МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационные системы и технологии

Специальность 1-40 01 01 «Программное обеспечение информационных

технологий»

Специализация Программирование интернет приложений

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**К КУРСОВОМУ ПРОЕКТУ НА ТЕМУ:**

«Web-приложение «Социальная сеть Subscribers»»

Выполнил студент Демьянов Владислав Русланович

(Ф.И.О.)

Руководитель проекта Дубовик М.В.

(учен. степень, звание, должность, Ф.И.О., подпись)

Заведующий кафедрой к.т.н., доц. Смелов В.В .

(учен. степень, звание, должность, Ф.И.О., подпись)

Курсовой проект защищен с оценкой

Минск

**Содержание**

[Введение 4](#_Toc104118123)

[1 Постановка задачи 5](#_Toc104118124)

[1.1 Актуальность решаемой задачи 5](#_Toc104118125)

[1.2 Обзор и анализ прототипов 6](#_Toc104118126)

[1.3 Средства разработки 8](#_Toc104118127)

[2 Проектирование приложения 9](#_Toc104118128)

[2.1 Проектирование базы данных 9](#_Toc104118129)

[2.2 Разработка архитектуры приложения 10](#_Toc104118130)

[3 Руководство пользователя 16](#_Toc104118131)

[4 Тестирование приложения 21](#_Toc104118132)

[Заключение 24](#_Toc104118133)

[Список использованных источников 25](#_Toc104118134)

[ПРИЛОЖЕНИЕ A 26](#_Toc104118135)

[ПРИЛОЖЕНИЕ Б 27](#_Toc104118136)

# **Введение**

В пояснительной записке представлена спецификация Web-приложения «Социальная сеть Subscribers». Тема курсового проекта (социальная сеть), выбрана по причине высокой популярности и актуальности использования социальных сетей в современном мире. Это связано с тем, что социальные сети объединяют много функциональных возможностей: они служат средством для общения, знакомств, создания социальных отношений; являются источником новостей, обучающей информации и ещё многим другим. Социальная сеть «Subscribers» предоставляет лишь немногие из описанных выше возможностей. Основной функционал приложения: диалоги, посты, лайки, комментарии, поиск пользователей, подписки.

Далее приводится краткий список технологий, которые использовались при разработке приложения. Оно состоит из двух сайтов – клиент и сервер. Клиент – это Angular 13.1.1 приложение. Сервер – это NestJS 4.4 приложение. В качестве СУБД для проектирования базы данных была выбрана Microsoft SQL Server 2019. Для разработки приложения также использовались: TypeScript версии 4.5.2; ORM-технология Sequlize версии 4.28; IDE WebStorm, Socket.IO.

В процессе выполнения курсового проекты было выполнено:

* аналитический обзор литературы по теме проекта;
* изучение требование требований, определение вариантов использования;
* анализ и проектирование архитектуры приложения (построение диаграмм, проектирование бизнес-слоя, представления и данных);
* проектирование структуры базы данных;
* разработка дизайна пользовательского интерфейса;
* кодирование программного средства;
* тестирование и отладка программного средства;
* оформление пояснительной записки;
* сдача курсового проекта.

# **1 Постановка задачи**

## **1.1 Актуальность решаемой задачи**

Социальные сети характеризуются своей интерактивностью, связностью и пользовательским контентом. В современном обществе использование социальных сетей стало необходимой повседневной деятельностью. Социальные сети обычно используются для социального взаимодействия и доступа к новостям и информации, а также для принятия решений. Это ценный инструмент общения с другими людьми на местном и глобальном уровнях, а также для обмена, создания и распространения информации. Социальные сети могут влиять на решения потребителей о покупке с помощью обзоров, маркетинговой тактики и рекламы. По сути, социальные сети сильно влияют на нашу способность общаться, формировать отношения, получать доступ и распространять информацию, а также принимать наилучшее решение.

Одной из крупнейших отраслей, на которую повлияли социальные сети, являются новости и журналистика. Поскольку в настоящее время Интернет является самым быстрым и простым способом получения новостей, количество печатных СМИ сокращается, а издания вынуждены публиковать свои статьи в Интернете. Еще один эффект этой передовой технологии заключается в том, что теперь практически каждый может считать себя «журналистом» и публиковать новости, которые он считает точными и достоверными. Почти мгновенно мы можем создавать, делиться и распространять новости или сплетни по всему миру. Социальные сети только упростили эту задачу. Главные новости распространяются через Твиттер и другие сайты социальных сетей, и люди во всем мире узнают о них почти мгновенно. Сила социальных сетей позволила нашему обществу быть гораздо более осведомленным о мировых событиях и новостях.

Согласно отчёту Nielsen Social Media Report, у активных пользователей социальных сетей больше шансов подвергнуться влиянию в автономном режиме. Социальные сети теперь начинают влиять на процессы принятия решений пользователями. Постоянно публикуются обзоры и блоги с описанием колледжей, ресторанов, продуктов, компаний и т. д. Некоторые люди научились полагаться на эти обзоры при принятии решений. Например, Университет Джона Хопкинса создал «Hopkins Interactive», сайт социальной сети, созданный нынешними студентами для будущих студентов. На этом сайте потенциальные студенты могут увидеть неотредактированные версии опыта студентов в Хопкинсе. Потенциальные учащиеся посещают этот сайт, чтобы заглянуть за кулисы школы и определить, подходит ли им школа.

Кроме того, инструменты социальных сетей стали новым обязательным инструментом в мире маркетинга. Цифровой и интернет-маркетинг находятся на подъеме, и те, кто не представлен в социальных сетях, постепенно вымрут. Предприниматели и владельцы малого бизнеса начинают полагаться на сайты социальных сетей, чтобы распространять информацию о своих компаниях.  Интернет-маркетинг на данном этапе является почти синонимом социальных сетей, потому что он почти гарантированно достигает наибольшей аудитории и производит наибольший эффект. Будущее мира маркетинга обещает быть сформированным социальными сетями.

Возможности социальных сетей безграничны. Люди редко проводят день, не используя социальные сети и не ссылаясь на них. Независимо от того, используются ли они для общения, обучения или принятия решений, социальные сети останутся и будут продолжать влиять на наше общество.

## **1.2 Обзор и анализ прототипов**

Существуют общемировые социальные сети, которыми пользуются люди со всего мира, и локальные, в основном заточенные под одну страну: для России и СНГ это всем известные «Одноклассники», ВКонтакте и менее популярные сервисы Mail.ru. Среди общемировых уверенное лидерство держит Facebook.

Facebook. Эта соцсеть была основана Марком Цукербергом в 2004 году и в короткие сроки достигла огромной популярности. На сегодняшний день аудитория составляет более 2 млрд человек, то есть больше четверти населения планеты. Сайтом пользуются люди из разных стран мира. В России активные пользователи – журналисты, карьеристы, бизнесмены и те, кто по той или иной причине часто общается с иностранцами.

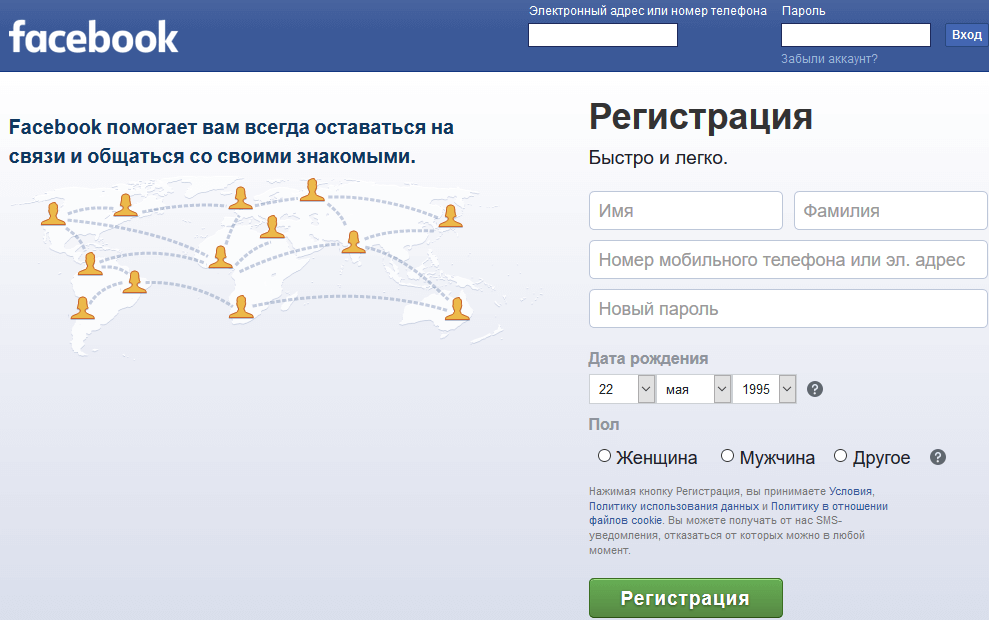


Рисунок 1.1 – Социальная сеть Facebook

Twitter. Изначально появился как сервис, позволяющий делиться со своей аудиторией короткими сообщениями: предполагалось, что это в первую очередь будут новости. Был запущен в 2006 году, сегодня популярен среди молодежи и используется преимущественно в творческих и развлекательных целях.



Рисунок 1.2 – Социальная сеть Twitter

Instagram. Относительно молодая соцсеть, запущенная в 2010 году и ориентированная на изображения. В 2012 ее приобрел Facebook. На сегодняшний день основная тематика площадки – мода, красота, путешествия, все, что связано с визуальным контентом. Instagram имеет крайне широкие коммерческие возможности: многие интернет-магазины существуют преимущественно как инстааккаунты.

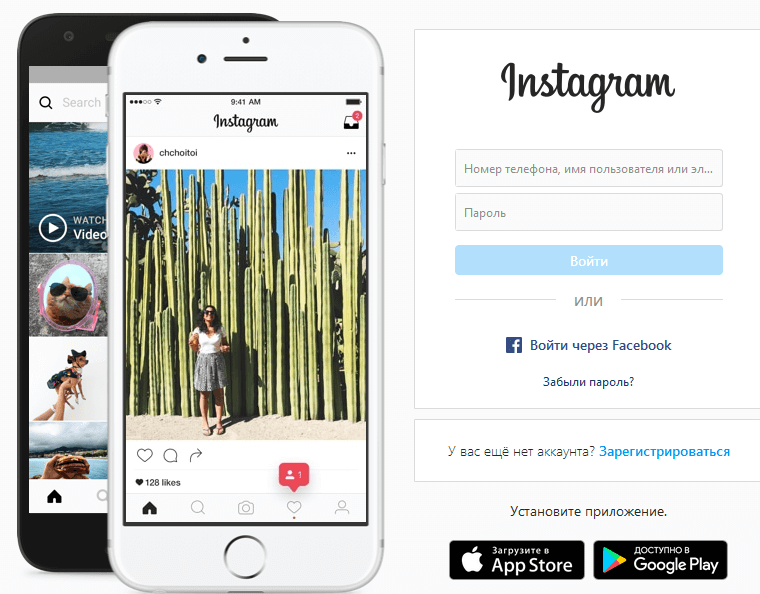


Рисунок 1.3 – Социальная сеть Instagram

## **1.3 Средства разработки**

При разработке приложения были использованы следующие технологии:

* JS фреймворк Angular 13.1.1;
* JS фреймворк NestJS 4.4;
* СУБД Microsoft SQL Server 2019;
* IDE WebStorm;
* Socket.IO;
* TypeScript 4.5.2;
* ORM Sequlize;
* Postman;
* Chrome DevTools.

Angular – это фреймворк от Google для создания SPA веб-приложений – на языках программирования TypeScript, JavaScript.

NestJS – это фреймворк для создания серверных приложений на базе NodeJS.

СУБД Microsoft SQL Server 2019 – система управления реляционными базами данных, разработанная корпорацией Microsoft. Основой используемый язык запросов – Transact-SQL. Transact-SQL является реализацией стандарта ANSI/ISO по структурированному языку запросов SQL с расширениями.

WebStorm – интегрированная среда разработки на JS, CSS и HTML.

SocketIO – библиотека JavaScript для веб-приложений и обмена данными в реальном времени. Использует протокол WebSocket, но, если нужно, использует и другие технологии, например, Flash Socket, AJAX Long Polling, AJAX Multipart Stream, предоставляя тот же интерфейс.

TypeScript – строго типизированный язык программирования, основанный и компилируемый на JavaScript.

Sequlize – это ORM-библиотека для приложений на Node.js, которая осуществляет сопоставление таблиц в базе данных и отношений между ними с классами.

Postman – это HTTP-клиент для тестирования API. HTTP-клиенты тестируют отправку запросов с клиента на сервер и получение ответа от сервера.

Chrome DevTools – это набор инструментов, встроенных в браузер Google Chrome, для создания и отладки сайтов. С их помощью можно просматривать исходный код сайта, отлаживать работу Front-End: HTML, CSS и JavaScript. DevTools позволяет проверять сетевой трафик, быстродействие сайта и многое другое.

# **2 Проектирование приложения**

## **2.1 Проектирование базы данных**

Для хранения данных в приложении используется реляционная база данных MS SQL Server.

Логическая схема базы данных, спроектированной в ходе разработки приложения представлена на рисунке 2.1.

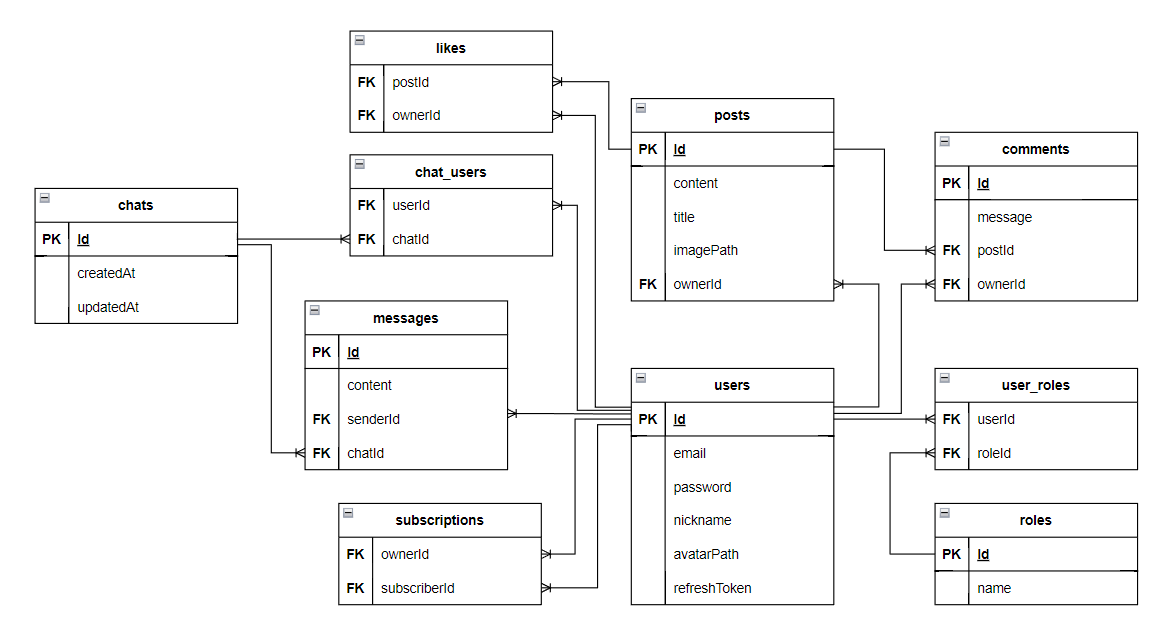


Рисунок 2.1 – Диаграмма базы данных

База данных состоит из 10 таблиц.

Центральной сущность в базе данных является пользователь (таблица users). Ей принадлежат сообщения (таблица messages, связь один ко многим), посты (таблица posts, связь один ко многим), комментарии (таблица comments, связь один ко многим), лайки (таблица likes, связь один ко многим), подписки (таблица subscriptions, связь один ко многим), чаты (таблица chats, связь многие ко многим) и роли (таблица roles, связь многие ко многим).

Пользователи могут оставлять комментарии под постом, т.е. у поста может быть много комментариев, но комментарий может принадлежать только одному посту. Поэтому была использована связь один ко многим. Точно такая же логика с лайками.

Пользователи могут создавать чаты между двумя или более пользователей, по этой причине таблицы chats и users имеют связь многие ко многим.

Пользователи могут отправлять сообщения в определённый чат. Для этого используется связь один ко многим.

## **2.2 Разработка архитектуры приложения**

Приложение состоит из трёх других – клиент, Http-сервер и WebSocket-сервер.

**Http-сервер.** Основное назначение http-сервера заключается в принятии http запроса от клиента, его маршрутизации, обработки и в конечном счёте отправки ответа клиенту. Для реализации http-сервера был выбран NodeJS фреймворк NestJS.

Его структура состоит из модулей. Модуль представляет собой класс с декоратором @Module(). Декоратор @Module() предоставляет метаданные, которые Nest использует для организации структуры приложения. Приложение должно иметь корневой модуль. Корневой модуль – это место, где Nest начинает упорядочивать дерево приложения. Модуль может импортировать и экспортировать компоненты, которые являются Provider (это всё что помечено декоратором @Injectable()), например – Service, Repository, Factory, Helper и т.д. Для этого модуль по умолчанию – сингелтон. Пример простейшего модуля представлен в листинге 2.1.

|  |
| --- |
| @Module({  **controllers**: [UsersController],  **providers**: [UsersService],  **imports**: [  SequelizeModule.*forFeature*([  User,  Role,  UserRoles,  Subscription,  ]),  RolesModule,  ***forwardRef***(() => AuthModule),  ChatsModule,  ],  **exports**: [UsersService], }) **export class** UsersModule { } |

Листинг 2.1 – Модуль Users

Как можно видеть в листинге 2.1, декоратор @Module принимает один объект, имеющий свойства providers, controllers, imports, exports.

Providers – провайдеры, которые будут инициализироваться инжектором Nest и которые могут быть общими, по крайней мере, в этом модуле.

Controllers – набор контроллеров, определенных в данном модуле, которые должны быть инициализированы.

Imports – список импортированных модулей, экспортирующих провайдеры, которые требуются в данном модуле.

Exports – подмножество providers, которые предоставляются этим модулем в других модулях, импортирующих этот модуль.

Как правило модуль состоит из следующих компонент: controllers и providers.

**Controllers** – классы помеченные декоратором @Controller(). Контроллер содержит маршруты или end-points, или как их ещё называют «Ручки». Они отвечают за обработку входящих запросов и возврат ответов клиенту. Пример контроллера представлен в листинге 2.2.

|  |
| --- |
| @Controller() **export class** ChatsController {   **private static** *basePath*: **string** = **'chats'**;   **constructor**(**private chatsService**: ChatsService) {  }   @Post(ChatsController.*basePath*)  @HttpCode(HttpStatus.*CREATED*)  **public async** create(@Body() chatCreateData: Chat): Promise<ChatDto> {  **return await this**.**chatsService**.create(chatCreateData);  }   @Get(**`/users/:userId/**${ ChatsController.*basePath* }**`**)  @HttpCode(HttpStatus.*OK*)  **public async** getUserChats(@Param(**'userId'**) userId: **number**): Promise<ChatDto[]> {  **return await this**.**chatsService**.getAllByUserId(***Number***(userId));  } } |

Листинг 2.2 – Контроллер Chats

В контроллере Chats представленном в листинге 2.2, есть два end-point для создания чата и получения чатов в которых состоит пользователь.

Первый end-point обрабатывает Post-запрос с телом, содержащим экземпляр класса Chat.

Второй end-point обрабатывает Get-запрос, содержащий в URL параметр с id пользователя.

Также в данный контроллер внедряется сервис ChatService, возможно это благодаря тому, что он является классом помеченным декоратором @Injectable().

**Providers** – это фундаментальная концепция в Nest. Многие из основных классов Nest могут рассматриваться как провайдеры – сервисы, репозитории, фабрики, хелперы и т.д. Основная идея провайдера заключается в том, что он может быть внедрён как зависимость; это означает, что объекты могут создавать различные отношения друг с другом, а функция подключения экземпляров объектов может быть в значительной степени делегирована системе выполнения Nest. Пример класса-провайдера представлен в листинге 2.3.

|  |
| --- |
| @Injectable() **export class** ChatsService {   **constructor**(@InjectModel(Chat) **private chatModel**: **typeof** Chat) {  }   **public async** create(chatCreateData: Chat): Promise<ChatDto> {  **const** chat = **await this**.**chatModel**.*create*(chatCreateData);  **if** (chatCreateData.**users**.**length** > 0) {  **await** chat.$set(**'users'**, chatCreateData.**users**.map(user => user.**id**));  }  **const** rawChat = **await this**.**chatModel**.*findByPk*(chat.**id**, {  **include**: [User, Message, { **model**: Message, **include**: [User] }],  });  **return** *parseChatToDto*(rawChat);  }   **public async** getAllByUserId(userId: **number**): Promise<ChatDto[]> {  **const** chats = **await this**.**chatModel**.*findAll*({  **include**: [User, Message, { **model**: Message, **include**: [User] }],  });  **return** chats  .filter(chat => chat.**users**.some(user => user.**id** === ***Number***(userId)))  .map(chat => *parseChatToDto*(chat));  } } |

Листинг 2.3 – Провайдер ChatService

Теперь можно добавить этот класс в массив providers, чтобы сообщить модулю, что он может использовать его для внедрения и в массив exports, чтобы экспортировать его при подключении модуля к другому модулю.

**Фильтры исключений.** Nest поставляется со встроенным слоем исключений, который отвечает за обработку всех необработанных исключений в приложении. Из коробки, это действие выполняется встроенным глобальным фильтром исключений, который обрабатывает исключения типа HttpException. В разработанном приложении как раз он и используется. Однако при необходимости можно реализовать пользовательский фильтр, для этого нужно создать класс, который реализует интерфейс ExceptionFilter. И затем использовать с помощью декоратора @UseFilters() локально, либо можно подключить его глобально с помощью метода useGlobalFilters().

**Валидация данных.** Для валидации данных в Nest можно использовать Pipes. Это классы, аннотированные декоратором @Injectable() и реализующие интерфейс PipeTransform. Как и с исключениями можно реализовать пользовательские Pipes или использовать встроенные.

**Guards.** Для защиты маршрутов в приложении используются Guards. Они представляют собой классы, аннотированными декоратором @Injectable() и реализующие интерфейс CanActivate. Пример Guard представлен в приложении А.

Guard должен реализовывать метод canActivate(), который возвращает true либо false. Если возвращается true – запрос будет обработан. Если возвращается false – запрос будет отклонён.

**Клиент.** Это приложение, написанное на Angular. Angular и Nest, если не брать во внимание то, что один используется для Front-end разработки, а другой для Back-end, очень похожи. Это связанно с тем, что разработчик Nest вдохновлялся архитектурой Angular.

В Angular архитектура также состоит из модулей и должен быть корневой модуль.

Чтобы класс стал модулем, его нужно пометить декоратором @NgModule().

Этот декоратор принимает объект со следующими свойствами:

* declarations – компоненты, директивы и фильтры корневого модуля;
* exports – компоненты, сервисы, директивы и фильтры, доступные другим модулям;
* imports – другие модули, используемые в корневом модуле;
* providers – сервисы приложения;
* bootstrap – имя главного компонента приложения.

**Компонент.** Часть интерфейса приложения с собственной логикой. Для того чтобы класс стал компонентом, нужно пометить его декоратором @Component(). Пример компонента представлен в листинге 2.4.

|  |
| --- |
| @Component({  **selector**: **'app-posts'**,  **templateUrl**: **'./posts.component.html'**,  **styleUrls**: [**'./posts.component.scss'**] }) **export class** PostsComponent {   **constructor**() { } } |

Листинг 2.4 – Компонент Posts

Этот декоратор принимает объект, который имеет следующие свойства:

* selector – название компонента;
* templateUrl или template – HTML-разметка в виде пути к файлу HTML или строки;
* providers – список сервисов, поставляющих данные для компонента;
* styles – массив путей к CSS-файлам, содержащим стили для создаваемого компонента.

Чтобы использовать созданный компонент нужно добавить его в declarations описанный ранее. И затем можно использовать его в качестве тега в другом компоненте (название тега соответствует имени в selector).

**Сервисы.** Они нужны для предоставления данных компонентам. Сервисы относятся к провайдерам, соответственно нужно пометить класс декоратором @Injectable(). Пример сервиса представлен в листинге 2.5.

|  |
| --- |
| @Injectable({  **providedIn**: **'root'** }) **export class** ChatService {   **constructor**(**private http**: HttpClient) { }   **public** getUserChats(userId: **number**): Observable<Chat[]> {  **return this**.**http**.get<Chat[]>(**`**${ ApiEndpoints.*Users* }**/**${ userId }**/chats`**);  }   **public** create(chatCreateData: Chat): Observable<Chat> {  **return this**.**http**.post<Chat>(**`**${ ApiEndpoints.*Chats* }**`**, chatCreateData);  } } |

Листинг 2.5 – Сервис Chat

Сервис представленный в листинге 2.5 служит для отправки запросов на сервер. Для этого он внедряет сервис HttpClient. Затем он используется в методе getUserChats (GET-запрос) и create (POST-запрос).

Методы, определённые в HttpClient являются асинхронными и возвращают Observable, который выдаёт запрошенные данные при получении ответа. Observable или наблюдаемый объект, позволяет подписаться на него и обрабатывать 0, 1 или более событий, которые производит этот объект. Причём можно отменить подписку. Это значит, что мы можем сделать запрос и, если вдруг нам он больше не нужен просто отменить его, не дожидаясь ответа. Это большое преимущество над теми же Promise.

**WebSocket-сервер.** Это приложение прослушивает порт TCP-сервера, который следует определённому протоколу. Он позволяет после handshake (рукопожатие) клиента с сервером установить соединение по протоколу WebSocket, которое работает поверх TCP. Соединение по протоколу WebSocket – это двунаправленное постоянное соединение.

В Nest для создания WebSocket сервера можно использовать шлюзы. Шлюз – это просто класс, помеченный декоратором @WebSocketGateway(). Так как шлюзы не зависят от платформы, они совместимы с любой библиотекой WebSockets после создания адаптера. В данном случае использовалась библиотека socket.io. Пример шлюза приведён в приложении Б.

Чтобы подписаться на какое-либо событие, которое будет инициировать клиент, нужно пометить метод декоратором @SubscribeMessage('EventName') и для извлечения тела сообщения пометить параметр метода декоратором @MessageBody. Но можно обойтись и без декоратора для получения тела запроса. Пример такого обработчика приведён в листинге 2.6.

|  |
| --- |
| @SubscribeMessage(**'sendPrivateMessageToServer'**) **public async** handlePrivateMessage(client: Socket, messageCreateData: Message): Promise<**void**> {  **const** message = **await this**.**messagesService**.create(messageCreateData);  **this**.**server** .in(messageCreateData.**chatId**.toString())  .emit(**'sendPrivateMessageToClient'**, message); } |

Листинг 2.6 – Метод обрабатывающий запрос от клиента

В методе из листинга 2.6 есть аргумент с типом Socket. Это экземпляр сокета конкретной платформы. Здесь он используется для генерации события конкретной группе клиентов, заданной в методе in.

# **3 Руководство пользователя**

При первом запуске приложения пользователь может только зарегистрироваться. Демонстрация регистрации нового пользователя представлена на рисунке 3.1.

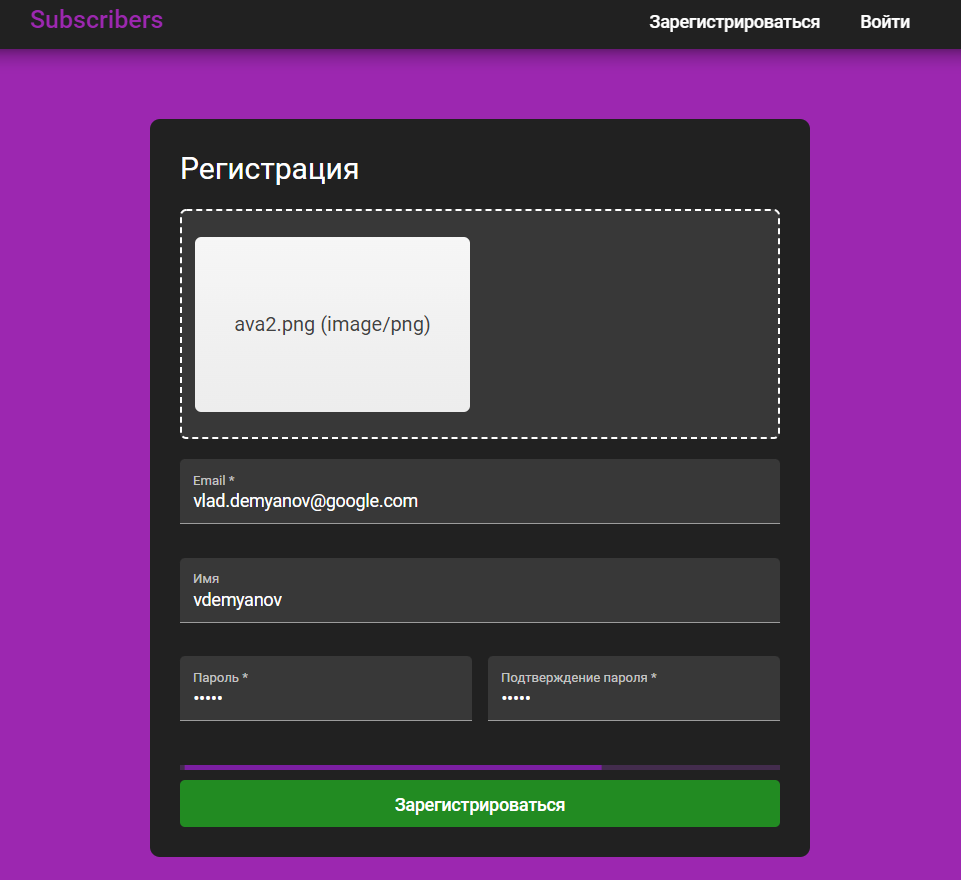


Рисунок 3.1 – Форма регистрации пользователя

После регистрации пользователь может войти в приложение. Форма входа в приложении представлена на рисунке 3.2.

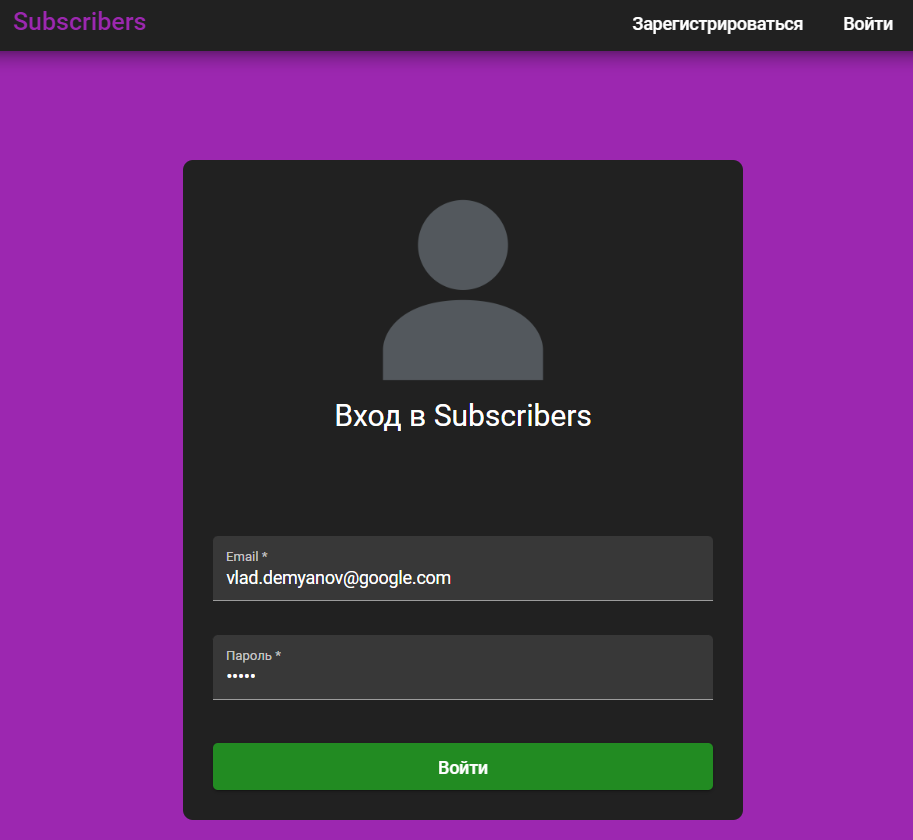


Рисунок 3.2 – Форма входа в приложение

После входа в приложение пользователь может создавать посты, подписываться на других пользователей, создавать чаты, лайкать посты, оставлять комментарии под постами. Далее на рисунке 3.3 представлен процесс создания нового поста.

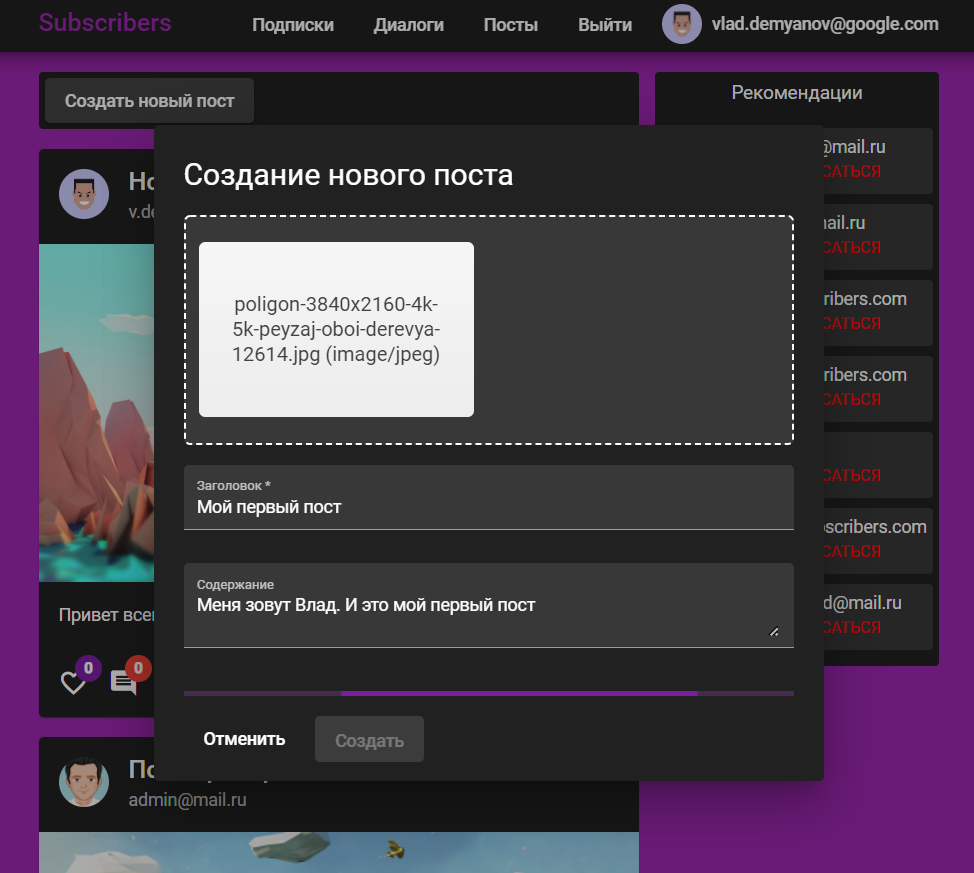


Рисунок 3.3 – Создание нового поста

После создания поста он появился в ленте постов. Это показано на рисунке 3.4.

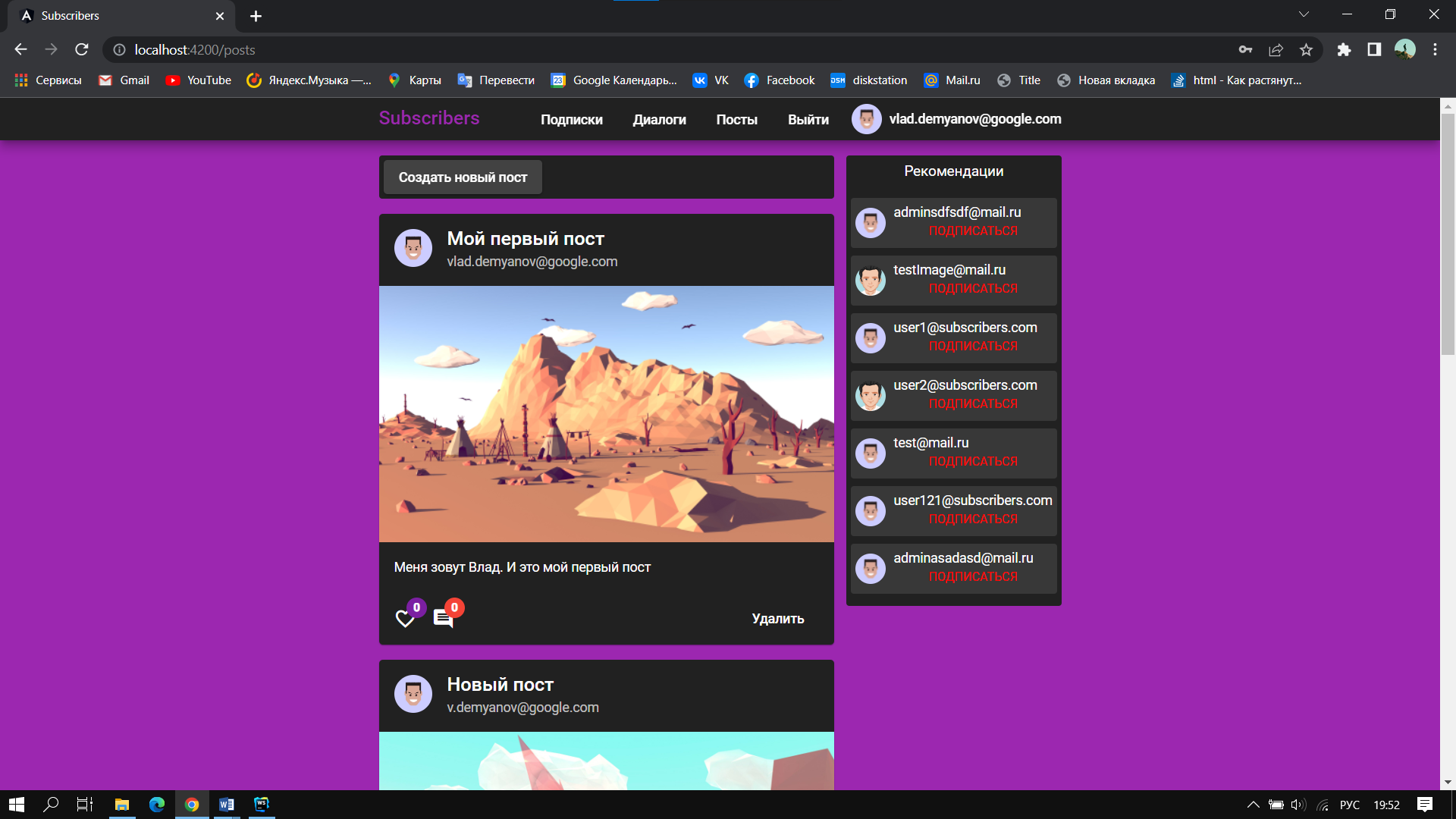


Рисунок 3.4 – Главная страница

Зарегистрированный пользователь, как уже было сказано выше, может подписываться на других пользователей. Как можно посмотреть свои подписки показано на рисунке 3.5.

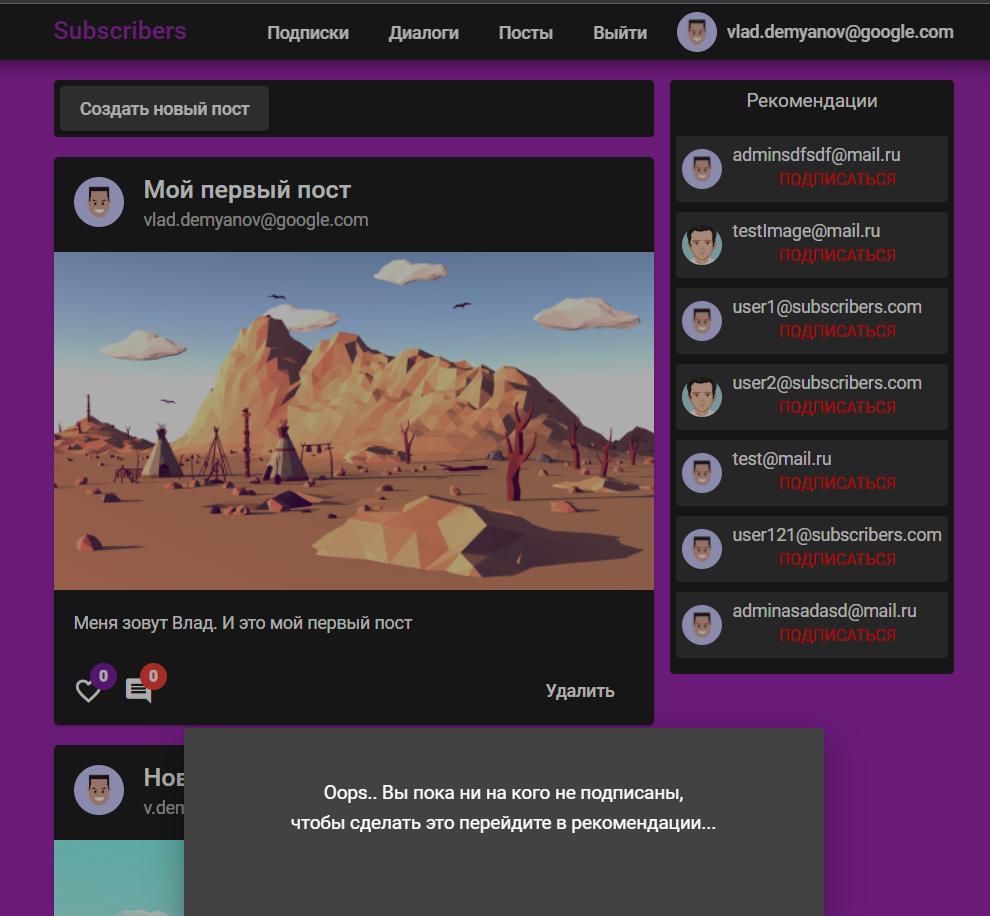


Рисунок 3.5 – Просмотр подписок

Так как пользователь ещё не подписан ни на кого, то ему отображается соответствующее сообщение. Чтобы подписаться на кого-нибудь нужно перейти либо в рекомендации, либо в диалоги, где можно найти конкретного пользователя. Процесс подписки показан на рисунке 3.6.

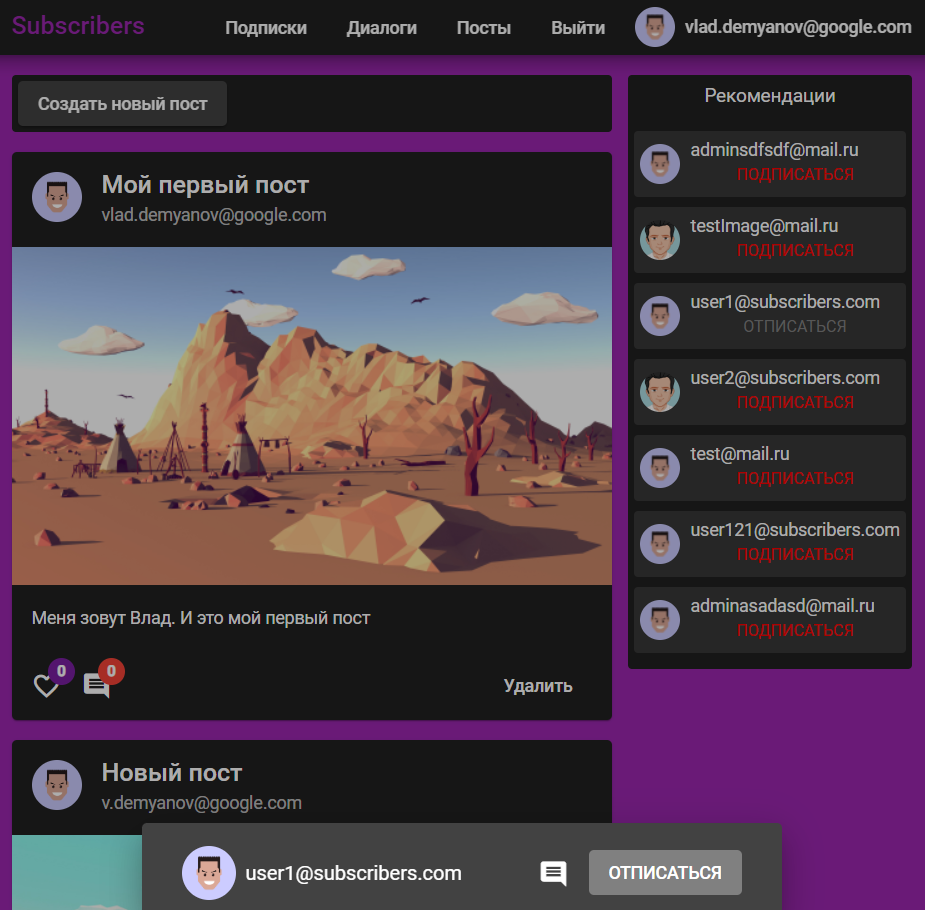


Рисунок 3.6 – Подписка на пользователя

Пользователь может поставить лайк и написать комментарий под постом. Это показано на рисунке 3.7.

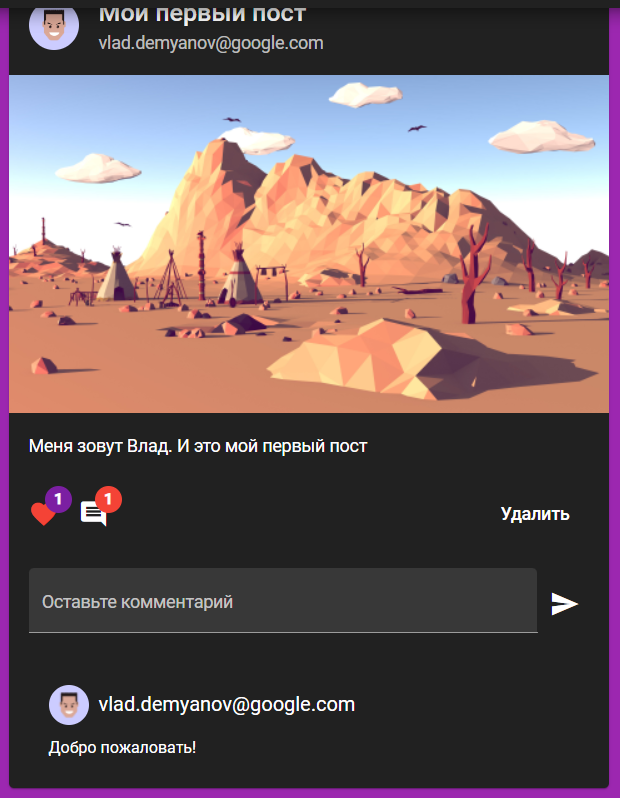


Рисунок 3.7 – Комментарии и лайки

В приложении есть страничка с диалогами. Она представлена на рисунке 3.8.

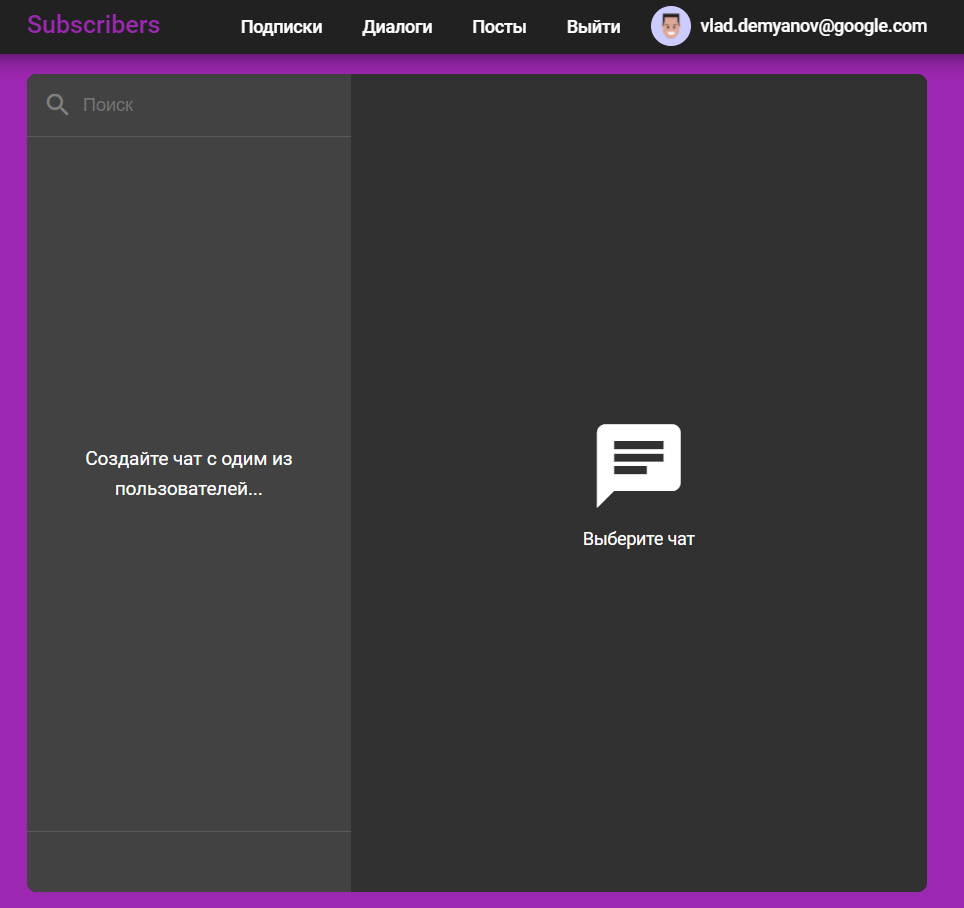


Рисунок 3.8 - Диалоги

Как можно заметить, пока у пользователя нет ни одного чата. Для того чтобы создать первый чат можно перейти в подписки либо ввести имя пользователя в поиск, и если система его найдёт, то нажать на иконку с чатом. Поиск представлен на рисунке 3.9.

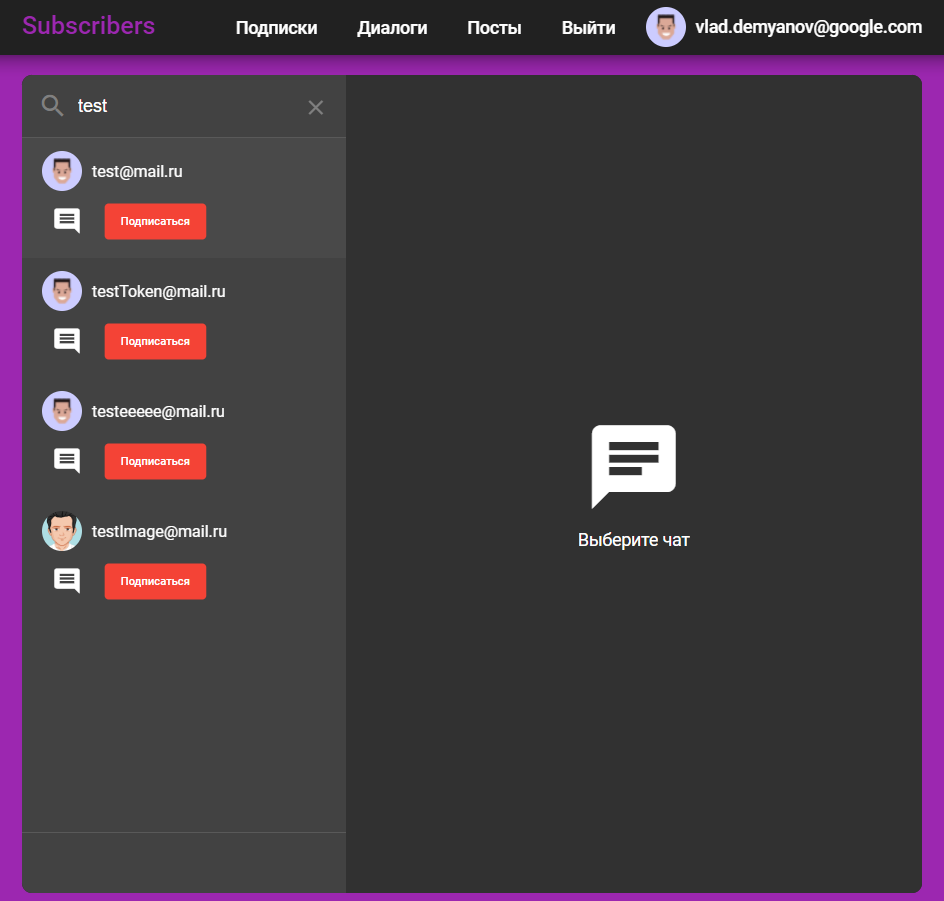


Рисунок 3.10 – Поиск пользователей

На рисунке 3.11 показан процесс переписки двух пользователей.

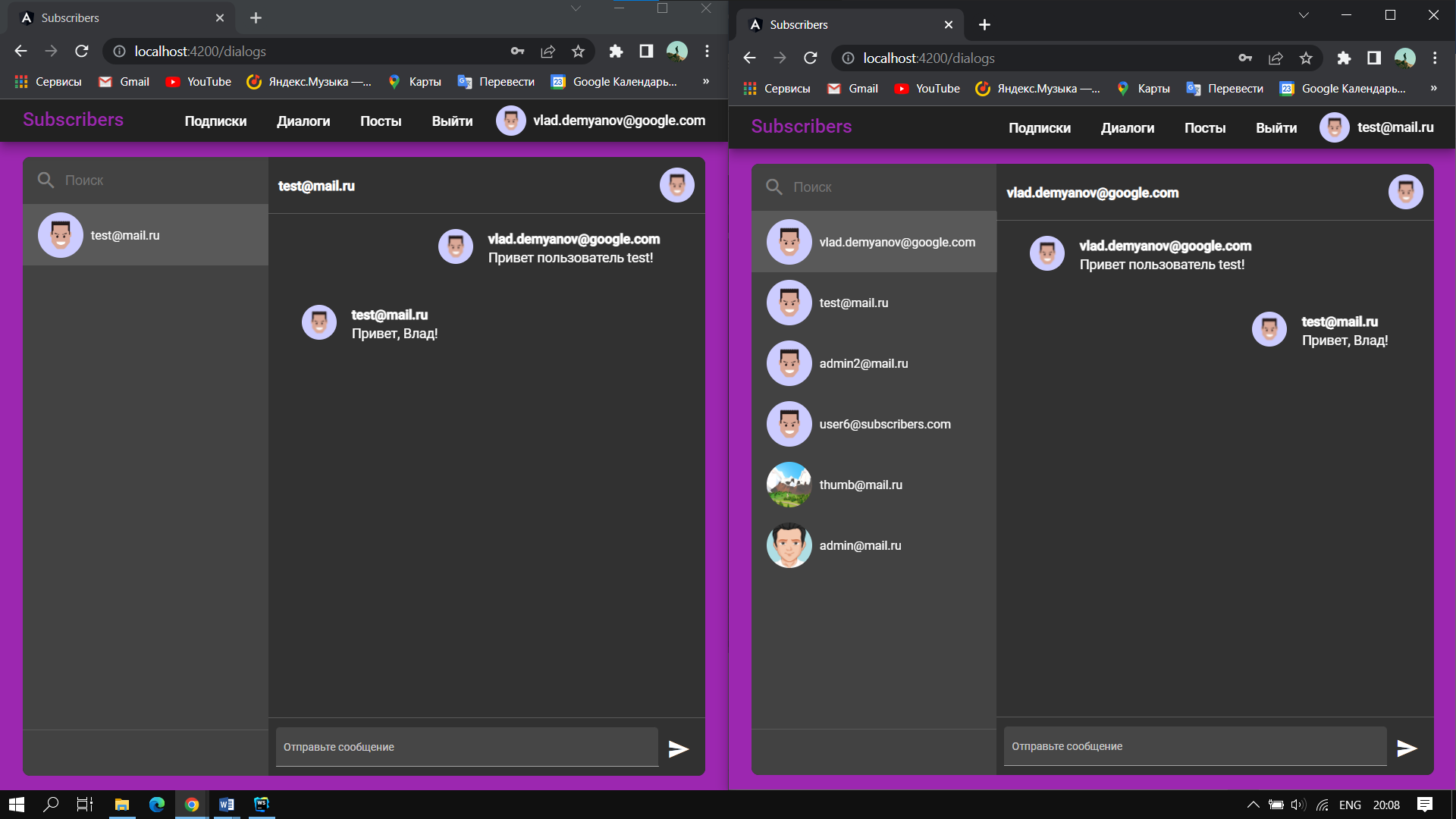


Рисунок 3.11 – Переписка двух пользователей

Последнее, что может делать пользователь – это удалять посты. Если это администратор, то он может удалять любые посты. Если просто пользователь, то только свои посты.

# **4 Тестирование приложения**

Сервер протестирован позитивными и негативными тестами с помощью Postman. Ниже приведено несколько запросов к API.

Первым будет позитивный тест удаления поста. Для того, чтобы он прошёл успешно нужно чтобы пользователь был аутентифицирован, и чтобы пост, который мы хотим удалить существовал. Для этого мы заранее создадим новый пост и возьмём валидный access-токен.

На рисунке 4.1 мы войдём в систему и скопируем полученный в ответе токен.

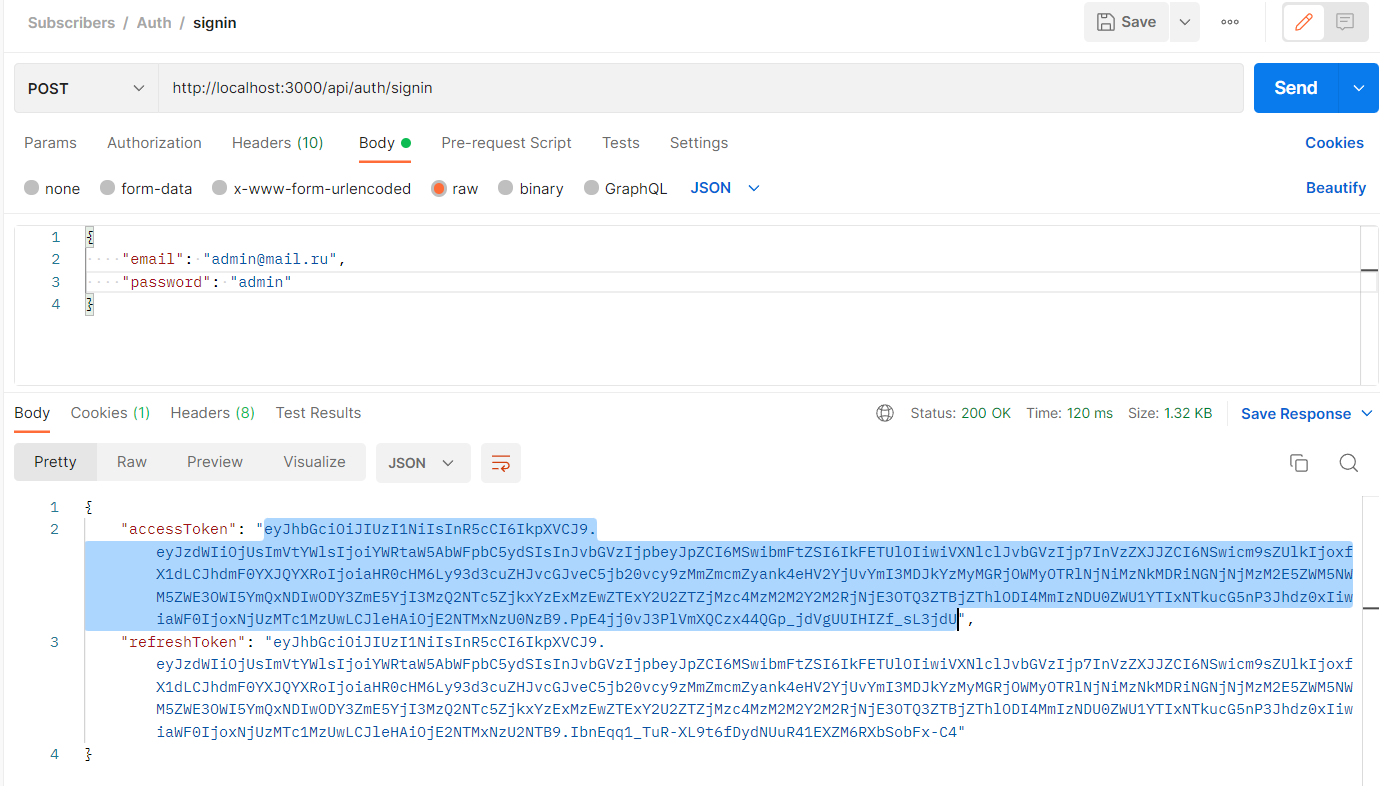


Рисунок 4.1 – Аутентификация в приложении

Далее на рисунке 4.2 мы создадим новый пост. Для его создания нам потребуется также access-токен.

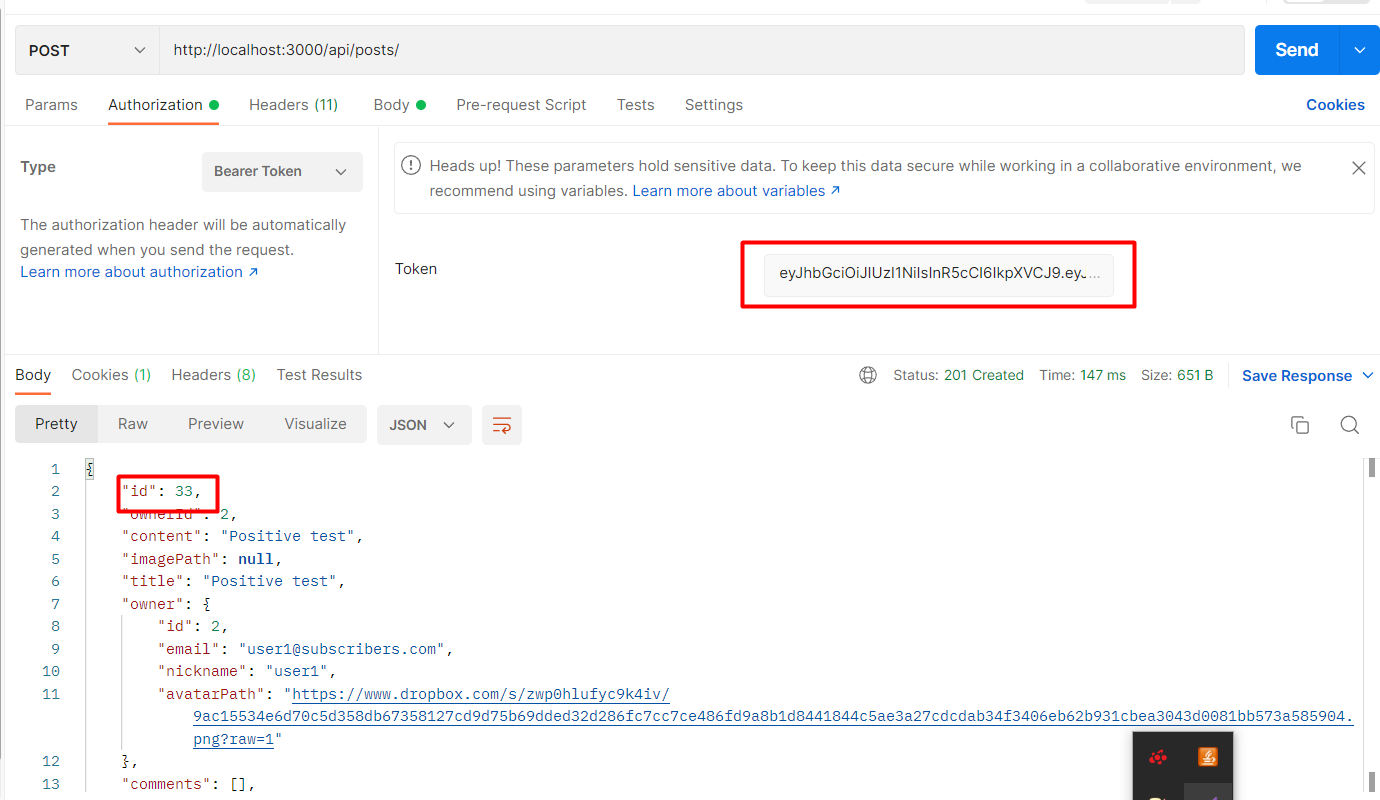


Рисунок 4.2 – Создание нового поста

И наконец удаление этого поста (рисунок 4.3).

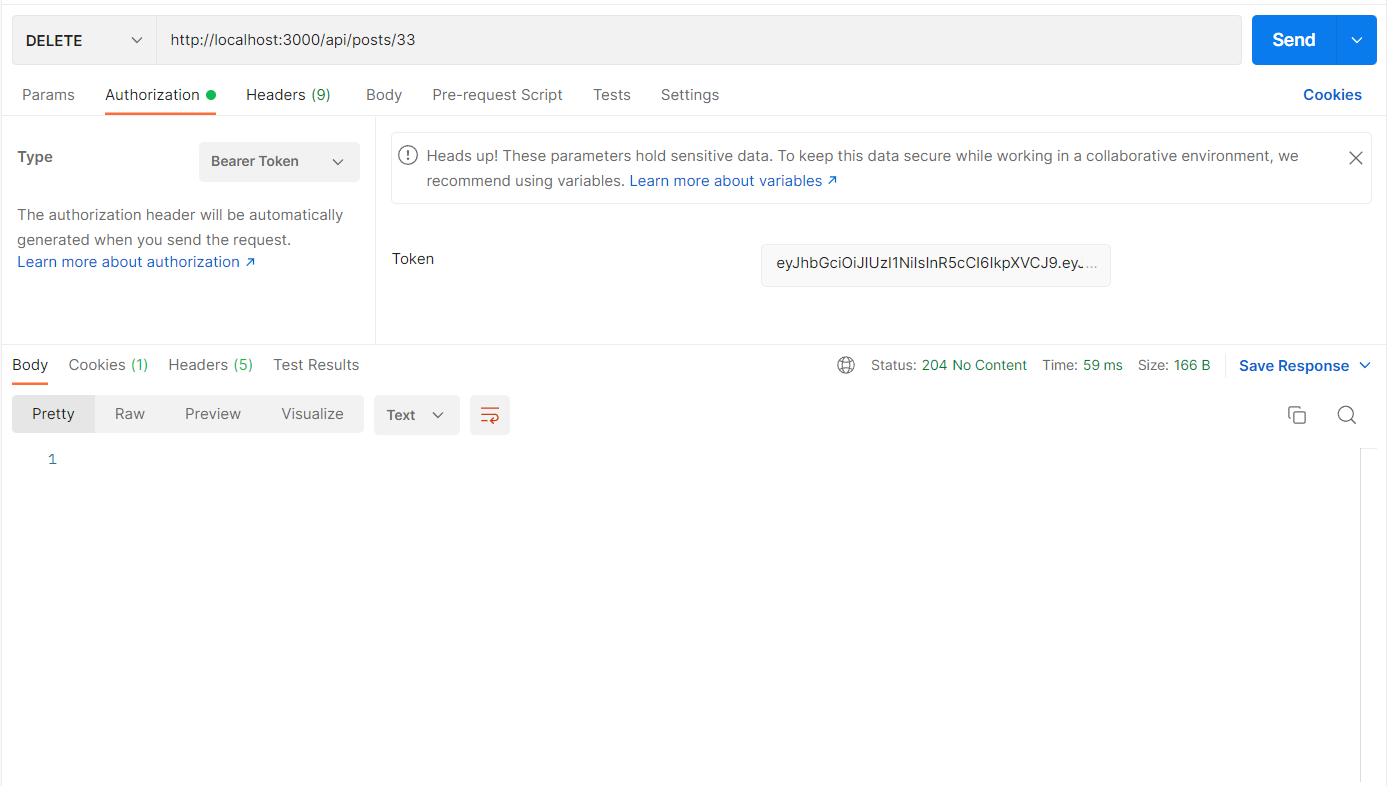


Рисунок 4.3 – Удаление поста

Второй тест – негативный. Попробуем удалить пост, которого не существует.

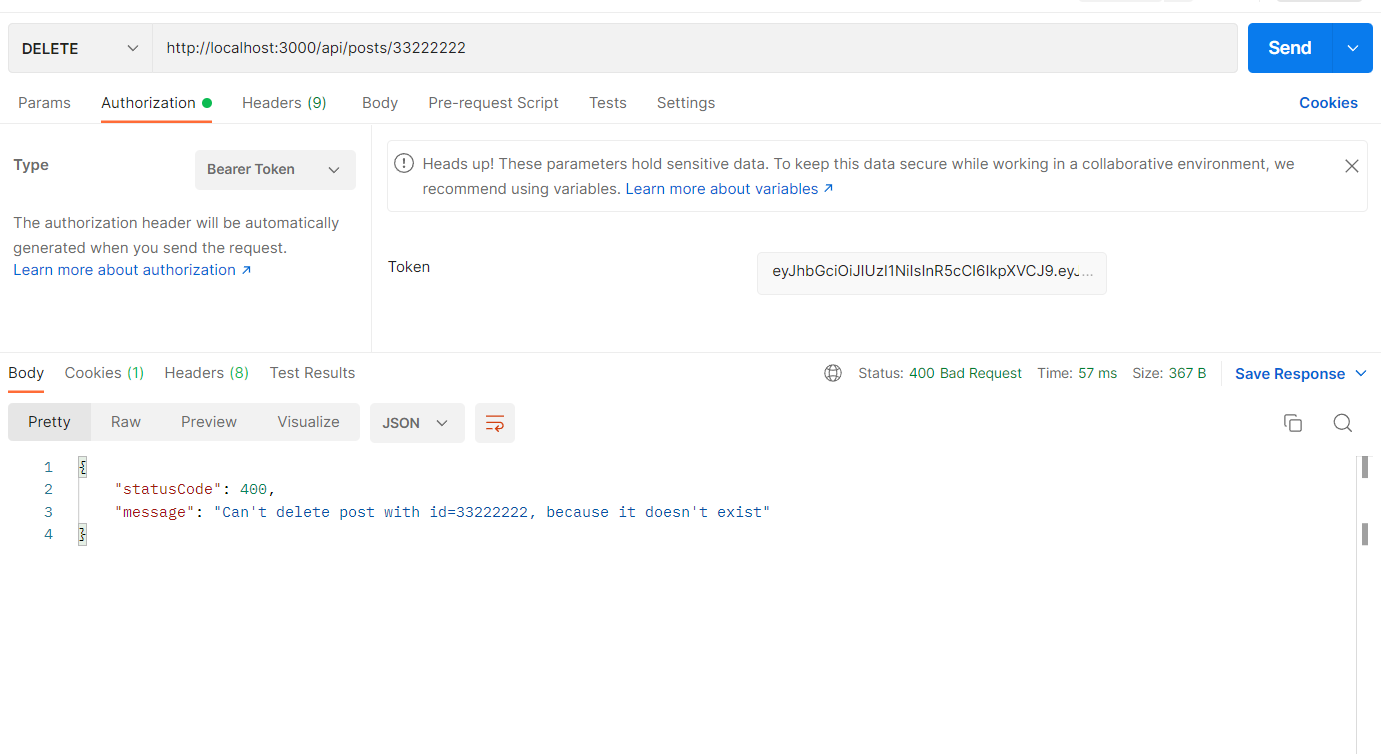


Рисунок 4.4 – Попытка удалить несуществующий пост

Теперь введём невалидный access-токен и сделаем запрос. Это отображено на рисунке 4.5. Кроме того у нас введён id несуществующего поста, но выдаст сообщение об ошибке связанное именно и аутентификацией, так запрос проходит в первую очередь через Guards.

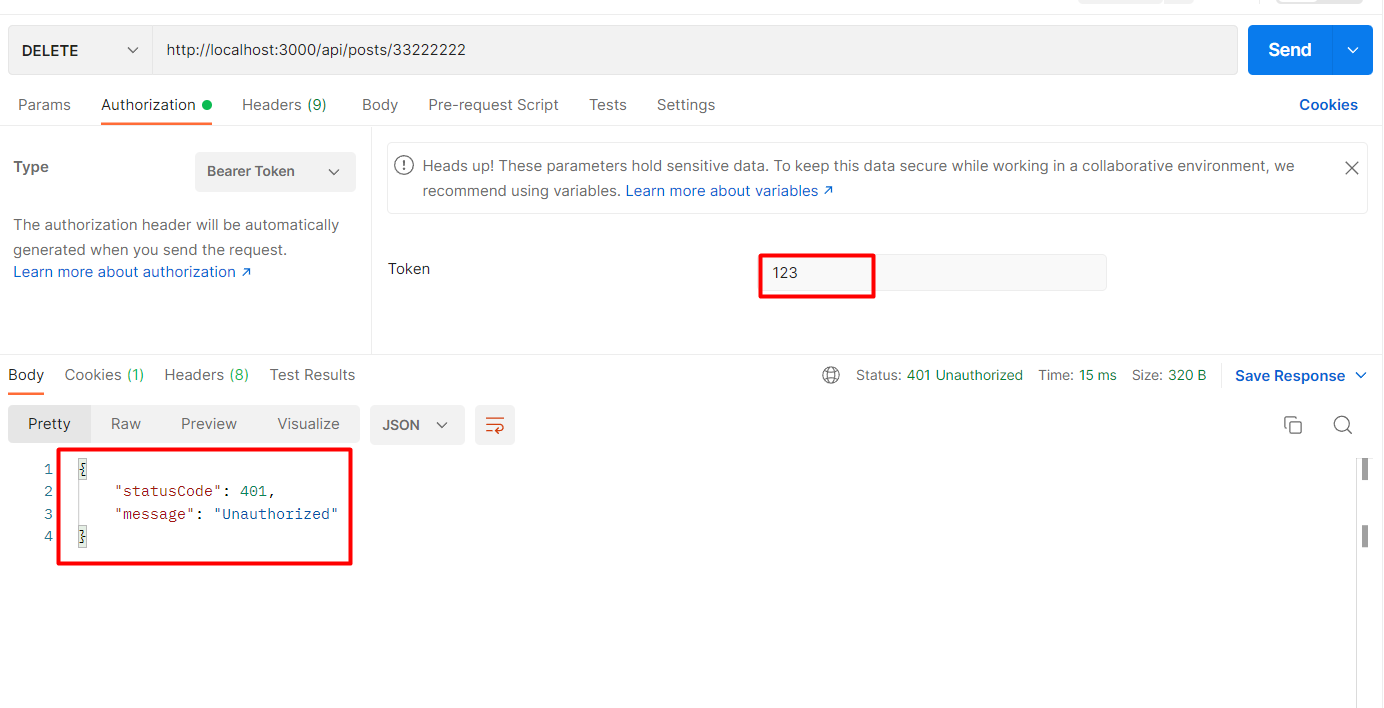


Рисунок 4.5 – Удаление поста с не валидным токеном

Как видно из рисунка 4.5 HTTP-сервер вернул сообщение об ошибке с кодом 401.

Ниже приведён пример тестирования валидации на стороне клиента. Сценарий такой: если пользователь не заполнил обязательные поля, то кнопка отправки формы должна быть заблокирована, и должно появляться соответствующее сообщение об ошибке. Это показано на рисунке 4.6.

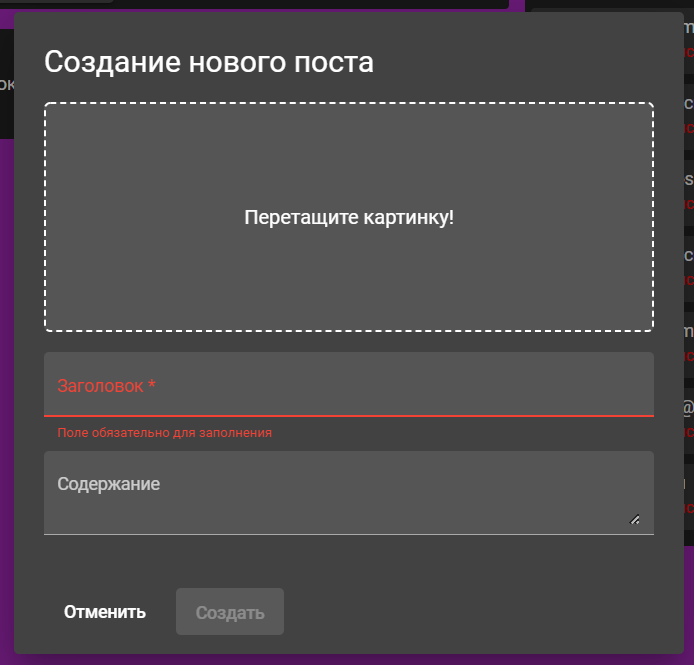
****

Рисунок 1 – Валидация формы для создания нового поста

# **Заключение**

В результате выполнения курсовой работы было разработано Web-приложение. Оно представляет собой социальную сеть.

В рамках работы над проектом был проведён обзор аналогичных решений, изучен теоретический материал, определён основной функционал приложения, выбран технологический стек, спроектирована архитектура приложения и структура базы данных, разработан дизайн интерфейса разработано руководство пользователя, проведено тестирование приложения.

В итоге приложение состоит из базы данных, которая содержит 10 связанных таблиц; http-сервера, который предоставляет API для пользователя; WebSocket-сервера, позволяющего создавать риалтайм соединение, создавать чаты между пользователями, обрабатывать событие отправки клиентом сообщения и генерировать событие отправки сообщения клиентам; клиента, который делает запросы к серверам, описанным выше и отображающий вытянутые данные.

Функционально приложение позволяет:

* аутентифицироваться и авторизироваться;
* создавать и удалять посты;
* добавлять посты в понравившиеся;
* оставлять комментарии под постами;
* подписываться и отписываться от пользователей;
* вести чаты между пользователями.

Исходя из полученных результатов можно сделать вывод, что приложение разработано в соответствии с техническим заданием, условия которого выполнены.

# **Список использованных источников**

1 NestJS – A progressive Node.js framework [Электронный ресурс]. – Режим доступа: <https://nestjs.com/> – Дата доступа: 22.05.2022.

2 Angular [Электронный ресурс]. – Режим доступа: <https://angular.io/> – Дата доступа: 22.05.2022.

3 TypeScript: JavaScript With Syntax For Types [Электронный ресурс]. – Режим доступа: <https://www.typescriptlang.org/> – Дата доступа: 22.05.2022.

4 Sequelize | Feature-rich ORM for modern TypeScript and Node.js ORM [Электронный ресурс]. – Режим доступа: <https://sequelize.org/> – 22.05.2022.

5 Socket.IO [Электронный ресурс]. – Режим доступа: <https://socket.io/> – Дата доступа: 22.05.2022.

6 Блинова Е.А. Курс лекций по базам данных / Е.А. Блинова

7 Смелов В.В. Курс лекций по программированию серверных кросс-платформенных приложений / Смелов В.В

8 METANIT.COM – Сайт о программировании [Электронный ресурс]. – Режим доступа: <https://metanit.com/> – Дата доступа: 22.05.2022.

9 Современный учебник JavaScript [Электронный ресурс]. – Режим доступа: <https://learn.javascript.ru/> – Дата доступа: 22.05.2022.

10 MDN Web Docs – Mozilla [Электронный ресурс]. – Режим доступа: <https://developer.mozilla.org/ru/> – Дата доступа: 22.05.2022.

# ПРИЛОЖЕНИЕ A

Guard для аутентификации по JWT токену

|  |
| --- |
| @Injectable() **export class** JwtAccessAuthGuard **extends *AuthGuard***(**'jwt'**) {   **constructor**(  **private reflector**: Reflector,  **private jwtService**: JwtService,  ) {  **super**();  }   **public async** canActivate(context: ExecutionContext): Promise<**boolean**> {  **const** isPublic = **this**.**reflector**.getAllAndOverride<**boolean**>(***IS\_PUBLIC\_KEY***, [  context.getHandler(),  context.getClass(),  ]);  **if** (isPublic) {  **return true**;  }   **const** requiredRoles = **this**.**reflector**.getAllAndOverride<RoleName[]>(***ROLES\_KEY***, [  context.getHandler(),  context.getClass(),  ]);  **const** canActivate = (**await super**.canActivate(context)) **as boolean**;  **if** (!requiredRoles || !canActivate) {  **return** canActivate;  }   **const** request = context.switchToHttp().getRequest();  **const** token = request.get(**'authorization'**)  .replace(**'Bearer'**, **''**)  .trim();  **const** payload = **this**.**jwtService**.verify(token, {  **secret**: ***process***.**env**.**AT\_SECRET\_KEY**,  }) **as** JwtPayload;   **const** areRolesValid = payload.**roles**.some(role => requiredRoles.includes(role.**name as** RoleName))  **return** areRolesValid && canActivate;  } } |

# ПРИЛОЖЕНИЕ Б

WebSocket-сервер

|  |
| --- |
| @WebSocketGateway(3003, {  **namespace**: **'ws'**,  **cors**: {  **origin**: **'\*'**,  }, }) **export class** AppGateway **implements** OnGatewayInit, OnGatewayConnection, OnGatewayDisconnect {   @WebSocketServer()  **public server**: Server;   **constructor**(**private messagesService**: MessagesService) {  }  @SubscribeMessage(**'sendPrivateMessageToServer'**)  **public async** handlePrivateMessage(client: Socket, messageCreateData: Message): Promise<**void**> {  **const** message = **await this**.**messagesService**.create(messageCreateData);  **this**.**server** .in(messageCreateData.**chatId**.toString())  .emit(**'sendPrivateMessageToClient'**, message);  }  @SubscribeMessage(**'joinPrivateRoom'**)  **public** handleJoinPrivateRoom(client: Socket, roomId: **number**): **void** {  client.join(roomId.toString());  }   @SubscribeMessage(**'leavePrivateRoom'**)  **public** handleLeavePrivateRoom(client: Socket, roomId: **number**): **void** {  client.leave(roomId.toString());  }  **public** afterInit(server: Server): **void** {  ***console***.log(**'Init WS Server'**);  }   **public** handleConnection(client: Socket, ...args: **any**[]): **void** {  ***console***.log(**`Client connected:** ${ client.**id** }**`**);  }   **public** handleDisconnect(client: Socket): **void** {  ***console***.log(**`Client disconnected:** ${ client.**id** }**`**);  } } |