МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ПОДІЙНО-ОРІЄНТОВАНОГО ВЕБЗАСТОСУНКУ ЗАСОБАМИ REACT ТА NODE.JS»**

Виконав: студент 4 курсу, групи 6.1219-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
І.С. Тюшняков
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, PhD, Столярова А.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення

(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.

(підпис)

« 07 » 02 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Тюшнякову Іллі Сергійовичу

(прізвище, ім'я та по-батькові)

1. Тема роботи (проєкту) Розробка подійно-орієнтованого вебзастосунку засобами React та Node.js

керівник роботи (проєкту) Столярова Анастасія Валеріївна, PhD

(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 26 » січня 2023 року № 102-с

2. Строк подання студентом роботи 07.06.2023

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка подійно-орієнтованого вебзастосунку засобами React та Node.js.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

презентація за темою докладу

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Кудін. О.В., доцент кафедри програмної інженерії		
2	Кудін. О.В., доцент кафедри програмної інженерії		
3	Кудін. О.В., доцент кафедри програмної інженерії		

7. Дата видачі завдання 07.02.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.02.2023	
2.	Збір вихідних даних.	13.02.2023	
3.	Обробка методичних та теоретичних	27.02.2023	
	джерел.		
4.	Розробка першого та другого розділу.	11.04.2023	
5.	Розробка третього розділу.	16.05.2023	
6.	Оформлення та нормоконтроль	01.06.2023	
	кваліфікаційної роботи.		
7.	Захист кваліфікаційної роботи.	21.06.2023	

Студент _____ І.С.Тюшняков
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.В. Столярова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка подійно-орієнтованого вебзастосунку засобами React та Node.js»: 87 с., 43 рис., 11 джерел, 2 додатки.

БЛОГ, ВЕБДОДАТОК, РОЗРОБКА ВЕБДОДАТКІВ, EXPRESS, MONGODB, REACT, NODE.JS.

Об’єкт дослідження – використання React та Node.js для розробки вебзастосунку.

Мета роботи: дослідження процесу розробки вебзастосунку та розробка блогу засобами React та Node.js.

Методи дослідження – методи програмної інженерії.

У роботі проведено дослідження процесу розробки вебзастосунку з використанням React та Node.js. Наведено аналіз та аргументацію вибору інструментів, визначено вимоги до вебзастосунку, побудовано діаграми та схеми, створено ескізи дизайну, розроблено фронтенд і бекенд блогу та протестовано додаток.

Результати роботи можуть бути використані для дослідження розробки фулстек вебзастосунків.

SUMMARY

Bachelor’s Qualifying Paper «Development of an Event-driven Web Application using React and Node.js»: 87 pages, 43 figures, 11 references, 2 supplements.

BLOG, WEB APPLICATION, WEB APPLICATION DEVELOPMENT, EXPRESS, MONGODB, REACT, NODE.JS.

The object of the study is using React and Node.js for development of a web application.

The aim of the study is to research process of development of a web application and development of a blog using React and Node.js.

The methods of research are methods of software engineering.

Work contains analysis of the process of development a web application using React and Node.js. The analysis and reasoning of the choice of tools are presented, the requirements for the web application are determined, diagrams and schemes are built, design sketches are created, front-end and back-end of the blog are developed, and application is tested.

Results of the work can be used for research of development of full-stack web applications.

ЗМІСТ

ВСТУП

У наш час технології швидко розвиваються, роблячи широкі кроки, так само швидко, як і змінюється світ навколо нас. Кожен день відбувається велика кількість подій і новин, люди бажають знати та ділитися ними один з одним, щоб надати їм таку можливість – на допомогу приходять інформаційні ресурси, соціальні мережі та блоги.

Отже, дослідження та розробка блогу засобами React і Node.js є актуальною задачею.

Метою кваліфікаційної роботи є дослідження процесу розробки вебзастосунку, а саме блогу засобами React та Node.js.

Задачі, які необхідно розв’язати для досягнення поставленої мети:

1) проаналізувати існуючі засоби розробки вебзастосунків;
2) визначити необхідний функціонал для розробки блогу;
3) розробити макети дизайну сторінок;
4) розробити фронтенд частину блогу, використовуючи визначені інструменти;
5) розробити бекенд блогу;
6) протестувати розроблений вебзастосунок.

Об’єкт дослідження – використання React та Node.js для розробки вебзастосунків.

Методи дослідження – методи програмної інженерії.

Структурно курсова робота складається з таких компонентів: вступ, 3 розділи, висновки, 2 додатки, перелік посилань з 11 найменувань.

В першому розділі проведено аналіз існуючих засобів розробки вебзастосунків, визначено необхідний функціонал розроблюваного вебдодатку. В другому розділі було наведено процес проєктування блогу. В третьому розділі показано процес розробки вебзастосунку, наведено детальні пояснення функціоналу роботи блогу.

1 АНАЛІЗ ВИМОГ ДО ЗАСТОСУНКУ

1.1 Аналіз засобів розробки вебзастосунків

Технології швидко розвиваються у наш час, кожен день розробляють нові вебдодатки, вони нас оточують і ми з ними взаємодіємо кожен день, наприклад онлайн-магазини, стрічка новин, стрімінгові сервіси, блоги, соціальні мережі, різні сайти та багато подібних. Галузь веб-розробки розширюється, стає глибшою, дорожчою та складнішою, замовники потребують все більшого і ширшого функціоналу від застосунків, тому розробники відповідально ставляться до вибору якісних актуальних інструментів, щоб добре та вчасно виконувати задачі, котрі ставить перед ними замовник. На даний момент дуже популярною та актуальною галуззю є вебдодатки – для їх розробки створюють нові технології, фреймворки, мови програмування, або покращують те, що вже було розроблено, розширюючи можливості або функціонал інструментів.

Найбільш відомими та актуальними засобами розробки вебдодатків є мова програмування JavaScript, бібліотека React, Node.js, система управління базами даних MongoDB, каскадна таблиця стилів для роботи з візуалом SCSS та допоміжний фреймворк для роботи з серверною частиною Express.

Стек технологій, який складається з MongoDB, Express, React та Node, називають MERN стек, це абревіатура складена з перших літер назв технологій, у такому наборі інструментів є все, щоб розробити зовнішню частину застосунку, серверну частину, та базу даних для збереження і використання інформації. MERN дозволяє розробляти якісні та багатофункціональні сучасні вебдодатки.

Кожен інструмент стеку технологій MERN є необхідним для розробки сучасного повноцінного вебзастосунку.

Першим інструментом є MongoDB – це NoSQL база даних, що

використовує документи і колекції, а не таблиці, як реляційні БД, швидко видає відповіді з серверу, також є можливість мати доступ до інформації без коду, через графічну оболонку Compass. В цій БД можливо змінювати початкову структуру даних, вона дозволяє зберігати інформацію у вигляді JSON-подібних об’єктів, що є дуже зручним з практичної точки зору та візуально. MongoDB має сервери в різних місцях світу, тому для швидкості роботи є можливість створювати БД якнайближче для зменшення часу обробки запитів. MongoDB використовують такі компанії як Forbes та Ebay.

Другим інструментом стеку є Express – це фреймворк на базі Node для розробки серверної частини додатку. Express надає тонкий шар фундаментальних функцій веб-застосунків, які не заважають працювати з давно знайомими функціями Node.js. Він спрощує та скорочує код, має велику швидкість роботи, завдяки Express є можливість налагоджувати маршрутизацію, а також створюється API сервера для взаємодії фронтенда та бекенда. Фреймворк, маючи у своєму розпорядженні безліч службових методів HTTP та проміжних обробників, дозволяє створити надійний API швидко та легко [9]. Компанія Uber використовує цей фреймворк для свого додатку.

Третім інструментом стеку MERN є React, це бібліотека для мови програмування JavaScript. React має компоненти, що прискорюють розробку додатків, а саме інтерфейсу, використовуючи шаблони та функціональну мову програмування. Через динамічний рендерінг окремих, потрібних у той або інший час, частин додатку прискорюється його робота, React може замінити HTML, так як сам має функціонал для створення розмітки сторінки. Компанії, які використовують React у своїх додатках: Facebook, Instagram, E-Bay та Sony.

Четвертим інструментом стеку є Node.js, це асинхронне подійне JavaScript-оточення, Node.js спроєктований для побудови мережевих додатків, що масштабуються [10]. Node також дає можливість створювати інструменти та серверні програми за допомогою JavaScript, тобто Fullstack-додатки, завдяки цій програмній платформі у розробника з’являється можливість

працювати з файловою системою, чого не зробити простим JavaScript.

Окрім стеку MERN використовується JavaScript, це мова програмування, яка використовується для створення скриптів у веб-сторінках, але також застосовується для серверної розробки, використовуючи пакет як Node.JS. Останнім часом популярність JavaScript збільшилась завдяки успішній платформі Node.js – найпопулярнішому кроссплатформеному середовищі виконання JavaScript поза браузером. Node.js дозволяє розробникам використовувати JavaScript як мову сценаріїв для автоматизації роботи на ПК та створення повнофункціональних HTTP та WebSocket-серверів [10].

Для налагодження дизайну та гарного зовнішнього вигляду сторінок у додатку використовується SASS – це мова таблиць стилів, скомпільована з CSS. Він дозволяє використовувати змінні, вкладені правила, міксини, функції тощо з повністю сумісним із CSS синтаксисом. SASS допомагає добре організовувати великі таблиці стилів і дозволяє легко ділитися дизайном в проєктах і між ними [8]. SASS є дуже популярним і ефективним інструментом для задання стилів.

Для розробки якісного вебдодатку мови програмування та фреймворків не достатньо, тому слід використовувати деякі NPM пакети, бібліотеки, наприклад jsonwebtoken для створення токенів доступу та для передачі даних при аутентифікації в клієнт-серверних додатках.

Bcrypt – це бібліотека, яка допоможе хешувати та шифрувати паролі.

Mongoos – це зручна бібліотека, яка створює зв’язок між MongoDB та середовищем виконання JavaScript Node, без неї неможливо використовувати бази даних MongoDB.

Для завантаження файлів на сервер необхідна бібліотека Multer – це middleware для фреймворку express, Multer обробляє тільки тип форми multipart/form-data.

Таким чином, на основі проведеного аналізу популярних існуючих інструментів, бібліотек та фреймворків, зважаючи на всю вище зазначену

інформацію, для розробки власного подійно-орієнтованого блогу буде використовуватися популярний, потужний, ефективний та зручний стек технологій MERN, для надання гарного вигляду сторінкам буде використано SASS і бібліотеки з npm пакетами, такі як mongoos, jsonwebtoken, bcrypt та multer для розробки решти необхідного функціоналу.

1.2 Визначення функціональних вимог

До початку розробки програмного забезпечення насамперед необхідно визначити вимоги до майбутнього проєкту, а саме які функції він буде виконувати та що за можливості надавати користувачам програмного продукту.

Додаток вважається подійно-орієнтованим, коли програма або її компоненти змінюються у відповідь на дії користувача. Також саме подійно-орієнтоване програмування визначається як спосіб розробки програми, де головним циклом є отримання повідомлення про подію та сама обробка події.

Метою роботи є розробка подійно-орієнтованого вебдодатку, а саме блогу з використанням React, Node.js та інших технологій, для досягнення цієї мети повинен бути розроблений конкретний функціонал притаманний сучасним вебзастосункам.

В першу чергу розробляється бекенд частина, а саме:

- створення бази даних для зберігання інформації та користувачів;
- HTTP запити на вебзастосунок;
- функціонал для реєстрації;
- валідація даних при реєстрації;
- шифрування паролів з використанням бібліотеки bcrypt;
- підключення до бази даних;
- авторизація;

– функціонал для створення статей;
– можливість завантаження зображень на сервер.

Після розробки основного серверного, схованого від очей користувача, функціоналу йде не менш важливий другий етап – створення візуальної та подійно-орієнтованої частини продукту, яка буде зв'язана з бекендом. Фронтенд, це те, що буде бачити користувач та з чим він буде взаємодіяти, у цій частині будуть такі вимоги до проєкту, як:

– створення React додатку;
– розробка каркасу сторінок блогу jsx розміткою;
– задання стилів SCSS для гарного вигляду;
– підключення бібліотеки для створення HTTP запитів Axios;
– роути для сторінок;
– зв’язання функціоналу для авторизації з бекендом;
– збереження токена користувача в LocalStorage;
– допрацювання форми реєстрації;
– форма створення статті з використанням редактору тексту SimpleMDE;
– функціонал для посилання запиту на видалення статті;
– завантаження зображення з використанням бібліотеки Multer;
– відправка статті на сервер;
– редагування статей користувачем;
– перевірка введених в форму реєстрації даних засобами валідації.

Після задовільненя всіх вище заначених вимог з двох етапів розробки, користувач матиме такі можливості у подійно-орієнтованому вебдодатку, як:

– реєстрація;
– авторизація;
– вихід з акаунту;
– додавання статей;
– перегляд статей;

– завантаження зображення для статті;

– редагування статей;

– видалення своєї статті;

– відсутність можливості редагування та видалення статей інших користувачів;

– додавання тегів для статті;

– перегляд останніх п’яти тегів з нових статей.

Таким чином було визначено та описано функціональні вимоги до майбутнього блогу, який буде відповідати усім вимогам вебдодатку розробленого засобами React та Node.js з використанням бази даних MongoDB і з заданням стилів для сторінок за допомогою SASS. Процес проєктування, розробки дизайну та реалізації всіх вище зазначених функцій додатку буде описано в наступних розділах.

1.3 Висновки до розділу 1

В розділі 1 було проведено аналіз існуючих сучасних технологій для розробки вебдодатків, було обрано та описано засоби React, Node.js, а також необхідні для створення повноцінного додатку інструменти, як SASS, Mongodb та Express. Також у цьому розділі було описано вимоги до проєкту, наведено визначення подійно-орієнтованого додатку, описано його функціонал і можливості користувача у готовому продукті.

2 ПРОЄКТУВАННЯ ВЕБДОДАТКУ

2.1 Проєктування структури вебдодатку

Після вибору інструментів для розробки вебдодатку, а також визначення функціональних вимог і можливостей користувача у вебзастосунку, наступним етапом йде проєктування.

Перш за все необхідно розробити структурну схему, її суть в тому, щоб показати ієрархію та набір компонентів сайту, а також в цілому структуру додатку (рис. 2.1).

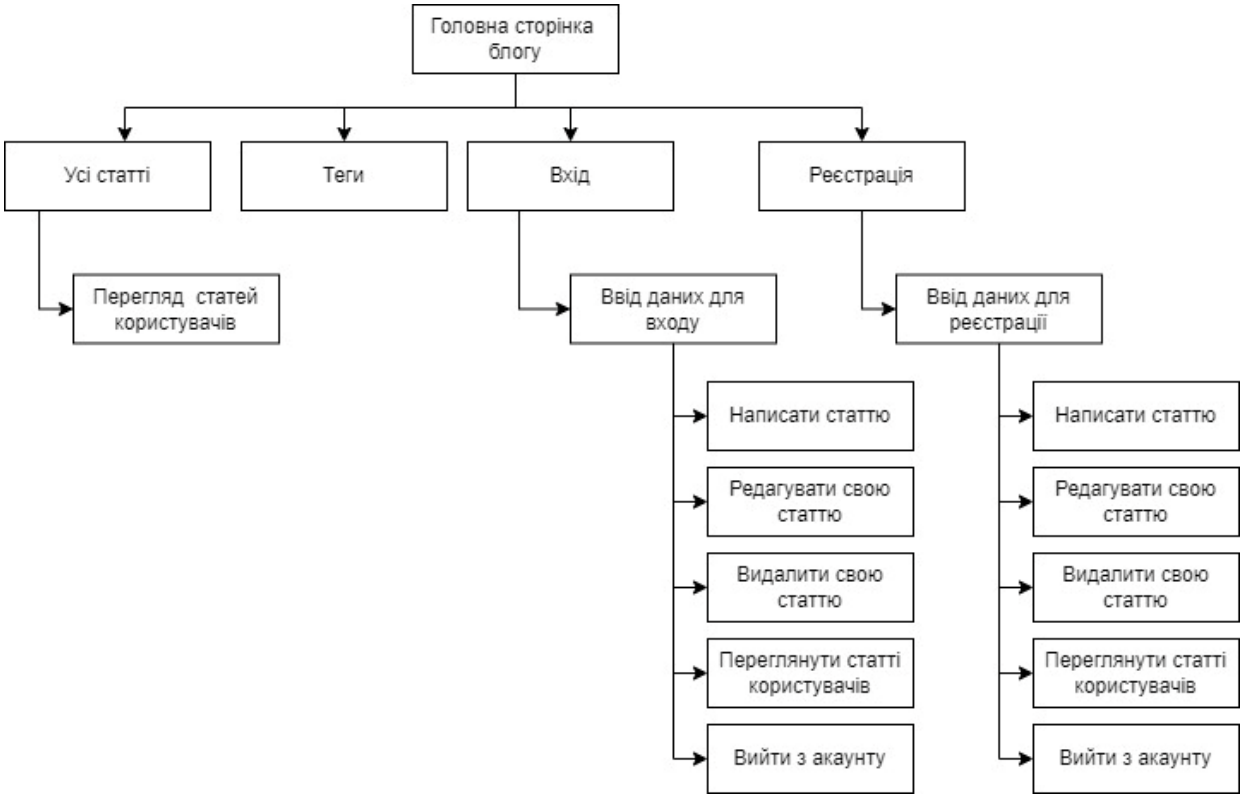

Рисунок 2.1 – Структурна схема додатку

Додаток вважається подійно-орієнтованим, коли програма або її компоненти змінюються у відповідь на дії користувача, як було визначено в розділі 1, тому далі було розроблено діаграму активності, її використовують для демонстрації взаємодії користувача з продуктом і результатів після

конкретних дій, та можливостей, які доступні користувачу. Діаграма також показує як змінюється стан компонентів при взаємодії з ними.

Діаграма активності демонструє, як користувач взаємодіє з додатком, що він може робити та як буду змінюватися компоненти застосунку у відповідь на певні дії користувача (рис. 2.2).

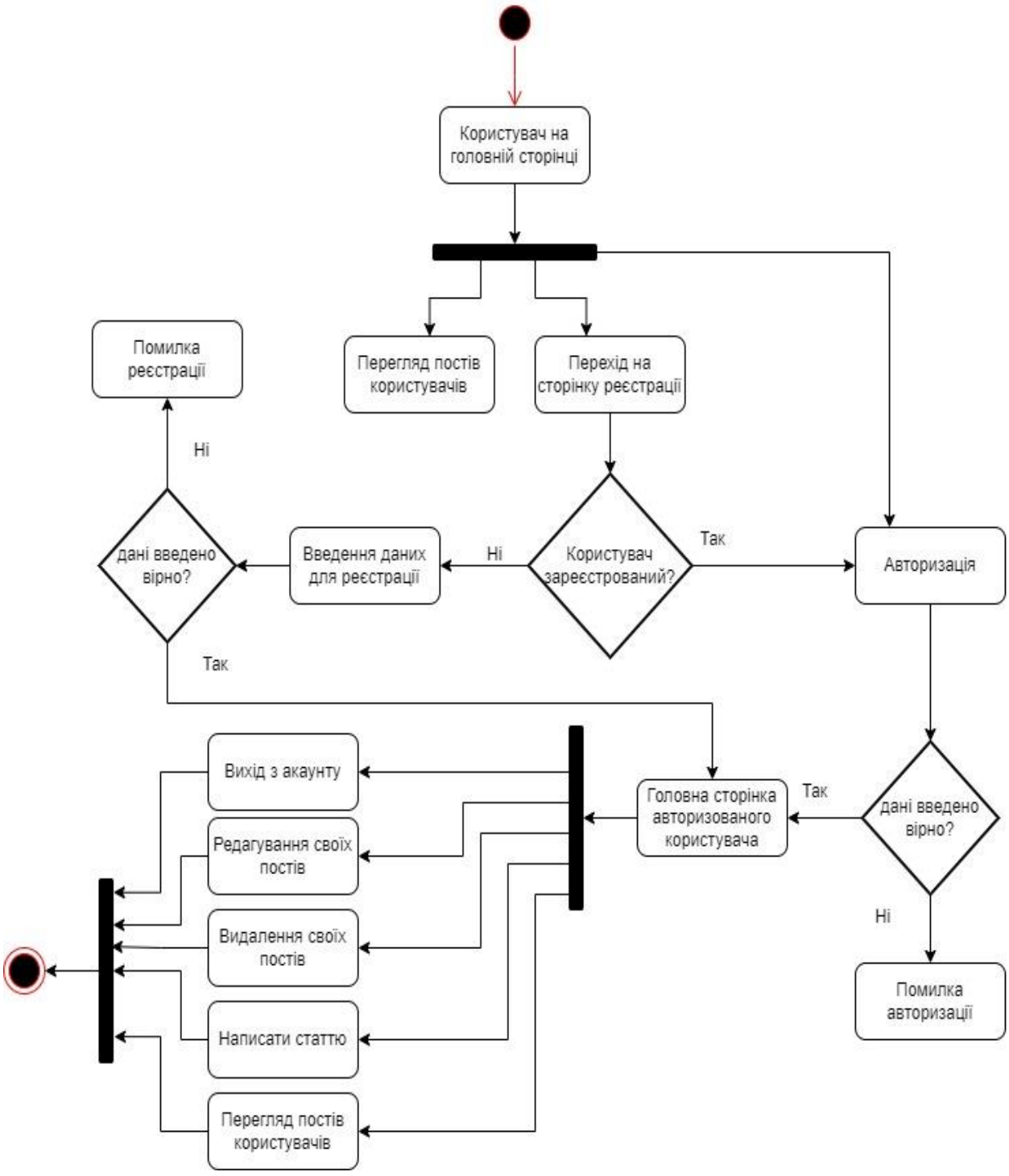

Рисунок 2.2 – UML діаграма активності додатку

Далі представлено діаграму, яка відображає принцип роботи майбутнього функціоналу реєстрації та авторизації користувача у вебдодатку (рис. 2.3).

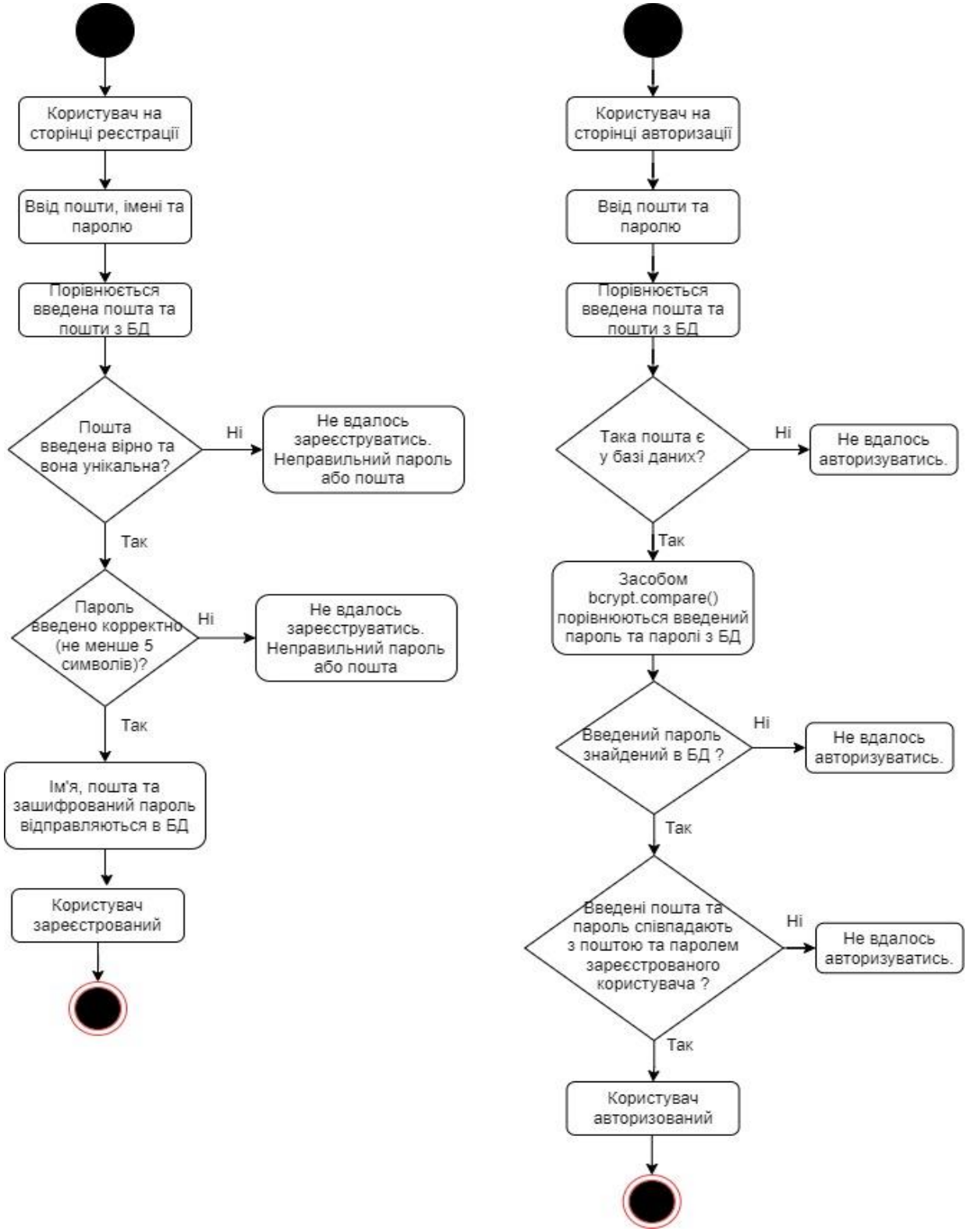

Рисунок 2.3 – Принцип роботи реєстрації та авторизації в додатку

Також в процесі проєктування необхідно створити схему NoSQL бази даних – оскільки вона складається з документів, а не з таблиць, на рисунку 2.4 зображено схеми двох необхідних для роботи додатку документів: користувачів та постів.

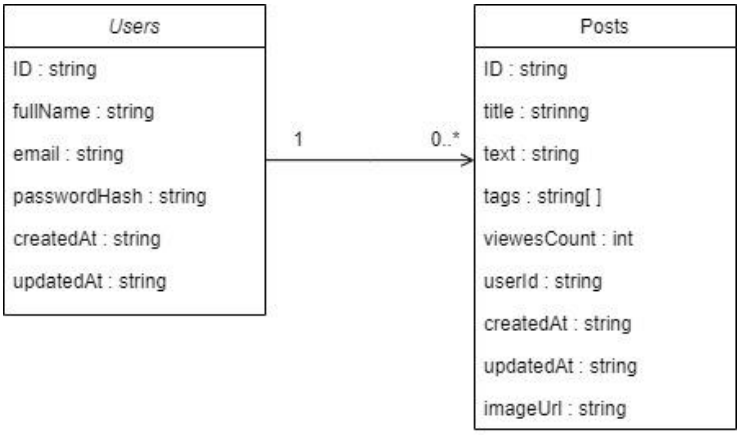

Рисунок 2.4 – Схема бази даних додатку

Для повноцінного розуміння принципів побудови додатку, його компонентів та взаємозв’язку між ними спроєктовано діаграму компонентів. Ця діаграма відображає, які компоненти будуть створені та використані при розробці вебдодатку та як вони будуть пов’язані (рис. 2.5).

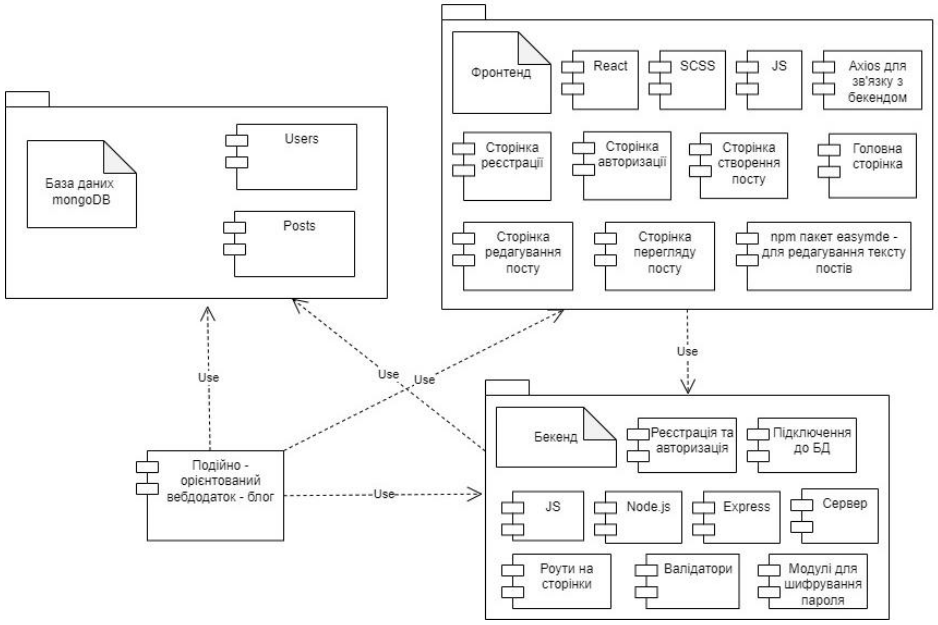

Рисунок 2.5 – Діаграма компонентів додатку

В результаті було спроєктовано структурну схему блогу для візуалізації компонентів, його діаграму активності, схему роботи функціоналу реєстрації та авторизації, схему документів бази даних, а також діаграму компонентів. Побудовані схеми та діаграми показують, як будуть задовільнені усі вимоги до проєкту.

2.2 Розробка дизайну сторінок блогу

Після побудови діаграм та схем, для початку розробки додатку треба створити скетчі дизайну сторінок блогу.

Розробка скетчів сторінок необхідна для верстки вебдодатку, вона показує положення кнопок на сторінці, розташування певних елементів, та в цілому якою приблизно повинна бути верстка. Скетч не є відображенням кінцевого вигляду сторінок, він лише задає напрямок в якому буде рухатись розробник верстки, тож кінцевий дизайн може зазнати деяких змін.

Першим скетчем є головна сторінка блогу, на ній мається схематично зображене положення кнопок реєстрації, авторизації, положення статей та вікна з тегами (рис. 2.6).

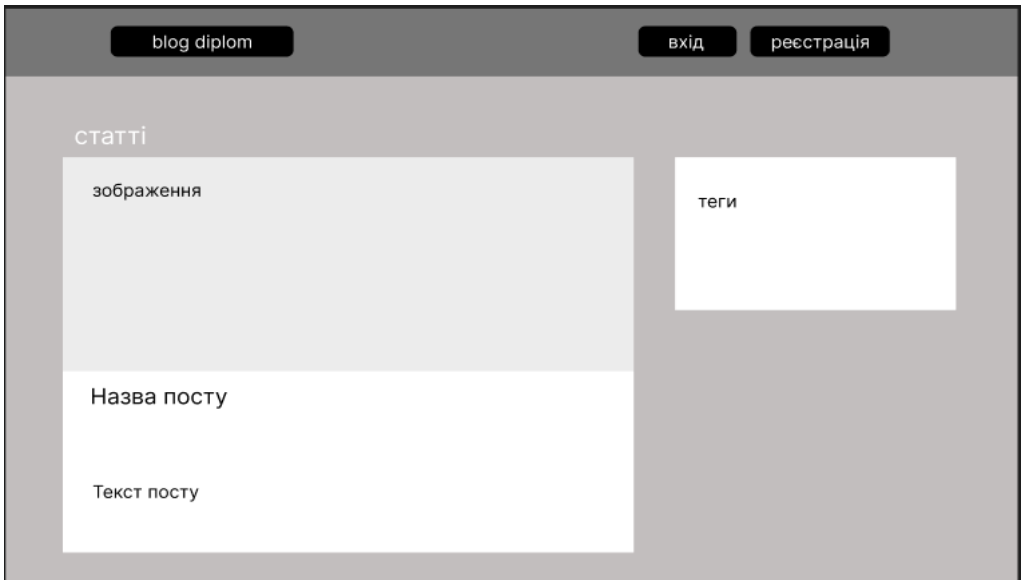

Рисунок 2.6 – Скетч головної сторінки блогу

Далі скетч перегляду посту, на ньому відображено схематичний вигляд відкритої статті, є поле для зображення, назви і тексту (рис. 2.7).

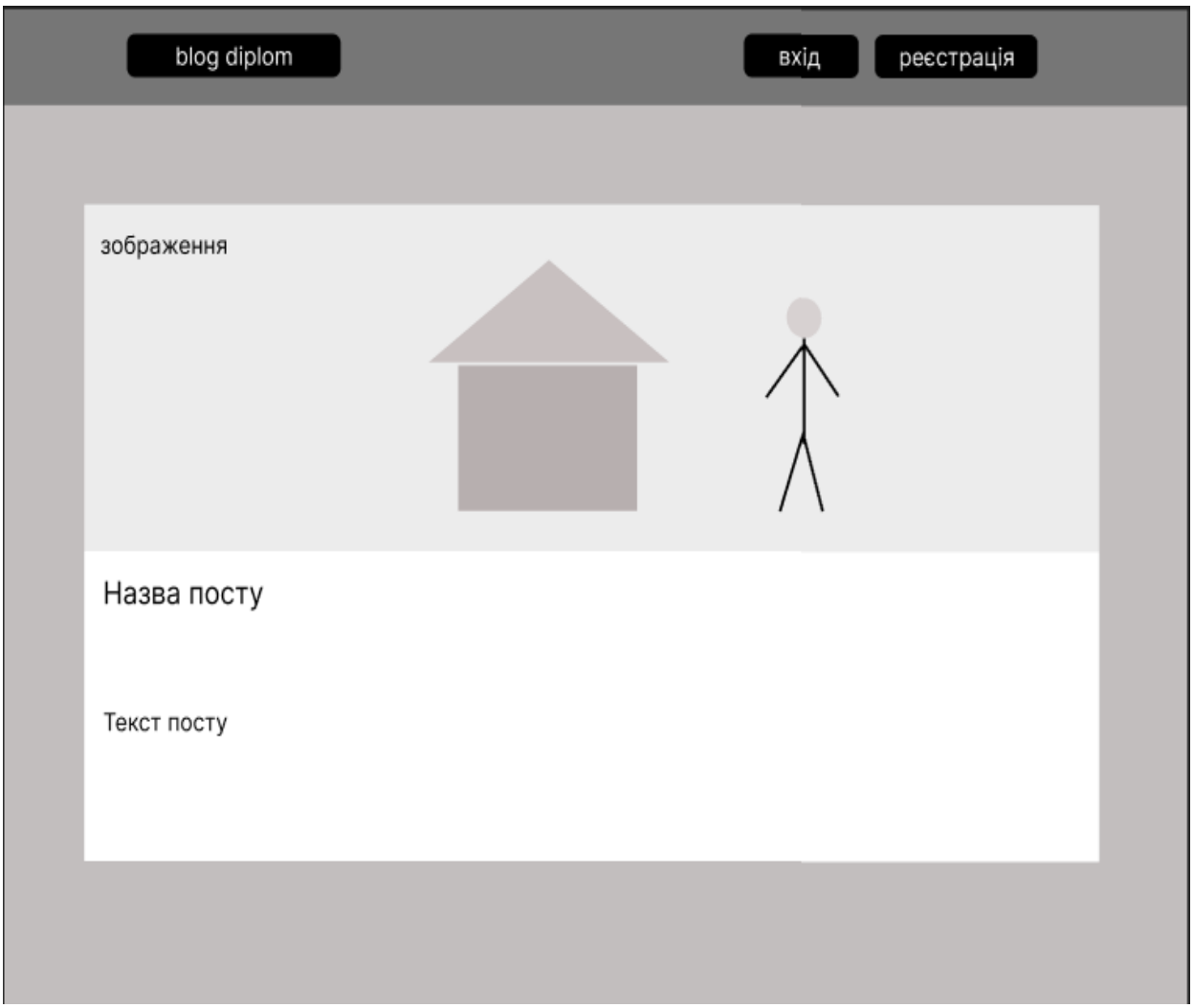

Рисунок 2.7 – Скетч сторінки перегляду посту

Скетч створення статті відображає поле вводу заголовку для посту, місце для головного тексту, кнопку для завантаження зображення, яке буде відображатись при відкритті статті або на головній сторінці вибора публікації, кнопки «опублікувати» та «скасувати», що буде скасовувати публікацію або редагування.

Такий самий вигляд буде мати і сторінка редагування статей, за виключенням кнопки «опублікувати», її у подальшій розробці буде замінено на кнопку «зберегти» (див. рис. 2.8).

Рисунок 2.8 – Скетч сторінки створення статті

Скетч сторінки авторизації, на ньому зображено форму для вводу даних, таких як пароль та логін, а також кнопка входу (рис. 2.9).

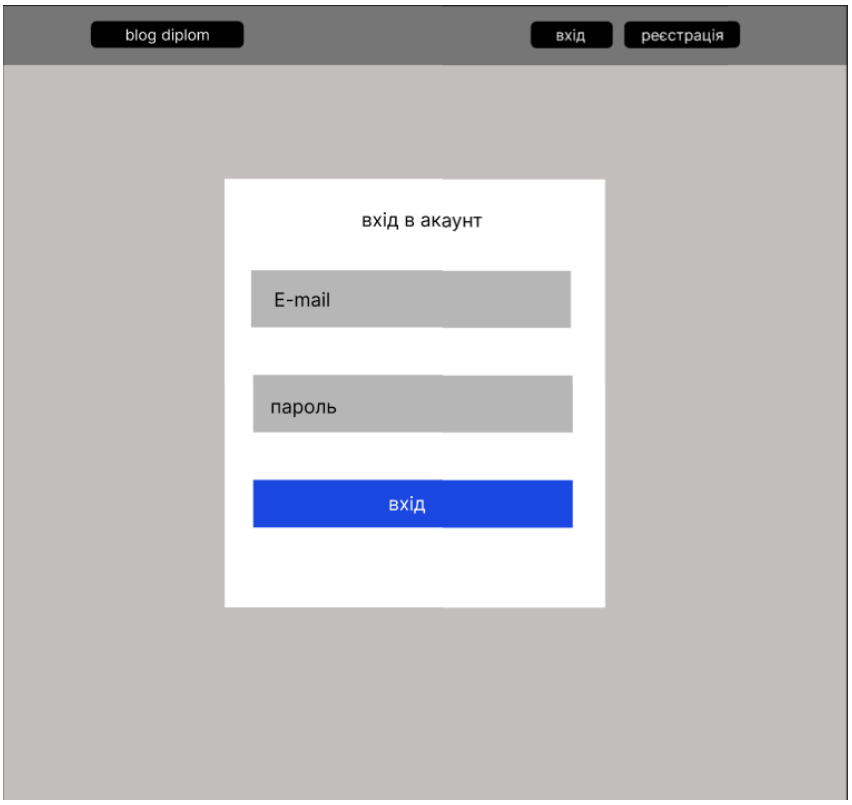

Рисунок 2.9 – Скетч сторінки авторизації користувача

Останнім скетчем є сторінка реєстрації, схожа на скетч авторизації за виключенням додаткового поля вводу для повного імені (рис. 2.10).

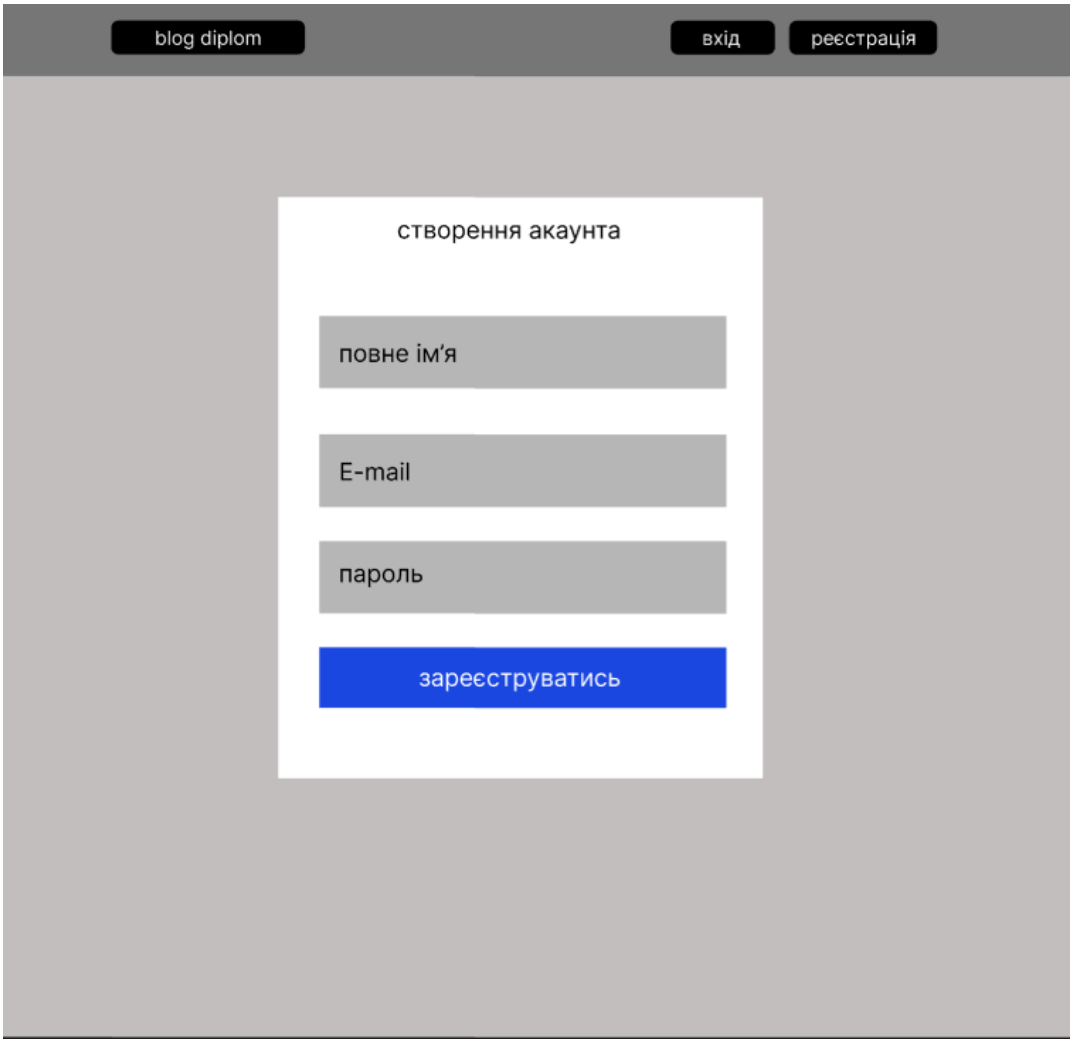

Рисунок 2.10 – Скетч сторінки реєстрації користувача

Таким чином у розділі 2.2 було розроблено скетчі, схематичні зображення майбутньої верстки сторінок вебдодатку. Завдяки їм верстальник буде розуміти, які сторінки треба розробити, а також мати уяву про приблизний вигляд блогу, що полегшить написання коду розмітки, задання стилів елементам та дасть розуміння де і які елементи повинні розташовуватись.

2.3 Висновки до розділу 2

У розділі 2 було спроєктовано майбутній вебдодаток за допомогою UML діаграми активності, структурної схеми для демонстрації наповнення та функціоналу, розроблено схему принципу роботи реєстрації та авторизації в додатку, розроблено схему бази даних та діаграму компонентів вебзастосунку.

Діаграма активності показує, яким чином буде взаємодіяти з додатком користувач і як саме працює вебзастосунок. Структурна схема відображає сторінки та компоненти блогу. Схема реєстрації та авторизації показує як проходить процес авторизування або створення акаунта та в який момент пароль шифрується і відправляється в базу даних. Для демонстрації компонентів додатку, з чого саме складається вебзастосунок, було створено відповіду діаграму. Також було розроблено скетчі дизайну проєкту, для полегшення верстки та конкретизації і візуалізації вимог до приблизного зовнішнього вигляду сторінок і розташування елементів на них.

Таким чином розроблені скетчі, UML-діаграми та схеми демонструють, як будуть реалізовані вимоги до проєкту, які були визначені у підрозділі 1.2.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація бекенд частини додатку

В попередніх розділах було проведено аналіз технологій та обрані такі інструменти, як Node, React, Express, MongoDB, SASS, якими буде розроблено додаток. Були описані вимоги до проекту та функціоналу. В розділі 2 було спроектовано майбутній додаток за допомогою діаграми активності та структурної схеми, також розроблено скетчі дизайну сторінок вебзастосунку. Таким чином, після всіх вище зазначених дій все готово для розробки блогу.

Першим кроком буде розробка бекенд частини для можливості взаємодії блогу с сервером: буде реалізовано реєстрацію, відправку даних на сервер, завантаження URL-адреси зображень в базу даних та створено саму БД для зберігання інформації про майбутніх користувачів та пости.

Першим кроком буде створено проєкт, він буде ініціалізований за допомогою команди «npm init», після виконання команди з’явиться папка «node modules», в якій в майбутньому будуть зберігатись файли для роботи встановлених пакетів, фреймворків та модулей. Після виконання команди «npm init», також з’являться два файли, «package.json» та «package-lock.json», після ініціалізації проєкту необхідно встановити потрібні пакети та бібліотеки для подальшої розробки бекенд частини додатку. Будуть встановлені такі залежності, як: bcrypt для шифруваняя паролів (для того, що б зловмисник, який зможе отримати доступ до даних з БД, не зміг використати акаунти користувачів, тому що в БД паролі будуть зберігатися зашифрованими, а розшифровуватись тільки для авторизації користувача), cors для надання можливості переходити на сторонні роути, фреймворк express для розробки серверної частини додатку, express-validator для перевірки правильності введених даних, jsonwebtoken для створеня токенів та для корегування ним можливостей користувача, бібліотека mongoos необхідна для роботи з базою

даних на mongodb, multer для завантаження файлів на сервер та nodemon, котрий буде перезавантажувати сервер після збереження змін у проєкті. Також в файл було додано скрипт «"start:dev": "nodemon index.js"» для швидкого запуску майбутньої серверної частини додатку, в результаті завантаження залежностей у проєкт засобами команди «npm install» та назви пакету – файл «package.json» буде мати такий вигляд, як на рисунку 3.1.

```
{
  "name": "blog-mern",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "start:dev": "nodemon index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.0.1",
    "cors": "^2.8.5",
    "express": "^4.18.1",
    "express-validator": "^6.14.1",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^6.3.5",
    "multer": "^1.4.5-lts.1",
    "nodemon": "^2.0.16"
  }
}
```

Рисунок 3.1 – Файл package.json

Після підключення залежностей було додано в файл package.json рядок «"type": "module"», завдяки цій дії тепер є можливість імпортувати бібліотеки, фреймворки та пакети за стандартом ES6.

Далі створюємо express-додаток та імпортуємо бібліотеку cors для уникнення помилок при переходах між різними роутами, також було розроблено прослуховування додатку на порт 4444 і додано обробник помилок у разі їх виникнення (див. рис. 3.2).

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

const app = express();//створюємо експресс додаток

app.use(express.json());//використовуємо в додатку json для його читання
app.use(cors());//щоб не було помилок з переходами по різним роутам
app.listen(process.env.PORT || 4444, (err) => {//прикріпили додаток на порт
  if (err) {
    return console.log(err);//якщо сервер не запустився - виводимо помилку
  }

  console.log('my server is OK');//якщо запустився - виводимо повідомлення
});
```

Рисунок 3.2 – Створення Express-додатку

Оскільки спочатку розробляється бекенд частина додатку, то необхідним буде протестувати в майбутньому запити POST, GET, UPDATE, DELETE. У цьому допоможе програма «Insomnia» – завдяки їй з’являється можливість робити запити на бекенд без фронтенд частини, ця програма використовується поки не буде розроблено візуальну частину проекту.

Оскільки розробляється блог з бекендом, то є необхідність зберігати дані користувачів і постів на сервері, для цього необхідно створити базу даних. У попередньому розділі було обрано MongoDB, тому при переході на офіціальний сайт створюємо акаунт, створюємо базу даних обравши безкоштовний тариф на 512 мегабайт, навіть при використанні безкоштовного тарифу – немає потреби піклуватися про захист інформації яка буде зберігатися в базі даних, так як сама mongoDB автоматично шифрує дані, щоб захистити іх від зловмисників. Після виконання вище описаних дій, буде отримано такий результат (див. рис. 3.3).

Після реєстрації та створення бази даних, наступним кроком є її підключення до проєкту. Натиснувши на кнопку «Connect» відкриється вікно, в якому міститься посилання на створену базу даних, після копіювання посилання встановлюємо в додаток бібліотеку mongoose, без неї додаток не буде мати можливості підключатись до бази даних. Наступним кроком буде підключення проєкту до створеної бази даних за допомогою скопійованого

посилання (див. рис. 3.4).

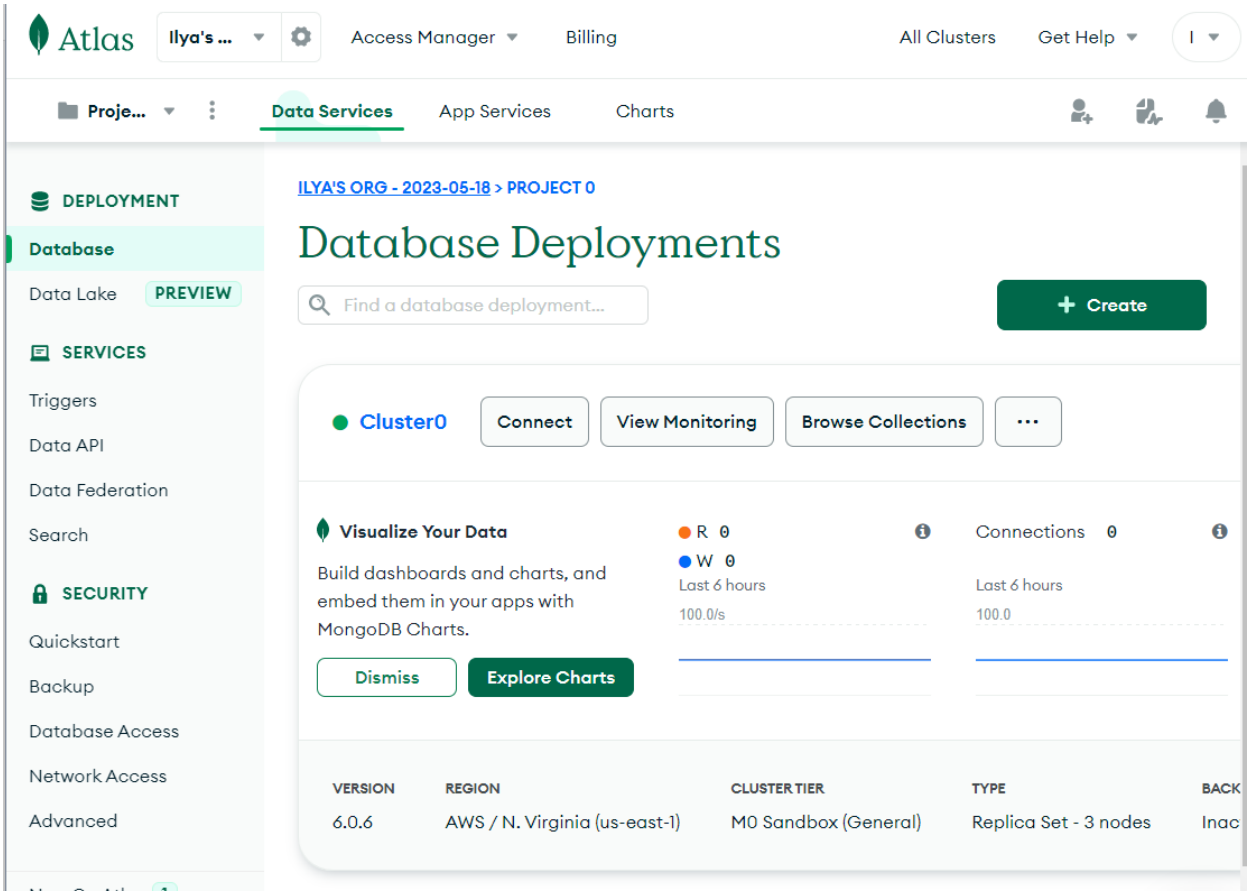

Рисунок 3.3 – Створена база даних

```
import mongoose from 'mongoose';

mongoose        //робота з монгоДБ
  .connect("mongodb+srv://ilyasoooper:littlelesssomepass@cluster0.b94imhr.mongodb
.net/blog?retryWrites=true&w=majority")//посилання на бд
  .then(() => console.log('БД в порядку'))
  .catch((err) => console.log('Помилка БД', err));
```

Рисунок 3.4 – Підключення додатку до бази даних

Було імпортовано бібліотеку, підключено додаток до бази даних за допомогою посилання, також оброблено помилки в разі їх виникнення.

Після підключення бази даних було використано бібліотеку mongoose для створення таблиці структури моделі посту для подальшого використання, модель має такі об’єкти як: заголовок, текст посту, теги, лічильник переглядів, зв’язок між таблицями користувача і статті, посилання на зображення та час

створення посту (рис. 3.5).

```
import mongoose from 'mongoose';

const PostSchema = new mongoose.Schema(//схема посту для зберігання в бд
  {
    title: {
      type: String,
      required: true,
    },
    text: {
      type: String,
      required: true,
      unique: true,
    },
    tags: {
      type: Array,
      default: [],
    },
    viewsCount: {
      type: Number,
      default: 0,
    },
    user: {//ObjectId буде ссилатися на ref: 'User' (конкретного користувача)
      type: mongoose.Schema.Types.ObjectId,//спеціальний тип з бд (id)
      ref: 'User',//створили relationshep - зв'язок між двома таблицями (юзер та пост)
      required: true,
    },
    imageUrl: String,
  },
  {
    timestamps: true,
  },
);
export default mongoose.model('Post', PostSchema);
```

Рисунок 3.5 – Модель посту

Також необхідною частиною блогу є модель користувача, вона також була створена засобами mongoose, таблиця структури має такі поля: повне ім’я, пошта, зашифрований пароль, який буде реалізований далі, та дата створення користувача (рис. 3.6).

```
import mongoose from 'mongoose';

const UserSchema = new mongoose.Schema(//схема користувача для бд
  {
    fullName: {
      type: String,
      required: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    passwordHash: {
      type: String,
      required: true,
    },
    avatarUrl: String,
  },
  {
    timestamps: true,
  },
);

export default mongoose.model('User', UserSchema);//експортуємо схему користувача
```

Рисунок 3.6 – Модель користувача

Після створення моделей користувача та посту вони з’являться в MongoDB, далі буде розроблено функціонал для постів і тегів, наприклад отримання усіх постів, отримання тегів, отримання одного посту по його id та створення посту.

Для того щоб отримати теги з постів було розроблено асинхронну функцію, яка перебирає створені статті, знаходить в них теги та повертає п’ять останніх тегів, якщо база даних не відповідає, то методом «try catch» буде повернута помилка зі статусом 500 (див. рис. 3.7).

```
import PostModel from '../models/Post.js';//передали модель посту

export const getLastTags = async (req, res) => {
  try {
    const posts = await PostModel.find().limit(5).exec();
    const tags = posts
      .map((obj) => obj.tags)//перебрали теги посту
      .flat()//витягнули статті
      .slice(0, 5);//узяли теги  з 5ти останніх статей
```

Рисунок 3.7 – Функція отримання тегів з останніх постів

Далі було створено функцію для отримання усіх створених статей, котра робить асинхронний запит, знаходить усі статті користувачів, повертає їх у відповідь, а якщо є помилка з сервером, то буде отримано відповідне повідомлення. Функція знадобиться далі при створені роуту, котрий після запиту на нього буде повертати усі статті, збережені у базі даних (див. рис. 3.8).

```
export const getAllPosts = async (req, res) => {//отримати усі статті
  try {
    const posts = await PostModel.find().populate('user').exec();//зв'язок між статтями через користувача
    res.json(posts);//повертаємо усі статті
  } catch (err) {
    console.log(err);
    res.status(500).json({
      message: 'Не вдалося отримати статті',
    });
  }
};
```

Рисунок 3.8 – Функція повернення усіх постів

Було створено функцію для отримання однієї статті, вона знадобиться в подальшому для того, щоб реалізувати функціонал, який працює так: при натисканні на статтю у фронтенд частині, функція бере id статті на яку натиснули та повертає користувачу пост з таким самим id. В подальшому, у фронтенд частині, за повернутим id конкретна стаття буде відкриватись.

Функція працює таким чином: користувач робить запит на конкретну статтю, функція асинхронно намагається знайти id статті на яку був зроблений

запит, після знаходження стаття відкриється – це буде зроблено у фронтенд частині, також лічильник переглядів буде збільшено на 1, таким самим чином, як знаходження статті для перегляду, буде знаходитись стаття при запиті на її редагування. У функції передбачено декілька типів помилок, конструкція «if» або «try catch» поверне помилку, якщо статтю не вдалося повернути через проблеми з сервером, або через відсутність статті, або зовсім не вдалося отримати доступ до всіх статей в базі даних (див. рис. 3.9).

```
export const getOne = async (req, res) => {//отримати одну статтю
  try {
    const postId = req.params.id;//взяли id статті

    PostModel.findOneAndUpdate(//знаходимо статтю та редагуємо, потім повертаємо
      {
        _id: postId,//знаходимо статтю по id
      },
      {
        $inc: { viewsCount: 1 },//коли отримуємо(переглядаємо) статтю - збільшуємо лічільник на 1
      },
      {
        returnDocument: 'after',//повертаємо оновлений результат
      },
      (err, doc) => {//якщо проблеми з сервером - повертаємо помилку
        if (err) {
          console.log(err);
          return res.status(500).json({//статус помилки сервера
            message: 'Не вдалося повернути статтю',
          });
        }
        if (!doc) {//виконується коли повернется undefined
          return res.status(404).json({
            message: 'Стаття не знайдена',
          });
        }
        res.json(doc);
      },
    ).populate('user');
  }
```

Рисунок 3.9 – Отримання однієї статті по id

Наступним кроком є реалізація функціоналу для видалення. Було розроблено функцію, яка знаходить статтю по id та видаляє, після чого

повертає відповідь «true», якщо була помилка, то її буде повернено (див. рис. 3.10).

```
export const remove = async (req, res) => {
  try {
    const postId = req.params.id;

    PostModel.findOneAndDelete(//знаходимо один документ та видаляємо його по
id
      {
        _id: postId,
      },
      (err, doc) => {
        if (err) {
          console.log(err);//вивід помилки якщо є

          return res.status(500).json({
            message: 'Не вдалося видалити статті',
          });
        }

        if (!doc) {//якщо статті нема то повернемо статус 404
          return res.status(404).json({
            message: 'Стаття не знайдена',
          });
        }
        res.json({
          success: true,//повертяємо відповідь якщо стаття видалилась
        });
      },
    );
  }
```

Рис 3.10 – Функція видалення посту

Важливою також є реалізація функції для створення постів, вона асинхронно у методі «try catch» робить запит на створення посту, якщо успішно – то зберігає та повертає створену статтю, або при невдачі поверне статус 500 з текстом помилки (див. рис. 3.11).

```
export const create = async (req, res) => {
  try {
    const doc = new PostModel({//створюємо документ (пост)
      title: req.body.title,//заголовок
      text: req.body.text,//текст статті
      imageUrl: req.body.imageUrl,//картинка
      tags: req.body.tags.split(','),//теги через кому
      user: req.userId,//айді юзера
    });
    const post = await doc.save();// створюємо його

    res.json(post);//поертаємо відповідь (створений пост)
  } catch (err) {
    console.log(err);//або повертаємо помилку зі статусом 500 (помилка сервера)
    res.status(500).json({
      message: 'Не вдалося створити статтю',
    });
  }
```

Рисунок 3.11 – Функція додавання постів

Не менш важливою є функція редагування посту, в ній асинхронно робиться запит на отримання статті по її id, потім оновлюються конкретні поля.

Також необхідним є id користувача, тому що далі буде розроблений функціонал, завдяки якому неавторизований користувач не зможе редагувати свої статті. Також не буде можливості редагувати чужі статті, оскільки буде порівнюватись id користувача та id автора посту, якщо вони співпадають, то користувач є автором посту і зможе його редагувати. Після редагування функція повертає «true», або помилку при проблемах з сервером (див. рис. 3.12).

Далі було розроблено функціонал для роботи з користувачами. Першою є функція реєстрації, імпортовано бібліотеки для хешування та створення токенів. Функція бере пароль, шифрує за алгоритмом, створює модель користувача з поштою, іменем та зашифрованим паролем, також створює і записує в об’єкт користувача токен який буде дійсний 30 днів, відправляє дані у БД після успішної реєстрації. Якщо сталась помилка, то буде повернуто відповідне повідомлення (див. рис. 3.13).

```
export const forPostUpdate = async (req, res) => {//оновлення статті (по id)
  try {
    const postId = req.params.id;//узяли айді з параметрів

    await PostModel.updateOne(//знаходимо  статтю по айді  та оновлюємо її
      {
        _id: postId,
      },
      {//те що можемо оновлювати (редагування статті)
        title: req.body.title,
        text: req.body.text,
        imageUrl: req.body.imageUrl,
        user: req.userId,
        tags: req.body.tags.split(','),//для нормального відображення тегів через кому
      },
    );
    res.json({
      success: true,
    });
  } catch (err) {
    console.log(err);

    res.status(500).json({
      message: 'Не вдалося оновити статтю',
    });}};
```

Рисунок 3.12 – Функція редагування статті

```
export const registrationUsers = async (req, res) => {//регістрація користувача
  try {
    const password = req.body.password;//беремо пароль
    const salt = await bcrypt.genSalt(10);//сіль, алгоритм шифрування, послідовність даних для криптоключа
    const hash = await bcrypt.hash(password, salt);
    const doc = new UserModel({
      email: req.body.email,
      fullName: req.body.fullName,
      avatarUrl: req.body.avatarUrl,
      passwordHash: hash,//хешований пароль
    });

    const user = await doc.save();//підготований документ збережемо в бд
    const token = jwt.sign(//генеруємо токен та передаємо в нього інформацію
      {
        _id: user._id,//зашифрували об`єкт за допомогою ключа
      },
      'secret123',//ключ для шифровки
      {
        expiresIn: '30d',//час життя токену в днях
      },
    );
    const { passwordHash, ...userData } = user._doc;//взяли інформацію про хеш пароля та про користувача
    res.json({
      ...userData,
      token,//повертаємо токен та інформацію об`єкта
    });
  }
};
```

Рисунок 3.13 – Функція реєстрації та шифрування пароля

Було розроблено функцію авторизації, яка є важливою для повноцінного блогу. Функція авторизації шукає в базі даних пароль, розшифровує та порівнює з введеним паролем, логін також порівнюється, якщо подібних в БД немає, то повертає відповідне повідомлення, також при вдалій авторизації користувач отримує новий токен (див. рис. 3.14).

```
export const loginingUsers = async (req, res) => {
  try {
    const user = await UserModel.findOne({ email: req.body.email });//шукаємо емейл в бд
    if (!user) {
      return res.status(404).json({
        message: 'Користувача не знайдено',
      });
    }

    //порівнюємо пароль користувача з паролем з документу , чи сходяться вони
    const isValidPass = await bcrypt.compare(req.body.password, user._doc.passwordHash);

    if (!isValidPass) {//якщо не зійшлись паролі при логінінгу - повертаємо помилку
      return res.status(400).json({
        message: 'невірний логін або пароль',
      });
    }
    const token = jwt.sign(//створюємо токен
      {
        _id: user._id,//зашифрували об`єкт за допомогою ключа
      },
      'secret123',//ключ для шифровки
      {
        expiresIn: '30d',//час життя токена
      },
    );
    const { passwordHash, ...userData } = user._doc;
    res.json({
      ...userData,
      token,
    });
  }
};
```

Рисунок 3.14 – Функція авторизації

Окрім шифрування паролю, його треба і розшифровувати, далі було розроблено для цього функцію, вона бере та розшифровує токен і об’єкт з

даними користувача, дістає з нього id. Якщо розшифрований id співпадає з id користувача, який робить запит, наприклад, на редагування посту, то доступ буде надано і він буде мати можливість змінити дані посту. Також якщо id не співпадають, то користувачу буде відмовлено у доступі (рис. 3.15).

```
import jwt from 'jsonwebtoken';//бібліотека для створеня токенів
export default (req, res, next) => {//next - якщо 1ша ф-я не виповнилась, то виконае другу
  const token = (req.headers.authorization || '').replace(/Bearer\s?/, '');//отримали токен чи ні - прибераємо слово bearer (слово прийшло з insomnia)
  if (token) {/перевірка на наявність
    try {
      const decoded = jwt.verify(token, 'secret123');//розшифровуємо токен по ключу
      req.userId = decoded._id;//передали з розшифровки id юзера, вшили це в реквест користувача
      next();//якщо помилок нема - програма продовжує роботу
    } catch (e) {//якщо токен не змогли розшифрувати видаємо помилку
      return res.status(403).json({
        message: 'Немає доступу',
      });
  }
  } else {//повертаємо помилку
    return res.status(403).json({
      message: 'Немає доступу',    });  }};
```

Рисунок 3.15 – Функція розшифрування токену та порівняння для надання доступу

Далі було розроблено функцію для зупинки виконання запиту при помилці валідації. Якщо помилок нема, то програма продовжить роботу, інакше поверне помилку і подальше виконання запиту буде скасовано (див. рис. 3.16).

```
export default (req, res, next) => {//якщо буде помилка при валідації, то далі запит виконуватись не буде
  const errors = validationResult(req);//запис результата валідації
  if (!errors.isEmpty()) {
    return res.status(400).json(errors.array());//якщо є помилка то повертає   }
  next();//якщо помилок нема - програма продовжує роботу};
```

Рисунок 3.16 – Функція скасування виконування запиту при помилці валідації

Також було розроблено валідатори засобами бібліотеки Express-validations. Функціонал бібліотеки автоматично проводить перевірку на коректність даних, спираючись на критерії вказані розробником. Бібліотека корисна тим, що розробнику не потрібно самому з нуля розробляти логіку валідаторів (рис. 3.17).

```
export const validLogining = [
  body('email', 'Невірний формат пошти').isEmail(),
  body('password', 'Пароль повинен бути мінімум 5 символів').isLength({ min: 5 }),
];

export const validRegistration = [
  body('email', 'Невірний формат пошти').isEmail(),//перевірка чи це емейл, якщо так - пропускає
  body('password', 'Пароль повинен бути мінімум 5 символів').isLength({ min: 5 }),
  body('fullName', 'Вкажіть ім`я').isLength({ min: 3 }),
  body('avatarUrl', 'Невірне посилання на аватарку').optional().isURL(),
];

export const ValidPostsCreating = [//валідатор інформації для створення посту
  body('title', 'Введіть  заголовок статті').isLength({ min: 3 }).isString(),
  body('text', 'Введіть  текст статті').isLength({ min: 3 }).isString(),
  body('tags', 'Невірний  формат тегів').optional().isString(),
  body('imageUrl', 'Невірне посилання на зображення').optional().isString(),
];//зображення для посту опціональне
```

Рисунок 3.17 – Валідація даних при створенні посту, реєстрації або авторизації

Для завантаження файлів, а саме зображень для постів блогу, було імпортовано бібліотеку Multer, вона дає додатковий функціонал для Express, який дозволяє працювати з файлами. Було розроблено функцію завантаження URL-картинки на сервер для подальшого використання, та збереження зображення у папці на диску комп’ютера (див. рис. 3.18).

Після розробки функціоналу бекенду, валідаціїї, завантаження зображень, реєстрації, авторизації, моделі посту та моделі користувача, є необхідною частиною розробка роутів для переходу між сторінками, котрі будуть розроблені у фронтенд частині додатку. Були використані такі запити, як GET, POST, PATCH, DELETE.

```
import multer from 'multer';
const storage = multer.diskStorage({//для сховища зображень
  destination: (_, __, cb) => {//не вказуємо запрос та файл, тільки колбек
    cb(null, 'uploads');  },

  filename: (_, file, cb) => {//file - назва файлу
    cb(null, file.originalname);//беремо оригінальну назву файлу  },});
const upload = multer({ storage });//створили сховище зображень ф-я дозволить використовувати multer

const upload = multer({ storage });//створили сховище зображень ф-я дозволить використовувати multer
app.post('/upload', checkAuth, upload.single('image'), (req, res) => {//очікуємо надходженя картинки, потім викон. ф-ї
  res.json({

    url: `/uploads/${req.file.originalname}`,//у відповіді клієнту - по якому шляху збережено зображеня  });
});
```

Рисунок 3.18 – Завантаження URL-зображення та його збереження засобами бібліотеки Multer

Також створені роути на реєстрацію, авторизацію, перевірку авторизованості користувача і додані до них раніше розроблені валідатори для перевірки коректності введених даних.

Також було створено роути для завантаження на майбутній фронтенд тегів, на отримання усіх постів, роут на отримання конкретної статті.

Також захищені роути з перевіркою авторизації, а саме перевірка на ідентичність id корстувача і автора для надання доступу на видалення статті та на її редагування (див. рис. 3.19).

Таким чином було розроблено бекенд частину додатку, створено базу даних MongoDB яка сама шифрує дані в ній, щоб завадити зловмисникам отримати доступ до особистої інформації користувачів вебдодатку, також були створені роути, функціонал для реєстрації, авторизації, валідація полів вводу, завантаження зображень, додавання посту, видалення посту та його редагування. Цей функціонал є необхідним для повноцінного блогу. У наступному підрозділі буде розроблено фронтенд частину вебдодатку.

```
        app.post('/auth/login', validLogining, validHandlErr, UsrControl.loginingUsers);//валідація та авторизація
        app.post('/auth/register', validRegistration, validHandlErr, UsrControl.registrationUsers);
        app.get('/auth/me', checkingAuthorization, UsrControl.getMe);//перевірка авторизації

        app.post('/upload', checkingAuthorization, upload.single('image'), (req, res) => {//очікуємо надходженя картинки, потім викон. ф-ї
          res.json({
            url: `/uploads/${req.file.originalname}`,//у відповіді клієнту - по якому шляху збережено зображеня
          });
        });

        app.get('/tags', PostsControl.getLastTags);

        app.get('/posts', PostsControl.getAllPosts);//отримати усі статті(масив статей)
        app.get('/posts/tags', PostsControl.getLastTags);//не готово (не встиг)
        app.get('/posts/:id', PostsControl.getOne);//отримати одну статтю
        //захищені роути (додавати статті, редагувати та видаляти можуть тільки авторизовані)
        app.post('/posts', checkingAuthorization, ValidPostsCreating, validHandlErr, PostsControl.createPost);//створити статтю
        app.delete('/posts/:id', checkingAuthorization, PostsControl.remove);//видалити статтю (по id)(видалення працює тільки у авторизованих користувачів)
        app.patch(//оновити статтю
          '/posts/:id',
          checkingAuthorization,
          ValidPostsCreating,
          validHandlErr,
          PostsControl.forPostUpdate,
);
```

Рисунок 3.19 – Роути на сторінки блогу

3.2 Реалізація фронтенд частини додатку

Подійно-орієнтований додаток – це програма, в якій виконання визначається подіями та відповідями на них, а саме діями користувача та відповідями на його дії, частіше за все використовують клавіатуру, мишу або

смартфон для взаємодії з вебдодатками.

У цьому підрозділі буде розроблено фронтенд частину блогу з якою буде взаємодіяти користувач, та необхідний функціонал для цього засобами React, Axios та інших бібліотек, таким чином буде досягнено подійно-орієнтованість. Першим кроком буде створено кореневий елемент для рендерінгу майбутнього додатку, та підключено необхідні бібліотеки, які будуть також використовуватись при подальшій розробці елементів (рис. 3.20).

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(//рендеринг додатку
  <>
    <CssBaseline />
    <ThemeProvider theme={theme}>
      <BrowserRouter>
        <Provider store={store}>
          <App />
        </Provider>
      </BrowserRouter>
    </ThemeProvider>
  </>);
```

Рисунок 3.20 – Підключення бібліотек та створення кореневого елементу для рендерінгу додатку

Після створення корневого елементу необхідно створити сам додаток засобами React.

Також було створено роути на сторінки блогу, а саме реєстрації, головної сторінки, сторінки посту, редагування посту та авторизації, при переході на конкретний роут буде відкриватись відповідний елемент (див. рис. 3.21).

Далі було використано бібліотеку Axios для полегшення роботи з роутами між сторінками блогу. Було вказано базовий шлях до елементів і додавання токену до запитів, для того щоб керувати можливостями користувача, наприклад забороняти редагувати чужі пости (див. рис. 3.22).

```
function App() {//
  const dispatch = useDispatch();//метод дозволяє відправити дію диспетчеру та змінити стан програми.
  const isAuth = useSelector(selectIsAuth);//перевірка на авторизованість
  React.useEffect(() => {//хук, для заміни деяких методів життєвого циклу компонента.
    dispatch(fetchAuthMe());  }, []);
  return (    <>
    <Header />
    <Container maxWidth="lg">
     <Routes>
      <Route path="/" element={<Home/>} />
      <Route path="/posts/:id" element={<FullPost/>} />
      <Route path="/posts/:id/edit" element={<AddPost/>} />
      <Route path="/add-post" element={<AddPost/>} />
      <Route path="/login" element={<Login/>} />
      <Route path="/register" element={<Registration/>} />
     </Routes>
    </Container>
   </>
  );
}
export default App;
```

Рисунок 3.21 – Додаток створений засобами React з роутами для сторінок

```
import axios from "axios";
const instance = axios.create({
   baseURL: 'http://localhost:4444',//щоб не дописувати постійно самому базовий шлях
});
instance.interceptors.request.use((config) => {//при будь якому запиту - перевіряє наявність токену, якщо нема - додає його
   config.headers.Authorization = window.localStorage.getItem('token');
   return config; });
export default instance;
```

Рисунок 3.22 – Використання бібліотеки Axios для полегшення роботи з роутами

Наступним кроком є створення головної сторінки, створення розмітки на jsx, додавання стилів для неї засобами SCSS і функціоналу для отримання постів і зображень для досягнення подійної орієнтованості проєкту. Розмітку і SCSS стилі наступних сторінок буде наведено в додатку Б.

Отримуватися пости, зображення та стан авторизованості користувача будуть з бази даних, щоб в залежності від авторизованості в хедер додавати кнопку створення посту, а якщо користувач не авторизований – то кнопку

реєстрації і входу (рис. 3.23).

```
export const Home = () => {//головна сторінка та її функціонал
  const dispatch = useDispatch();//метод дозволяє відправити дію диспетчеру та змінити стан програми.
  const userData = useSelector((state) => state.auth.data); //вилучатє дані про авторизацію зі стану сховища
  const { posts, tags} = useSelector((state) => state.posts);//стан постів
  const isPostsLoading = posts.status === 'loading';//перевірка на статус завантаження тегів і посту
  const isTagsLoading = tags.status === 'loading';
  React.useEffect(()=> {
    dispatch(fetchPosts());
    dispatch(fetchTags());
  }, []);
  Return(
    <>
      <Tabs style={{ marginBottom: 15 }} value={0} aria-label="basic tabs example">
        <Tab label="Останні статті" />
        {/* <Tab label="Популярні" /> */}
      </Tabs>
      <Grid container spacing={4}>
        <Grid xs={8} item>

          {(isPostsLoading ? [...Array(5)] : posts.items).map((obj, index) => //підгрузка усіх статей з бд
           isPostsLoading ? (
           <Post key={index} isLoading={true}/>
           ) : (
            <Post
              id={obj._id}
              title={obj.title}
              imageUrl={obj.imageUrl ? `http://localhost:4444${obj.imageUrl}` : ''}//підгрузка зображення, якщо є, інаше вивід без нього
              user={obj.user}
              createdAt={obj.createdAt}

              viewsCount={obj.viewsCount}

              tags={obj.tags}
              isEditable={userData?._id === obj.user._id}
            />
           ),
           )}
        </Grid>
        <Grid xs={4} item>
          <TagsBlock items={tags.items} isLoading={isTagsLoading} />

        </Grid>
      </Grid>
    </>
  );
};
```

Рисунок 3.23 – Функціонал та jsx розмітка головної сторінки блогу

Також було розроблено сторінку відкритого посту засобами jsx. Функція підвантажує дані постів з бази даних, шукає id пост на який було натиснуто та по id повертає його, якщо такий в пост є в наявності (рис. 3.24).

```
export const FullPost = () => {
  const [data, setData] = React.useState();
  const [isLoading, setLoading] = React.useState(true);
  const { id } = useParams();

 React.useEffect(()=> {
  axios
  .get(`/posts/${id}`)//.get(`http://localhost:4444/posts/${id}`)//такий варіант працює, підгрузка посту

  .then((res) => {
   setData(res.data);

   setLoading(false); })
  .catch((err) => {//обробник помилок
   console.warn(err);
   alert('Помилка при отриманні статті.')  });}, []);
 if (isLoading) {
  return <Post isLoading={isLoading} isFullPost />;}
 }
```

Рисунок 3.24 – Розмітка сторінки відкритої статті та функціонал для знаходженні статті в базі даних

Далі було створено сторінку реєстрації засобами jsx та SCSS і функціонал для неї.

Найважливіші функції тут це реєстрація та перевірка коректності введених даних засобами валідації, якщо було введено дані існуючого акаунту, або некоректні дані, то буде повернуто повідомлення «Не вдалось зарєєструватись».

Якщо дані введено вірно, то після натискання кнопки реєстрації пароль буде зашифровано бекендом, а після цього усі дані відправляться в базу даних (див. рис. 3.25).

```
export const Registration = () => {//реєстрація, компонент та функціонал
  const isAuth = useSelector(selectIsAuthorized)
  const dispatch = useDispatch();
  const {
    register,
    handleSubmit,
    formState: { errors, isValid }
  } = useForm({//форма реєстрації
    defaultValues: {

      fullName: 'Вольфганг Амадей',//для більш швидкої реєстрації та її тестування
      email: 'wolf@mail.ua',//щоб зайвий раз не вводити дані самому
      password: '11223',
    },
    mode:'onChange',
  });
  const onSubmit = async (values) => {//відпр форм
    const data = await dispatch(fetchRegister(values));

  if (!data.payload) {
    return alert('не вдалось зареєструватись!');
  }

    if ('token' in data.payload) {//додавання токена
      window.localStorage.setItem('token', data.payload.token);
    }
  };

  if (isAuth) {
    return <Navigate to="/" />//на головну якщо нема помилок
}
```

Рисунок 3.25 – Функціонал для реєстрації і розмітка відповідної сторінки

Аналогічним способом було створено сторінку авторизації засобами jsx розмітки та SCSS стилів.

Також було розроблено функціонал для перевірки коректності введених даних, і після авторизації переведення користувача на головну сторінку (див. рис. 3.26).

```
export const Login = () => {//функціонал для авторизації

  const isAuth = useSelector( selectIsAuthorized );//перевірка на авторизованість
  const dispatch = useDispatch();
  const {
    register,
    handleSubmit,

    formState: { errors, isValid } //
  } = useForm({
    defaultValues: {
      email: 'vvvasya123@mail.ua',
      password: '12345',
    },
    mode:'onChange',
  });

 const onSubmit = async (values) => {//відправка форми авторизування
  const data = await dispatch(fetchAuth(values));

 if (!data.payload) {
  return alert('не вдалось авторизуватись!');
 }

  if ('token' in data.payload) {
    window.localStorage.setItem('token', data.payload.token);//якщо не буде токену, то його додасть
  }
 };

 if (isAuth) {//перехід на головну після авторизації
  return <Navigate to="/" />
}
```

Рисунок 3.26 – Розмітка сторінки авторизації та розроблений відповідний функціонал

Далі було розроблено розмітку сторінки додавання посту засобами jsx та задання йому стилів SCSS. Функція додавання посту бере дані, введені в поля для тексту, картинку, теги, та зберігає їх разом з id у базу даних після натискання кнопки «зберегти». Також було розроблено функцію редагування статті: якщо користувач авторизований і є автором статті, що перевіряється по id, то після натискання кнопки для редагування бекенд робить запит на сторінку, потім користувача переводить на відповідну сторінку по id посту. Збереження оновленої статті таке як і звичайної створеної статті – засобом кнопки «зберегти». Таким чином в застосунку було реалізовано додатковий функціонал подійно-орієнтованості додатку (рис. 3.27).

```
export const AddPost = () => {//ф-я для додавання посту
  const {id} = useParams();// передали id посту
  const navigate = useNavigate();//hook
  const isAuth = useSelector(selectIsAuthorized)//
  const [setLoading] = React.useState(false);//завантаження посту на сервер
  const [text, setText] = React.useState('');//value - зберігає введене в редакторі
  const [title, setTitle] = React.useState('');
  const [tags, setTags] = React.useState('');
  const [imageUrl, setImageUrl] = React.useState('');
  const inputFileRef = React.useRef(null);//hook для отримання зображення
  const isEdting = Boolean(id);//для розуміння чи ми на сторінці редагування
  const handleChangeFile = async (event) => {
   try {
    const formData = new FormData();
    const file = event.target.files[0];
    formData.append('image', file);
    const { data } = await axios.post('/upload', formData);//запит на завантаження файла при створенні
посту
    setImageUrl(data.url);
   } catch (err) {
    console.warn(err);
    alert('помилка при завантаженні файла! ');
   }
  };
  const onClickRemoveImage = () => {
   setImageUrl('');//видалення картинки
  };
  const onChange = React.useCallback((value) => {//отримує value (введений текст)
   setText(value);
  }, []);
  const onSubmit = async () => { //відправка статті на бекенд
   try {
    setLoading(true);
    const fields = {//передаємо об'єкт з полями на сервер
     title,
     imageUrl,
     tags,//бекенд сам робить split - масив тегів через кому
     text,     };
    const { data } = isEditing //якщо редагуємо то
    ? await axios.patch(`/posts/${id}`, fields)//такий запит (patch - запит на апдейт посту)
    : await axios.post('/posts', fields);//або інший
    const _id = isEditing ? id : data._id;
    navigate(`/posts/${_id}`);//напривить на конкретний пост по id
   } catch (err) {
    console.warn(err);
    alert('Помилка при створенні статті !');
   }  };
  React.useEffect(() => {//перевірка на id посту, для редагування
   if (id) {//якщо id є, то отримуємо відповідь по роуту - пост, для подальшого редагування
    axios
     .get(`/posts/${id}`)
     .then(({data}) => {
      setTitle(data.title);
      setText(data.text);
      setImageUrl(data.imageUrl);
      setTags(data.tags.join(','));
     }).catch(err => {
      console.warn(err);
      alert('Помилка при отриманні статті!')
     });
   }
 }, []);
```

Рисунок 3.27 – Розмітка та функціонал сторінки додавання та редагування посту

Далі було розроблено пост блогу засобами jsx та SCSS, сам пост буде відображатися на головній сторінці.

Створені та збережені у базі даних пости будуть виводитись на головний екран у вигляді переліку.

Також було додано функціонал для видалення посту, після натискання відповідної кнопки автором (рис. 3.28).

```
export const Post = ({id, title, createdAt, imageUrl,  user, viewsCount,  tags, isFullPost,
isLoading,  isEditable, }) => {  const dispatch = useDispatch();
  if (isLoading) {
    return <PostSkeleton />; }
  const onClickRemove = () => {
    if (window.confirm('Ви впевнені, що бажаєте видалити статтю ?')) {
      dispatch(fetchRemovePost(id));    }  };
```

Рисунок 3.28 – Пост на головній сторінці та функціонал його видалення

Після розробки елементів та сторінок було створено головну сторінку, на якій буде відображатись перелік постів з бази даних, та останні теги статей, реалізацію тегів наведено в додатку Б.

Сторінка хедеру окрім розмітки на jsx та SCSS має функцію виходу з акаунту, якщо користувач авторизований, і навпаки – авторізації або реєстрації (див. рис. 3.29).

Зовнішній вигляд, а саме кнопки на хедері змінюються в залежності від авторизованості користувача, код перевірки на авторизованість наведено в додатку Б.

Далі наведено основні розроблені сторінки блогу, такі як головна, сторінка реєстрації, сторінка відкритого посту та його створення.

На головній сторінці відображаються створені користувачами пости, їх кількість переглядів, п’ять тегів останніх створених статей, кнопка додавання статті і виходу з акаунту, якщо користувач авторизований, інакше в хедері буде відображено кнопки входу в акаунт або реєстрації.

```
export const Header = () => {

  const dispatch = useDispatch();

  const isAuth = useSelector( selectIsAuthorized );//змінюємо хедер в залежності від авторизованості користувача

  const onClickLogout = () => {//вихід з акаунту

    if (window.confirm('Ви впевнені, що бажаєте вийти ?')) {
      dispatch(logout());

      window.localStorage.removeItem('token');
//після виходу з акаунту прибераємо токен з локалстореджа

    }

  }; return (
    <div className={styles.root}>
      <Container maxWidth="lg">

        <div className={styles.inner}>

          <Link className={styles.logo} to="/">
            <div>My_Diplom_Blog</div>
</Link>
          <div className={styles.buttons}>
            {isAuth ? (

              <>
              <Link to="/add-post">
                <Button variant="contained">Написати статтю</Button>
              </Link>

              <Button onClick={onClickLogout} variant="contained" color="error">
                Вийти
              </Button>
 </>
) : (
<>
              <Link to="/login">
                <Button variant="contained">Увійти</Button>
              </Link>

              <Link to="/register">
                <Button variant="contained">Створити аккаунт</Button>
              </Link> </>  )}

          </div>
        </div>
      </Container>
    </div>  ); };
```

Рисунок 3.29 – Головна сторінка блогу та функціонал зміни вигляду хедеру в залежності від авторизованості

При наведенні курсору на пост який створений його автором – будуть відображатися кнопки видалення та редагування (рис 3.30).

Неавторизовані користувачі також мають можливість переглядати статті, але не можуть редагувати, додавати та видаляти пости.

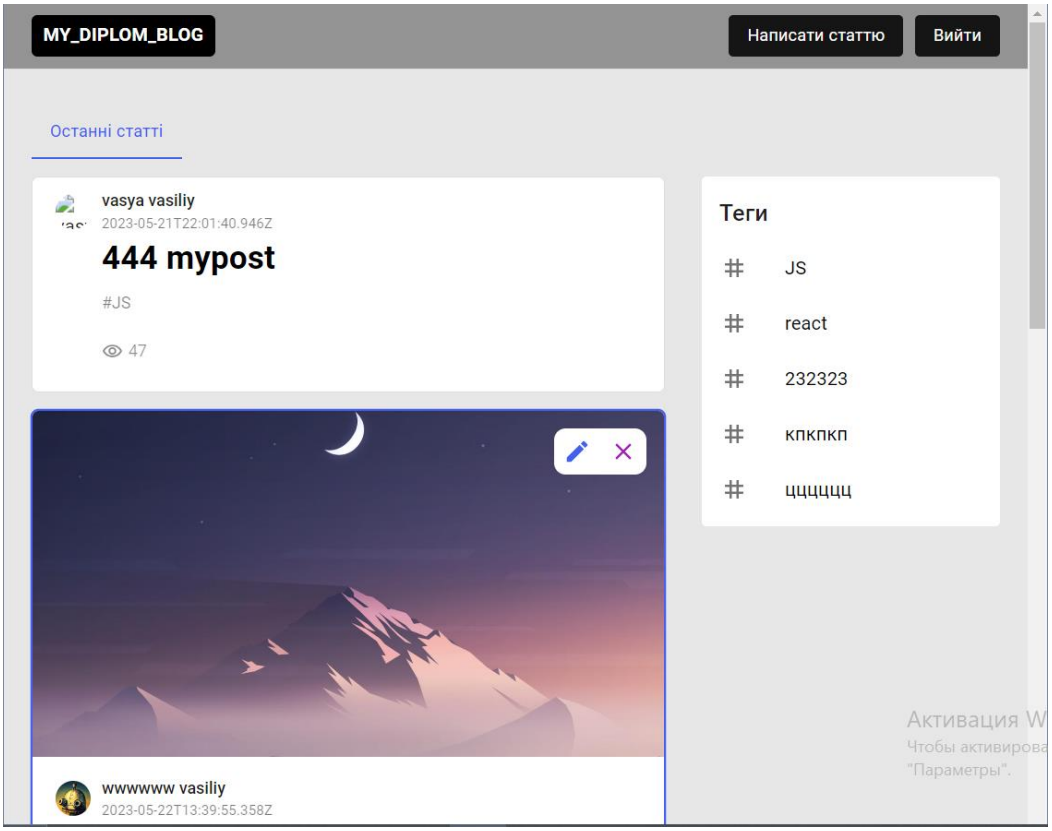

Рисунок 3.30 – Головна сторінка блогу

Однією з основних сторінок з важливим функціоналом є сторінка реєстрації. На ній користувач буде вводити свої дані, якщо вони коректні, пройдуть валідацію, та не буде аналогічної зареєстрованої пошти, то користувач зареєструється після натискання відповідної кнопки, його пароль зашифрує бекенд та дані будуть відправлені в БД, там вони будуть зберігатися та використовуватися для авторизації і інших потреб вебзастосунку (див. рис. 3.31).

Іншою частиною функціоналу додатку є можливість додавання та редагування постів. Якщо користувач авторизований і є автором посту, то він буде мати можливість редагувати свій пост натиснувши відповідну кнопку на

головній сторінці. При редагуванні або створенні посту є можливість додавати картинку, видаляти її, редагувати теги, змінювати основний текст та заголовок (див. рис. 3.32).

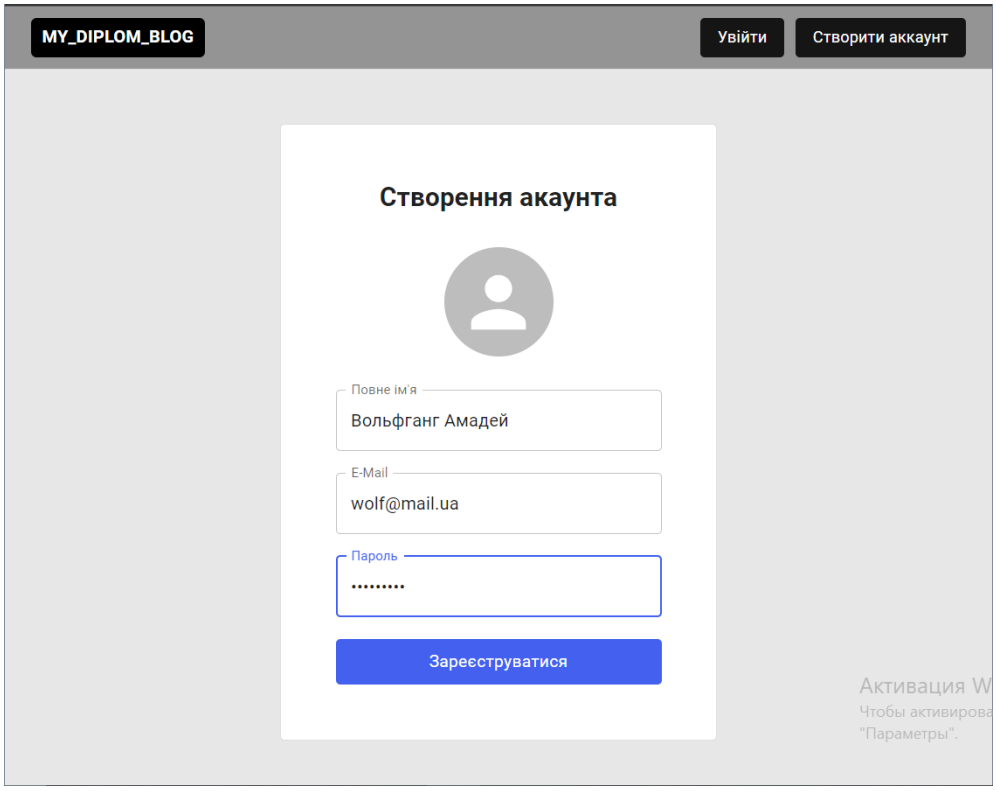

Рисунок 3.31 – Сторінка реєстрації

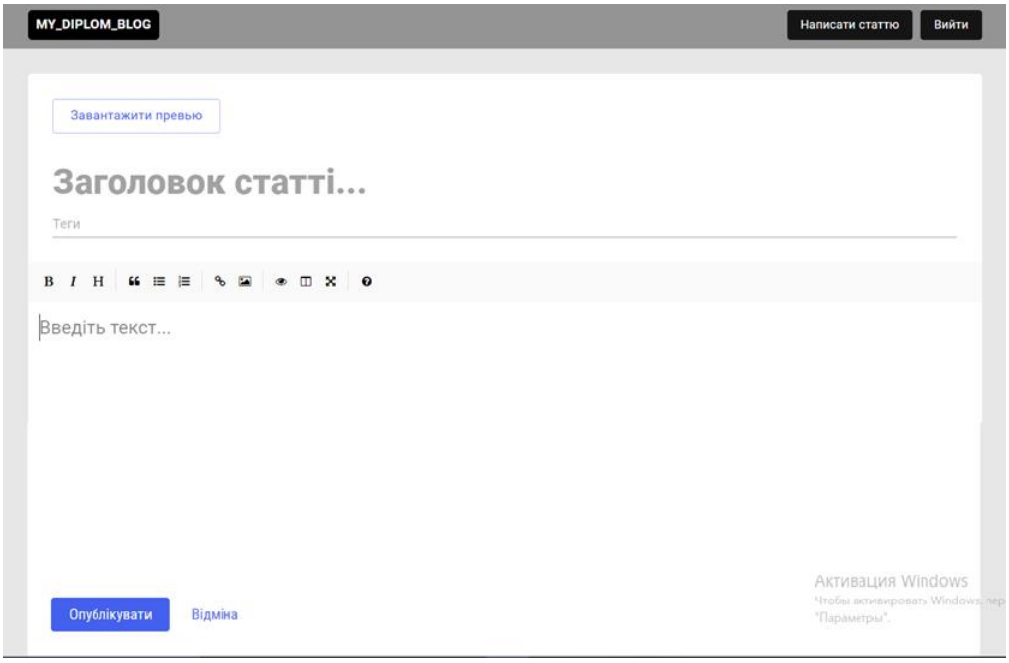

Рисунок 3.32 – Сторінка створення та редагування посту

В результаті створення статті користувачі будуть мати можливість її переглядати, при кожному відкритті лічільник переглядів буде збільшуватися на один, так усі будуть бачити скільки разів пост було переглянуто (див. рис. 3.33).

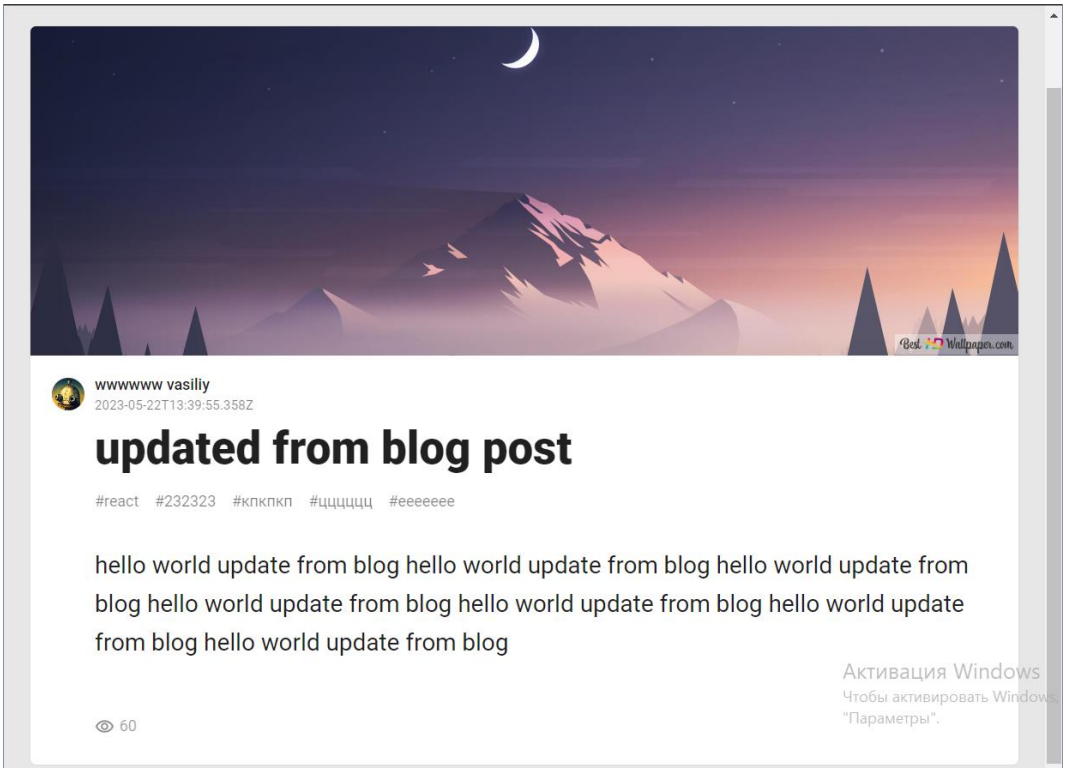

Рисунок 3.33 – Сторінка перегляду посту

Таким чином у підрозділі 3.2 було продемонстровано процес розробки фронтенд частини проекту.

Було реалізовано сторінки реєстрації, головну з її елементами, сторінку авторизації, створення посту та редагування, також розроблено навігацію між сторінками.

Таким чином, відповідно до визначення подійно-орієнтованого додатку, усі розроблені функції та інтерфейс для взаємодії користувача з додатком роблять вебзастосунок подійно-орієнтованим.

3.3 Висновки до розділу 3

У розділі 3 було створено базу даних засобами MongoDB, розроблено бекенд частину додатку засобами Node.js і Express, створено роути, запити на сторінки та функціонал для реєстрації, авторизації, валідації полів вводу, шифрування паролів, також запити на завантаження зображень в базу даних, на додавання посту, видалення посту та на його редагування автором. Такий функціонал є необхідною частиною повноцінного блогу. Таким самим чином була розроблена на React фронтенд частина проекту. Було реалізовано сторінки для реєстрації, головна з її елементами постів і хедером, сторінка авторизації, створення посту та редагування. Також зв’язано фронтенд з бекендом, розроблено навігацію між сторінками і функціонал для взаємодії користувача з додатком для досягнення подійно-орієнтованості.

Також додаток, а саме весь його функціонал, було ретельно протестовано засобами ручного тестування.

ВИСНОВКИ

Отже, в цій роботі було проведено аналіз інструментів і технологій, спроектовано та розроблено блог, що складається з фронтенд і бекенд частин та бази даних для зберігання інформації про пости і користувачів. Після розробки було протестовано додаток засобами ручного тестування.

У розділі 1 було проведено аналіз технологій для розробки вебдодатків, було наведено визначення подійно-орієнтованого додатку, було обрано та описано засоби React, Node.js, а також інструменти, як SASS, Mongodb та Express. Також у першому розділі було описано вимоги до проекту, його функціонал і можливості користувача.

У розділі 2 було спроектовано блог за допомогою UML-діаграми активності, структурної схеми для демонстрації наповнення та функціоналу, також створено схему роботи реєстрації та авторизації користувача, було розроблено схему NoSQL бази даних блогу, реалізовано діаграму компонентів. Було розроблено скетчі дизайну сторінок. Таким чином розроблені скетчі, діаграми та структурні схеми демонструють, як будуть реалізовані вимоги до проекту.

У розділі 3 було створено базу даних засобами MongoDB, розроблено бекенд додатку засобами Node.js і Express, створено роути, запити та функціонал для реєстрації, авторизації, валідації полів вводу, також запити на завантаження зображень в базу даних, на додавання посту, видалення посту та на його редагування. Також була розроблена фронтенд частина проекту на React, Scss і Axios. Було створено сторінки для реєстрації, головна з елементами постів і хедером, сторінка авторизації, створення посту та редагування. Розроблено навігацію між сторінками, функціонал для взаємодії користувача з вебзастосунком і було зв’язано фронтенд з бекендом.

Таким чином у роботі було розроблено подійно-орієнтований вебзастосунок засобами React та Node.js.

ПЕРЕЛІК ДЖЕРЕЛ

1. Banks A., Porcello E. Learning React Modern Patterns for Developing React Apps. Sebastopol : «O'Reilly Media», 2020. 294 p.

2. Stefanov S., React: Up & Running. Sebastopol : «O'Reilly Media», 2022. 213 p.

3. Flanagan D., JavaScript: The Definitive Guide. Sebastopol : «O'Reilly Media», 2011. 1078 p.

4. Resig J., Bibeault B., Secrets of the JavaScript Ninja. Shelter Island : «Manning Publications Co», 2013. 369 p.

5. Chodorow K., Mongodb: The Definitive Guide. Sebastopol : «O'Reilly Media», 2013. 409 p.

6. Powers S., Learning Node. Sebastopol : «O'Reilly Media», 2012. 374 p.

7. Brown E., Web Development with Node and Express. Sebastopol : «O'Reilly Media», 2014. 306 p.

8. Documentation. Sass-lang. URL: https://sass-lang.com/documentation/ (дата звернення: 08.05.2023).

9. Guide. Expressjs. URL: https://expressjs.com/en/guide/routing.html (дата звернення 03.04.2023).

10. Документація. Nodejs. URL: https://nodejs.org/uk/docs (дата звернення 01.04.2023).

11. Guide. Jsonwebtoken. URL: http://surl.li/hzerc (дата звернення 03.04.2023).

ДОДАТОК А

Код бекенд частини

```
{
  "name": "blog-mern",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "start": "node index.js",
    "start:dev": "nodemon index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.0.1",
    "cors": "^2.8.5",
    "express": "^4.18.1",
    "express-validator": "^6.14.1",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^6.3.5",
    "multer": "^1.4.5-lts.1",
    "nodemon": "^2.0.16"
  }
}
import express from 'express';

import multer from 'multer';//бібліотека для завантаження файлів(зображень)

import cors from 'cors';

import mongoose from 'mongoose';

import { validRegistration, validLogining, ValidPostsCreating } from './validations.js';//імпорт валідації авторизації/регістпації

import { validHandlErr, checkingAuthorization } from './utils/index.js';

import { UsrControl, PostsControl } from './controllers/index.js';

mongoose        //робота з монгоДБ

  .connect("mongodb+srv://ilyasoooper:littlelesssomepass@cluster0.b94imhr.mongodb.net/blog?retryWrites=true&w=majority")//посилання на мою бд

  .then(() => console.log('БД в порядку'))//мб в ссилке пароль от ака монго(littlelesssomepass) а не wwwwww

  .catch((err) => console.log('Помилка БД', err)); //зараз бд працює (прошлий урл process.env.MONGODB_URI)

const app = express();//створюємо експресс додаток

const storage = multer.diskStorage({//для сховища зображень

  destination: (_, __, cb) => {//не вказуємо запрос та файл, тільки колбек

    cb(null, 'uploads');

  },
```

```
  filename: (_, file, cb) => {//file - назва файлу
    cb(null, file.originalname);//беремо оригінальну назву файлу
  },
});
const upload = multer({ storage });//створили сховище зображень, ф-я дозволить використовувати multer
app.use(express.json());//використовуємо в додатку json для його читання
app.use(cors());//щоб не було помилок з переходами по різним доменам
app.use('/uploads', express.static('uploads'));//при переході на цей роут - експрес перевіряє наявність файлу в папці uploads
//роути валідація повертаємо помилки якщо нема помилок виконуемо логін і тд
app.post('/auth/login', validLogining, validHandlErr, UsrControl.loginingUsers);//валідація та авторизація
app.post('/auth/register', validRegistration, validHandlErr, UsrControl.registrationUsers);
app.get('/auth/me', checkingAuthorization, UsrControl.getMe);//перевірка авторизації
app.post('/upload', checkingAuthorization, upload.single('image'), (req, res) => {//очікуємо надходженя картинки, потім викон. ф-ї
  res.json({
    url: `/uploads/${req.file.originalname}`,//у відповіді клієнту - по якому шляху збережено зображеня
  });
});
app.get('/tags', PostsControl.getLastTags);
app.get('/posts', PostsControl.getAllPosts);//отримати усі статті(масив статей)
app.get('/posts/tags', PostsControl.getLastTags);//не готово (не встиг)
app.get('/posts/:id', PostsControl.getOne);//отримати одну статтю
//захищені роути (додавати статті, редагувати та видаляти можуть тільки авторизовані)
app.post('/posts', checkingAuthorization, ValidPostsCreating, validHandlErr, PostsControl.createPost);//створити статтю
app.delete('/posts/:id', checkingAuthorization, PostsControl.remove);//видалити статтю (по id)(видалення працює тільки у авторизованих користувачів)
app.patch(//оновити статтю
  '/posts/:id',
  checkingAuthorization,
  ValidPostsCreating,
  validHandlErr,
  PostsControl.forPostUpdate,
);
app.listen(process.env.PORT || 4444, (err) => {//прикріпили додаток на порт
  if (err) {
    return console.log(err);//якщо сервер не запустився - виводимо помилку
```

```
  }
  console.log('my server is OK');//якщо запустився - виводимо повідомлення
});
export { default as checkingAuthorization } from './checkAuth.js';//помістили все з файлу в змінну та її експортували для подальшого використання
export { default as validHandlErr } from './handleValidationErrors.js';
import { validationResult } from 'express-validator';
export default (req, res, next) => {//якщо буде помилка при валідації, то далі запрос виконуватись не буде
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json(errors.array());//якщо є помилка то повертяє
  }
  next();//якщо помилок нема - програма продовжує роботу
};
import jwt from 'jsonwebtoken';//бібліотека для створеня токенів
//авторизація
export default (req, res, next) => {//next - якщо 1ша ф-я не виповнилась, то виконае другу
  const token = (req.headers.authorization || '').replace(/Bearer\s?/, '');//отримали токен чи ні - прибераємо слово bearer (слово прийшло з insomnia)
  if (token) {
    try {
      const decoded = jwt.verify(token, 'secret123');//розшифровуємо токен по ключу
      req.userId = decoded._id;//передали з розшифровки id юзера, вшили це в реквест користувача
      next();//якщо помилок нема - програма продовжує роботу
    } catch (e) {//якщо токен не змогли розшифрувати видаємо помилку
      return res.status(403).json({
        message: 'Немає доступу',
      });
    }
  } else {
    return res.status(403).json({
      message: 'Немає доступу',
    });
  }
};

const PostSchema = new mongoose.Schema(
```

```
  {
    title: {
      type: String,
      required: true,
    },
    text: {
      type: String,
      required: true,
      unique: true,
    },
    tags: {
      type: Array,
      default: [],
    },
    viewsCount: {
      type: Number,
      default: 0,
    },
    user: {//ObjectId буде ссилатися на ref: 'User' (конкретного користувача)
      type: mongoose.Schema.Types.ObjectId,//спеціальний тип з бд (id)
      ref: 'User',//створили relationshep - зв'язок між двома таблицями (юзер та пост)
      required: true,
    },
    imageUrl: String,
  },
  {
    timestamps: true,
  },
);
export default mongoose.model('Post', PostSchema);
const UserSchema = new mongoose.Schema(
  {
    fullName: {
      type: String,
      required: true,
    },
    email: {
```

```
      type: String,
      required: true,
      unique: true,
    },
    passwordHash: {
      type: String,
      required: true,
    },
    avatarUrl: String,
  },
  {
    timestamps: true,
  },
);
export default mongoose.model('User', UserSchema);
import jwt from 'jsonwebtoken';//бібліотека для створення токенів
import bcrypt from 'bcrypt';//для шифрування(хешування)

import UserModel from '../models/User.js';
export const registrationUsers = async (req, res) => {//регістрація користувача
  try {
    const password = req.body.password;//беремо пароль
    const salt = await bcrypt.genSalt(10);//сіль, алгоритм шифрування, послідовність даних для криптоключа
    const hash = await bcrypt.hash(password, salt);
    const doc = new UserModel({
      email: req.body.email,
      fullName: req.body.fullName,
      passwordHash: hash,
    });
    const user = await doc.save();//підготований документ збережемо в бд
    const token = jwt.sign(//генеруємо токен та передаємо в нього інформацію
      {
        _id: user._id,//зашифрували об`єкт за допомогою ключа
      },
      'secret123',//ключ для шифровки
      {
```

```
        expiresIn: '30d',//час життя токену в днях
      },
    );
    const { passwordHash, ...userData } = user._doc;//взяли інформацію про хеш пароля та про користувача
    res.json({
      ...userData,
      token,//повертаємо токен та інформацію об`єкта
    });
  } catch (err) {
    console.log(err);
    res.status(500).json({
      message: 'Не вдалося зареєструватися',
    });
  }
};
export const loginingUsers = async (req, res) => {
  try {
    const user = await UserModel.findOne({ email: req.body.email });//шукаємо емейл в бд
    if (!user) {
      return res.status(404).json({
        message: 'Користувача не знайдено',
      });
    }
    //порівнюємо пароль користувача з паролем з документу , чи сходяться вони
    const isValidPass = await bcrypt.compare(req.body.password, user._doc.passwordHash);
    if (!isValidPass) {//якщо не зійшлись паролі при логінінгу - повертаємо помилку
      return res.status(400).json({
        message: 'невірний логін або пароль',
      });
    }
    const token = jwt.sign(//створюємо токен
      {
        _id: user._id,//зашифрували об`єкт за допомогою ключа
      },
      'secret123',//ключ для шифровки
      {
```

```
      expiresIn: '30d',//час життя токена
    },
   );
   const { passwordHash, ...userData } = user._doc;
   res.json({
    ...userData,
    token,
   });
  } catch (err) {
   console.log(err);
   res.status(500).json({
    message: 'Не вдалося авторизуватися',
   });
  }
};
export const getMe = async (req, res) => {
  try {
   const user = await UserModel.findById(req.userId);//знаходимо користувача в базі даних по id
   if (!user) {
    return res.status(404).json({
     message: 'Користувача не знайдено',
    });
   }
   const { passwordHash, ...userData } = user._doc;
   res.json(userData);
  } catch (err) {
   console.log(err);
   res.status(500).json({
    message: 'Немає доступу',
   });
  }
};
import PostModel from '../models/Post.js';//передали модель посту

export const getLastTags = async (req, res) => {
  try {
   const posts = await PostModel.find().limit(5).exec();
```

```
    const tags = posts
      .map((obj) => obj.tags)//перебрали
      .flat()//витягнули статті
      .slice(0, 5);//узяли теги  з 5ти останніх статей
    res.json(tags);
  } catch (err) {
    console.log(err);
    res.status(500).json({
      message: 'Не вдалось отримати теги',
    });
  }
};
export const getAllPosts = async (req, res) => {//отримати усі статті
  try {
    const posts = await PostModel.find().populate('user').exec();//зв'язок між статтями через користувача
    res.json(posts);//повертаємо усі статті
  } catch (err) {
    console.log(err);
    res.status(500).json({
      message: 'Не вдалося отримати статті',
    });
  }
}
export const getOne = async (req, res) => {//отримати одну статтю
  try {
    const postId = req.params.id;//взяли id статті

    PostModel.findOneAndUpdate(
      {
        _id: postId,//знаходимо статтю по id
      },
      {
        $inc: { viewsCount: 1 },//коли отримуємо статтю - збільшуємо лічільник на 1
      },
      {
        returnDocument: 'after',//повертаємо оновлений результат
      },
```

```
      (err, doc) => {//якщо проблеми з сервером - повертаємо помилку
        if (err) {
          console.log(err);
          return res.status(500).json({
            message: 'Не вдалося повернути статтю',
          });
        }
        if (!doc) {//виконується коли повернется undefined
          return res.status(404).json({
            message: 'Стаття не знайдена',
          });
        }
        res.json(doc);
      },
    ).populate('user');
  } catch (err) {
    console.log(err);
    res.status(500).json({
      message: 'Не вдалося отримати статті',
    });
  }
};
export const remove = async (req, res) => {
  try {
    const postId = req.params.id;
    PostModel.findOneAndDelete(//знаходимо один документ та видаляємо його по id
      {
        _id: postId,    },
      (err, doc) => {
        if (err) {
          console.log(err);
          return res.status(500).json({
            message: 'Не вдалося видалити статті',
          });
        if (!doc) {
          return res.status(404).json({
            message: 'Стаття не знайдена',
```

```
        });
      }
      res.json({
        success: true,//повертяємо відповідь якщо стаття видалилась
      });
    },
  );
 } catch (err) {
  console.log(err);
  res.status(500).json({
    message: 'Не вдалося отримати статті',
  });
 }
};
export const createPost = async (req, res) => {
 try {
  const doc = new PostModel({//створюємо документ (пост)
    title: req.body.title,
    text: req.body.text,
    imageUrl: req.body.imageUrl,
    tags: req.body.tags.split(','),
    user: req.userId,
  });
  const post = await doc.save();//після створення документу - створюємо його
  res.json(post);//поертаємо відповідь (створений пост)
 } catch (err) {
  console.log(err);
  res.status(500).json({
    message: 'Не вдалося створити статтю',
  });
 }
};
export const forPostUpdate = async (req, res) => {//оновлення статті (по id)
 try {
  const postId = req.params.id;
  await PostModel.updateOne(//знаходимо  статтю та оновлюємо її
    {
```

```
        _id: postId,
      },
      {//те що можемо оновлювати (редагування статті)
        title: req.body.title,
        text: req.body.text,
        imageUrl: req.body.imageUrl,
        user: req.userId,
        tags: req.body.tags.split(','),//для нормального відображення тегів
      },
    );
    res.json({
      success: true,
    });
  } catch (err) {
    console.log(err);
    res.status(500).json({
      message: 'Не вдалося оновити статтю',
    });
  }
};
export * as UsrControl from './UserController.js';//беремо все з файла в змінну та експортуємо її
export * as PostsControl from './PostController.js';
```

ДОДАТОК Б

Код фронтенду

```
{
  "name": "react-blog",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@emotion/react": "^11.9.0",
    "@emotion/styled": "^11.8.1",
    "@mui/icons-material": "^5.8.0",
    "@mui/material": "^5.8.0",
    "@reduxjs/toolkit": "^1.8.2",
    "@testing-library/jest-dom": "^5.16.4",
    "@testing-library/react": "^13.2.0",
    "@testing-library/user-event": "^13.5.0",
    "axios": "^0.27.2",
    "clsx": "^1.1.1",
    "easymde": "^2.16.1",
    "prettier": "^2.6.2",
    "react": "^18.1.0",
    "react-dom": "^18.1.0",
    "react-hook-form": "^7.32.0",
    "react-markdown": "^8.0.3",
    "react-redux": "^8.0.2",
    "react-router-dom": "^6.3.0",
    "react-scripts": "5.0.1",
    "react-simplemde-editor": "^5.0.2",
    "sass": "^1.52.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

```
import { createTheme } from "@mui/material/styles"
export const theme = createTheme({
  shadows: ["none"],
  palette: {
    primary: {
      main: "#4361ee",
    },
  },
  typography: {
    button: {
      textTransform: "none",
      fontWeight: 400,
    },
  },
});
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Roboto';
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  background-color: #e7e7e7 !important;
}

import React from "react";
import { Provider } from "react-redux";
import { BrowserRouter } from 'react-router-dom';//для роутінга додатку
import ReactDOM from "react-dom/client";
import App from "./App";
import CssBaseline from "@mui/material/CssBaseline";

import "./index.scss";//стилі
import { ThemeProvider } from "@mui/material";
import { theme } from "./theme";
import store from "./redux/store";

const root = ReactDOM.createRoot(document.getElementById("root"));
//рендер додатку
```

```
root.render(
  <>
    <CssBaseline />
    <ThemeProvider theme={theme}>
      <BrowserRouter>
        <Provider store={store}>
          <App />
        </Provider>
      </BrowserRouter>
    </ThemeProvider>
  </>
);
import axios from "axios";

const instance = axios.create({
    baseURL: 'http://localhost:4444',//щоб не дописувати постійно самому базовий шлях
});

instance.interceptors.request.use((config) => {//при будь якому запиту - перевіряє наявність токену, якщо нема - додає його
    config.headers.Authorization = window.localStorage.getItem('token');//до авторізації додаємо токен зі сховища локалстореджу
    return config;
})
export default instance;

import { Routes, Route } from 'react-router-dom';
import { useDispatch, useSelector } from 'react-redux';
import Container from "@mui/material/Container";

import { Header } from "./components";
import { Home, FullPost, Registration, AddPost, Login } from "./pages";//компоненти
import React from 'react';
import { fetchAuthMe, selectIsAuthorized } from './redux/slices/auth';
// import { selectIsAuth } from './redux/slices/auth';
function App() {
  const dispatch = useDispatch();
  const isAuth = useSelector(selectIsAuthorized);
```

```
  React.useEffect(() => {
    dispatch(fetchAuthMe());
  }, []);
//основні компоненти блогу
  return (
    <>
      <Header />
      <Container maxWidth="lg">
        <Routes>
          <Route path="/" element={<Home/>} />
          <Route path="/posts/:id/edit" element={<AddPost/>} />
          <Route path="/posts/:id" element={<FullPost/>} />

          <Route path="/add-post" element={<AddPost/>} />
          <Route path="/login" element={<Login/>} />
          <Route path="/register" element={<Registration/>} />
        </Routes>
      </Container>
    </>
  );
}
export default App;
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit'
import axios from '../../axios';
export const fetchAuth = createAsyncThunk('auth/fetchAuth', async(params) => {
    const { data } = await axios.post('/auth/login', params);
    return data;
});

export const fetchRegister = createAsyncThunk('auth/fetchRegister', async(params) => {
    const { data } = await axios.post('/auth/register', params);
    return data;
});
export const fetchAuthMe = createAsyncThunk('auth/fetchAuthMe', async() => {
    const { data } = await axios.get('/auth/me');
    return data;
```

```
});
const initialState = {
  data: null,
  status: 'loading',
};
export const selectIsAuthorized = (state) => Boolean(state.auth.data)
export const reducerAuthorization = authSlice.reducer
export const { logout } = authSlice.actions;
export const fetchPosts = createAsyncThunk('posts/fetchPosts', async()=>{
   const { data } = await axios.get('/posts');//запит на пости
   return data;
});
export const fetchTags = createAsyncThunk('posts/fetchTags', async()=>{
   const { data } = await axios.get('/tags');//запит на теги
   return data;
});
export const fetchRemovePost = createAsyncThunk('posts/fetchRemovePost', async(id)=>
   axios.delete(`/posts/${id}`),//запит на видалення постів
);
//видалення статті
      [fetchRemovePost.pending]: (state, action) => {//завантаження
         state.posts.items = state.posts.items.filter((obj) => obj._id !== action.meta.arg);
      }
export const postsReducer = postsSlice.reducer;
import { configureStore } from '@reduxjs/toolkit';
import { postsReducer } from './slices/posts';
import { reducerAuthorization } from './slices/auth';

const store = configureStore({//сховище редакса
   reducer: {
      posts: postsReducer,
      auth: reducerAuthorization,
   },
});
export default store;

import React from 'react';
```

```
import { useDispatch, useSelector } from 'react-redux'//hook
import Tabs from '@mui/material/Tabs';
import Tab from '@mui/material/Tab';
import Grid from '@mui/material/Grid';
import { Post } from '../components/Post';
import { TagsBlock } from '../components/TagsBlock';
import { fetchPosts, fetchTags } from '../redux/slices/posts';
export const Home = () => {//головна сторінка та її функціонал
  const dispatch = useDispatch();//метод дозволяє відправити дію диспетчеру та змінити стан програми.
  const userData = useSelector((state) => state.auth.data); //вилучатє дані про авторизацію зі стану сховища
  const { posts, tags} = useSelector((state) => state.posts);//стан постів
  const isPostsLoading = posts.status === 'loading';//перевірка на статус завантаження тегів і посту
  const isTagsLoading = tags.status === 'loading';
  React.useEffect(()=> {
    dispatch(fetchPosts());
    dispatch(fetchTags());
  }, []);

  return (
    <>
      <Tabs style={{ marginBottom: 15 }} value={0} aria-label="basic tabs example">
        <Tab label="Останні статті" />
        {/* <Tab label="Популярні" /> */}
      </Tabs>
      <Grid container spacing={4}>
        <Grid xs={8} item>
          {(isPostsLoading ? [...Array(5)] : posts.items).map((obj, index) => //підгрузка усіх статей з бд
          isPostsLoading ? (
          <Post key={index} isLoading={true}/>
          ) : (
            <Post
              id={obj._id}
              title={obj.title}
              imageUrl={obj.imageUrl ? `http://localhost:4444${obj.imageUrl}` : ''}//підгрузка зображення, якщо є,
інаше вивід без нього
              user={obj.user}
              createdAt={obj.createdAt}
```

```
            viewsCount={obj.viewsCount}
            tags={obj.tags}
            isEditable={userData?._id === obj.user._id}
          />
        ),
        )}
      </Grid>
      <Grid xs={4} item>
        <TagsBlock items={tags.items} isLoading={isTagsLoading} /
      </Grid>
    </Grid>
  </>
 );
};
import React from "react";
import { useParams } from "react-router-dom";
import { Post } from "../components/Post";
import axios from "../axios";
import { ReactMarkdown } from "react-markdown/lib/react-markdown";//для гарного відображення тексту в редакторі статті
export const FullPost = () => {
 const [data, setData] = React.useState();
 const [isLoading, setLoading] = React.useState(true);
 const { id } = useParams();
React.useEffect(()=> {
 axios
 .get(`/posts/${id}`)//.get(`http://localhost:4444/posts/${id}`)//такий варіант працює, підгрузка посту
 .then((res) => {
   setData(res.data);
   setLoading(false);
 })
 .catch((err) => {
   console.warn(err);
   alert('Помилка при отриманні статті.')
 });
}, []);
```

```
if (isLoading) {
  return <Post isLoading={isLoading} isFullPost />;
}
  return (
    <>
      <Post
        id={data._id}
        title={data.title}
        imageUrl={data.imageUrl ? `http://localhost:4444${data.imageUrl}`: ''} //для відображення зображення
        //якщо нема зображення то пусте поле
        user={data.user}
        createdAt={data.createdAt}
        viewsCount={data.viewsCount}
        //commentsCount={3}
        tags={data.tags}
        isFullPost>
        <ReactMarkdown children={data.text} />
      </Post>
    </>
  );
};



import styles from './Login.module.scss';
import { fetchRegister, selectIsAuthorized } from '../../redux/slices/auth';//відправка даних реєстрації та перевірка на авторизацію після реєстр.


export const Registration = () => {//реєстрація, компонент та функціонал
  const isAuth = useSelector(selectIsAuthorized)
  const dispatch = useDispatch();
  const {
    register,
    handleSubmit,
    formState: { errors, isValid }
  } = useForm({//форма реєстрації
    defaultValues: {
      fullName: 'Вольфганг Амадей',
```

```
    email: 'wolf@mail.ua',
    password: '11223',
  },
  mode:'onChange',
});

const onSubmit = async (values) => {//відпр форм
  const data = await dispatch(fetchRegister(values));

if (!data.payload) {
  return alert('не вдалось зареєструватись!');
}

  if ('token' in data.payload) {//додавання токена
    window.localStorage.setItem('token', data.payload.token);
  }
};

if (isAuth) {
  return <Navigate to="/" />//на головну якщо нема помилок
}

return (
  <Paper classes={{ root: styles.root }}>
    <Typography classes={{ root: styles.title }} variant="h5">
      Створення акаунта
    </Typography>
    <div className={styles.avatar}>
      <Avatar sx={{ width: 100, height: 100 }} />
    </div>
    <form onSubmit={handleSubmit(onSubmit)}>
    <TextField
        error = {Boolean(errors.fullName?.message)}
        helperText={errors.fullName?.message}
        {...register('fullName', { required: 'вкажіть повне ім`я' })}
        className={styles.field}
        label="Повне ім'я"
```

```
          fullWidth
        />
      <TextField
          error = {Boolean(errors.email?.message)}
          helperText={errors.email?.message}
          type="email"
          {...register('email', { required: 'вкажіть пошту' })}
          className={styles.field}
          label="E-Mail"
          fullWidth
        />
      <TextField
          error = {Boolean(errors.password?.message)}
          helperText={errors.password?.message}
          type="password"
          {...register('password', { required: 'вкажіть пароль' })}
          className={styles.field}
          label="Пароль"
          fullWidth
        />
      <Button disabled={!isValid} type="submit" size="large" variant="contained" fullWidth>
        Зареєструватися
      </Button>
      </form>
    </Paper>
  );
};
import Button from '@mui/material/Button';
import Avatar from '@mui/material/Avatar';

import { Navigate } from 'react-router-dom';
import { useForm } from 'react-hook-form';
import Typography from '@mui/material/Typography';
 import TextField from '@mui/material/TextField';
import Paper from '@mui/material/Paper';

import React from "react";
```

```
import { useDispatch, useSelector } from "react-redux";
import { Navigate } from "react-router-dom";// useNavigate

import Typography from "@mui/material/Typography";
import TextField from "@mui/material/TextField";
import Paper from "@mui/material/Paper";
import Button from "@mui/material/Button";
import { useForm } from 'react-hook-form'
import { fetchAuth, selectIsAuthorized } from "../../redux/slices/auth";

import styles from "./Login.module.scss";

export const Login = () => {//функціонал для авторизації
  const isAuth = useSelector( selectIsAuthorized );//перевірка на авторизованість
  const dispatch = useDispatch();
  const {
    register,
    handleSubmit,
    formState: { errors, isValid }
  } = useForm({
    defaultValues: {
      email: 'vvvasya123@mail.ua',
      password: '12345',
    },
    mode:'onChange',
  });

const onSubmit = async (values) => {//відправка форми авторизування
  const data = await dispatch(fetchAuth(values));

if (!data.payload) {
  return alert('не вдалось авторизуватись!');
}

  if ('token' in data.payload) {
    window.localStorage.setItem('token', data.payload.token);//якщо не буде токену, то його додасть
  }
```

```
};

if (isAuth) {//перехід на головну після авторизації
  return <Navigate to="/" />
} console.log('Login',isAuth);

  return (
    <Paper classes={{ root: styles.root }}>
      <Typography classes={{ root: styles.title }} variant="h5">
        Вхід в аккаунт
      </Typography>
      <form onSubmit={handleSubmit(onSubmit)}>
        <TextField
          className={styles.field}
          label="E-Mail"
          error = {Boolean(errors.email?.message)}
          helperText={errors.email?.message}
          type="email"
          {...register('email', { required: 'вкажіть пошту' })}
          fullWidth
        />
        <TextField
          className={styles.field}
          label="Пароль"
          error = {Boolean(errors.password?.message)}
          helperText={errors.password?.message}
          {...register('password', { required: 'вкажіть пароль' })}
          fullWidth
        />
        <Button disabled={!isValid} type="submit" size="large" variant="contained" fullWidth>
          Вхід
        </Button>
      </form>
    </Paper>
  );
};
.root {
```

```
  width: 400px;
  padding: 50px;
  border: 1px solid #dedede;
  margin: 50px auto;
}

.field {
  margin-bottom: 20px !important;
}

.title {
  text-align: center !important;
  font-weight: bold !important;
  margin-bottom: 30px !important;
}
.root {
  width: 400px;
  padding: 50px;
  border: 1px solid #dedede;
  margin: 50px auto;
}

.field {
  margin-bottom: 20px !important;
}

.title {
  text-align: center !important;
  font-weight: bold !important;
  margin-bottom: 30px !important;
}

.avatar {
  display: flex;
  justify-content: center;
  margin-bottom: 30px;
}
```

```
import React from 'react';
import { useSelector } from 'react-redux';
import TextField from '@mui/material/TextField';
import SimpleMDE from 'react-simplemde-editor';//бібліотека для створення редактору тексту
import Paper from '@mui/material/Paper';
import Button from '@mui/material/Button';
import { useNavigate, Navigate, useParams } from 'react-router-dom';

import 'easymde/dist/easymde.min.css';
import { selectIsAuthorized } from '../../redux/slices/auth';
import axios from '../../axios';
import styles from './AddPost.module.scss';
  const onClickRemoveImage = () => {
    setImageUrl('');//видалення картинки
  };

  const isEditing = Boolean(id);//для розуміння чи ми на сторінці редагування

  const handleChangeFile = async (event) => {
    try {
      const formData = new FormData();
      const file = event.target.files[0];
      formData.append('image', file);
      const { data } = await axios.post('/upload', formData);//запит на завантаження файла при створенні посту
      setImageUrl(data.url);
    } catch (err) {
      console.warn(err);
      alert('помилка при завантаженні файла! ');
    }
  };

export const AddPost = () => {//ф-я для додавання посту
  const {id} = useParams();// передали id посту
  const navigate = useNavigate();//hook
  const isAuth = useSelector(selectIsAuthorized)//
  const [ setLoading] = React.useState(false);//завантаження посту на сервер
  const [text, setText] = React.useState('');//value - зберігає введене в редакторі
```

```
const [title, setTitle] = React.useState('');
const [tags, setTags] = React.useState('');
const [imageUrl, setImageUrl] = React.useState('');
const inputFileRef = React.useRef(null);//hook для отримання зображення



const onChange = React.useCallback((value) => {//отримує value (введений текст)
  setText(value);
}, []);

const onSubmit = async () => { //відправка статті на бекенд
  try {
    setLoading(true);

    const fields = {//передаємо об'єкт з полями на сервер
      title,
      imageUrl,
      tags,//бекенд сам робить split - масив тегів через кому
      text,
    };
    const { data } = isEditing //якщо редагуємо то
    ? await axios.patch(`/posts/${id}`, fields)//такий запит (patch - запит на апдейт посту)
    : await axios.post('/posts', fields);//або інший
    const _id = isEditing ? id : data._id;
    navigate(`/posts/${_id}`);//напривить на конкретний пост по id
  } catch (err) {
    console.warn(err);
    alert('Помилка при створенні статті !');
  }
};
React.useEffect(() => {//перевірка на id посту, для редагування
  if (id) {//якщо id є, то отримуемо відповідь по роуту - пост, для подальшого редагування
    axios
      .get(`/posts/${id}`)
      .then(({data}) => {
        setTitle(data.title);
        setText(data.text);
```

```
        setImageUrl(data.imageUrl);
        setTags(data.tags.join(','));
      }).catch(err => {
        console.warn(err);
        alert('Помилка при отриманні статті!')
      });
    }
  }, []);

  const options = React.useMemo(//useMemo потрібен для роботи стороннього компоненту
    () => ({
      autofocus: true,
      placeholder: 'Введіть текст...',
      spellChecker: false,
      status: false,
      autosave: {
       enabled: true,
    maxHeight: '300px',
        delay: 800,
      },
    }),
    [],
  );

  if (!window.localStorage.getItem('token') && !isAuth) {//якщо не знайшовся токен і ми не авторизовані то будемо направлені на головну
    return <Navigate to="/" />
  }

  return (
    <Paper style={{ padding: 30 }}>
      <Button onClick={() => inputFileRef.current.click()} variant="outlined" size="large">
        Завантажити превью
      </Button>
      <input ref={inputFileRef} type="file" onChange={handleChangeFile} hidden />
      {imageUrl && (
        <>
```

```
          <Button variant="contained" color="error" onClick={onClickRemoveImage}>
            Видалити
          </Button>
          <img className={styles.image} src={`http://localhost:4444${imageUrl}`} alt="Uploaded" />
        </>
      )}
      <TextField
        classes={{ root: styles.title }}
        variant="standard"
        placeholder="Заголовок статті..."
        value={title}
        onChange={(e) => setTitle(e.target.value)}
        fullWidth
      />
      <TextField
        value={tags}
        onChange={(e) => setTags(e.target.value)}
        classes={{ root: styles.tags }}
        variant="standard"
        placeholder="Теги"
        fullWidth
      />
      <SimpleMDE className={styles.editor} value={text} onChange={onChange} options={options} />
      <div className={styles.buttons}>
        <Button onClick={onSubmit} size="large" variant="contained">
          {isEditing ? 'Зберегти': 'Опублікувати'}
        </Button>
        <a href="/">
          <Button size="large">Відміна</Button>
        </a>
      </div>
    </Paper>
  );
};

import React from "react";
import { SideBlock } from "./SideBlock";
```

```
import ListItemIcon from "@mui/material/ListItemIcon";
import TagIcon from "@mui/icons-material/Tag";
import List from "@mui/material/List";
import ListItem from "@mui/material/ListItem";
import ListItemText from "@mui/material/ListItemText";
import Skeleton from "@mui/material/Skeleton";
import ListItemButton from "@mui/material/ListItemButton";
export const TagsBlock = ({ items, isLoading = true }) => {//вивід 5ти тегів з постів на головній сторінці, або скелетон при завантаженні
  return (
    <SideBlock title="Теги">
      <List>
        {(isLoading ? [...Array(5)] : items).map((name, i) => (
          <a
            style={{ textDecoration: "none", color: "black" }}
            href={`/tags/${name}`}
          >
            <ListItem key={i} disablePadding>
              <ListItemButton>
                <ListItemIcon>
                  <TagIcon />
                </ListItemIcon>
                {isLoading ? (
                  <Skeleton width={100} />  ) : (
                  <ListItemText primary={name} />
                )}
              </ListItemButton>
            </ListItem>
          </a>
        ))}
      </List>
    </SideBlock>
  );
};
import React from 'react';
import styles from './UserInfo.module.scss';
```

```
export const UserInfo = ({ avatarUrl, fullName, additionalText }) => {//інформація про користувача
  return (
    <div className={styles.root}>
          <div className={styles.userDetails}>
        <span className={styles.userName}>{fullName}</span>
        <span className={styles.additional}>{additionalText}</span>
      </div>
    </div>
  );
};
.userName {
  font-weight: 500;
  font-size: 14px;
}
.userDetails {
  display: flex;
  flex-direction: column;
}
.additional {
  font-size: 12px;
  opacity: 0.6;
}
export const SideBlock = ({ title, children }) => {//бокова панель, її вивід
  return (
    <Paper classes={{ root: styles.root }}>
      <Typography variant="h6" classes={{ root: styles.title }}>
        {title}
      </Typography>
      {children}
    </Paper>  ); };
import clsx from 'clsx';
import IconButton from '@mui/material/IconButton';
import DeleteIcon from '@mui/icons-material/Clear';
import EditIcon from '@mui/icons-material/Edit';
import EyeIcon from '@mui/icons-material/RemoveRedEyeOutlined';
import CommentIcon from '@mui/icons-material/ChatBubbleOutlineOutlined';
import styles from './Post.module.scss';
```

```
import { PostSkeleton } from './Skeleton';
import { fetchRemovePost } from '../../redux/slices/posts';
  const onClickRemove = () => {
    if (window.confirm('Ви впевнені, що бажаєте видалити статтю ?')) {
      dispatch(fetchRemovePost(id));    }
export const Post = ({//дані посту
  id,  title,
  createdAt,//час створення
  imageUrl,
  user,//хто автор
  tags, viewsCount, children,
  isFullPost,//відритий чи ні
  isLoading,//завантжений чи ні
  isEditable,//чи редагується
}) => {
  const dispatch = useDispatch();
  if (isLoading) {
    return <PostSkeleton />;//вивід скелетона, тобто візуалізація завантаження поки не прогрузиться
  }  };
//повертає пост, якщо не завантажився то виводить візуал процесу завантаження
  return (
    <div className={clsx(styles.root, { [styles.rootFull]: isFullPost })}>
      {isEditable && (
        <div className={styles.editButtons}>
          <Link to={`/posts/${id}/edit`}>
            <IconButton color="primary">
              <EditIcon />
            </IconButton>
          </Link>
          <IconButton onClick={onClickRemove} color="secondary">
            <DeleteIcon />
          </IconButton>
        </div>
      )}
      {imageUrl && (
        <img
          className={clsx(styles.image, { [styles.imageFull]: isFullPost })}
```

```
        src={imageUrl}
        alt={title}
      />     )}
    <div className={styles.wrapper}>
      <UserInfo {...user} additionalText={createdAt} />
      <div className={styles.indention}>
        <h2 className={clsx(styles.title, { [styles.titleFull]: isFullPost })}>
          {isFullPost ? title : <Link to={`/posts/${id}`}>{title}</Link>}
        </h2>
        <ul className={styles.tags}>
          {tags.map((name) => (
            <li key={name}>
              <Link to={`/tag/${name}`}>#{name}</Link>
            </li>
          ))}
        </ul>
        {children && <div className={styles.content}>{children}</div>}
        <ul className={styles.postDetails}>
          <li>
            <EyeIcon />
            <span>{viewsCount}</span>
          </li>
        </ul>
      </div>
    </div>
  </div>
 );
};
import { Link } from 'react-router-dom';
import Button from '@mui/material/Button';
import styles from './Header.module.scss';
import Container from '@mui/material/Container';
import { logout, selectIsAuthorized } from '../../redux/slices/auth';
export const Header = () => {
 const dispatch = useDispatch();
 const isAuth = useSelector( selectIsAuthorized );//змінюємо хедер в залежності від авторизованості користувача
```

```
const onLogout = () => {//вихід з акаунту
  if (window.confirm('Ви впевнені, що бажаєте вийти ?')) {
    dispatch(logout());
    window.localStorage.removeItem('token');//після виходу з акаунту прибераємо токен з локалстореджа
  }
};
return (
  <div className={styles.root}>
    <Container maxWidth="lg">
      <div className={styles.inner}>
        <Link className={styles.logo} to="/">
          <div>My_Diplom_Blog</div>
        </Link>
        <div className={styles.buttons}>
          {isAuth ? (
            <>
              <Link to="/add-post">
                <Button variant="contained">Написати статтю</Button>
              </Link>
              <Button onClick={onClickLogout} variant="contained" color="error">
                Вийти
              </Button>
            </>
          ) : (
            <>
              <Link to="/login">
                <Button variant="contained">Увійти</Button>
              </Link>
              <Link to="/register">
                <Button variant="contained">Створити аккаунт</Button>
              </Link>
            </>
          )}
        </div>
      </div>
    </Container>
  </div>
```

```
  );
};
.root {
  background-color: #949494;
  padding: 10px 0;
  border-bottom: 1px solid #e0e0e0;
  margin-bottom: 30px;
}
.logo {
  background-color: black;
  color: #fff;
  line-height: 35px;
  text-transform: uppercase;
  letter-spacing: 0.15px;
  border-radius: 5px;
  padding: 0 10px;
  font-weight: 700;
  text-decoration: none;
  &:hover {
    background-color: #4361ee;
  }
}
.buttons {
  button {
    margin-left: 10px;
    background-color: #151515;
  }
}
.inner {
  display: flex;
  justify-content: space-between;
}
```