

**Задание №3**

**Стажировка**

«Веб-приложение для публикации постов - авторизация»

Выполнил: Ноздряков Богдан Валериевич

Проверил: Пармузин Александр Игоревич

Санкт-Петербург

2024

## Условие:

Реализовать:

1. API регистрации(где на почту приходит сообщение) + API verify(при перехождении по ссылке аккаунт пользователя активируется) – с помощью nodemailer
2. API логина, после которого генерится сессионная кука, с которой пользователь может иметь доступ до создания постов(без логина все методы сервиса недоступны)

Реализация должна быть выполнена без JWT.

Также требуется внедрить логику, когда при старте приложения – проверяется существует ли указанная база данных. Если существует, то нужно подключиться к ней. Если не существует, то нужно ее создать, а после подключиться к ней.

## Решение:

### Проверка наличия базы данных:

Приложение работает через локальный PostgreSQL сервер. На данный момент – приложение может подключаться только к уже существующей базе данных на сервере. Приложение не может создавать саму базу данных в случае ее отсутствия.

Исправим это. Сделаем так, чтобы при запуске приложения – сначала проверялось есть ли на сервере база данных. Если она есть, то просто происходило подключение к этой базе данных. Если же ее нет, то сначала создавалась указанная база данных, а уже после – происходило подключение к ней.

Для этого – перейдем в модуль sequelize (./src/sequelize). Нужно просто немного изменить SequelizeModule (./src/sequelize/sequelize.module.ts):

```
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.createDbIfNotExists();
    await this.seqService.sync();
  }

  public async createDbIfNotExists(): Promise<void> {
    const client = new Client({
      ...this.dbConfig,
      user: this.dbConfig.username,
```

```

        database: this.dbConfig.dialect
    });

    await client.connect();

    const isDbExist = await client.query(
        `SELECT datname FROM pg_catalog.pg_database WHERE datname =
        '${this.dbConfig.database}'`
    );

    if (isDbExist.rows.length === 0) {
        await client.query(`CREATE DATABASE ${this.dbConfig.database}`);
    }

    await client.end();
}
}

```

Мы добавили функцию createDbIfNotExists в класс модуля. Эта функция как-раз и отвечает за логику, которую необходимо внедрить. Эта же функция вызывается в методе инициализации sequelize – onModuleInit, перед синхронизацией с базой данных.

Разберем метод createDbIfNotExists:

Данный метод будет проверять наличие базы данных, а также создавать ее, в случае необходимости, за счет библиотеки pg, которая отвечает за работу с PostgreSQL базами данных.

Сначала мы создаем экземпляр клиента pg с помощью полученной конфигурации базы данных:

```

const client = new Client({
    ...this.dbConfig,
    user: this.dbConfig.username,
    database: this.dbConfig.dialect
});

```

Дальше мы подключаемся к серверу базы данных, чтобы можно было проверить наличие базы данных:

```

await client.connect();

```

Дальше мы пишем SQL-запрос через клиента pg к системной таблице pg\_database для проверки наличия базы данных с указанным именем:

```

const isDbExist = await client.query(
    `SELECT datname FROM pg_catalog.pg_database WHERE datname =
    '${this.dbConfig.database}'`
);

```

Дальше мы просто проверяем результат запроса – и если указанной базы данных не существует, то пишем SQL-запрос через клиента pg, который будет создавать эту базу данных:

```
if (isDbExist.rows.length === 0) {  
    await client.query(`CREATE DATABASE ${this.dbConfig.database}`);  
}
```

В конце закрываем соединение:

```
await client.end();
```

Дальше sequelize сможет синхронизироваться с базой данных, даже если она не существует.

### **Авторизация:**

Перед тем, как внедрять авторизацию в приложение – внедрим специальные модули-помощники. В нашем приложении авторизация реализована с помощью подхода: почта-пароль. Значит нам нужно обезопасить эти два компонента. Сделаем это, внедрив следующее:

- Модуль mailer
- Модуль crypto

Разберемся с каждым по порядку:

#### 1) Модуль mailer (./src/mailer):

Пользователь при регистрации должен указать свою почту. Соответственно нам нужно перед тем, как регистрировать пользователя – проверить, что он ввел свою почту. Для этого воспользуемся следующим подходом:

Когда пользователь отправит свою почту и пароль – ему на почту отправится письмо со ссылкой, перейдя по которой у пользователя активируется аккаунт и он сможет полноценно пользоваться приложением. Чтобы отправить ему письмо на почту – потребуется использовать сервис, который позволяет отправлять письма на почту. Мы для этого воспользуемся библиотекой nodemailer.

Создадим MailerTransporter (./src/mailer/mailer.transporter.ts):

```
class MailerTransporter {  
    private readonly mailerTransporter: Transporter;  
    private readonly transporterConfig: SMTPTransport.Options;  
  
    constructor(transporterConfig: SMTPTransport.Options) {  
        this.transporterConfig = transporterConfig;  
        this.mailerTransporter = nodemailer.createTransport(this.transporterConfig);  
    }  
}
```

```

public async sendMailByTransporter(mailOptions: SendMailOptions): Promise<void> {
    await this.mailerTransporter.sendMail(mailOptions);
}

public getMailerTransporter(): Transporter {
    return this.mailerTransporter;
}
}

```

Данный класс будет создавать транспортный объект nodemailer – mailerTransporter в конструкторе класса через специальный конфиг transporterConfig, переданный в конструктор. Транспортный объект – это то, через что будут отправляться сообщения на различные email. Грубо говоря – это наша почта, с которой мы отсылаем все письма в нашем приложении. Конфиг transporterConfig будет содержать данные, которые необходимы для создания транспортного объекта, например, почту, с которой будут отправляться письма и пароль от почты.

Также этот класс позволяет получить созданный транспортный объект через метод getMailerTransporter.

Также этот класс позволяет отправить письмо на почту через созданный транспортный объект в методе sendMailByTransporter, который принимает специальный конфиг mailOptions. Данный конфиг mailOptions будет содержать почту, на которую нужно отправить письмо, тему письма, а также текст письма.

За счет этого класса – мы сможем отправить письмо на почту, которую указал пользователь при регистрации для верификации этой почты.

Также реализуем класс MailerModule (./src/mailer/mailer.module.ts):

```

class MailerModule {
    constructor(transporterConfig: SMTPTransport.Options) {
        dependencyContainer.registerInstance('mailerTransporter', new
        MailerTransporter(transporterConfig));
    }
}

```

Данный класс в своем конструкторе создает экземпляр класса MailerTransporter, передав в него полученный конфиг для создания транспортного объекта (не забываем создать зависимость, так как в приложении реализован Dependency Injection).

Теперь нам нужно просто создать этот модуль в самом главном модуле приложения app.module.ts (AppModule), передав в него конфигурацию для создания транспортного объекта:

```

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('mailerModule', new MailerModule({
  host: 'localhost',
  service: process.env.EMAIL_SERVICE_DEV,
  auth: {
    user: process.env.EMAIL_USERNAME_DEV,
    pass: process.env.EMAIL_PASSWORD_DEV
  },
  tls: {
    rejectUnauthorized: false
  }
})));
dependencyContainer.registerInstance('userModule', new UserModule());
dependencyContainer.registerInstance('postModule', new PostModule());
}
}

```

Здесь мы создаем экземпляр MailerModule и передаем в него конфиг для создания транспортного объекта:

- host – хост, к которому будет осуществляться подключение для отправки электронных писем. В данном случае мы указываем localhost, что означает, что почтовый сервер находится на том же компьютере, что и приложение
- service – тип почтового сервиса, через который будут отправляться письма. В данном случае, мы берем значение из переменной окружения EMAIL\_SERVICE\_DEV, куда мы укажем 'gmail'
- auth – объект с данными для аутентификации на почтовом сервисе:

1. user – почта, с которой будут отправлять письма (учетная запись). В данном случае – берем значение из переменной окружения EMAIL\_USERNAME\_DEV
  2. pass – пароль от учетной записи. В данном случае – берем значение из переменной окружения EMAIL\_PASSWORD\_DEV
- tls – настройка безопасности подключения:
    1. rejectUnauthorized: false – SSL/TLS подключения будут принимать даже если сертификаты не могут быть верифицированы (это нужно указать, так как наше приложение работает пока что локально через http)

Данный конфиг использует настройки, которые подходят для локального тестирования. Но такой конфиг не подходит под production-версию проекта. Так как на данном этапе – мы разрабатываем наше приложение, то есть работаем локально, то мы будем пока что использовать данный конфиг. В будущем, при переходе на production-версию – данный конфиг будет изменен.

## 2) Модуль crypto (./src/crypto):

При регистрации – пользователь также будет вводить свой пароль. Это значит, что в базе данных будет храниться пароль пользователя. Для того, чтобы обезопасить хранение пароля – введем логику, при которой пароль будет хэширован перед записью в базу данных. Это значит, что в базе данных будет храниться пароль пользователя в хэшированном виде, а также при всех запросах получения пользователя – пароль будет предоставлен именно в хэшированном виде. Особенность хэширования будет заключаться в том, что пароль можно будет захэшировать, но при этом нельзя никак расхэшировать обратно. Данный механизм значительно повышает безопасность хранения пароля.

Для данной реализации – создадим модуль crypto. Реализуем в нем класс CryptoProvider (./src/crypto/crypto.provider.ts):

```
class CryptoProvider {
  public async hashStringBySHA256(str: string): Promise<string> {
    return new Promise((resolve, reject) => {
      try {
        const hash = crypto.createHash('sha256');
        hash.update(str);
        resolve(hash.digest('hex'));
      }
      catch (err) {
        reject(err);
      }
    });
  }
}
```

Данный класс будет предоставлять различный функционал от библиотеки crypto, которая предоставляет различные криптографические функции. На данный момент – мы создали функцию `hashStringBySHA256`, которая хэширует переданную строку с помощью алгоритма SHA-256 и возвращает созданный хэш.

Также добавим модуль `CryptoModule` (`./src/crypto/crypto.module.ts`), который будет создавать экземпляр класса `CryptoProvider`:

```
class CryptoModule {
  constructor() {
    dependencyContainer.registerInstance('cryptoProvider', new CryptoProvider());
  }
}
```

Теперь создадим этот модуль `CryptoModule` в главном модуле приложения `AppModule`, чтобы мы могли пользоваться `CryptoProvider` в приложении:

```
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('mailerModule', new MailerModule({
      host: 'localhost',
      service: process.env.EMAIL_SERVICE_DEV,
      auth: {
        user: process.env.EMAIL_USERNAME_DEV,
        pass: process.env.EMAIL_PASSWORD_DEV
      },
      tls: {
```



```

        rejectUnauthorized: false
    }
  });
  dependencyContainer.registerInstance('cryptoModule', new CryptoModule());
  dependencyContainer.registerInstance('userModule', new UserModule());
  dependencyContainer.registerInstance('postModule', new PostModule());
}
}

```

Теперь нам нужно перейти в UserService (./src/domain/user/user.service.ts), так как именно там происходит создание пользователя в базе данных, то есть запись пароля пользователя в таблицу пользователя. Это происходит в методе createUser:

```

public async createUser(userData: IUserCreate): Promise<User | never> {
  userData.password = await this.cryptoProvider.hashStringBySHA256(userData.password);
  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);
}

```

Здесь мы в объекте запроса изменяем пароль пользователя, захэшировав его через экземпляр класса CryptoProvider через метод hashStringBySHA256. После того, как мы изменили пароль в объекте запроса на его хэш – мы уже дальше спокойно создаем пользователя в базе данных. Теперь в базе данных будет храниться именно хэш пароля, а не сам пароль. Мы смогли воспользоваться классом CryptoProvider в классе UserService за счет того, что инyectировали его в конструктор класса UserService:

```

class UserService {
  private readonly cryptoProvider: CryptoProvider;

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

```

```

];

constructor(cryptoProvider: CryptoProvider) {
  this.cryptoProvider = cryptoProvider;
}

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 getUserByEmail(email: string): Promise<User | never> {
  const user = await User.findOne({
    where: { email },
    include: this.userAssociations
  });
  if (!user) {

```

```

        throw new NotFound(`User with email: ${email} - is not found`);
    }

    return user;
}

public async createUser(userData: IUserCreate): Promise<User | never> {
    userData.password = await this.cryptoProvider.hashStringBySHA256(userData.password);
    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);
}

public async updateUserById(id: string, newUserData: IUserUpdate): Promise<User | never> {
    const user = await this.getUserById(id);
    Object.assign(user, newUserData);
    await user.save();

    return user;
}

public async deleteUserById(id: string): Promise<void> {
    const user = await this.getUserById(id);
    await user.destroy();
}
}

```

Сам UserService создается в UserModule и именно там передается синглтон CryptoProvider в конструктор:

```

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

    constructor() {
        this.userService = new UserService(
            dependencyContainer.getInstance<CryptoProvider>('cryptoProvider')
        );
    }
}

```

```

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

```

Все это возможно за счет реализованного Dependency Injection в приложении.

Теперь можно переходить непосредственно к реализации авторизации в приложении.

Для реализации заданной авторизации – создадим новый модуль auth (./src/auth), который будет отвечать за все, что связано с авторизацией. Исходя из заданных требований – нам потребуется создать:

- AuthModule (./src/auth/auth.module.ts)
- AuthRouter (./src/auth/auth.routes.ts)
- AuthController (./src/auth/auth.controller.ts)
- AuthGuard (./src/auth/auth.guard.ts)

Разберемся с каждым по порядку:

#### 1) AuthModule:

Данный модуль будет создавать экземпляры классов AuthController и AuthRouter, а также передавать в их конструкторы все классы, чья функциональность потребуется данным классам:

```

class AuthModule {
  constructor(cookieConfig: CookieOptions) {
    dependencyContainer.registerInstance('authController', new AuthController(
      dependencyContainer.getInstance<MailerTransporter>('mailerTransporter'),
      dependencyContainer.getInstance<CryptoProvider>('cryptoProvider'),
      dependencyContainer.getInstance<UserController>('userController'),
      cookieConfig
    ));

    dependencyContainer.registerInstance('authRouter', new AuthRouter());
  }
}

```

То есть классу AuthController – потребуются функциональности из классов MailerTransporter, CryptoProvider и UserController.

Также не забудем создать этот модуль в главном модуле приложения AppModule:

```
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('mailerModule', new MailerModule({
      host: 'localhost',
      service: process.env.EMAIL_SERVICE_DEV,
      auth: {
        user: process.env.EMAIL_USERNAME_DEV,
        pass: process.env.EMAIL_PASSWORD_DEV
      },
      tls: {
        rejectUnauthorized: false
      }
    }));
    dependencyContainer.registerInstance('cryptoModule', new CryptoModule());
    dependencyContainer.registerInstance('userModule', new UserModule());
    dependencyContainer.registerInstance('authModule', new AuthModule({
      httpOnly: true,
      secure: false,
      sameSite: 'lax',
      maxAge: 300000
    }));
    dependencyContainer.registerInstance('postModule', new PostModule());
  }
}
```

## 2) AuthRouter:

```
class AuthRouter {
  private readonly authRouter: Router;
  private readonly authController: AuthController;

  constructor() {
    this.authRouter = express.Router();
    this.authController =
dependencyContainer.getInstance<AuthController>('authController');
    this.setupAuthRouter();
  }

  public getAuthRouter(): Router {
    return this.authRouter;
  }

  private setupAuthRouter(): void {
    this.authRouter.get('/signin', (...args) => this.authController.signin(...args));
    this.authRouter.get('/signup/verify', (...args) =>
this.authController.verifyEmail(...args));
    this.authRouter.post('/signup', (...args) => this.authController.signup(...args));
    this.authRouter.use(authGuard);
    this.authRouter.get('/logout', (...args) => this.authController.logout(...args));
  }
}
```

Данный класс определяет все маршруты для авторизации в приложении, вызывая для соответствующего маршрута – определенный метод из AuthController. Данный роутер работает по тем же принципам, что и остальные роутеры в приложении. И его также нужно зарегистрировать в main.ts:

```
app.use('/auth',
dependencyContainer.getInstance<AuthRouter>('authRouter').getAuthRouter());
```

Здесь мы устанавливаем, что все маршруты авторизации будут начинаться с /auth. Данный роутер добавляет следующие маршруты:

- /auth/signup – маршрут для регистрации
- /auth/signup/verify – маршрут для верификации email
- /auth/signin – маршрут для аутентификации
- /auth/logout – маршрут для выхода из аккаунта

## 3) AuthController:

Данный контроллер будет предоставлять все методы, связанные с авторизацией. Для реализации всех функциональностей данного контроллера – нужно

инъектировать в него другие классы, которые будут предоставлять свои функциональности, необходимые для работы текущего контроллера:

```
class AuthController {
  constructor(
    private readonly mailerTransporter: MailerTransporter,
    private readonly cryptoProvider: CryptoProvider,
    private readonly userController: UserController,
    private readonly cookieConfig: CookieOptions
  ) { }
```

Реализуем четыре метода класса AuthController:

a. signup:

```
public async signup(req: Request, res: Response, next: NextFunction) {
  try {
    this.userController.isReqUserDataCreateValid(req, res);

    const userData = req.body;
    const verificationToken: string =
this.cryptoProvider.generateSecureVerificationToken();
    const verLink = process.env.CUR_URL +
`/auth/signup/verify?token=${verificationToken}`;

    await this.mailerTransporter.sendMailByTransporter({
      to: userData.email,
      subject: 'Verify your email',
      html: `

Hello, it's Daccord Service) We are glad to welcome you as a user of
our application!


```

```
}  
}
```

Данный метод будет отвечать за регистрацию пользователя в приложении. Каждый раз, когда пользователь будет пытаться зарегистрироваться – будет вызываться этот метод. Это будет post-метод, который принимает в теле запроса json. Напомню, что тело должно соответствовать этим схемам:

- IUserCreate:

```
interface IUserCreate {  
  name: string;  
  role: 'admin' | 'user';  
  email: string;  
  password: string;  
  contacts?: IUserContact[];  
}
```

Схема UserCreateSchema по IUserCreate:

```
const UserCreateSchema: ObjectSchema<IUserCreate> = Joi.object({  
  name: Joi.string().min(2).max(15).required(),  
  role: Joi.string().valid('admin', 'user').required(),  
  email: Joi.string().email().required(),  
  password: Joi.string().pattern(  
    new RegExp('^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$%*?&]).{8,}$')  
  ).required(),  
  contacts: Joi.array().items(UserContactSchema).optional()  
});
```

- IUserContact:

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

Схема UserContactSchema по IUserContact:

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

Чтобы проверить, что тело запроса соответствует заданным схемам – нужно провести валидацию тела запроса через данные схемы. В контроллере



UserController (/src/domain/user/user.controller.ts) есть специальный метод, который именно и проводит данную валидацию isReqUserDataCreateValid:

```
public isReqUserDataCreateValid(req: Request, res: Response): void | Response {
  const userDataCreate = req.body;
  const { error } = UserCreateSchema.validate(userDataCreate);

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

Данный метод возвращает ответ 422 о ошибке валидации, если тело запроса не соответствует схемам. В противном случае – метод ничего не возвращает.

Именно это мы и выполняем сначала в методе signup, и именно поэтому мы инъецировали до этого в AuthController - UserController:

```
this.userController.isReqUserDataCreateValid(req, res);
```

Мы инъецировали в AuthController – UserController и за счет это можем вызвать метод из UserController, который будет проводить нам валидацию тела запроса. То есть перед тем, как зарегистрировать пользователя – мы должны убедиться, что он отправил верные данные для создания данного пользователя в базе данных. В этом методе – мы как-раз это и проверяем.

Если ошибки валидации нет (пользователь отправил верное тело запроса), то дальше нам необходимо проверить – свою ли почту ввел пользователь. Как было сказано ранее – для этого мы отправим письмо на почту пользователя, которое будет содержать ссылку для активации его аккаунта. Ссылка будет содержать маршрут до другого метода в AuthController – до verifyEmail. Логика будет такая:

В методе signup создается ссылка, перейдя по которой заработает другой метод AuthController - verifyEmail. В эту ссылку вставляется уникальный сгенерированный токен в запросе. Этот токен сохраняется в созданный куки на ограниченное время (то есть у пользователя будет ограниченное время, чтобы активировать свой аккаунт. Если пользователь не успеет, то куки с токеном истечет и придется заново проходить процедуру регистрации). Затем с помощью nodemailer письмо отправляется на почту пользователя, которая будет содержать данную ссылку. После метод signup завершается, отправляя ответ о том, что письмо активации аккаунта отправлено пользователю на почту.

Для генерации уникального токена – мы создадим в уже описанный ранее класс CryptoProvider метод generateSecureVerificationToken:

```
class CryptoProvider {
  public async hashStringBySHA256(str: string): Promise<string> {
```

```

return new Promise((resolve, reject) => {
  try {
    const hash = crypto.createHash('sha256');
    hash.update(str);
    resolve(hash.digest('hex'));
  }
  catch (err) {
    reject(err);
  }
});
}

public generateSecureVerificationToken(): string {
  return crypto.randomBytes(20).toString('hex');
}
}

```

Данный метод будет генерировать случайный уникальный токен и возвращать его. Именно поэтому мы инъектировали CryptoProvider в AuthController.

После создания generateSecureVerificationToken и инъекции CryptoProvider в AuthController, выполним в методе signup следующее:

```

const userDataCreate = req.body;
const verificationToken: string = this.cryptoProvider.generateSecureVerificationToken();
const verLink = process.env.CUR_URL + `/auth/signup/verify?token=${verificationToken}`;

```

Здесь:

- userDataCreate – получаем тело запроса, которое содержит данные для создания пользователя
- verificationToken – генерируем с помощью generateSecureVerificationToken случайный уникальный токен
- verLink – создаем ссылку на метод verifyEmail и передаем в запрос созданный verificationToken  
([http://localhost:5000/auth/signup/verify?token=\\${verificationToken}](http://localhost:5000/auth/signup/verify?token=${verificationToken})) – будет создана такая ссылка)

Далее мы должны отправить письмо на почту пользователя, которое будет содержать созданную ссылку verLink. Для этого мы воспользуемся описанным ранее MailerTransporter (именно поэтому мы инъектировали его в AuthController):

```

await this.mailerTransporter.sendMailByTransporter({
  to: userDataCreate.email,
  subject: 'Verify your email',

```

```
html: `

Hello, it's Daccord Service) We are glad to welcome you as a user of  
our application!

  
<p>Please confirm your email by clicking on the link: <a href="${verLink}">Verify  
Email</a></p>`  
});
```

Здесь вызывается метод `sendMailByTransporter` у `MailerTransporter`. В данный метод передается описанный ранее конфиг `mailOptions`, содержащий следующие данные:

- Кому отправляется письмо (на почту пользователя, которую получаем через `userDataCreate.email`)
- `subject` – тема письма (Верификация почты)
- `html` – создаем текст письма (Пишем, что письмо от нашего приложения и для того, чтобы зарегистрироваться на нашем приложении – нужно перейти по данной ссылке, тем самым подтвердив свою почту (ссылка – это созданный ранее `verLink`))

После отправки письма – мы должны сохранить сгенерированный токен `verificationToken`, который содержится в ссылке, отправленной пользователю на почту. Также мы должны сохранить тело запроса, которое содержит данные для создания пользователя. Токен нам нужно сохранить, так как когда пользователь перейдет по ссылке из своей почты – будет вызываться метод `verifyEmail`. Этот метод будет проверять – действительно ли указанная почта принадлежит пользователю и если выяснится, что принадлежит, то активирует аккаунт пользователя. Чтобы совершить данную проверку – методу `verifyEmail` нужно сравнить токен, который пришел в запросе, а также токен, который был сгенерирован в методе `signup`. И именно поэтому мы должны сохранить `verificationToken` в методе `signup`. Тело запроса нужно сохранить, чтобы `verifyEmail` мог создать пользователя в базе данных в случае успешной верификации email на основе данного тела.

И токен верификации, и тело запроса сохраним через куки-файлы. Для этого нам нужно передать в конструкторе класса `AuthController` – специальный конфиг `cookieConfig` типа `CookieOptions`, который будет настраивать куки (именно поэтому мы до этого инъектировали в `AuthController` – `cookieConfig`).

Теперь мы можем создать куки и сохранить там токен верификации и тело запроса:

```
res.cookie('authToken', verificationToken, this.cookieConfig);  
res.cookie('userDataCreate', userDataCreate, this.cookieConfig);
```

И именно для этого – мы в главном модуле приложения `AppModule` передавали в `AuthModule` специальный конфиг для куки:

```

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('mailerModule', new MailerModule({
      host: 'localhost',
      service: process.env.EMAIL_SERVICE_DEV,
      auth: {
        user: process.env.EMAIL_USERNAME_DEV,
        pass: process.env.EMAIL_PASSWORD_DEV
      },
      tls: {
        rejectUnauthorized: false
      }
    }));
    dependencyContainer.registerInstance('cryptoModule', new CryptoModule());
    dependencyContainer.registerInstance('userModule', new UserModule());
    dependencyContainer.registerInstance('authModule', new AuthModule({
      httpOnly: true,
      secure: false,
      sameSite: 'lax',
      maxAge: 300000
    }));
    dependencyContainer.registerInstance('postModule', new PostModule());
  }
}

```

AuthModule инжектирует этот куки-конфиг в AuthController, чтобы настроить куки, через которые мы сохраняем токен верификации и тело запроса.

Здесь:

- `httpOnly: true` – куки будут передаваться только через HTTP
- `secure: false` – куки будут отправлять по незашифрованному соединению (мы используем такую настройку, так как на данный момент приложение работает локально через http. Такую настройку можно использовать для локальной разработки. Для production-версии – нужно будет изменить на `true`)
- `sameSite: 'lax'` - Указывает политику, которая определяет, когда браузер должен отправлять cookie вместе с запросами к другим доменам. Значение `lax` означает, что cookie будет отправлен только при прямых запросах или при переходе по ссылкам
- `maxAge` – устанавливаем время жизни куки в миллисекундах. В данном случае время жизни – 5 минут. Это означает, что, у пользователя будет 5 минут, чтобы зайти на свою почту и активировать свой аккаунт через ссылку. Если пользователь опоздает – время куки истечет и пользователю придется заново выполнять процедуру регистрации

После сохранения токена верификации и тела запроса в куки - отправляем ответ со статусом 200 о том, что пользователю на почту отправлено письмо для активации аккаунта:

```
return res.status(200).send(`Email successfully sent to ${userDataCreate.email}`);
```

На этом – метод регистрации `signup` завершает свою работу.

b. `verifyEmail`:

```
public async verifyEmail(req: Request, res: Response, next: NextFunction) {
  try {
    const { token } = req.query;
    if (
      !req.cookies || !req.cookies['authToken'] || !req.cookies['userDataCreate'] ||
      req.cookies['authToken'] !== token
    ) {
      return res.status(422).send(
        "Account has not been verified! The token didn't match or has expired. Return to the registration page and try to register again."
      );
    }

    await this.userController.createUser(req.cookies['userDataCreate']);
    return res.redirect(process.env.CUR_URL + '/api');
  }
  catch (err) {
```

```
    next(err);  
  }  
}
```

Данный метод, как было сказано ранее – является вторым этапом метода `signup`. Метод `verifyEmail` проверяет, что пользователь ввел именно свою почту. Если проверка пройдет успешно, то данный метод создаст пользователя в базе данных. Именно этот метод вызывается после того, как пользователь перейдет по ссылке, отправленной ему на почту.

Разберем подробно данный метод:

Здесь мы получаем токен, который был отправлен в запросе (в `signup` в ссылку передается сгенерированный токен):

```
const { token } = req.query;
```

Дальше мы проводим проверку на то, что полученный токен равен сохраненному в куки токenu верификации, а также, что куки, в который сохранено тело запроса, существует:

```
if (  
  !req.cookies || !req.cookies['authToken'] || !req.cookies['userDataCreate'] ||  
  req.cookies['authToken'] !== token  
) {  
  return res.status(422).send(  
    "Account has not been verified! The token didn't match or has expired. Return  
    to the registration page and try to register again."  
  );  
}
```

Если эта проверка не проходит, то возвращается ответ со статусом 422, сообщающий, что аккаунт не был активирован, так как сохраненный токен верификации либо не совпадает с переданным токеном, либо сохраненный токен верификации истек и нужно попробовать опять зарегистрироваться.

Если же проверка прошла успешно, то пользователь создается в базе данных за счет метода `createUser` контроллера `UserContoller`:

```
await this.userController.createUser(req.cookies['userDataCreate']);
```

После успешного создания пользователя в базе данных – происходит перенаправление пользователя на главную страницу приложения, чтобы ему не пришлось самому возвращаться и перезагружать страницу:

```
return res.redirect(process.env.CUR_URL + '/api');
```

На этом метод `verifyEmail` завершает свою работу. Теперь пользователь остается только воспользоваться методом `signin`, чтобы войти в систему.

с. `signin`:

```
public async signin(req: Request, res: Response, next: NextFunction) {
  try {
    const user = await this.userController.getUserByEmail(req);
    const userPassword = req.query.userPassword as string;
    const hashUserPassword = await
this.cryptoProvider.hashStringBySHA256(userPassword);

    if (user.dataValues.password !== hashUserPassword) {
      return res.status(401).send('Unauthorized - incorrect password');
    }

    req.session.userId = user.dataValues.id;
    return res.status(200).json({ status: 200, data: user, message: "Authorization was
successful" });
  }
  catch (err) {
    next(err);
  }
}
```

С помощью этого метода пользователь будет авторизоваться в приложении. Чтобы авторизоваться – нужно отправить `get`-запрос, содержащий почту зарегистрированного пользователя и его пароль.

Этот метод получает переданную почту и пароль, ищет пользователя с таким паролем. Если такой пользователь будет найден, то метод сравнит пароль этого пользователя с переданным паролем. Если все совпало, то метод создает сессионную куку, с помощью которой пользователь авторизируется.

Разберем подробно:

Здесь метод получает пользователя по переданному email с помощью инъектированного контроллера `UserController`:

```
const user = await this.userController.getUserByEmail(req);
```

Если пользователя с таким email не существует, то метод `getUserByEmail` вернет ответ со статусом 404, который сообщает, что пользователь с указанным email не найден. Если же пользователь найден, то дальше метод получает переданный пароль:

```
const userPassword = req.query.userPassword as string;
```

Дальше метод должен сравнить полученный пароль из запроса, а также полученный пароль пользователя. Все пароли пользователей хранятся в базе данных в захешированном виде (как уже описывалось ранее – пароли хэшируются с помощью класса CryptoProvider и его метода hashStringBySHA256). Обратно получить пароль из хэша нельзя, поэтому единственное что остается – это захешировать полученный в запросе пароль с помощью этого же метода hashStringBySHA256 и сравнить два этих хэша. Если пароли действительно совпадают, то их хэши будут равны. Это мы здесь и делаем:

```
const hashUserPassword = await this.cryptoProvider.hashStringBySHA256(userPassword);

if (user.dataValues.password !== hashUserPassword) {
  return res.status(401).send('Unauthorized - incorrect password');
}
```

Если хэши паролей не совпадают, то значит и сами пароли не совпадают. В таком случае – возвращается ответ 401, сообщающий, что пароли не совпали.

Если же хэши совпали, то значит и пароли совпали. В таком случае создается сессионная кука, которая связывается с id пользователя, который вошел в систему, а после отправляется ответ со статусом 200, сообщающий о том, что авторизация прошла успешно:

```
req.session.userId = user.dataValues.id;
return res.status(200).json({ status: 200, data: user, message: "Authorization was successful" });
```

На этом метод signin завершает свою работу. Пользователь вошел в систему и может пользоваться функционалом, который требует авторизации.

Важно уточнить – чтобы пользоваться сессиями, нужно в main.ts импортировать следующий модуль, который позволяет создавать сессии:

```
import session from 'express-session';
```

Также в main.ts нужно настроить сессии:

```
app.use(session({
  secret: process.env.SECRET!,
  resave: false,
  saveUninitialized: true,
  cookie: {
    httpOnly: true,
    secure: false,
```



```
sameSite: 'lax',  
maxAge: 7200000  
}  
)))
```

Здесь мы устанавливаем сессии в наше приложения, передавая конфиг для сессий:

- `secret` – ключевое слово, которое используется для шифрования данных сессии. В данном случае – берем это слово из `env`-переменной `SECRET`
- `resave: false` – контроль поведения сессии при сохранении. В данном случае – сессия не будет перезаписываться даже если клиент отключился и затем снова подключился к серверу
- `saveUninitialized: true` - определяет, следует ли сохранять новые сессии, которые были инициализированы, но еще не изменялись. Установка этого значения в `true` означает, что все новые сессии будут сохраняться, даже если они не были изменены
- `cookie` – объект, содержащий настройки куки, которые используются для управления сессией:
  - a. `httpOnly: true` – куки будут передаваться только через HTTP
  - b. `secure: false` – куки будут отправлять по незашифрованному соединению (мы используем такую настройку, так как на данный момент приложение работает локально через `http`. Такую настройку можно использовать для локальной разработки. Для `production`-версии – нужно будет изменить на `true`)
  - c. `sameSite: 'lax'` - Указывает политику, которая определяет, когда браузер должен отправлять `cookie` вместе с запросами к другим доменам. Значение `lax` означает, что `cookie` будет отправлен только при прямых запросах или при переходе по ссылкам
  - d. `maxAge` – устанавливаем время жизни куки в миллисекундах. В данном случае время жизни – 2 часа. Это означает, что, когда пользователь авторизуется в системе – он сможет два часа взаимодействовать с приложением как авторизованный пользователь (даже если пользователь выйдет из сайта и заново зайдет – не нужно будет заново авторизоваться). По истечению двух часов – пользователю придется заново авторизоваться в системе. Также если вдруг наш сервер перезапустится – пользователю придется заново авторизоваться, даже если два часа еще не прошли

Также, чтобы мы могли связывать сессию с `id` пользователя – нужно расширить модуль `express-session`, указав в интерфейсе `SessionData` (данные, которые может

хранить сессия) новый параметр `userId` типа `string`. Сделаем это в конфиге приложения (`./src/app.config.ts`):

```
declare module 'express-session' {  
  interface SessionData {  
    userId: string;  
  }  
}  
  
export default () => ({  
  port: parseInt(process.env.PORT as string, 10) || 3000  
});
```

То есть наша авторизация работает через механизм сессий. Сессия связывается с пользователем через его `id` и таким образом – наше приложение будет понимать какой пользователь к какой сессии относится и авторизован ли пользователь. Сама сессия хранится в специальном куки, который будет содержать `id` сессии, который генерируется через установленный в конфиге `secret`. Сам кук будет временным. То есть, когда кук истечет – сессия (кук хранит `id` сессии) будет недоступна и приложение будет воспринимать пользователя как неавторизованного. Таким образом приложение и будет определять авторизован ли пользователь или нет. Для этого проверяется – существует ли сессия, привязанная к пользователю или нет. Если да, то пользователь авторизован. Если нет, то нет.

d. `logout`:

```
public logout(req: Request, res: Response, next: NextFunction) {  
  try {  
    req.session.destroy(() => {  
      return res.status(200).json({ status: 200, message: "Logged out successfully" });  
    });  
  }  
  catch (err) {  
    next(err);  
  }  
}
```

Данный метод позволяет пользователю выйти из аккаунта. Для этого данный метод просто уничтожает сессию пользователя с помощью метода `destroy` и возвращает ответ со статусом 200 о том, что пользователь успешно вышел из аккаунта

#### 4) AuthGuard:

```
const authGuard = (req: Request, res: Response, next: NextFunction) => {  
  if (!req.session.userId) {  
    return res.status(401).send(' Unauthorized' );  
  }  
  
  next();  
};
```

Данная функция является защитником, защищающим все маршруты, которые требуют авторизации, от пользователей, которые не прошли авторизацию. Как я уже описал ранее – если сессия, привязанная к пользователю через его id доступна, то пользователь авторизован. Если нет, то нет.

Здесь – если такая сессия недоступна, то возвращается ответ, сообщающий о том, что пользователю нужно авторизоваться перед тем, как получить доступ к заданному функционалу:

```
if (!req.session.userId) {  
  return res.status(401).send(' Unauthorized' );  
}
```

Если же такая сессия доступна, то просто вызывается next, тем самым передавая запрос на обработку следующему middleware или маршруту в цепочке.

В нашем приложении authGuard будет применяться:

- a. /auth: методы регистрации /auth/signup, авторизации /auth/signin, верификации email /auth/signup/verify не будут требовать авторизации. Поэтому для данных маршрутов authGuard не будет применен. Однако авторизация требуется для метода выхода из системы /auth/logout, поэтому для этого маршрута мы добавим authGuard. Чтобы это реализовать – мы должны в роутере AuthRouter добавить authGuard для маршрута /auth/logout следующим образом:

```
class AuthRouter {  
  private readonly authRouter: Router;  
  private readonly authController: AuthController;  
  
  constructor() {  
    this.authRouter = express.Router();  
    this.authController =  
dependencyContainer.getInstance<AuthController>(' authController' );  
    this.setupAuthRouter();  
  }
```

```

public getAuthRouter(): Router {
    return this.authRouter;
}

private setupAuthRouter(): void {
    this.authRouter.get('/signin', (...args) => this.authController.signin(...args));
    this.authRouter.get('/signup/verify', (...args) =>
this.authController.verifyEmail(...args));
    this.authRouter.post('/signup', (...args) => this.authController.signup(...args));
    this.authRouter.use(authGuard);
    this.authRouter.get('/logout', (...args) => this.authController.logout(...args));
}
}

```

- b. /api/users: здесь абсолютно во все маршруты будет внедрен authGuard, за исключением метода получения всех пользователей. Чтобы это реализовать — мы должны в роутере UserRouter добавить authGuard для всех маршрутов, кроме /api/users следующим образом:

```

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.use(authGuard);
        this.userRouter.get('/:id', (...args) => this.userController.getUserById(...args));
        this.userRouter.put('/:id', (...args) =>
this.userController.updateUserById(...args));
        this.userRouter.patch('/:id', (...args) =>
this.userController.updateUserById(...args));
        this.userRouter.delete('/:id', (...args) =>
this.userController.deleteUserById(...args));
    }
}

```

- с. /api/posts: здесь абсолютно во все маршруты будет внедрен authGuard. Чтобы это реализовать – мы должны в роутере PostRouter добавить authGuard для всех маршрутов следующим образом:

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

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

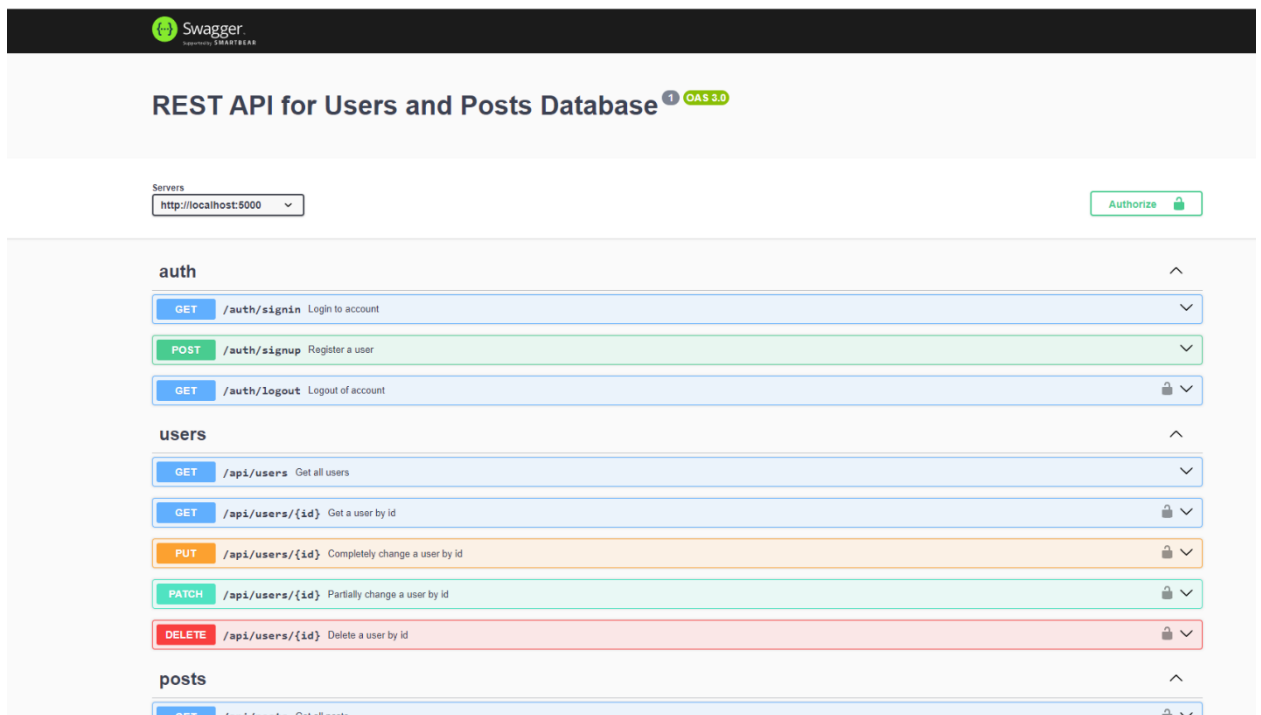
  public getPostRouter(): Router {
    return this.postRouter;
  }

  private setupPostRouter(): void {
    this.postRouter.use(authGuard);
    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));
    this.postRouter.put('/:id', (...args) =>
      this.postController.updatePostById(...args));
    this.postRouter.patch('/:id', (...args) =>
      this.postController.updatePostById(...args));
    this.postRouter.delete('/:id', (...args) =>
      this.postController.deletePostById(...args));
  }
}
```

## Тесты:

Тестировать приложение мы будем с помощью OpenApi-спецификации Swagger. Для этого – добавим в swagger.yaml (./src/swagger/swagger.yaml) новый реализованный модуль auth с методами для авторизации, а также установим защиту для всех методов, которые требуют авторизации.

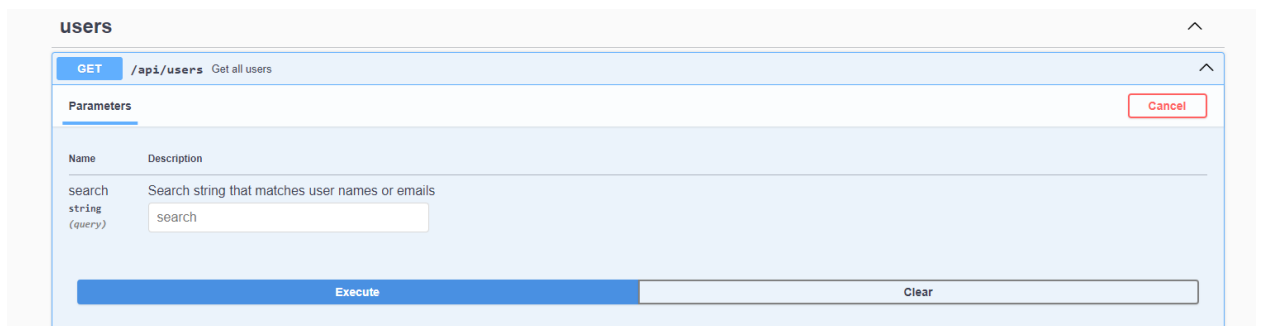
Запустим приложение с помощью npm run start:dev и перейдем на <http://localhost:5000/api:>

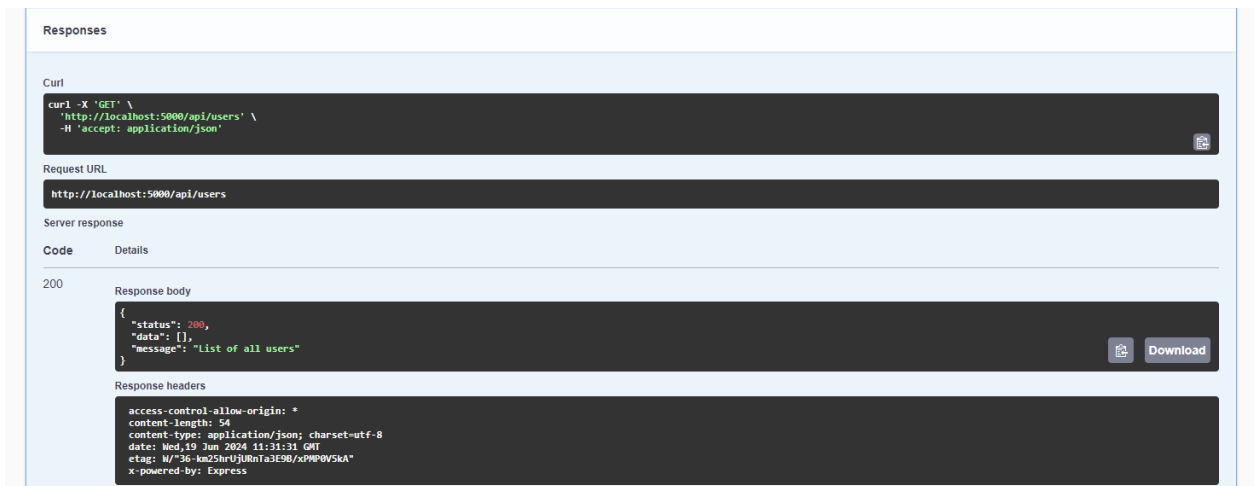


Как можно заметить – новый модуль auth добавился. Все методы, которые требуют авторизации – помечены замком.

Теперь можно перейти к тестам:

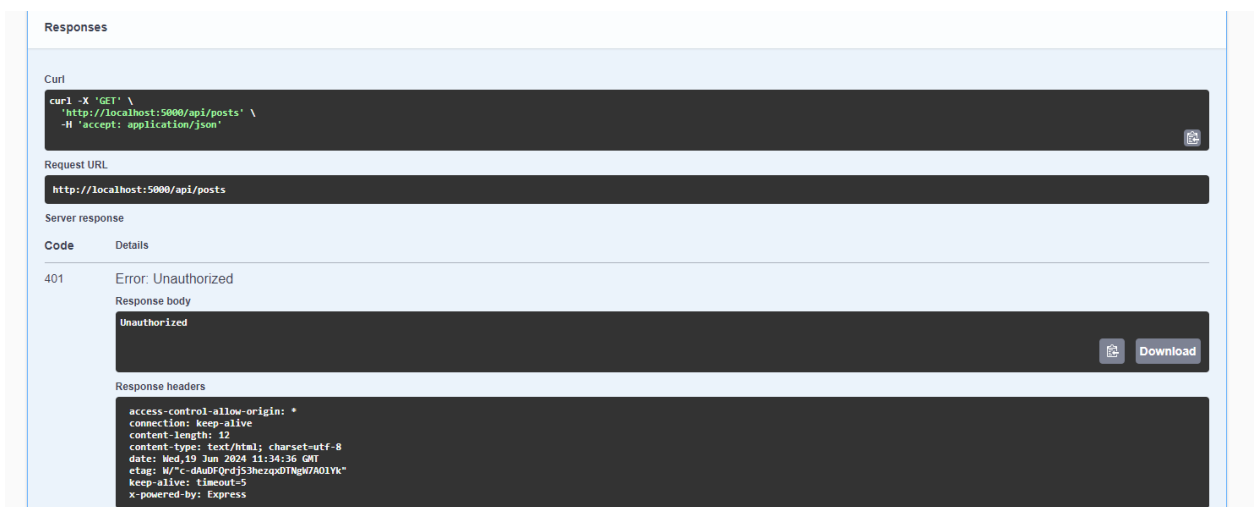
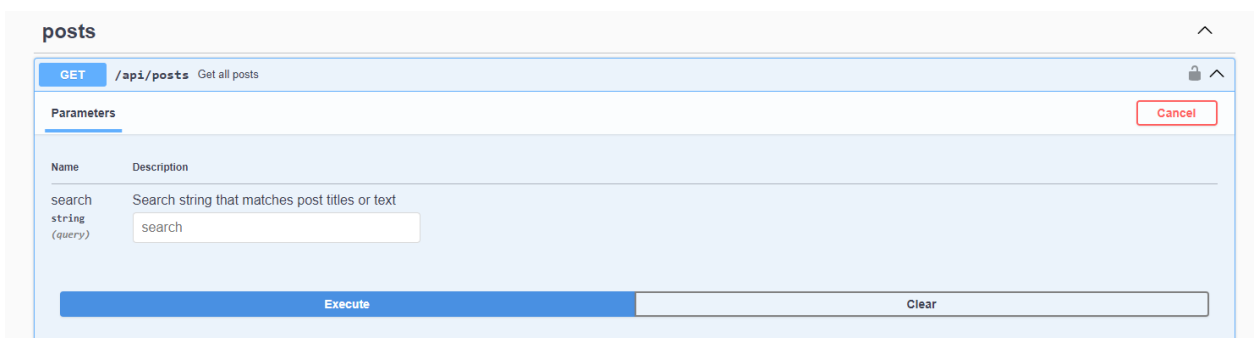
**Сначала попробуем выполнить метод Get all users, который не требует авторизации. Выберем этот метод, нажмем Try it out и после Execute:**





Как можно заметить – метод выполнялся без ошибок со статусом 200 и вернул пустой массив, так как еще мы не создали никаких пользователей. Данный метод не требует авторизации, поэтому он отработал правильно.

**Теперь попробуем обратиться к методу, который требует авторизации. Например, к Get all posts. Выберем этот метод, нажмем Try it out и после Execute:**



Как можно заметить – метод вернул ответ 401 Unauthorized. Все правильно – этот метод требует авторизации, мы не авторизировались, значит сработал authGuard и не дал выполнить нам этот метод.

**Теперь попробуем зарегистрироваться. Для этого выберем метод /auth/signup Register a user. Мы попробуем создать клиента со следующими данными:**

```
{
  "name": "jason_statham",
  "role": "user",
  "email": "st1035@mail.ru",
  "password": "bjghHdghfj123!!??",
  "contacts": [
    {
      "type": "telegram",
      "value": "link_to_telegram"
    },
    {
      "type": "fb",
      "value": "link_to_fb"
    }
  ]
}
```

Мы указали почту [st1035@mail.ru](mailto:st1035@mail.ru) – это значит, что на эту почту должно прийти письмо для верификации. Выберем сам метод, нажмем Try it out, введем данный объект в тело и нажмем Execute:



auth

GET /auth/signin Login to account

POST /auth/signup Register a user

Parameters

No parameters

Request body required

application/json

Examples:

[Modified value]

```
{
  "name": "jason_statham",
  "role": "user",
  "email": "st1035@mail.ru",
  "password": "bjghHdghfj1231!??",
  "contacts": [
    {
      "type": "telegram",
      "value": "link_to_telegram"
    },
    {
      "type": "fb",
      "value": "link_to_fb"
    }
  ]
}
```

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/auth/signup' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "jason_statham",
    "role": "user",
    "email": "st1035@mail.ru",
    "password": "bjghHdghfj1231!??",
    "contacts": [
      {
        "type": "telegram",
        "value": "link_to_telegram"
      },
      {
        "type": "fb",
        "value": "link_to_fb"
      }
    ]
  }'
```

Request URL

http://localhost:5000/auth/signup


Server response

Code	Details
200	<p>Response body</p> <p>Email successfully sent to st1035@mail.ru</p> <p>Download</p>

Как можно заметить – нам пришел ответ 200 о том, что на почту [st1035@mail.ru](mailto:st1035@mail.ru) отправлено письмо активирования аккаунта. Значит – данный метод отработал правильно и отправил письмо верификации на почту пользователя.

Теперь перейдем на почту [st1035@mail.ru](mailto:st1035@mail.ru) и убедимся, что письмо действительно пришло:

## Verify your email

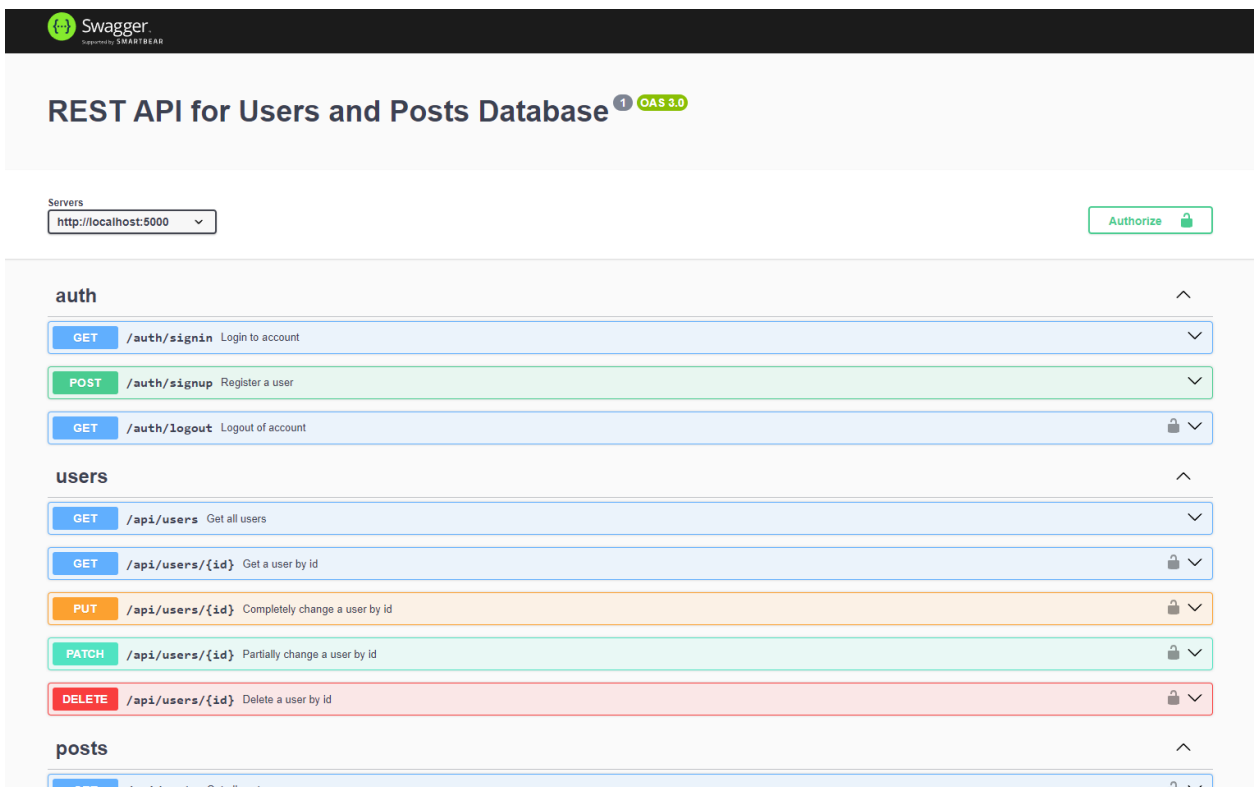
•  solarbvb@gmail.com Сегодня, 14:40  
Кому: вам

Hello, it's Daccord Service) We are glad to welcome you as a user of our application!

Please confirm your email by clicking on the link: [Verify Email](#)

Как можно заметить – письмо действительно пришло пользователю на почту.

Теперь перейдем по этой ссылке:



The screenshot shows the Swagger UI interface for a REST API titled "REST API for Users and Posts Database" with an OpenAPI 3.0 specification. The interface includes a "Servers" dropdown set to "http://localhost:5000" and an "Authorize" button. The API is organized into three main sections: "auth", "users", and "posts".

- auth**
  - GET `/auth/signin`: Login to account
  - POST `/auth/signup`: Register a user
  - GET `/auth/logout`: Logout of account
- users**
  - GET `/api/users`: Get all users
  - GET `/api/users/{id}`: Get a user by id
  - PUT `/api/users/{id}`: Completely change a user by id
  - PATCH `/api/users/{id}`: Partially change a user by id
  - DELETE `/api/users/{id}`: Delete a user by id
- posts**
  - GET `/api/posts`: Get all posts

Как можно заметить – переход по этой ссылке направил нас на страницу Swagger (считается как главная страница для тестов).

Проверим – создался ли новый пользователь с помощью метод **Get all users**, который не требует авторизации. Выберем этот метод, нажмем **Try it out**, а после **Execute**:

users

GET /api/users Get all users

Parameters

Name	Description
search	Search string that matches user names or emails
string	
(query)	<input type="text" value="search"/>

Execute

Clear

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": "854868f8-95f4-4b4e-8532-057a04f54a66",
      "name": "jason statham",
      "role": "user",
      "email": "st1035@mail.ru",
      "password": "195a1a0c651abf9960b9a08d7b4c8349680e694d58aa0f0d9f104bce025e7ff",
      "createdAt": "2024-06-19T11:28:25.483Z",
      "updatedAt": "2024-06-19T11:42:41.323Z",
      "rating": 0,
      "posts": [],
      "contacts": [
        {
          "id": "b949447f-4b43-45a5-8c58-ac41c2b3ce10",
          "type": "telegram",
          "value": "link_to_telegram",
          "userId": "854868f8-95f4-4b4e-8532-057a04f54a66",
          "createdAt": "2024-06-19T11:42:41.375Z",
          "updatedAt": "2024-06-19T11:42:41.375Z"
        },
        {
          "id": "152ed3de-b326-4710-bcc7-373851a887d7",
          "type": "fb",
          "value": "link_to_fb",
          "userId": "854868f8-95f4-4b4e-8532-057a04f54a66",
          "createdAt": "2024-06-19T11:42:41.375Z",
          "updatedAt": "2024-06-19T11:42:41.375Z"
        }
      ]
    }
  ]
}
```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 791
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 11:44:51 GMT
etag: W/"317-pr9Qb0716L5gfAghdWqo7Z78c4k"
keep-alive: timeout=5
x-powered-by: Express
```

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": "1951a0c651abf9960b0e8d7b4c8349680e694d580aa0f0d9f104bce025c7ff",
  "createdAt": "2024-06-19T11:28:25.483Z",
  "updatedAt": "2024-06-19T11:42:41.323Z",
  "rating": 0,
  "posts": [],
  "contacts": [
    {
      "id": "0949d47f-4b43-45a5-8c58-ac41c2b3ce10",
      "type": "telegram",
      "value": "link_to_telegram",
      "userId": "854868f8-95f4-4b4e-8532-057a04f54a66",
      "createdAt": "2024-06-19T11:42:41.375Z",
      "updatedAt": "2024-06-19T11:42:41.375Z"
    },
    {
      "id": "152ed3de-b326-4710-bcc7-373851a887d7",
      "type": "fb",
      "value": "link_to_fb",
      "userId": "854868f8-95f4-4b4e-8532-057a04f54a66",
      "createdAt": "2024-06-19T11:42:41.375Z",
      "updatedAt": "2024-06-19T11:42:41.375Z"
    }
  ],
  "subscribers": []
},
{
  "message": "List of all users"
}
```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 791
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 11:44:51 GMT
etag: W/"317-n900716L5gfAghdQp7778:4k"
keep-alive: timeout=5
x-powered-by: Express
```

Как можно заметить – пользователь создан в базе данных.

Теперь проверим что будет, если мы сейчас попытаемся выполнить метод, требующий авторизации. Должен сработать authGuard и выдать ответ 401, так как пока что мы просто зарегистрировали пользователя в системе, но мы не вошли еще в систему. Попробуем выполним метод Get a post by id. Выберем этот метод, нажмем Try it out, введем в параметр id типа uuid, а после Execute:

posts

GET /api/posts Get all posts

POST /api/posts Create a new post

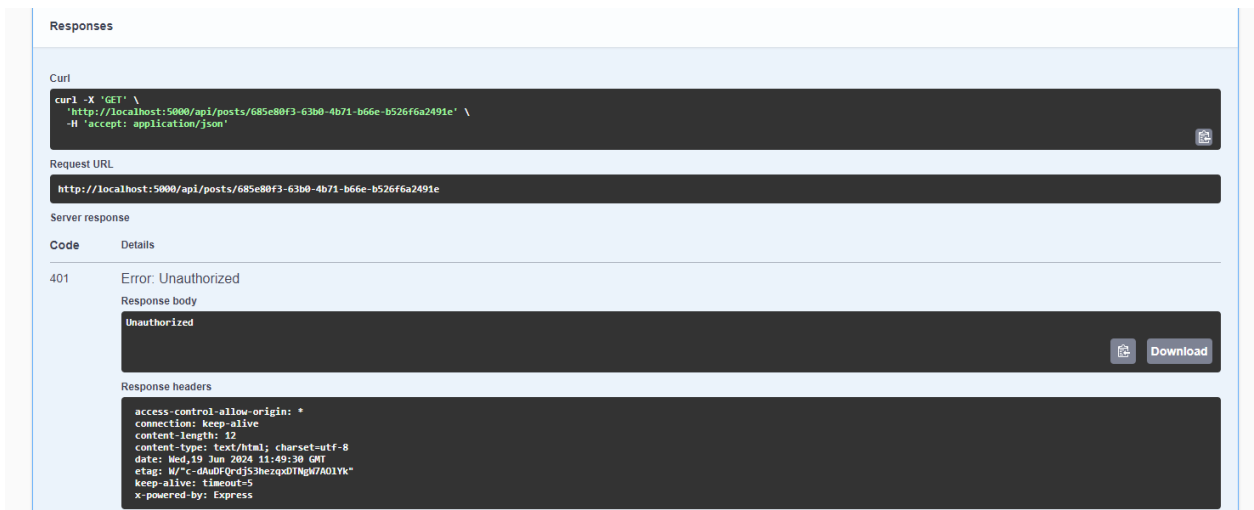
GET /api/posts/{id} Get a post by id

Parameters

Name	Description
id * required	Post ID. Example: 685e80f3-63b0-4b71-b66e-b526f6a2491e
string(\$uuid)	
(path)	

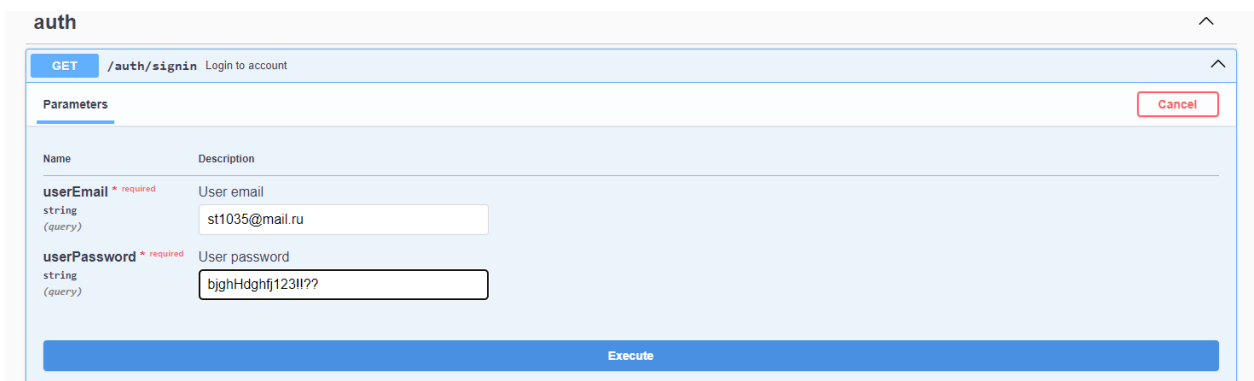
685e80f3-63b0-4b71-b66e-b526f6a2491e

Execute



Как можно заметить возвращается ответ 401 – то есть все работает правильно, authGuard не дает доступ пользователю, который не авторизовался.

**Теперь авторизируемся в системе. Для этого выберем метод `/auth/signin` Login to account, нажмем Try it out, введем почту и пароль пользователя, которого мы создали и нажмем Execute:**



Responses

Curl

Request URL

Server response

Code	Details
200	<div>Response body</div> <div>Response headers</div>

Responses

Curl

Request URL

Server response

Code	Details
200	<div>Response body</div> <div>Response headers</div>

Как можно заметить – мы успешно прошли авторизации. Теперь мы вошли в систему, как пользователь `jason_statham`.

Теперь попытаемся выполнить какой-то метод, который требует авторизации. Например, выберем метод `Create a new post`. Мы создадим свой следующий пост (это будет пост пользователя `jason_statham`):

```
{  
  "title": "Властелин колец - книги против фильмов",  
  "access": "public",  
  "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин  
Кольц...",  
  "tags": [  
    "фильмы",  
    "книги",  
    "Властелин Кольц"  
  ],  
  "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66"  
}
```

Выберем этот метод, нажмем Try it out, введем данный объект в тело запроса и нажмем Execute:

The screenshot shows a REST client interface with the following components:

- Header:** "posts" with a collapse icon.
- Method Selection:** Two tabs: "GET /api/posts Get all posts" (blue) and "POST /api/posts Create a new post" (green, active).
- Parameters:** A section with "No parameters" and buttons for "Cancel" and "Reset".
- Request body:** Labeled "required" with a dropdown menu set to "application/json".
- Examples:** A dropdown menu showing "[Modified value]".
- JSON Body:** A text area containing the following JSON:

```
{  
  "title": "Властелин колец - книги против фильмов",  
  "access": "public",  
  "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Кольц...",  
  "tags": [  
    "фильмы",  
    "книги",  
    "Властелин Кольц"  
  ],  
  "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66"  
}
```
- Execute:** A large blue button at the bottom labeled "Execute".

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/posts' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "Властелин колец - книги против фильмов",
    "access": "public",
    "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
    "tags": [
      "фильмы",
      "книги",
      "Властелин Колец"
    ],
    "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66"
  }'
```

Request URL

http://localhost:5000/api/posts

Server response

CodeDetails

201

Response body

```
{
  "status": 201,
  "data": {
    "id": "dd3bdf0f-acbb-4d2f-95fb-646b03169758",
    "title": "Властелин колец - книги против фильмов",
    "access": "public",
    "createdAt": "2024-06-19T11:28:25.471Z",
    "updatedAt": "2024-06-19T11:57:34.146Z",
    "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
    "rating": 0,
    "tags": [
      "фильмы",
      "книги",
      "Властелин Колец"
    ],
    "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66",
    "author": {
      "id": "854868f8-95f4-4b4e-8532-057a04f54a66",
      "name": "jason_statham",
      "role": "user",
      "email": "st1035@mail.ru",
      "password": "195a1a0c651abf9960b9a08d7b4c8349680e694d580aa0f0d9f104bce025c7ff",
      "createdAt": "2024-06-19T11:28:25.483Z",
      "updatedAt": "2024-06-19T11:42:41.323Z",
      "rating": 0
    }
  },
  "message": "Post successfully created"
}
```

Download

CodeDetails

201

Response body

```
{
  "status": 201,
  "data": {
    "id": "dd3bdf0f-acbb-4d2f-95fb-646b03169758",
    "title": "Властелин колец - книги против фильмов",
    "access": "public",
    "createdAt": "2024-06-19T11:28:25.471Z",
    "updatedAt": "2024-06-19T11:57:34.146Z",
    "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
    "rating": 0,
    "tags": [
      "фильмы",
      "книги",
      "Властелин Колец"
    ],
    "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66",
    "author": {
      "id": "854868f8-95f4-4b4e-8532-057a04f54a66",
      "name": "jason_statham",
      "role": "user",
      "email": "st1035@mail.ru",
      "password": "195a1a0c651abf9960b9a08d7b4c8349680e694d580aa0f0d9f104bce025c7ff",
      "createdAt": "2024-06-19T11:28:25.483Z",
      "updatedAt": "2024-06-19T11:42:41.323Z",
      "rating": 0
    }
  },
  "message": "Post successfully created"
}
```

Download

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 840
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 11:57:34 GMT
etag: W/"348-kf5d83rMRVWZ7c0lM83bcva0z7c"
keep-alive: timeout=5
location: /api/posts/dd3bdf0f-acbb-4d2f-95fb-646b03169758
x-powered-by: Express
```

Как можно заметить – пост пользователя jason\_statham был успешно создан, так как мы прошли авторизацию через /auth/signin.



Также выполним еще метод **Get all posts**, также требующий авторизации:

posts

GET /api/posts

Get all posts

Parameters

Cancel

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

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

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "dd3bdf0f-acbb-4d2f-95fb-646b03169758",
      "title": "Властелин колец - книга против фильмов",
      "access": "public",
      "createdAt": "2024-06-19T11:28:25.471Z",
      "updatedAt": "2024-06-19T11:57:34.146Z",
      "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
      "rating": 0,
      "tags": [
        "фильмы",
        "книги",
        "Властелин Колец"
      ],
      "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66",
      "author": {
        "id": "854868f8-95f4-4b4e-8532-057a04f54a66",
        "name": "jason_statham",
        "role": "user",
        "email": "st1035@mail.ru",
        "password": "195a1a0c651abf9960b9a08d7b4c8349680e694d580aa0f0d9f104bce025e7ff",
        "createdAt": "2024-06-19T11:28:25.483Z",
        "updatedAt": "2024-06-19T11:42:41.323Z",
        "rating": 0
      }
    }
  ]
}
```

Download

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 834
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 11:59:36 GMT
etag: W/"342-7fLvbYQIChp026w05prb+d2jPc"
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": "dd3bdf0f-acbb-4d2f-95fb-646b03169758",
  "title": "Властелин колец - книги против фильмов",
  "access": "public",
  "createdAt": "2024-06-19T11:28:25.471Z",
  "updatedAt": "2024-06-19T11:57:34.140Z",
  "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
  "rating": 0,
  "tags": [
    "фильмы",
    "книги",
    "Властелин Колец"
  ],
  "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66",
  "author": {
    "id": "854868f8-95f4-4b4e-8532-057a04f54a66",
    "name": "jason_statham",
    "role": "user",
    "email": "st1035@mail.ru",
    "password": "195a1a8c651abf9960b9a08d7b4c8349680e694d580aa0f0d9f104bce025e7ff",
    "createdAt": "2024-06-19T11:28:25.483Z",
    "updatedAt": "2024-06-19T11:42:41.323Z",
    "rating": 0
  }
},
{
  "message": "List of all posts"
}
```

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 834
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 11:59:36 GMT
etag: W/"342-7f13dy0f1chp02oVno5prb-d2jpc"
keep-alive: timeout=5
x-powered-by: Express
```

Как можно заметить – данный метод также выполнен успешно, так как мы авторизовались в системе.

Теперь закроем страницу со Swagger, заново на нее зайдем и попробуем выполнить какой-то метод, требующий авторизации. Такой метод должен выполняться успешно, так как пользователь должен быть все еще авторизованным из-за того, что сессионная кука живет 2 часа, а также наше приложение не перезапускалось.

Выберем метод **Get a user by id**, который требует авторизации, нажмем **Try it out**, введем **id** пользователя, которого мы создали, и нажмем **Execute**:

users

GET /api/users Get all users

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

Parameters

Cancel

Name	Description
id * required string(\$uuid) (path)	User ID. Example: 223d526d-5064-455e-9daf-6e7e3ad3e77d 854868f8-95f4-4b4e-8532-057a04f54a66

Execute

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/users/854868f8-95f4-4b4e-8532-057a04f54a66' \
-H 'accept: application/json'
```

Request URL

http://localhost:5000/api/users/854868f8-95f4-4b4e-8532-057a04f54a66

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{   "status": 200,   "data": {     "id": "854868f8-95f4-4b4e-8532-057a04f54a66",     "name": "jason_statham",     "role": "user",     "email": "st1035@mail.ru",     "password": "195a1ab0c51abf9960b9a08d7b4c8349680e694d580aa0f0d9f104bce025c7ff",     "createdAt": "2024-06-19T11:28:25.483Z",     "updatedAt": "2024-06-19T11:42:41.323Z",     "rating": 0,     "posts": [       {         "id": "dd3bdf0f-acbb-4d2f-95fb-646b03169758",         "title": "Властелин колец - книги против фильмов",         "access": "public",         "createdAt": "2024-06-19T11:28:25.471Z",         "updatedAt": "2024-06-19T11:57:34.146Z",         "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",         "rating": 0,         "tags": [           "фильмы",           "книжки",           "Властелин Колец"         ],         "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66"       }     ]   } }</pre></div><div>Download</div></div>

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 1280
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 12:05:36 GMT
etag: W/"500-f80e7Q2ppACgJdfJuaP7f1-gJMQ"
keep-alive: timeout=5
x-powered-by: Express
```

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/users/854868f8-95f4-4b4e-8532-057a04f54a66' \
-H 'accept: application/json'
```

Request URL

http://localhost:5000/api/users/854868f8-95f4-4b4e-8532-057a04f54a66

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{   "books": [     {       "id": "dd3bdf0f-acbb-4d2f-95fb-646b03169758",       "title": "Властелин Колец - книги против фильмов",       "access": "public",       "createdAt": "2024-06-19T11:28:25.471Z",       "updatedAt": "2024-06-19T11:57:34.146Z",       "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",       "rating": 0,       "tags": [         "фильмы",         "книжки",         "Властелин Колец"       ],       "authorId": "854868f8-95f4-4b4e-8532-057a04f54a66"     }   ],   "contacts": [     {       "id": "b949447f-4b43-45a5-8c58-ac41c2b3ce10",       "type": "telegram",       "value": "link to telegram",       "userId": "854868f8-95f4-4b4e-8532-057a04f54a66",       "createdAt": "2024-06-19T11:42:41.375Z",       "updatedAt": "2024-06-19T11:42:41.375Z"     },     {       "id": "452ed3de-b326-4710-bcc7-373851a887d7",       "type": "fb",       "value": "link to fb",       "userId": "854868f8-95f4-4b4e-8532-057a04f54a66",       "createdAt": "2024-06-19T11:42:41.375Z",       "updatedAt": "2024-06-19T11:42:41.375Z"     }   ],   "subscribers": [] },   "message": "User details" }</pre></div><div>Download</div></div>

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 1280
content-type: application/json; charset=utf-8
date: Wed, 19 Jun 2024 12:05:36 GMT
etag: W/"500-f80e7Q2ppACgJdfJuaP7f1-gJMQ"
keep-alive: timeout=5
x-powered-by: Express
```

Как можно заметить – все работает правильно, метод успешно выполнен, так как пользователь все еще авторизирован несмотря на то, что пользователь закрыл страницу и заново на нее зашел.

Теперь осталось проверить, что пользователь может выйти из аккаунта. Для этого выберем метод `/auth/logout` Logout of account, нажмем Try it out и после Execute:

auth

GET /auth/signin Login to account

POST /auth/signup Register a user

GET /auth/logout Logout of account

Parameters

No parameters

Execute

Responses

Code	Description	Links
200	Logout was successfully	No links
400	Bad request	No links
401	Unauthorized	No links
500	Internal server error	No links

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/auth/logout' \
  -H 'accept: */*'
```

Request URL

http://localhost:5000/auth/logout

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{   "status": 200,   "message": "Logged out successfully" }</pre></div><div>Download</div></div> <div><div>Response headers</div><div><pre>access-control-allow-origin: * content-length: 50 content-type: application/json; charset=utf-8 date: Wed, 19 Jun 2024 12:09:23 GMT etag: W/"32-uPULuqW0K0uQxSTd+XL8Q9CI6Q" x-powered-by: Express</pre></div></div>

Как можно заметить – метод выполнен успешно. Это значит, что пользователь вышел из своего аккаунта.

Теперь проверим, что пользователь действительно вышел из аккаунта. Для этого попробуем выполнить метод, требующий авторизацию. Данный метод – должен вернуть ответ 401, так как мы вышли из аккаунта.

Попытаемся выполнить метод `Get all posts`, требующий авторизацию. Выберем его, нажмем Try it out и после Execute:

posts

GET

/api/posts

Get all posts

Parameters

Cancel

Name	Description
search string (query)	Search string that matches post titles or text
	<input type="text" value="search"/>

Execute

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
401	Error: Unauthorized

Response body

Unauthorized

Download

Response headers

```
access-control-allow-origin: *
connection: keep-alive
content-length: 12
content-type: text/html; charset=utf-8
date: Wed, 19 Jun 2024 12:12:24 GMT
etag: W/"c-dAuJFQrdj53hezqDINGW/A01Yk"
keep-alive: timeout=5
x-powered-by: Express
```

Как можно заметить – мы получили ответ 401. Значит все работает правильно – мы вышли из системы с помощью /auth/logout и теперь authGuard не даст нам выполнять методы, требующие авторизации.