МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий

Кафедра Информационных систем и технологий

Специальность 1-40 01 01 «Программное обеспечение информационных технологий»

Специализация 1-40 01 01 10 «Программное обеспечение информационных технологий (программирование интернет-приложений)»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовому проекту на тему:**

Web-приложение «Расписание транспорта»

Выполнил студент Коржова Валерия Сергеевна

(Ф.И.О.)

Руководитель проекта ст. препод. Дубовик М.В.

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

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

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

Консультанты ст. препод. Дубовик М.В.

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

Нормоконтролер ст. препод. Дубовик М.В.

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

Курсовой проект защищен с оценкой

Минск 2023

**Содержание**

[Введение 3](#_Toc134465384)

[1 Постановка задачи 4](#_Toc134465385)

[1.1 Обзор аналогов 4](#_Toc134465386)

[2 Проектирование web-приложения 6](#_Toc134465387)

[2.1 Диаграмма использования 6](#_Toc134465388)

[2.2 Диаграмма развертывания 7](#_Toc134465389)

[2.3 Проектирование базы данных 8](#_Toc134465390)

[3 Разработка web-приложения 11](#_Toc134465391)

[3.1 Разработка серверной части 11](#_Toc134465392)

[3.2 Разработка клиентской части 14](#_Toc134465393)

[4 Тестирование web-приложения 17](#_Toc134465394)

[5 Руководство пользователя 22](#_Toc134465395)

[5.1 Регистрация пользователя 22](#_Toc134465396)

[5.2 Аутентификация и авторизация пользователя 23](#_Toc134465397)

[5.3 Просмотр постов 24](#_Toc134465398)

[5.4 Действия с постами 25](#_Toc134465399)

[5.5 Администрирование 27](#_Toc134465400)

[Заключение 29](#_Toc134465401)

[Список используемых источников 30](#_Toc134465402)

[Приложение А 31](#_Toc134465403)

[Приложение Б 41](#_Toc134465404)

[Приложение В 42](#_Toc134465405)

# Введение

Главная задачей курсового проекта является разработка Web-приложения, позволяющего пользователю просматривать расписание городского транспорта и его маршрут. Также осуществлена возможность покупки проездных билетов разных типов на различные типы транспорта. Программное средство должно быть выполнено с использованием асинхронного программирования, взаимодействовать с базой данных. Отображение, бизнес логика и хранилище данных должны быть максимально независимы друг от друга для возможности расширения.

Для создания клиентской части был использован язык JavaScript и библиотека React.js.

Для серверной части приложения использовалась платформа Node.js с фреймворком Express, для работы с базой данных использовалась ORM Mongoose. Express - это фреймворк для Node.js, который предоставляет простой и удобный способ создания веб-приложений и API. Он позволяет быстро создавать серверные приложения на Node.js и предоставляет множество полезных функций и инструментов для работы с запросами, ответами, маршрутизацией, обработкой ошибок и другими задачами, связанными с веб-разработкой. Mongoose - это ORM для работы с MongoDB. Она позволяет удобно и эффективно создавать модели данных, выполнять запросы к базе данных и управлять взаимодействием между приложением и базой данных MongoDB. Mongoose предоставляет множество инструментов для работы с данными, включая валидацию, преобразование типов данных, хуки жизненного цикла и другие возможности, которые значительно упрощают и ускоряют разработку приложений, использующих MongoDB. В данном приложения использовалась СУБД MongoDB. [1]

Записка состоит из введения, 5 глав, заключения и приложений.

В первой главе рассматривается постановка задачи.

Во второй главе представлена разработка архитектуры приложения и структура курсового проекта.

В третьей главе описана разработка функциональной модели и модели данных программного средства.

Четвёртая глава состоит из руководства пользователя.

В пятой главе описано тестирование приложения.

В заключении описывается результат работы над курсовым проектом, выполненные цели.

В приложении содержится листинг кода некоторых классов и моделей.

1 Постановка задачи

Главная задача курсового проекта является разработка web-приложения, которое позволяет пользователю просматривать расписание городского транспорта. В данном курсовом проекте требовалось реализовать следующие задачи:

* регистрация и авторизация пользователя;
* изменение информации о пользователе;
* просматривание расписание транспорта;
* обмен данными между клиентом и сервером должен производиться по протоколу HTTPS;
* возможность добавления нового транспорта администратором;
* возможность покупки проездных билетов пользователем;
* получение уведомлений пользователем об истечении проездного билета;
* просмотр остановок и маршрутов на карте пользователем.

## 1.1 Обзор аналогов

Современные города предлагают множество вариантов общественного транспорта для удобства жителей и гостей города. Однако, часто бывает сложно следить за расписанием движения транспорта и планировать свой маршрут. В таких случаях, приложения для просмотра расписания транспорта становятся незаменимым инструментом для многих людей.

На рисунке 1.1 представлен внешний вид web-приложения «Minsktrans». Это приложение предназначено для просмотра расписания прибытия транспортных средств на остановки.

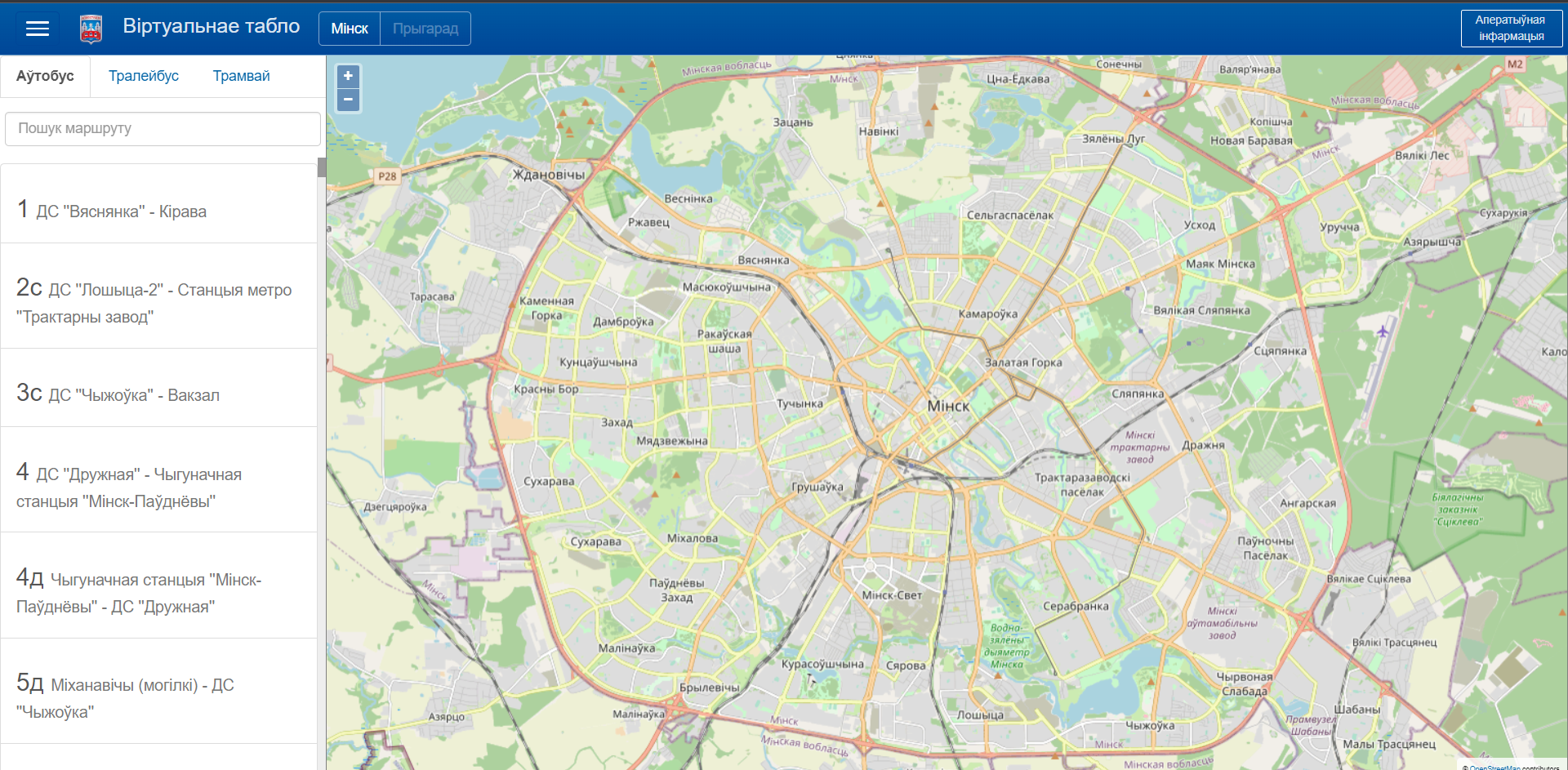


Рисунок 1.1 – Главная страница приложения «Minsktrans»

Главными преимуществами данного ресурса является актуальность расписания, хорошая визуализация маршрутов.

Из выявленных недостатков данного ресурса необходимо выделить то, что пользователь не может купить проездной билет.

Следующий аналог представлен на рисунке 1.2 – web-приложение «minsk.btrans».

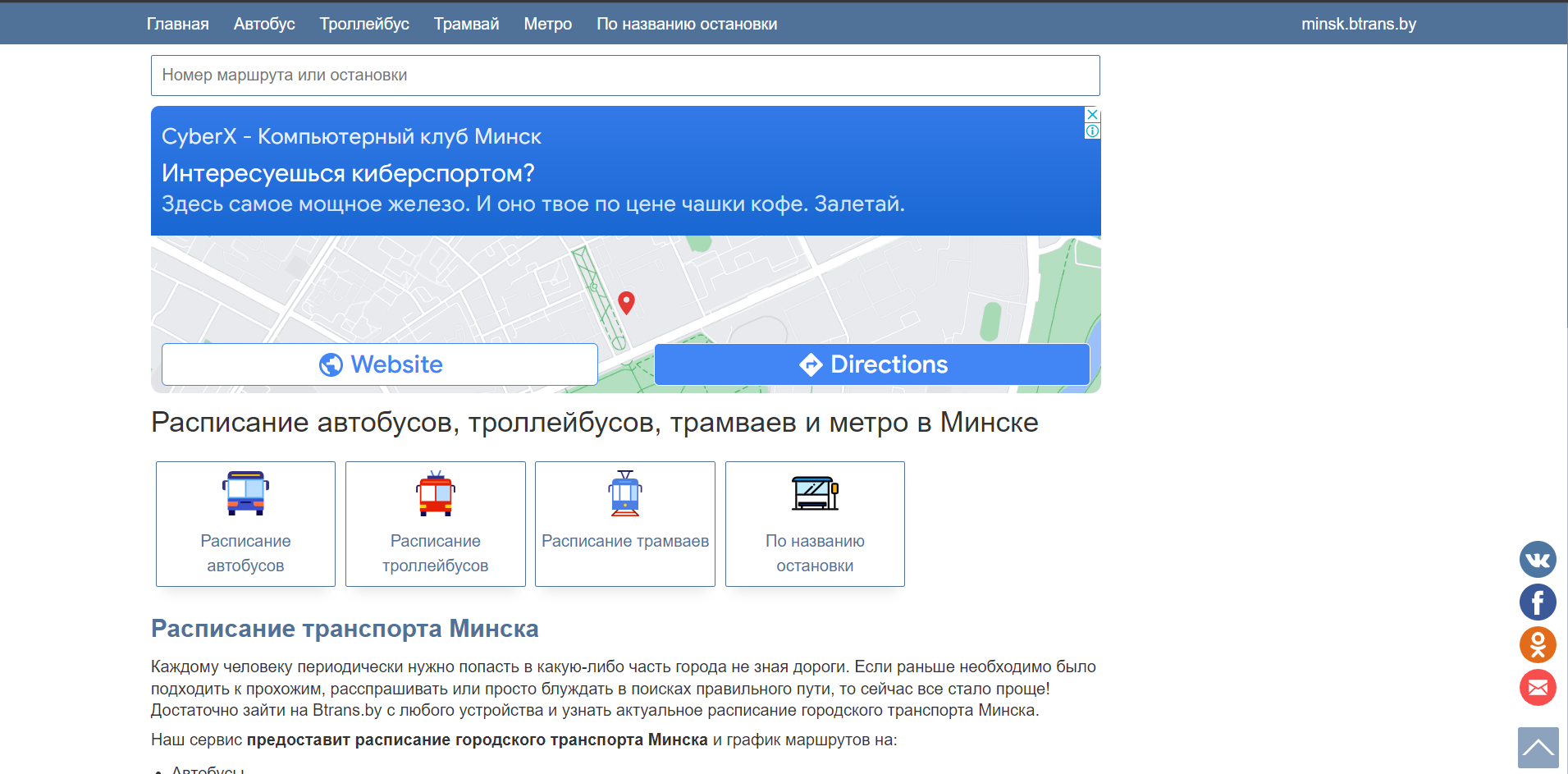


Рисунок 1.2 – Главная страница приложения «minsk.btrans»

Преимущества данного приложения заключается в просмотра расписания транспорта, наличие поиска остановок, фильтрация по типу транспорта.

Главным недостатком этого приложения является отсутствие карты с нанесенными на нее остановками.

2 Проектирование web-приложения

## 2.1 Диаграмма использования

Диаграмма использования в UML является графическим представлением набора элементов, связанных между собой в виде графа. Она используется для моделирования и визуализации вариантов использования (use cases) в системе. В диаграмме использования варианты использования представляются в виде овалов с их именами, а действующие лица (actors) изображаются в виде человечков, представляющих пользователей системы или внешние системы, взаимодействующие с системой.

Варианты использования могут быть связаны друг с другом различными типами связей. Обобщение (generalization) используется для представления отношения "является" между вариантами использования, где один вариант использования является более общим случаем другого. Расширение (extend relationship) используется, когда один вариант использования может быть расширен опциональными шагами другого варианта использования. Включение (include relationship) используется, когда один вариант использования включает другой вариант использования как обязательный шаг.

Кроме того, действующие лица также могут быть связаны между собой с помощью связей обобщения, что позволяет представить иерархию ролей или типов действующих лиц.

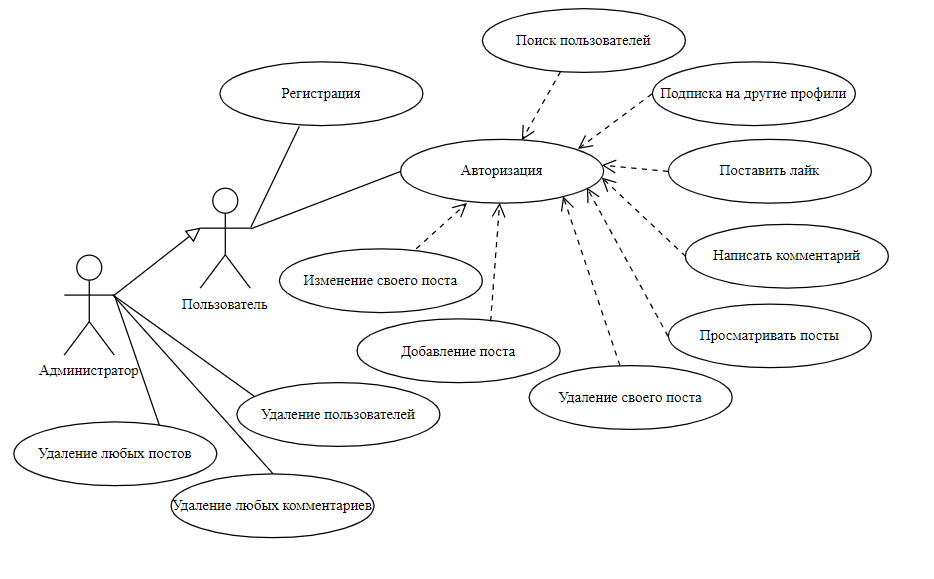
В целом, диаграмма использования UML предоставляет визуальное представление вариантов использования и их взаимосвязей, что помогает лучше понять и описать функциональные возможности системы и взаимодействие с ее пользователем.

Рисунок 2.1 – Диаграмма использования

Исходя из диаграммы использования можно выделить 2 основыные роли в приложении: пользователь и администратор, администратор владеет всеми возможностями пользователя, а также удаление постов, пользователей и комментариев.

## 2.2 Диаграмма развертывания

Диаграмма развертывания, также известная как диаграмма размещения, используется для визуализации конфигурации и топологии распределенной программной системы. Она представляет собой графическое представление, которое показывает, как компоненты системы размещены на отдельных узлах и как физические соединения устанавливаются между аппаратными устройствами, используемыми в системе.

Диаграмма развертывания является инструментом для моделирования элементов и компонентов программной системы, которые существуют только во время выполнения (в runtime). Она фокусируется на представлении компонентов-экземпляров программы, таких как исполняемые файлы или динамические библиотеки, которые активно используются во время работы системы. На диаграмме развертывания не показываются компоненты, которые не используются во время выполнения системы.

В результате, диаграмма развертывания предоставляет информацию о размещении компонентов системы на физических узлах, а также о связях между ними через физические сетевые соединения. Это помогает визуализировать и лучше понять организацию и взаимодействие компонентов системы на различных узлах, что полезно для архитекторов и разработчиков при проектировании и развертывании распределенных программных систем.

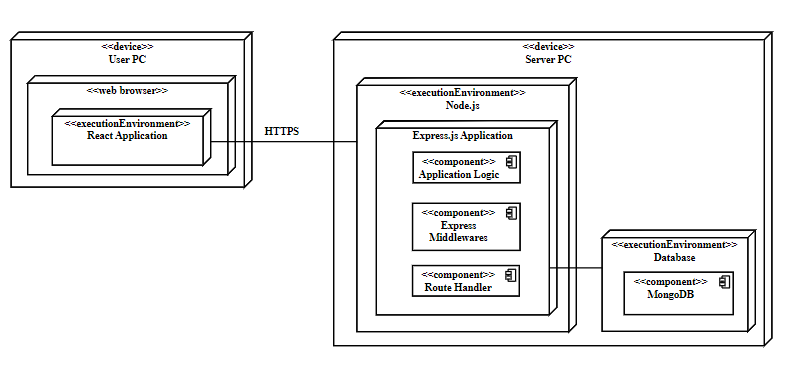


Рисунок 2.2 – Структурная схема приложения

## Приложение состоит из трех основных сервисов, которые обеспечивают его функциональность и взаимодействие с данными:

## 1. React.js - сервис, отвечающий за реализацию frontend-части приложения. Он обеспечивает создание пользовательского интерфейса, отображение данных и взаимодействие с пользователем на стороне клиента.

## 2. Node.js - сервис, реализующий backend-часть приложения. Он обрабатывает клиентские запросы, осуществляет бизнес-логику, взаимодействие с базой данных и другими сервисами. Node.js обеспечивает обработку запросов от клиента и генерацию ответов.

## 3. MongoDB - сервис, обеспечивающий доступ к данным. Это нереляционная база данных, которая хранит данные в формате JSON-подобных документов. MongoDB позволяет приложению хранить и извлекать данные, а также выполнять операции поиска, добавления, обновления и удаления данных.

## Каждый из этих сервисов в приложении работает независимо друг от друга и предоставляет определенный функционал. Они взаимодействуют между собой с помощью JSON, который используется для передачи данных между клиентской и серверной частями приложения. Клиентские запросы сначала обрабатываются Node.js сервером, затем React сервис генерирует соответствующее представление, которое передается обратно клиенту и отображается на странице веб-браузера.

## Такая архитектура приложения позволяет достичь высокой изолированности и гибкости, облегчая расширение и модификацию функционала приложения.

## 2.3 Проектирование базы данных

Важным аспектом при проектировании и создании базы данных является тщательный анализ предметной области приложения. Это позволяет разработать модель данных, которая точно отражает взаимодействие с данными в общем и соответствует требованиям приложения.

Корректный анализ предметной области, проектирование и разработка базы данных должны привести к созданию модели, которая предоставляет все необходимые функциональные возможности для пользователей. Это означает, что модель должна быть способна эффективно хранить, организовывать и обрабатывать данные, а также поддерживать необходимую бизнес-логику приложения.

Кроме того, грамотно разработанная модель данных должна быть гибкой и расширяемой, чтобы учитывать возможные изменения в предметной области и требования пользователя. Это позволит легко адаптировать базу данных к новым ситуациям и расширять ее функциональность, не нарушая целостность данных и эффективность работы системы.

Таким образом, правильный анализ, проектирование и разработка базы данных обеспечивают создание модели, которая соответствует потребностям пользователя и обеспечивает эффективное взаимодействие с данными. Модели базы данных для программного средства представлена на рисунке 2.3.



Рисунок 2.3 – Модель базы данных

Грамотно спроектированная модель базы данных является основой инфраструктуры базы данных и обеспечивает правильное взаимодействие пользовательских таблиц. Схема базы данных должна отображать связи между таблицами, чтобы корректно представлять их взаимосвязи. Одной из основных таблиц в базе данных является таблица "Posts", которая хранит информацию о постах. Эта таблица устанавливает связь один-ко-многим с таблицей "Comments", где хранятся комментарии, написанные пользователями. Также, таблица "Posts" связана с таблицей "Users", где содержится информация о пользователях.

В таблице "Users" хранятся все зарегистрированные пользователи, и их роли определяются через столбец "isadmin", который указывает, является ли пользователь администратором или нет.

Это упорядоченное описание основных таблиц базы данных, которые являются частью инфраструктуры и обеспечивают правильное взаимодействие между данными в системе. Описание структуры таблицы «Users» представлено в таблице 2.1.

Таблица 2.1 – Структура таблицы «Users»

|  |  |  |
| --- | --- | --- |
| Название столбца | Тип, ограничение целостности | Описание столбца |
| id | string, primary key | Идентификатор пользователя |
| username | string(25), unique | Логин пользователя |
| email | string | Почтовый адрес пользователя |
| password | string | Пароль |
| fullname | string | Имя пользователя |
| isadmin | boolean, default:0 | Идентификатор роли |
| story | string(200) | Текст к профилю |
| avatar | string | Фото пользователя |
| createdate | data | Дата создания |
| updatedate | data | Дата последнего изменения |

В таблице «Posts» хранится информация о посте. Описание структуры таблицы «Posts» представлено в таблице 2.2.

Таблица 2.2 – Структура таблицы «Posts»

|  |  |  |
| --- | --- | --- |
| Название столбца | Тип, ограничение целостности | Описание столбца |
| id | string, primary key | Идентификатор поста |
| user\_id | string, foreign key | Идентификатор пользователя |
| comment\_id | string, foreign key | Идентификатор комментария |
| content | string | Текст поста |
| images | array | Фото поста |
| likes | string | Количество лайков |
| createdate | data | Дата создания |
| updatedate | data | Дата последнего изменения |

Таблица «Followers» содержит всех подписчиков. Её структура представлена в таблице 2.3.

Таблица 2.3 – Структура таблицы «Friends»

|  |  |  |
| --- | --- | --- |
| Название столбца | Тип, ограничение целостности | Описание столбца |
| id | string, primary key | Идентификатор таблицы |
| user\_id | string, foreign key | Идентификатор пользователя |
| friend\_id | string, foreign key | Идентификатор подписки |

В таблице «Comments» хранятся все комментарии к постам. Описание структуры таблицы «Comments» представлено в таблице 2.4.

Таблица 2.4 – Структура таблицы «Comments»

|  |  |  |
| --- | --- | --- |
| Название столбца | Тип, ограничение целостности | Описание столбца |
| id | string, primary key | Идентификатор таблицы |
| user\_id | string, foreign key | Идентификатор пользователя |
| content | string | Текст комментария |
| createdate | data | Дата создания |
| updatedate | data | Дата изменения |

Таким образом при проектировании базы данных используется 4 таблицы и связь один-ко-многим.

3 Разработка web-приложения

## В процессе выполнения курсового проекта было создано веб-приложение, которое использует архитектурный стиль REST API. Архитектура приложения разделена на два компонента: серверная часть, называемая "backside", и клиентское представление, называемое "frontside".

## Компонент "backside" отвечает за обработку запросов от клиентов, выполнение бизнес-логики и взаимодействие с базой данных или другими внешними сервисами. Он предоставляет REST API, который клиенты могут использовать для отправки запросов и получения данных от сервера.

## Компонент "frontside" отвечает за представление данных и взаимодействие с пользователем. Он обычно реализован с использованием HTML, CSS и JavaScript, и отображает пользовательский интерфейс, с которым пользователи могут взаимодействовать. "frontside" использует REST API, предоставленный "backside", для получения данных от сервера и отправки запросов.

## Такая архитектура позволяет разделить ответственность между серверной и клиентской частями приложения, обеспечивает модульность и удобство разработки. REST API обеспечивает стандартизированный способ взаимодействия между компонентами и позволяет создавать гибкое и масштабируемое веб-приложение.

## 3.1 Разработка серверной части

На рисунке 3.1 отображена структура серверной части разрабатываемого программного средства, включающая в себя директории, конфигурационные файлы и файлы скриптов.

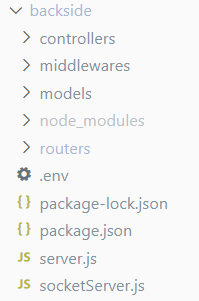


Рисунок 3.1 – Структура проекта сервера

Входной точкой в приложение является файл server.js в нем запускается прослушивание запросов, посылаемых приложению работающему на определенном порту. Веб-сервер, подключает необходимые для работы сервера модули, выполняется начальная инициализация приложения и регистрация маршрутизаторов. Также в server.js идёт подключение к базе данных.

В директории «controllers» содержатся контроллеры. Контроллеры – это файлы, которые содержат основную маршрутизатор и сопоставляемую логику приложения. Контроллеры взаимодействуют с базой данных через модели. Пример одного из методов контроллера приведен в листинге 3.1. Другие примеры контроллеров находятся в Приложении А.

friend: async (req, res) => {

*try* {

            const existingFriend = *await* Friend.find({ friend: req.body.user,user: req.body.friend});

*if* (existingFriend.length >0) {

*return* res.status(400).json({ msg: 'Friendship already exists' });

            } *else* {

                const friend = new Friend({friend: req.body.user,user: req.body.friend,});

*await* friend.save();

                res.json({ msg: "Friend added" });

            }

        } *catch* (err) {

*return* res.status(500).json({ msg: err.message });

        }

},

Листинг 3.1 – Метод из контроллера friendCtrl

Директория «models» содержит файлы с классами-сущностями базы данных. Приложение использует ORM-библиотеку «Mongoose» для взаимодействия с базой данных. Данные, которые используются в «Mongoose», описываются определенной схемой по которой будет создана модель. Пример схемы поста приведён в листинге 3.2. Другие схемы находятся в Приложении Б.

const mongoose =require('mongoose');

const postSchema =mongoose.Schema({

    content: String,

    images :{

        type:Array,

        default:[]

    },

    likes:[{type: mongoose.Types.ObjectId, ref:'user'}],

    comments:[{type: mongoose.Types.ObjectId, ref:'comment'}],

    user:[{type: mongoose.Types.ObjectId, ref:'user'}],

},{

    timestamps:true

})

module.exports=mongoose.model('post',postSchema);

Листинг 3.2 – Схема поста

Директория «middlewares» содержит файл, реализующий функцию auth (листинг 3.3), которая используется для аутентификации пользователей.

const auth =async(req,res,next)=>{

*try* {

        const token =req.header('Authorization')

*if*(!token) *return* res.status(500).json({msg:'Not Valid'})

        const decoded =jwt.verify(token, process.env.ACCESSTOKENSECRET)

*if*(!decoded) *return* res.status(500).json({msg:'Not Valid'})

        const user = *await* Users.findOne({\_id:decoded.id})

        req.user =user;

        next();

    } *catch* (error) {

*return* res.status(500).json({msg:error.message})

    }

}

module.exports = auth;

Листинг 3.3 – Функция аунтификации

Эта функция извлекает токен из заголовка Authorization запроса, декодирует его с помощью jwt.verify() и получает объект пользователя, связанный с этим токеном. Если все эти действия прошли успешно, функция вызывает next(), чтобы передать управление следующей middleware-функции в цепочке обработки запросов. Если что-то идет не так (например, токен недействителен), функция отправляет ответ с кодом состояния 500 и сообщением об ошибке в формате JSON.

Директория «routes» содержит файлы маршрутизации. Пример одного из файлов, а именно маршруты для контроллера postCtrl, приведен в листинге 3.4. Другие примеры контроллеров находятся в Приложении В.

const router = require('express').Router()

const auth = require('../middlewares/auth')

const postCtrl =require('../controllers/postCtrl')

router.post('/posts',auth,postCtrl.createPost)

router.get('/posts',auth,postCtrl.getPost)

router.get('/posts/all',auth,postCtrl.allPost)

router.get('/posts/:id',auth,postCtrl.userPost)

router.patch('/post/:id',auth,postCtrl.updatePost)

router.patch('/post/:id/like',auth,postCtrl.likePost)

router.patch('/post/:id/dislike',auth,postCtrl.dislikePost)

router.delete('/posts/:id',auth,postCtrl.deletePost)

router.patch('/post/:id/comm',auth,postCtrl.delcommupdatePost)

module.exports = router;

Листинг 3.4 – Маршруты контроллера postCtrl

Файл socketServer.js содержит реализацию технологии web-sockets. А именно оставление комментариев под постами в реальном времени. Пример использования технологии web-sockets приведён в листинге 3.5.

const socketio = require('socket.io');

const socketServer = (server) => {

    const io = socketio(server);

    const connectedUsers = {};

    io.on('connection', (socket) => {

        console.log(`User connected: ${socket.id}`);

        const userId = socket.handshake.auth.userId;

        let socketId = connectedUsers[userId];

*if* (socketId) {

            socket.join(`user\_${userId}`);

        } *else* {

            socketId = socket.id;

            connectedUsers[userId] = socketId;

            socket.join(`user\_${userId}`);

        }

        socket.on('new-comment', (comment) => {

            console.log(`New comment: ${JSON.stringify(comment.content)}`);

            io.to(`user\_${comment.userId}`).emit('new-comment', comment);

        });

        socket.on('disconnect', () => {

            console.log(`User disconnected: ${socket.id}`);

            delete connectedUsers[userId];

        });

    });

};

module.exports = socketServer;

Листинг 3.5 – Пример использования технологии web-sockets

Протокол WebSocket («веб-сокет»), описанный в спецификации [RFC 6455](http://tools.ietf.org/html/rfc6455), обеспечивает возможность обмена данными между браузером и сервером через постоянное соединение. Данные передаются по нему в обоих направлениях в виде «пакетов», без разрыва соединения и дополнительных HTTP-запросов.

## 3.2 Разработка клиентской части

На рисунке 3.2 отображена структура клиентской части разрабатываемого программного средства.

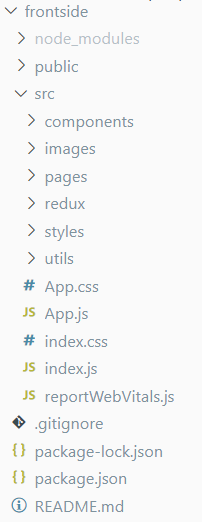


Рисунок 3.2 – Структура проекта клиента

Клиентская часть приложения написана на React, JavaScript-библиотеке для создания пользовательских интерфейсов (UI), разработанная Facebook. React позволяет создавать многокомпонентные приложения, которые отображают данные в режиме реального времени, не перезагружая страницу. Он использует декларативный подход для описания состояния компонентов и их поведения, что делает его простым в использовании и понимании.

Директория «src» содержит исходный код клиентской части приложения. Она содержит следующие файлы и папки:

* index.js: Это основной файл, который является точкой входа в приложение React. Он обычно импортирует компонент App и отображает его в корневом элементе DOM с помощью метода ReactDOM.render().
* App.js: Это компонент, который является основным компонентом приложения и содержит все другие компоненты.
* index.css: Это файл стилей для приложения. Он содержит общие стили для всех компонентов.
* components: Это папка, которая содержит все компоненты приложения. Каждый компонент обычно находится в отдельном файле и импортируется в основной файл App.js.
* utils: Это директория, которая содержит вспомогательные функции и утилиты, которые используются в приложении. Например, валидацию данных.
* images: Папка, которая содержит все изображения, используемые в приложении.
* pages: Данная папка, содержит компоненты-страницы приложения, каждый из которых имеет свой маршрут и отображает определенный контент на странице. Компоненты-страницы используют компоненты из папки «components» для построения пользовательского интерфейса.
* redux: Содержит код для реализации Redux-стора в приложении. Redux – это библиотека управления состоянием приложения, которая позволяет хранить данные в централизованном хранилище и управлять ими с помощью действий (actions) и редьюсеров (reducers).
* styles: Это папка, которая содержит файлы стилей для компонентов и страниц приложения.

При разработке клиентской части на React, удобно разделять компоненты, стили и вспомогательные функции по отдельным папкам для лучшей организации кода и его поддержки. Использование библиотеки Redux также позволяет управлять состоянием приложения в централизованном хранилище и уменьшить связность между компонентами.

4 Тестирование web-приложения

Тестирование программного обеспечения – проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. Для предотвращения возникновения исключительных ситуаций, которые препятствуют нормальной работе пользователя, их необходимо обработать. Функциональное тестирование проводится для проверки качества программного продукта, а точнее того, какие функции реализованы и насколько верно это сделано. Иными словами, это способ узнать, сможет ли разработанное ПО при определённых условиях выполнять необходимые задачи. В реализуемом мной программном средстве предусмотрены механизмы, которые направлены на их предотвращение.

Рассмотрим вход пользователя в систему. Приложение проверяет корректно ли введены данные и выводит соответствующее сообщение, представлено на рисунке 4.1



Рисунок 4.1 – Ошибка некорректных данных

Все поля в окнах авторизации и регистрации являются обязательными к заполнению, пример ошибки заполнения представлен на рисунке 4.2.

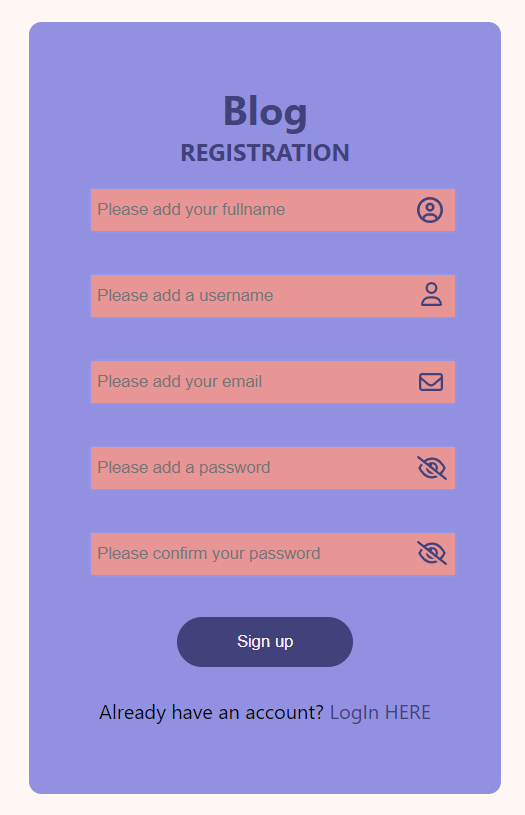


Рисунок 4.2 – Ошибка запылёнения полей

Кроме этого поля также проверяются на корректное заполнение, пример на рисунке 4.3. Примером такого валидируемого поля может являться поле email в окне регистрации.



Рисунок 4.3 – Ошибка заполнения поля email

Поле username является уникальным, следовательно, при регистрации уже существующего пользователя, сервер ответит ошибкой (рис. 4.4).

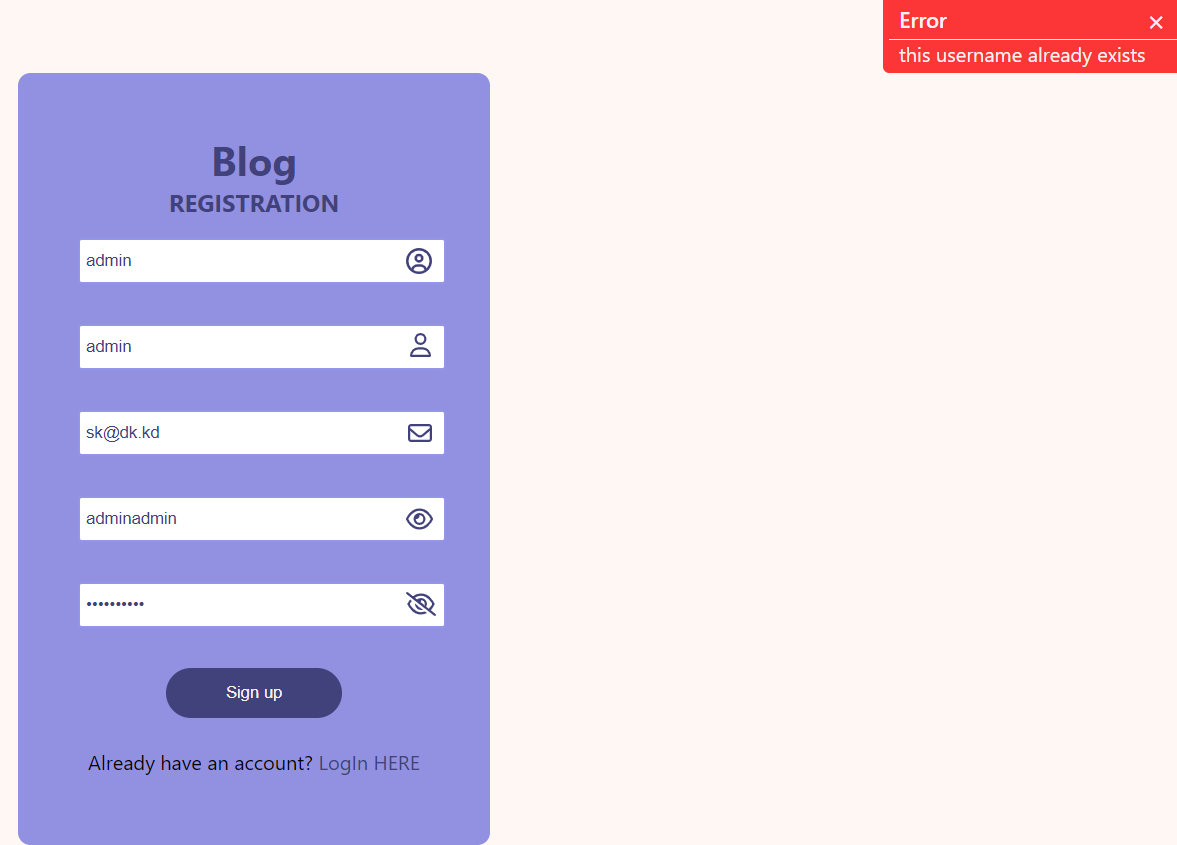


Рисунок 4.4 – Регистрация существующего пользователя

После успешной авторизации пользователь попадает на главную страницу, где ему доступна возможность создать новый пост. При создании поста, пользователь может прикрепить фотографию и добавить текстовое описание. Однако, при создании поста, обязательно должно быть заполнено только поле с фотографией. Если пользователь попытается создать пост без прикрепленной фотографии, он увидит соответствующее сообщение об ошибке (рис. 4.5).

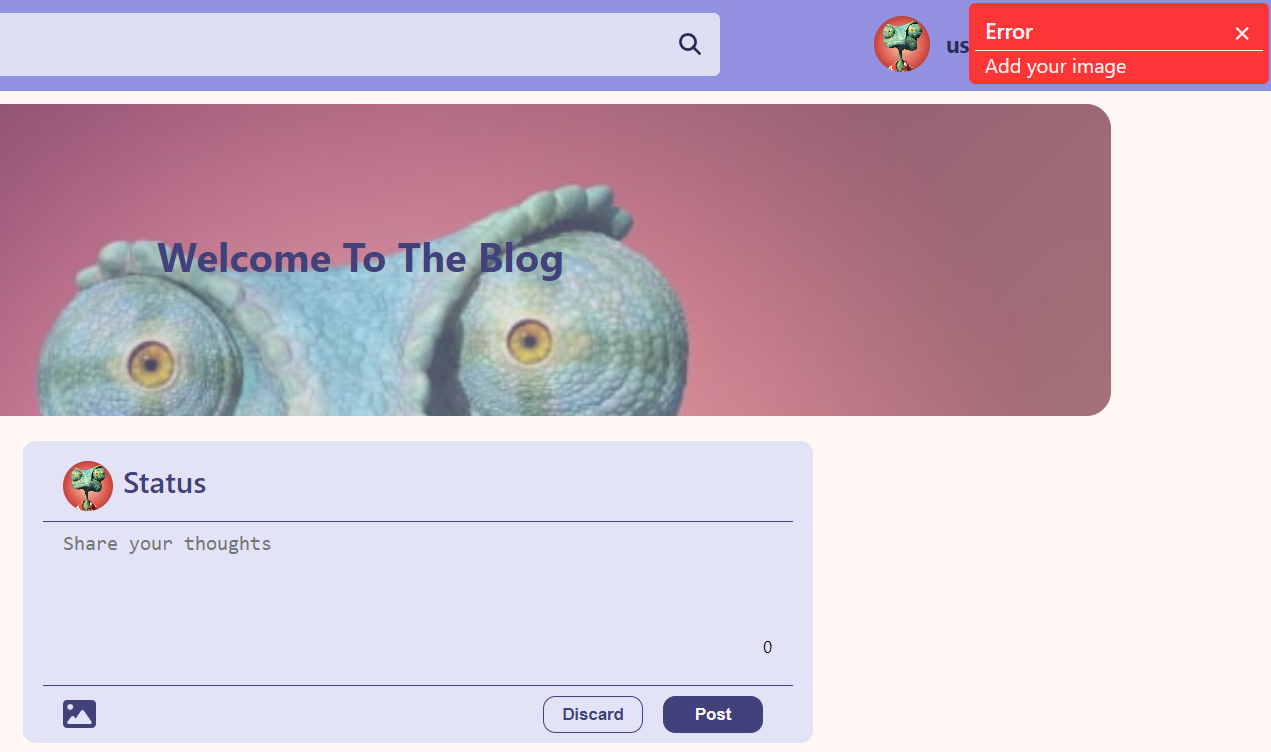


Рисунок 4.5 – Ошибка публикации поста

Пользователь имеет право изменять свои посты, однако если удалить все фотоиз поста выведется сообщение как на рисунке 4.6.

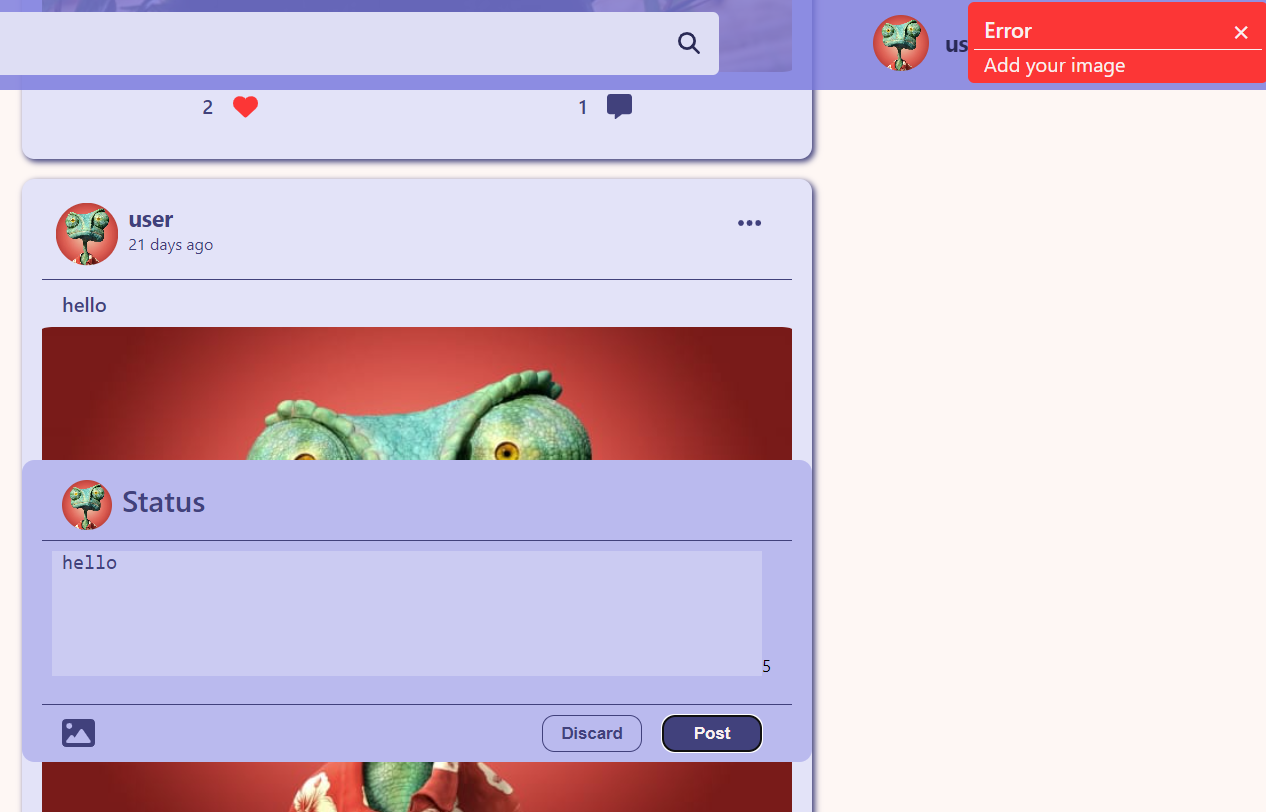


Рисунок 4.6 – Ошибка изменения поста

На странице профиль можно изменить информацию о себе и аватар, для аватара поддерживаются расширения jpeg и png, если фото не имеет данного расширения выведется ошибка представленная на рисунке 4.7.

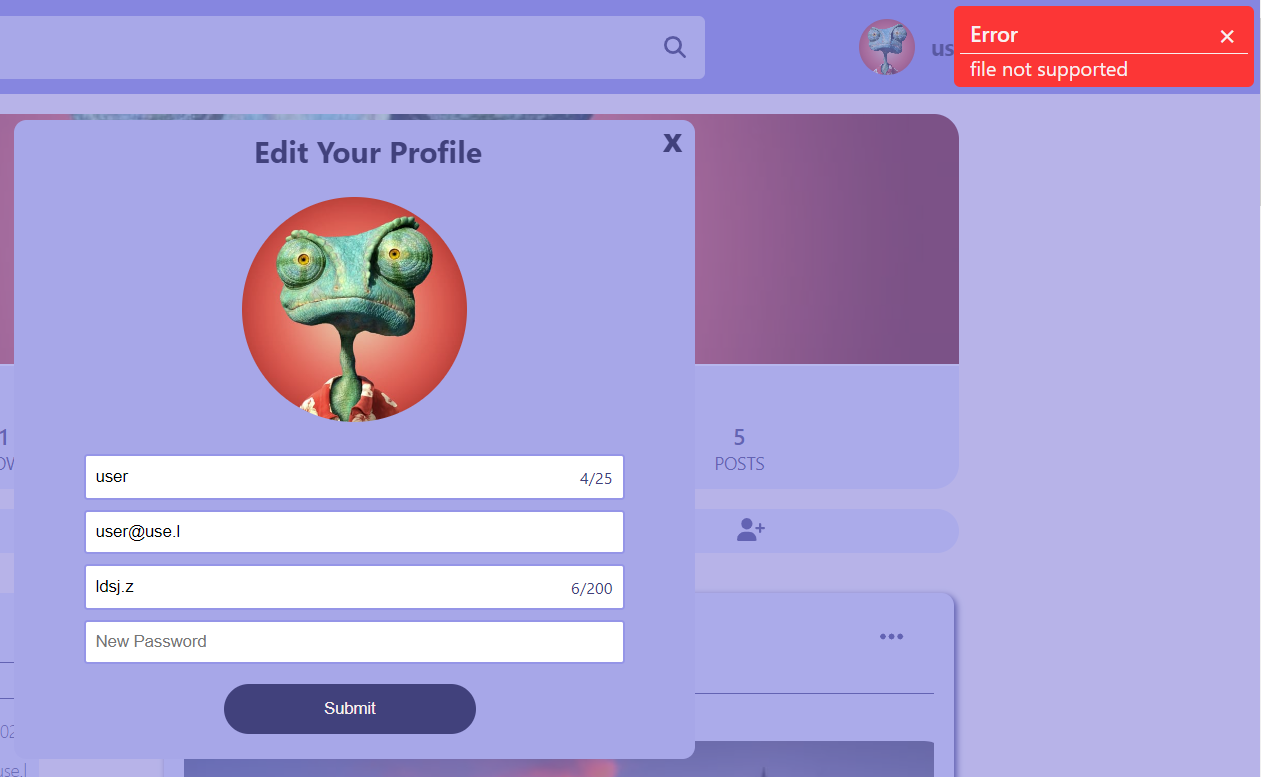


Рисунок 4.7 – Ошибка расширения

Администратор также может столкнуться с ошибками при использовании программного средства, как и любой другой пользователь. Например, при попытке добавить пост фото или при изменении собственного профиля.

5 Руководство пользователя

## 5.1 Регистрация пользователя

Для регистрации необходимо заполнить форму, которая продемонстрирована на рисунке 5.1.

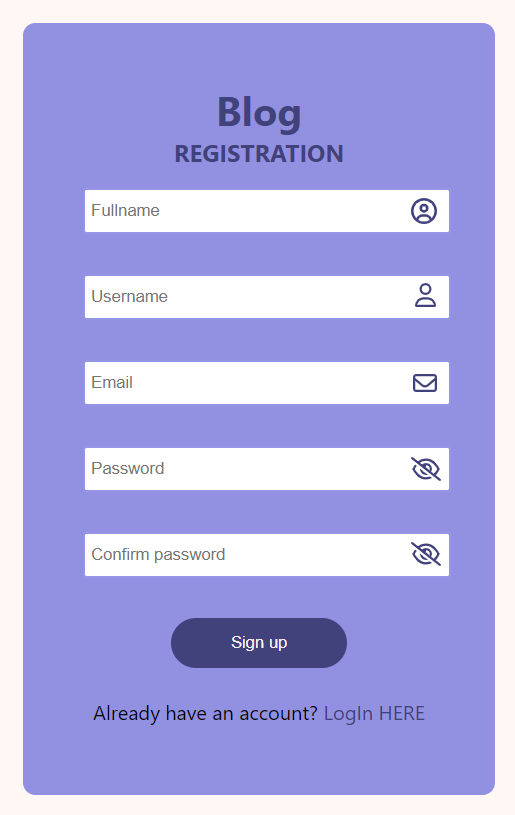


Рисунок 5.1 – Форма регистрации

После отправки пользователем запроса на регистрацию, происходит валидация всех полей формы. Если валидация будет неуспешной, пользователь увидит сообщения об ошибках в соответствующих полях для ввода информации. В случае успешной регистрации пользователь переадресовывается на главную страницу приложения.

## 5.2 Аутентификация и авторизация пользователя

Для аутентификации необходимо ввести пароль, логин и нажать на кнопку «Войти», что позволит проверить, существует ли пользователь с введенными данными в системе. Все поля являются обязательными для заполнения.

Если аутентификация прошла успешно и соблюдены все выше описанные правила, то происходит перенаправление на главную страницу, что означает, что пользователь был успешно авторизован. Скриншот формы авторизации представлен на рисунке 5.2.

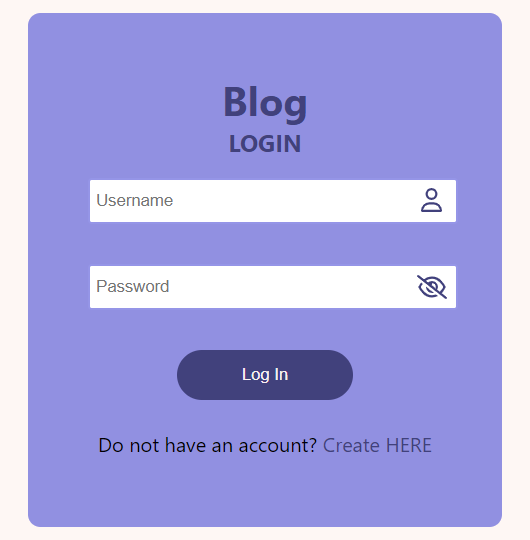


Рисунок 5.2 – Форма авторизации

При входе в систему авторизированному пользователю выдается токен, который записывается в cookies, тем самым позволяя разграничить возможности пользователей на сайте.

В данном курсовом проекте логика приложения задумана так, что пользователь, который не вошел в систему имеет доступ только к страницам регистрации и авторизации. Если пользователь авторизовался, то у него появляется доступ к возможностям приложения.

## 5.3 Просмотр постов

Для того, чтобы пользователь смог увидеть сообщения, ему достаточно зайти на сайт (рисунок 5.3).

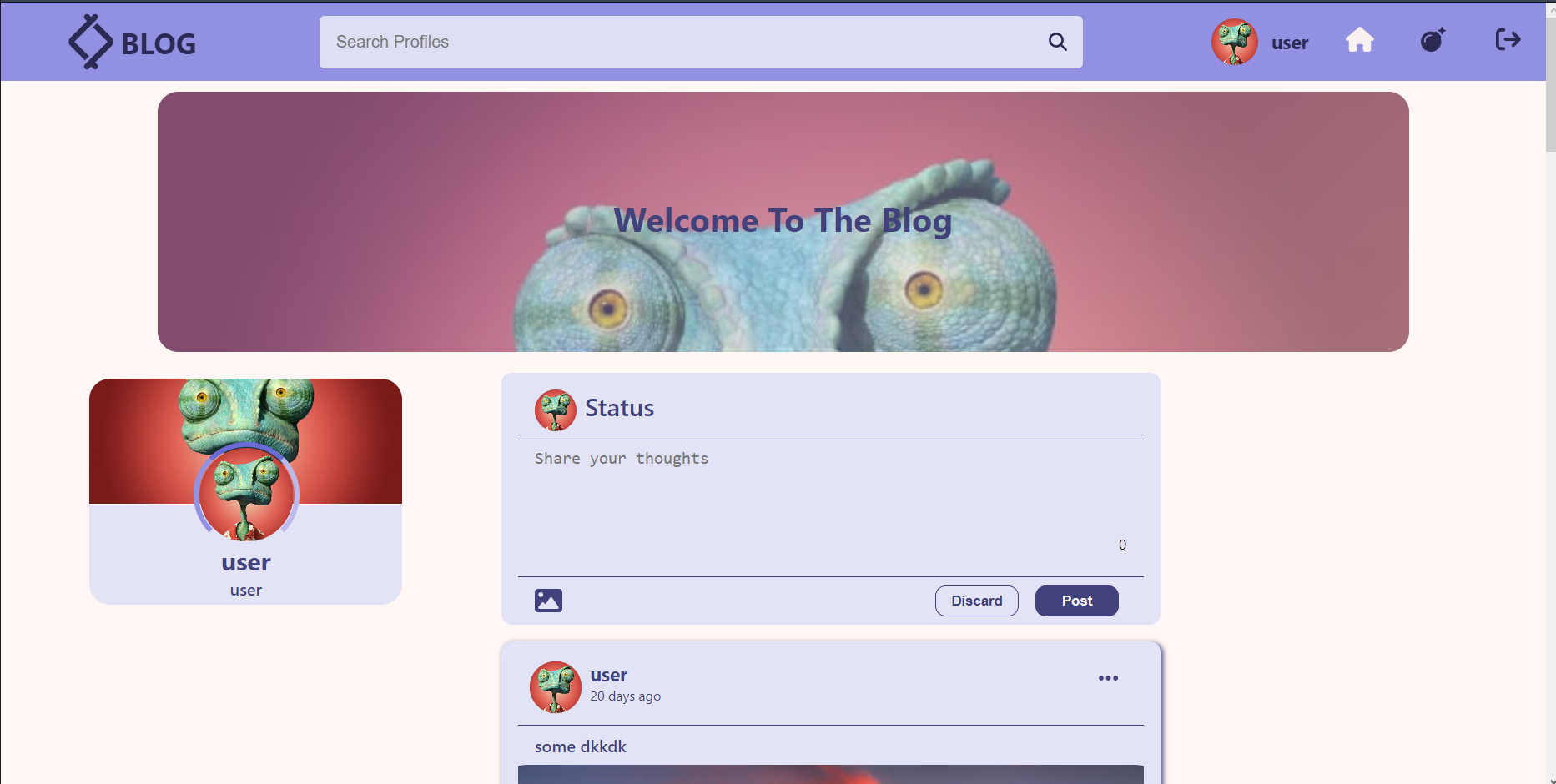


Рисунок 5.3 – Главная страница приложения

На данной странице также можно просматривать посты свои и своих друзей, добавлять посты.

Просматривать свои посты можно в собственном профиле (рис 5.4).

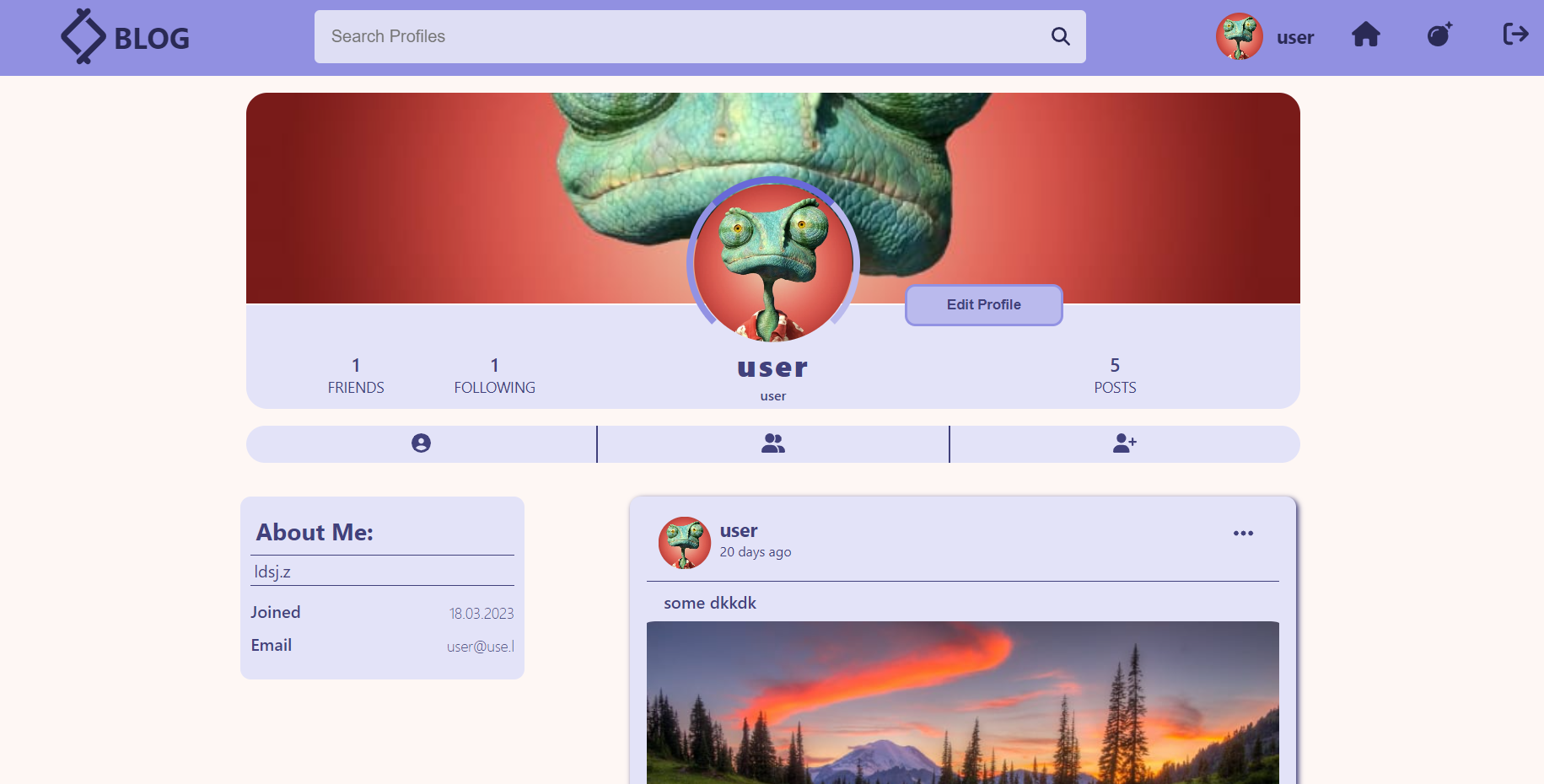


Рисунок 5.4 – Страница профиля

На данной странице также можно изменить профиль, нажав кнопку «Edit Profile», просмотреть своих друзей и подписчиков. Если перейти на страницу чужого профиля, то на месте изменения профиля появиться кнопка «Add Friend», которая позволяет подписаться на профиль или «Un Friend» чтобы отписаться от профиля. Скриншот демонстрирующий эту страницу представлен на рисунке 5.5.

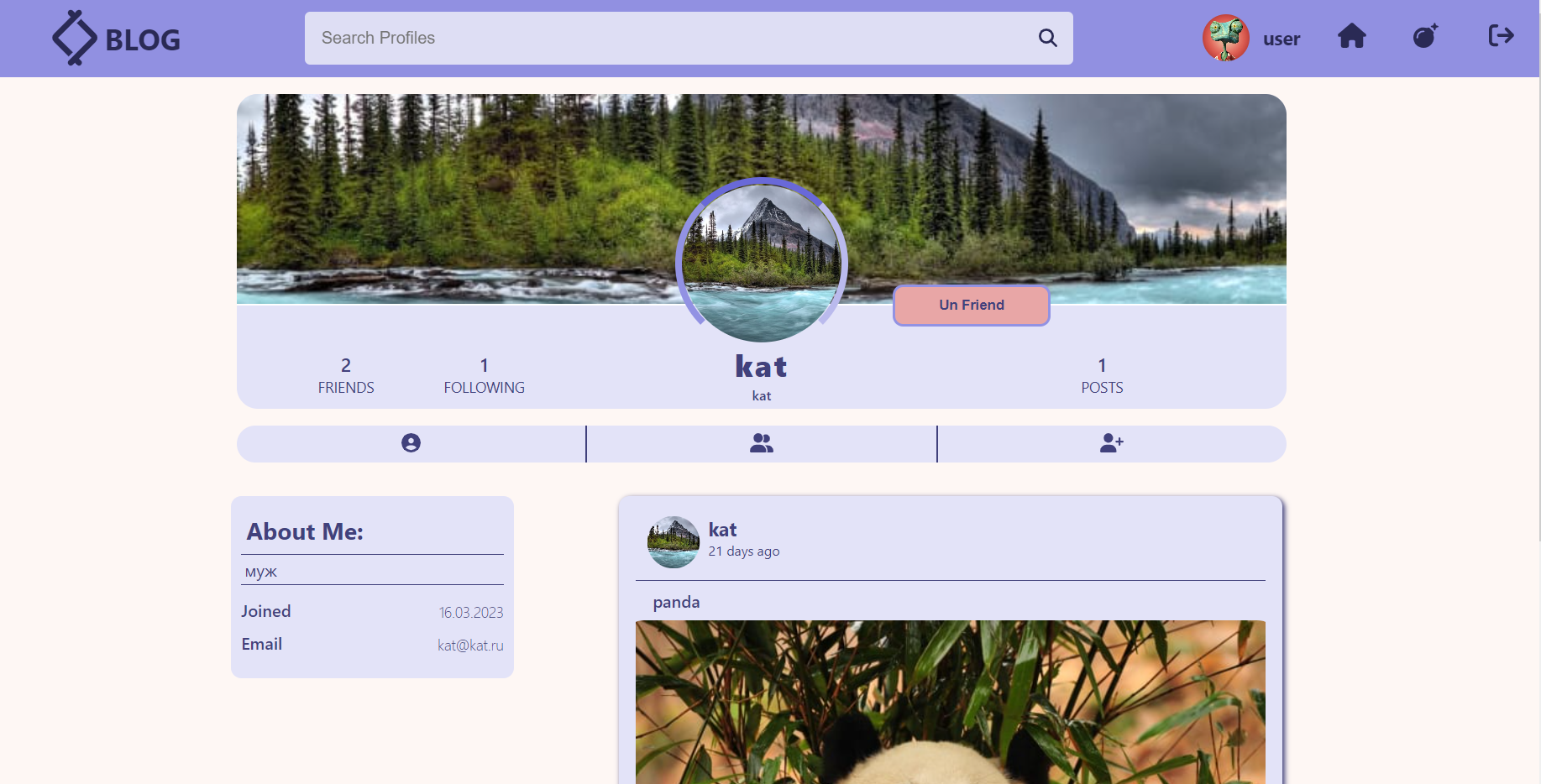


Рисунок 5.5 – Страница чужого профиля

Кроме кнопки эта страница ничем не отличается от профиля авторизованного пользователя.

## 5.4 Действия с постами

Добавить профиль можно на главной странице в форме, представленной на рисунке 5.6.

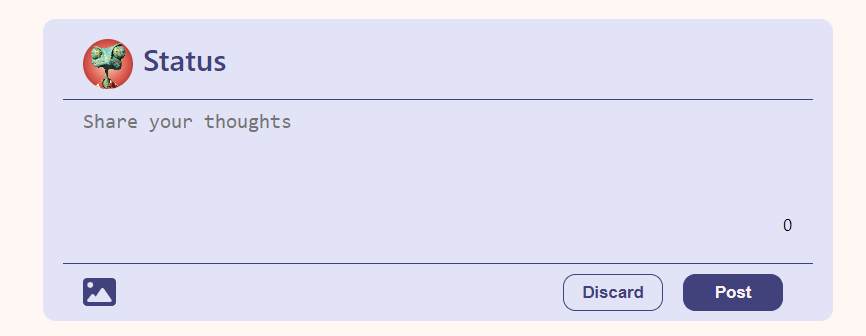


Рисунок 5.6 – Форма добавления поста

В текстовое поле можно добавить описание к посту, обязательно нужно нажать на иконку картинки, чтобы добавить фото к посту, для публикации поста следует нажать «Post».

Чтобы удалить или изменить пост нужно нажать 3 точки, как показано на рисунке 5.7 и выбрать нужное действие.



Рисунок 5.7 – Пост

Изменять и удалять можно только собственные посты, у чужих нет этих 3 точек.

Для того чтобы отметить понравившийся пост нужно нажать на сердечко, если вам оно разонравилось нужно нажать 2 раз на сердечко, и оно изменит цвет. Чтобы просмотреть все комментарии или оставить свой нужно нажать на иконку комментария в правом нижнем углу, как показано на рисунке 5.8.

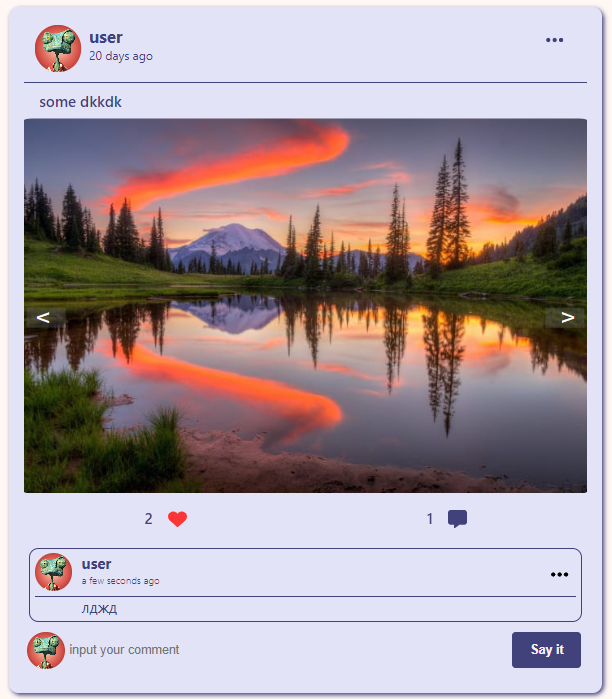


Рисунок 5.8 – Добавление комментария

Введя текст нажмите кнопку «Say it» чтобы опубликовать комментарии, изменять и удалять свои комментарии можно точно так же, как и посты, нажав три точки на нужном комментарии.

## 5.5 Администрирование

В данном курсовом проекте была разработана возможность администрирования. Администратор может удалять пользователей, посты, комментарии. Для того чтобы войти в админ панель, нужно авторизоваться как администратор и в хедере страницы нажать на иконку графика как показано на рисунке 5.9.

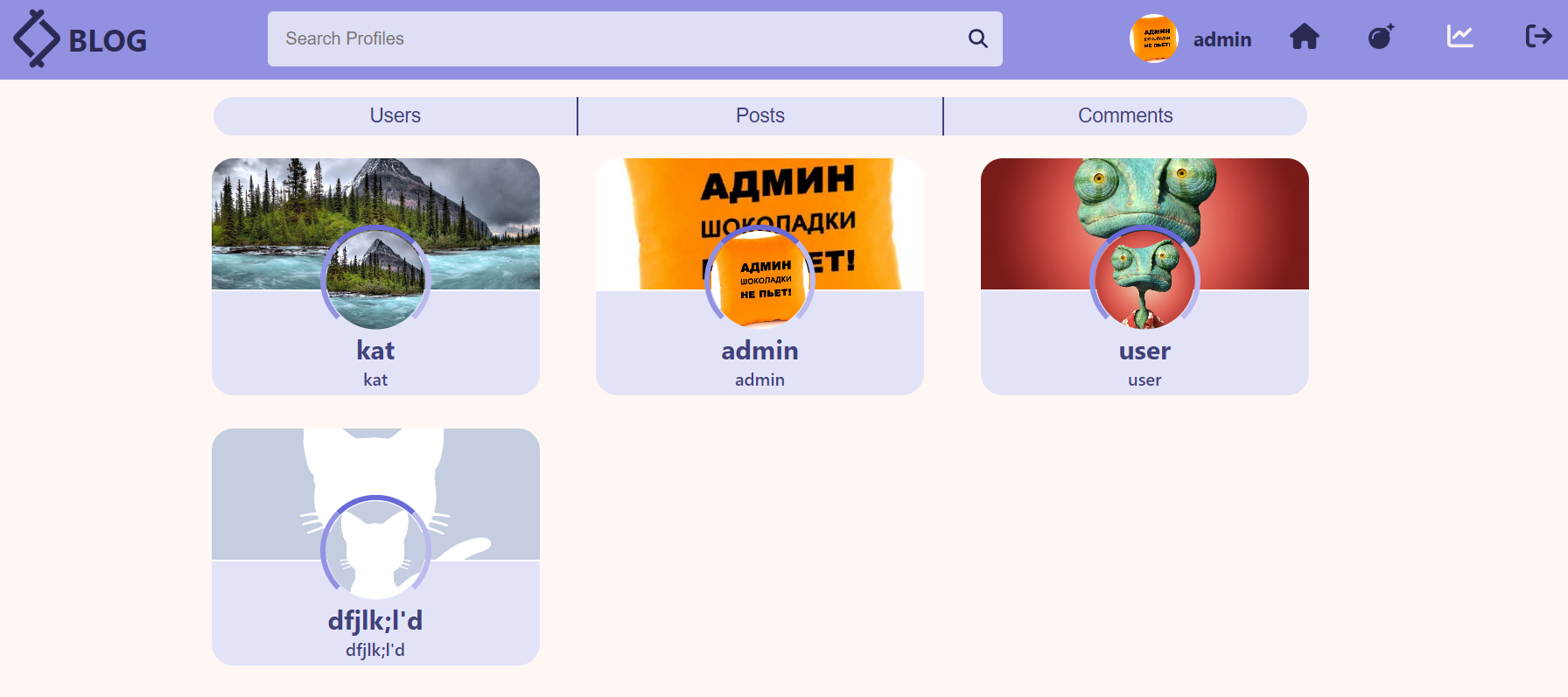


Рисунок 5.9 – Вход в панель администратора

Далее можно выбрать с чем именно вы хотите работать, например, чтобы удалить пользователя нужно нажать Users (рис. 5.10) и навести на пользователя, который хотите удалить, справа появиться крестик, нажав который пользователь удалиться.

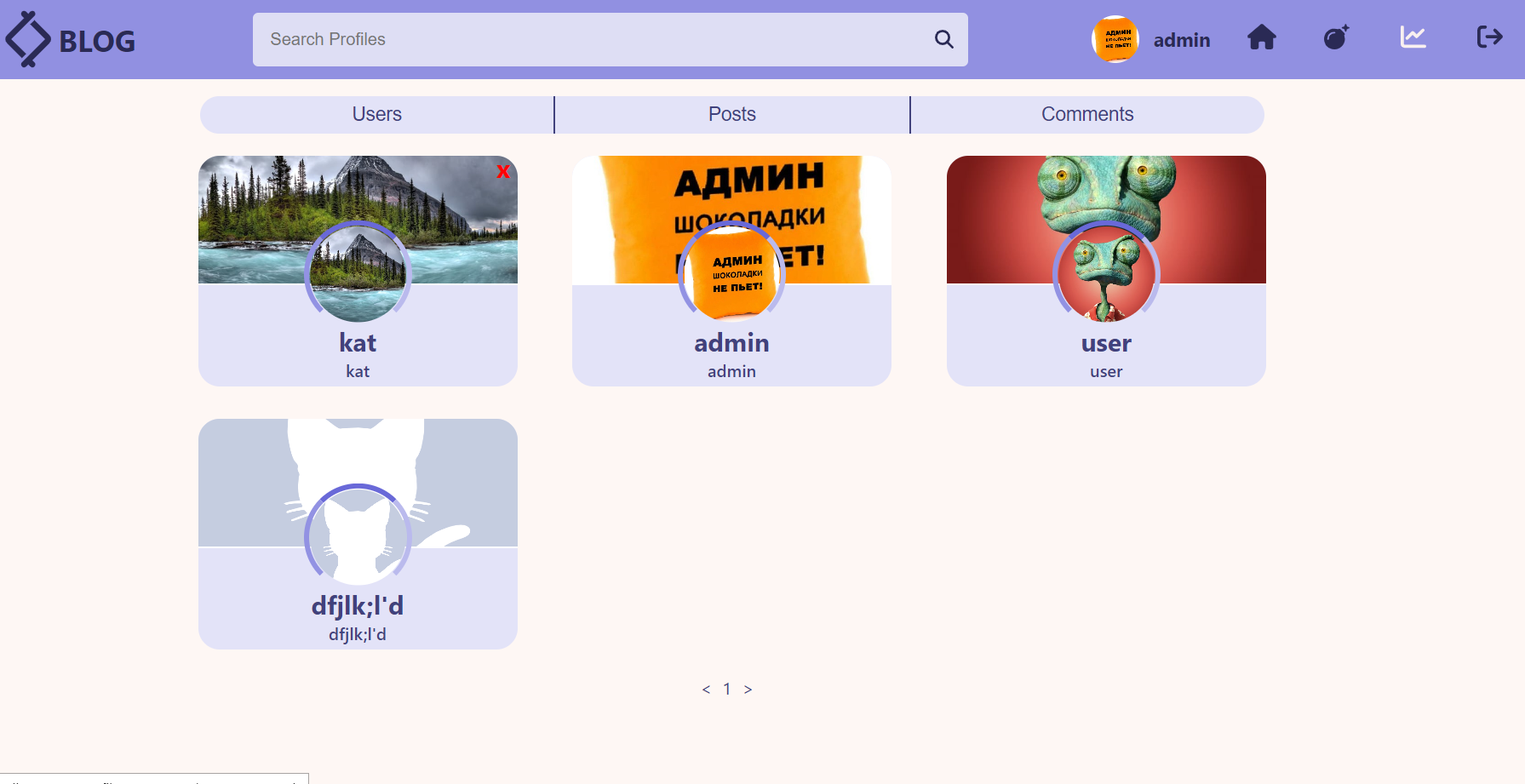


Рисунок 5.10 – Удаление пользователя

Удалить посты и комментарии можно аналогично перейдя по нужным страничкам.

# Заключение

В процессе решения поставленной задачи была достигнута поставленная цель по созданию веб-приложения «Блог».

Перед началом разработки был произведен аналитический обзор прототипов приложений подобной тематики и определение функциональных возможностей разрабатываемого приложения.

В процессе выполнения курсовой работы была спроектирована база данных для хранения в ней информации. База данных была разработана с помощью системы управления базами данных «MongoDB». Также был создан сервер и клиент с пользовательским интерфейсом. В том числе было реализовано тестирование программного продукта.

Также следует отметить, что для данного проекта был реализован сертификат LAB.crt для использования HTTPS протокола.

При разработке выполнены следующие пункты:

* регистрация и авторизация пользователей;
* разделение ролей;
* добавление, удаление, редактирование постов, комментариев;
* работа с изображениями.
* сохранение информации в базе данных;
* реализация архитектуры MVC;
* обмен данными между клиентом и сервером должен производиться по протоколу HTTPS;
* отмечать понравившиеся посты;
* редактировать профиль;
* подписываться на другие профили.

В соответствии с полученным результатом работы программы можно сделать вывод, что разработанная программа работает верно, а требования технического задания выполнены в полном объёме.

# Список используемых источников

1. HTTPS документация [Электронный ресурс]. – Режим доступа: <https://nodejs.org/dist/latest-v18.x/docs/api/https.html>. – Дата доступа: 08.05.2023.
2. Socket.io документация [Электронный ресурс]. – Режим доступа: <https://socket.io/docs/v4/>. – Дата доступа: 04.05.2023;
3. Документация Mongoose [Электронный ресурс].– Режим доступа: <https://mongoosejs.com/docs/guide.html> .– Дата доступа: 15.04.2023.
4. Документация React [Электронный ресурс]. – Режим доступа: <https://react.dev/learn/choosing-the-state-structure>– Дата доступа: 07.04.2023.

# Приложение А

const Users =require('../models/userModel')

const bcrypt =require('bcrypt')

const jwt =require('jsonwebtoken')

const authCtrl={

    register: async(req,res)=>{

*try*{

            const{fullname,username,email,password}= req.body;

            const newUsername =username.toLowerCase().replace(/ /g,'');

            const user\_name = *await* Users.findOne({username: newUsername})

*if*(user\_name) *return* res.status(400).json({msg:'this username already exists'})

            const Email = *await* Users.findOne({email: email})

*if*(Email) *return* res.status(400).json({msg:'this email already exists'})

*if*(password.length < 6) *return* res.status(400).json({msg:'password must be atleast 6 characters long'})

            const passwordHash = *await* bcrypt.hash(password,13);

            const newUser = new Users({fullname, username:newUsername, email, password:passwordHash})

            const access\_token = createAccessToken({id:newUser.\_id});

            const refresh\_token = createRefreshToken({id:newUser.\_id});

            res.cookie('refreshtoken',refresh\_token,{

                httpOnly:true,

                path:'/api/refresh\_token',

                maxAge: 24\*30\*60\*60\*1000*//30 days*

            })

*await* newUser.save();

            res.json({

                msg:"registration success",

                access\_token,

                user:{

                ...newUser.\_doc,

                password:''

            }})

        }*catch*(err){

*return* res.status(500).json({msg:err.message})

        }

    },

    login: async(req,res)=>{

*try*{

            const {username, password} =req.body;

            const user =*await* Users.findOne({username})

*if*(!user) *return* res.status(400).json({msg: 'User does not exists'})

            const isMatch = *await* bcrypt.compare(password,user.password);

*if*(!isMatch) *return* res.status(400).json({msg: 'User Password is incorrect'})

            const access\_token = createAccessToken({id:user.\_id});

            const refresh\_token = createRefreshToken({id:user.\_id});

            res.cookie('refreshtoken',refresh\_token,{

                httpOnly:true,

                path:'/api/refresh\_token',

                maxAge: 24\*30\*60\*60\*1000*//30 days*

            })

            res.json({

                msg:"login success",

                access\_token,

                user:{

                ...user.\_doc,

                password:''

            }})

        }*catch*(err){

*return* res.status(500).json({msg:err.message})

        }

    },

    logout: async(req,res)=>{

*try*{

            res.clearCookie('refreshtoken',{path:'/api/refresh\_token'})

            res.json({msg: 'Logged out'})

        }*catch*(err){

*return* res.status(500).json({msg:err.message})

        }

    },

    generateAccessToken: async(req,res)=>{

*try*{

            const rf\_token =req.cookies.refreshtoken;

*if*(!rf\_token) *return* res.status(400).json({msg:'Please login now'})

            jwt.verify(rf\_token, process.env.REFRESHTOKENSECRET,async(err,result)=>{

*if*(err) *return* res.status(400).json({msg:'No Please login now'})

                const user =*await* Users.findById(result.id).select('-password')

*if*(!user) *return* res.status(400).json({msg:'User does not exist'})

                const access\_token = createAccessToken({id: result.id});

                res.json({

                    access\_token,

                    user

                })

            } )

        }*catch*(err){

*return* res.status(500).json({msg:err.message})

        }

    }

}

const createAccessToken =(payload)=>{

*return* jwt.sign(payload, process.env.ACCESSTOKENSECRET,{expiresIn:"1d"})

}

const createRefreshToken =(payload)=>{

*return* jwt.sign(payload, process.env.REFRESHTOKENSECRET,{expiresIn:"30d"})

}

module.exports =authCtrl

const Comments = require('../models/commentModel');

const Posts = require('../models/postModel')

const commentCtrl ={

    createComment: async(req,res)=>{

*try* {

            const {content,postId} =req.body;

            const post =*await* Posts.findById(postId)

*if*(!post) *return* res.status(400).json({msg: "no post found"})

            const newComment = *await* new Comments({user:req.user.\_id, content})

*await* newComment.save();

*await* Posts.findOneAndUpdate({\_id:postId},{

                $push:{comments: newComment.\_id}

            })

*return* res.json({newComment})

        } *catch* (err) {

*return* res.status(500).json({msg: err.message})

        }

    },

    updateComment: async(req,res)=>{

*try* {

            const {content} =req.body;

*await* Comments.findOneAndUpdate({\_id:req.params.id, user:req.user.\_id}, {content})

*return* res.json({msg:'update success'})

        } *catch* (err) {

*return* res.status(500).json({msg: err.message})

        }

    },

    deleteComment: async(req,res)=>{

*try* {

*await* Comments.findOneAndDelete({\_id:req.params.id})

*return* res.json({msg:'delete success'})

        } *catch* (err) {

*return* res.status(500).json({msg: err.message})

        }

    }

}

module.exports= commentCtrl;

const Friend = require("../models/friendModel");

const friendCtrl = {

    getFriends: async(req,res)=>{

*try* {

            const user = *await* Friend.find({user: req.params.id})

            const friend = *await* Friend.find({friend: req.params.id})

            res.json({user, friend})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    countOfFriends:  async (req, res) => {

*try* {

            const count = *await* Friend.count({ user: req.params.id});

            res.json({ count });

        } *catch* (err) {

*return* res.status(500).json({ msg: err.message });

        }

    },

    countOfFollowers:  async (req, res) => {

*try* {

            const count = *await* Friend.count({ friend: req.params.id});

            res.json({ count });

        } *catch* (err) {

*return* res.status(500).json({ msg: err.message });

        }

    },

    friend: async (req, res) => {

*try* {

            const existingFriend = *await* Friend.find({ friend: req.body.user,user: req.body.friend});

*if* (existingFriend.length >0) {

*return* res.status(400).json({ msg: 'Friendship already exists' });

            } *else* {

                const friend = new Friend({friend: req.body.user,user: req.body.friend,});

*await* friend.save();

                res.json({ msg: "Friend added" });

            }

        } *catch* (err) {

*return* res.status(500).json({ msg: err.message });

        }

    },

    unfriend: async (req, res) => {

*try* {

            const deletedCount = *await* Friend.findOneAndDelete({friend: req.body.user,user: req.body.friend,});

*if* (deletedCount == null) {

*return* res.status(400).json({ msg: 'Friendship does not exist' });

            }

            res.json({ msg: "Friend removed" });

        } *catch* (err) {

*return* res.status(500).json({ msg: err.message });

        }

    },

};

module.exports = friendCtrl;

const Posts =require('../models/postModel')

const Comments =require('../models/commentModel')

const postCtrl ={

    createPost: async( req, res)=>{

*try* {

            const {content, images} =req.body;

            const user =req.user.\_id;

            const newPost =new Posts({ content, images, user})

*await* newPost.save();

*return* res.status(200).json({msq:"Post saved", newPost})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    getPost: async( req, res)=>{

*try* {

            const friendsList= req.friends.map(friend => friend.friend);

            friendsList.push(req.user.\_id)

            const posts = *await* Posts.find({user: {$in: friendsList}})

            .populate("user likes", "username avatar fullname")

            .populate({

                path:'comments',

                populate:{

                    path:"user",

                    select:"-password"

                }

            })

            .sort({createdAt: -1});

*if*(!posts) *return* res.status(400).json({msg:"No post found"})

*return* res.status(200).json({

                msg:'post found',

                result: posts.length,

                posts

            })

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    allPost: async( req, res)=>{

*try* {

            const posts = *await* Posts.find()

            .populate("user likes", "username avatar fullname")

            .populate({

                path:'comments',

                populate:{

                    path:"user",

                    select:"-password"

                }

            })

            .sort({createdAt: -1});

*if*(!posts) *return* res.status(400).json({msg:"No post found"})

*return* res.status(200).json({

                msg:'post found',

                result: posts.length,

                posts

            })

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    userPost: async( req, res)=>{

*try* {

            const posts = *await* Posts.find({user: req.params.id})

            .populate("user likes", "username avatar fullname")

            .populate({

                path:'comments',

                populate:{

                    path:"user",

                    select:"-password"

                }

            })

            .sort({createdAt: -1});

*if*(!posts) *return* res.status(400).json({msg:"No post found"})

*return* res.status(200).json({

                msg:'post found',

                result: posts.length,

                posts

            })

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    deletePost:async( req, res)=>{

*try* {

            const post = *await* Posts.findOneAndDelete({\_id:req.params.id})

*await* Comments.deleteMany({\_id:{$in:post.comments}})

*return* res.json({

                msg:"Post deletes"

            })

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    delcommupdatePost: async( req, res)=>{

*try* {

            const {comments}=req.body.newComment

            const post =*await* Posts.findOneAndUpdate({\_id:req.body.newComment.\_id},{comments})

            .populate("user likes", "username avatar fullname")

*return* res.status(200).json({

                msq:"Post update",

                newPost: {...post.\_doc, comments}

            })

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    updatePost: async( req, res)=>{

*try* {

            const {content, images} =req.body;

            const post =*await* Posts.findOneAndUpdate({\_id:req.params.id},{ content, images})

            .populate("user likes", "username avatar fullname")

*return* res.status(200).json({

                msq:"Post update",

                newPost: {...post.\_doc, content, images}

            })

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    likePost:async(req, res)=>{

*try* {

            const post =*await* Posts.find({\_id:req.params.id, likes:req.user.\_id})

*if*(post.length>0) *return* res.status(400).json({msg:"you have already like this post"})

*await* Posts.findOneAndUpdate({\_id:req.params.id},{

                $push:{likes: req.user.\_id}

            },{new:true})

*return* res.json({msg:"Post Likes"})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    dislikePost:async(req, res)=>{

*try* {

            const a= *await* Posts.findOneAndUpdate({\_id:req.params.id},{

                $pull:{likes: req.user.\_id}

            },{new:true})

*return* res.json({msg:"Post disLikes"})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

}

module.exports =postCtrl;

const Friend = require("../models/friendModel");

const Posts = require("../models/postModel");

const Comments = require("../models/commentModel");

const Users = require("../models/userModel");

const bcrypt =require('bcrypt')

const userCtrl ={

    searchUser: async(req,res)=>{

*try* {

            const users = *await* Users.find({username: {$regex: req.query.username}}).limit(10)

            .select('fullname username avatar')

            res.json({users})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    getUser: async(req,res)=>{

*try* {

            const user = *await* Users.findOne({\_id: req.params.id})

*if*(!user) *return* res.status(400).json({msg:"No user Exist"})

            res.json({user})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    allUser: async(req,res)=>{

*try* {

            const user = *await* Users.find()

            .select('-password')

*if*(!user) *return* res.status(400).json({msg:"No user Exist"})

            res.json({user})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    updateUser: async(req,res)=>{

*try* {

            const {\_id, avatar, email, story, fullname, password} = req.body

            const passwordHash = *await* bcrypt.hash(password,13);

*if*(!fullname)  *return* res.status(400).json({msg:"Full name is requires"})

*await* Users.findOneAndUpdate({\_id},{

                email, story, fullname, avatar, password: passwordHash

            },{new: true})

            res.json({msg:'update success'})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    },

    deleteUser: async(req,res)=>{

*try* {

*await* Friend.deleteMany({friend: req.params.id})

*await* Friend.deleteMany({user: req.params.id})

            const post = *await* Posts.deleteMany({user:req.params.id})

*await* Comments.deleteMany({\_id:{$in:post.comments}})

*await* Comments.deleteMany({user:req.params.id})

*await* Users.findOneAndDelete({\_id:req.params.id})

            res.json({msg:'delete success'})

        } *catch* (err) {

*return* res.status(500).json({msg:err.message})

        }

    }

}

module.exports =userCtrl;

# Приложение Б

const mongoose =require('mongoose');

const commentSchema =mongoose.Schema({

    content:{ type:String, required:true},

    user:{type:mongoose.Types.ObjectId, ref:'user'}

},{ timestamps:true})

module.exports=mongoose.model('comment',commentSchema);

const mongoose =require('mongoose');

const friendSchema = mongoose.Schema({

    user: { type: mongoose.Types.ObjectId, ref: 'user' },

    friend: { type: mongoose.Types.ObjectId, ref: 'user' },

});

module.exports = mongoose.model('friend', friendSchema);

const mongoose =require('mongoose');

const userSchema =mongoose.Schema({

    username:{ type:String, trim:true, unique:true, maxLength:25, required:true},

    fullname:{ type:String, trim:true, required:true},

    email:{ type:String, trim:true, required:true

    password:{ type:String, required:true},

    story:{ type:String, default:'', maxLength:200},

    avatar:{

        type:String,

        default:'https://images-wixmp-ed30a86b8c4ca887773594c2.wixmp.com/f/202ffa91-500c-4ed3-9f1d-59416d87c196/d34f0su-6585aa78-2cdb-4403-8785-bdbbfd26cd8d.png?token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJ1cm46YXBwOjdlMGQxODg5ODIyNjQzNzNhNWYwZDQxNWVhMGQyNmUwIiwic3ViIjoidXJuOmFwcDo3ZTBkMTg4OTgyMjY0MzczYTVmMGQ0MTVlYTBkMjZlMCIsImF1ZCI6WyJ1cm46c2VydmljZTpmaWxlLmRvd25sb2FkIl0sIm9iaiI6W1t7InBhdGgiOiIvZi8yMDJmZmE5MS01MDBjLTRlZDMtOWYxZC01OTQxNmQ4N2MxOTYvZDM0ZjBzdS02NTg1YWE3OC0yY2RiLTQ0MDMtODc4NS1iZGJiZmQyNmNkOGQucG5nIn1dXX0.fjvIOB8Mp5iye1xQKds\_vWkY5\_y4Z63F0Dlot4ToLJE',

    },

    isadmin:{ type:Boolean, default:0, },

},{    timestamps:true})

module.exports=mongoose.model('user',userSchema);

# Приложение В

const router =require('express').Router();

const authCtrl =require('../controllers/authCtrl')

router.post('/register', authCtrl.register);

router.post('/login', authCtrl.login);

router.post('/logout', authCtrl.logout);

router.post('/refresh\_token', authCtrl.generateAccessToken);

module.exports = router;

const router = require('express').Router()

const auth = require('../middlewares/auth')

const commentCtrl =require('../controllers/commentCtrl')

router.post('/comment',auth,commentCtrl.createComment)

router.patch('/comment/:id',auth,commentCtrl.updateComment)

router.delete('/comment/:id',auth,commentCtrl.deleteComment)

module.exports = router;

const router = require('express').Router()

const auth = require('../middlewares/auth')

const friendCtrl =require('../controllers/friendCtrl')

router.get('/fri/:id',auth,friendCtrl.getFriends)

router.get('/friends/:id',auth,friendCtrl.countOfFriends)

router.get('/followers/:id',auth,friendCtrl.countOfFollowers)

router.post('/user/:id/friend',auth, friendCtrl.friend)

router.post('/user/:id/unfriend',auth, friendCtrl.unfriend)

module.exports =router;

const router = require('express').Router()

const auth = require('../middlewares/auth')

const userCtrl =require('../controllers/userCtrl')

router.get('/search',auth,userCtrl.searchUser)

router.get('/user/all',auth,userCtrl.allUser)

router.get('/user/:id',auth,userCtrl.getUser)

router.patch('/user',auth, userCtrl.updateUser)

router.delete('/user/:id',auth, userCtrl.deleteUser)

module.exports =router;