IT CAREER HUB

Описание финального проекта

IT CAREER HUB

Концепция проекта

Вы - веб-разработчик в крупной компании и получили новый проект - создание мини-версии сервиса для обмена фотографиями и постами, где пользователи могут регистрироваться, авторизоваться, создавать публикации, оставлять комментарии и ставить лайки. Кроме того, будет реализована возможность управления профилем, включая редактирование информации и загрузку аватара, а также поисковая система для поиска пользователей. Все данные будут храниться в базе данных MongoDB, с которой мы будем взаимодействовать через библиотеку Mongoose.

IT CAREER HUB

Цель проекта

Создание backend API для сервиса, подобного Instagram.

IT CAREER HUB

Этапы работы над проектом

Разработка основных функций backend API для сервиса:

- Реализация регистрации и авторизации пользователей с использованием JWT.
- Создание функционала для работы с постами: добавление, удаление, редактирование.
- Настройка возможности лайкать и комментировать посты.
- Разработка поисковой системы для поиска пользователей.
- Взаимодействие с MongoDB через Mongoose для хранения всех данных.

Макет проекта

Изучите макет проекта.

Настройка проекта

Давайте для начала создадим базовую структуру нашего проекта и установим все необходимые пакеты, создадим важные файлы и подключимся к базе данных MongoDB.

Инициализация проекта Node.js

1. Откройте терминал в папке, где будет находиться ваш проект.
2. Для инициализации нового проекта введите команду: npm init -y

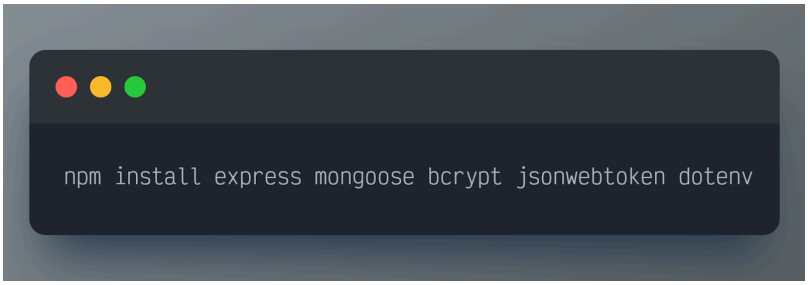
Эта команда создаст файл `package.json`, в котором будет храниться информация о вашем проекте и всех установленных библиотеках.

Установка необходимых пакетов

Теперь установим все пакеты, которые нам понадобятся для создания нашего backend API.

- Express — веб-фреймворк для Node.js, который поможет нам создавать сервер.
- Mongoose — библиотека для работы с MongoDB, которая упрощает управление данными в базе.
- bcrypt — библиотека для шифрования паролей.
- jsonwebtoken (JWT) — инструмент для создания токенов авторизации.
- dotenv — пакет для работы с переменными окружения, например, для хранения конфиденциальных данных, таких как пароль к базе данных.

Для установки этих пакетов введите в терминале следующую команду:



Эта команда установит все нужные библиотеки и добавит их в файл `package.json`, чтобы они всегда были доступны в нашем проекте.

Создание структуры проекта

Чтобы проект был организован и поддерживать его было проще, создадим несколько папок, в которых будут храниться различные части кода:

- config/ — здесь будут настройки проекта, такие как подключение к базе данных.
- controllers/ — здесь будут функции, которые будут управлять логикой нашего приложения (например, регистрация пользователей, создание постов).
- middlewares/ — здесь будут храниться промежуточные функции, такие как проверка авторизации.
- models/ — в этой папке будут схемы данных для MongoDB (например, модель пользователя и поста).
- routes/ — здесь будут файлы, в которых мы будем описывать маршруты (например, на какой URL должен реагировать сервер).

Подключение к MongoDB через Mongoose

Теперь настроим подключение к базе данных MongoDB, чтобы наше приложение могло сохранять и получать данные, такие как пользователи и посты.

1. В папке **config/** создайте файл `db.js`. В этом файле мы опишем, как подключаться к базе данных:

JavaScript

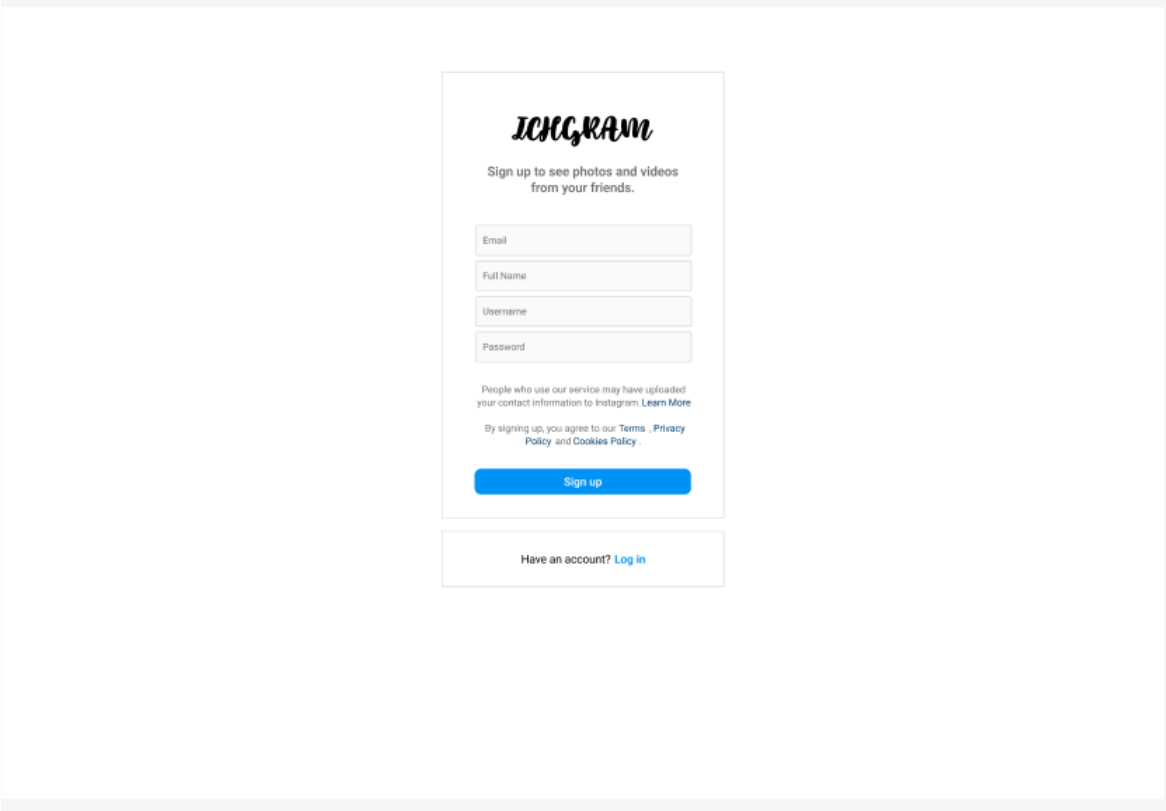
```
```javascript
const mongoose = require('mongoose');
require('dotenv').config();

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('MongoDB connected');
  } catch (error) {
    console.error('MongoDB connection error:', error);
    process.exit(1); // Остановка приложения при ошибке подключения
  }
};

module.exports = connectDB;
```
```

2. Теперь создайте файл **.env** в корне проекта, чтобы хранить конфиденциальные данные, например, строку подключения к MongoDB. В файл `.env` добавьте строку:

JavaScript

```
```env
MONGO_URI=ваша_строка_подключения_к_MongoDB
```
```

3. После этого, подключение к базе данных нужно использовать в основном файле сервера. Создайте файл `server.js` в корне проекта и подключите базу данных:

JavaScript

```javascript
const express = require('express');
const connectDB = require('./config/db');
require('dotenv').config();

const app = express();

// Подключаемся к MongoDB
connectDB();

app.use(express.json()); // Для работы с JSON

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Регистрация и авторизация



Мы используем JWT для авторизации пользователей, поскольку это удобно и безопасно для работы с API. Когда пользователь логинится, мы создаем токен с его данными, и дальше этот токен отправляется с каждым запросом на защищенные маршруты. Мы уже проходили JWT, так что здесь просто применим это на практике.

Регистрация пользователей: хэширование паролей с bcrypt, валидация данных
Для регистрации нам нужно убедиться, что данные пользователя корректны (например, email должен быть уникальным), и безопасно сохранить его пароль. Для этого будем использовать bcrypt для хэширования паролей. Это защитит нас, если вдруг произойдет утечка данных. Валидация данных также важна — проверим, что все поля заполнены и правильные.

Код:

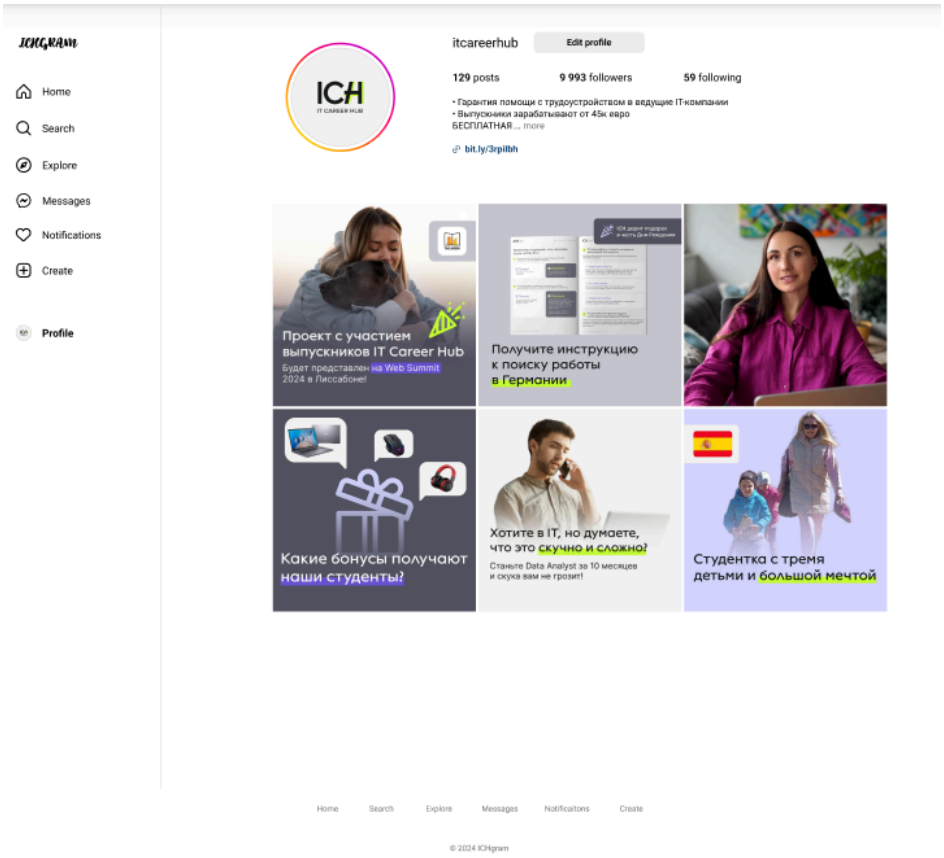
1. Создадим модель пользователя: Здесь мы описываем, какие данные у пользователя (имя, email, пароль, полное имя). Используем Mongoose для создания схемы. Не забываем про хэширование пароля через bcrypt.

2. Маршруты для регистрации и авторизации: Мы создадим два основных маршрута — для регистрации и логина. Они будут обрабатывать входящие запросы, проверять данные и возвращать JWT, если все успешно.

3. Контроллеры: Для регистрации проверим, существует ли пользователь с таким email или username, хэшируем пароль и сохраняем пользователя. Для логина проверим email и введенный пароль и сгенерируем токен.

4. Middleware для защиты маршрутов: Это промежуточный код, который проверяет токен и защищает маршруты. Если токен валидный — пользователь получает доступ, если нет — отклоняем запрос.

Работа с профилем пользователя



Мы реализуем две основные функции:

1. Получение данных профиля: Позволяет получить данные профиля любого пользователя по его уникальному идентификатору.
2. Обновление профиля: Пользователь может редактировать своё имя, биографию и загружать новое изображение профиля.

Это основные возможности для взаимодействия с профилем. Мы уже создали пользователя при регистрации, и теперь будем использовать его данные для работы с профилем.

Код:

1. Получение данных профиля: Мы будем использовать контроллер для запроса профиля пользователя. Здесь важно исключить конфиденциальную информацию, такую как пароль, чтобы случайно не передать его на клиент. Для этого мы используем метод `.select('-password')` при запросе данных.

2. Обновление профиля: Здесь пользователь сможет изменить своё имя, биографию или изображение профиля. В коде мы уже проверяем, что если эти поля присутствуют в запросе, то они будут обновлены.

3. Работа с изображением профиля: Мы можем сохранять изображение в формате Base64, что позволяет передавать его в виде строки. Base64 — это метод кодирования бинарных данных, таких как изображения, в текстовый формат, который легко можно передать через API.

Пример кода для обработки изображения на бэкенде

Мы можем использовать библиотеку `multer` для обработки загрузки файлов на бэкенде, и затем преобразовывать файл в Base64.

1. Установка зависимостей:

Для работы с загрузкой изображений нам нужно установить `multer`: npm install multer

2. Настройка загрузки файлов с использованием `multer`:

В коде мы можем настроить обработку загружаемых изображений, а затем преобразовать изображение в строку Base64 и сохранить его.

```
import multer from 'multer';
import fs from 'fs';
import User from '../models/userModel.js';
import getUserIdFromToken from '../utils/helpers.js';

// Настройка multer для загрузки файлов в память
const storage = multer.memoryStorage();
const upload = multer({ storage: storage });

// Обновление профиля пользователя
export const updateUserProfile = async (req, res) => {

  const userId = getUserIdFromToken(req);

  try {
    const user = await User.findById(userId);

    if (!user) {
      return res.status(404).json({ message: 'Пользователь не найден' });
    }

    const { username, bio } = req.body;

    if (username) user.username = username;
    if (bio) user.bio = bio;

    // Проверка наличия файла изображения в запросе
    if (req.file) {
      // Преобразуем изображение в Base64
      const base64Image = req.file.buffer.toString('base64');

      // Добавляем префикс типа данных для корректного сохранения
      const base64EncodedImage = `data:${req.file.mimetype};base64,${base64Image}`;

      // Сохраняем изображение в формате Base64 в профиле пользователя
      user.profile_image = base64EncodedImage;
    }

    const updatedUser = await user.save();
    res.status(200).json(updatedUser);
  } catch (error) {
    res.status(500).json({ message: 'Ошибка обновления профиля', error: error.message });
  }
};

// Middleware для загрузки изображения
export const uploadProfileImage = upload.single('profile_image');
```

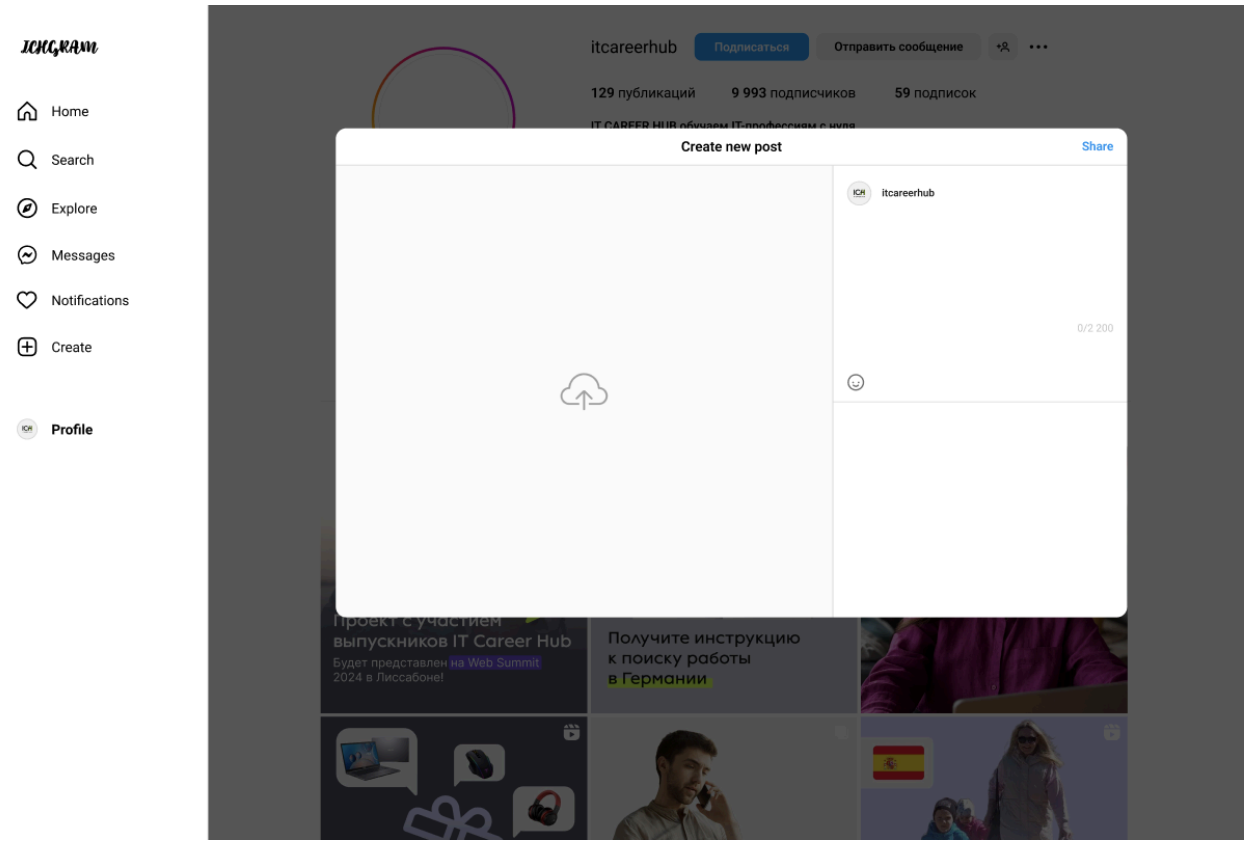
Как это работает:

1. `multer`: Используем `multer` для загрузки файлов. В данном случае мы используем `memoryStorage`, что позволяет сохранять файл прямо в оперативной памяти, не записывая его на диск.

2. `req.file.buffer`: Получаем загруженный файл и обращаемся к его бинарным данным через `req.file.buffer`. Затем мы конвертируем этот буфер в строку Base64 с помощью метода `.toString('base64')`.

3. Формат Base64: Чтобы сохранить изображение корректно, важно добавить префикс типа данных (например, ``). Это необходимо, чтобы клиент мог правильно обработать изображение при отображении.

Создание постов



Работа с постами включает в себя создание, отображение, обновление и удаление постов пользователя. Для этого мы создаем модель постов (postModel.js), которая будет хранить информацию о каждом посте: текстовое описание, изображение (сохраняем его в формате Base64, как и для профиля пользователя), а также ссылку на автора поста.

Модель взаимодействует с MongoDB через Mongoose, что позволяет легко выполнять CRUD операции (создание, чтение, обновление, удаление) над документами в базе данных.

Код:

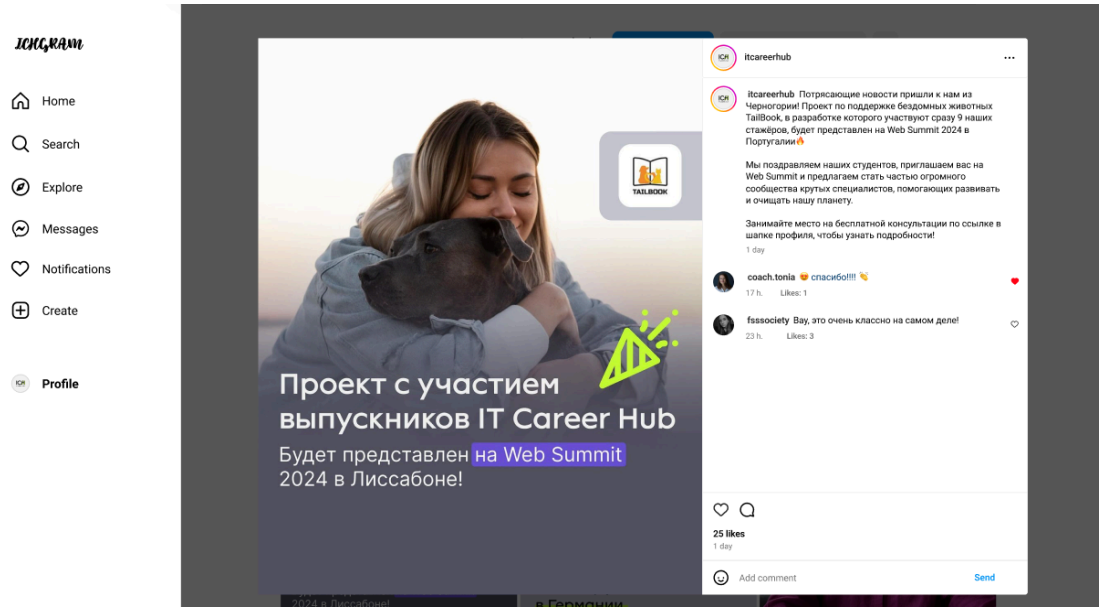
Для работы с постами создаётся контроллер postController.js, который будет включать основные функции для выполнения операций с постами:

1. Получение всех постов пользователя: Позволяет вывести все посты, созданные конкретным пользователем.

2. Создание поста: Принимает изображение, которое конвертируется в формат Base64, и текстовое описание. Пост сохраняется в базе данных вместе с ссылкой на автора.
3. Удаление поста: Удаляет пост по его ID, если пользователь является автором.
4. Получение конкретного поста по ID: Возвращает информацию о посте, включая описание и изображение.
5. Обновление поста: Позволяет изменить описание или изображение поста.
6. Получение всех постов: Возвращает список всех постов (например, для ленты новостей).

Маршруты для всех этих операций реализуются в postRoutes.js, где каждая функция контроллера подключается к соответствующему маршруту API.

Взаимодействие с постами: лайки и комментарии



Для полноценного взаимодействия пользователей с постами нужно реализовать функционал лайков и комментариев. Для этого создаём отдельные модели лайков и комментариев (likeModel.js, commentModel.js). Эти модели будут связываться с пользователями и постами через ссылки (ID), что позволит отслеживать, кто оставил лайк или комментарий, и к какому посту они относятся.

- Модель лайков (likeModel.js): Каждый лайк будет привязан к пользователю и конкретному посту. Это позволяет пользователям ставить лайки постам, а также отслеживать количество лайков у каждого поста.
- Модель комментариев (commentModel.js): Комментарии включают текст сообщения, пользователя (автора комментария), и пост, к которому комментарий добавлен. Это позволяет пользователям комментировать посты, и добавлять взаимодействие с контентом.

Код:

Для реализации взаимодействия с постами мы создаём контроллеры:

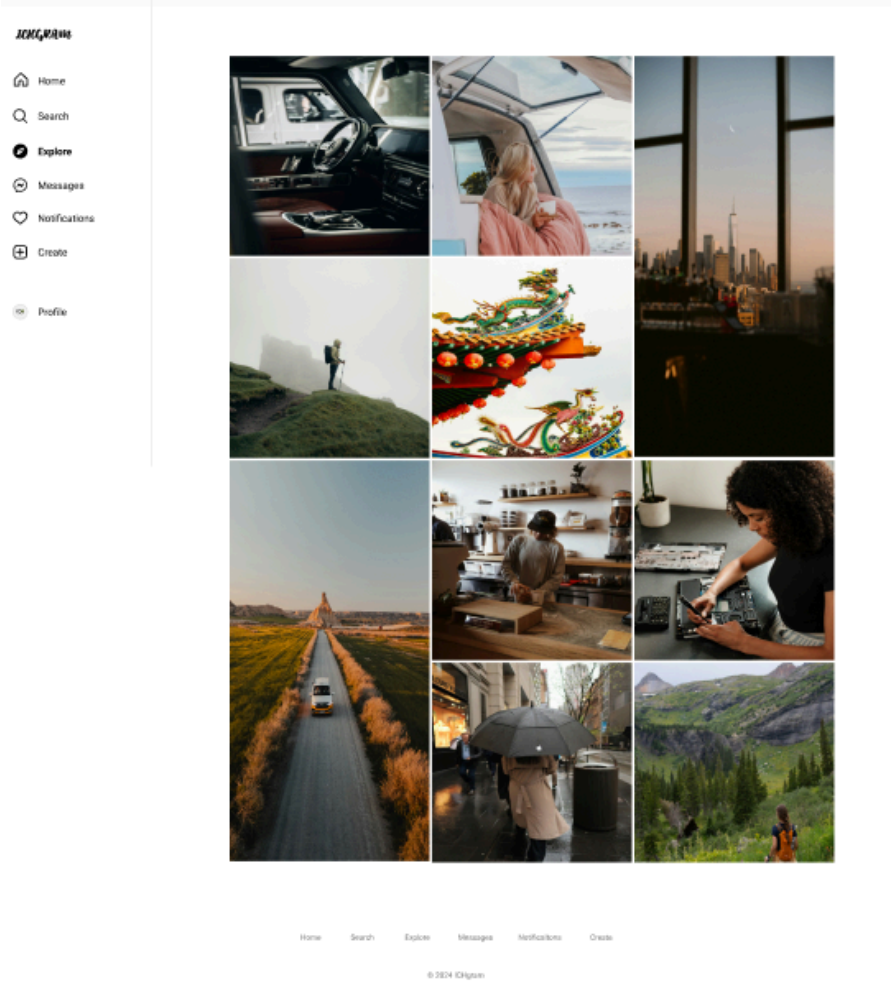
1. Добавление лайков (likeController.js): Когда пользователь нажимает на "лайк" под постом, проверяется, существует ли уже лайк от этого пользователя на

данном посте. Если лайка нет, он создаётся; если лайк уже есть, его можно удалить (реализуем возможность отмены лайка).

2. Добавление комментариев (commentController.js): Пользователь может добавлять комментарий под постом, передавая текст сообщения. Комментарий сохраняется с привязкой к посту и пользователю, который его оставил.
3. Получение лайков и комментариев: Реализуем функции для отображения всех лайков и комментариев к посту. Это нужно для того, чтобы показывать количество лайков и список комментариев на фронтенде.

Все маршруты для лайков и комментариев прописываются в likeRoutes.js и commentRoutes.js, где подключаем контроллеры для добавления, удаления лайков и работы с комментариями.

Поисковая система



Поисковая система — важный элемент взаимодействия с платформой, который помогает пользователям находить интересующий контент и других пользователей. Мы реализуем два ключевых аспекта:

1. Поиск пользователей: Это основная функция, которая позволяет искать других пользователей по имени или username. Когда пользователь вводит ключевые слова в поисковую строку, система ищет совпадения среди зарегистрированных пользователей. Это упрощает процесс нахождения друзей или других интересных аккаунтов, с которыми можно взаимодействовать.

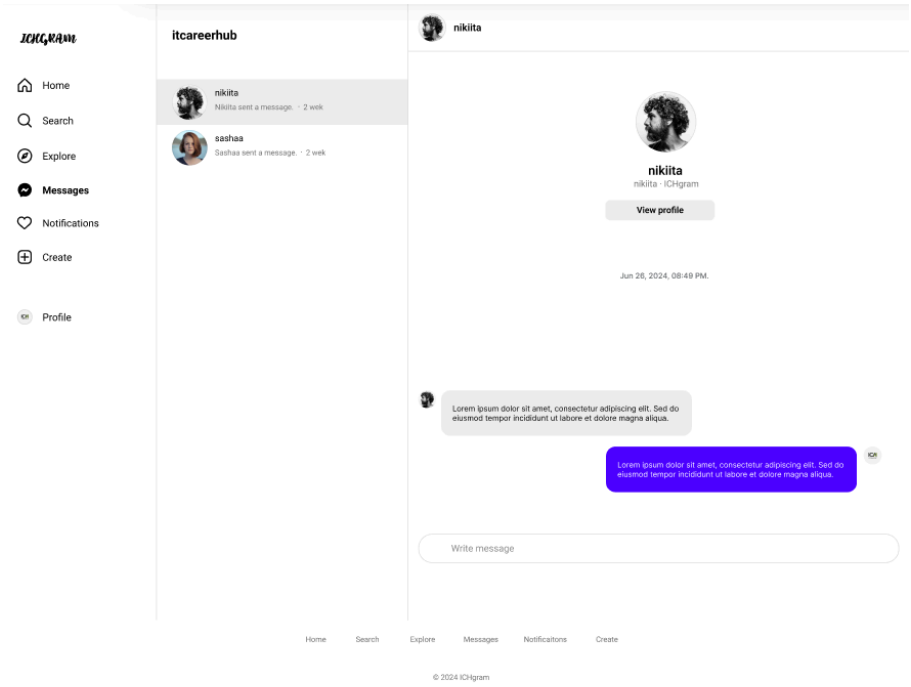
2. Раздел Explore: Вкладка Explore показывает все посты в случайном порядке, предлагая пользователям открыть для себя новый контент без необходимости задавать конкретные критерии поиска.

Поиск пользователей: Реализуется через запросы к базе данных, где система ищет совпадения по указанным ключевым словам среди профилей пользователей. Это помогает быстро находить пользователей с определёнными именами или никами.

Explore — произвольный порядок постов: Посты отображаются случайным образом. Мы не применяем конкретные фильтры или критерии для отбора постов, позволяя пользователям случайным образом открывать для себя новое.

Система обмена сообщениями*

***дополнительный функционал, опционально, выполняется по желанию**



Система обмена сообщениями между пользователями в реальном времени — это важная часть социальных платформ. Для того чтобы сообщения передавались мгновенно, удобно использовать веб-сокеты через библиотеку `socket.io`. Веб-сокеты обеспечивают двустороннюю связь между клиентом и сервером, что позволяет отправлять и получать сообщения без необходимости перезагрузки страницы или периодических запросов к серверу.

Основные компоненты системы обмена сообщениями:

- Socket.io: предоставляет возможность устанавливать постоянное соединение между клиентом и сервером, позволяя обмениваться сообщениями в реальном времени.
- Модель сообщений (messageModel.js): хранит сообщения с полями отправителя, получателя, текста сообщения и времени отправки.
- Socket-события: для передачи данных через сокеты можно создать события, например, `message` для отправки и получения сообщений, `connect` и `disconnect` для отслеживания соединений пользователей.

Реализация:

1. Установка socket.io:

Для начала необходимо установить библиотеку `socket.io` на сервере и клиенте: npm install socket.io

2. Настройка серверной части:

После установки необходимо настроить сервер Express для работы с веб-сокетами. Это можно сделать, интегрируя `socket.io` в сервер Express. При подключении пользователей сервер будет отправлять сообщения напрямую клиентам и принимать сообщения от них.

3. Подключение и передача сообщений:

После установки сокетов сервер сможет принимать и отправлять сообщения в реальном времени. В случае обмена сообщениями, когда пользователь отправляет сообщение, сервер будет передавать его получателю через событие сокета. Это позволяет мгновенно обновлять список сообщений у всех вовлеченных участников чата.

Код:

1. Socket-события:

Для реализации чата нужно создать события для отправки и получения сообщений. Например, событие `message` будет передавать сообщение от одного пользователя к другому через сервер:

- Событие отправки: Когда один пользователь отправляет сообщение, оно через сокет передается на сервер.
- Событие получения: Сервер перенаправляет это сообщение другому пользователю, который получает его через свой сокет.

2. Хранение сообщений:

Также нам необходимо сохранять сообщения в базе данных. Это позволяет пользователям видеть историю сообщений, даже если они отключались от чата.

3. Поддержка множественных чатов:

При помощи `rooms` (комнат) или `namespaces` в Socket.io можно организовать чаты между несколькими парами пользователей, чтобы сообщения поступали только тем, кто участвует в конкретном чате.

Реализация системы обмена сообщениями с использованием Socket.io

1. Установка необходимых пакетов

Для начала нужно установить зависимости: npm install express socket.io mongoose

2. Настройка сервера с Socket.io

Для начала нужно передать объект `io` в контроллер. Для этого сделаем так, чтобы `io` можно было использовать в контроллерах.

```
import express from 'express';
import { createServer } from 'http';
import { Server } from 'socket.io';
import dotenv from 'dotenv';
import mongoose from 'mongoose';
import messageRoutes from './routes/messageRoutes.js';

dotenv.config();

const app = express();
const httpServer = createServer(app);

// Создание instance для Socket.io
const io = new Server(httpServer, {
  cors: {
    origin: '*',
  },
});

// Middleware для передачи io в req
app.use((req, res, next) => {
  req.io = io;
  next();
});

// Подключение к MongoDB
mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

// Маршруты сообщений
app.use('/api/messages', messageRoutes);

const PORT = process.env.PORT || 5000;
httpServer.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

3. Контроллер сообщений `messageController.js`

Теперь в `messageController` можно использовать `req.io` для отправки сообщений через веб-сокеты, а основная логика отправки сообщений останется в контроллере.

```
import Message from '../models/messageModel.js';
import User from '../models/userModel.js';

// Получение всех сообщений между двумя пользователями
export const getMessages = async (req, res) => {
  const { userId, targetUserId } = req.params;

  try {
    const messages = await Message.find({
      $or: [
        { sender_id: userId, receiver_id: targetUserId },
        { sender_id: targetUserId, receiver_id: userId },
      ],
    }).sort({ created_at: 1 });

    res.status(200).json(messages);
  } catch (error) {
    res.status(500).json({ error: 'Ошибка при получении сообщений' });
  }
};

// Отправка сообщения
export const sendMessage = async (req, res) => {
  const { userId, targetUserId } = req.params;
  const { message_text } = req.body;

  try {
    // Проверка пользователей
    const user = await User.findById(userId);
    const targetUser = await User.findById(targetUserId);

    if (!user || !targetUser) {
      return res.status(404).json({ error: 'Пользователь не найден' });
    }

    // Сохранение сообщения в базе данных
    const message = new Message({
      sender_id: userId,
      receiver_id: targetUserId,
      message_text,
      created_at: new Date(),
    });

    await message.save();

    // Отправляем сообщение через Socket.io
    req.io.to(targetUserId).emit('receiveMessage', {
      senderId: userId,
      messageText: message_text,
      createdAt: message.created_at,
    });

    res.status(201).json(message);
  } catch (error) {
    res.status(500).json({ error: 'Ошибка при отправке сообщения' });
  }
};
```

4. Клиентская часть

Теперь клиент может подключиться через Socket.io и получать новые сообщения в реальном времени.

5. Еще раз, как это работает:

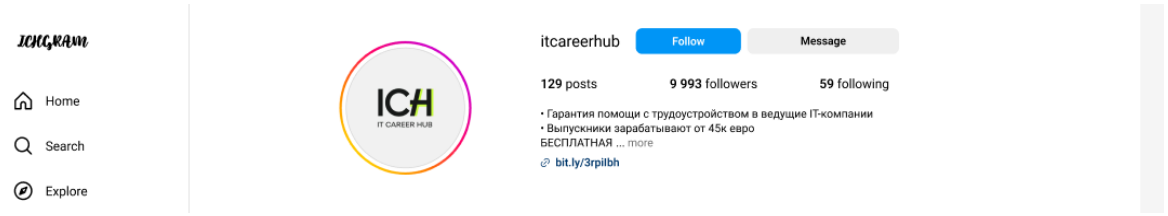
Пользователь подключается к серверу через Socket.io и присоединяется к "комнате", уникальной для каждого пользователя.

Когда пользователь отправляет сообщение, оно отправляется на сервер через событие `sendMessage`.

Сервер сохраняет сообщение в MongoDB и отправляет его целевому пользователю, используя его ID для направления сообщения в нужную "комнату".

Получатель сообщения получает его в реальном времени через событие `receiveMessage`, и оно отображается в чате.

Подписки



Функционал подписок позволяет пользователям подписываться друг на друга, а также получать уведомления о новых постах от тех, на кого они подписаны. Реализация требует создания модели для хранения подписок и разработки маршрутов и контроллеров для подписки, отписки и получения списка подписчиков и подписок.

Функции:

1. Получение подписчиков пользователя:

Когда пользователь хочет увидеть, кто на него подписан, выполняется запрос, который ищет все записи в базе данных, где пользователь является объектом подписки. Эти данные можно использовать, чтобы показать список всех подписчиков.

2. Получение подписок пользователя:

Если пользователь хочет увидеть, на кого он сам подписан, выполняется запрос для поиска всех записей, где он является субъектом подписки (`follower_id`). Это позволяет показать список тех, на кого пользователь подписан.

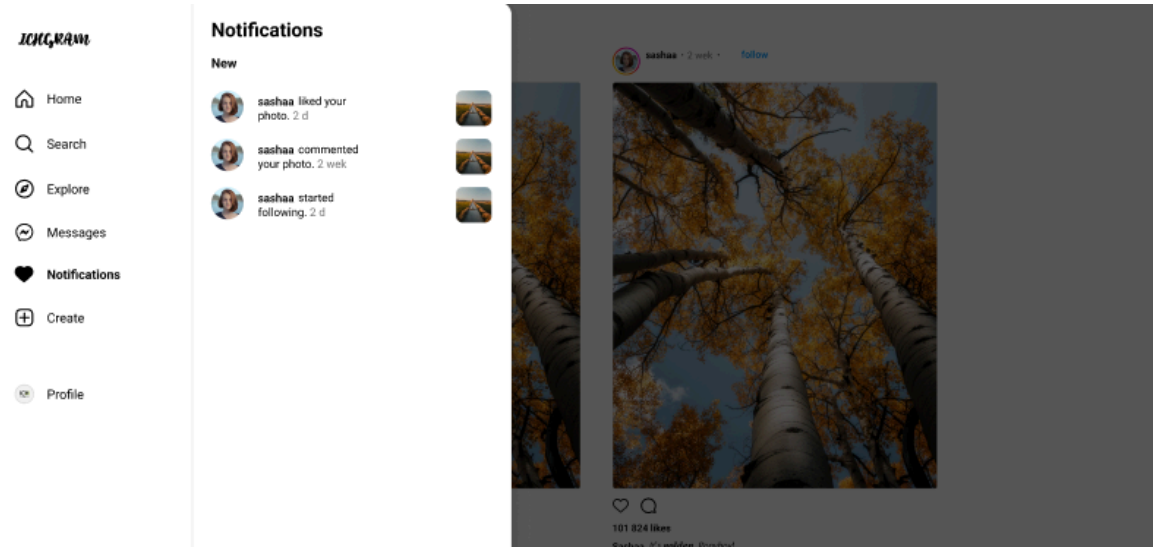
3. Подписка на пользователя:

Чтобы подписаться на другого пользователя, создается новая запись в базе данных, где указывается идентификатор пользователя, который подписывается, и того, на кого он подписывается. Проверяется, существует ли уже такая подписка, чтобы избежать дублирования. При успешной подписке можно отправлять уведомление пользователю, на которого подписались.

4. Отписка от пользователя:

Когда пользователь решает отписаться, удаляется запись из базы данных, связывающая подписчика и пользователя. После успешной отписки можно удалить или обновить соответствующие уведомления.

Уведомления



Панель навигации и система уведомлений играют ключевую роль в пользовательском опыте, помогая пользователям ориентироваться на платформе и получать информацию о взаимодействиях, таких как лайки, комментарии или новые подписчики.

1. Система уведомлений: Уведомления информируют пользователей о различных взаимодействиях с их контентом или профилем. Например, когда кто-то лайкает пост, оставляет комментарий, или новый пользователь начинает подписываться на профиль. Система уведомлений помогает пользователям быть в курсе активности на их аккаунте и стимулирует к дальнейшему взаимодействию.

2. Маршруты для уведомлений: Для уведомлений создаются маршруты, которые позволяют пользователю получать список своих уведомлений. Эти маршруты должны обеспечивать доступ к данным о новых подписках, лайках и комментариях.

3. Контроллер для уведомлений: Контроллер будет отвечать за получение уведомлений из базы данных и возврат их пользователю в удобном формате. Он будет извлекать уведомления из соответствующей коллекции MongoDB и отправлять их пользователю через API.

IT CAREER HUB

Клиент

Вы уже знакомы с React и имеете опыт работы с ним: мы писали компоненты, использовали хуки, создавали маршруты, и работали с состоянием через `useState` и `useEffect`. В этом проекте на React будем продолжать использовать эти же концепции для реализации интерфейса, включая работу с постами, профилем пользователя, комментариями, лайками и чатом. Основное внимание в этом разделе уделим тому, как загружать изображения для постов и как подключать WebSocket для чата.

Реализация input для загрузки изображения

Одной из ключевых задач нашего приложения является возможность пользователям создавать посты с изображениями. На бекенде мы уже настроили логику, которая преобразует изображение в формат base64 для хранения. На фронтенде нам нужно настроить элемент, через который пользователи будут загружать изображения, и отправлять их на сервер.

Для этого на странице создания поста создадим обычный HTML-элемент `<input>` с атрибутом `type="file"`. Он позволяет пользователю выбирать изображение с устройства.

После выбора файла его нужно отправить на сервер. Мы создадим состояние в компоненте для хранения выбранного файла и реализуем функцию для отправки данных на наш бекенд через запрос `POST`. Важно отметить, что отправка файла будет осуществляться с использованием `FormData`, что позволяет удобно работать с файлами.

Пример кода:

```
const [selectedFile, setSelectedFile] = useState(null);

const handleFileChange = (e) => {
  setSelectedFile(e.target.files[0]);
};

const handleSubmit = async () => {
  const formData = new FormData();
  formData.append('image', selectedFile);

  try {
    const response = await fetch('/api/posts', {
      method: 'POST',
      body: formData,
    });

    if (!response.ok) {
      throw new Error('Ошибка при загрузке изображения');
    }

    const result = await response.json();
    console.log('Изображение успешно загружено', result);
  } catch (error) {
    console.error('Ошибка:', error);
  }
};
```

Пользователь выбирает изображение, а мы отправляем его на сервер, где оно уже преобразуется в формат base64 и сохраняется.

Подключение к WebSocket при открытии чата

Следующий важный функционал, который нужно реализовать на фронтенде, — это работа с WebSocket. Для чатов важно, чтобы сообщения передавались в реальном времени, и для этого мы будем использовать WebSocket. Мы подключаемся к серверу через WebSocket, когда пользователь открывает чат.

Каждый чат будет представлять собой отдельную комнату, к которой мы подключаемся. Это значит, что при открытии чата с конкретным пользователем мы должны присоединиться к определённой комнате, чтобы получать сообщения именно этого чата.

На фронтенде логика подключения выглядит следующим образом:

- Когда компонент чата загружается (пользователь открывает чат), мы инициируем подключение к WebSocket через `useEffect`.

- Подключаемся к определённой комнате с помощью события `joinRoom`, используя идентификатор чата.
- Слушаем входящие сообщения с помощью обработчика события `message`, и обновляем список сообщений на экране.

Пример:

```
useEffect(() => {
  const socket = io('http://localhost:4000');
  const roomId = chatId; // идентификатор чата

  socket.emit('joinRoom', roomId);

  socket.on('message', (message) => {
    setMessages((prevMessages) => [...prevMessages, message]);
  });

  return () => {
    socket.emit('leaveRoom', roomId);
    socket.off();
  };
}, [chatId]);
```

Таким образом, подключение к WebSocket происходит автоматически, когда пользователь открывает чат. Мы также отключаемся от комнаты при закрытии чата, что экономит ресурсы и предотвращает избыточное количество соединений.

Заключение

С загрузкой изображений всё просто: мы используем `<input type="file" />` и отправляем файл на сервер для дальнейшей обработки.

Подключение к WebSocket происходит при открытии чата, что позволяет нам в реальном времени отправлять и получать сообщения.

Все эти задачи легко реализуются с помощью уже знакомых вам инструментов в React, что ускорит работу над проектом и обеспечит эффективное взаимодействие между фронтендом и бекендом.

IT CAREER HUB

Критерии оценки

Во время защиты преподаватель даст комментарии по вашей работе, а затем (во внеурочное время выставит оценку за проект в LMS).

1 — (отлично) (sehr gut) (very good, excellent!) — 81-100%
2 — (хорошо) (gut) (good, well above average!) — 61-80%
3 — (удовлетворительно) (befriedigend) (satisfactory, average!) — 41-60%
4 — (неудовлетворительно) (ausreichend) (sufficient, borderline!) — 21-40%
5 — (неудовлетворительно) (nicht ausreichend) (not sufficient, failed!) — до 20%

Для более объективной оценки мы подготовили таблицу критериев. Ориентируйтесь на них при подготовке проекта, используйте как чек-лист:

Критерий	1	2	3	4	5
Архитектура и соответствие макету	Структура проекта организована (config, routes, controllers, models и т.п.), компоненты в основном переиспользуемые, **интерфейс визуально соответствует Figma**, стили и адаптивность соблюдены.	В целом структура правильная, макет реализован с незначительными отклонениями (цвет, отступы, шрифты), навигация работает.	Основные страницы соответствуют макету, но видны серьёзные отклонения (компоненты, сетка, кнопки).	Частичное совпадение с макетом, нарушена логика интерфейса и визуальный стиль.	Архитектура не организована, макет не учтён.
Авторизация, профиль и безопасность (JWT, bcrypt)	Регистрация, логин, валидация, JWT реализованы. Пароли хэшируются. Доступ к защищённым маршрутам реализован через middleware.	Не хватает 1-2 функций. JWT-логика корректна.	Базовая авторизация работает, но без безопасности или нестабильно.	Авторизация неполная, логика нарушена.	JWT, логин, защита не реализованы.

Посты и работа с изображениями	Реализованы некоторые CRUD-функции с постами. Данные в основном корректно обрабатываются и сохраняются.	Посты создаются и читаются, но редактирование/удаление частично или с багами.	Добавление постов работает, но изображения или авторство не обрабатываются.	Есть начальная реализация без хранения или отображения.	Работа с постами и изображениями не реализована.
Взаимодействие с контентом (лайки, комментарии, поиск)	Реализованы лайки, комментарии, поиск пользователей, отдельный feed "Explore". Все работает стабильно и возвращает актуальные данные.	Присутствуют ≥2 функции из 3, работают корректно.	Только базовая реализация одного из взаимодействий.	Есть заготовки, но баги и отсутствие логики.	Эти функции не реализованы.
Подписки, уведомления и обмен сообщениями (socket.io)	Работают подписки/отписки, отображаются уведомления (лайки, комментарии, фолловеры).	Реализовано ≥2 компонента (например, подписки и уведомления).	Один функционал реализован стабильно.	Реализовано с ошибками или логически не завершено.	Дополнительный функционал отсутствует.
Качество кода и API (структура, валидация, ошибки)	Код структурирован, маршруты логичны, контроллеры разбиты, присутствует валидация, вся логика покрыта `try-catch`, API документирован или читаем.	В целом читаемо, но есть повторения, частичная обработка ошибок.	Логика работает, но код смешан, нет валидации или повторяемость.	Сильное смешение логики, отсутствуют проверки, нет catch-блоков.	Код не структурирован, API не воспроизводится.
Вопросы преподавателя (вес этого критерия 3)	Ответ логичный и аргументированный. Упоминаются	Возможны 1–2 несущественные ошибки, упрощения или неполное	Ответы неуверенные, студент теряется, но с подсказками	Видно, что студент знает, к какой части проекта относится	Ответы типа «не знаю», «этим не я занимался», «не понял, что

	конкретные аспекты проекта, используются корректные термины, нет неточностей.	раскрытие ответа. Студент всё равно показывает хорошее понимание темы.	выходит на суть. Используются обобщённые формулировки, мало конкретики.	вопрос, но затрудняется в деталях. Может дать общую догадку, назвать инструмент или этап, но не раскрыть суть.	это». Нет даже общего представления, с чем связан вопрос. Подсказки не помогают.

7 критериев, 6 из них вносят равный вклад, а ответы на вопросы преподавателя учитываются трижды..

Оценка за проект = $\frac{\text{сумма оценок по каждому критерию}}{9}$, округляется до 0.5 включительно в меньшую сторону.

Пример:

Критерий	**Оценка**
Архитектура и соответствие макету	1
Авторизация, профиль и безопасность (JWT, bcrypt)	2
Посты и работа с изображениями	2
Взаимодействие с контентом (лайки, комментарии, поиск)	1
Подписки, уведомления и обмен сообщениями (socket.io)	3
Качество кода и API (структура, валидация, ошибки)	1
Вопросы преподавателя	2 (*3)
Сумма баллов	**16**
Итоговая оценка	**2**

Важно: качественная реализация дополнительных функций может положительно повлиять на оценку, если основной функционал имеет **незначительные недочеты** (на усмотрение преподавателя можно вычесть из оценки до округления **0,1 балла**).