Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка веб-платформи для створення та введення блогів з використанням React, Node.js та MongoDB

Виконав(ла):
студент(ка) 4 курсу групи СП-41
спеціальності 121
«Інженерія програмного забезпечення»
(шифр і назва спеціальності)

_____ Марчук Д. В.
(підпис) (прізвище та ініціали)

Керівник _____ Мудрик І. Я.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю. А.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М. Р.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2024

АНОТАЦІЯ

Кваліфікаційна робота бакалавра. Тернопільський національний технічний університет імені Івана Пулюя, кафедра програмної інженерії, спеціальність 121 «Інженерія програмного забезпечення». ТНТУ, 2024. Сторінок 79, рисунків 17, додатків 5, презентація.

Тема: Розробка веб-платформи для створення та введення блогів з використанням React, Node.js та MongoDB.

Кваліфікаційна робота бакалавра присвячена розробці веб-платформи для створення та управління блогами з використанням React, Node.js та MongoDB.

Метою цієї роботи є створення ефективної та зручної веб-платформи, яка дозволяє користувачам взаємодіяти з блогами, використовуючи можливості сучасних технологій веб-розробки.

У першому розділі проводиться аналіз обраної предметної області, визначаються вимоги до платформи для блогів. Розробляються діаграми варіантів використання, обґрунтовується вибір середовища розробки та ключових технологій, включаючи React для фронтенду, Node.js для серверної частини та MongoDB як базу даних.

Другий розділ кваліфікаційної роботи описує проектування бази даних, моделювання архітектури, а також деталі процесу розробки як серверної, так і клієнтської частин додатку. Розглядаються ключові функції та процедури тестування, впроваджені для забезпечення функціональності та продуктивності.

Об'єктом дослідження є сучасна веб-платформа для ведення блогів та нереляційна база даних MongoDB.

Предметом дослідження є інструменти та методи розробки веб-додатків для блогінгу з використанням стека MERN (MongoDB, Express.js, React, Node.js).

Ключові слова: Full-Stack розробка, MERN Stack, NoSQL база даних, проектування, блогова платформа, реєстрація користувача.

ANNOTATION

Bachelor's certification work. Ternopil Ivan Puluj National Technical University, Department of Software Engineering, specialty 121 "Software Engineering". TNTU, 2024. Pages 79, figures 17, appendices 5, presentation.
Topic: Development of a web platform for creating and managing blogs using React, Node.js, and MongoDB.

The bachelor's certification work is focused on the development of a web-platform for creating and managing blogs using React, Node.js, and MongoDB.

The purpose of this study is to create an efficient and user-friendly platform that allows users to interact with blogs, leveraging the capabilities of modern web development technologies.

In the first section, an analysis of the selected subject area is conducted, detailing the requirements for the blogging platform. Use-case diagrams are developed, and the selection of the development environment and key technologies, including React for the frontend, Node.js for the server-side, and MongoDB as the database, is justified.

The second section describes the database design for the web-application, architecture modeling, and details the development process of both the server and client parts of the application. Key features and challenges encountered during development are discussed, along with the testing procedures implemented to ensure functionality and performance.

The object of research is a modern web-platform for blogging and the non-relational database MongoDB.

The subject of research is the tools and methods of developing web-applications for blogging using the MERN (MongoDB, Express.js, React, Node.js) stack.

Keywords: Full-Stack development, MERN Stack, NoSQL database, projecting, blogging platform, user registration.

ЗМІСТ

ВСТУП

У цифрову епоху можливість ділитися ідеями, історіями та інформацією за допомогою написання блогів стала невід'ємною частиною нашої онлайн-культури. Оскільки як окремі особи, так і організації шукають ефективні платформи для спілкування зі своїми аудиторіями, попит на зручні та універсальні системи управління блогами продовжує зростати. Ця дипломна робота зосереджена на розробці надійної веб-платформи, призначеної для створення та управління блогами, з використанням сучасних веб-технологій React, Node.js та MongoDB.

Метою створення даної веб-платформи є розробка інтуїтивно зрозумілого та високопродуктивного інструменту для ведення блогів, який дозволить користувачам легко створювати, редагувати та публікувати контент. Платформа повинна забезпечувати зручне управління блогами, включаючи функції коментування та категоризації постів. Крім того, система має бути адаптивною та масштабованою, здатною обробляти велику кількість користувачів і контенту без втрати продуктивності. Використовуючи переваги React для фронтенду, Node.js для бекенду та MongoDB для бази даних, платформа прагне забезпечити безперебійну роботу користувача, одночасно гарантуючи оптимальну продуктивність та гнучкість. Важливою складовою є також забезпечення безпеки даних користувачів та надійність зберігання інформації.

React, популярна JavaScript-бібліотека для створення користувацьких інтерфейсів, обрана завдяки своїй компонентній архітектурі та здатності створювати динамічні, високопродуктивні односторінкові додатки (SPA). Завдяки застосуванню віртуального DOM, React ефективно оновлює та рендерить лише ті компоненти, які зазнали змін, що забезпечує швидку та інтерактивну взаємодію з користувачем.

Node.js, серверне середовище виконання JavaScript, пропонує потужну, подієво-орієнтовану архітектуру, яка дозволяє створювати масштабовані мережеві додатки. Використання єдиного мовного середовища для фронтенду і бекенду

значно спрощує процес розробки та підтримки додатка, забезпечуючи швидкий обмін даними між клієнтською і серверною частинами.

MongoDB, NoSQL-база даних, обрана завдяки своїй гнучкості у обробці різних типів даних та можливості масштабування для обробки зростаючих обсягів інформації. Документна модель даних MongoDB дозволяє зберігати дані у форматі JSON, що забезпечує просту інтеграцію з JavaScript-додатками та ефективне зберігання й пошук інформації.

Ключові етапи розробки веб-платформи включали в себе ретельне проектування архітектури системи з урахуванням вимог до продуктивності, масштабованості та безпеки. Після цього було виконано реалізацію базового функціоналу, включаючи можливість створення, редагування та видалення блогових постів, а також управління коментарями і категоріями. Під час цього етапу активно використовувалися методики Agile-розробки для забезпечення поетапного розвитку продукту та швидкого реагування на зміни вимог.

Під час розробки платформи особлива увага приділялася тестуванню та забезпеченню якості програмного забезпечення. Для цього використовувалися різноманітні методи тестування, включаючи UNIT-тестування, інтеграційне тестування та тестування користувацького інтерфейсу. Автоматизовані тести були створені з використанням таких інструментів, як Jest та Mocha для Node.js, а також React Testing Library для фронтенду.

Впровадження системи передбачає розгортання веб-платформи на хмарному сервісі AWS, що забезпечує високу доступність та масштабованість додатка. Процес впровадження включає налаштування безперервної інтеграції та безперервного розгортання (CI/CD) з використанням різних інструментів, що дозволять автоматизувати процеси тестування та розгортання.

Завдяки цьому проекту буде продемонстровано, як сучасні веб-технології можуть бути використані для створення ефективного інструменту для ведення блогів, надаючи цінні знання та практичний досвід для майбутніх розробок у сфері веб-додатків.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ БЛОГОВИХ ПЛАТФОРМ

1.1 Аналіз предметної області блогової платформи

В сучасному світі блоги залишаються важливим засобом висловлення думок, обміну інформацією та створення спільнот. З розвитком Інтернету та соціальних медіа вони стали не тільки платформою для особистого вираження, але й засобом комунікації, маркетингу та навіть заробітку.

Проте, існуючі платформи для створення блогів можуть мати обмежену функціональність, бути складними у використанні або не забезпечувати потрібного рівня безпеки та приватності для користувачів. Вибір правильної платформи може стати проблемою для багатьох авторів та власників веб-блогів.

Одним із актуальних викликів є потреба у сучасних технологіях та інструментах для розробки платформ, які б не лише забезпечували функціональність, але й були зручними для використання, мали чистий та привабливий інтерфейс, а також гарантували високий рівень безпеки даних користувачів.

Тому розробка нової веб-платформи для створення та введення блогів, заснованої на передових технологіях, таких як React, Node.js та MongoDB, є актуальною та важливою задачею, яка може відповісти на потреби сучасних авторів та користувачів Інтернету.

Для того, щоб розробити унікальні переваги для власного продукту необхідно провести аналіз конкурентів та зібрати важливі відомості про те, що вже існує та які потреби задовольняються. На основі цього аналізу можна виявити можливості для покращення та уникнути повторення помилок конкурентів. Також цей аналіз допомагає краще зрозуміти аудиторію та її вимоги, що є ключовим аспектом при створенні ефективного та конкурентоспроможного продукту.

Medium – платформа для публікації контенту, яка набула значної популярності в останні роки. Вона відрізняється зручним інтерфейсом та спрощеним процесом публікації, що робить її ідеальним варіантом для тих, хто

хоче зосередитися на контенті, а не на технічних деталях веб-сайту [1]. Medium також надає можливості для спільного публікування, коментування та взаємодії з іншими авторами.

При використанні даного сервісу можна виділити наступні переваги: Medium пропонує чистий, мінімалістичний інтерфейс, який полегшує читання та публікацію контенту, користувачі можуть публікувати статті на Medium та розповідати історії разом з іншими авторами, Medium має широку аудиторію, що дозволяє новим авторам отримати більше читачів та взаємодіяти з іншими користувачами. З недоліків було виявлено те, що у порівнянні з іншими платформами, Medium має обмежені можливості кастомізації дизайну та функціональності та жорсткий контроль за контентом, що означає, що іноді автори можуть зіткнутися з обмеженнями або втратити контроль над своїм контентом.

Blogger – інша відома платформа для створення блогів, яка належить компанії Google. Вона пропонує простий у використанні інтерфейс, інтеграцію з іншими сервісами Google та безкоштовне хостинг. Її інтуїтивний інтерфейс дозволяє швидко створювати та редагувати контент. Інтеграція з Google дає можливість легко інтегрувати платформу з іншими сервісами компанії, такими як Google Analytics та AdSense [2].

Проте, Blogger має обмежені можливості кастомізації, що може не задовольняти потреби користувачів, що шукають більше гнучкості в дизайні та функціональності. Також, оскільки це сервіс з хостингом, користувачі не мають такого ж контролю над розвитком та розширенням платформи, як у випадку з самостійними інсталяціями WordPress, наприклад.

1.2 Формування вимог до веб-платформи для ведення блогів

Для забезпечення успіху на ринку, при розробці веб-платформи для ведення блогів, необхідно врахувати кілька ключових аспектів. Визначення

функціональних і нефункціональних вимог, а також вимог до інтерфейсу користувача, відіграє вирішальну роль у створенні платформи, яка буде відповідати потребам користувачів та забезпечувати їм найкращий досвід.

Простота використання є важливою характеристикою будь-якої веб-платформи для ведення блогів. Користувачі повинні мати змогу легко створювати та редагувати свої записи без зайвих труднощів. Крім того, організація контенту грає ключову роль у забезпеченні зручного пошуку та навігації для користувачів. Можливість категоризації та тегування записів допомагає у цьому.

Взаємодія користувачів також є важливою функціональною вимогою. Система коментування та можливість модерації коментарів сприяє активному обговоренню та взаємодії серед користувачів. Крім того, створення особистих мереж та можливість обміну контентом сприяє зближенню та спільноті.

Привабливий та доступний дизайн є ключовою нефункціональною вимогою. Інтерфейс повинен бути зрозумілим і привабливим, з мінімальним навантаженням для користувача. Крім того, важливо забезпечити адаптивність для різних пристроїв та розмірів екранів.

Безпека є важливою складовою будь-якої веб-платформи, особливо коли йдеться про збереження особистих даних користувачів. Надійний захист даних включає використання шифрування та інших методів для запобігання несанкціонованому доступу до особистої інформації користувачів. Крім того, важливо встановити механізми для виявлення та запобігання можливих загроз безпеці, таких як злому акаунтів або атаки на систему.

Один із способів підвищення безпеки платформи полягає в інтеграції з відомими системами ідентифікації, такими як Google або AWS. Це дозволяє користувачам використовувати вже існуючі облікові записи для входу на платформу, забезпечуючи швидкий та зручний доступ, а також використання надійних методів аутентифікації. Такий підхід сприяє не лише зручності для користувачів, а й підвищує рівень безпеки, оскільки відбувається використання вже перевірених механізмів захисту [3].

Нарешті, продуктивність визначається швидкістю завантаження сторінок та їх ефективністю у роботі з великим обсягом контенту та користувачів. Масштабованість платформи є також важливою, оскільки вона дозволяє забезпечити продуктивну роботу у майбутньому при зростанні користувачів та обсягу контенту.

1.3 Опис варіантів використання блогової платформи

Діаграма варіантів використання допомагає уявити, як система буде використовуватися різними категоріями користувачів та які дії вони зазвичай виконують. Це важливий інструмент для визначення функціональних вимог системи і забезпечення її відповідності потребам користувачів.

На діаграмі варіантів використання зображено наступні компоненти: актори, що представляють користувачів або інші системи, які взаємодіють із системою. Вони можуть бути як реальними людьми, так і іншими системами, що здійснюють взаємодію через API. Варіанти використання, що описують дії, які можуть бути виконані актором у межах системи. Кожен варіант використання представляє окремий функціональний аспект системи Зв'язки, що показують, які актори взаємодіють з якими варіантами використання.

На діаграмі варіантів використання будуть зображені основні дії, які користувачі можуть виконувати в системі, включаючи створення блогів, написання та редагування постів, коментування і перегляд профілю. Ця діаграма дозволить нам краще розуміти, як система буде функціонувати в реальному середовищі та як користувачі будуть взаємодіяти з нею.

В системі виділено наступні типи акторів:

– Користувач: Представляє усіх користувачів системи, які можуть здійснювати різні дії на платформі.

– Авторизований користувач: Користувач, який увійшов у систему під своїм обліковим записом і має доступ до створення та редагування блогів і постів.

Діаграма варіантів використання для розуміння поведінки системи зображена на рисунку (див. рис. 1.1).

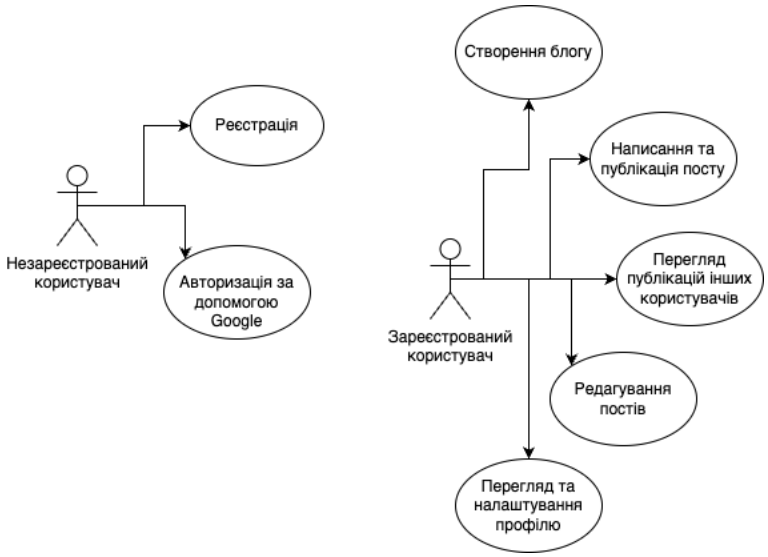

Рисунок 1.1 – Діаграма варіантів використання

Опис типових сценаріїв використання:

– Створення нового блогу: Користувач увійшовши до системи, має можливість створити новий блог. Він обирає опцію "Створити новий блог", після чого заповнює необхідну інформацію про блог, таку як назва, опис, тематика тощо. Після підтвердження, блог додається до списку блогів користувача. Ключові дії: Вхід в систему, заповнення форми, підтвердження створення блогу.

– Написання та публікація посту: Користувач обирає блог, в якому хоче опублікувати новий пост. Він створює новий пост, додаючи заголовок, текст та можливо зображення або інші медіа. Після завершення, пост публікується у вибраному блозі. Ключові дії: Вибір блогу, створення посту, додавання контенту, публікація посту.

– Редагування посту: Користувач може редагувати свої опубліковані пости. Він вибирає пост, який потрібно змінити, вносить необхідні зміни та зберігає оновлену версію посту. Ключові дії: Вибір посту для редагування, внесення змін, збереження оновленої версії.

– Коментування постів: Користувач може залишити коментар під будь-яким опублікованим постом. Він переглядає пости, обирає той, під яким хоче залишити коментар, та додає свій коментар до обговорення. Ключові дії: Вибір посту для коментування, написання коментаря, публікація коментаря.

– Перегляд профілю і налаштувань: Користувач може переглядати та редагувати свій профіль та налаштування, такі як зміна паролю чи електронної адреси. Ключові дії: Вхід в профіль, перегляд інформації, редагування налаштувань, збереження змін.

1.4 Вибір середовища розробки

Вибір середовища розробки є критично важливим етапом у створенні веб-платформи для ведення блогів. Це рішення впливатиме на продуктивність, масштабованість, безпеку та зручність підтримки проекту. Процес розробки включатиме наступні етапи:

– Розробка фронтенду: для створення користувацького інтерфейсу обрано технологію React, оскільки він забезпечує високу продуктивність завдяки віртуальному DOM, який мінімізує маніпуляції зі справжнім DOM і оптимізує оновлення інтерфейсу користувача. Це дозволить створити швидкий та інтерактивний веб-застосунок. React застосовує компонентну архітектуру, яка дозволяє розробникам створювати незалежні та багаторазові компоненти. Це полегшить розробку, тестування та підтримку коду. React має велику спільноту розробників, що забезпечить наявність багатьох ресурсів, інструментів та бібліотек, що прискорить розробку. Також завдяки можливості використання серверного рендерингу, React покращує SEO-оптимізацію веб-додатків, що є важливим для блогової платформи, де важливо, щоб контент був доступним для пошукових систем [4].

– Розробка бекенду: для розробки серверної частини обрано технологію Node.js, оскільки це єдина мова програмування, що дозволить уніфікувати розробку за допомогою використання JavaScript як на клієнтській, так і на серверній стороні. Це полегшить обмін кодом між фронтендом і бекендом. Також завдяки величезній екосистемі npm (Node Package Manager), Node.js пропонує безліч модулів і бібліотек, що полегшують розробку та розширення функціональності додатку [5]. Node.js дозволяє легко масштабувати додаток горизонтально, розподіляючи навантаження між різними процесами та серверами. Це важливо для забезпечення стабільної роботи платформи при зростанні кількості користувачів.

– Інтеграція бази даних: для зберігання даних обрано технологію MongoDB. MongoDB є документною базою даних, яка дозволяє зберігати дані у форматі JSON-подібних документів. Це надає гнучкість у зміні структури даних без необхідності складних міграцій, що ідеально підходить для динамічних веб-застосунків [6]. Завдяки індексуванню MongoDB забезпечує швидкий доступ до даних і можливість виконання складних запитів. MongoDB дозволяє зберігати вкладені документи і складні структури даних, що спрощує моделювання даних для блогів, де можуть бути різні типи контенту, коментарі та категорії.

2 ПРОЕКТУВАННЯ ТА РОЗРОБКА БЛОГОВОЇ ПЛАТФОРМИ

2.1 Огляд підходу до тестування та розробки блогової платформи

У цьому розділі буде детально розглянуто підхід до проектування та розробки веб-платформи для ведення блогів. Основна мета – створення зручного, функціонального та ефективного інструменту для користувачів, які бажають публікувати та редагувати контент в інтернеті. Цей підпункт має на меті представити загальний огляд процесу розробки, підходів і методологій, що використовувалися для досягнення цієї мети.

Процес розробки розпочався з ретельного планування та аналізу вимог. Було зібрано всі необхідні вимоги за допомогою аналізу конкурентів, а також визначено функціональні та нефункціональні характеристики платформи. Цей етап є фундаментальним для успішної реалізації проекту, оскільки забезпечує чітке розуміння того, що має бути розроблено і як це буде відповідати потребам користувачів.

Для розробки платформи було обрано гнучку методологію Agile, яка дозволяє ефективно реагувати на зміни вимог і забезпечує поступове вдосконалення продукту. Гнучкість і адаптивність Agile дозволяє швидко реагувати на зміни вимог та пріоритетів. Це особливо важливо в динамічних середовищах, де потреби клієнтів та ринкові умови можуть змінюватися дуже швидко. Покращена якість продукту завдяки регулярному тестуванню та зворотному зв'язку команди можуть своєчасно виявляти і виправляти помилки. Це дозволяє підтримувати високий рівень якості продукту протягом всього циклу розробки. Загалом, Agile є потужним підходом до розробки програмного забезпечення, який дозволяє підвищити гнучкість, продуктивність і задоволеність клієнтів. Використання Agile методологій допомагає швидко адаптуватися до змін і постійно вдосконалювати продукт, що є критично важливим у сучасному світі технологій [7].

Архітектурне проектування є ключовим етапом у розробці веб-платформи для ведення блогів, оскільки воно визначає загальну структуру системи та забезпечує відповідність всіх її компонентів вимогам проекту. Цей процес починається з визначення функціональних і нефункціональних вимог, таких як продуктивність, масштабованість і безпека.

Вибір архітектурного стилю, наприклад "клієнт-сервер" або мікросервіси, визначає, як компоненти системи будуть взаємодіяти між собою. Для веб-платформи зазвичай обирають React для фронтенду через його ефективність і компонентну архітектуру, Node.js для бекенду завдяки високій продуктивності та MongoDB як базу даних через її гнучкість.

Проектування компонентів включає детальну розробку інтерфейсу користувача, API, бази даних і систем автентифікації. Важливо визначити, як компоненти взаємодіють через інтерфейси та протоколи, такі як HTTP/HTTPS і JSON. Особлива увага приділяється безпеці, зокрема шифруванню даних, багатофакторній автентифікації та захисту від атак.

Масштабованість і продуктивність забезпечуються через використання балансувальників навантаження та оптимізацію коду. Правильно спроектована архітектура дозволяє платформі легко адаптуватися до змін вимог і впроваджувати нові функції, забезпечуючи її успішний розвиток і підтримку.

Розробка бази даних є критичним етапом у створенні будь-якої веб-платформи, особливо якщо мова йде про платформу для створення та введення блогів. MongoDB обрано в якості бази даних для проекту через декілька причин: гнучкість схеми даних, горизонтальне масштабування, підтримка JSON-подібних документів та швидкість розробки. MongoDB є NoSQL базою даних, що означає, що вона не вимагає строгої схеми, у порівнянні з традиційними реляційними базами даних. Це дозволяє легко змінювати структуру даних під час розвитку проекту без необхідності виконання складних міграцій. MongoDB добре підходить для проектів, які мають високі навантаження або потребують горизонтального масштабування. Ви можете додавати нові сервери для обробки більшого обсягу даних без значного перепроектування вашої бази даних.

MongoDB зберігає дані у форматі JSON-подібних документів, що робить його досить природнім вибором для веб-розробки, оскільки дані можуть легко відображатися на об'єктах JavaScript. Крім того, MongoDB спрощує розробку за рахунок своєї гнучкості та простоти у використанні.

Фронтенд був розроблений з використанням React для створення динамічного та інтерактивного інтерфейсу користувача. Компонентний підхід React дозволяє розробляти незалежні та багаторазові компоненти, що спрощує підтримку та розвиток платформи.

Бекенд, побудований на Node.js, забезпечує швидку та ефективну обробку запитів від клієнтів. Серверна частина відповідає за автентифікацію користувачів, обробку даних і взаємодію з базою даних. Використання Node.js дозволяє мати єдину мову програмування на обох кінцях системи, що спрощує розробку та підтримку.

Інтеграція різних компонентів системи проводилася поступово, з постійним тестуванням на кожному етапі. Для забезпечення якості програмного забезпечення використовувалися автоматизовані та ручні методи тестування. Особлива увага приділялася тестуванню функціональності, безпеки та продуктивності платформи.

2.2 Проектування бази даних

При проектуванні бази даних для веб-платформи для ведення блогів, створення відповідної структури бази даних базується на потребах і функціональних можливостях системи. Ось короткий огляд кожної колекції і її значення в контексті платформи:

Схема blog містить в собі наступні поля:

- blog_id: Унікальний ідентифікатор блогу.
- title: Заголовок блогу.
- banner: URL зображення, що використовується як банер для блогу.
- des: Короткий опис блогу, обмежений 200 символами.

– content: Масив змісту блогу.

– tags: Масив тегів, які описують блог.

– author: Посилання на автора блогу.

– activity: Інформація про активність блогу (лайки, коментарі, кількість переглядів).

– comments: Посилання на коментарі, що відносяться до цього блогу.

– draft: Позначка, що вказує, що блог є чернеткою.

– timestamps: Дата та час створення блогу.

Колекція blogs призначена для зберігання інформації про блоги, що створені користувачами. Вона дозволяє відображати вміст статей, їх заголовки, теги та інші деталі, а також забезпечує можливість взаємодії з коментарями, лайками та іншими діями, пов'язаними з блогами.

Кожен запис у цій колекції представляє блог на платформі. Поля блога включають заголовок, зображення-банер, короткий опис, вміст блогу, теги, автора, активність (лайки, коментарі, читання), коментарі та інформацію про статус чернетки. Кожен блог пов'язаний з користувачем через поле author, що містить посилання на користувача, який створив цей блог. Також, кожен блог може містити коментарі, представлені в полі comments, яке містить масив посилань на коментарі.

Схема comments:

– blog_id: Посилання на блог, до якого належить коментар.

– blog_author: Посилання на автора блогу, до якого належить коментар.

– comment: Текст коментаря.

– children: Посилання на дочірні коментарі (відповіді) до поточного коментаря.

– commented_by: Посилання на користувача, який залишив коментар.

– isReply: Позначка, що вказує, чи є цей коментар відповіддю на інший.

– parent: Посилання на батьківський коментар, якщо цей коментар є відповіддю на інший.

– timestamps: Дата та час створення коментаря.

Колекція comments використовується для зберігання коментарів, що залишаються користувачами під блогами. Вона дозволяє організувати обговорення статей та взаємодію між користувачами, а також забезпечує можливість створення відповідей на коментарі та їхню структуру.

Кожен запис у цій колекції представляє коментар до блогу. Поля коментаря включають текст коментаря, посилання на блог, до якого належить коментар, автора коментаря та інформацію про відповіді та батьківський коментар. Коментарі можуть мати дочірні коментарі, які зберігаються в полі children, а також батьківський коментар, до якого вони належать (якщо такий є).

Схема notification:

- type: Тип сповіщення (лайк, коментар, відповідь).
- blog: Посилання на блог, до якого відноситься сповіщення.
- notification_for: Посилання на користувача, якому призначено сповіщення.
- user: Посилання на користувача, який викликав сповіщення (лайкнув, залишив коментар).
- comment: Посилання на коментар, який викликав сповіщення (якщо це сповіщення про коментар).
- reply: Посилання на відповідь, яка викликала сповіщення (якщо це сповіщення про відповідь на коментар).
- replied_on_comment: Посилання на коментар, до якого була відповідь (якщо це сповіщення про відповідь на коментар).
- seen: Позначка, що вказує, чи було сповіщення переглянуте.

Колекція notification використовується для створення сповіщень про події на платформі, таких як лайки, коментарі та відповіді. Вона дозволяє повідомляти користувачів про активність та важливі події на платформі, а також відстежувати та керувати сповіщеннями.

Кожен запис у цій колекції представляє сповіщення, яке створюється при події на платформі (наприклад, лайк, коментар, відповідь). Сповіщення можуть бути пов'язані з блогами, користувачами, коментарями та відповідями. Поля

включають тип сповіщення, посилання на блог, користувача, який створив сповіщення, та інші відповідні посилання на елементи, що стосуються події. Інформація про статус сповіщення (прочитане або непрочитане) зберігається в полі seen.

Схема user:

- personal_info: Особиста інформація про користувача (ім'я, електронна пошта, пароль, ім'я користувача, біографія, посилання на аватарку).
- social_links: Посилання на соціальні мережі користувача.
- account_info: Інформація про активність користувача (кількість опублікованих постів, кількість читань).
- google_auth: Позначка, що вказує, чи користувач автентифікувався через Google.
- blogs: Посилання на блоги, які належать цьому користувачу.
- timestamps: Дата та час реєстрації користувача.

Колекція users використовується для зберігання даних про користувачів платформи, зокрема їх особистої інформації, соціальних посилань та активності. Вона дозволяє керувати доступом та привілеями користувачів, а також забезпечує можливість взаємодії користувачів зі своїми профілями та іншими аспектами платформи.

Кожен запис у цій колекції представляє користувача платформи. У кожного користувача є особиста інформація (ім'я, електронна пошта, пароль, ім'я користувача, біографія, посилання на аватарку), соціальні посилання та інформація про активність (кількість опублікованих постів, кількість читань). Кожен користувач може мати декілька блогів, тому є поле blogs, яке містить масив посилань на блоги, які належать цьому користувачу.

Взаємозв'язки між колекціями здійснюються за допомогою посилань на об'єкти (ID або посилання) у відповідних полях. Наприклад, блог може посилатися на автора через його ID у полі author, а коментар може посилатися на блог, до якого він належить, та на автора коментаря. Таким чином, створюються зв'язки між різними елементами бази даних.

2.3 Моделювання архітектури системи блогової платформи

Структура фронтенду цього додатку побудована згідно з принципами компонентної архітектури React. Основні частини фронтенду включають компоненти і сторінки, що відповідають за різні аспекти відображення та взаємодії з користувачем. Давайте розглянемо структуру додатку ближче:

– Navbar (components/navbar.component.js): Цей компонент відповідає за верхню панель навігації, яка містить логотип, пошук та посилання на різні частини додатку.

– SideNav (components/sidenavbar.component.js): Компонент бічної панелі навігації, що містить посилання на основні функції користувача, такі як управління блогами, налаштування, сповіщення тощо.

– UserAuthForm (pages/userAuthForm.page.js): Компонент для авторизації та реєстрації користувача.

– Editor (pages/editor.page.js): Сторінка для створення та редагування блогів.

– HomePage (pages/home.page.js): Головна сторінка додатку, де відображаються останні блоги або рекомендації.

– SearchPage (pages/search.page.js): Сторінка для пошуку блогів за ключовими словами.

– ProfilePage (pages/profile.page.js): Сторінка профілю користувача.

– BlogPage (pages/blog.page.js): Сторінка окремого блогу з можливістю перегляду та коментування.

– ChangePassword (pages/change-password.page.js): Сторінка для зміни паролю користувача.

– EditProfile (pages/edit-profile.page.js): Сторінка для редагування особистої інформації користувача.

– Notifications (pages/notifications.page.js): Сторінка зі списком сповіщень для користувача.

– ManageBlogs (pages/manage-blogs.page.js): Сторінка для керування блогами (редагування, видалення тощо).

– Для маршрутизації використовується бібліотека react-router-dom, яка дозволяє визначити шляхи та компоненти, що пов'язані з ними. Маршрутизація вбудована у структуру компонентів, що дозволяє відображати різні частини додатку залежно від URL.

– Використовуються контексти UserContext та ThemeContext для збереження стану користувача та теми додатку відповідно. Це дозволяє передавати ці дані усім компонентам, які їх потребують, без необхідності прокидання їх через пропси.

Тепер необхідно візуалізувати дану інформацію, для цього використаємо UML-діаграму класів. Діаграми класів у рамках UML є важливим інструментом для моделювання структури програмного забезпечення. Вони надають зручний спосіб візуалізації класів програми, їх атрибутів та методів, а також взаємозв'язків між ними.

Ці діаграми застосовуються для кількох цілей. По-перше, вони використовуються для візуалізації структури програми, що допомагає розробникам зрозуміти її складові елементи та взаємодію між ними. Також вони використовуються під час проектування програмного забезпечення для визначення архітектурних рішень та взаємодії між компонентами системи.

Діаграми класів також є частиною технічної документації програмного забезпечення, де вони надають зручний та структурований спосіб представлення інформації про структуру системи. Крім того, вони можуть бути використані для аналізу та рефакторингу коду, допомагаючи виявляти можливі проблеми та недоліки у структурі програми, а також покращувати якість та читабельність коду.

Загалом, діаграми класів є важливим інструментом для розробників програмного забезпечення, який допомагає зрозуміти та аналізувати структуру програми, а також сприяє ефективному проектуванню та розробці програмного забезпечення.

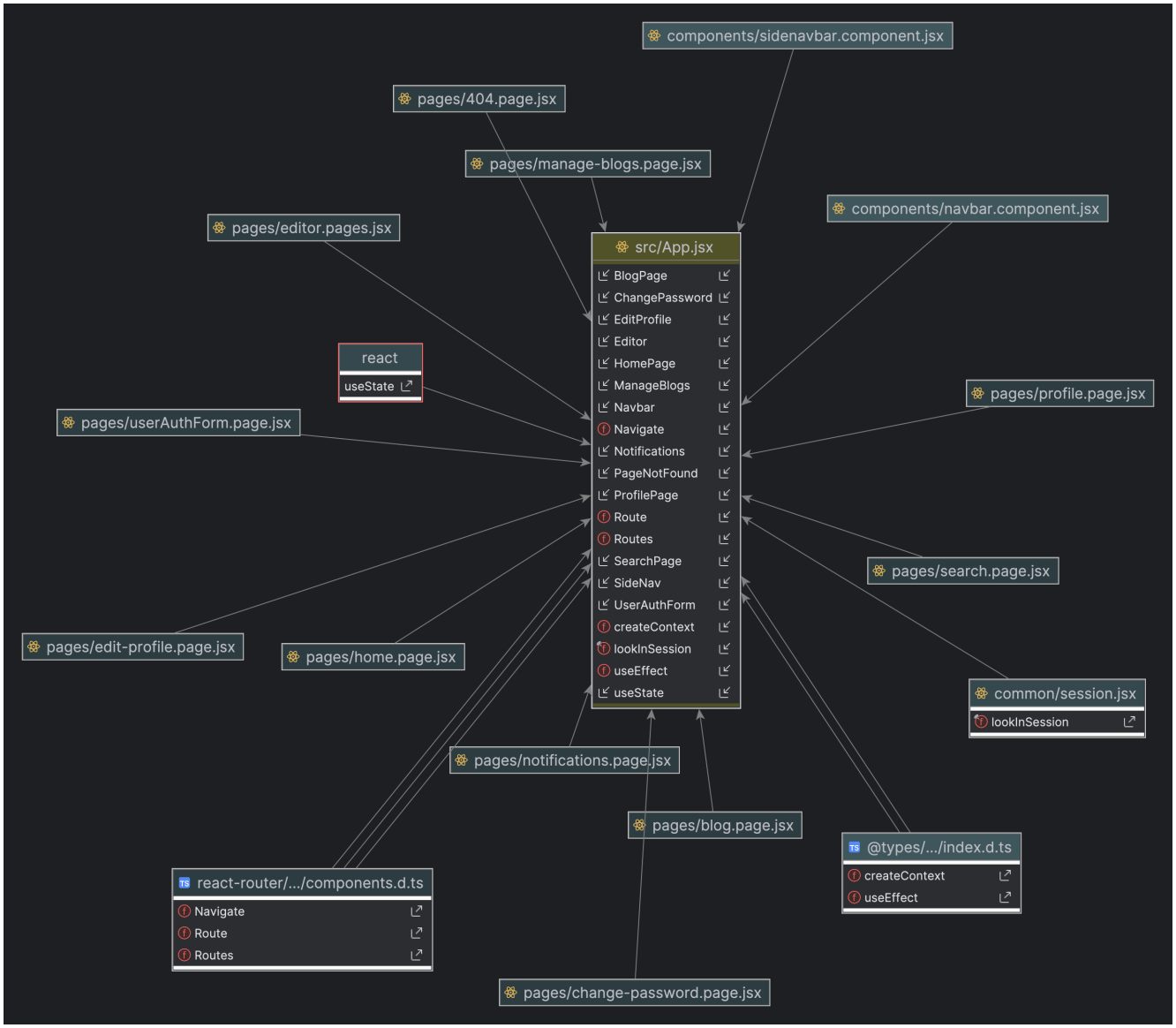

Рисунок 2.1 – Діаграма класів основної програми App

Ця структура дозволяє легко розширювати та підтримувати фронтенд додатку, оскільки різні його частини логічно розділені на компоненти та сторінки з чіткою відповідальністю. Обрана структура фронтенду має кілька ключових переваг, які забезпечують основні потреби користувача та полегшують розвиток та підтримку додатку.

Модульність грає важливу роль у структурі. Кожен компонент відповідає за певну частину функціональності, що дозволяє легко відокремлювати та перевикористовувати код. Наприклад, компонент Navbar відповідає тільки за верхню панель навігації, а компонент Editor - за редагування блогів. Це робить код більш читабельним та керованим.

Структура також дозволяє легко розширювати додаток. Компоненти та сторінки логічно розділені, що дає змогу додавати нові функції та сторінки без зміни вже існуючого коду. Наприклад, якщо з'явиться потреба в новому компоненті, такому як функція пошуку, можна створити відповідну сторінку та компоненти, і додати їх до маршрутизації.

Структура дотримується принципів зручного користувацького досвіду. Навігаційні панелі дозволяють легко переміщатися між різними частинами додатку, а форми авторизації та реєстрації надають зручний спосіб для взаємодії з додатком.

Для кращого управління станом додатку використовуються контексти UserContext та ThemeContext, що дозволяє зберігати та передавати важливу інформацію по всьому дереву компонентів без необхідності передачі через пропси. Це полегшує управління станом додатку та додає консистентність до інтерфейсу.

2.4 Розробка серверної частини блогової платформи

Встановлено Node.js та ініціалізовано проект за допомогою npm init. Завантажено необхідні пакети, такі як Express, mongoose, bcrypt, jwt за допомогою команди npm install express mongoose bcrypt jwt.

Створено файл .env для зберігання змінних середовища, таких як рядок підключення до бази даних та секретний ключ для JWT. Встановлено dotenv пакет та підключено його у основному файлі для завантаження змінних середовища.

Підключено до бази даних MongoDB за допомогою Mongoose. Використано mongoose.connect() для підключення, з використанням змінної середовища process.env.DB_LOCATION для рядка підключення. Налаштовано автоматичне створення індексів в MongoDB (див. рис. 2.2).

```
mongoose.connect(process.env.DB_LOCATION, options: {
    autoIndex: true
})
```

Рисунок 2.2 – Підключення до бази даних MongoDB за допомогою mongoose

Для зручності схеми створено в окремих файлах, для подальшої роботи з ними підключаємо наступні файли (див. рис. 2.3).

```
import User from './Schema/User.js';
import Blog from './Schema/Blog.js';
import Notification from "./Schema/Notification.js";
import Comment from "./Schema/Comment.js";
```

Рисунок 2.3 – Створення схем та моделей бази даних

Напишемо код, що реалізує аутентифікацію і авторизацію користувачів, використовуючи JWT (JSON Web Tokens) для створення та верифікації токенів доступу. Ось деякі ключові моменти:

– Реєстрація користувача (/signup): Користувачі можуть реєструватися, вводячи свої дані, такі як повне ім'я, електронна адреса та пароль. Пароль хешується перед збереженням у базу даних за допомогою bcrypt. Після успішної реєстрації користувач отримує токен доступу, який потім може використовуватися для аутентифікації.

– Вхід користувача (/signin): Користувачі можуть увійти, вводячи свою електронну адресу та пароль. Сервер перевіряє електронну адресу та порівнює хеш паролю збережений у базі даних. Якщо дані вірні, користувачеві надсилається токен доступу.

– Аутентифікація з Google (/google-auth): Користувачі можуть також увійти за допомогою автентифікації Google. Після успішної аутентифікації користувача отримується токен доступу, який можна використовувати для авторизації. В коді є POST-точка доступу /google-auth, призначена для обробки

запитів автентифікації Google. Ця точка доступу очікує access_token, отриманий з клієнтської бібліотеки Google Sign-In. Для перевірки ідентифікаційного токена, отриманого від клієнта, Google використовується метод verifyIdToken з Firebase Admin SDK для перевірки автентичності ідентифікаційного токена. Цей метод забезпечує, що токен підписаний Google та містить дійсну інформацію про користувача. Після успішної перевірки ідентифікаційного токена сервер перевіряє, чи існує користувач з вказаною адресою електронної пошти в базі даних. Якщо користувач існує, він входить у систему; в іншому випадку створюється новий обліковий запис користувача, використовуючи інформацію, надану Google. Якщо це новий користувач, сервер створює новий документ користувача в базі даних, використовуючи інформацію, отриману від Google, таку як адреса електронної пошти користувача, ім'я та зображення профілю. Якщо це існуючий користувач, він може оновити інформацію про профіль користувача, якщо це необхідно. Після успішної автентифікації сервер генерує токен JSON Web Token (JWT) за допомогою унікального ідентифікатора користувача (зазвичай ідентифікатор користувача з бази даних) та секретного ключа. Потім цей токен повертається клієнту, дозволяючи користувачеві отримати доступ до захищених ресурсів на сервері. Для подальших запитів, які потребують автентифікації, клієнт включає JWT у заголовок Authorization HTTP-запиту. Потім сервер перевіряє JWT, щоб переконатися, що запит надійшов від аутентифікованого користувача. Код функції знаходиться в додатках (див. Додаток А)

– Перевірка токенів доступу: Функція verifyJWT перевіряє токен доступу у заголовку запиту. Вона розшифровує токен за допомогою секретного ключа, перевіряє його підпис, щоб забезпечити, що токен не був змінений під час передачі, та часові межі дії, а також перевіряє інші клейми токена. Якщо всі перевірки успішні, токен вважається дійсним, і користувач має доступ до захищених ресурсів. В іншому випадку сервер повідомляє про проблему з автентифікацією. Реалізація функції зображена на рисунку (див. рис. 2.4).

```
const verifyJWT = (req, res, next) : any | undefined  => {

    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(" ")[1];

    if(token == null){
        return res.status(401).json({ error: "No access token" })
    }

    jwt.verify(token, process.env.SECRET_ACCESS_KEY, options: (err, user) : any | undefined  => {
        if(err) {
            return res.status(403).json({ error: "Access token is invalid" })
        }

        req.user = user.id
        next()
    })

}
```

Рисунок 2.4 – Реалізація функції verifyJWT

Система управління користувачами та їхніми профілями реалізована через спеціальний веб-інтерфейс або за допомогою API. Користувачі мають можливість виконувати наступні дії:

– Отримання профілів користувачів: Зареєстровані користувачі можуть отримати свій профіль, введенням своїх авторизаційних даних або іншими методами аутентифікації. Після успішної аутентифікації їм надається доступ до свого профілю з усією збереженою інформацією.

– Оновлення профілів користувачів: Користувачі можуть змінювати інформацію в своєму профілі, таку як ім'я, електронну пошту, пароль і т. д. Якщо вони бажають змінити свій аватар, вони можуть завантажити нове зображення, яке буде відображатися як їхнє представлення.

– Видалення профілів користувачів: Користувачі також мають можливість видалити свій профіль, що призведе до видалення всієї збереженої інформації про них з системи. Ця дія може бути незворотньою, тому перед видаленням профілю користувачам може бути надана можливість підтвердити свої наміри.

Реалізовано наступні операції з блогами:

– Створення блогів: Користувачі можуть створювати нові блоги за допомогою маршруту /create-blog. У цьому маршруті користувачі передають дані про блог, такі як заголовок, опис, зображення обкладинки, теги і вміст. Користувачі можуть вказати, чи це чернетка, яку можна зберегти, або публікувати блог одразу.

– Читання блогів: Користувачі можуть читати блоги за допомогою маршруту /get-blog. У цьому маршруті користувач передає ідентифікатор блогу, і сервер повертає вміст блогу. Якщо блог - це чернетка, він не буде доступний для перегляду.

– Оновлення блогів: Користувачі можуть оновлювати існуючі блоги за допомогою маршруту /create-blog, передавши нові дані блогу. У разі успішного оновлення сервер повертає ідентифікатор оновленого блогу.

– Видалення блогів: Користувачі можуть видаляти свої блоги за допомогою маршруту /delete-blog, передаючи ідентифікатор блогу для видалення. Після видалення блогу сервер повертає підтвердження видалення.

– Публікація і збереження блогів як чернеток: При створенні блогу користувач може вказати, що це чернетка, яку слід зберегти для подальшого оновлення або публікації.

– Механізм лайків блогів: Користувачі можуть виражати своє вподобання до блогів, натискаючи кнопку "лайк". Це реалізовано за допомогою маршрутів /like-blog та /isliked-by-user. Після натискання кнопки "лайк" сервер виконує необхідні дії і повертає відповідні дані.

Система сповіщень, яка реалізована для користувачів при лайках, коментуванні та відповідях на їхні дії, може мати наступний функціонал:

– Структура сповіщень: Кожне сповіщення містить інформацію про те, хто здійснив дію (наприклад, хто поставив лайк або написав коментар), тип дії (лайк, коментар, відповідь), на який об'єкт дії це сталося (наприклад, блог), а також, можливо, вміст цієї дії (текст коментаря або відповіді).

– Генерація сповіщень: Сповіщення можуть бути генеровані при кожній відповідній дії. Наприклад, якщо користувач поставив лайк або написав коментар,

система може автоматично створити сповіщення для власника контенту, на який була відповідна дія.

– Пов'язані з об'єктами дій: Система сповіщень може бути пов'язана з об'єктами дій, такими як блоги або коментарі. Це означає, що користувач може отримати сповіщення про дії, що стосуються конкретних об'єктів, на які вони підписані або які вони створили.

– Доставка сповіщень: Сповіщення можуть бути доставлені користувачам різними способами, такими як електронна пошта, сповіщення у додатку або повідомлення в браузері. Користувачі також можуть мати можливість вибирати, які типи сповіщень вони бажають отримувати і через які канали.

– Призначення пріоритетів: Сповіщення можуть мати різні рівні пріоритету в залежності від типу дії або контексту. Наприклад, сповіщення про коментар може мати вищий пріоритет, ніж сповіщення про лайк.

– Можливість взаємодії зі сповіщеннями: Користувачі можуть мати можливість взаємодіяти зі сповіщеннями, наприклад, переглядати контент, на який вони вказують, або відповідати на коментарі безпосередньо зі сповіщення.

– Система оповіщень: Для ефективного управління сповіщеннями може бути створена система оповіщень, яка дозволяє користувачам переглядати, відмічати як прочитані або видаляти сповіщення, а також налаштовувати свої уподобання щодо отримання сповіщень.

2.5 Тестування блогової платформи

Перший крок у ручному тестуванні сайту полягає у створені нового користувача. Заповнено всі необхідні поля, такі як ім'я, електронна пошта та пароль. Після цього натиснули кнопку "Зареєструватися". Система успішно створила нового користувача, про що було повідомлено відповідним сповіщенням на екрані.

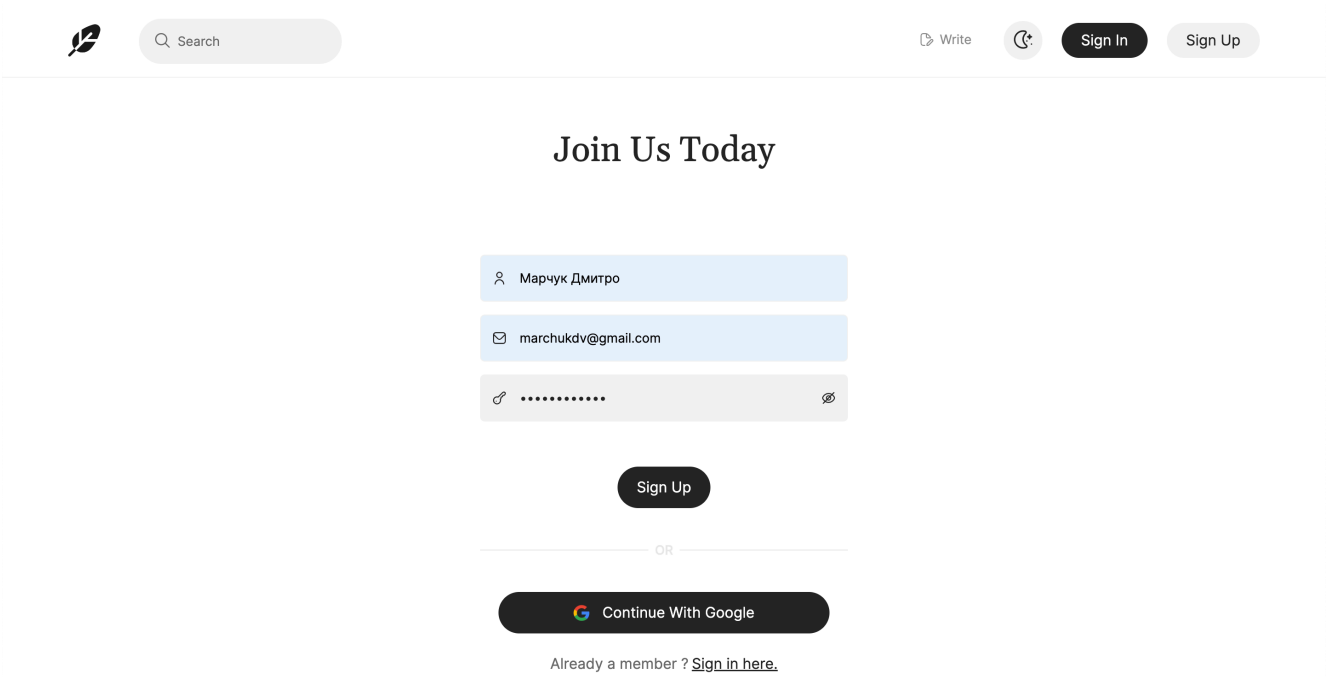

Рисунок 2.5 – Реєстрація нового користувача

Після створення користувача переходимо на сторінку входу і вводимо новостворені облікові дані (електронну пошту та пароль). Натиснувши кнопку "Увійти", успішно входимо в особистий профіль. Відображення даних профілю та навігація по сайту пройшли без помилок.

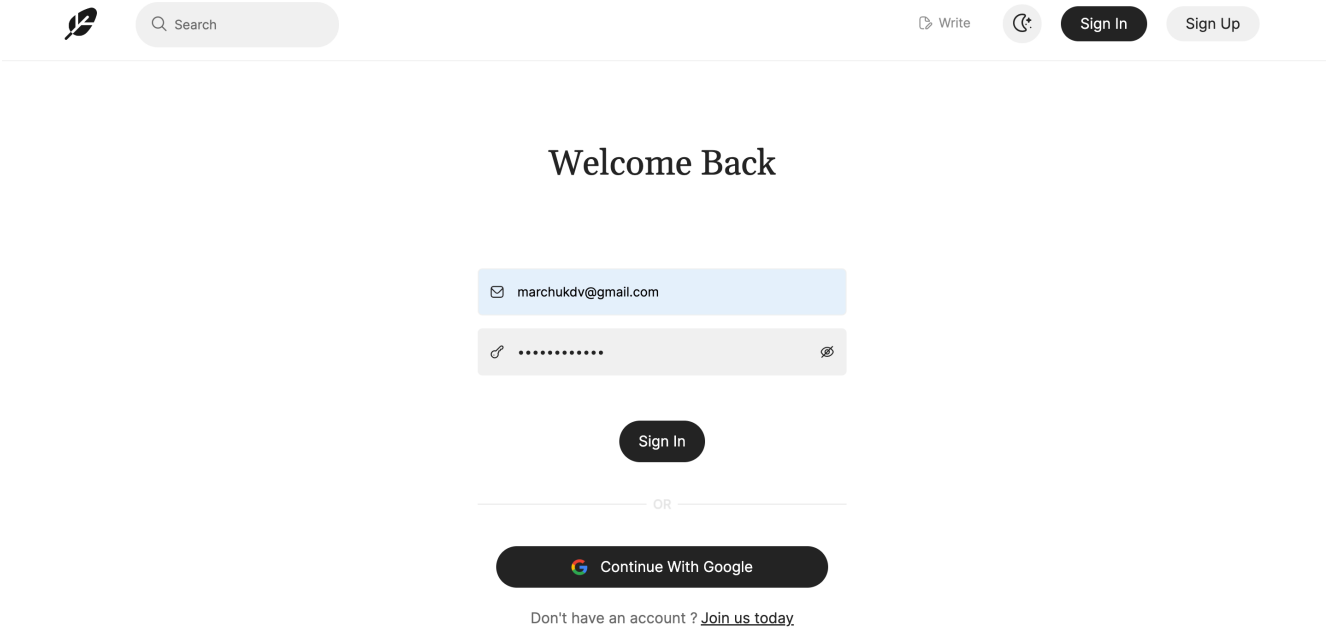

Рисунок 2.6 – Вхід у систему

Зовнішній вигляд головної сторінки зображено на рисунку (див. рис. 2.7).

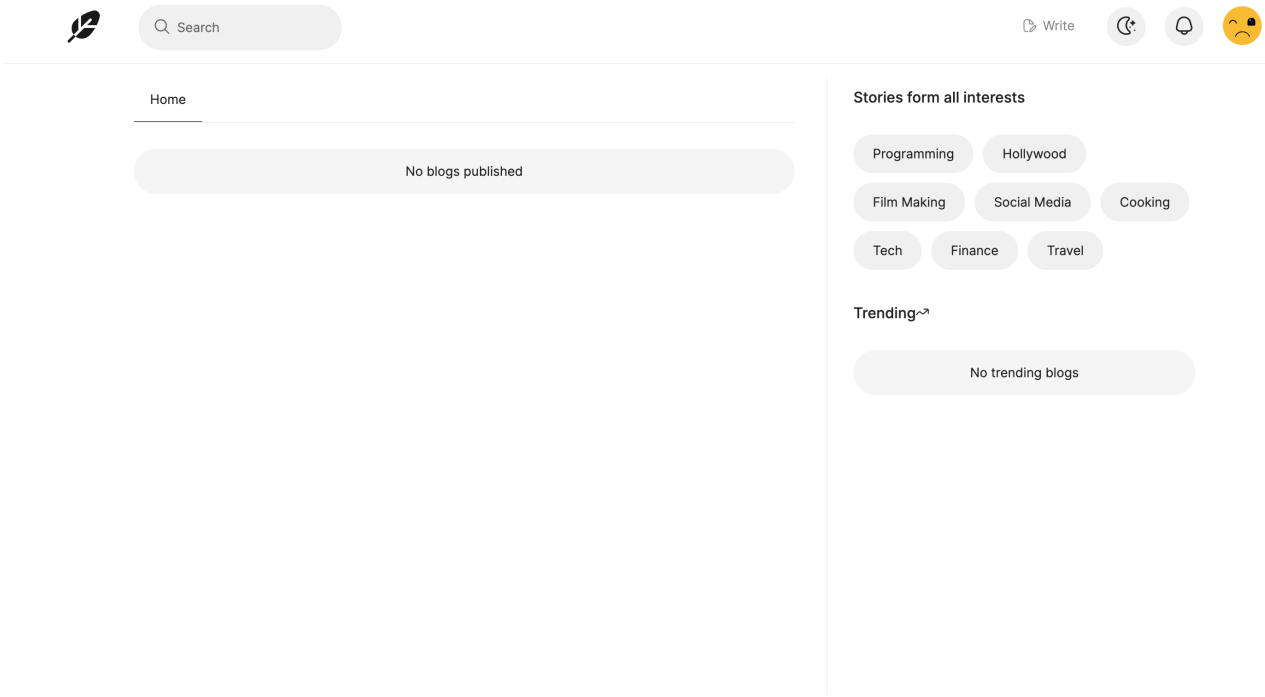

Рисунок 2.7 – Головна сторінка

Сторінку редагування профілю зображено на рисунку (див. рис. 2.8).

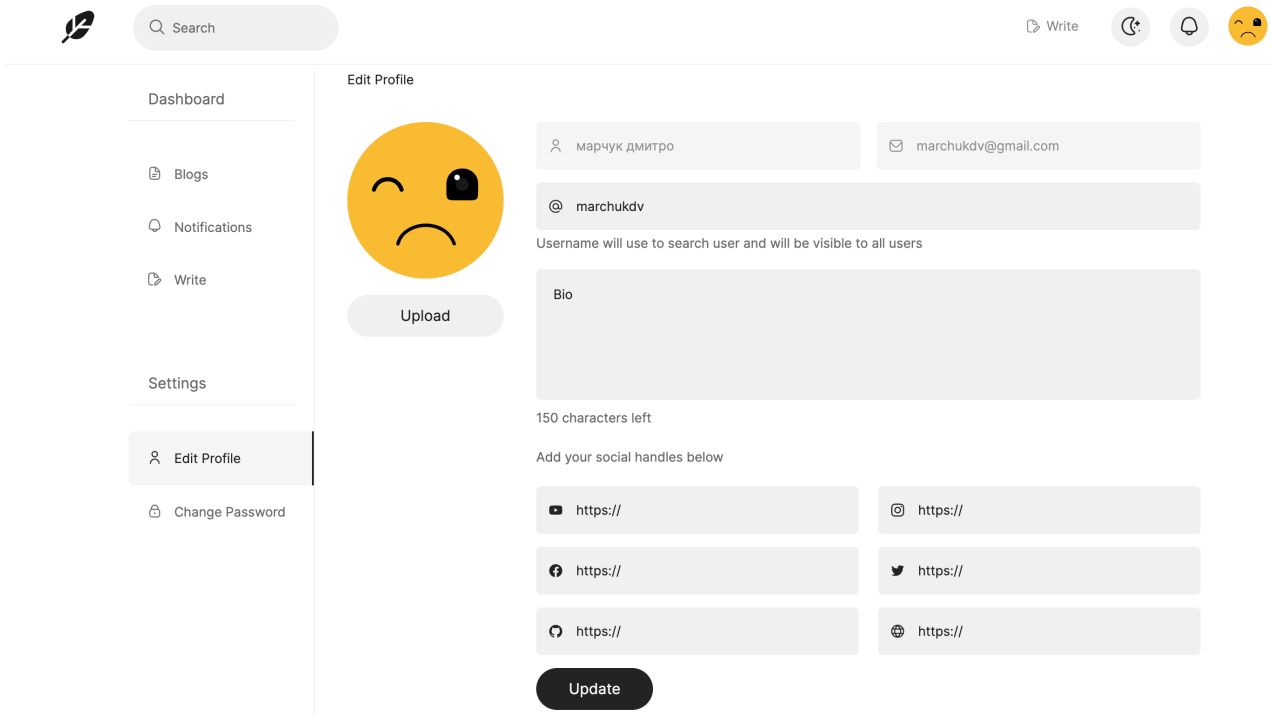

Рисунок 2.8 – Редагування профілю

Після входу в систему переходимо до розділу створення блогу. У відповідних полях заповнено заголовок блогу, текст статті та додали зображення (див. рис. 2.9).

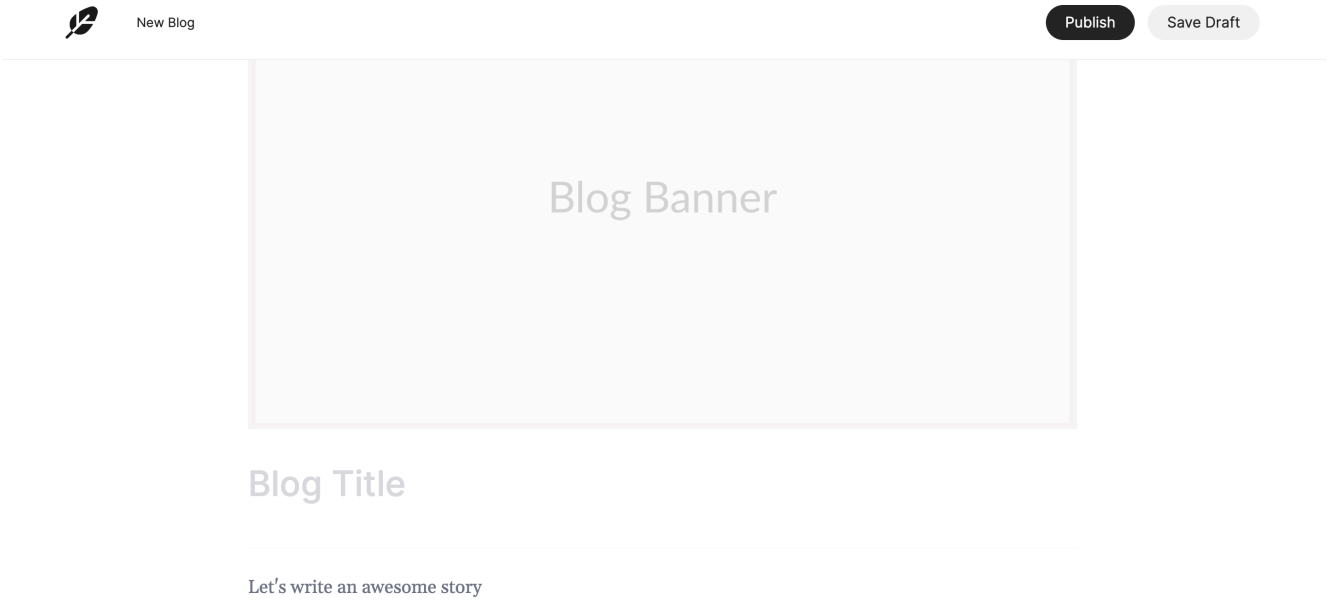

Рисунок 2.9 – Процес публікації блогу. Текстовий редактор

Рисунок 2.10 – Процес публікації блогу. Заповнення редактору текстом та банером

Після заповнення всіх необхідних полів натиснули кнопку "Опублікувати".

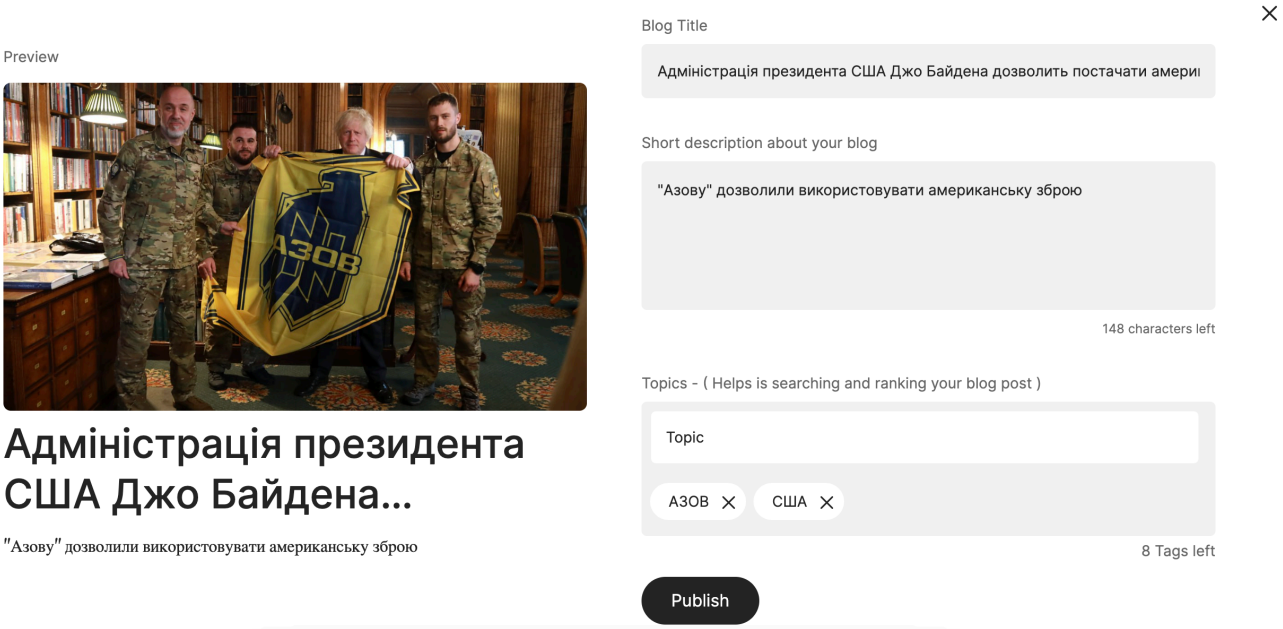

Рисунок 2.11 – Процес публікації блогу. Попередній перегляд блогу

Блог був успішно опублікований і став доступний для інших користувачів.

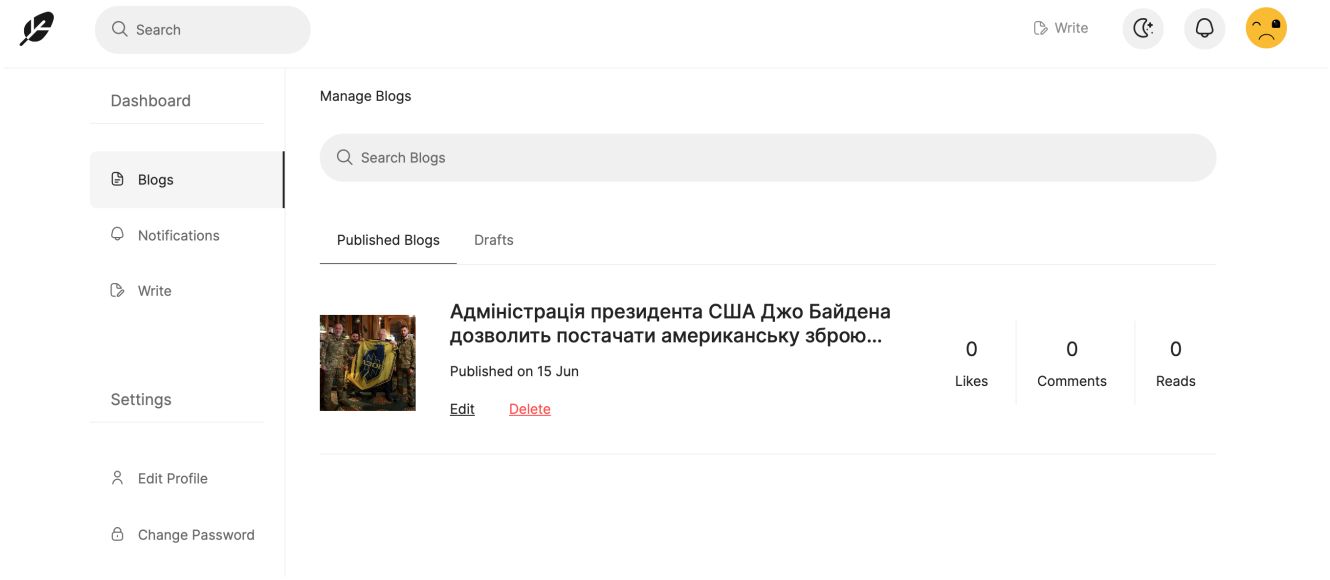

Рисунок 2.12 – Опублікований блог в стрічці блогів

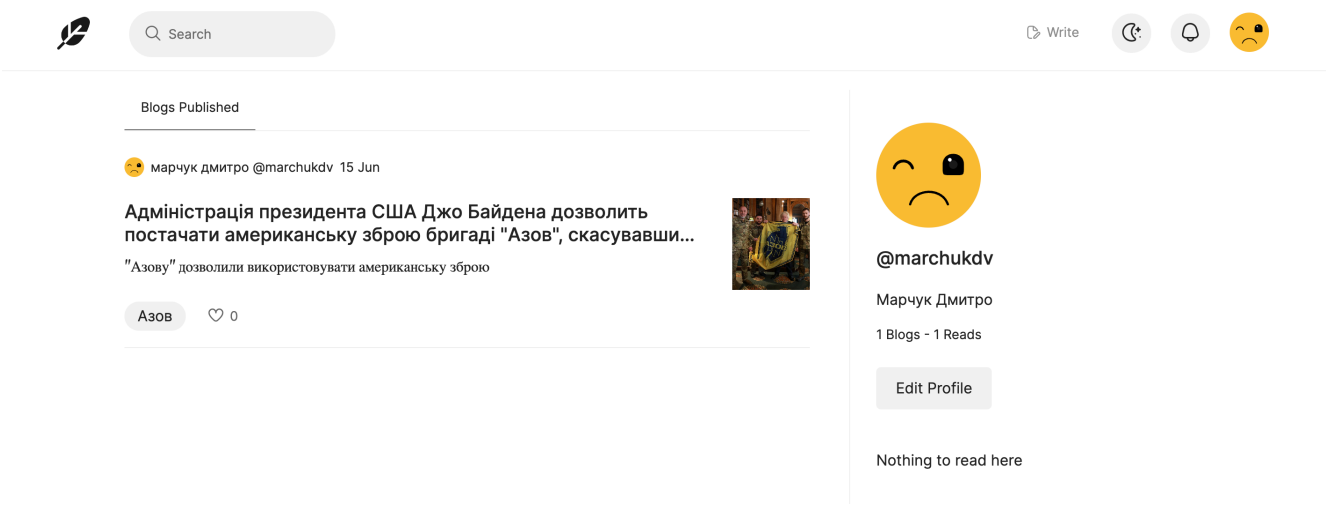

Рисунок 2.13 – Опублікований блог в профілі

Інші користувачі мають змогу переглядати блог, коментувати його та ставити лайки. Перевірено функціональність коментарів, залишивши декілька тестових повідомлень під блогом від різних облікових записів. Також перевірено, чи працюють лайки: кілька користувачів поставили лайки блогу, і кількість лайків правильно відображалася.

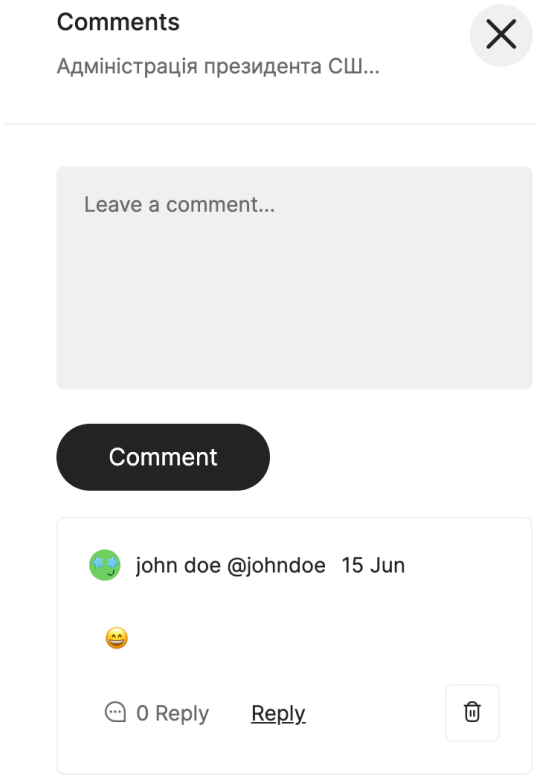

Рисунок 2.14 – Коментування блогу

Система сповіщень функціонувала відповідно до очікувань. Кожне нове сповіщення про коментар або лайк миттєво відображалося у верхньому правому куті сайту. Це забезпечує зручність для користувачів, дозволяючи їм оперативно реагувати на активність навколо їхнього контенту. Сповіщення містили детальну інформацію про подію, що дозволяло швидко орієнтуватися у взаємодії з іншими користувачами.

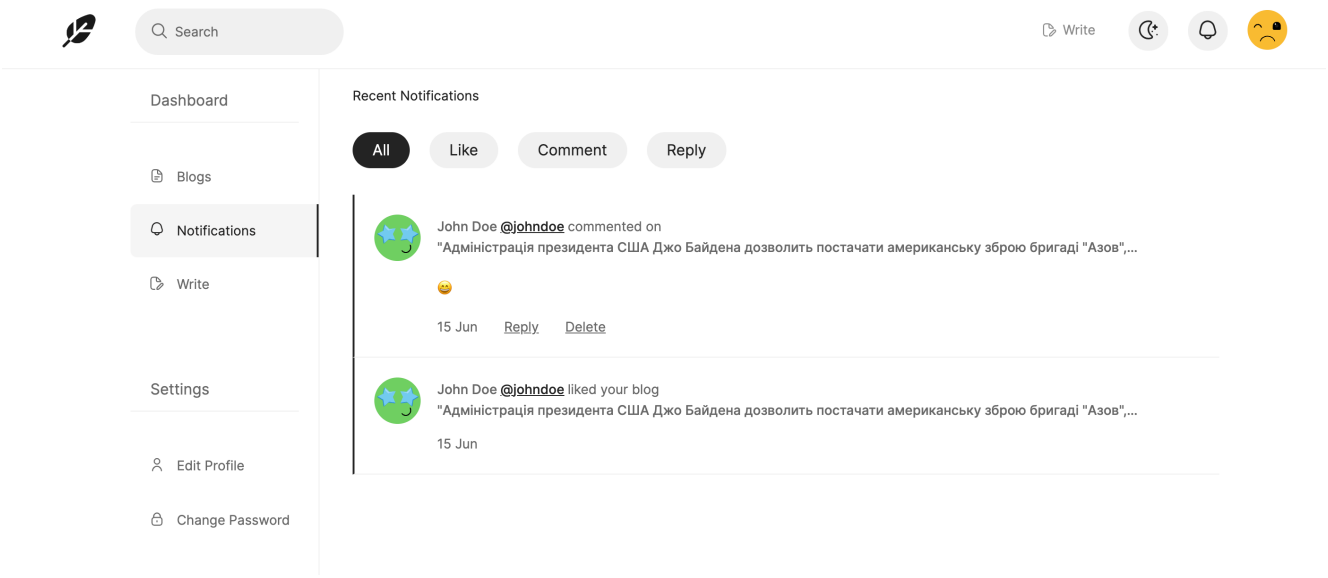

Рисунок 2.15 – Перевірка роботи сповіщень. Повідомлення про лайк на пості та новий коментар

Усі перевірені функції працювали коректно і відповідали очікуванням. Це свідчить про високу якість розробки та тестування системи. З точки зору користувача, сайт надає зручний та інтуїтивно зрозумілий інтерфейс, а також забезпечує безперебійну роботу основних функцій. У разі подальшого розвитку або впровадження нових функцій, важливо підтримувати цей високий рівень якості, здійснюючи регулярне тестування та валідацію нових елементів системи.

Таким чином, ручне тестування показало, що сайт готовий до активного використання, і користувачі можуть без проблем реєструватися, створювати та публікувати контент, а також взаємодіяти з іншими користувачами через коментарі та лайки.

Метою автоматизованого тестування було перевірити коректність та стабільність функції входу в систему за допомогою логіна та пароля. Для цього ми використано REST API, що дозволило автоматизувати процес і забезпечити швидку та ефективну перевірку [9].

Встановлено необхідні бібліотеки та інструменти для роботи з REST API Налаштувано тестове середовище з актуальними обліковими даними для тестування. Написано скрипт для виконання HTTP-запитів на сервер з використанням методу POST для надсилання логіна та пароля. Реалізовано обробки відповідей сервера, включаючи перевірку статус-кодів та валідацію отриманих даних. Автоматичне виконання скриптів для різних наборів тестових даних, включаючи як валідні, так і невалідні логін і пароль. Під час тестування було проведено декілька ітерацій із різними наборами даних. Усі тести були виконані успішно, що підтвердило коректну роботу функції входу за логіном і паролем. Основні результати включають:

– Коректність обробки валідних даних: Вхід в систему за допомогою правильного логіна та пароля був успішним у всіх випадках. Сервер повертав відповідний статус-код (200 OK) та валідний токен автентифікації.

– Обробка невалідних даних: Система коректно обробляла спроби входу з неправильними обліковими даними, повертаючи відповідні помилки (наприклад, 401 Unauthorized). Усі повідомлення про помилки були інформативними та відповідали очікуванням.

– Перевірка безпеки: Тести показали, що система не допускає вхід при використанні неправильного логіна чи пароля, забезпечуючи захист користувацьких даних. Всі запити проходили через захищене з'єднання (HTTPS), що гарантує безпечну передачу даних.

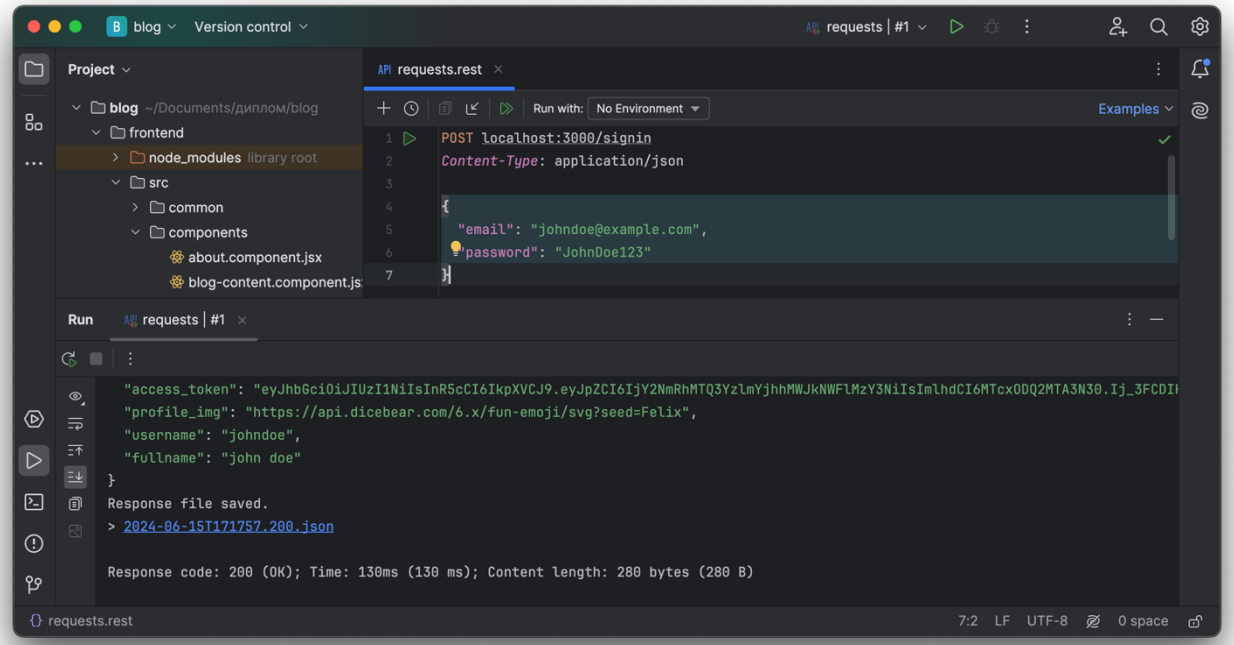

Рисунок 2.16 – Автоматизоване тестування. Перевірка входу у систему

Автоматизоване тестування входу за логіном і паролем за допомогою REST API підтвердило стабільну та коректну роботу цієї функції. Успішне проходження всіх тестів демонструє, що система здатна надійно автентифікувати користувачів, забезпечуючи як зручність, так і безпеку.

Цей результат свідчить про високий рівень готовності системи до активного використання користувачами. Використання автоматизованого тестування значно прискорило процес перевірки та дозволило охопити ширший спектр тестових сценаріїв.

3 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

3.1 Проведення рятувальних та інших невідкладних робіт

Рятувальні та інші невідкладні роботи включають дії, спрямовані на рятування людей і надання допомоги потерпілим, ліквідацію та локалізацію аварій, а також створення умов для подальшого відновлення виробничої діяльності об’єкта. Основні завдання рятувальних та інших невідкладних робіт полягають у забезпеченні безпеки постраждалих, ліквідації наслідків надзвичайних ситуацій та мінімізації матеріальних втрат.

При проведенні рятувальних робіт необхідно здійснювати низку важливих заходів, спрямованих на ліквідацію наслідків надзвичайних ситуацій та забезпечення безпеки. Основні заходи включають розвідування районів, локалізацію та гасіння пожеж, а також знешкодження вибухонебезпечних предметів.

Розвідування районів є першим і надзвичайно важливим етапом рятувальних робіт. Воно передбачає проведення ретельної розвідки зон, ділянок та об’єктів, де необхідно ліквідувати наслідки надзвичайних ситуацій. Це включає визначення та локалізацію зони надзвичайної ситуації для забезпечення ефективності подальших дій. Окрім того, важливо виявити та позначити райони, де є радіоактивне, хімічне або біологічне зараження, щоб уникнути додаткових ризиків для рятувальників та постраждалих.

Локалізація та гасіння пожеж є наступним етапом, який передбачає створення проїздів у завалах і на зараженій території для забезпечення доступу рятувальних команд. Важливим аспектом є локалізація та ліквідація аварій на комунально-енергетичних та технологічних мережах, що може включати відновлення порушених ліній зв’язку, які є критично важливими для координації рятувальних робіт. Також необхідно укріпити або, навпаки, руйнувати нестійкі конструкції, які можуть становити загрозу для безпеки рятувальників та перешкоджати проведенню рятувальних робіт.

Знешкодження вибухонебезпечних предметів є окремим важливим завданням, яке включає виявлення, знешкодження та знищення знайдених боєприпасів та інших вибухонебезпечних предметів. Це особливо актуально в зонах конфліктів або в районах, де можливі терористичні акти. Ретельна робота з вибухонебезпечними предметами допомагає уникнути додаткових вибухів та травм серед рятувальників та постраждалих.

Успішність проведення рятувальних робіт оцінюється за кількістю врятованих людей та збережених матеріальних цінностей. Для досягнення високих показників у цьому питанні необхідно враховувати кілька важливих факторів.

Перш за все, важлива завчасна підготовка сил і засобів для проведення рятувально-навігаційних робіт. Це включає не лише наявність необхідного обладнання, але й навчання та тренування рятувальників, підготовку техніки та матеріалів, які можуть знадобитися в різних надзвичайних ситуаціях.

Друге, що має значення, це завчасне планування та створення угрупувань сил і засобів. Це передбачає розміщення цих сил і засобів на місцевості відповідно до задуму керівника рятувальних робіт. Важливо, щоб це розміщення забезпечувало послідовне та ефективне використання ресурсів, мінімізуючи час на їх переміщення та максимізуючи оперативність реагування на зміну ситуації.

Організація і ведення розвідки району надзвичайної ситуації є наступним важливим елементом успіху. Це дозволяє отримати точну інформацію про стан району, масштаби ураження, наявність перешкод і потенційних небезпек, що допомагає правильно планувати і координувати рятувальні роботи.

Швидке висування рятувальних формувань в осередок ураження також є критично важливим. Це включає рішуче рятування людей, яке передбачає подавання повітря в завалені захисні споруди в перші 3-4 години після аварії, що є життєво необхідним для виживання постраждалих. Надання першої медичної допомоги ураженим у перші 12-14 годин також є надзвичайно важливим для збереження життя і здоров’я людей. Основні рятувальні роботи мають бути завершені за першу добу, що дозволить мінімізувати ризики для постраждалих і забезпечити ефективну допомогу.

Безперервне ведення рятувальних робіт до їх повного завершення є наступним критично важливим аспектом. Це означає, що роботи повинні вестися позмінно, щоб забезпечити постійну активність та зниження втоми серед рятувальників. Мінімальна тривалість роботи зміни становить 2-4 години, що дозволяє рятувальникам зберігати високу продуктивність та ефективність протягом усього періоду проведення рятувальних робіт.

Для безпосередньої організації та координування аварійно-рятувальних та інших невідкладних робіт створюється штаб з ліквідації наслідків НС. Керівник робіт з ліквідації наслідків НС особисто відповідає за управління всіма заходами з ліквідації наслідків НС.

Проведення рятувальних та інших невідкладних робіт є складним та відповідальним процесом, який вимагає чіткої організації, своєчасного реагування та ефективної координації всіх залучених сил і засобів. Успіх цих робіт значною мірою залежить від готовності рятувальних підрозділів, оперативності їх дій та професіоналізму фахівців [15].

3.2 Значення автоматизації виробничих процесів в питаннях охорони праці

Автоматизація виробничих процесів відіграє ключову роль у підвищенні безпеки на робочих місцях. Завдяки впровадженню автоматизованих систем керування, механізації та комплексної автоматизації, виробництво стає безпечнішим і більш ефективним. Сучасні технології дозволяють мінімізувати людський фактор, який часто є джерелом помилок та нещасних випадків на виробництві.

Автоматизовані системи виконують важкі та небезпечні фізичні роботи, що знижує ризик травм та професійних захворювань серед працівників. Зокрема, використання роботизованих маніпуляторів та автоматичних конвеєрів дозволяє уникнути підняття важких вантажів і тривалої роботи в незручних положеннях.

Автоматичні системи здатні виконувати завдання з високою точністю, що мінімізує можливість помилок і аварій. Наприклад, сучасні системи автоматичного контролю та регулювання технологічних процесів можуть своєчасно виявляти відхилення від норми і коригувати їх, запобігаючи аварійним ситуаціям.

Автоматизація знижує необхідність виконання повторюваних та монотонних завдань, що позитивно впливає на психологічний стан працівників. Це допомагає уникнути емоційного вигорання та підвищує загальний рівень задоволеності працею.

Автоматизація дозволяє створювати більш безпечні та комфортні умови праці. Наприклад, автоматичні системи можуть контролювати мікроклімат на робочих місцях, забезпечуючи оптимальну температуру та вологість, що сприяє збереженню здоров'я працівників.

Автоматизація також сприяє зниженню рівня травматизму та професійних захворювань, підвищуючи загальну безпеку на виробництві. У впровадженні автоматизованих систем важливо враховувати аспекти ергономіки та зручності робочих місць, щоб забезпечити максимальний комфорт і безпеку для операторів.

Автоматизація процесів є важливою не лише у виробничих середовищах, але й у сфері розробки програмного забезпечення. Наприклад, у проекті розробки веб-платформи для створення та введення блогів з використанням технологій React, Node.js та MongoDB автоматизація може значно покращити ефективність і якість кінцевого продукту.

Використання автоматизованих інструментів для розгортання інфраструктури дозволяє швидко і безпомилково налаштовувати середовище розробки. Інструменти, такі як Docker та Kubernetes, спрощують процес створення, тестування та впровадження веб-додатків, забезпечуючи стабільність та надійність системи.

Інструменти автоматичного тестування, такі як Jest для React та Mocha для Node.js, дозволяють автоматизувати процеси тестування коду, забезпечуючи високу якість програмного забезпечення. Це мінімізує ризик введення багів та покращує загальну стабільність платформи.

Використання систем контролю версій, таких як Git, дозволяє автоматизувати процеси управління змінами, що сприяє ефективній координації між розробниками. Автоматизація процесів злиття та розгортання змін значно скорочує час розробки та впровадження нових функцій.

Автоматизовані системи моніторингу, такі як Prometheus та Grafana, забезпечують своєчасне виявлення та вирішення проблем на платформі. Це підвищує її надійність та безперебійність роботи, дозволяючи розробникам зосередитися на створенні нових функцій замість вирішення непередбачених проблем.

Таким чином, автоматизація в розробці веб-платформ, як і в промисловому виробництві, підвищує продуктивність, якість і безпеку, створюючи більш комфортні умови для роботи та мінімізуючи ризики, пов'язані з людськими помилками [18].

ВИСНОВКИ

Розробка веб-платформи для створення та введення блогів з використанням React, Node.js та MongoDB була успішно завершена. В процесі роботи над проєктом було досягнуто всіх поставлених цілей.

Було обґрунтовано актуальність створення блог-платформи в сучасному цифровому світі, де створення та публікація контенту є невід'ємною частиною комунікації та самовираження.

Детально описано функціональні вимоги до системи, зокрема можливість реєстрації та автентифікації користувачів, створення, редагування та видалення блог-постів, коментування та лайкання постів. Нефункціональні вимоги включали забезпечення безпеки, високу продуктивність, масштабованість та зручність у користуванні.

Розробка веб-платформи для створення та введення блогів з використанням React, Node.js та MongoDB була успішно завершена. В процесі роботи над проєктом було досягнуто всіх поставлених цілей.

Було обґрунтовано актуальність створення блог-платформи в сучасному цифровому світі, де створення та публікація контенту є невід'ємною частиною комунікації та самовираження.

Детально описано функціональні вимоги до системи, зокрема можливість реєстрації та автентифікації користувачів, створення, редагування та видалення блог-постів, коментування та лайкання постів. Нефункціональні вимоги включали забезпечення безпеки, високу продуктивність, масштабованість та зручність у користуванні. Користувач може створити новий блог, заповнивши необхідну інформацію, таку як назва блогу, опис та вибравши категорію. Після створення блогу користувач стає його власником і може публікувати в ньому пости. Автор блогу може написати новий пост, вказавши заголовок, текст, додавши зображення або відео. Після завершення написання посту, автор може зберегти його як чернетку або опублікувати для читачів. Автор має можливість редагувати свої

попередні пости. Це включає зміну тексту, додавання або видалення зображень, зміну заголовка та інших елементів посту. Після редагування пост може бути збережений або опублікований знову. Зареєстровані користувачі можуть залишати коментарі під постами. Це включає написання тексту коментаря та, при необхідності, редагування або видалення власних коментарів. Коментарі можуть бути лайкані іншими користувачами, що сприяє взаємодії та обговоренню. Користувачі можуть переглядати та редагувати свої профілі, включаючи зміну аватару, імені, електронної пошти та інших персональних даних. Також вони можуть змінювати налаштування конфіденційності та повідомлень.

Для реалізації проєкту було обрано стек технологій MERN (MongoDB, Express.js, React, Node.js). MongoDB використовувалась як NoSQL база даних для зберігання інформації про користувачів, блоги, пости та коментарі. Її гнучкість та масштабованість дозволили ефективно управляти даними. Express.js служив як серверний фреймворк для Node.js, що забезпечив побудову RESTful API для взаємодії з базою даних та клієнтським інтерфейсом. Express.js забезпечив легку конфігурацію та управління маршрутами. React використовувався для створення динамічного та інтерактивного фронтенду. React забезпечив компонентний підхід до розробки інтерфейсу, що сприяло повторному використанню коду та покращенню продуктивності. Node.js став основою для серверної частини, дозволивши обробку асинхронних запитів та забезпечивши високу продуктивність серверних процесів. Node.js сприяв швидкій розробці та підтримці коду.

Система пройшла ретельне тестування, включаючи ручні тести користувацького інтерфейсу та автоматизовані тести REST API, що дозволило виявити та усунути можливі помилки та забезпечити стабільну роботу платформи.

В результаті виконаної роботи було створено повнофункціональну веб-платформу для блогів, яка відповідає всім визначеним вимогам. Отримані знання та досвід під час розробки цього проєкту стануть корисними у майбутній професійній діяльності, зокрема у сфері розробки веб-додатків та використанні сучасних технологій і методологій розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Medium [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/.

2. Blogger [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Blogger.

3. Amazon AWS Services [Електронний ресурс] – Режим доступу до ресурсу: https://www.softwareone.com/uk-ua/cloud-services/aws.

4. React Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://uk.legacy.reactjs.org/tutorial/tutorial.html.

5. Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Node.js.

6. MongoDB Tutorial [Електронний ресурс] – Режим доступу до ресурсу: https://devzone.org.ua/post/osnovy-mongodb.

7. Методологія Agile [Електронний ресурс] – Режим доступу до ресурсу: https://brainrain.com.ua/uk/chto-takoe-agile-ua/.

8. Rest API [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/shcho-take-rest-api/.

9. Смілянський, В. В. Основи роботи з базою даних MongoDB. Київ: Видавництво «Техніка», 2019. 320 с.

10. Гуменюк, І. В. REST API: розробка та впровадження. Київ: Видавництво «Либідь», 2018. 180 с.

11. Сидоренко, Н. П. Вступ до React: практичні приклади. Одеса: Видавництво «Маяк», 2020. 290 с.

12. Тарасенко, С. В. Використання Express для створення веб-додатків. Дніпро: Видавництво «Світ», 2019. 270 с.

13. Данилюк, В. В. Програмування на Node.js: повний курс. Харків: Видавництво «Ранок», 2018. 400 с.

14. Паламарчук, О. В. Основи роботи з Amazon Web Services. Львів: Видавництво «Світ знань», 2019. 310 с.

15. Безпека життєдіяльності та цивільний захист / О. Г.Левченко, О. В. Землянська, Н. А. Праховнік, В. В. Зацарний. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 267 с.

16. Гандзюк М.П., Желібо Є.П., Халімовський М.О. Основи охорони праці: Підручник. 5-е вид. / За ред. М.П. Гандзюка. – К.: Каравела, 2011. – 384 с.

17. Іванова О. В. Конспект лекцій з дисципліни “Безпека життєдіяльності та основи охорони праці” / О. В. Іванова, Н. М. Ювченко. – Одеса, 2018. – 156 с.

18. Навчальний посібник «Безпека життєдіяльності» / Т. Є.Стиценко, Г. В. Пронюк, Н. М. Сердюк, І. І. Хондак. – Харків, 2018. – 200 с.

19. Сєріков Я. О. Безпека життєдіяльності та охорона праці / Я. О. Сєріков, Л. Ф. Коженевські, М. В. Хворост. – Харків, 2021. – 255 с.

20. Методичні вказівки до виконання атестаційної роботи магістра спеціальності 121 – Інженерія програмного забезпечення (Освітньо-професійна програма - «Програмне забезпечення систем», Освітньо-наукова програма - «Інженерія програмного забезпечення») для студентів усіх форм навчання / Упор.: М.Р. Петрик, Д.М. Михалик, О.Ю. Петрик, Г.Б. Цуприк - Тернопіль: ТНТУ, 2020 – 51с.

ДОДАТКИ

Додаток А Лістинг серверної частини програми

```
import express from 'express';
import mongoose from 'mongoose';
import 'dotenv/config'
import bcrypt from 'bcrypt';
import { nanoid } from 'nanoid';
import jwt from 'jsonwebtoken';
import cors from 'cors';
import admin from "firebase-admin";
import serviceAccountKey from "./blog-9757f-firebase-adminsdk-upoty-
9bd477cdd1.json" assert { type: "json" }
import { getAuth } from "firebase-admin/auth";
import aws from "aws-sdk";

import User from './Schema/User.js';
import Blog from './Schema/Blog.js';
import Notification from "./Schema/Notification.js";
import Comment from "./Schema/Comment.js";

const server = express();
let PORT = 3000;

admin.initializeApp({
    credential: admin.credential.cert(serviceAccountKey)
})

let emailRegex = /^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/; // regex
for email
let passwordRegex = /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,20}$/; // regex for
password

server.use(express.json());
server.use(cors())

mongoose.connect(process.env.DB_LOCATION, {
    autoIndex: true
})

// setting up s3 bucket
const s3 = new aws.S3({
    region: process.env.AWS_BUCKET_REGION,
    accessKeyId: process.env.AWS_ACCESS_KEY,
    secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY
})

const generateUploadURL = async () => {

    const date = new Date();
    const imageName = `${nanoid()}-${date.getTime()}.jpeg`;

    return await s3.getSignedUrlPromise('putObject', {
        Bucket: process.env.AWS_BUCKET_NAME,
        Key: imageName,
        Expires: 1000,
        ContentType: "image/jpeg"
```

```
    })

}

const verifyJWT = (req, res, next) => {

    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(" ")[1];

    if(token == null){
        return res.status(401).json({ error: "No access token" })
    }

    jwt.verify(token, process.env.SECRET_ACCESS_KEY, (err, user) => {
        if(err) {
            return res.status(403).json({ error: "Access token is invalid"
})
        }

        req.user = user.id
        next()
    })

}

const formatDatatoSend = (user) => {

    const access_token = jwt.sign({ id: user._id },
process.env.SECRET_ACCESS_KEY)

    return {
        access_token,
        profile_img: user.personal_info.profile_img,
        username: user.personal_info.username,
        fullname: user.personal_info.fullname
    }
}

const generateUsername = async (email) => {
    let username = email.split("@")[0];

    let isUsernameNotUnique = await User.exists({ "personal_info.username":
username }).then((result) => result)

    isUsernameNotUnique ? username += nanoid().substring(0, 5) : "";

    return username;

}

// upload image url route
server.get('/get-upload-url', (req, res) => {
    generateUploadURL().then(url => res.status(200).json({ uploadURL: url
}))
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ error: err.message })
    })
```

```
})

server.post("/signup", (req, res) => {

    let { fullname, email, password } = req.body;

   // validating the data from frontend
   if(fullname.length < 3){
        return res.status(403).json({ "error": "Fullname must be at least 3
letters long" })
   }
   if(!email.length){
        return res.status(403).json({ "error": "Enter Email" })
   }
   if(!emailRegex.test(email)){
        return res.status(403).json({ "error": "Email is invalid" })
   }
   if(!passwordRegex.test(password)){
        return res.status(403).json({ "error": "Password should be 6 to 20
characters long with a numeric, 1 lowercase and 1 uppercase letters" })
   }

   bcrypt.hash(password, 10, async (err, hashed_password) => {

        let username = await generateUsername(email);

        let user = new User({
            personal_info: { fullname, email, password: hashed_password,
username }
        })

        user.save().then((u) => {

            return res.status(200).json(formatDatatoSend(u))

        })
        .catch(err => {

            if(err.code == 11000) {
                return res.status(500).json({ "error": "Email already
exists" })
            }

            return res.status(500).json({ "error": err.message })
        })

   })

})

server.post("/signin", (req, res) => {

    let { email, password } = req.body;

    User.findOne({ "personal_info.email": email })
    .then((user) => {
        if(!user){
            return res.status(403).json({ "error": "Email not found" });
```

```
        }


        if(!user.google_auth){

            bcrypt.compare(password, user.personal_info.password, (err,
result) => {

                if(err) {
                    return res.status(403).json({ "error": "Error occured
while login please try again" });
                }

                if(!result){
                    return res.status(403).json({ "error": "Incorrect
password" })
                } else{
                    return res.status(200).json(formatDatatoSend(user))
                }

            })

        } else {
            return res.status(403).json({ "error": "Account was created
using google. Try logging in with google." })
        }

    })
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ "error": err.message })
    })

})

server.post("/google-auth", async (req, res) => {

    let { access_token } = req.body;

    getAuth()
    .verifyIdToken(access_token)
    .then(async (decodedUser) => {

        let { email, name, picture } = decodedUser;

        picture = picture.replace("s96-c", "s384-c");

        let user = await User.findOne({"personal_info.email":
email}).select("personal_info.fullname personal_info.username
personal_info.profile_img google_auth").then((u) => {
            return u || null
        })
        .catch(err => {
            return res.status(500).json({ "error": err.message })
        })

        if(user) { // login
            if(!user.google_auth){
```

```
                return res.status(403).json({ "error": "This email was
signed up without google. Please log in with password to access the
account" })
            }
        }
        else { // sign up

            let username = await generateUsername(email);

            user = new User({
                personal_info: { fullname: name, email, username },
                google_auth: true
            })

            await user.save().then((u) => {
                user = u;
            })
            .catch(err => {
                return res.status(500).json({ "error": err.message })
            })

        }

        return res.status(200).json(formatDatatoSend(user))

    })
    .catch(err => {
        return res.status(500).json({ "error": "Failed to authenticate you
with google. Try with some other google account" })
    })

})

server.post("/change-password", verifyJWT, (req, res) => {

    let { currentPassword, newPassword } = req.body;

    if(!passwordRegex.test(currentPassword) ||
!passwordRegex.test(newPassword)){
        return res.status(403).json({ error: "Password should be 6 to 20
characters long with a numeric, 1 lowercase and 1 uppercase letters" })
    }

    User.findOne({ _id: req.user })
    .then((user) => {

        if(user.google_auth){
            return res.status(403).json({ error: "You can't change
account's password because you logged in through google" })
        }

        bcrypt.compare(currentPassword, user.personal_info.password, (err,
result) => {
            if(err) {
                return res.status(500).json({ error: "Some error occured
while changing the password, please try again later" })
            }
```

```
            if(!result){
                return res.status(403).json({ error: "Incorrect current
password" })
            }

            bcrypt.hash(newPassword, 10, (err, hashed_password) => {

                User.findOneAndUpdate({ _id: req.user }, {
"personal_info.password": hashed_password })
                .then((u) => {
                    return res.status(200).json({ status: 'password
changed' })
                })
                .catch(err => {
                    return res.status(500).json({ error: 'Some error
occured while saving new password, please try again later' })
                })

            })
        })

    })
    .catch(err => {
        console.log(err);
        res.status(500).json({ error : "User not found" })
    })

})

server.post('/latest-blogs', (req, res) => {

    let { page } = req.body;

    let maxLimit = 5;

    Blog.find({ draft: false })
    .populate("author", "personal_info.profile_img personal_info.username
personal_info.fullname -_id")
    .sort({ "publishedAt": -1 })
    .select("blog_id title des banner activity tags publishedAt -_id")
    .skip((page - 1) * maxLimit)
    .limit(maxLimit)
    .then(blogs => {
        return res.status(200).json({ blogs })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/all-latest-blogs-count", (req, res) => {

    Blog.countDocuments({ draft: false })
    .then(count => {
        return res.status(200).json({ totalDocs: count })
    })
    .catch(err => {
```

```
        console.log(err.message);
        return res.status(500).json({ error: err.message })
    })

})

server.get("/trending-blogs", (req, res) => {

    Blog.find({ draft: false })
    .populate("author", "personal_info.profile_img personal_info.username
personal_info.fullname -_id")
    .sort({ "activity.total_read": -1, "activity.total_likes": -1,
"publishedAt": -1 })
    .select("blog_id title publishedAt -_id")
    .limit(5)
    .then(blogs => {
        return res.status(200).json({ blogs })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/search-blogs", (req, res) => {

    let { tag, query, author, page, limit, eliminate_blog } = req.body;

    let findQuery;

    if(tag){
        findQuery = { tags: tag, draft: false, blog_id: { $ne:
eliminate_blog } };
    } else if(query){
        findQuery = { draft: false, title: new RegExp(query, 'i') }
    } else if(author) {
        findQuery = { author, draft: false }
    }

    let maxLimit = limit ? limit : 2;

    Blog.find(findQuery)
    .populate("author", "personal_info.profile_img personal_info.username
personal_info.fullname -_id")
    .sort({ "publishedAt": -1 })
    .select("blog_id title des banner activity tags publishedAt -_id")
    .skip((page - 1) * maxLimit)
    .limit(maxLimit)
    .then(blogs => {
        return res.status(200).json({ blogs })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/search-blogs-count", (req, res) => {
```

```
    let { tag, author, query } = req.body;

    let findQuery;

    if(tag){
        findQuery = { tags: tag, draft: false };
    } else if(query){
        findQuery = { draft: false, title: new RegExp(query, 'i') }
    } else if(author) {
        findQuery = { author, draft: false }
    }

    Blog.countDocuments(findQuery)
    .then(count => {
        return res.status(200).json({ totalDocs: count })
    })
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ error: err.message })
    })

})

server.post("/search-users", (req, res) => {

    let { query } = req.body;

    User.find({ "personal_info.username": new RegExp(query, 'i') })
    .limit(50)
    .select("personal_info.fullname personal_info.username
personal_info.profile_img -_id")
    .then(users => {
        return res.status(200).json({ users })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/get-profile", (req, res) => {

    let { username } = req.body;

    User.findOne({ "personal_info.username": username })
    .select("-personal_info.password -google_auth -updatedAt -blogs")
    .then(user => {
        return res.status(200).json(user)
    })
    .catch(err => {
        console.log(err);
        return res.status(500).json({ error: err.message })
    })

})

server.post("/update-profile-img", verifyJWT, (req, res) => {
```

```
    let { url } = req.body;

    User.findOneAndUpdate({ _id: req.user }, { "personal_info.profile_img":
url })
    .then(() => {
        return res.status(200).json({ profile_img: url })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/update-profile", verifyJWT, (req, res) => {

    let { username, bio, social_links } = req.body;

    let bioLimit = 150;

    if(username.length < 3){
        return res.status(403).json({ error: "Username should be at least 3
letters long" });
    }

    if(bio.length > bioLimit){
        return res.status(403).json({ error: `Bio should not be more than
${bioLimit} characters` });
    }

    let socialLinksArr = Object.keys(social_links);

    try {

        for(let i = 0; i < socialLinksArr.length; i++){
            if(social_links[socialLinksArr[i]].length){
                let hostname = new
URL(social_links[socialLinksArr[i]]).hostname;

                if(!hostname.includes(`${socialLinksArr[i]}.com`) &&
socialLinksArr[i] != 'website'){
                    return res.status(403).json({ error:
`${socialLinksArr[i]} link is invalid. You must enter a full link` })
                }

            }
        }

    } catch (err) {
        return res.status(500).json({ error: "You must provide full social
links with http(s) included" })
    }

    let updateObj = {
        "personal_info.username": username,
        "personal_info.bio": bio,
        social_links
    }
```

```
    User.findOneAndUpdate({ _id: req.user }, updateObj, {
        runValidators: true
    })
    .then(() => {
        return res.status(200).json({ username })
    })
    .catch(err => {
        if(err.code == 11000){
            return res.status(409).json({ error: "username is already
taken" })
        }
        return res.status(500).json({ error: err.message })
    })

})

server.post('/create-blog', verifyJWT, (req, res) => {

    let authorId = req.user;

    let { title, des, banner, tags, content, draft, id } = req.body;

    if(!title.length){
        return res.status(403).json({ error: "You must provide a title" });
    }

    if(!draft){
        if(!des.length || des.length > 200){
            return res.status(403).json({ error: "You must provide blog
description under 200 characters" });
        }

        if(!banner.length){
            return res.status(403).json({ error: "You must provide blog
banner to publish it" });
        }

        if(!content.blocks.length){
            return res.status(403).json({ error: "There must be some blog
content to publish it" });
        }

        if(!tags.length || tags.length > 10){
            return res.status(403).json({ error: "Provide tags in order to
publish the blog, Maximum 10" });
        }
    }

    tags = tags.map(tag => tag.toLowerCase());

    let blog_id = id || title.replace(/[^a-zA-Z0-9]/g, ' ').replace(/\s+/g,
"-").trim() + nanoid();

    if(id){

        Blog.findOneAndUpdate({ blog_id }, { title, des, banner, content,
tags, draft: draft ? draft : false })
```

```
        .then(() => {
            return res.status(200).json({ id: blog_id });
        })
        .catch(err => {
            return res.status(500).json({ error: "Failed to update total
posts number" })
        })

    } else{

        let blog = new Blog({
            title, des, banner, content, tags, author: authorId, blog_id,
draft: Boolean(draft)
        })

        blog.save().then(blog => {

            let incrementVal = draft ? 0 : 1;

            User.findOneAndUpdate({ _id: authorId }, { $inc : {
"account_info.total_posts" : incrementVal }, $push : { "blogs": blog._id }
})
            .then(user => {
                return res.status(200).json({ id: blog.blog_id })
            })
            .catch(err => {
                return res.status(500).json({ error: "Failed to update
total posts number" })
            })

        })
        .catch(err => {
            return res.status(500).json({ error: err.message })
        })

    }

})

server.post("/get-blog", (req, res) => {

    let { blog_id, draft, mode } = req.body;

    let incrementVal = mode != 'edit' ? 1 : 0;

    Blog.findOneAndUpdate({ blog_id }, { $inc : { "activity.total_reads":
incrementVal } })
    .populate("author", "personal_info.fullname personal_info.username
personal_info.profile_img")
    .select("title des content banner activity publishedAt blog_id tags")
    .then(blog => {

        User.findOneAndUpdate({ "personal_info.username":
blog.author.personal_info.username }, {
            $inc : { "account_info.total_reads": incrementVal }
        })
        .catch(err => {
            return res.status(500).json({ error: err.message })
```

```
        })

        if(blog.draft && !draft){
            return res.status(500).json({ error: 'you can not access draft
blogs' })
        }

        return res.status(200).json({ blog });

    })
    .catch(err => {
        return res.status(500).json({ error: err.message });
    })

})

server.post("/like-blog", verifyJWT, (req, res) => {

    let user_id = req.user;

    let { _id, islikedByUser } = req.body;

    let incrementVal = !islikedByUser ? 1 : -1;

    Blog.findOneAndUpdate({ _id }, { $inc: { "activity.total_likes":
incrementVal } })
    .then(blog => {

        if(!islikedByUser){
            let like = new Notification({
                type: "like",
                blog: _id,
                notification_for: blog.author,
                user: user_id
            })

            like.save().then(notification => {
                return res.status(200).json({ liked_by_user: true })
            })
        } else{

            Notification.findOneAndDelete({ user: user_id, blog: _id, type:
"like" })
            .then(data => {
                return res.status(200).json({ liked_by_user: false })
            })
            .catch(err => {
                return res.status(500).json({ error: err.message });
            })

        }

    })

})

server.post("/isliked-by-user", verifyJWT, (req, res) => {
```

```
    let user_id = req.user;

    let { _id } = req.body;

    Notification.exists({ user: user_id, type: "like", blog: _id })
    .then(result => {
        return res.status(200).json({ result })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/add-comment", verifyJWT, (req, res) => {

    let user_id = req.user;

    let { _id, comment, blog_author, replying_to, notification_id } =
req.body;

    if(!comment.length) {
        return res.status(403).json({ error: 'Write something to leave a
comment' });
    }

    // creating a comment doc
    let commentObj = {
        blog_id: _id, blog_author, comment, commented_by: user_id,
    }

    if(replying_to){
        commentObj.parent = replying_to;
        commentObj.isReply = true;
    }

    new Comment(commentObj).save().then(async commentFile => {

        let { comment, commentedAt, children } = commentFile;

        Blog.findOneAndUpdate({ _id }, { $push: { "comments":
commentFile._id }, $inc : { "activity.total_comments": 1,
"activity.total_parent_comments": replying_to ? 0 : 1 },  })
        .then(blog => { console.log('New comment created') });

        let notificationObj = {
            type: replying_to ? "reply" : "comment",
            blog: _id,
            notification_for: blog_author,
            user: user_id,
            comment: commentFile._id
        }

        if(replying_to){

            notificationObj.replied_on_comment = replying_to;

            await Comment.findOneAndUpdate({ _id: replying_to }, { $push: {
```

```
children: commentFile._id } })
            .then(replyingToCommentDoc => {
notificationObj.notification_for = replyingToCommentDoc.commented_by })

            if(notification_id){
                Notification.findOneAndUpdate({ _id: notification_id }, {
reply: commentFile._id })
                .then(notificaiton => console.log('notification updated'))
            }

        }

        new Notification(notificationObj).save().then(notification =>
console.log('new notification created'));

        return res.status(200).json({
            comment, commentedAt, _id: commentFile._id, user_id, children
        })

    })


})

server.post("/get-blog-comments", (req, res) => {

    let { blog_id, skip } = req.body;

    let maxLimit = 5;

    Comment.find({ blog_id, isReply: false })
    .populate("commented_by", "personal_info.username
personal_info.fullname personal_info.profile_img")
    .skip(skip)
    .limit(maxLimit)
    .sort({
        'commentedAt': -1
    })
    .then(comment => {
        console.log(comment, blog_id, skip)
        return res.status(200).json(comment);
    })
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ error: err.message })
    })

})

server.post("/get-replies", (req, res) => {

    let { _id, skip } = req.body;

    let maxLimit = 5;

    Comment.findOne({ _id })
    .populate({
        path: "children",
```

```
        options: {
            limit: maxLimit,
            skip: skip,
            sort: { 'commentedAt': -1 }
        },
        populate: {
            path: 'commented_by',
            select: "personal_info.profile_img personal_info.fullname
personal_info.username"
        },
        select: "-blog_id -updatedAt"
    })
    .select("children")
    .then(doc => {
        console.log(doc);
        return res.status(200).json({ replies: doc.children })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

const deleteComments = ( _id ) => {
    Comment.findOneAndDelete({ _id })
    .then(comment => {

        if(comment.parent){
            Comment.findOneAndUpdate({ _id: comment.parent }, { $pull: {
children: _id } })
            .then(data => console.log('comment delete from parent'))
            .catch(err => console.log(err));
        }

        Notification.findOneAndDelete({ comment: _id }).then(notification
=> console.log('comment notification deleted'))

        Notification.findOneAndUpdate({ reply: _id }, { $unset: { reply: 1
} }).then(notification => console.log('reply notification deleted'))

        Blog.findOneAndUpdate({ _id: comment.blog_id }, { $pull: {
comments: _id }, $inc: { "activity.total_comments": -1 },
"activity.total_parent_comments": comment.parent ? 0 : -1 })
        .then(blog => {
            if(comment.children.length){
                comment.children.map(replies => {
                    deleteComments(replies)
                })
            }
        })

    })
    .catch(err => {
        console.log(err.message);
    })
}

server.post("/delete-comment", verifyJWT, (req, res) => {
```

```
    let user_id = req.user;

    let { _id } = req.body;

    Comment.findOne({ _id })
    .then(comment => {

        if( user_id == comment.commented_by || user_id ==
comment.blog_author ){

            deleteComments(_id)

            return res.status(200).json({ status: 'done' });

        } else{
            return res.status(403).json({ error: "You can not delete this
commet" })
        }

    })

})

server.get("/new-notification", verifyJWT, (req, res) => {

    let user_id = req.user;

    Notification.exists({ notification_for: user_id, seen: false, user: {
$ne: user_id } })
    .then(result => {
        if( result ){
            return res.status(200).json({ new_notification_available: true
})
        } else{
            return res.status(200).json({ new_notification_available: false
})
        }
    })
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ error: err.message })
    })

})

server.post("/notifications", verifyJWT, (req, res) => {
    let user_id = req.user;

    let { page, filter, deletedDocCount } = req.body;

    let maxLimit = 10;

    let findQuery = { notification_for: user_id, user: { $ne: user_id } };

    let skipDocs = ( page - 1 ) * maxLimit;

    if(filter != 'all'){
```

```
        findQuery.type = filter;
    }

    if(deletedDocCount){
        skipDocs -= deletedDocCount;
    }

    Notification.find(findQuery)
    .skip(skipDocs)
    .limit(maxLimit)
    .populate("blog", "title blog_id")
    .populate("user", "personal_info.fullname personal_info.username
personal_info.profile_img")
    .populate("comment", "comment")
    .populate("replied_on_comment", "comment")
    .populate("reply", "comment")
    .sort({ createdAt: -1 })
    .select("createdAt type seen reply")
    .then(notifications => {

        Notification.updateMany(findQuery, { seen: true })
        .skip(skipDocs)
        .limit(maxLimit)
        .then(() => console.log('notification seen'));

        return res.status(200).json({ notifications });

    })
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ error: err.message });
    })

})

server.post("/all-notifications-count", verifyJWT, (req, res) => {

    let user_id = req.user;

    let { filter } = req.body;

    let findQuery = { notification_for: user_id, user: { $ne: user_id } }

    if(filter != 'all'){
        findQuery.type = filter;
    }

    Notification.countDocuments(findQuery)
    .then(count => {
        return res.status(200).json({ totalDocs: count })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})

server.post("/user-written-blogs", verifyJWT, (req, res) => {
```

```
    let user_id = req.user;

    let { page, draft, query, deletedDocCount } = req.body;

    let maxLimit = 5;
    let skipDocs = (page - 1) * maxLimit;

    if(deletedDocCount){
        skipDocs -= deletedDocCount;
    }

    Blog.find({ author: user_id, draft, title: new RegExp(query, 'i') })
    .skip(skipDocs)
    .limit(maxLimit)
    .sort({ publishedAt: -1 })
    .select(" title banner publishedAt blog_id activity des draft -_id ")
    .then(blogs => {
        return res.status(200).json({ blogs })
    })
    .catch(err => {
        return res.status(500).json({ error: err.message });
    })

})

server.post("/user-written-blogs-count", verifyJWT, (req, res) => {

    let user_id = req.user;

    let { draft, query } = req.body;

    Blog.countDocuments({ author: user_id, draft, title: new RegExp(query,
'i') })
    .then(count => {
        return res.status(200).json({ totalDocs: count })
    })
    .catch(err => {
        console.log(err.message);
        return res.status(500).json({ error: err.message });
    })

})

server.post("/delete-blog", verifyJWT, (req, res) => {

    let user_id = req.user;
    let { blog_id } = req.body;

    Blog.findOneAndDelete({ blog_id })
    .then(blog => {

        Notification.deleteMany({ blog: blog._id }).then(data =>
console.log('notifications deleted'));

        Comment.deleteMany({ blog_id: blog._id }).then(data =>
console.log('comments deleted'));
```

```
        User.findOneAndUpdate({ _id: user_id }, { $pull: { blog: blog._id
}, $inc: { "account_info.total_posts": -1 } })
        .then(user => console.log('Blog deleted'));

        return res.status(200).json({ status: 'done' });

    })
    .catch(err => {
        return res.status(500).json({ error: err.message })
    })

})


server.listen(PORT, () => {
    console.log('listening on port -> ' + PORT);
})
```

Додаток Б Лістинг схем для створення бази даних

```
import mongoose, { Schema } from "mongoose";

const blogSchema = mongoose.Schema({

    blog_id: {
        type: String,
        required: true,
        unique: true,
    },
    title: {
        type: String,
        required: true,
    },
    banner: {
        type: String,
        // required: true,
    },
    des: {
        type: String,
        maxlength: 200,
        // required: true
    },
    content: {
        type: [],
        // required: true
    },
    tags: {
        type: [String],
        // required: true
    },
    author: {
        type: Schema.Types.ObjectId,
        required: true,
        ref: 'users'
    },
    activity: {
```

```
        total_likes: {
            type: Number,
            default: 0
        },
        total_comments: {
            type: Number,
            default: 0
        },
        total_reads: {
            type: Number,
            default: 0
        },
        total_parent_comments: {
            type: Number,
            default: 0
        },
    },
    comments: {
        type: [Schema.Types.ObjectId],
        ref: 'comments'
    },
    draft: {
        type: Boolean,
        default: false
    }

},
{
    timestamps: {
        createdAt: 'publishedAt'
    }

})

export default mongoose.model("blogs", blogSchema);
import mongoose, { Schema } from "mongoose";

const commentSchema = mongoose.Schema({
```

```
    blog_id: {
        type: Schema.Types.ObjectId,
        required: true,
        ref: 'blogs'
    },
    blog_author: {
        type: Schema.Types.ObjectId,
        required: true,
        ref: 'blogs',
    },
    comment: {
        type: String,
        required: true
    },
    children: {
        type: [Schema.Types.ObjectId],
        ref: 'comments'
    },
    commented_by: {
        type: Schema.Types.ObjectId,
        require: true,
        ref: 'users'
    },
    isReply: {
        type: Boolean,
        default: false,
    },
    parent: {
        type: Schema.Types.ObjectId,
        ref: 'comments'
    }

},
{
    timestamps: {
        createdAt: 'commentedAt'
    }
})
```

```
export default mongoose.model("comments", commentSchema)
import mongoose, { Schema } from "mongoose";

const notificationSchema = mongoose.Schema({
    type: {
        type: String,
        enum: ["like", "comment", "reply"],
        required: true
    },
    blog: {
        type: Schema.Types.ObjectId,
        required: true,
        ref: 'blogs'
    },
    notification_for: {
        type: Schema.Types.ObjectId,
        required: true,
        ref: 'users'
    },
    user: {
        type: Schema.Types.ObjectId,
        required: true,
        ref: 'users'
    },
    comment: {
        type: Schema.Types.ObjectId,
        ref: 'comments'
    },
    reply: {
        type: Schema.Types.ObjectId,
        ref: 'comments'
    },
    replied_on_comment:{
        type: Schema.Types.ObjectId,
        ref: 'comments'
    },
    seen: {
        type: Boolean,
```

```
        default: false
    }
},
{
    timestamps: true
}
)

export default mongoose.model("notification", notificationSchema)
import mongoose, { Schema } from "mongoose";

let profile_imgs_name_list = ["Garfield", "Tinkerbell", "Annie", "Loki",
"Cleo", "Angel", "Bob", "Mia", "Coco", "Gracie", "Bear", "Bella", "Abby",
"Harley", "Cali", "Leo", "Luna", "Jack", "Felix", "Kiki"];
let profile_imgs_collections_list = ["notionists-neutral", "adventurer-
neutral", "fun-emoji"];

const userSchema = mongoose.Schema({

    personal_info: {
        fullname: {
            type: String,
            lowercase: true,
            required: true,
            minlength: [3, 'fullname must be 3 letters long'],
        },
        email: {
            type: String,
            required: true,
            lowercase: true,
            unique: true
        },
        password: String,
        username: {
            type: String,
            minlength: [3, 'Username must be 3 letters long'],
            unique: true,
        },
        bio: {
```

```
            type: String,
            maxlength: [200, 'Bio should not be more than 200'],
            default: "",
        },
        profile_img: {
            type: String,
            default: () => {
                return
`https://api.dicebear.com/6.x/${profile_imgs_collections_list[Math.floor(Ma
th.random() *
profile_imgs_collections_list.length)]}/svg?seed=${profile_imgs_name_list[M
ath.floor(Math.random() * profile_imgs_name_list.length)]}`
            }
        },
    },
    social_links: {
        youtube: {
            type: String,
            default: "",
        },
        instagram: {
            type: String,
            default: "",
        },
        facebook: {
            type: String,
            default: "",
        },
        twitter: {
            type: String,
            default: "",
        },
        github: {
            type: String,
            default: "",
        },
        website: {
            type: String,
            default: "",
```

```
        }
    },
    account_info:{
        total_posts: {
            type: Number,
            default: 0
        },
        total_reads: {
            type: Number,
            default: 0
        },
    },
    google_auth: {
        type: Boolean,
        default: false
    },
    blogs: {
        type: [ Schema.Types.ObjectId ],
        ref: 'blogs',
        default: [],
    }

},
{
    timestamps: {
        createdAt: 'joinedAt'
    }

})

export default mongoose.model("users", userSchema);
```

Додаток В Лістинг фронтенд частини програми

```
import { Routes, Route, Navigate } from "react-router-dom";
import Navbar from "./components/navbar.component";
import UserAuthForm from "./pages/userAuthForm.page";
import { createContext, useEffect, useState } from "react";
import { lookInSession } from "./common/session";
import Editor from "./pages/editor.pages";
import HomePage from "./pages/home.page";
import SearchPage from "./pages/search.page";
import PageNotFound from "./pages/404.page";
import ProfilePage from "./pages/profile.page";
import BlogPage from "./pages/blog.page";
import SideNav from "./components/sidenavbar.component";
import ChangePassword from "./pages/change-password.page";
import EditProfile from "./pages/edit-profile.page";
import Notifications from "./pages/notifications.page";
import ManageBlogs from "./pages/manage-blogs.page";

export const UserContext = createContext({})

export const ThemeContext = createContext({});

const darkThemePreference = () => window.matchMedia("(prefers-color-scheme:
dark)").matches;

const App = () => {

    const [userAuth, setUserAuth] = useState({});

    const [ theme, setTheme ] = useState(() => darkThemePreference() ?
"dark" : "light" );

    useEffect(() => {

        let userInSession = lookInSession("user");
        let themeInSession = lookInSession("theme");
```

```
        userInSession ? setUserAuth(JSON.parse(userInSession)) :
setUserAuth({ access_token: null })

        if (themeInSession) {
            setTheme(() => {

                document.body.setAttribute('data-theme', themeInSession);

                return themeInSession;

            })
        } else {
            document.body.setAttribute('data-theme', theme)
        }

    }, [])


    return (
        <ThemeContext.Provider value={{ theme, setTheme }}>
            <UserContext.Provider value={{userAuth, setUserAuth}}>
                <Routes>
                    <Route path="/editor" element={<Editor />} />
                    <Route path="/editor/:blog_id" element={<Editor />} />
                    <Route path="/" element={<Navbar />}>
                        <Route index element={<HomePage />} />
                        <Route path="dashboard" element={<SideNav />} >
                            <Route path="blogs" element={<ManageBlogs />}
/>
                            <Route path="notifications"
element={<Notifications />} />
                        </Route>
                        <Route path="settings" element={<SideNav />} >
                            <Route path="edit-profile"
element={<EditProfile />} />
                            <Route path="change-password"
element={<ChangePassword />} />
                        </Route>
                        <Route path="signin" element={<UserAuthForm
```

```
type="sign-in" />} />
                        <Route path="signup" element={<UserAuthForm
type="sign-up" />} />
                        <Route path="search/:query" element={<SearchPage
/>} />
                        <Route path="user/:id" element={<ProfilePage />} />
                        <Route path="blog/:blog_id" element={<BlogPage
/>}/>
                        <Route path="*" element={<PageNotFound />} />
                    </Route>
                </Routes>
            </UserContext.Provider>
        </ThemeContext.Provider>
    );

}

export default App;
```

Додаток Г Публікація у науковій конференції

УДК 004.41
Марчук Д. – ст. гр. СП-41
Тернопільський національний технічний університет імені Івана Пулюя

РОЗРОБКА БЛОГ-ПЛАТФОРМИ З ВИКОРИСТАННЯМ REACT ТА NODE.JS

Науковий керівник: д. ф., с. в. Мудрик І. Я.

Marchuk D.
Ternopil Ivan Puluj National Technical University

DEVELOPMENT OF BLOGGING PLATFORM USING REACT AND NODE.JS

Supervisor: PhD, Senior Lecturer Mudryk I. Y.

Ключові слова: веб-розробка, інтерактивність, ефективність, React, Node.js
Keywords: web-development, interactivity, efficiency, React, Node.js

У світі, де швидкість, ефективність і зручність використання цифрових продуктів стають ключовими факторами, створення блог-платформи на базі Node.js та React має стратегічне значення. Ця платформа буде не лише відповідати потребам користувачів у зручному та ефективному способі публікації контенту, але й стане важливим інструментом для розвитку онлайн-спільнот та обміну ідеями.

Використання Node.js дозволяє побудувати швидку та масштабовану серверну частину, яка забезпечить надійність та ефективність платформи. Разом з React, який дозволяє розробникам створювати динамічні та інтерактивні інтерфейси з високою швидкістю, це створить неперевершений досвід для користувачів.[1]

MongoDB в якості бази даних забезпечить гнучкість та швидкодію при зберіганні та обробці великих обсягів даних, що дозволить платформі легко масштабуватися з ростом користувачів та обсягів контенту.

Такий підхід до розробки блог-платформи дозволить створити інноваційний продукт, який відповідає сучасним вимогам і стандартам веб-розробки. Він не лише задовольнить потреби користувачів у зручному способі спілкування та публікації контенту, але й стане важливим кроком у розвитку сучасних веб-технологій.

Література

1. Why Node.js is the Best Framework for Web Development? [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/why-node-js-is-the-best-framework-for-web-development.

Додаток Д Диск з роботою