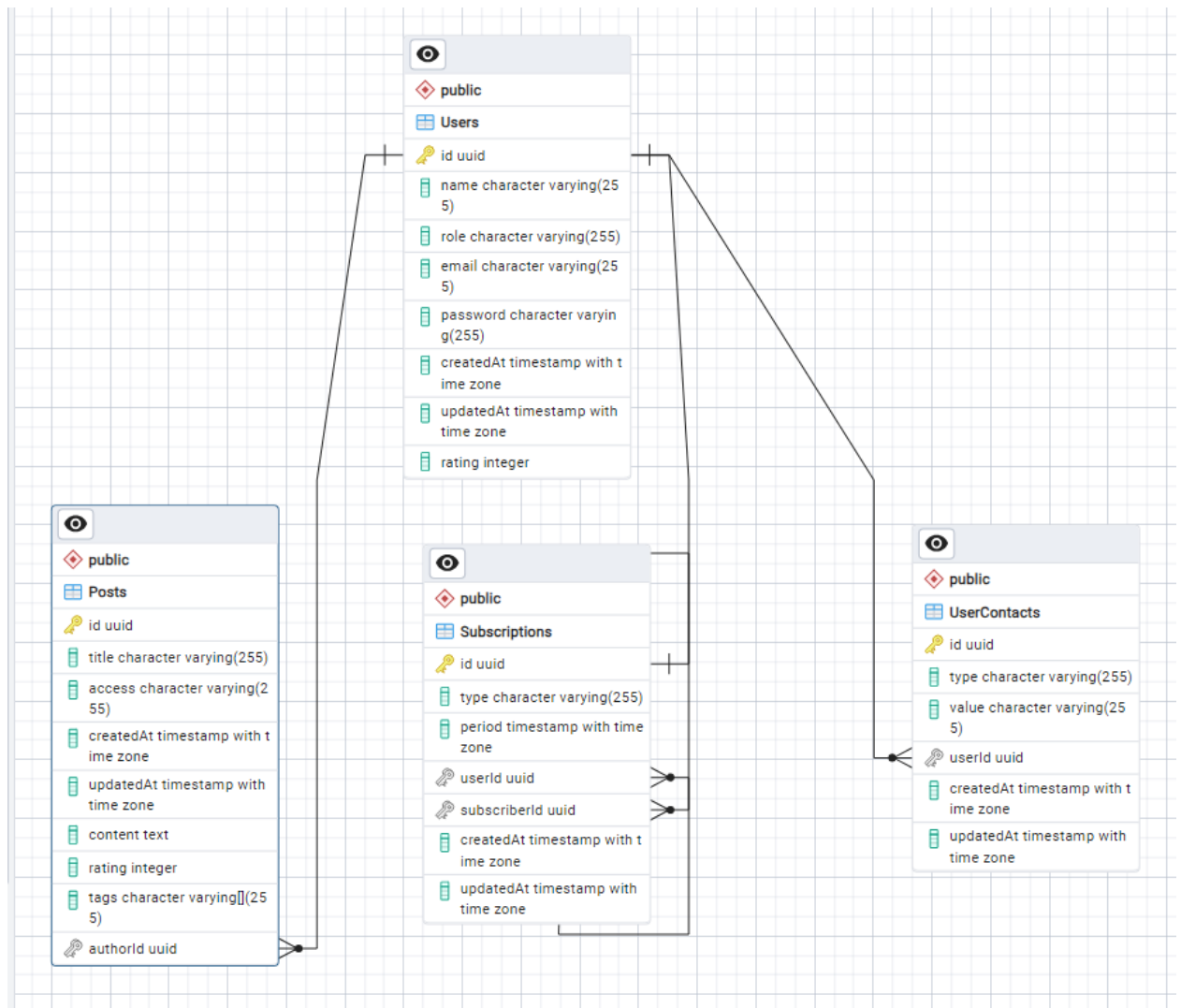


Схема в виде ERD-диаграммы (./db.png):



Подключение к локальному серверу PostgreSQL:

После определения модуля sequelize и models – стало возможным инициализировать ORM Sequelize в приложении и подключиться к локальному серверу PostgreSQL и его базе данных. Для этого импортируем в главный модуль приложения AppModule (app.module.ts) – SequelizeModule (sequelize.module.ts), а также все определенные модели, и создадим эту зависимость в методе load:

```
class AppModule {
  public async load(): Promise<void> {
    dotenv.config();

    dependencyContainer.registerInstance('seqModule', new SequelizeModule(
      {
        dialect: 'postgres',
```

```

        host: process.env.DATABASE_HOST_DEV!,
        port: Number(process.env.DATABASE_PORT_DEV!),
        username: process.env.DATABASE_USERNAME_DEV!,
        password: process.env.DATABASE_PASSWORD_DEV!,
        database: process.env.DATABASE_NAME_DEV!
    },
    [
        User,
        UserContact,
        Post,
        Subscription
    ]
));
await dependencyContainer.getInstance<SequelizeModule>('seqModule').onModuleInit();

dependencyContainer.registerInstance('appController', new AppController());
}
}

```

Здесь мы создаем зависимость SequelizeModule, передав в нее объект для подключения к серверу базы данных и к самой базе данных (параметры для подключения берем из файла .env), а также массив, содержащий все модели, которые должны создаваться в базе данных как таблицы:

```

dependencyContainer.registerInstance('seqModule', new SequelizeModule(
    {
        dialect: 'postgres',
        host: process.env.DATABASE_HOST_DEV!,
        port: Number(process.env.DATABASE_PORT_DEV!),
        username: process.env.DATABASE_USERNAME_DEV!,
        password: process.env.DATABASE_PASSWORD_DEV!,
        database: process.env.DATABASE_NAME_DEV!
    },
    [
        User,
        UserContact,
        Post,
        Subscription
    ]
));

```

Здесь мы получаем эту зависимость и вызываем у нее метод onModuleInit, который инициализирует SequelizeModule, запустив саму инициализацию ORM Sequelize и подключение к базе данных:

```

await dependencyContainer.getInstance<SequelizeModule>('seqModule').onModuleInit();

```

Модуль middleware:

На данный момент нам понадобится этот модуль, чтобы внедрить глобальный обработчик ошибок в приложении. Для это создадим подмодуль error, в котором определим errorHandler. Он будет ловить различные ошибки, возникающие при работе с api и возвращать ответ на эти ошибки с соответствующим статусом, а также логировать эти ошибки:

```
function errorHandler(err: Error, req: Request, res: Response, next: NextFunction): Response {
  let status = 500;
  let message = 'Internal Server Error';

  if (err instanceof HttpError) {
    status = err.statusCode;
    message = err.message;
  }

  console.log("ErrorHandler: ", err);

  return res.status(status).json({
    statusCode: status,
    message: message,
  });
}
```

Также зарегистрируем этот обработчик глобально в main.ts:

```
app.use(errorHandler);
```

Модуль swagger:

Создадим в этом модуле два файла:

- swagger.config.ts:

```
const specs = YAML.load(__dirname + '/swagger.yaml');
const setupSwagger = (app: express.Application): void => {
  if (!specs.servers) {
    specs.servers = [];
    specs.servers.push({
      url: process.env.CUR_URL
    });
  }

  app.use('/api', swaggerUi.serve, swaggerUi.setup(specs));
}
```

```
}
```

Данный конфиг будет предоставлять функцию, которая получает написанную swagger-документацию из файла `swagger.yaml`, а затем внедряет эту документацию, а также `SwaggerUI` в приложение

- `swagger.yaml` – этот файл содержит написанную Swagger-документацию под текущий `api`

Также нужно обязательно подключить конфиг `swagger.config.ts` в приложении (`main.ts`):

```
setupSwagger(app);
```

Модуль domain (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`):

Опишем метод `Post` для создания пользователя, а также методы `Get` для его получения. 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 задать валидацию, которая будет проходить, когда на сервер пришел запрос для создания пользователя, чтобы убедиться, что на сервер отправился правильный объект создания.

С такой валидацией – нам поможет библиотека `joi`. Создадим в `./src/domain/user` директорию `validation`. Она будет содержать две директории:

- 1) `interface` – определяет интерфейсы для объектов, которые приходят для метода `Post` создания пользователя в запросе:

- a. `user.interface.ts`:

```
interface IUser {  
  name: string;  
  role: string;  
  email: string;  
  password: string;  
  rating?: number;  
  contacts?: IUserContact[];  
};
```

Данный интерфейс задает именно тот объект, который я описывал ранее – объект, который должен передаваться для метода `Post` создания пользователя в теле запроса

- b. `user.contact.interface.ts`:

```
interface IUserContact {  
  type: string;  
  value: string;  
};
```

Данный интерфейс задает все то, что должно находиться внутри массива `contacts` объекта запроса

- 2) `schema` – определяет схемы для `joi`, на основе которых `joi` будет проводить валидацию объекта, который пришел в теле запроса при методе `Post` создания пользователя:

- a. `user.schema.ts`:

```
const UserSchema: ObjectSchema<IUser> = Joi.object({  
  name: Joi.string().min(2).max(15).required(),  
  role: Joi.string().min(4).max(5).required(),  
  email: Joi.string().email().required(),  
  password: Joi.string().pattern(new RegExp('^[a-zA-Z0-9]{8,30}$')).required(),  
  rating: Joi.number().integer().optional(),  
});
```

```
contacts: Joi.array().items(UserContactSchema).optional()
});
```

Joi будет на основании этой схемы проверять объект запроса для метода Post создания пользователя. Он проверяет все то, что я описал ранее. Например, он смотрит, чтобы в объекте обязательно было поле name, и при этом данное поле должно быть строкой, содержащей минимум 2, максимум 15 символов и т.д.

b. user.contact.schema.ts:

```
const UserContactSchema: ObjectSchema<IUserContact> = Joi.object({
  type: Joi.string().required(),
  value: Joi.string().required()
});
```

Joi на основании этой схемы будет проверять все то, что находится внутри поля contacts объекта объекта запроса для метода Post создания пользователя.

После настройки валидации – можно переходить к самому api.

Сначала напишем роутер (./src/domain/user/user.routes.ts):

```
class UserRouter {
  private readonly userRouter: Router;
  private readonly userController: UserController;

  constructor() {
    this.userRouter = express.Router();
    this.userController =
      dependencyContainer.getInstance<UserController>('userController');
    this.setupUserRouter();
  }

  public getUserRouter(): Router {
    return this.userRouter;
  }

  private setupUserRouter(): void {
    this.userRouter.get('/', (...args) => this.userController.getAllUsers(...args));
    this.userRouter.get('/:id', (...args) => this.userController.getUserById(...args));
    this.userRouter.post('/', (...args) => this.userController.createUser(...args));
  }
}
```

Данный роутер отвечает за то, чтобы создать роутер пользователей, который будет отвечать за обработку HTTP-запросов, связанных с пользователями. Этот роутер создает маршруты для следующих операций:

- a. GET / - получение всех пользователей
- b. GET /:id – получение определенного пользователя по его id
- c. POST / - создание пользователя

Для выполнения данных операций – роутер использует контроллер пользователей UserController, в котором определены функции для данных операций.

Теперь определим этот контроллер UserController (./src/domain/user.controller.ts):

```
class UserController {
  private readonly userService: UserService;

  constructor(userService: UserService) {
    this.userService = userService;
  }

  public async getAllUsers(req: Request, res: Response, next: NextFunction) {
    try {
      const searchSubstring = req.query.search || '';
      const users = await this.userService.getAllUsers(searchSubstring as string);

      if (!res.headersSent) {
        return res.status(200).json({ status: 200, data: users, message: "List of all users" });
      }
    } catch (err) {
      next(err);
    }
  }

  public async getUserById(req: Request, res: Response, next: NextFunction) {
    try {
      const id = req.params.id;
      const user = await this.userService.getUserById(id);

      if (user) {
        return res.status(200).json({ status: 200, data: user, message: "User details" });
      }
    } catch (err) {
      next(err);
    }
  }
}
```



```

public async createUser(req: Request, res: Response, next: NextFunction) {
  try {
    const userData = req.body;
    const { error } = UserSchema.validate(userData);

    if (error) {
      return res.status(422).send(`Validation error: ${error.details[0].message}`);
    }

    const newUser = await this.userService.createUser(userData);

    if (newUser) {
      return res.status(201).location(`/api/users/${newUser.id}`).json(
        { status: 201, data: newUser, message: "User successfully created" }
      );
    }
  }
  catch (err) {
    next(err);
  }
}
}

```

Рассмотрим каждую функцию:

1. `getAllUsers` – функция, которая возвращает массив всех пользователей. Возможна фильтрация через подстроку, которая является параметром URL при запросе. В данной функции сначала определяется – есть ли параметр фильтрации `searchSubstring`. Затем вызывается смежный метод `getAllUsers` из сервиса `UserService`, передавая туда `searchSubstring`. Далее если заголовки ответа еще не отправились, то отправляется ответ со статусом 200, содержащий массив всех пользователей, а также сообщение об успешности операции
2. `getUserById` – функция, которая возвращает определенного пользователя по переданному `id`. Сначала данная функция получает переданный `id`. Затем вызывается смежный метод `getUserById` из сервиса `UserService`, передавая туда этот `id`. После, если пользователь найден - отправляется ответ со статусом 200, содержащий данного пользователя, а также сообщение об успешности операции
3. `createUser` – функция, которая создает пользователя. Сначала данная функция получает переданный объект в тело запроса. Затем проводится валидация этого объекта с помощью `Joi` . Если есть какие-то ошибка валидации, то отправляется ответ со статусом 422, который говорит о том, что есть ошибка в валидации, а также какая именно ошибка. Если же

ошибки валидации нет, то вызывается смежный метод `createUser` из сервиса `UserService`, передавая туда объект запроса, прошедший валидацию. Далее если пользователя был создан, то возвращается ответ со статусом 201, заголовок ответа `Location` которого содержит ссылку на созданного пользователя, а также сообщение о том, что пользователь успешно создан

Также каждый метод контроллера обернут в `try catch`. Это нужно, что передать какую-либо ошибку, полученную в результате выполнения `api` – в глобальный обработчик ошибок `errorHandler`, определенный ранее.

Теперь рассмотрим сервис `UserService`. Данный сервис содержит логику взаимодействия с `Sequelize`. Именно через него контроллер получает пользователей:

```
class UserService {
  private readonly userAssociations = [
    { model: Post, as: 'posts' },
    { model: UserContact, as: 'contacts' },
    {
      model: Subscription,
      as: 'subscribers'
    }
  ];

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

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

```

    return users;
  }

  public async getUserById(id: string): Promise<User | never> {
    const user = await User.findOne({
      where: { id },
      include: this.userAssociations
    });
    if (!user) {
      throw new NotFound(`User with id: ${id} - is not found`);
    }

    return user;
  }

  public async createUser(userData: IUser): Promise<User | never> {
    const newUser = await User.create({
      ...userData
    });

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

      await UserContact.bulkCreate(contactsData);
    }

    return this.getUserById(newUser.id);
  }
}

```

Рассмотрим все функции:

1. `getAllUsers` – функция, которая возвращает массив всех пользователей. Сначала проверяется задана ли подстрока для фильтрации. Если не задана, то просто ищутся все пользователи через Sequelize за счет метода `findAll`. Если же фильтрация задана, то через Sequelize подстрока будет сначала искаться у всех пользователей, у которых в имени задана данная подстрока. Если таких совпадений нет, то подстрока будет искаться у всех пользователей, у которых в названии почты задана данная подстрока. Если таких совпадений нет, то выбрасывается исключение 404 о том, что пользователи не найдены по заданной подстроке. Если исключение не было брошено, то возвращается полученный массив пользователей
2. `getUserById` – функция, которая возвращает определенного пользователя по переданному `id`. Данная функция ищет пользователя по `id` через Sequelize с

помощью метода `findOne`. Если пользователь не найден, то бросается исключение 404 о том, что пользователь по такому `id` не найден. Если исключение не бросается, то найденный пользователь возвращается

3. `createUser` – функция, которая создает пользователя. Сначала данная функция создает пользователя на основе переданного объекта в `newUser` через `Sequelize` с помощью метода `create`. Однако у нас задана ассоциация `contacts` и автоматически `Sequelize` не создает связанную запись в таблице `UserContact`. Поэтому мы должны сделать это сами. Как-раз это мы и делаем далее через создание контакта в `contactsData`, а после записи контакта в таблицу `UserContact` через `bulkCreate`. Далее мы просто ищем созданного пользователя по его `id` и возвращаем его (мы та делаем, чтобы получить через метод `createUser` пользователя со всеми ассоциациями)

Также в классе `UserService` содержится данный массив:

```
private readonly userAssociations = [
  { model: Post, as: 'posts' },
  { model: UserContact, as: 'contacts' },
  {
    model: Subscription,
    as: 'subscribers'
  }
];
```

Он нужен, чтобы все методы возвращали пользователя со всеми его ассоциациями.

Теперь осталось создать `UserModule` (`user.module.ts`), который будет создавать `UserRouter`, `UserController` и `UserService` в правильном порядке:

```
class UserModule {
  private readonly userController: UserController;
  private readonly userService: UserService;

  constructor() {
    this.userService = new UserService();
    this.userController = new UserController(this.userService);
    dependencyContainer.registerInstance('userService', this.userService);
    dependencyContainer.registerInstance('userController', this.userController);
    dependencyContainer.registerInstance('userRouter', new UserRouter());
  }
}
```

Не забудем импортировать этот модуль в `app.module.ts` и создать там соответствующую зависимость:

```
class AppModule {
```

```

public async load(): Promise<void> {
  dotenv.config();

  dependencyContainer.registerInstance('seqModule', new SequelizeModule(
    {
      dialect: 'postgres',
      host: process.env.DATABASE_HOST_DEV!,
      port: Number(process.env.DATABASE_PORT_DEV!),
      username: process.env.DATABASE_USERNAME_DEV!,
      password: process.env.DATABASE_PASSWORD_DEV!,
      database: process.env.DATABASE_NAME_DEV!
    },
    [
      User,
      UserContact,
      Post,
      Subscription
    ]
  ));
  await dependencyContainer.getInstance<SequelizeModule>('seqModule').onModuleInit();

  dependencyContainer.registerInstance('appController', new AppController());
  dependencyContainer.registerInstance('appRouter', new AppRouter());
  dependencyContainer.registerInstance('userModule', new UserModule());
}
}

```

2) Посты (./src/domain/post):

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

Опишем метод Post для создания поста пользователя, а также методы Get для его получения. Для начала – нужно решить, 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. access – тип string, не может быть undefined или null. Представляет из себя доступ поста
3. content – тип string, не может быть undefined или null. Представляет из себя текст поста
4. rating – может быть undefined или null. Однако если это свойство задано – оно должно быть числом типа Number. Представляет из себя рейтинг поста

5. tags – может быть undefined или null. Однако если это свойство задано – оно должно быть массивом строк. Представляет из себя теги поста
6. authorId - тип string, не может быть undefined или null. Представляет из себя id пользователя, который создает свой пост

Чтобы это реализовать – нужно задать валидацию, которая будет проходить, когда на сервер пришел запрос для создания пользователя, чтобы убедиться, что на сервер отправился правильный объект создания.

С такой валидацией – нам поможет библиотека `joi`. Создадим в `./src/domain/post` директорию `validation`. Она будет содержать две директории:

- 1) interface – определяет интерфейсы для объектов, которые приходят для метода Post создания поста в запросе:

a. `post.interface.ts`:

```
interface IPost {  
  title: string;  
  access: string;  
  content: string;  
  rating?: number;  
  tags?: string[];  
  authorId: string;  
};
```

Данный интерфейс задает именно тот объект, который я описывал ранее – объект, который должен передаваться для метода Post создания поста в теле запроса

- 2) schema – определяет схемы для `joi`, на основе которых `joi` будет проводить валидацию объекта, который пришел в теле запроса при методе Post создания поста:

a. `post.schema.ts`:

```
const PostSchema: ObjectSchema<IPost> = Joi.object({  
  title: Joi.string().min(4).max(50).required(),  
  access: Joi.string().min(6).max(7).required(),  
  content: Joi.string().required(),  
  rating: Joi.number().integer().optional(),  
  tags: Joi.array().items(Joi.string()).optional(),  
  authorId: Joi.string().uuid().required()  
});
```

Joi будет на основании этой схемы проверять объект запроса для метода Post создания поста. Он проверяет все то, что я описал ранее

После настройки валидации – можно переходить к самому api.

Сначала напишем роутер (./src/domain/post/post.routes.ts):

```
class PostRouter {
  private readonly postRouter: Router;
  private readonly postController: PostController;

  constructor() {
    this.postRouter = express.Router();
    this.postController =
dependencyContainer.getInstance<PostController>('postController');
    this.setupPostRouter();
  }

  public getPostRouter(): Router {
    return this.postRouter;
  }

  private setupPostRouter(): void {
    this.postRouter.get('/', (...args) => this.postController.getAllPosts(...args));
    this.postRouter.get('/:id', (...args) => this.postController.getPostById(...args));
    this.postRouter.post('/', (...args) => this.postController.createPost(...args));
  }
}
```

Данный роутер отвечает за то, чтобы создать роутер постов, который будет отвечать за обработку HTTP-запросов, связанных с постами. Этот роутер создает маршруты для следующих операций:

- d. GET / - получение всех постов
- e. GET /:id – получение определенного поста по его id
- f. POST / - создание поста

Для выполнения данных операций – роутер использует контроллер постов PostController, в котором определены функции для данных операций.

Теперь определим этот контроллер PostController (./src/domain/post.controller.ts):

```
class PostController {
  constructor(private readonly postService: PostService) {}

  public async getAllPosts(req: Request, res: Response, next: NextFunction) {
    try {
      const searchSubstring = req.query.search || '';
      const posts = await this.postService.getAllPosts(searchSubstring as string);
```

```

        if (!res.headersSent) {
            return res.status(200).json({ status: 200, data: posts, message: "List of all posts" });
        }
    }
    catch (err) {
        next(err);
    }
}

public async getPostById(req: Request, res: Response, next: NextFunction) {
    try {
        const id = req.params.id;
        const post = await this.postService.getPostById(id);

        if (post) {
            return res.status(200).json({ status: 200, data: post, message: "Post details"
});
        }
    }
    catch (err) {
        next(err);
    }
}

public async createPost(req: Request, res: Response, next: NextFunction) {
    try {
        const postData = req.body;
        const { error } = PostSchema.validate(postData);

        if (error) {
            return res.status(422).send(`Validation error: ${error.details[0].message}`);
        }

        const newPost = await this.postService.createPost(postData);
        if (newPost) {
            return res.status(201).location(`/api/posts/${newPost.id}`).json(
                { status: 201, data: newPost, message: "Post successfully created" }
            );
        }
    }
    catch (err) {
        next(err);
    }
}
}

```

Рассмотрим каждую функцию:

1. `getAllPosts` – функция, которая возвращает массив всех постов. Возможна фильтрация через подстроку, которая является параметром URL при запросе. В данной функции сначала определяется – есть ли параметр фильтрации `searchSubstring`. Затем вызывается смежный метод `getAllPosts` из сервиса `PostService`, передавая туда `searchSubstring`. Далее если заголовки ответа еще не отправились, то отправляется ответ со статусом 200, содержащий массив всех постов, а также сообщение об успешности операции
2. `getPostById` – функция, которая возвращает определенный пост по переданному `id`. Сначала данная функция получает переданный `id`. Затем вызывается смежный метод `getPostById` из сервиса `PostService`, передавая туда этот `id`. После, если пост найден - отправляется ответ со статусом 200, содержащий данный пост, а также сообщение об успешности операции
3. `createPost` – функция, которая создает пост. Сначала данная функция получает переданный объект в тело запроса. Затем проводится валидация этого объекта с помощью `joí`. Если есть какие-то ошибки валидации, то отправляется ответ со статусом 422, который говорит о том, что есть ошибка в валидации, а также какая именно ошибка. Если же ошибки валидации нет, то вызывается смежный метод `createPost` из сервиса `PostService`, передавая туда объект запроса, прошедший валидацию. Далее если пост был создан, то возвращается ответ со статусом 201, заголовок ответа `Location` которого содержит ссылку на созданный пост, а также сообщение о том, что пост успешно создан

Также каждый метод контроллера обернут в `try catch`. Это нужно, что передать какую-либо ошибку, полученную в результате выполнения `api` – в глобальный обработчик ошибок `errorHandler`, определенный ранее.

Теперь рассмотрим сервис `PostService`. Данный сервис содержит логику взаимодействия с `Sequelize`. Именно через него контроллер получает посты:

```
class PostService {
  private readonly postAssociations = [
    { model: User, as: 'author' }
  ];

  constructor(private readonly userService: UserService) { }

  public async getAllPosts(searchSubstring: string): Promise<Post[] | never> {
    let posts = [];
    if (!searchSubstring) {
      posts = await Post.findAll({
        include: this.postAssociations
      });
    }
  }
}
```

```

    }
    else {
      posts = await Post.findAll({
        where: {
          [Op.or]: [
            { title: { [Op.like]: `%${searchSubstring}%` } },
            { content: { [Op.like]: `%${searchSubstring}%` } }
          ]
        },
        include: this.postAssociations
      });

      if (posts.length === 0) {
        throw new NotFound(`Posts by search substring: ${searchSubstring} - are not found`);
      }
    }

    return posts;
  }

  public async getPostById(id: string): Promise<Post | never> {
    const post = await Post.findOne({
      where: { id },
      include: this.postAssociations
    });
    if (!post) {
      throw new NotFound(`Post with id: ${id} - is not found`);
    }

    return post;
  }

  public async createPost(postData: IPost): Promise<Post | never> {
    await this.userService.getUserById(postData.authorId);

    const newPost = await Post.create({
      ...postData
    });

    return this.getPostById(newPost.id);
  }
}

```

Рассмотрим все функции:

1. `getAllPosts` – функция, которая возвращает массив всех постов. Сначала проверяется задана ли подстрока для фильтрации. Если не задана, то просто ищутся все посты через Sequelize за счет метода `findAll`. Если же

фильтрация задана, то через Sequelize подстрока будет сначала искаться у всех постов, у которых в заголовке задана данная подстрока. Если таких совпадений нет, то подстрока будет искаться у всех постов, у которых в тексте задана данная подстрока. Если таких совпадений нет, то выбрасывается исключение 404 о том, что посты не найдены по заданной подстроке. Если исключение не было брошено, то возвращается полученный массив постов

2. `getPostById` – функция, которая возвращает определенный пост по переданному `id`. Данная функция ищет пост по `id` через Sequelize с помощью метода `findOne`. Если пост не найден, то бросается исключение 404 о том, что пост по такому `id` не найден. Если исключение не бросается, то найденный пост возвращается
3. `createPost` – функция, которая создает пост. Данная функция сначала ищет пользователя по переданному в `postData` `authorId` через внедренный сервис `UserService` (`./src/domain/user/user.service.ts`). Если пользователь по такому `id` был найден, то создается соответствующий пост через Sequelize с помощью метода `create`, а после данный пост ищется через метод `getPostById` по его `id` (мы так делаем, чтобы получить через метод `createPost` пост со всеми ассоциациями) и возвращается. Если пользователь с таким `id` не найден, то бросается соответствующее исключение.

Также в классе `PostService` содержится данный массив:

```
private readonly postAssociations = [  
  { model: User, as: 'author' }  
];
```

Он нужен, чтобы все методы возвращали пост со всеми его ассоциациями.

Теперь осталось создать `PostModule` (`post.module.ts`), который будет создавать `PostRouter`, `PostController` и `PostService` в правильном порядке:

```
class PostModule {  
  private readonly postController: PostController;  
  private readonly postService: PostService;  
  
  constructor() {  
    this.postService = new  
PostService(dependencyContainer.getInstance<UserService>('userService'));  
    this.postController = new PostController(this.postService);  
    dependencyContainer.registerInstance('postService', this.postService);  
    dependencyContainer.registerInstance('postController', this.postController);  
  }  
}
```

```

    dependencyContainer.registerInstance('postRouter', new PostRouter());
  }
}

```

Не забудем импортировать этот модуль в app.module.ts и создать там соответствующую зависимость:

```

class AppModule {
  public async load(): Promise<void> {
    dotenv.config();

    dependencyContainer.registerInstance('seqModule', new SequelizeModule(
      {
        dialect: 'postgres',
        host: process.env.DATABASE_HOST_DEV!,
        port: Number(process.env.DATABASE_PORT_DEV!),
        username: process.env.DATABASE_USERNAME_DEV!,
        password: process.env.DATABASE_PASSWORD_DEV!,
        database: process.env.DATABASE_NAME_DEV!
      },
      [
        User,
        UserContact,
        Post,
        Subscription
      ]
    ));
    await dependencyContainer.getInstance<SequelizeModule>('seqModule').onModuleInit();

    dependencyContainer.registerInstance('appController', new AppController());
    dependencyContainer.registerInstance('appRouter', new AppRouter());
    dependencyContainer.registerInstance('userModule', new UserModule());
    dependencyContainer.registerInstance('postModule', new PostModule());
  }
}

```

Тесты:

Для того, чтобы протестировать написанный api – воспользуемся OpenApi спецификацией – модулем Swagger, который мы до этого установили.

Конфиг swagger – мы установили в main.ts:

```

async function bootstrap() {
  const appModule = new AppModule();
  dependencyContainer.registerInstance('appModule', new AppModule());
  await appModule.load();
}

```

```

const app = express();
const { port } = config();

process.env.CUR_URL = process.env.NODE_ENV === 'development' ?
  `${process.env.URL_DEV}:${port}` : process.env.URL_PROD;

app.use(cors({
  origin: '*',
  optionsSuccessStatus: 200
}));
app.use(express.json());
app.use('/', dependencyContainer.getInstance<AppRouter>('appRouter').getAppRouter());
app.use('/api/users',
dependencyContainer.getInstance<UserRouter>('userRouter').getUserRouter());
app.use('/api/posts',
dependencyContainer.getInstance<PostRouter>('postRouter').getPostRouter());

setupSwagger(app);

app.use(errorHandler);

app.listen(port, () => {
  console.log(`Server running on ${process.env.CUR_URL} - ${process.env.NODE_ENV}`);
});
}

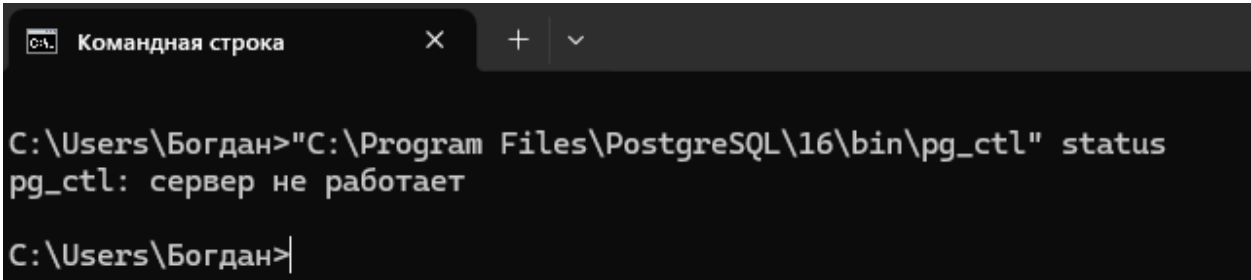
bootstrap();

```

Также мы создали саму документацию в ./src/swagger/swagger.yaml.

Теперь после запуска приложения командой `npm run start:dev` — можно 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:

```
Командная строка
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>
```

```
Командная строка
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:

a. Метод Post (Создание пользователя):

-Выберем метод Post для пользователей (/api/users Create a new user), нажмем Try it out. Передавать будем следующий объект:

```
{
  "name": "lionel",
  "role": "admin",
  "email": "lionelmail@mail.ru",
  "password": "hjkjgklJJFHF7813",
  "contacts": [
    {
      "type": "phone",
      "value": "lionel_phone"
    },
    {
      "type": "fb",
      "value": "lionel_fb"
    }
  ]
}
```

Введем этот объект:

users

GET /api/users Get all users

POST /api/users Create a new user

Parameters

No parameters

Request body required

application/json

```
{
  "name": "lionel",
  "role": "admin",
  "email": "lionel@mail.ru",
  "password": "hjkjkljJHF7813",
  "contacts": [
    {
      "type": "phone",
      "value": "lionel_phone"
    },
    {
      "type": "fb",
      "value": "lionel_fb"
    }
  ]
}
```

Далее нажмем Execute и получим результат запроса:

Responses

Curl

```
curl -X 'POST' \
'http://localhost:5000/api/users' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "name": "lionel",
  "role": "admin",
  "email": "lionel@mail.ru",
  "password": "hjkjkljJHF7813",
  "contacts": [
    {
      "type": "phone",
      "value": "lionel_phone"
    },
    {
      "type": "fb",
      "value": "lionel_fb"
    }
  ]
}'
```

Request URL

http://localhost:5000/api/users

Server response

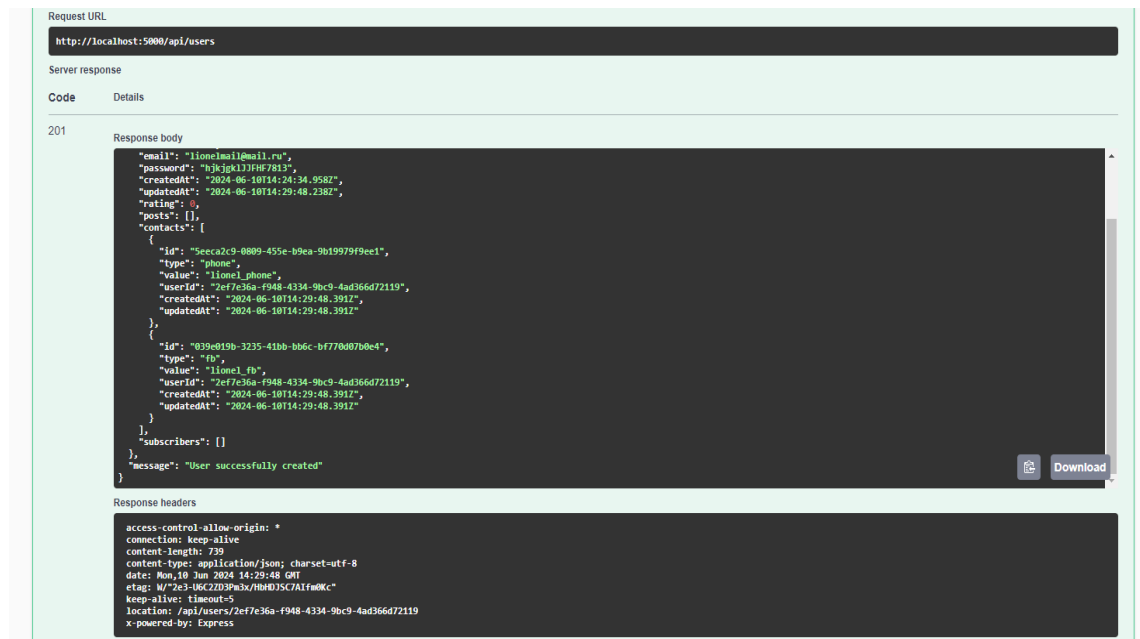
Code

Details

201

Response body

```
{
  "status": 201,
  "data": {
    "id": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
    "name": "lionel",
    "role": "admin",
    "email": "lionel@mail.ru",
    "password": "hjkjkljJHF7813",
    "createdAt": "2024-06-10T14:24:34.958Z",
    "updatedAt": "2024-06-10T14:29:48.238Z",
    "rating": 0,
    "posts": [],
    "contacts": [
      {
        "id": "5eeca2c9-0809-455c-b9ea-9b19979f9ee1",
        "type": "phone",
        "value": "lionel_phone",
        "userId": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
        "createdAt": "2024-06-10T14:29:48.391Z",
        "updatedAt": "2024-06-10T14:29:48.391Z"
      }
    ]
  }
}
```

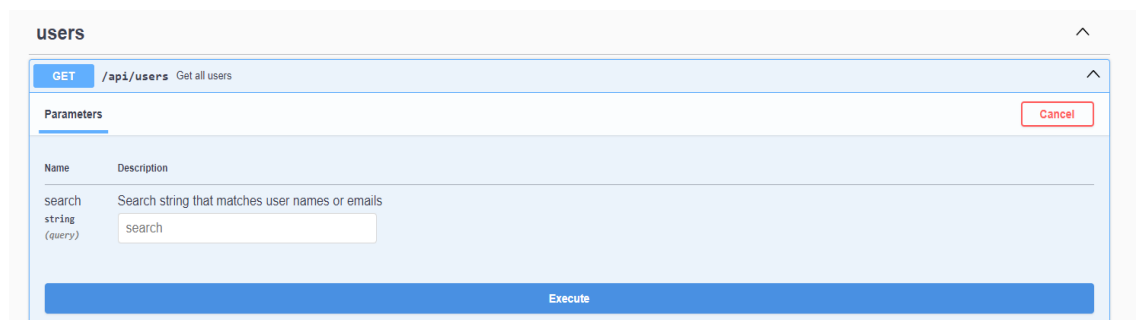


Как можно заметить – все прошло успешно, вернулся ответ 201 о том, что пользователь создан, а также сам пользователь

б. Метод Get (Получение всех пользователей):

Примечание: данный метод вернет помимо созданного пользователя на предыдущем этапе – еще других созданных пользователей, так как я до этого также тестировал это

Теперь выберем метод Get (/api/users Get all users) для получения всех пользователей. Нажмем Try it out, а после Execute и увидим результат:



Responses

Curl

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

Request URL

http://localhost:5000/api/users

Server response

Code Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
      "name": "messii",
      "role": "user",
      "email": "mess@gmail.com",
      "password": "jkggikljs81",
      "createdAt": "2024-06-10T10:21:56.664Z",
      "updatedAt": "2024-06-10T10:23:46.243Z",
      "rating": 44,
      "posts": [
        {
          "id": "fdaaeff6-78f1-4b08-a2cc-2e6c05ede91a",
          "title": "Внастрем коней - мой изгнанный",
          "access": "public",
          "createdAt": "2024-06-10T10:26:22.853Z",
          "updatedAt": "2024-06-10T10:26:34.417Z",
          "content": "Скорая поговорка об очень известном...",
          "rating": 22,
          "tags": [
            "Внастрем коней"
          ],
          "authorId": "e1078dc6-24ea-4e37-82d6-e4f929851c20"
        }
      ],
      "contacts": [
```

200

Response body

```
    {
      "contacts": [
        {
          "id": "06f85fc7-b76f-4255-9f05-1b201324a5f6",
          "type": "telegram",
          "value": "link_to_telegram",
          "userId": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
          "createdAt": "2024-06-10T10:23:46.282Z",
          "updatedAt": "2024-06-10T10:23:46.282Z"
        }
      ],
      "subscribers": []
    },
    {
      "id": "176a3570-472b-4ee3-8817-927d49f0d79e",
      "name": "some_awesome",
      "role": "user",
      "email": "user_mail@mail.ru",
      "password": "nvkdjKKK123",
      "createdAt": "2024-06-09T20:29:08.185Z",
      "updatedAt": "2024-06-09T20:30:27.466Z",
      "rating": 24,
      "posts": [],
      "contacts": [
        {
          "id": "93d6a887-e349-4e51-9c04-6c2e248fef39",
          "type": "vk",
          "value": "link_to_vk",
          "userId": "176a3570-472b-4ee3-8817-927d49f0d79e",
```

Code Details

200

Response body

```
        {
          "value": "link_to_vk",
          "userId": "176a3570-472b-4ee3-8817-927d49f0d79e",
          "createdAt": "2024-06-09T20:30:27.586Z",
          "updatedAt": "2024-06-09T20:30:27.586Z"
        }
      ],
      {
        "id": "1620041a-4871-467d-a14b-ecf11839d256",
        "type": "phone",
        "value": "8 982 408 35 71",
        "userId": "176a3570-472b-4ee3-8817-927d49f0d79e",
        "createdAt": "2024-06-09T20:30:27.586Z",
        "updatedAt": "2024-06-09T20:30:27.586Z"
      }
    ],
    "subscribers": []
  },
  {
    "id": "5c182407-0783-4c3f-b880-78bfd4aeeef83",
    "name": "bogdan_super",
    "role": "admin",
    "email": "st1035@mail.ru",
    "password": "hgjfhksh382",
    "createdAt": "2024-06-09T20:26:50.136Z",
    "updatedAt": "2024-06-09T20:28:21.070Z",
    "rating": 0,
    "posts": [
      {
        "id": "343af758-4f79-46ac-ab0a-e03a7fdbc584",
        "title": "Украинцы",
```


Code

Details

200

Response body

```
{
  "title": "Устранены",
  "access": "public",
  "createdAt": "2024-06-09T20:51:14.877Z",
  "updatedAt": "2024-06-09T20:52:14.260Z",
  "content": "Сегодня я познакомился с вами своей головой болела...",
  "rating": 0,
  "tags": [
    "бума",
    "генпеппер"
  ],
  "authorId": "5cf82407-0783-4c3f-b880-78bfd4aeef83"
},
{
  "contacts": [],
  "subscribers": []
},
{
  "id": "61bcad5f-ca1c-4f9d-89a7-54bc9a7a8098",
  "name": "eric markan",
  "role": "user",
  "email": "aevs@gmail.com",
  "password": "685h0D3f",
  "createdAt": "2024-06-09T20:36:33.851Z",
  "updatedAt": "2024-06-09T20:37:52.339Z",
  "rating": 0,
  "posts": [],
  "contacts": [
    {
      "id": "37709bbb-cf9e-4490-afe9-73d38d9a6991",
      "type": "vk",
      "value": "link_to_vk",
      "userId": "61bcad5f-ca1c-4f9d-89a7-54bc9a7a8098",
      "createdAt": "2024-06-09T20:37:52.392Z",
      "updatedAt": "2024-06-09T20:37:52.392Z"
    },
    {
      "id": "dc719d2e-367d-4a41-bb09-be8c8c627b4e",
      "type": "fb",
      "value": "link_to_fb",
      "userId": "61bcad5f-ca1c-4f9d-89a7-54bc9a7a8098",
      "createdAt": "2024-06-09T20:37:52.392Z",
      "updatedAt": "2024-06-09T20:37:52.392Z"
    }
  ],
  "subscribers": []
},
{
  "id": "2ef7c36a-f948-4334-9bc9-4ad366d72119",
  "name": "lionel",
  "role": "admin",
  "email": "lionel@mail.ru",
  "password": "h3k1pk11fHf7813",
  "createdAt": "2024-06-10T14:24:34.958Z",
  "updatedAt": "2024-06-10T14:29:48.238Z"
}
```

 Download


Code

Details

200

Response body


```
{
  "posts": [],
  "contacts": [
    {
      "id": "37709bbb-cf9e-4490-afe9-73d38d9a6991",
      "type": "vk",
      "value": "link_to_vk",
      "userId": "61bcad5f-ca1c-4f9d-89a7-54bc9a7a8098",
      "createdAt": "2024-06-09T20:37:52.392Z",
      "updatedAt": "2024-06-09T20:37:52.392Z"
    },
    {
      "id": "dc719d2e-367d-4a41-bb09-be8c8c627b4e",
      "type": "fb",
      "value": "link_to_fb",
      "userId": "61bcad5f-ca1c-4f9d-89a7-54bc9a7a8098",
      "createdAt": "2024-06-09T20:37:52.392Z",
      "updatedAt": "2024-06-09T20:37:52.392Z"
    }
  ],
  "subscribers": []
},
{
  "id": "2ef7c36a-f948-4334-9bc9-4ad366d72119",
  "name": "lionel",
  "role": "admin",
  "email": "lionel@mail.ru",
  "password": "h3k1pk11fHf7813",
  "createdAt": "2024-06-10T14:24:34.958Z",
  "updatedAt": "2024-06-10T14:29:48.238Z"
}
```

 Download

Responses

Curl

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



Request URL

```
http://localhost:5000/api/users
```

Server response


Code

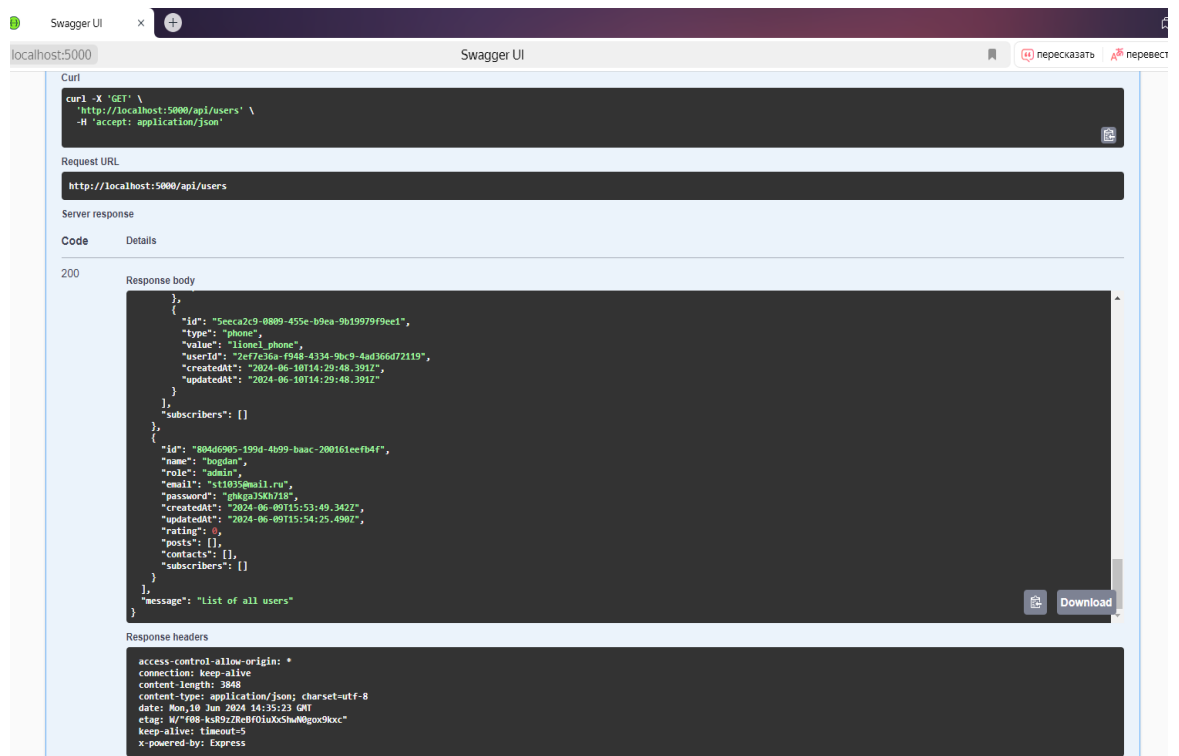
Details

200

Response body

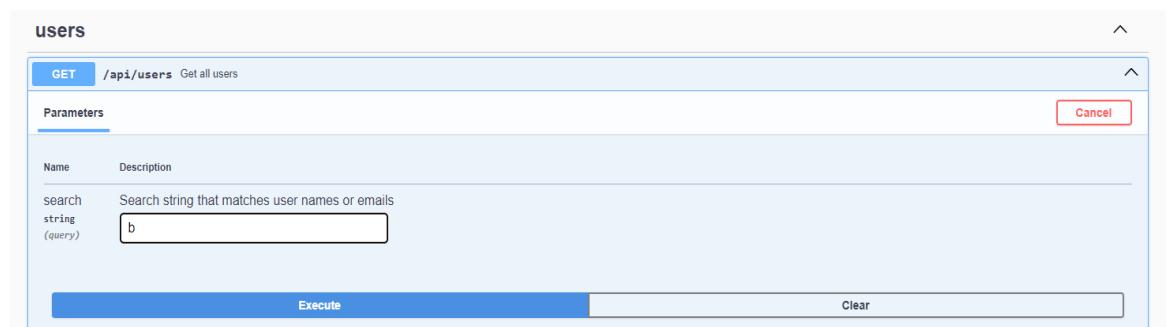
```
{
  "password": "h3k1pk11fHf7813",
  "createdAt": "2024-06-10T14:24:34.958Z",
  "updatedAt": "2024-06-10T14:29:48.238Z",
  "rating": 0,
  "posts": [],
  "contacts": [
    {
      "id": "039e019b-3235-41bb-bb6c-bf770d07b0e4",
      "type": "fb",
      "value": "lionel_fb",
      "userId": "2ef7c36a-f948-4334-9bc9-4ad366d72119",
      "createdAt": "2024-06-10T14:29:48.391Z",
      "updatedAt": "2024-06-10T14:29:48.391Z"
    },
    {
      "id": "5ee2a2c0-0809-455e-b9ea-9b19979f9ee1",
      "type": "phone",
      "value": "lionel_phone",
      "userId": "2ef7c36a-f948-4334-9bc9-4ad366d72119",
      "createdAt": "2024-06-10T14:29:48.391Z",
      "updatedAt": "2024-06-10T14:29:48.391Z"
    }
  ],
  "subscribers": []
},
{
  "id": "804d6905-199d-4b99-baac-200161eebf4f",
  "name": "bogdan",
  "role": "admin",
  "password": "h3k1pk11fHf7813",
  "createdAt": "2024-06-10T14:24:34.958Z",
  "updatedAt": "2024-06-10T14:29:48.238Z"
}
```

 Download



-Как можно заметить - все прошло успешно, вернулся ответ 200, содержащий созданного нами пользователя, а также всех созданных до этого пользователей.

-Теперь попробуем применить фильтр – найти всех пользователей, у которых в имени есть буква b. Для этого введем в подстроку b и нажмем Execute:



Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/users?search=b' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/users?search=b

Server response

Code Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "5ef82407-0783-4e3f-b880-78bfd4aef83",
      "name": "bogdan_super",
      "role": "admin",
      "email": "st1035@mail.ru",
      "password": "hgjfhksh382",
      "createdAt": "2024-06-09T20:26:50.136Z",
      "updatedAt": "2024-06-09T20:28:21.070Z",
      "rating": 0,
      "posts": [
        {
          "id": "343af758-4f79-46ac-ab0a-e83a7fdb584",
          "title": "Усталость",
          "access": "public",
          "createdAt": "2024-06-09T20:51:14.877Z",
          "updatedAt": "2024-06-09T20:52:24.250Z",
          "content": "Сегодня я поделись с вами своей головной болью...",
          "rating": 0,
          "tags": [
            "боль",
            "депрессия"
          ],
          "authorId": "5ef82407-0783-4e3f-b880-78bfd4aef83"
        }
      ]
    }
  ],
}
```

Download

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/users?search=b' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/users?search=b

Server response

Code Details

200

Response body

```
{
  "content": "Сегодня я поделись с вами своей головной болью...",
  "rating": 0,
  "tags": [
    "боль",
    "депрессия"
  ],
  "authorId": "5ef82407-0783-4e3f-b880-78bfd4aef83"
},
{
  "contacts": [],
  "subscribers": []
},
{
  "id": "804d6905-199d-4b99-baac-200161eebf4f",
  "name": "bogdan",
  "role": "admin",
  "email": "st1035@mail.ru",
  "password": "ghkgaj5kh718",
  "createdAt": "2024-06-09T15:53:49.342Z",
  "updatedAt": "2024-06-09T15:54:25.490Z",
  "rating": 0,
  "posts": [],
  "contacts": [],
  "subscribers": []
}
},
"message": "List of all users"
}
```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 950
content-type: application/json; charset=utf-8
date: Mon, 18 Jun 2024 14:44:13 GMT
etag: W/"3b6-jv2iYMLQ1tr3qth9T2EgcmYo"
keep-alive: timeout=5
x-powered-by: Express
```

Download

-Как можно заметить – вернулся ответ со статусом 200, содержащий всех пользователей, у которых в имени есть буква b. Значит все правильно. Также можно фильтровать и с другой подстрокой, и можно фильтровать не только по имени, но и по почте

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

Получим созданного нами пользователя. Для этого выберем метод Get (/api/users/{id} Get a user by id), нажмем Try it out и введем в поле User id – id созданного нами пользователя, а дальше нажмем Execute:

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

Parameters

Name

Description

id * required

User id

string(\$uuid)

(path)

2ef7e36a-f948-4334-9bc9-4ad366d72119

Execute

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/users/2ef7e36a-f948-4334-9bc9-4ad366d72119' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/users/2ef7e36a-f948-4334-9bc9-4ad366d72119
```

Server response

Code

Details

200

Response body

```
{
  "status": 200,
  "data": {
    "id": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
    "name": "lionel",
    "role": "admin",
    "email": "lionel@mail.ru",
    "password": "hjkjgkljFHF7813",
    "createdAt": "2024-06-10T14:24:34.958Z",
    "updatedAt": "2024-06-10T14:29:48.238Z",
    "rating": 0,
    "posts": [],
    "contacts": [
      {
        "id": "5eca2c9-0809-455e-b9ea-9b19979f9ee1",
        "type": "phone",
        "value": "lionel_phone",
        "userId": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
        "createdAt": "2024-06-10T14:29:48.391Z",
        "updatedAt": "2024-06-10T14:29:48.391Z"
      },
      {
        "id": "039e019b-3235-41bb-bb6c-bf770d07b0e4",
        "type": "fb",
        "value": "lionel_fb",
        "userId": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
        "createdAt": "2024-06-10T14:29:48.391Z",
        "updatedAt": "2024-06-10T14:29:48.391Z"
      }
    ]
  }
}
```

Responses

Curl

```
curl -X 'GET' \
  http://localhost:5000/api/users/2ef7e36a-f948-4334-9bc9-4ad366d72119' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/users/2ef7e36a-f948-4334-9bc9-4ad366d72119

Server response

Code Details

200

Response body

```
{
  "email": "lionelmail@mail.ru",
  "password": "hjkjgklj3FH7813",
  "createdAt": "2024-06-10T14:24:34.958Z",
  "updatedAt": "2024-06-10T14:29:48.238Z",
  "rating": 0,
  "posts": [],
  "contacts": [
    {
      "id": "5ee2a2c9-0809-455e-b9ea-9b19979f9ee1",
      "type": "phone",
      "value": "lionel_phone",
      "userId": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
      "createdAt": "2024-06-10T14:29:48.391Z",
      "updatedAt": "2024-06-10T14:29:48.391Z"
    },
    {
      "id": "039ed19b-3235-41bb-bb6c-bf770d07b0e4",
      "type": "fb",
      "value": "lionel_fb",
      "userId": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
      "createdAt": "2024-06-10T14:29:48.391Z",
      "updatedAt": "2024-06-10T14:29:48.391Z"
    }
  ],
  "subscribers": [],
  "message": "User details"
}
```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 726
content-type: application/json; charset=utf-8
date: Mon, 10 Jun 2024 14:49:41 GMT
etag: W/"2d6-+U8u0izxeyxnc+G+f8Q0zUWbQ"
keep-alive: timeout=5
x-powered-by: Express
```

Как можно заметить – вернулся ответ со статусом 200, содержащий созданного нами пользователя. Значит – все хорошо

2) Посты:

a. Метод Post (Создание поста):

-Выберем метод Post для постов (/api/posts Create a new post), нажмем Try it out. Создадим пост пользователю, которого мы создали до этого. Передавать будем следующий объект:

```
{
  "title": "Хороши ли новые Звездные войны",
  "access": "public",
  "content": "Я вчера ходил в кино на новый фильм Звездные войны. Я хочу поделиться своими мыслями...",
  "tags": [
    "кино",
    "отдых",
    "Звездные войны"
  ],
  "authorId": "2ef7e36a-f948-4334-9bc9-4ad366d72119"
}
```

Введем этот объект:

posts

GET /api/posts Get all posts

POST /api/posts Create a new post

Parameters

No parameters

Request body required

application/json

```
{
  "title": "Хороши ли новые Звездные войны",
  "access": "public",
  "content": "Я вчера ходил в кино на новый фильм Звездные войны. Я хочу поделиться своими мыслями...",
  "tags": [
    "кино",
    "отзыв",
    "Звездные войны"
  ],
  "authorId": "2ef7e36a-f948-4334-9bc9-4ad366d72119"
}
```

Execute

Далее нажмем Execute и получим результат запроса:

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/posts' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "Хороши ли новые Звездные войны",
    "access": "public",
    "content": "Я вчера ходил в кино на новый фильм Звездные войны. Я хочу поделиться своими мыслями...",
    "tags": [
      "кино",
      "отзыв",
      "Звездные войны"
    ],
    "authorId": "2ef7e36a-f948-4334-9bc9-4ad366d72119"
  }'
```

Request URL

```
http://localhost:5000/api/posts
```

Server response

Code

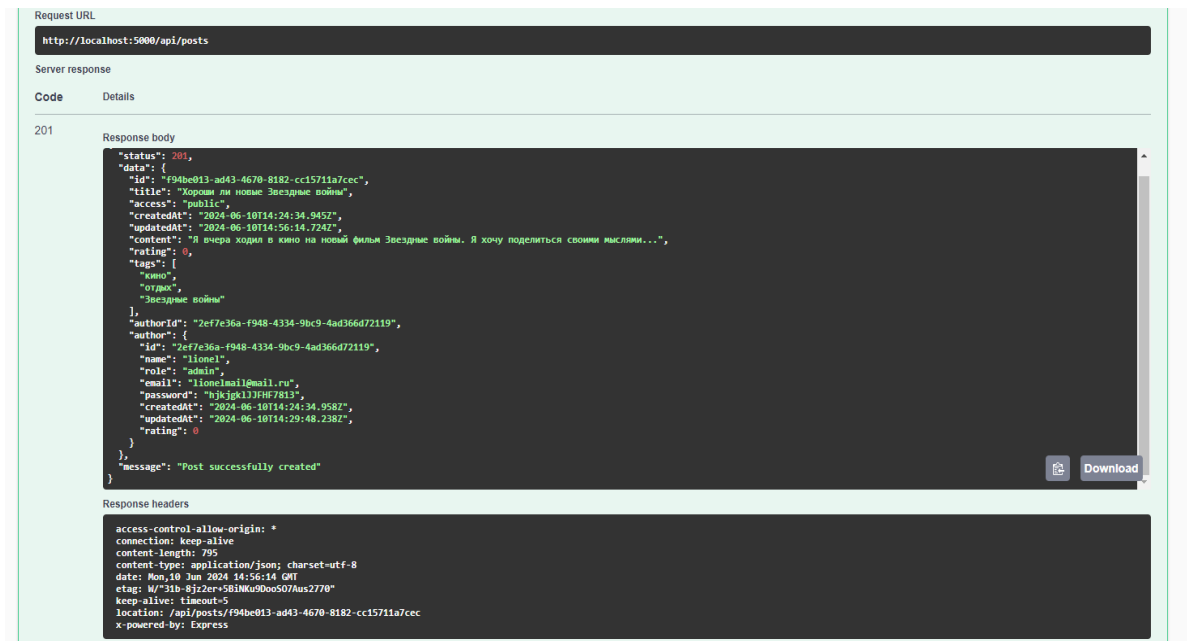
Details

201

Response body

```
{
  "status": 201,
  "data": {
    "id": "f94be013-ad43-4670-8182-cc15711a7cec",
    "title": "Хороши ли новые Звездные войны",
    "access": "public",
    "createdAt": "2024-06-10T14:24:34.945Z",
    "updatedAt": "2024-06-10T14:56:14.724Z",
    "content": "Я вчера ходил в кино на новый фильм Звездные войны. Я хочу поделиться своими мыслями...",
    "rating": 0,
    "tags": [
      "кино",
      "отзыв",
      "Звездные войны"
    ],
    "authorId": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
    "author": {
      "id": "2ef7e36a-f948-4334-9bc9-4ad366d72119",
      "name": "lionel",
      "role": "admin",
      "email": "lionel@mail.ru",
      "password": "hjkjgkljFH7813",
      "createdAt": "2024-06-10T14:24:34.958Z",
      "updatedAt": "2024-06-10T14:29:48.238Z",
      "rating": 0
    }
  },
  "message": "Post successfully created"
}
```

Download

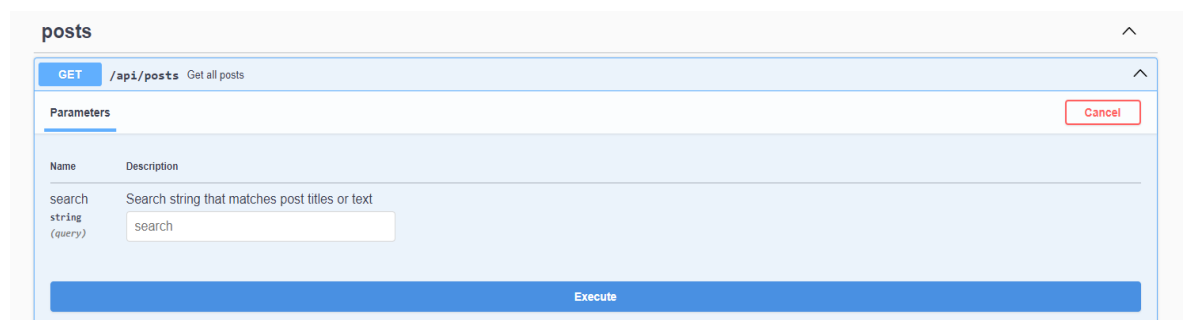


Как можно заметить – все прошло успешно, вернулся ответ 201 о том, что пост пользователя создан, а также сам пост

б. Метод Get (Получение всех постов):

!Примечание: данный метод вернет помимо созданного поста на предыдущем этапе – еще другие созданные посты, так как я до этого также тестировал это

Теперь выберем метод Get (/api/posts Get all posts) для получения всех постов. Нажмем Try it out, а после Execute и увидим результат:



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

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "343af758-4f79-46ac-ab0a-e03a7fdc584",
      "title": "Успехи",
      "access": "public",
      "createdAt": "2024-06-09T20:51:14.877Z",
      "updatedAt": "2024-06-09T20:52:24.250Z",
      "content": "Сегодня я познакомился с вами своей родной болью...",
      "rating": 0,
      "tags": [
        "бонус",
        "специал"
      ],
      "authorId": "5ef82407-0783-4e3f-b880-78bf04aef83",
      "author": {
        "id": "5ef82407-0783-4e3f-b880-78bf04aef83",
        "name": "bogdan super",
        "role": "admin",
        "email": "st1035@mail.ru",
        "password": "hgjfdmsh32f",
        "createdAt": "2024-06-09T20:26:50.136Z",
        "updatedAt": "2024-06-09T20:28:21.070Z",
        "rating": 0
      }
    }
  ]
}
```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 2011
content-type: application/json; charset=utf-8
date: Mon, 10 Jun 2024 14:59:06 GMT
etag: W/"7db-36c1gh0ICAF7/u06Cc7xL3p24g"
keep-alive: timeout=5
x-powered-by: Express
```

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

Response body

```
{
  "id": "1daef45-78f1-4b08-a2cc-2efc05ede91a",
  "title": "Всплески эмоций - мой мир",
  "access": "public",
  "createdAt": "2024-06-10T10:26:22.853Z",
  "updatedAt": "2024-06-10T10:26:34.417Z",
  "content": "Сегодня поговорю об очень известном...",
  "rating": 22,
  "tags": [
    "Всплески эмоций"
  ],
  "authorId": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
  "author": {
    "id": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
    "name": "messi",
    "role": "user",
    "email": "mcs@gmail.com",
    "password": "jhsjkl1234",
    "createdAt": "2024-06-10T10:21:56.664Z",
    "updatedAt": "2024-06-10T10:23:46.243Z",
    "rating": 44
  }
},
{
  "id": "f94be013-ad43-4670-8182-cc15711a7cec",
  "title": "Хорошо ли мне безумие войны",
  "access": "public",
  "createdAt": "2024-06-10T14:24:34.945Z",

```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 2011
content-type: application/json; charset=utf-8
date: Mon, 10 Jun 2024 14:59:06 GMT
etag: W/"7db-36c1gh0ICAF7/u06Cc7xL3p24g"
keep-alive: timeout=5
x-powered-by: Express
```

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	<p>Response body</p> <pre>{ "id": "f94be013-ad43-4670-8182-cc15711a7cec", "title": "Хороши ли новые Звездные войны", "access": "public", "createdAt": "2024-06-10T14:24:34.945Z", "updatedAt": "2024-06-10T14:56:14.724Z", "content": "Я вчера ходил в кино на новый фильм Звездные войны. Я хочу поделиться своими мыслями...", "rating": 0, "tags": ["кино", "отзывы", "Звездные войны"], "authorId": "2ef7e36a-f948-4334-9bc9-4ad366d72119", "author": { "id": "2ef7e36a-f948-4334-9bc9-4ad366d72119", "name": "Lionel", "role": "admin", "email": "lionel@mail.ru", "password": "hjkjgklj3fHf7813", "createdAt": "2024-06-10T14:24:34.958Z", "updatedAt": "2024-06-10T14:23:48.238Z", "rating": 0 } }, { "message": "List of all posts" }</pre> <p>Response headers</p> <pre>access-control-allow-origin: * connection: keep-alive content-length: 2011 content-type: application/json; charset=utf-8 date: Mon, 10 Jun 2024 14:59:06 GMT etag: W/"7db-36a1g1MOICAF1/w6Cc7xL3pZ4g" keep-alive: timeout=5 x-powered-by: Express</pre>

-Как можно заметить - все прошло успешно, вернулся ответ 200, содержащий созданный нами пост, а также все созданные до этого посты.

-Теперь попробуем применить фильтр – найти все посты, у которых в заголовке есть подстрока “мой”. Для этого введем в подстроку “мой” и нажмем Execute:

posts

GET /api/posts Get all posts

Parameters

Name	Description
search string (query)	Search string that matches post titles or text

MOY

Execute Clear

Responses

Curl

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

Request URL

```
http://localhost:5000/api/posts?search=МОНБСМОНБСМОНБС
```

Server response

CodeDetails

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "1daecf45-78f1-4b08-a2cc-2e6c05ede91a",
      "title": "Внастремм конец - мой взррррр",
      "access": "public",
      "createdAt": "2024-06-10T10:26:22.853Z",
      "updatedAt": "2024-06-10T10:26:34.417Z",
      "content": "Сероия поговорим об очень известном...",
      "rating": 22,
      "tags": [
        "Внастремм конец"
      ],
      "authorId": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
      "author": {
        "id": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
        "name": "messil",
        "role": "user",
        "email": "mes@gmail.com",
        "password": "jkgzjkljjs8t",
        "createdAt": "2024-06-10T10:21:56.664Z",
        "updatedAt": "2024-06-10T10:23:46.243Z",
        "rating": 44
      }
    }
  ],
  "message": "List of all posts"
}
```

Download

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "1daecf45-78f1-4b08-a2cc-2e6c05ede91a",
      "title": "Внастремм конец - мой взррррр",
      "access": "public",
      "createdAt": "2024-06-10T10:26:22.853Z",
      "updatedAt": "2024-06-10T10:26:34.417Z",
      "content": "Сероия поговорим об очень известном...",
      "rating": 22,
      "tags": [
        "Внастремм конец"
      ],
      "authorId": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
      "author": {
        "id": "e1078dc6-24ea-4e37-82d6-e4f929851c20",
        "name": "messil",
        "role": "user",
        "email": "mes@gmail.com",
        "password": "jkgzjkljjs8t",
        "createdAt": "2024-06-10T10:21:56.664Z",
        "updatedAt": "2024-06-10T10:23:46.243Z",
        "rating": 44
      }
    }
  ],
  "message": "List of all posts"
}
```

Download

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 669
content-type: application/json; charset=utf-8
date: Mon, 10 Jun 2024 15:03:04 GMT
etag: W/"29d-D2Iu6IueQhucQ9M034aCoGJ8Q"
keep-alive: timeout=5
x-powered-by: Express
```

-Как можно заметить – вернулся ответ со статусом 200, содержащий все посты, у которых в заголовке есть подстрока “мой”. Значит все правильно. Также можно фильтровать и с другой подстрокой, и можно фильтровать не только по заголовку, но и по тексту

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

Получим созданный нами пост. Для этого выберем метод Get (/api/posts/{id} Get a post by id), нажмем Try it out и введем в поле Post id – id созданного нами поста, а дальше нажмем Execute:

posts

GET /api/posts Get all posts

POST /api/posts Create a new post

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

Parameters

Cancel

Name	Description
id * required string(\$uuid) (path)	Post id

f94be013-ad43-4670-8182-cc15711a7cec

Execute

Responses

Curl

```
curl -X 'GET' \  
  'http://localhost:5000/api/posts/f94be013-ad43-4670-8182-cc15711a7cec' \  
  -H 'accept: application/json'
```

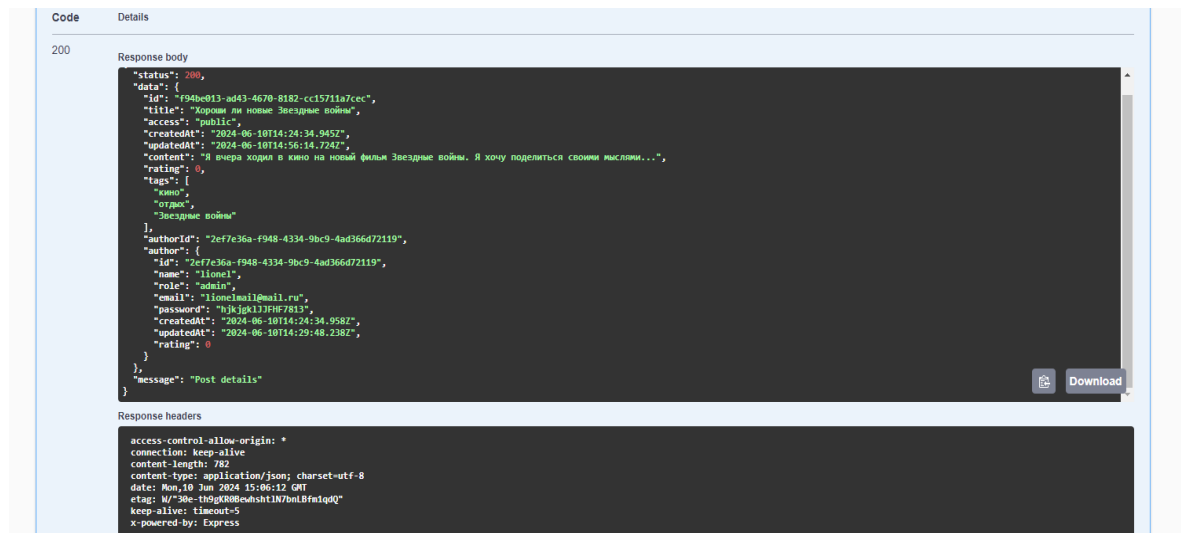
Request URL

http://localhost:5000/api/posts/f94be013-ad43-4670-8182-cc15711a7cec

Server response

Code	Details
200	<p>Response body</p> <pre>{ "status": 200, "data": { "id": "f94be013-ad43-4670-8182-cc15711a7cec", "title": "Хороши ли новые Воздушные войны", "access": "public", "createdAt": "2024-06-10T14:24:34.945Z", "updatedAt": "2024-06-10T14:56:14.724Z", "content": "Я вчера ходил в кино на новый фильм Воздушные войны. Я хочу поделиться своими мыслями...", "rating": 0, "tags": ["кино", "отдых", "Воздушные войны"] }, "authorId": "2ef7e36a-f948-4334-9bc9-4ad366d72119", "author": { "id": "2ef7e36a-f948-4334-9bc9-4ad366d72119", "name": "lionel", "role": "admin", "email": "lionel@mail.ru", "password": "hjkjgkl3jfh7813", "createdAt": "2024-06-10T14:24:34.958Z", "updatedAt": "2024-06-10T14:29:48.238Z", "rating": 0 } }, "message": "Post details" }</pre> <p>Response headers</p> <pre>access-control-allow-origin: * connection: keep-alive content-length: 782 content-type: application/json; charset=utf-8 date: Mon, 10 Jun 2024 15:06:12 GMT etag: W/"30e-thpX08mdshst1K7mLBfm1qdQ" keep-alive: timeout=5 x-powered-by: Express</pre>

Download



Как можно заметить – вернулся ответ со статусом 200, содержащий созданный нами пост. Значит – все хорошо

Также все эти методы корректно обрабатывают различные ошибки, например, валидации, поиска и т.д., и затем корректно возвращает их как ответ.