МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВЕБСЕРВІСУ НОВИН З ЕЛЕМЕНТАМИ СОЦІАЛЬНОЇ МЕРЕЖІ»**

Виконав: студент 4 курсу, групи 6.1210-2пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Д.А. Чернов
(ініціали та прізвище)
Керівник доцент кафедри програмної інженерії, к.ф.-м.н., Мильцев О.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Чернову Данилу Анатолійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебсервісу новин з елементами соціальної мережі

керівник роботи Мильцев Олександр Михайлович, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 21 » грудня 2023 року № 2180-с

2. Строк подання студентом роботи 03.06.2024 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Основні теоретичні відомості.
2. Моделювання та проєктування.
3. Розробка онлайн застосунку для перегляду та створення новин з елементами соціальної мережі.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 25.12.2023 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	09.01.2024	
2.	Збір вихідних даних.	30.01.2024	
3.	Обробка методичних та теоретичних	16.02.2024	
	джерел.		
4.	Розробка першого та другого розділу.	28.03.2024	
5.	Розробка третього розділу.	20.05.2024	
6.	Оформлення та нормоконтроль	27.05.2024	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	21.06.2024	

.

Студент _____ Д.А. Чернов
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.М. Мильцев
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка вебсервісу новин з елементами соціальної мережі»: 128 с., 33 рис., 33 джерела, 2 додатка.

ВЕБЗАСТОСУНОК, СОЦІАЛЬНА МЕРЕЖА, ДІДЖИТАЛ ГАЗЕТА, ІНТЕРНЕТ ЗАСОБИ МАСОВОЇ ІНФОРМАЦІЇ, ВЕБКОМУНІКАЦІЯ, СТВОРЕННЯ ТА ПЕРЕГЛЯД КОНТЕНТУ.

Об’єкт дослідження – інтернет сервіси та застосунки які використовуються як засобами масової інформації так і звичайними користувачами для створення або вживання контенту. Сучасні рішення діджитал газет та контент просторів.

Мета роботи: створити простий та приємний вебзастосунок, адаптувавши звичайну модель газети до сучасного вигляду. Застосунок має відповідати мінімалістичним вимогам, не бути перенавантаженим контентом і при цьому має давати змогу в повній мірі реалізувати свою думку користувачу.

Метод дослідження – спостереження, порівняння, експеримент, аналіз і синтез.

SUMMARY

Bachelor’s qualifying paper “Development of News Web Service with Social Network Elements”: 128 pages, 33 figures, 33 references, 2 supplements.

WEB APPLICATION, SOCIAL NETWORK, DIGITAL NEWSPAPER, ONLINE MEDIA, WEB COMMUNICATION, CONTENT CREATION AND VIEWING.

The object of the study is internet services and applications used by both media and ordinary users to create or consume content. Modern solutions for digital newspapers and content spaces.

The aim of the study is to create a simple and pleasant web application, adapting the usual newspaper model to a modern look. The app had to meet minimalist requirements, not be overloaded with content, and still allow the user to fully express their opinions.

The methods of research are observation, comparison, experiment, analysis and synthesis.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. Засоби масової інформації пройшли довгу історію розвитку, з поступовою модернізацією. Преса завжди має бути актуальною, як зовні так і за змістом. Звичайно щось актуально назавжди, проте апгрейдів не уникнути.

Найсучаснішою формою розповсюдження інформації є інтернет. Вебзастосунки, або вебсервіси для розповсюдження інформації стали в наш час чимось звичайним та буденним. Вебсайти це все ж досить молодий напрямок розповсюдження інформації, в порівнянні з більш класичними видами преси, тож якогось ідеального рішення, або формули, ще не існує. Проте ближче за все підібралась архітектура мінімалістичності, інформативності та лаконічності.

Мета дослідження. Проаналізувати існуючі рішення вебсервісів подачі масової інформації, та виведення абстрактного розуміння формули максимально ефективного способу проєктування контент простору в інтернеті.

Завдання дослідження. Для реалізації заданої мети виділено такі завдання:

- дослідити існуючі рішення вебсайтів великих засобів масової інформації;
- сформувати архітектуру вебсервісу, бази даних;
- реалізувати візуальну частину застосунку;
- реалізувати функціональну складову.

Об’єктом дослідження є вебсервіси, вебсайти та вебзастосунки подачі масової інформації.

Предметом дослідження є теоретичні та практичні реалізації інтернет ЗМІ як сучасної форми класичної преси.

Методами дослідження є спостереження, порівняння, експеримент, аналіз і синтез.

Для виконання поставлених задач використовувалися електронні ресурси, розробка програмного застосунку відбувалася в інтегрованому середовищі розробки WebStorm.

Бакалаврська робота складається зі вступу, трьох розділів, висновків та переліку посилань.

1 ВИБІР ТЕХНОЛОГІЙ ДЛЯ РЕАЛІЗАЦІЇ ВЕБСЕРВІСУ НОВИН

1.1 Основна інформація про full-stack вебсервіси

Сайти вебсервіси засобів масової інформації – системи у котрих користувач може переглядати контент який підпадає під категорію новин, як світових і локальних. Програмна реалізація цих систем включає у себе реалізацію frontend та backend частини застосунку.

Frontend – це те, що бачать користувачі, і включає візуальні елементи, такі як кнопки, прапорці, графіку та текстові повідомлення. Він дозволяє користувачам взаємодіяти з додатком [1].

Термін frontend відноситься до графічного інтерфейсу користувача (GUI), з яким користувачі можуть безпосередньо взаємодіяти, наприклад, навігаційні меню, елементи дизайну, кнопки, зображення і графіки [1]. З технічної точки зору, сторінка або екран, який користувач бачить з декількома компонентами інтерфейсу, називається об’єктною моделлю документа (DOM) [1]. Три основні комп’ютерні мови впливають на те, як користувачі взаємодіють з інтерфейсом:

– HTML визначає структуру інтерфейсу та різні елементи DOM;
– CSS (Каскадні таблиці стилів) визначають стиль вебзастосунку, включаючи макет, шрифти, кольори та візуальний стиль;
– JavaScript додає рівень динамічної функціональності, маніпулюючи DOM.

JavaScript може запускати зміни на сторінці та відображати нову інформацію [1]. Це означає, що інтерфейс може обробляти основні взаємодії (або запити) користувача, такі як відображення календаря або перевірка того, чи ввів користувач дійсну адресу електронної пошти. Більш складні запити frontend передає до backend.

Backend – це дані та інфраструктура, які забезпечують роботу додатку. Він зберігає та обробляє дані додатку для користувачів [1].

Іноді backend називають серверною частиною, backend додатку керує загальною функціональністю вебзастосунку. Коли користувач взаємодіє з frontend, він надсилає запит до backend у форматі HTTP [1]. Backend обробляє запит і повертає відповідь.

Коли backend обробляє запит, він зазвичай взаємодіє з наступним:

– серверами баз даних для отримання або зміни відповідних даних;
– мікросервісами, які виконують частину завдань, що запитуються користувачем;
– сторонніми API для збору додаткової інформації або виконання додаткових функцій.

Backend використовує кілька комунікаційних протоколів і технологій для виконання запиту. Backend поєднує в собі методи паралелізму, такі як розподіл запитів між багатьма серверами, кешування і дублювання даних.

1.2 Платформа Node.js

Node.js – це кросплатформне середовище виконання з відкритим вихідним кодом, яке дозволяє розробникам виконувати JavaScript-код на стороні сервера. Розроблений Райаном Далом у 2009 році, Node.js використовує для виконання коду рушій V8 JavaScript, спочатку розроблений для Google Chrome [3]. Це середовище виконання зробило революцію у веброзробці, дозволивши використовувати JavaScript, який традиційно був обмежений браузером, для створення масштабованих і високопродуктивних серверних додатків.

1.2.1 Ключові особливості Node.js

Асинхронність та керованість подіями: Node.js розроблено на основі асинхронної моделі вводу/виводу, що не блокується [3]. Це означає, що такі

операції, як читання з файлової системи або запит до бази даних, не блокують виконання іншого коду. Замість цього, ці операції виконуються у фоновому режимі, а функція зворотного виклику викликається після завершення операції. Такий підхід робить Node.js високоефективним і здатним обробляти велику кількість одночасних з’єднань з мінімальними накладними витратами.

Однопотоковий цикл обробки подій: Node.js працює в однопотоковому циклі обробки подій [3]. Хоча це може здатися обмеженням, це свідомий вибір, який спрощує одночасну роботу. Цикл обробки подій обробляє кілька запитів, вивантажуючи операції (наприклад, завдання вводу/виводу) на ядро системи, яке зазвичай оптимізоване для ефективної роботи з такими завданнями. Ця модель допомагає Node.js досягти високої пропускної здатності та масштабованості.

Швидке виконання: використовуючи двигун V8, Node.js виконує JavaScript-код на високій швидкості. V8 компілює JavaScript безпосередньо в машинний код перед виконанням, що призводить до підвищення продуктивності під час виконання. Ця ефективність робить Node.js придатним для створення додатків у реальному часі, таких як чат-додатки, онлайн ігри та сервіси прямих трансляцій.

1.2.2 Переваги Node.js

Масштабованість: Node.js за своєю суттю є масштабованим завдяки своїй неблокуючій архітектурі, керованій подіями. Він здатен керувати декількома одночасними з’єднаннями, що робить його ідеальним для додатків з високим трафіком.

Уніфікована мова: використовуючи JavaScript як для розробки на стороні клієнта, так і сервера, Node.js забезпечує уніфіковане середовище розробки. А це спрощує процес розробки, полегшує повторне використання коду, покращує співпрацю між frontend та backend розробниками.

Багата екосистема: Node.js може похизуватися багатою екосистемою, яку забезпечує npm (Node Package Manager), що є найбільшою колекцією бібліотек і модулів з відкритим вихідним кодом. npm дозволяє розробникам легко ділитися, знаходити та інтегрувати широкий спектр функціональних можливостей, скорочуючи час і зусилля на розробку.

Додатки в режимі реального часу: неблокована природа Node.js робить його особливо придатним для додатків, що працюють в режимі реального часу. Такі додатки, як обмін миттєвими повідомленнями, оновлення в реальному часі та інструменти для спільної роботи значно виграють від здатності Node.js ефективно обробляти декілька з'єднань у реальному часі.

1.2.3 Варіанти використання Node.js

Вебсервери та API: Node.js широко використовується для створення вебсерверів та RESTful API. Його здатність обробляти асинхронні запити робить його чудовим вибором для розробки API, які повинні обслуговувати численні запити без погіршення продуктивності.

Мікросервіси: Node.js є популярним вибором для побудови архітектури мікросервісів. Його легка природа і здатність обробляти асинхронні завдання роблять його ідеальним для створення невеликих, незалежних сервісів, які можна розгортати і масштабувати індивідуально.

Інтернет речей (IoT): ефективність Node.js та архітектура, керована подіями, роблять його придатним для додатків Інтернету речей, які часто вимагають одночасної роботи з численними пристроями та датчиками. Node.js може ефективно керувати потоком даних між пристроями та сервером.

Односторінкові додатки (SPA): Node.js часто використовується разом з такими фреймворками, як React, Angular або Vue.js для створення SPA. Ці додатки виграють від здатності Node.js безперешкодно обслуговувати frontend і backend код, забезпечуючи більш плавний користувацький досвід.

1.3 Фреймворк Next.js

Next.js – це надійний та універсальний фреймворк для веброзробки з відкритим вихідним кодом, побудований на основі Node.js, React та Webpack. Розроблений компанією Vercel він призначений для того, щоб розробники могли створювати готові до виробництва додатки з мінімальною конфігурацією [2]. Next.js надає широкий набір функцій, які покращують досвід розробки та оптимізують продуктивність вебдодатків.

1.3.1 Ключові особливості Next.js

Рендеринг на стороні сервера (SSR): Next.js підтримує серверний рендеринг «з коробки», що дозволяє рендерити вебсторінки на сервері перед відправкою клієнту. Це підвищує продуктивність і покращує пошукову оптимізацію (SEO), надаючи повністю відображені HTML-сторінки як пошуковим роботам, так і користувачам, що призводить до швидшого початкового завантаження сторінок.

Статична генерація сайту (SSG): Next.js також підтримує статичну генерацію сайтів, яка передбачає попередній рендеринг сторінок під час збірки. Це означає, що HTML генерується один раз і обслуговує кожен запит, що призводить до швидшого завантаження та зменшення навантаження на сервер. SSG особливо корисна для сайтів з великим обсягом контенту, де він змінюється нечасто.

Інкрементна статична регенерація (ISR): особливістю Next.js є інкрементна статична регенерація, яка дозволяє розробникам оновлювати статичний контент після початкового процесу збірки. ISR дозволяє регенерувати сторінки у фоновому режимі по мірі надходження нових запитів, поєднуючи переваги статичної генерації з динамічним оновленням контенту.

Маршрутизація на основі файлів: Next.js використовує файлову

систему маршрутизації, що спрощує процес визначення маршрутів. Сторінки створюються шляхом додавання файлів до каталогу сторінок, де кожен файл відповідає маршруту. Ця інтуїтивно зрозуміла система зменшує складність, пов’язану з традиційною маршрутизацією, і покращує організацію коду.

Маршрути API: Next.js включає підтримку маршрутів API, що дозволяє створювати кінцеві точки API безпосередньо в додатку. Розміщуючи JavaScript-файли в каталозі pages/api, розробники можуть обробляти запити API та створювати повноцінні додатки без потреби в окремому сервері backend.

Вбудована підтримка CSS та Sass: Next.js надає вбудовану підтримку CSS та Sass, що дозволяє розробникам імпортувати файли CSS та використовувати модулі CSS для локального застосування стилів. Це допомагає підтримувати чисту і зручну для обслуговування кодову базу. Він також підтримує styled-jsx, рішення CSS-in-JS для стилізації компонентів.

Оптимізація зображень: Next.js пропонує функцію автоматичної оптимізації зображень, яка подає зображення у найбільш відповідних форматах і розмірах для пристрою користувача. Це покращує час завантаження та зменшує використання пропускної здатності, сприяючи загальному покращенню користувацького досвіду.

Інтернаціоналізована маршрутизація: Next.js підтримує інтернаціоналізовану маршрутизацію, що дозволяє розробникам легко створювати багатомовні вебсайти. Ця функція дозволяє конфігурувати локальні підпункти та автоматичне визначення мови, що спрощує роботу з глобальною аудиторією.

Гаряча заміна модулів (HMR): під час розробки Next.js використовує гарячу заміну модулів для автоматичного оновлення модулів у додатку без необхідності повного перезавантаження сторінки. Це значно прискорює процес розробки та підвищує продуктивність завдяки миттєвому зворотному зв’язку.

Продуктивність та оптимізація: Next.js розроблений з думкою про продуктивність. Він включає автоматичне розділення коду, яке гарантує, що

завантажується лише необхідний код для кожної сторінки, зменшуючи час початкового завантаження та покращуючи продуктивність додатку. Крім того, такі функції, як автоматична статична оптимізація та рендеринг на стороні сервера, гарантують, що додатки, створені за допомогою Next.js, будуть швидкими та ефективними.

Екосистема та спільнота: Next.js виграє від активної спільноти та багатої екосистеми плагінів та інструментів. Обширна документація, навчальні посібники та приклади, надані Vercel та спільнотою, полегшують розробникам початок роботи та пошук рішень поширених проблем. Платформа хостингу Vercel оптимізована для додатків Next.js, пропонуючи можливості безперешкодного розгортання та масштабування.

1.4 Бібліотека React

React – це декларативна, ефективна та гнучка JavaScript бібліотека для створення користувацьких інтерфейсів [5]. Вона була розроблена і підтримується компанією Facebook та спільнотою індивідуальних розробників і компаній. Вперше React був представлений у 2013 році і з того часу став однією з найпопулярніших бібліотек для веброзробки завдяки своєму унікальному підходу до створення веб та мобільних додатків.

1.4.1 Основні поняття React

Компоненти: React впроваджує компонентну архітектуру, яка заохочує інкапсуляцію коду в незалежні частини, що можуть бути використані повторно та самостійно керують власним станом і рендерингом [5]. Компонент у React може бути простим, як функція, що повертає частину інтерфейсу користувача, або складним, як цілий додаток. Компоненти можуть бути вкладені в інші компоненти, що дозволяє створювати складні

додатки з простих компонентів.

JSX (JavaScript XML): JSX – це розширення React, яке дозволяє розробникам писати HTML-подібний синтаксис безпосередньо в JavaScript-коді [6]. Цей синтаксис не є обов'язковим, але він широко використовується, оскільки робить структуру інтерфейсу користувача компонента більш читабельною та зрозумілою з першого погляду. Код JSX перетворюється на виклики JavaScript до API React за допомогою препроцесорів, таких як Babel.

Віртуальний DOM: React використовує віртуальну DOM (Document Object Model) – концепцію програмування, згідно з якою віртуальне представлення інтерфейсу користувача зберігається в пам'яті і синхронізується з реальним DOM за допомогою бібліотеки, наприклад, ReactDOM [6]. Цей процес, відомий як узгодження, дозволяє React ефективно оновлювати DOM при зміні стану компонента, мінімізуючи витрати на оновлення.

Потік даних та управління станами: React по своїй суті підтримує односпрямований потік даних, тобто дані мають один і тільки один спосіб бути переданими в інші частини [6]. Це робить поведінку додатку передбачуваною і легшою для розуміння. Управління станами в React можна здійснювати різними способами, починаючи від локального стану в компонентах і закінчуючи просунутими бібліотеками управління станами, такими як Redux або Context API, які допомагають керувати станами в декількох компонентах.

1.4.2 Ключові особливості та переваги React

Декларативний характер: React дозволяє безболісно створювати інтерактивні інтерфейси користувача, розробляючи прості представлення для кожного стану у додатку. Коли дані змінюються, React ефективно оновлює та повторно рендерить лише потрібні компоненти. Така декларативна природа робить код більш передбачуваним і легшим для налагодження.

Компонентно-орієнтований: створення інкапсульованих компонентів, які керують власним станом, дозволяє створювати складні інтерфейси з простих частин. Компоненти в React можна використовувати повторно, що покращує робочий процес розробки та зменшує ймовірність помилок.

Потужна підтримка спільноти: React виграє від потужної підтримки спільноти та широкої екосистеми. Існує незліченна кількість бібліотек, інструментів та фреймворків, створених на основі React, таких як Next.js для SSR-додатків, Gatsby для статичних вебсайтів та багато інших, що розширюють його функціональність та застосовність у різних сферах.

Багатий набір інструментів: React супроводжується багатим набором інструментів для розробки, зокрема React Developer Tools – розширенням для браузера, яке надає інспектор React для візуалізації ієрархій компонентів, перевірки станів і пропсів компонентів та багато іншого, що значно полегшує налагодження та оптимізацію.

1.4.3 Варіанти використання

Односторінкові додатки (SPA): потужний алгоритм узгодження та ефективні оновлення React роблять його ідеальним для адаптивних та інтерактивних користувацьких інтерфейсів, яких вимагають сучасні SPA.

Мобільні додатки: використовуючи React Native, розробники можуть створювати мобільні додатки, які працюють на платформах iOS та Android, використовуючи ту саму модель компонентів React.

Прогресивні вебдодатки (PWA): компонентна архітектура React та його здатність легко інтегруватися з сервісами та іншими PWA-технологіями роблять його надійним вибором для створення додатків, які повинні працювати в автономному режимі та добре працювати на мобільних пристроях.

1.5 TypeScript

TypeScript – це статично типізована мова програмування з відкритим вихідним кодом, розроблена і підтримувана компанією Microsoft [7]. Вона є доповненням JavaScript, тобто базується на JavaScript, додаючи статичні визначення типів. Це доповнення має на меті покращити процес розробки, надаючи більш надійний інструментарій та можливості перевірки помилок, що призводить до створення більш надійних та зручних для підтримки кодових баз.

1.5.1 Ключові особливості та переваги TypeScript

Статична типізація: однією з основних особливостей TypeScript є статична типізація, яка дозволяє розробникам визначати типи для змінних, параметрів функцій, значень, що повертаються, та властивостей об’єктів [8]. Перевірка типів відбувається під час компіляції, що допомагає виявити помилки на ранніх стадіях розробки, зменшуючи кількість помилок під час виконання та покращуючи якість коду.

Виведення типів: TypeScript включає потужну систему виведення типів, яка автоматично визначає тип змінної на основі її ініціалізації [8]. Ця функція дозволяє розробникам скористатися перевагами перевірки типів без явного коментування типів, роблячи код чистішим і легшим для читання, залишаючись при цьому надійним.

Інтерфейси та псевдоніми типів: TypeScript вводить інтерфейси та псевдоніми типів, які дозволяють розробникам визначати власні типи та описувати форми об’єктів [8]. Ці можливості полегшують створення складних типів, гарантуючи, що об’єкти відповідають певним структурам, а також покращують документування коду та його читабельність.

Розширені можливості типів: TypeScript підтримує розширені можливості типів, такі як типи об’єднань, перетинів, кортежів та узагальнень

[8]. Ці можливості забезпечують більшу гнучкість і точність у визначенні типів, що дозволяє створювати компоненти та функції, які можна багаторазово використовувати і які є типо-безпечними.

Класи та об’єктно-орієнтоване програмування: TypeScript розширює синтаксис класів JavaScript функціями, типовими для традиційних об’єктно-орієнтованих мов програмування, такими як модифікатори доступу (public, private, protected), абстрактні класи та декоратори [8]. Ці вдосконалення полегшують розробку та підтримку великомасштабних додатків з чіткою, структурованою кодовою базою.

Система модулів: TypeScript підтримує синтаксис модулів ES6, що дозволяє розробникам імпортувати та експортувати модулі, що сприяє модульності коду та його повторному використанню [8]. Модульна система також включає підтримку просторів імен, що дозволяє краще організувати та інкапсулювати код.

Інструментарій та підтримка IDE: TypeScript пропонує виняткову підтримку інструментальних засобів, включаючи потужні редактори коду та інтегровані середовища розробки (IDE), такі як Visual Studio Code, які надають розширені можливості, такі як інтелектуальне завершення коду, інструменти рефакторингу та комплексні можливості налагодження. Ці інструменти значно підвищують продуктивність розробників та якість коду.

Сумісність з JavaScript: будучи доповненням до JavaScript, TypeScript повністю сумісний з існуючим JavaScript-кодом [8]. Ця сумісність дозволяє поступово впроваджувати TypeScript в існуючі проєкти. Розробники можуть поступово додавати анотації типів до своєї кодової бази JavaScript, переходячи на TypeScript у власному темпі.

1.5.2 Варіанти використання

Великомасштабні додатки: Завдяки статичній типізації та розширеним можливостям роботи з типами вона підходить для великомасштабних

додатків, які потребують зручності обслуговування та надійності.

Фреймворки та бібліотеки: багато популярних фреймворків і бібліотек JavaScript, таких як Angular, побудовані на TypeScript, використовуючи його безпеку типів і засоби розробки.

Розробка на стороні сервера: TypeScript використовується у серверній розробці за допомогою Node.js, забезпечуючи безпечне середовище для розробки backend-сервісів.

Сучасна веброзробка: TypeScript широко використовується в сучасній веброзробці для створення складних інтерактивних вебдодатків з такими фреймворками, як React та Vue.js.

1.6 Tailwind CSS

Tailwind CSS – це низькорівневий CSS-фреймворк, який легко налаштовується і призначений для спрощення розробки сучасних вебінтерфейсів [14]. На відміну від традиційних CSS-фреймворків, які пропонують заздалегідь розроблені компоненти і теми, Tailwind пропонує підхід, орієнтований на використання, що дозволяє розробникам створювати власні дизайни безпосередньо в HTML.

1.6.1 Ключові особливості та переваги використання Tailwind CSS

Tailwind CSS наголошує на підході, орієнтованому на зручність використання, надаючи повний набір утиліт-класів для різних властивостей CSS, таких як поля, відступи, колір, типографіка та макет [13]. Ці утилітарні класи дозволяють розробникам застосовувати стилі безпосередньо до елементів HTML, що дає змогу швидко створювати прототипи та ітерації дизайну без написання власного CSS.

Налаштування та конфігурація: Tailwind CSS дуже добре

налаштовується, що дозволяє розробникам пристосовувати фреймворк до своїх конкретних потреб [13]. Конфігураційний файл (tailwind.config.js) можна використовувати для налаштування системи дизайну за замовчуванням, включаючи кольори, інтервали, шрифти та точки зупинки. Така гнучкість гарантує, що фреймворк може адаптуватися до різних вимог проєкту та систем дизайну.

Адаптивний дизайн: Tailwind CSS надає вбудовану підтримку адаптивного дизайну за допомогою класів утиліт, орієнтованих на мобільні пристрої [13]. Розробники можуть застосовувати стилі, умовно засновані на розмірі екрану, використовуючи адаптивні префікси, такі як sm:, md:, lg: і xl:, що полегшує створення адаптивних макетів, які безперешкодно працюють на різних пристроях і розмірах екранів.

Компонентний підхід: хоча Tailwind просуває методологію, що орієнтована на зручність використання, він також підтримує створення компонентів для багаторазового використання [13]. Розробники можуть виокремлювати повторювані патерни в компоненти, використовуючи класи утиліт для підтримки узгодженості та спрощення обслуговування. Такий підхід забезпечує модульну систему проєктування, де компоненти можна повторно використовувати та компонувати для побудови складних інтерфейсів.

Оптимізація продуктивності: Tailwind CSS включає декілька функцій для оптимізації продуктивності [13]. Фреймворк можна очистити від не використовуваних класів CSS у виробничих збірках, що значно зменшує розмір файлу CSS. Цей процес, відомий як «струшування дерева», гарантує, що до файлу будуть включені лише ті стилі, які використовуються в проєкті, що призводить до пришвидшення завантаження та покращення продуктивності.

Інтеграція з сучасними інструментами розробки: Tailwind CSS легко інтегрується з сучасними інструментами розробки та робочими процесами [13]. Його можна використовувати разом з популярними JavaScript-

фреймворками, такими як React, Vue.js та Angular, і він сумісний з різними інструментами збірки, такими як Webpack, Rollup та Parcel. Ця сумісність гарантує, що Tailwind можна легко інтегрувати в існуючі проєкти та середовища розробки.

Швидкість та ефективність: Tailwind CSS підвищує швидкість та ефективність розробки завдяки багатому набору попередньо визначених класів утиліт. Розробники можуть швидко застосовувати стилі без написання власного CSS, скорочуючи час, що витрачається на стилізацію, а також прискорюючи ітерації та створення прототипів.

Послідовність та зручність супроводу: підхід, орієнтований на утиліти, сприяє узгодженості у всьому додатку, заохочуючи використання попередньо визначених утилітних класів [13]. Це зменшує ймовірність розбіжностей у стилях і спрощує підтримку, оскільки стилі застосовуються безпосередньо до елементів, а не розпорошуються по декількох CSS-файлах.

Можливості кастомізації: Tailwind CSS дуже добре налаштовується, що дозволяє розробникам визначати власну систему дизайну та розширювати фреймворк відповідно до конкретних вимог проєкту [13]. Ця гнучкість гарантує, що фреймворк можна адаптувати до різних специфікацій дизайну та рекомендацій щодо брендингу.

Невеликі розміри пакунків: завдяки вбудованим можливостям очищення, Tailwind CSS може створювати невеликі пакети CSS, видаляючи невикористані стилі. Це призводить до зменшення розміру файлів, пришвидшення часу завантаження та покращення загальної продуктивності вебдодатків.

1.6.2 Варіанти використання

Прототипування та швидка розробка: Tailwind CSS ідеально підходить для створення прототипів і швидкої розробки завдяки своєму підходу,

орієнтованому на користь користувача, і широкому набору утилітарних класів. Розробники можуть швидко створювати проєкти, не витрачаючи час на кастомний CSS.

Кастомні системи дизайну: Tailwind CSS можна використовувати для створення кастомних систем дизайну, які відповідають певним принципам брендингу та дизайну. Гнучкість і можливість налаштування роблять його придатним для проєктів, які потребують індивідуального підходу до дизайну та стилістики.

Сучасні вебдодатки: Tailwind CSS добре інтегрується з сучасними фреймворками та інструментами веброзробки, що робить його підходящим вибором для створення адаптивних, масштабованих і підтримуваних вебдодатків. Його утилітарні класи полегшують створення складних інтерфейсів з мінімальною кількістю кастомного CSS.

1.7 Бази даних

База даних – це структурована система для збору, зберігання, управління та пошуку інформації [15]. Це важливий компонент сучасних обчислювальних середовищ, що дозволяє ефективно обробляти великі обсяги даних у різних сферах застосування – від управління бізнесом і фінансового аналізу до соціальних мереж і наукових досліджень.

1.7.1 Фундаментальні концепції баз даних

Моделі даних: визначає структуру та формат даних. Найпоширенішими моделями даних є ієрархічна модель, мережева модель, реляційна модель, а останнім часом – модель NoSQL, яка включає в себе підмоделі документів, ключ-значення, широких стовпців і графів [17]. Вибір моделі даних залежить

від конкретних вимог і обмежень програми, включаючи складність зв’язків між даними, потреби в масштабуванні та вимоги до продуктивності.

Схеми: план структури бази даних. Вона визначає, як організовані дані, включаючи таблиці, поля, зв’язки, індекси та обмеження [16]. У реляційній базі даних схема визначає стовпці та типи даних у кожній таблиці, а також зв’язки між таблицями.

Запити до бази даних: запит на доступ до даних. Запити дозволяють користувачам взаємодіяти з даними, отримувати їх і маніпулювати ними. SQL (Structured Query Language – мова структурованих запитів) є найбільш широко використовуваною мовою для запитів до реляційних баз даних, що надає потужні засоби для виконання складних пошуків і маніпуляцій з даними.

Властивості ACID (Atomicity, Consistency, Isolation, Durability): набір принципів, які гарантують надійну обробку транзакцій з базами даних. Ці властивості гарантують, що навіть у разі збою системи дані залишаться точними, послідовними та неушкодженими.

1.7.2 Типи баз даних

Реляційні бази даних: зберігають дані в таблицях, які пов’язані між собою відношеннями, що зазвичай визначаються первинними та зовнішніми ключами [17]. Реляційна модель дозволяє виконувати складні запити та аналіз за допомогою мови SQL, що робить її придатною для додатків, які вимагають суворої цілісності даних і складного управління транзакціями. Прикладами є MySQL, PostgreSQL, Oracle Database та Microsoft SQL Server.

Бази даних NoSQL: призначені для обробки великих обсягів структурованих, напівструктурованих і неструктурованих даних [17]. Вони особливо добре підходять для моделей даних, що швидко розвиваються, і легше масштабуються в певних вимірах, ніж традиційні реляційні бази

даних. NoSQL бази даних часто поділяють на такі типи, як бази даних документів (наприклад, MongoDB), сховища ключ-значення (наприклад, Redis), сховища з широкими стовпцями (наприклад, Cassandra) та бази даних графів (наприклад, Neo4j).

Бази даних in-Memory: зберігають дані в оперативній пам’яті комп’ютера, а не на диску, що забезпечує швидший доступ і час відгуку [15]. Вони ідеально підходять для додатків, які потребують обробки та швидкості в режимі реального часу, таких як кешування, управління сеансами та аналітика в реальному часі.

Розподілені бази даних: керують базою даних у різних фізичних місцях, незалежно від того, розподілені вони на декількох машинах в одному центрі обробки даних або розкидані по декількох центрах обробки даних. Ці системи призначені для підвищення відмовостійкості та доступності, зменшення затримок і обробки великих обсягів даних, як у технологіях розподіленого реєстру, таких як блокчейн.

1.7.3 Переваги використання баз даних

Ефективне управління даними: бази даних забезпечують систематизований та організований метод управління даними, що дозволяє швидко знаходити їх, ефективно зберігати та легко маніпулювати ними.

Масштабованість: сучасні бази даних розроблені для ефективного масштабування для обробки зростаючих обсягів даних і більш складних запитів, як шляхом вертикального оновлення, так і горизонтального масштабування.

Безпека: бази даних пропонують надійні функції безпеки, включаючи контроль доступу, шифрування та журнали аудиту, які допомагають захистити конфіденційні дані від несанкціонованого доступу та порушень.

Цілісність даних: бази даних забезпечують цілісність і узгодженість

даних за допомогою обмежень, транзакцій і властивостей ACID, зменшуючи аномалії даних і забезпечуючи точність операцій.

1.8 Система керування базами даних PostgreSQL

PostgreSQL, яку часто називають просто Postgres, – це сучасна реляційна система керування базами даних (СКБД) з відкритим вихідним кодом, яка робить акцент на розширюваності та відповідності стандартам SQL [18]. Спочатку розроблена в Каліфорнійському університеті в Берклі у 1980-х роках, PostgreSQL значно розвинулася протягом десятиліть, ставши однією з провідних систем управління базами даних, яку цінують за її надійність, гнучкість і здатність працювати з різними робочими навантаженнями, від окремих машин до вебсервісів з великою кількістю одночасних користувачів.

1.8.1 Основні особливості та можливості PostgreSQL

SQL сумісність і реляційна модель: PostgreSQL повністю відповідає стандарту SQL і пропонує багато розширених можливостей SQL, яких немає в інших СКБД. Вона підтримує повну підмножину SQL, включаючи транзакції, підвибірки, тригери, представлення, цілісність посилань на зовнішні ключі та складне блокування. PostgreSQL обробляє паралелізм за допомогою системи управління багатовекторним паралелізмом (MVCC), підвищуючи продуктивність і масштабованість за рахунок обробки великих обсягів транзакцій.

Розширюваність: визначальною особливістю PostgreSQL є її розширюваність. Користувачі можуть визначати та створювати власні типи даних, власні функції та навіть писати код на різних мовах програмування

без перекомпіляції бази даних [19]. Крім того, PostgreSQL підтримує різноманітні методи індексування, включаючи B-дерево, хеш, GiST, SP-GiST, GIN і BRIN індекси, які можна розширювати за допомогою визначених користувачем операторів.

Міцність і надійність: PostgreSQL відома своєю цілісністю та відмовостійкістю даних. Вона автоматично захищає транзакції, використовуючи властивості ACID (атомарність, узгодженість, ізоляція, довговічність), гарантуючи, що транзакції бази даних обробляються надійно. Крім того, він включає такі функції, як попередній запис, відновлення в момент часу, асинхронна реплікація та підтримка табличного простору для захисту даних і забезпечення відновлення після фізичних і логічних помилок.

Продуктивність і масштабованість: PostgreSQL добре підходить для роботи з великими наборами даних і обслуговування великої кількості одночасних користувачів завдяки вдосконаленому планувальнику/оптимізатору запитів і підтримці складних запитів. Її продуктивність можна підвищити за допомогою декількох налаштувань конфігурації, які дозволяють налаштовувати її відповідно до робочого навантаження і конкретного середовища.

Підтримка розширених типів даних: окрім традиційних типів даних, PostgreSQL підтримує геометричні примітиви, мережеві адреси та типи даних для повнотекстового пошуку. Вона також надає широку підтримку JSON та інших нереляційних типів даних, що робить її придатною для додатків, які потребують гнучкості баз даних NoSQL у поєднанні з надійністю реляційної системи.

Велика спільнота та відкритий вихідний код: будучи проєктом з відкритим вихідним кодом, PostgreSQL є безкоштовною у використанні, що сприяло її широкому розповсюдженню. Вона має активну спільноту, яка сприяє її безперервному розвитку і надає значну документацію, сторонні інструменти та підтримку. Ця спільнота також допомагає підтримувати бібліотеку розширень, які додають специфічні функції до стандартних

інсталяцій PostgreSQL, наприклад, PostGIS для географічних інформаційних систем (ГІС).

1.8.2 Застосування PostgreSQL

Вебсервіси: здатність обробляти кілька одночасних транзакцій робить її ідеальною для внутрішніх систем вебдодатків і сервісів.

ГІС-системи: завдяки розширенню PostGIS, PostgreSQL стає потужним інструментом для зберігання географічних даних і запитів.

Аналітика: підтримка складних запитів і можливість інтеграції з аналітичними інструментами робить її придатною для бізнес-аналітики та аналітичних додатків.

Корпоративні додатки: відповідність стандарту ACID і надійні можливості управління транзакціями роблять його надійним інструментом для критично важливих корпоративних систем баз даних.

1.9 Моделі архітектур програмного забезпечення

Моделі архітектури програмного забезпечення забезпечують високорівневі структури для проєктування та побудови програмних систем, визначаючи компоненти, взаємозв’язки та взаємодії, які складають систему [21]. Існує кілька різних моделей архітектури програмного забезпечення, кожна з яких має свої особливості, переваги та варіанти використання.

1.9.1 Основні моделі вебархітектури

Монолітна архітектура: у моделі монолітної архітектури весь програмний додаток розробляється і розгортається як єдине ціле [22]. Всі

компоненти, включаючи інтерфейс користувача, бізнес-логіку та рівні доступу до даних, тісно пов’язані між собою і виконуються в рамках одного процесу. Монолітні архітектури відносно прості у розробці та розгортанні, але можуть стати складними в підтримці та масштабуванні, коли додаток зростає в розмірі та складності.

Архітектура клієнт-сервер: модель клієнт-серверної архітектури розділяє додаток на дві окремі частини: клієнт, відповідальний за представлення користувацького інтерфейсу та обробку взаємодії з користувачем, і сервер, відповідальний за обробку запитів, виконання бізнес-логіки та управління даними [22]. Ця модель сприяє масштабованості, гнучкості та централізованому управлінню даними, що робить її придатною для широкого спектру додатків, від вебсистем до розподілених корпоративних додатків.

Архітектура мікросервісів: додаток на набір невеликих, незалежно розгорнутих сервісів, кожен з яких відповідає за певну бізнес-можливість [22]. Кожен мікросервіс взаємодіє з іншими через чітко визначені API, часто використовуючи легкі протоколи, такі як HTTP або черги повідомлень. Мікросервіси сприяють гнучкості, масштабованості та швидкому розвитку, але вони також створюють складнощі з точки зору оркестрування послуг, узгодженості даних та управління розгортанням.

Сервіс-орієнтована архітектура (SOA): підхід до проєктування програмного забезпечення, при якому додаток складається з слабо пов’язаних, багаторазово використовуваних і сумісних сервісів [22]. Ці сервіси надають функціональність через чітко визначені інтерфейси і можуть бути організовані для виконання складних бізнес-процесів. SOA наголошує на багаторазовому використанні послуг, сумісності та модульності, що робить її придатною для великомасштабних корпоративних систем з різноманітними технологічними стеками.

Архітектура керована подіями (Event-Driven Architecture, EDA: заснована на принципі створення та споживання подій, де події представляють собою важливі події або зміни стану в системі [22].

Компоненти в архітектурі, керованій подіями, взаємодіють асинхронно через події, забезпечуючи вільний зв’язок, масштабованість і реакцію в реальному часі. EDA добре підходить для систем зі складними вимогами до обробки подій, таких як платформи Інтернету речей, аналітика в реальному часі та архітектури мікросервісів, керованих подіями.

Багаторівнева архітектура: розділяє додаток на шари, кожен з яких відповідає за певний аспект функціональності (наприклад, презентація, бізнес-логіка, доступ до даних) [22]. Компоненти в межах кожного шару взаємодіють лише з компонентами в сусідніх шарах, дотримуючись суворого розмежування завдань. Багаторівнева архітектура сприяє модуляризації, повторному використанню та підтримці, що робить її популярним вибором як для монолітних, так і для розподілених систем.

1.10 Висновки до розділу 1

У даному розділі представлені основні поняття для розуміння проблеми і можливі рішення для реалізації поставлених задач. Був проведений аналіз і детальний опис програмних, архітектурних і модельних рішень потрібних для проєктування і реалізації програмного забезпечення.

Дивлячись на представленні програмні, архітектурні та модельні рішення був визначений стек технології потрібний для створення вебсайту:

- Next.js / React / TypeScript – для клієнтської частини;
- Tailwind CSS – для створення гнучкого, адаптивного мінімалістичного дизайну;
- Node.js – для серверної частини;
- PostgreSQL СКБД для SQL бази даних;
- клієнт-серверна архітектура.

2 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ

2.1 Проєктування бази даних

В якості бази даних буде використана реляційна база даних SQL написана на СКБД PostgreSQL. У загальній структурі бази даних мають бути присутні таблиці, розглянемо їх детальніше.

Таблиця адміністраторів з мінімальною інформацією адмінастратора. У цій таблиці мають бути присутні такі поля як:

– ідентифікатор адміністратора;
– псевдонім адміністратора;
– пароль адміністратора;
– електронна пошта;
– зображення профілю користувача.

Таблиця користувачів з повною інформацією користувача. У цій таблиці мають бути присутні такі поля як:

– ідентифікатор користувача;
– псевдонім користувача;
– пароль користувача;
– ім’я користувача;
– прізвище користувача;
– електронна пошта;
– список ідентифікаторів новин на які користувач додав позитивну оцінку;
– список ідентифікаторів новин на які користувач додав негативну оцінку;
– список ідентифікаторів коментарів на які користувач додав позитивну оцінку;
– список ідентифікаторів коментарів на які користувач додав негативну оцінку;

– зображення профілю користувача.

Таблиця новин з повною інформацією про новину. У цій таблиці мають бути присутні такі поля як:

– ідентифікатор новини;
– заголовок новини;
– основний текст новини;
– псевдонім користувача котрий створив новину;
– картинки новини;
– дата створення новини;
– позитивні оцінки новини;
– негативні оцінки новини.

Таблиця коментарів з повною інформацією про новину. У цій таблиці мають бути присутні такі поля як:

– ідентифікатор коментаря;
– ідентифікатор новини під якою написаний коментар;
– згадка іншого користувача;
– псевдонім користувача котрий написав коментар;
– текст коментаря;
– дата створення коментаря;
– позитивні оцінки коментаря;
– негативні оцінки коментаря.

Таблиця повідомлень з повною інформацією про новину. У цій таблиці мають бути присутні такі поля як:

– ідентифікатор повідомлення;
– основний текст повідомлення;
– згадка користувача;
– текст коментаря;
– ідентифікатор новини;
– дата створення повідомлення;
– псевдонім користувача;
– статус прочитання повідомлення.

2.2 Побудова UML діаграм

Діаграми уніфікованої мови моделювання (UML) – це стандартизована графічна нотація, яка використовується в програмній інженерії для візуалізації, специфікації, побудови та документування структури та поведінки програмних систем [24]. Діаграми UML забезпечують спільну мову і набір символів для передачі проєктних ідей, вимог і рішень між зацікавленими сторонами, включаючи розробників, дизайнерів, тестувальників і менеджерів проєктів. Ці діаграми допомагають полегшити співпрацю, аналіз і розуміння складних систем протягом усього життєвого циклу розробки програмного забезпечення.

Не менш важливою являється ER діаграма. Діаграма зв’язків між сутностями (див. рис. 2.1) – це тип блок-схеми, яка ілюструє, як «сутності», такі як люди, об’єкти або концепції, пов’язані один з одним в системі.

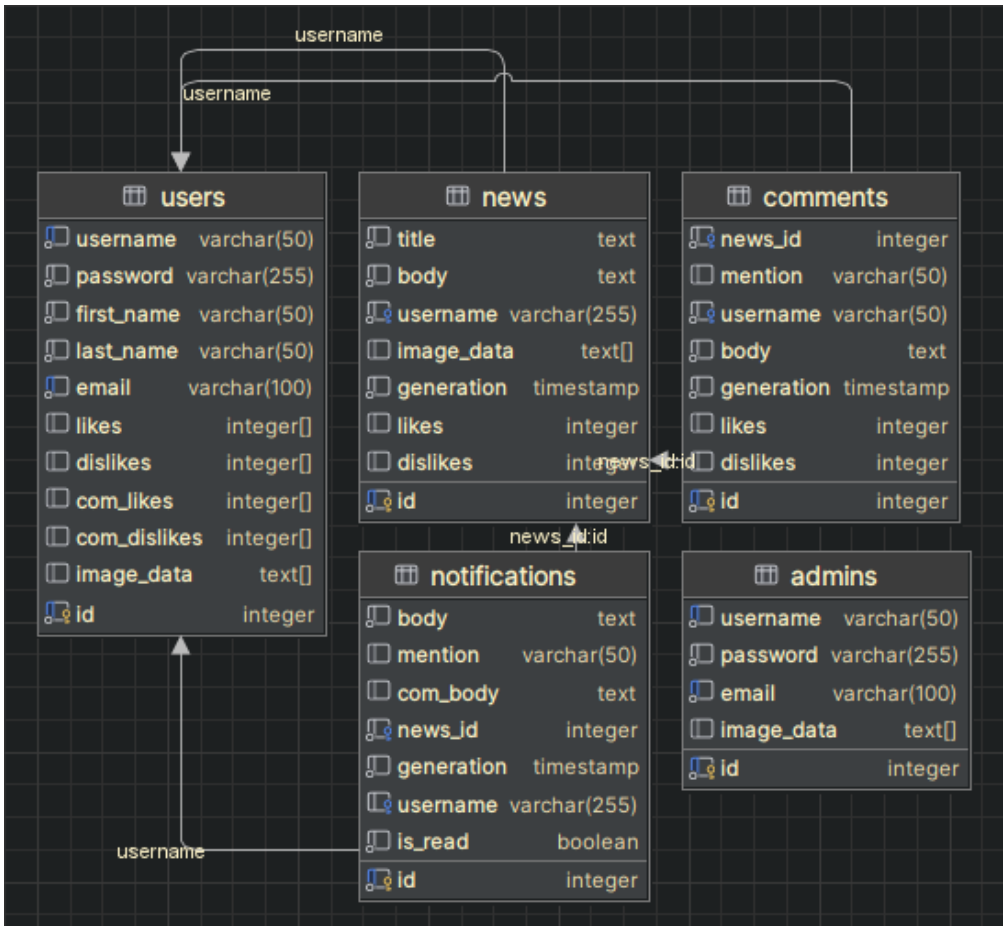

Рисунок 2.1 – ER діаграма бази даних

2.2.1 Структурні діаграми

Діаграми класів: відображають статичну структуру системи, ілюструючи класи, їхні атрибути, методи, зв’язки та залежності [24]. Вони показують план об’єктів системи та їх взаємодію, допомагаючи розробникам зрозуміти архітектуру та дизайн системи. На рисунку 2.2 представлена UML діаграма класів майбутнього вебзастосунку.

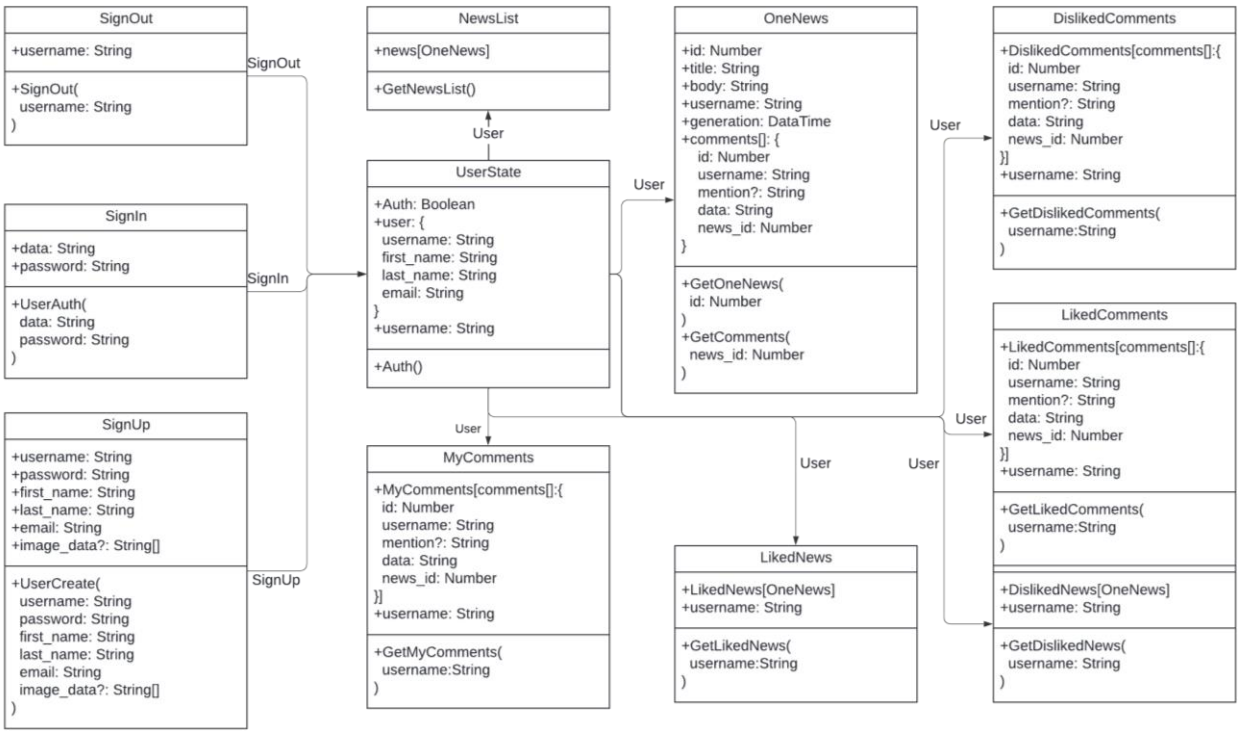

Рисунок 2.2 – Діаграма класів

Об’єктні діаграми: надають знімок системи в певний момент часу, показуючи екземпляри класів та їх взаємозв’язки [24]. Вони ілюструють конкретні об’єкти та їх атрибути, представляючи реальні сценарії або стани системи.

Діаграми пакетів: організовують і структурують компоненти системи у пакети, модулі або простори імен [24]. Вони показують залежності між пакетами і допомагають керувати складністю системи, розбиваючи її на керовані частини.

Діаграми компонентів (див. рис. 2.3): зображують фізичні компоненти

системи, такі як бібліотеки, виконувані файли та модулі багаторазового використання, а також їхні взаємозв’язки [24]. Вони ілюструють архітектуру реалізації та розгортання системи.

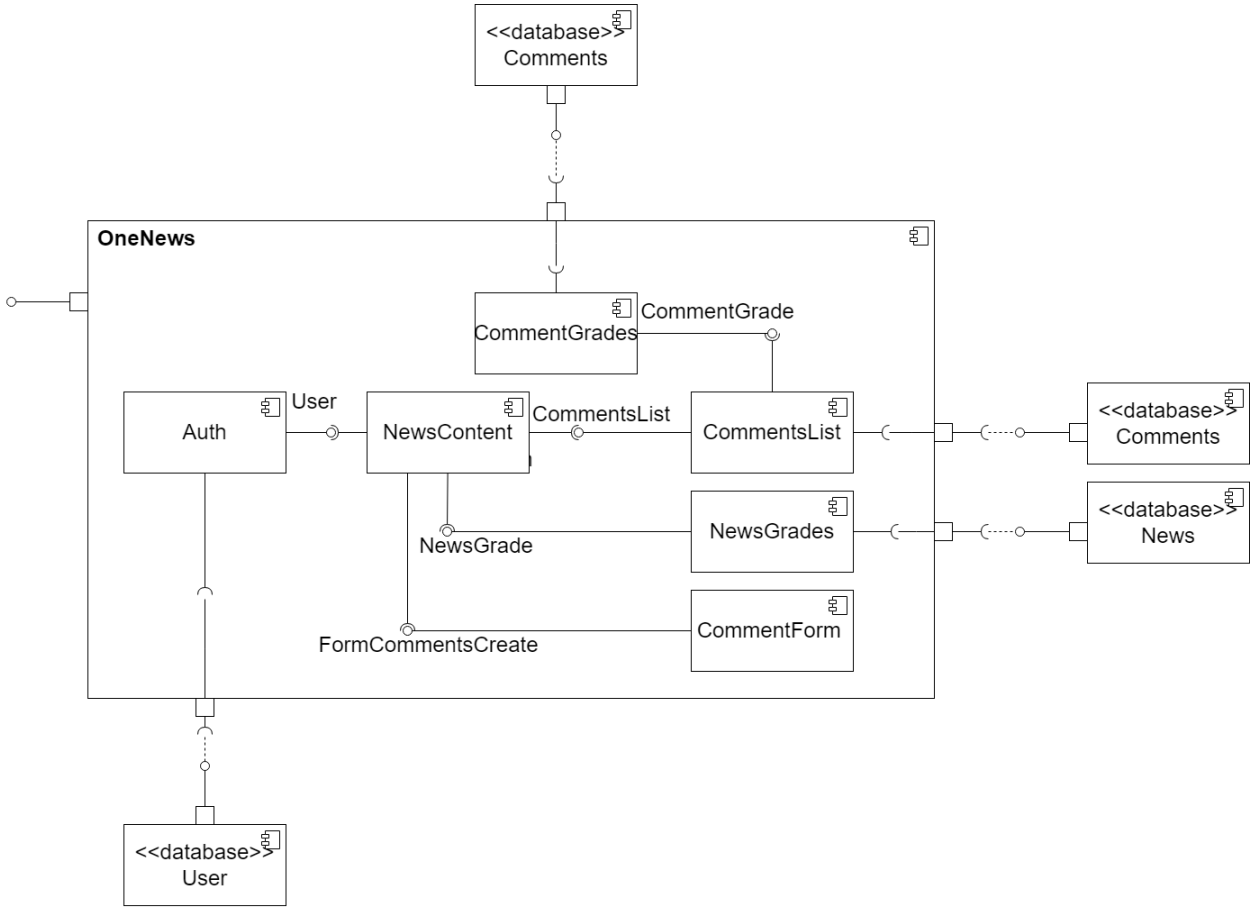

Рисунок 2.3 – Діаграма компонентів однієї новини

Композитні структурні діаграми: описують внутрішню структуру класу або компонента та його взаємодію з іншими частинами системи [24]. Вони показують, як складні об’єкти складаються з менших частин і як вони взаємодіють для досягнення функціональності.

2.2.2 Поведінкові діаграми

Діаграми варіантів використання (див. рис. 2.4): відображають функціональні вимоги до системи, ілюструючи акторів, варіанти

використання та їхні взаємозв’язки [24]. Вони допомагають визначити функціональні можливості системи, взаємодію користувачів та граничні контексти.

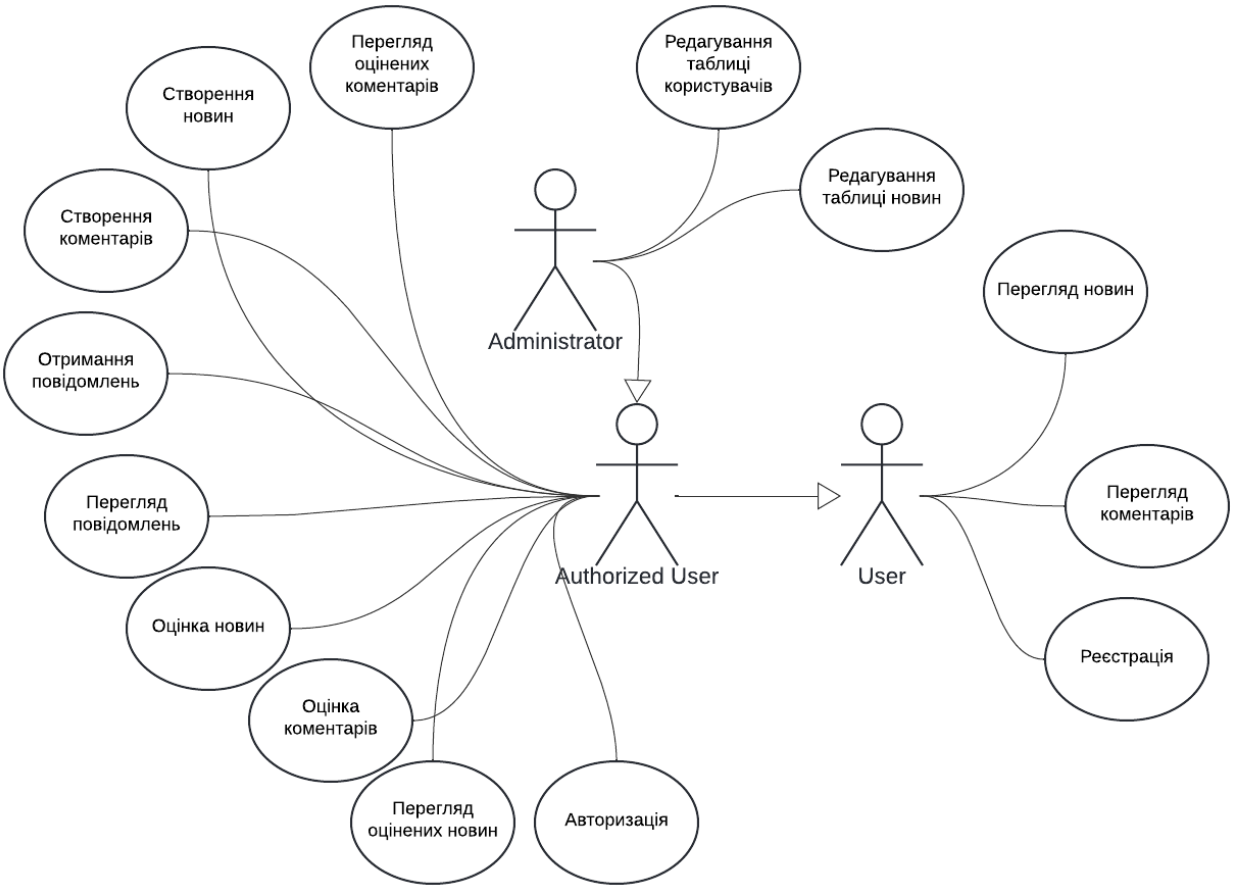

Рисунок 2.4 – Діаграми варіантів використання

Діаграми діяльності (див. рис. 2.5): моделюють динамічну поведінку системи, представляючи робочі потоки, процеси та види діяльності [24]. Вони візуалізують потік управління і даних між компонентами системи, показуючи послідовність дій і точки прийняття рішень.

Діаграми станів: моделюють поведінку об’єктів або компонентів як кінцевий набір станів і переходів між ними [24]. Вони представляють життєвий цикл об’єктів та їх залежну від стану поведінку, показуючи, як об’єкти реагують на події та змінюють стани.

Діаграми огляду взаємодії: надають огляд взаємодії між об’єктами або компонентами, поєднуючи елементи діаграм діяльності та послідовності [24].

Вони показують високорівневі взаємодії та потоки управління в складних сценаріях.

Рисунок 2.5 – Діаграма діяльності створення новини

Діаграми послідовності (див. рис. 2.6): відображають взаємодію між об’єктами або компонентами в часі, ілюструючи послідовність повідомлень, якими вони обмінюються [24]. Вони показують динамічну поведінку системи під час певного сценарію або варіанту використання.

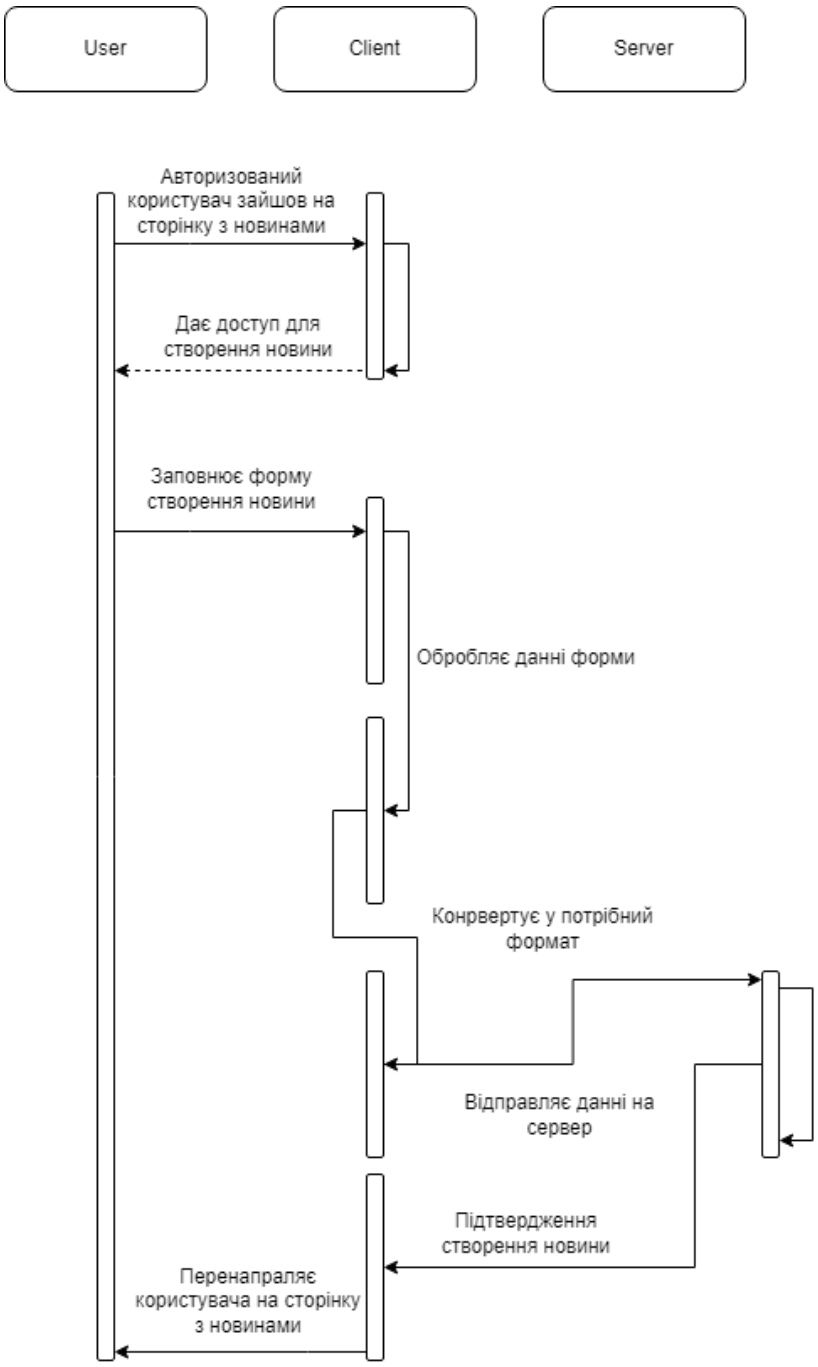

Рисунок 2.6 – Діаграма послідовності створення новини

2.2.3 Схема розгортання

Діаграми розгортання: зображують фізичне розгортання програмних компонентів, апаратних вузлів і мережевих з’єднань у розподіленій системі [24]. Вони ілюструють, як програмні артефакти розгортаються в апаратній інфраструктурі, показуючи топологію і конфігурацію розгортання.

2.3 Моделювання структури сторінок сайту

Потрібно провести моделювання структури сторінок (див. рис. 2.7). Початковою сторінкою може бути ознайомча сторінка, сторінка з загальною інформацією або навігаційними формами, для доступу до самого контенту.

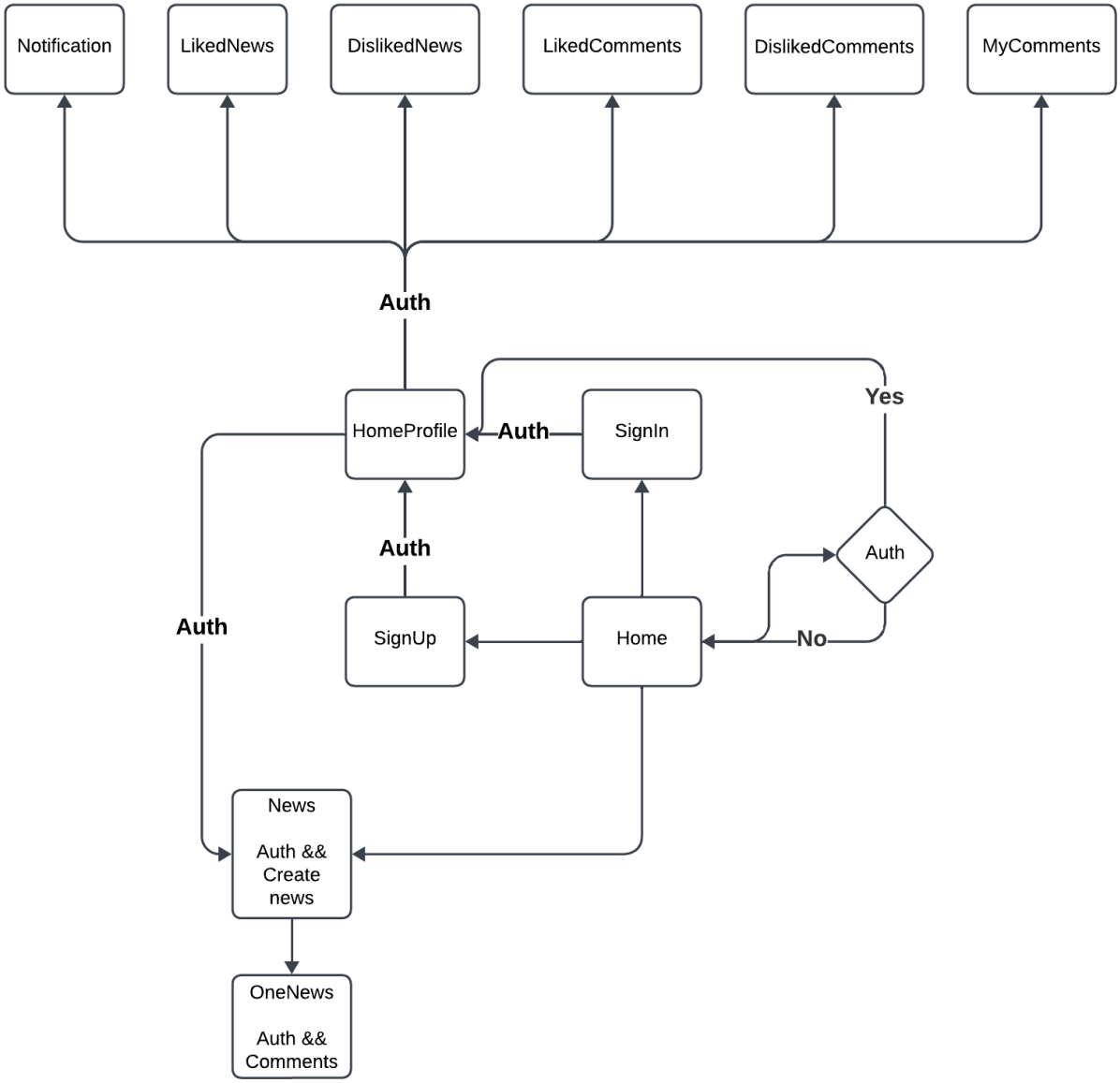

Рисунок 2.7 – Структура сторінок сайту

Стартовою була обрана сторінка із реєстрацією/авторизацією, задля того щоб мотивувати користувача для більш глибинної взаємодії із застосунком.

Після авторизації користувач матиме доступ компоненти профілю, та

до сторінок з додатковою інформацією про його власні коментарі та повідомлення, списки оцінених їм новин та коментарів, також стане доступна сторінка кастомізації профілю та можливість створювати новини.

Навігація до інших сторінок буде доступна і для неавторизованих/незареєстрованих користувачів, задля збереження можливості звичайного використання сайту засобу масової інформації.

Сторінка з новинами буде надавати відсортовані новини у певній «дозованій» кількості обраною користувачем.

Сторінка новини матиме тіло новини та компонент коментарів.

У компоненті можна переглянути існуючі коментарі, а також якщо користувач авторизован можна буде створити свій.

2.4 Проєктування архітектури застосунку

Під час вивчення можливих варіантів реалізації архітектур вебсайтів було вирішено що застосувати клієнт-серверну модель для реалізації проєкту.

Архітектура клієнт-сервер є базовою моделлю для проєктування та реалізації розподілених обчислювальних систем, де функціональність програми розподілена між двома окремими об’єктами: клієнтом і сервером [25]. Ця архітектурна модель широко використовується в різних сферах, включаючи вебдодатки, корпоративні системи та мережеві додатки, завдяки своїй масштабованості, гнучкості та простоті обслуговування.

2.4.1 Компоненти клієнт-серверної архітектури

Клієнт – це інтерфейсний компонент, який відповідає за взаємодію з користувачем та представлення користувацького інтерфейсу [25]. Це може бути веббраузер, мобільний додаток, десктопний додаток або будь-який

пристрій, здатний робити запити до сервера. Клієнт ініціює запити до сервера і обробляє відповіді, отримані від сервера. Основні компоненти клієнта включають:

- інтерфейс користувача (UI): відтворення графічного інтерфейсу користувача для програми, включаючи такі елементи, як форми, кнопки, меню та віджети;
- генерація запитів: генерація запитів до сервера на основі дій користувача, таких як надсилання форми, натискання посилання або виклик API;
- обробка відповідей: обробка відповідей, отриманих від сервера, наприклад, відображення даних, рендеринг сторінок або запуск дій на стороні клієнта;
- локальна обробка: виконання завдань локально на стороні клієнта, наприклад, перевірка вхідних даних, маніпуляції з даними на стороні клієнта та кешування.

Сервер – це внутрішній компонент, який відповідає за обробку клієнтських запитів, виконання бізнес-логіки та управління даними [25]. Він отримує запити від клієнтів, виконує необхідні операції та надсилає відповіді клієнтам. Сервери можуть варіюватися від простих HTTP-серверів до складних серверів додатків, здатних обробляти великі обсяги запитів. Основні компоненти сервера включають:

- обробка запитів: обробка вхідних запитів від клієнтів, перенаправлення їх до відповідних компонентів або сервісів та генерування відповідей;
- бізнес-логіка: реалізація основної функціональності додатку, такої як аутентифікація, авторизація, обробка даних та робочі процеси додатку;
- управління даними: взаємодія з базами даних, файловими системами або іншими джерелами даних для зберігання, отримання, оновлення та видалення даних, необхідних для програми;
- управління ресурсами: обслуговування статичних ресурсів

(наприклад, зображень, таблиць стилів, скриптів) та ефективне управління ресурсами програми для забезпечення оптимальної продуктивності та масштабованості.

2.4.2 Зв’язок між клієнтом і сервером

Зв’язок між клієнтом і сервером зазвичай відбувається через мережу з використанням стандартизованих протоколів, таких як HTTP, WebSocket або віддалений виклик процедур (RPC) [25]. Клієнт ініціює запити до сервера, вказуючи бажану операцію (наприклад, отримання даних, відправлення форми) і надаючи будь-які необхідні параметри. Сервер обробляє запит, виконує необхідну логіку і надсилає клієнту відповідь, що містить результат операції або будь-які необхідні дані.

2.4.3 Переваги клієнт-серверної архітектури

Масштабованість: архітектура клієнт-сервер підтримує горизонтальне і вертикальне масштабування, що дозволяє системам справлятися зі зростаючими навантаженнями шляхом додавання нових серверів або збільшення їхньої потужності.

Гнучкість: різні типи клієнтів (наприклад, веббраузери, мобільні додатки, десктопні додатки) можуть взаємодіяти з одним і тим же сервером, що забезпечує багатоплатформну підтримку та враховує різноманітні уподобання користувачів.

Централізоване управління даними: дані зберігаються та управляються централізовано на сервері, що забезпечує узгодженість, цілісність та безпеку в усьому додатку.

Модульність та ремонтопридатність: розподіл завдань між

клієнтськими та серверними компонентами сприяє модульності, що полегшує підтримку, оновлення та розширення додатку з часом.

Безпека: чутливі операції та дані можуть бути захищені на стороні сервера, що зменшує ризики безпеки, пов’язані з операціями та взаємодією на стороні клієнта.

2.4.4 Проблеми клієнт-серверної архітектури

Затримка: спілкування через мережу призводить до затримок, що впливає на швидкість відгуку програми, особливо в розподілених середовищах з високим мережевим трафіком.

Управління станами: керування станом клієнта та сеансу при виконанні декількох запитів може бути складним завданням, що вимагає використання таких механізмів, як сеанси, файли cookie або токени для підтримки стану.

Складність масштабованість сервера: забезпечення ефективної роботи сервера з великими обсягами одночасних запитів вимагає ретельного проєктування, розподілу ресурсів і стратегій масштабування.

2.5 Висновки до розділу 2

У цьому розділі було проведено проєктування архітектури бази даних. Розроблено модель її основних сутностей та зв’язків.

Було розглянуто основні положення UML діаграм та способів їх використання для демонстрації робочих процесів в середині баз даних, клієнтських частин застосунку та у використанні самими користувачами.

Було проведено моделювання структури сторінок вебзастосунку задля розуміння загальної архітектури сайти та планування ефективного та комфортного компонування сторінок та їх компонентів.

Після вивчення і аналізу існуючих архітектур програмних за стосунків було проведено проєктування архітектури поточного проєкту. До уваги були взяті специфіка проєкту, поставленні задачі, та обраний план реалізації.

Було проведено проєктування моделей для відображення користувацького досвіду для усвідомлення і аналізу правильності вибору минулих кроків.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБСЕРВІСУ ПЕРЕГЛЯДУ НОВИН З ЕЛЕМЕНТАМИ СОЦІАЛЬНОЇ МЕРЕЖІ

3.1 Операційна система використана під час розробки

Під час реалізації проєкту використовувався комп'ютер на операційній систему Linux Ubuntu.

Linux Ubuntu – популярний і широко використовуваний дистрибутив операційної системи Linux [33]. Його розробляє, підтримує та поширює британська компанія Canonical Ltd. разом із потужною спільнотою розробників з відкритим кодом. Ubuntu розроблена, щоб бути зручною, універсальною та доступною, що робить її придатною для широкого кола користувачів, від початківців до досвідчених розробників та системних адміністраторів.

Ubuntu базується на архітектурі Debian, відомій своєю стабільністю та надійністю. Вона дотримується принципів програмного забезпечення з відкритим вихідним кодом, надаючи користувачам вільний доступ до його вихідного коду і свободу модифікувати, поширювати і використовувати його на свій розсуд. При розробці Ubuntu основна увага приділяється простоті використання, регулярним оновленням та всебічній підтримці різноманітних апаратних платформ.

3.1.1 Ключові особливості Ubuntu

Інтерфейс: Ubuntu має чистий та інтуїтивно зрозумілий графічний інтерфейс користувача (GUI), в основному за замовчуванням використовує середовище робочого столу GNOME. Цей інтерфейс розроблений, щоб бути простим і ефективним, що робить його доступним для користувачів, які

переходять з інших операційних систем, таких як Windows або macOS.

Регулярні випуски та довгострокова підтримка (LTS): Ubuntu слідує передбачуваному графіку випуску: нові версії виходять кожні шість місяців у квітні та жовтні. Крім того, кожні два роки випускається версія LTS, яка пропонує розширену підтримку і обслуговування протягом п’яти років. LTS-версії надають перевагу в корпоративному середовищі завдяки їх стабільності та довгостроковим оновленням безпеки.

Широкі репозиторії програмного забезпечення: Ubuntu надає доступ до великих сховищ програмного забезпечення, що містять тисячі безкоштовних програм з відкритим кодом. Користувачі можуть легко встановлювати, оновлювати та керувати програмними пакетами за допомогою Advanced Package Tool (APT) та Ubuntu Software Center.

Безпека та оновлення: Ubuntu приділяє особливу увагу безпеці завдяки регулярним оновленням та виправленням. Дистрибутив містить вбудовані засоби безпеки, такі як брандмауер (UFW), AppArmor для захисту на рівні додатків, а також часті оновлення безпеки для захисту від вразливостей.

Кастомізація і гнучкість: Ubuntu дуже добре налаштовується, що дозволяє користувачам пристосовувати систему до своїх конкретних потреб. Користувачі можуть вибирати з різних середовищ робочого столу (наприклад, KDE, XFCE, LXDE) і встановлювати додаткове програмне забезпечення для розширення функціональності. Система підтримує широкий спектр апаратних архітектур, включаючи x86, ARM та PowerPC.

Спільнота та підтримка: Ubuntu має жваву та активну спільноту, яка надає обширну документацію, форуми та канали підтримки. Canonical пропонує комерційні варіанти підтримки для підприємств, які потребують професійної допомоги та технічної підтримки.

Ubuntu є кращим вибором для розробників та програмістів завдяки своїй сумісності з різними інструментами розробки, мовами програмування та фреймворками. Вона підтримує інтегровані середовища розробки (IDE), системи контролю версій та інструменти контейнеризації, такі як Docker.

3.2 Реалізація бази даних

Потрібно створити базу даних newspaper у середовищі PostgreSQL, для цього потрібно використати команду (див. рис. 3.1).

```
CREATE DATABASE newspaper;
```

Рисунок 3.1

Після створення бази даних потрібно підключитися до неї, для цього скористаємося командою (див. рис. 3.2).

```
\c newspaper
```

Рисунок 3.2

Далі потрібно створити потрібні таблиці.

3.2.1 Створення таблиці users

Таблиця користувачів (див. рис. 3.3) визначає структуру з різними полями для зберігання інформації про користувачів.

```
CREATE TABLE
  users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    likes INTEGER[] DEFAULT ARRAY[]::INTEGER[],
    dislikes INTEGER[] DEFAULT ARRAY[]::INTEGER[],
    com_likes INTEGER[] DEFAULT ARRAY[]::INTEGER[],
    com_dislikes INTEGER[] DEFAULT ARRAY[]::INTEGER[],
    image_data text[]
);
```

Рисунок 3.3 – Створення таблиці users

Детальний опис кожного поля в таблиці users.

Поле id, тип: SERIAL, атрибути: PRIMARY KEY. Опис: це поле є цілим числом, що автоматично інкрементується та унікально ідентифікує кожного користувача. Тип SERIAL автоматично генерує унікальні цілі значення, а обмеження PRIMARY KEY гарантує, що це значення буде унікальним для кожного запису.

Поле username, тип: VARCHAR(50), атрибути: UNIQUE NOT NULL. Опис: у цьому полі зберігається ім’я користувача з максимальною довжиною 50 символів. Обмеження UNIQUE гарантує, що два користувачі не можуть мати однакові імена, а обмеження NOT NULL гарантує, що це поле не може бути порожнім.

Поле password, тип: VARCHAR(255), атрибути: NOT NULL. Опис: у цьому полі зберігається хешований пароль користувача з максимальною довжиною 255 символів. Обмеження NOT NULL гарантує, що для кожного користувача має бути вказано пароль.

Поле first_name, тип: VARCHAR(50), атрибути: NOT NULL. Опис: у цьому полі зберігається ім’я користувача з максимальною довжиною 50 символів. Обмеження NOT NULL гарантує, що ім’я не може бути порожнім.

Поле last_name, тип: VARCHAR(50), атрибути: NOT NULL. Опис: у цьому полі зберігається прізвище користувача з максимальною довжиною 50 символів. Обмеження NOT NULL гарантує, що прізвище не може бути порожнім.

Поле email, тип: VARCHAR(100), атрибути: UNIQUE NOT NULL. Опис: у цьому полі зберігається адреса електронної пошти користувача з максимальною довжиною 100 символів. Обмеження UNIQUE гарантує, що жоден користувач не може мати однакову адресу електронної пошти, а обмеження NOT NULL гарантує, що це поле не може бути порожнім.

Поле likes, тип: INTEGER[], атрибути: DEFAULT ARRAY[]::INTEGER[]. Опис: у цьому полі зберігається масив цілих чисел, що представляють ідентифікатори новин, які користувач вподобав. Значення за замовчуванням – порожній цілочисельний масив.

Поле dislikes, тип: INTEGER[], атрибути: DEFAULT ARRAY[]::INTEGER[]. Опис: у цьому полі зберігається масив цілих чисел, що представляють ідентифікатори новин, які користувач не вподобав. Значення за замовчуванням – порожній цілочисельний масив.

Поле com_likes, тип: INTEGER[], атрибути: DEFAULT ARRAY[]::INTEGER[]. Опис: у цьому полі зберігається масив цілих чисел, що представляють ідентифікатори коментарів, які користувач вподобав. Значення за замовчуванням – порожній цілочисельний масив.

Поле com_dislikes, тип: INTEGER[], атрибути: DEFAULT ARRAY[]::INTEGER[]. Опис: у цьому полі зберігається масив цілих чисел, що представляють ідентифікатори коментарів, які користувач не вподобав. Значення за замовчуванням – порожній цілочисельний масив.

Поле image_data, тип: text[], атрибути: відсутні. Опис: у цьому полі зберігається масив текстових елементів, які можуть бути використані для зберігання різних даних про зображення, пов’язаних з користувачем. Для цього поля не визначено жодних додаткових обмежень.

3.2.2 Створення таблиці news

Таблиця новин (див. рис. 3.4) визначає структуру призначену для зберігання інформації про статті новин.

```
CREATE TABLE
  news (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    body TEXT NOT NULL,
    username VARCHAR(255) NOT NULL,
    image_data TEXT[],
    generation TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    likes INT DEFAULT 0,
    dislikes INT DEFAULT 0,
    FOREIGN KEY (username) REFERENCES users (username) ON DELETE CASCADE
);
```

Рисунок 3.4 – Створення таблиці news

Детальний опис кожного поля в таблиці news.

Поле id, тип: SERIAL, атрибути: PRIMARY KEY. Опис: це поле є цілим числом, що автоматично інкрементується та унікально ідентифікує кожну новину. Тип SERIAL автоматично генерує унікальні цілі значення, а обмеження PRIMARY KEY гарантує, що це значення буде унікальним для кожного запису.

Поле title, тип: TEXT, атрибути: NOT NULL. Опис: у цьому полі зберігається заголовок новини. Тип TEXT дозволяє зберігати рядки символів змінної довжини. Обмеження NOT NULL гарантує, що кожна новина повинна мати заголовок.

Поле body, тип: TEXT, атрибути: NOT NULL. Опис: у цьому полі зберігається основний зміст або тіло новини. Тип TEXT використовується для розміщення довгого контенту. Обмеження NOT NULL гарантує, що вміст тіла не може бути порожнім.

Поле username, тип: VARCHAR(255), атрибути: NOT NULL. Опис: у цьому полі зберігається ім’я користувача, який створив новину. Тип VARCHAR(255) дозволяє використовувати рядки символів змінної довжини до 255 символів. Обмеження NOT NULL гарантує, що автор статті завжди буде вказаний. Це поле також виконує роль зовнішнього ключа, пов’язуючи новину з користувачем у таблиці користувачів.

Поле image_data, тип: TEXT[], атрибути: відсутні. Опис: це поле зберігає масив текстових елементів, які представляють дані зображення, пов’язані з новинною статтею. Для цього поля не визначено додаткових обмежень, що дозволяє зберігати декілька зображень, пов’язаних зі статтею.

Поле generation, тип: TIMESTAMP, атрибути: NOT NULL DEFAULT CURRENT_TIMESTAMP. Опис: у цьому полі зберігається мітка часу, коли було створено новину. Тип TIMESTAMP записує інформацію про дату і час. Обмеження NOT NULL гарантує, що час створення завжди записується, а атрибут DEFAULT CURRENT_TIMESTAMP автоматично встановлює в полі поточну дату і час при додаванні нового запису.

Поле likes, тип: INT, атрибути: DEFAULT 0. Опис: у цьому полі

зберігається кількість позитивних оцінок (лайків), які отримала новина. Для цілочисельних значень використовується тип INT. Атрибут DEFAULT 0 ініціалізує це поле нулем при додаванні нового запису, забезпечуючи початковий підрахунок нульових вподобань.

Поле dislikes, тип: INT, атрибути: DEFAULT 0. Опис: у цьому полі зберігається кількість негативних оцінок (дизлайків), які отримала новина. Для цілих значень використовується тип INT. Атрибут DEFAULT 0 ініціалізує це поле нулем при додаванні нового запису, забезпечуючи початковий підрахунок нульових дизлайків.

3.2.3 Створення таблиці comments

Таблиця коментарів (див. рис. 3.5) визначає структуру призначену для зберігання інформації про коментарі до новинних статей.

```
CREATE TABLE
comments (
    id SERIAL PRIMARY KEY,
    news_id integer NOT NULL REFERENCES news (id) ON DELETE CASCADE,
    mention VARCHAR(50),
    username VARCHAR(50) NOT NULL REFERENCES users (username) ON DELETE CASCADE,
    body text NOT NULL,
    generation timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
    likes integer DEFAULT 0,
    dislikes integer DEFAULT 0
);
```

Рисунок 3.5 – Створення таблиці comments

Детальний опис кожного поля таблиці коментарів.

Поле id, тип: SERIAL, атрибути: PRIMARY KEY. Опис: це поле є цілим числом, що автоматично інкрементується та унікально ідентифікує кожен коментар. Тип SERIAL автоматично генерує унікальні цілі значення, а обмеження PRIMARY KEY гарантує, що це значення буде унікальним для кожного запису.

Поле news_id, тип: INTEGER, атрибути: NOT NULL REFERENCES news (id) ON DELETE CASCADE. Опис: у цьому полі зберігається ідентифікатор новини, до якої відноситься коментар. Для цілих значень використовується тип INTEGER. Обмеження NOT NULL гарантує, що кожен коментар буде пов’язаний з конкретною новинною статтею. Речення REFERENCES news (id) встановлює зв’язок зовнішнього ключа з полем id в таблиці новин, забезпечуючи посилальну цілісність. Речення ON DELETE CASCADE гарантує, що при видаленні новини всі пов’язані з нею коментарі також будуть видалені.

Поле mention, тип: VARCHAR(50), атрибути: відсутні. Опис: у цьому полі зберігається ім’я іншого користувача, згаданого в коментарі, якщо таке є. Тип VARCHAR(50) дозволяє використовувати рядки символів змінної довжини до 50 символів. На це поле не накладається ніяких додаткових обмежень, тому воно може бути необов’язковим.

Поле username, тип: VARCHAR(50), атрибути: NOT NULL REFERENCES users (username) ON DELETE CASCADE. Опис: у цьому полі зберігається ім’я користувача, який написав коментар. Тип VARCHAR(50) дозволяє використовувати рядки символів змінної довжини до 50 символів. Обмеження NOT NULL гарантує, що кожен коментар повинен мати асоційованого автора. Речення REFERENCES users (username) встановлює зв’язок зовнішнього ключа з полем username у таблиці users, забезпечуючи цілісність посилань. Речення ON DELETE CASCADE гарантує, що якщо користувача буде видалено, всі коментарі, написані цим користувачем, також будуть видалені.

Поле body, тип: TEXT, атрибути: NOT NULL. Опис: у цьому полі зберігається основний зміст коментаря. Тип TEXT дозволяє використовувати рядки символів змінної довжини, що підходить для довгого вмісту. Обмеження NOT NULL гарантує, що тіло коментаря не може бути порожнім.

Поле generation, тип: TIMESTAMP, атрибути: NOT NULL DEFAULT CURRENT_TIMESTAMP. Опис: це поле зберігає мітку часу, коли було створено коментар. Тип TIMESTAMP записує інформацію про дату і час.

Обмеження NOT NULL гарантує, що час створення завжди записується, а атрибут DEFAULT CURRENT_TIMESTAMP автоматично встановлює в полі поточну дату і час при додаванні нового запису.

Поле likes, тип: INTEGER, атрибути: DEFAULT 0. Опис: у цьому полі зберігається кількість позитивних оцінок (лайків), які отримав коментар. Для цілих значень використовується тип INTEGER. Атрибут DEFAULT 0 ініціалізує це поле нулем при додаванні нового запису, забезпечуючи початковий підрахунок нульових вподобань.

Поле dislikes, тип: INTEGER, атрибути: DEFAULT 0. Опис: у цьому полі зберігається кількість негативних оцінок (дизлайків), які отримав коментар. Для цілих значень використовується тип INTEGER. Атрибут DEFAULT 0 ініціалізує це поле нулем при додаванні нового запису, забезпечуючи початковий підрахунок нульових дизлайків.

3.2.4 Створення таблиці notifications

Таблиця сповіщень (див. рис. 3.6) визначає структуру призначену для зберігання інформації про сповіщення, пов’язані з новинними статтями та взаємодією з користувачами.

```
CREATE TABLE notifications (
  id SERIAL PRIMARY KEY,
  body TEXT NOT NULL,
  mention VARCHAR(50),
  com_body TEXT,
  news_id INTEGER NOT NULL REFERENCES news (id) ON DELETE CASCADE,
  generation TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  username VARCHAR(255) REFERENCES users (username) ON DELETE CASCADE,
  is_read BOOLEAN NOT NULL DEFAULT FALSE
);
```

Рисунок 3.6 – Створення таблиці notifications

Детальний опис кожного поля таблиці сповіщень.

Поле id, тип: SERIAL, Атрибути: PRIMARY KEY. Опис: це поле є цілим числом, що автоматично інкрементується та унікально ідентифікує

кожне сповіщення. Тип SERIAL автоматично генерує унікальні цілі значення, а обмеження PRIMARY KEY гарантує, що це значення буде унікальним для кожного запису.

Поле body, тип: TEXT, атрибути: NOT NULL. Опис: у цьому полі зберігається основний зміст повідомлення. Тип TEXT дозволяє передавати рядки символів змінної довжини. Обмеження NOT NULL гарантує, що кожне повідомлення повинно мати тіло.

Поле mention, тип: VARCHAR(50), атрибути: відсутні. Опис: у цьому полі зберігається ім’я іншого користувача, згаданого у сповіщенні, якщо таке є. Тип VARCHAR(50) дозволяє використовувати рядки символів змінної довжини до 50 символів. На це поле не накладається жодних додаткових обмежень, тому воно є необов’язковим.

Поле com_body, тип: TEXT, атрибути: відсутні. Опис: у цьому полі зберігається текст коментаря, пов’язаного зі сповіщенням, якщо це можливо. Тип TEXT дозволяє вводити рядки символів змінної довжини. На це поле не накладається жодних додаткових обмежень, тому воно є необов’язковим.

Поле news_id, тип: INTEGER, атрибути: NOT NULL REFERENCES news (id) ON DELETE CASCADE. Опис: у цьому полі зберігається ідентифікатор новини, пов’язаної зі сповіщенням. Для цілих значень використовується тип INTEGER. Обмеження NOT NULL гарантує, що кожне сповіщення буде пов’язане з конкретною новинною статтею. Речення REFERENCES news (id) встановлює зв’язок зовнішнього ключа з полем id у таблиці новин, забезпечуючи посилальну цілісність. Речення ON DELETE CASCADE гарантує, що при видаленні новини всі пов’язані з нею сповіщення також будуть видалені.

Поле generation, тип: TIMESTAMP, атрибути: NOT NULL DEFAULT CURRENT_TIMESTAMP. Опис: це поле зберігає мітку часу, коли було створено сповіщення. Тип TIMESTAMP записує інформацію про дату і час. Обмеження NOT NULL гарантує, що час створення завжди записується, а атрибут DEFAULT CURRENT_TIMESTAMP автоматично встановлює в полі поточну дату і час при додаванні нового запису.

Поле username, тип: VARCHAR(255), атрибути: REFERENCES users (username) ON DELETE CASCADE. Опис: у цьому полі зберігається ім’я користувача, якому адресовано сповіщення. Тип VARCHAR(255) дозволяє використовувати рядки символів змінної довжини до 255 символів. Речення REFERENCES users (username) встановлює зв’язок зовнішнього ключа з полем username в таблиці users, забезпечуючи посилальну цілісність. Речення ON DELETE CASCADE гарантує, що якщо користувача буде видалено, то всі сповіщення, адресовані цьому користувачеві, також будуть видалені.

Поле is_read, тип: BOOLEAN, атрибути: NOT NULL DEFAULT FALSE. Опис: це поле вказує на те, чи було прочитано повідомлення користувачем. Тип BOOLEAN зберігає значення TRUE або FALSE. Обмеження NOT NULL гарантує, що це поле завжди встановлено, а атрибут DEFAULT FALSE ініціалізує це поле значенням FALSE при додаванні нового запису, вказуючи на те, що сповіщення за замовчуванням не прочитано.

3.2.5 Створення таблиці admins

Таблиця admins (див. рис. 3.7) визначає структуру призначену для зберігання інформації про адміністраторів.

```
CREATE TABLE
  admins (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    image_data text[]
);
```

Рисунок 3.7 – Створення таблиці admins

Детальний опис кожного поля таблиці admins.

Поле id, тип: SERIAL, атрибути: PRIMARY KEY. Опис: це поле є цілим числом, що автоматично інкрементується та унікально ідентифікує

кожного адміністратора. Тип SERIAL автоматично генерує унікальні цілі значення, а обмеження PRIMARY KEY гарантує, що це значення буде унікальним для кожного запису.

Поле username, тип: VARCHAR(50), атрибути: UNIQUE NOT NULL. Опис: у цьому полі зберігається ім’я користувача адміністратора з максимальною довжиною 50 символів. Обмеження UNIQUE гарантує, що два адміністратори не можуть мати однакове ім’я користувача, а обмеження NOT NULL гарантує, що це поле не може бути порожнім.

Поле password, тип: VARCHAR(255), атрибути: NOT NULL. Опис: у цьому полі зберігається хешований пароль адміністратора з максимальною довжиною 255 символів. Обмеження NOT NULL гарантує, що для кожного адміністратора має бути вказано пароль.

Поле email, тип: VARCHAR(100), атрибути: UNIQUE NOT NULL. Опис: у цьому полі зберігається адреса електронної пошти адміністратора з максимальною довжиною 100 символів. Обмеження UNIQUE гарантує, що жоден адміністратор не може мати однакову адресу електронної пошти, а обмеження NOT NULL гарантує, що це поле не може бути порожнім.

Поле image_data, тип: TEXT[], атрибути: відсутні. Опис: у цьому полі зберігається масив текстових елементів, які можуть бути використані для зберігання різних даних зображення, пов’язаних з адміністратором. Для цього поля не визначено додаткових обмежень, що дозволяє зберігати декілька зображень, пов’язаних з адміністратором.

Для автоматичного створення бази даних та таблиць використовувався міні проєкт на Node.js. У проєкті була можливість створення всіх потрібних таблиць та заповнення їх тестовими даними.

3.3 Реалізація серверної частини

Після створення бази даних була створена серверна частина застосунку у якій відбувався зв’язок із базою даних та надання можливостей робити

запити до бази даних (додаток).

Основна частина роботи з запитами відбувалася у API вебзастосунку.

Була реалізована обробка клієнтських запитів. API обробляє запити, надіслані клієнтами та визначає відповідні дій на основі методу запиту (наприклад, GET, POST, PUT, DELETE).

взаємодіє з базою даних для виконання операцій CRUD (створення, читання, оновлення, видалення). Воно забезпечує цілісність, узгодженість і безпеку даних, керуючи з'єднаннями, виконуючи запити та обробляючи транзакції.

API керує автентифікацією (перевіркою ідентичності користувача) та авторизацією (наданням або забороною доступу до ресурсів на основі ролей і дозволів користувача). Це важливо для захисту конфіденційних даних і забезпечення доступу до певних функцій лише авторизованим користувачам.

3.3.1 Статус відповідей від сервера

У контексті вебсерверів розуміння статусу відповідей має вирішальне значення для діагностики проблем, підвищення продуктивності та забезпечення надійності вебдодатків. Відповіді сервера передаються за допомогою кодів статусу HTTP, які дають уявлення про результат виконання клієнтського запиту.

Інформаційні відповіді (100-199). Ці коди вказують на те, що сервер отримав запит і продовжує його обробку. 100 – продовжити [27]. Сервер отримав заголовки запиту, і клієнт повинен продовжити відправку тіла запиту. 101 – перемикання протоколів. Запитувач попросив сервер переключити протокол, і сервер підтверджує, що він це зробить.

Успішні відповіді (200-299). Ці коди означають, що запит клієнта був успішно отриманий, зрозумілий і прийнятий. 200 – ОК [27]. Запит успішно виконано. Значення успіху залежить від методу HTTP. 201 – створено. Запит було виконано, в результаті чого було створено новий ресурс. 204 – немає

вмісту. Сервер успішно обробив запит, але не повертає ніякого вмісту.

Повідомлення про перенаправлення (300-399). Ці коди вказують на те, що для завершення запиту користувачеві необхідно виконати подальші дії з боку агента користувача. 301 – переміщено назавжди. Ресурс назавжди переміщено за новою URL-адресою. 302 – знайдено. Ресурс тимчасово знаходиться за іншою URL-адресою. 304 – не змінено. Ресурс не був змінений з версії, вказаної в заголовках запиту.

Відповіді про помилки клієнта (400-499) [27]. Ці коди вказують на те, що запит містить неправильний синтаксис або не може бути виконаний. 400 – неправильний запит [27]. Сервер не може обробити запит через помилку клієнта (наприклад, неправильний синтаксис запиту). 401 – неавторизований. Потрібна автентифікація, яка не пройдена або ще не надана. 403 – заборонено. Сервер зрозумів запит, але відмовляється його авторизувати. 404 – не знайдено. Сервер не може знайти запитуваний ресурс.

Відповіді про помилки сервера (500-599). Ці коди вказують на те, що сервер не зміг виконати дійсний запит. 500 – внутрішня помилка сервера [27]. Загальне повідомлення про помилку, коли сервер стикається з неочікуваним станом. 502 – несправний шлюз. Сервер, виконуючи роль шлюзу або проксі-сервера, отримав недійсну відповідь від вхідного сервера. 503 – сервіс недоступний. Сервер наразі недоступний (перевантажений або перебуває на технічному обслуговуванні). 504 – таймаут шлюзу. Сервер, виконуючи роль шлюзу або проксі-сервера, не отримав своєчасної відповіді від висхідного сервера.

3.3.2 Запити до сервера

POST-запит UserAuth призначений для автентифікації користувачів, що дозволяє користувачам підтвердити свої облікові дані та отримати доступ до системи. Запит перевіряє, чи існує в системі користувач з наданою комбінацією email/username та password. Тіло запиту містить облікові дані

користувача, зазвичай у форматі JSON, включаючи адресу електронної пошти або ім’я користувача та відповідний пароль. Виконує SQL-запити, щоб перевірити, чи існує користувач з вказаною адресою електронної пошти/іменем користувача та паролем. Якщо користувач з наданими обліковими даними знайдений, повертає відповідь у форматі JSON з деталями користувача і кодом статусу 200, що вказує на успіх. Якщо користувач вже існує або надані облікові дані невірні, повертає повідомлення про помилку, що вказує на невдалу автентифікацію з кодом стану 500.

DELETE-запит UserAuth призначений для видалення облікового запису користувача з системи. Запит дозволяє адміністраторам або користувачам з відповідними правами видаляти облікові записи користувачів з системи. Тіло запиту зазвичай містить ім’я користувача, обліковий запис якого потрібно видалити. Виконує запит SQL DELETE для видалення запису користувача з таблиці users на основі наданого ім’я користувача. Якщо обліковий запис користувача успішно видалено, повертає відповідь у форматі JSON, що підтверджує видалення з кодом стану 200. Якщо облікового запису користувача не існує, повертає повідомлення про помилку, що вказує на те, що користувач не існує, з кодом статусу 500.

POST-запит UserCreate призначений для обробки створення нового користувача в базі даних. Запит перевіряє, чи існує користувач із заданим іменем користувача або електронною поштою. Якщо такого користувача не існує, він вставляє нового користувача в базу даних. Залежно від результату цих операцій, він відповідає відповідними повідомленнями та кодами статусу. Тіло запиту має містити усі дані користувача. Код викликає функцію UserCheck, щоб визначити, чи існує користувач з вказаним іменем користувача або адресою електронної пошти. Функція UserCheck приймає дані і пароль в якості параметрів для перевірки бази даних на наявність існуючого користувача. Якщо користувач існує, функція повертає статус 200. Виконує запит SQL INSERT для додавання даних нового користувача до таблиці користувачів у базі даних. Якщо надано зображення профілю, включає його до запиту на вставку в базу даних. Якщо користувача не існує,

функція дозволяє продовжити процес створення. Якщо новий обліковий запис користувача створено успішно, повертає відповідь у форматі JSON з кодом статусу 200 для позначення успіху.

POST-запит UserEdit призначений для зміни існуючих даних облікового запису користувача в системі. Запит надає користувачам можливість редагувати та оновлювати інформацію свого профілю за потреби. Тіло запиту має містити повну нову інформацію користувача та старе ім’я користувача, поточне ім’я користувача облікового запису, що редагується. Виконує запит SQL UPDATE для зміни даних облікового запису користувача в таблиці користувачів на основі наданого імені користувача. Якщо надано оновлене зображення профілю, включає його в запит на оновлення бази даних. Якщо дані облікового запису користувача успішно оновлено, повертає відповідь у форматі JSON зі значенням true і кодом статусу 200 для позначення успіху. Якщо операція оновлення завершилася невдало або жодних змін не було внесено, повертає false.

POST-запит NewsGet призначений для отримання новинних статей на основі різних критеріїв, таких як пагінація, пошукові запити та фільтрація за іменем користувача. Запит дозволяє користувачам переглядати статті новин з підтримкою посторінкової нумерації, шукати певні статті за ключовими словами та фільтрувати статті за іменем автора. Тіло запиту має налічувати номер сторінки новин, які потрібно отримати (для пагінації) максимальну кількість новин на сторінці, ключові слова для пошуку в заголовках, тілах або іменах користувачів (необов’язково), ім’я автора для фільтрування новинних статей. Створює SQL-запит для отримання статей новин з таблиці новин на основі наданих параметрів. Обробляє пагінацію, вказуючи потрібну сторінку і обмеження в SQL-запиті. Підтримує пошук новин, що містять певні ключові слова в заголовку, тілі або імені користувача. Опціонально фільтрує статті новин за іменем користувача, якщо його вказано. При успішній операції повертає відповідь у форматі JSON, що містить знайдені статті новин, а також інформацію про пагінацію, наприклад, загальну

кількість сторінок. Повертає код статусу 200 для позначення успішної операції.

POST-запит NewsCreate призначений для полегшення створення нових статей новин у додатку. Він обробляє вхідні запити, що містять інформацію про новину, яку потрібно створити, таку як заголовок, основний вміст, ім’я користувача автора і, за бажанням, дані зображення. Тіло запиту має налічувати заголовок новини, вміст тіла новини, ім’я користувача, який створив новину, дані зображення, пов’язані із зображенням новини (необов’язково). Створює SQL-запит для вставки нового запису в таблицю новин бази даних. Вставляє надані заголовок, тіло, ім’я користувача і, за бажанням, дані зображення у відповідні стовпці таблиці новин. Повертає відповідь у форматі JSON, яка вказує на успішність або неуспішність процесу створення новини. Якщо новина створена успішно, повертає JSON повідомлення з кодом статусу 200. Якщо під час процесу виникає помилка, повертає JSON з відповідним повідомленням про помилку і кодом статусу 500.

POST-запит NewsLike призначений для керування вподобаннями і не вподобаннями новин на основі взаємодії користувачів. Запит спочатку перевіряє, чи існує ідентифікатор у списку новин зі статусом like у запису користувача. Якщо ні, то відбувається перевірка на те чи існує ідентифікатор у списку новин зі статусом dislike у запису користувача, якщо такої оцінки користувач не надавав то статус like додається на новину а ідентифікатор новини вноситься до списку вподобаних новин у запису користувача. Якщо оцінка dislike все ж є, то спочатку її статус прибирається з новини, потім видаляється ідентифікатор зі списку новин зі статусом dislike у запису користувача, та відбувається виставлення статусу like. Тіло запиту має налічувати унікальний ідентифікатор коментаря до якого потрібно застосувати статус like, псевдонім користувача, який виконує дію виставлення статусу like. Якщо користувач успішно додав статус like або прибрав dislike до коментаря то буде надіслано успішну відповідь у форматі JSON. Будь-які помилки, що виникають під час виконання операції,

реєструються, і повертається повідомлення про помилку разом з кодом стану 400.

POST-запит NewsDislLike призначений для керування вподобаннями і не вподобаннями новин на основі взаємодії користувачів. Запит спочатку перевіряє, чи існує ідентифікатор у списку новин зі статусом dislike у запису користувача. Якщо ні, то відбувається перевірка на те чи існує ідентифікатор у списку новин зі статусом like у запису користувача, якщо такої оцінки користувач не надавав то статус dislike додається на новину а ідентифікатор новини вноситься до списку вподобаних новин у запису користувача. Якщо оцінка like все ж є, то спочатку її статус прибирається з новини, потім видаляється ідентифікатор зі списку новин зі статусом like у запису користувача, та відбувається виставлення статусу dislike. Тіло запиту має налічувати унікальний ідентифікатор коментаря до якого потрібно застосувати статус dislike, псевдонім користувача, який виконує дію виставлення статусу dislike. Якщо користувач успішно додав статус dislike або прибрав like до коментаря то буде надіслано успішну відповідь у форматі JSON. Будь-які помилки, що виникають під час виконання операції, реєструються, і повертається повідомлення про помилку разом з кодом стану 400.

POST-запит OneNewsGet призначений для отримання детальної інформації про конкретну новину за її унікальним ідентифікатором. Запит отримує всі стовпці даних для однієї новини за умови, що її ідентифікатор відомий і дійсний. Тіло запиту має налічувати унікальний ідентифікатор новини яку потрібно отримати. Функція виконує SQL-запит до таблиці новин, щоб отримати рядок, який відповідає наданому ідентифікатору. Якщо стаття з вказаним id існує в базі даних, функція повертає дані статті з кодом статусу 200. Якщо стаття не відповідає заданому ідентифікатору, функція повертає повідомлення про помилку, що стаття не існує, з кодом статусу 500. Будь-які помилки, що виникають під час процесу, такі як проблема з’єднання з базою даних або помилки SQL, відловлюються, записуються в журнал, і разом з повідомленням про помилку повертається код статусу 400.

POST-запит OneNewsEdit призначений для оновлення певних деталей існуючої новини, що ідентифікується її унікальним ідентифікатором. Запит дозволяє користувачам змінювати існуючі записи в базі даних новин, оновлюючи такі поля, як заголовок, основний текст і пов'язані з ним дані зображень, якщо вони надані. Тіло запиту має налічувати унікальний ідентифікатор новини яку потрібно відредагувати, новий заголовок для новини, новий текст новини, дані нового зображення, якщо наявне зображення потрібно оновити або замінити (необов'язково). Виконує запит SQL UPDATE до таблиці новин. Запит на оновлення змінює поля заголовку, тіла новини і, за бажанням, дані зображення, де ідентифікатор збігається з вказаною новинною статтею. Якщо запит успішно оновив рядок, він повертає відповідь у форматі JSON із зазначенням успіху (edit_news: true) разом із кодом стану 200. Якщо в результаті виконання запиту не було оновлено жодного рядка (наприклад, не знайдено жодної статті з заданим ID або не було внесено жодних змін до даних), повертається відповідь про відсутність змін (edit_news: false) з кодом стану 500. Фіксує і реєструє будь-які винятки, що виникають під час виконання процесу оновлення, наприклад, проблеми з підключенням до бази даних. У таких випадках повертає відповідь у форматі JSON з відповідним повідомленням про помилку і кодом стану 400.

DELETE-запит OneNewsDelete призначений для видалення конкретної новини з бази даних за її унікальним ідентифікатором. Дозволяє адміністраторам або користувачам з відповідними правами назавжди видалити новину на основі її ідентифікатора. Тіло запиту має налічувати унікальний ідентифікатор новини яку потрібно видалити. Виконує інструкцію SQL DELETE для таблиці новин, де id збігається з наданим ідентифікатором. Якщо запит успішно видаляє рядок (тобто стаття існувала і була видалена), він повертає відповідь у форматі JSON, що свідчить про успіх (delete_news: true) разом з кодом статусу 200. Якщо стаття не знайдена або вже, повертає відповідь про невдачу з повідомленням про помилку і кодом статусу 500. Будь-які винятки або помилки під час процесу, такі як проблеми з підключенням до бази даних, відловлюються, реєструються і

повертаються у відповіді з відповідним повідомленням про помилку і кодом стану 400.

POST-запит GradesGet призначеною для отримання кількості like і dislike для конкретної новини, за її унікальним ідентифікатором. Запит надає можливість отримати оцінку новини. Тіло запиту має налічувати унікальний ідентифікатор новини. Функція виконує операцію SQL SELECT над таблицею новин, щоб отримати рядок, що відповідає заданому ідентифікатору. Витягує like і dislike з вибраного рядка для відповіді. Якщо стаття існує і рядок успішно знайдено, функція повертає кількість вподобань і дизлайків разом з кодом статусу 200. Якщо для вказаного ідентифікатора не знайдено відповідної статті, повертається повідомлення про помилку з кодом статусу 500. Функція відловлює і реєструє будь-які помилки виконання, повертаючи загальне повідомлення про помилку з кодом стану 400, що зазвичай вказує на неправильний запит або інші помилки на стороні клієнта.

POST-запит CommentsGet призначеною для отримання коментарів з бази даних. Запит надає можливість отримання коментарів з бази даних на основі заданих фільтрів, які включають ідентифікатор новинної статті, ім’я користувача-коментатора, деталі пагінації та пошуковий термін, який може відповідати як тілу коментаря, так і згадкам. Тіло запиту має налічувати ідентифікатор новини для якої запитуються коментарі, псевдонім користувача який створив коментар, номер сторінки в пагінації, кількість коментарів на сторінці, пошуковий термін якому мають відповідати коментарі або в тілі, або в згадках. Починається з базового SQL-запиту SELECT * FROM comments. Динамічно створює умови SQL на основі наявності та комбінації параметрів для точного фільтрування коментарів. Застосовує впорядкування за ідентифікатором коментаря і пагінацію, якщо вказано. Без параметрів пагінації за замовчуванням застосовується впорядкування за ідентифікатором коментаря. Спочатку виконує запит підрахунку, щоб визначити загальну кількість коментарів, які відповідають критеріям фільтрації, що є важливим для пагінації. Виконує основний запит, створений для отримання фактичних даних про коментарі. Обчислює

загальну кількість сторінок на основі підрахунку та ліміту. Якщо запит успішний то товертає коментарі, загальну кількість і кількість сторінок зі статусом 200. Якщо запит помилковий то реєструє помилки і повертає повідомлення про помилку зі статусом 400, що вказує на проблеми на стороні клієнта або помилки в запиті.

POST-запит CommentsPost призначений для створення нових коментарів у базі даних, пов’язаних з конкретними статтями новин. Запит дозволяє користувачам залишати коментарі до новинних статей та керувати сповіщеннями про згадки в цих коментарях. Тіло запиту має налічувати: ідентифікатор новини до якої публікується коментар, ім’я користувача, який безпосередньо згадується в коментарі (необов’язково), ім’я користувача, який залишив коментар, текст коментаря. Якщо в коментарі є згадка, система не тільки записує коментар, але й створює повідомлення про те, що згаданому користувачеві було надіслано відповідь. Якщо в коментарі немає згадки, система записує коментар і перевіряє, чи стосується він новинної статті. Якщо так, то система надсилає повідомлення автору новини про те, що було розміщено новий коментар. Якщо коментар успішно опублікований (і будь-які сповіщення обробляються без проблем), буде надіслано успішну відповідь у форматі JSON. Якщо під час додавання коментарів виникають проблеми, вони фіксуються і повертається відповідне повідомлення про помилку. Помилки під час створення сповіщень також фіксуються і записуються в журнал, після чого повертається відповідь про помилку.

POST-запит CommentsLike призначений для керування вподобаннями і не вподобаннями коментарів на основі взаємодії користувачів. Запит спочатку перевіряє, чи існує ідентифікатор у списку коментарів зі статусом like у запису користувача. Якщо ні, то відбувається перевірка на те чи існує ідентифікатор у списку коментарів зі статусом dislike у запису користувача, якщо такої оцінки користувач не надавав то статус like додається на коментар а ідентифікатор коментаря вноситься до списку вподобаних коментарів у запису користувача. Якщо оцінка dislike все ж є, то спочатку її статус прибирається з коментаря, потім видаляється ідентифікатор зі списку

коментарів зі статусом dislike у запису користувача, та відбувається виставлення статусу like. Тіло запиту має налічувати дані унікальний ідентифікатор коментаря, до якого потрібно застосувати статус like, псевдонім користувача, який виконує дію виставлення статусу like. Якщо користувач успішно додав статус like або прибрав dislike до коментаря то буде надіслано успішну відповідь у форматі JSON. Будь-які помилки, що виникають під час виконання операції, реєструються, і повертається повідомлення про помилку разом з кодом стану 400.

POST-запит CommentsDislike призначений для керування не вподобаннями і вподобаннями коментарів на основі взаємодії користувачів. Запит спочатку перевіряє, чи існує ідентифікатор у списку коментарів зі статусом dislike у запису користувача. Якщо ні, то відбувається перевірка на те чи існує ідентифікатор у списку коментарів зі статусом like у запису користувача, якщо такої оцінки користувач не надавав то статус dislike додається на коментар а ідентифікатор коментаря вноситься до списку не вподобаних коментарів у запису користувача. Якщо оцінка like все ж є, то спочатку її статус прибирається з коментаря, потім видаляється ідентифікатор зі списку коментарів зі статусом like у запису користувача, та відбувається виставлення статусу dislike. Тіло запиту має налічувати: унікальний ідентифікатор коментаря, до якого потрібно застосувати статус dislike, псевдонім користувача, який виконує дію виставлення статусу dislike. Якщо користувач успішно додав статус dislike або прибрав like до коментаря то буде надіслано успішну відповідь у форматі JSON. Будь-які помилки, що виникають під час виконання операції, реєструються, і повертається повідомлення про помилку разом з кодом стану 400.

POST-запит CommentsGradesGet призначеною для отримання кількості like і dislike для конкретного коментаря, за його унікальним ідентифікатором (id). Запит надає можливість отримати оцінку коментаря. Тіло запиту має налічувати: унікальний ідентифікатор коментаря. Функція виконує операцію SQL SELECT над таблицею коментарів, щоб отримати рядок, що відповідає заданому id. Витягує like і dislike з вибраного рядка для відповіді. Якщо

коментар існує і рядок успішно знайдено, функція повертає кількість like і dislike разом з кодом статусу 200. Якщо для вказаного ідентифікатора не знайдено відповідної статті, повертається повідомлення про помилку з кодом статусу 500. Функція відловлює і реєструє будь-які помилки виконання, повертаючи загальне повідомлення про помилку з кодом стану 400, що зазвичай вказує на неправильний запит або інші помилки на стороні клієнта.

DELETE-запит CommentsDelete призначений для видалення конкретного коментаря з бази даних за його унікальним ідентифікатором (id). Дозволяє адміністраторам або користувачам з відповідними правами назавжди видалити коментар на основі його ідентифікатора. Тіло запиту має налічувати: унікальний ідентифікатор коментаря. Виконує інструкцію SQL DELETE для таблиці коментарів, де id збігається з наданим ідентифікатором. Якщо запит успішно видаляє коментар, він повертає відповідь у форматі JSON, що свідчить про успіх разом з кодом статусу 200. Якщо стаття не знайдена або вже, повертає відповідь про невдачу з повідомленням про помилку і кодом статусу 500. Будь-які винятки або помилки під час процесу, такі як проблеми з підключенням до бази даних, відловлюються, реєструються і повертаються у відповіді з відповідним повідомленням про помилку і кодом стану 400.

POST-запит NotiGet призначена для отримання сповіщень з бази даних на основі певних фільтрів, таких як ім’я користувача, налаштування пагінації та пошукові терміни. Запит дає можливість отримати сповіщення з бази даних, відфільтровані на основі критеріїв, наданих користувачем. Тіло запиту має налічувати: псевдонім користувача для якого отримуються сповіщення, номер сторінки в пагінації, максимальну кількість сповіщень для кожної сторінки, пошуковий термін який використовується для фільтрації сповіщень на основі їхнього вмісту. Динамічно створює SQL-запит на основі наданих фільтрів. Використовує умови, такі як збіг імені користувача та додаткові пошукові терміни, щоб звузити список сповіщень. Виконує SELECT-запит до таблиці сповіщень із заданими параметрами фільтрації. Додатково виконує запит підрахунку, щоб визначити загальну кількість сповіщень, які

відповідають критеріям фільтрації, що важливо для пагінації. Обчислює загальну кількість сторінок на основі підрахунку сповіщень і вказаного ліміту на сторінку. Якщо сповіщення знайдено на основі критеріїв фільтрації, повертає сповіщення разом із кількістю та деталями пагінації у відповіді JSON із кодом стану 200. Якщо сповіщення не знайдено, повертає повідомлення про помилку, що вказує на їхню відсутність, з кодом стану 500. Реєструє будь-які помилки, що виникають під час процесу, і повертає відповідну відповідь про помилку з кодом статусу 400 для проблем на стороні клієнта або поганих запитів.

POST-запит NotiRead призначена для позначення повідомлень як прочитаних або непрочитаних на основі наданих параметрів. Запит дозволяє користувачам керувати статусом своїх сповіщень, позначаючи їх як прочитані або непрочитані за бажанням. Тіло запиту має налічувати: унікальний ідентифікатор сповіщення яке потрібно позначити як прочитане або непрочитане, псевдонім користувача для якого буде оновлено статус сповіщення. Виконує запит SELECT для отримання сповіщення на основі наданого ідентифікатора та імені користувача. Це гарантує, що сповіщення існує і належить вказаному користувачеві. Оновлює поле is_read сповіщення за допомогою SQL UPDATE запиту, перемикаючи його значення між прочитаним і непрочитаним. Якщо сповіщення успішно знайдено і його статус оновлено, повертає відповідь у форматі JSON зі значенням true для позначення успіху. Якщо сповіщення або користувача не знайдено, або якщо під час процесу виникла помилка, повертає false або повідомлення про помилку відповідно.

DELETE-запит NotiDelete призначений для видалення сповіщень на основі наданих параметрів. Запит надає користувачеві можливість видаляти сповіщення зі свого списку сповіщень за потребою. Тіло запиту має налічувати: унікальний ідентифікатор сповіщення яке потрібно видалити,псевдонім користувача для якого буде видалено сповіщення. Виконує SELECT-запит для перевірки існування сповіщення на основі наданих ідентифікатора та імені користувача. Якщо сповіщення існує для

вказаного користувача, виконує запит DELETE, щоб видалити його з бази даних. Якщо сповіщення успішно знайдено і видалено, повертає відповідь у форматі JSON зі значенням true, що вказує на успіх. Якщо сповіщення або користувача не знайдено, або якщо під час видалення виникла помилка, повертає false або повідомлення про помилку відповідно.

Структура директорій API наведена на рисунку 3.8.

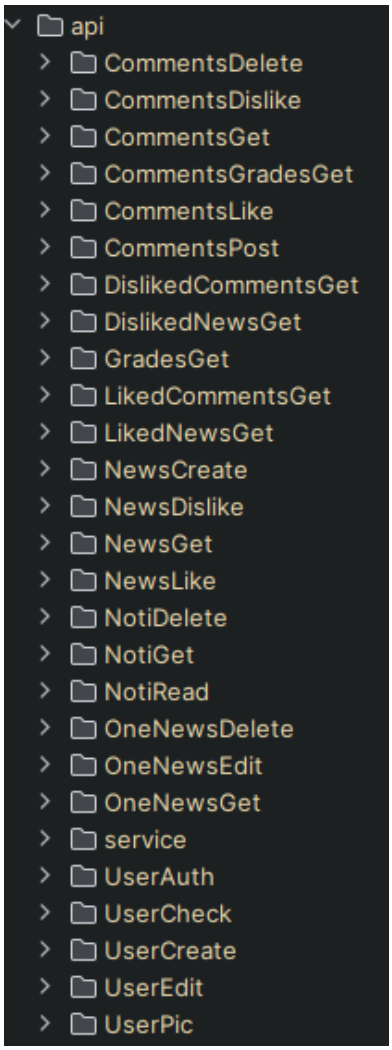

Рисунок 3.8 – Структура директорій API

3.4 Реалізація запитів до API

Далі було реалізовано клієнтську частину у якій були реалізовані fetch запити до серверного API.

Fetch у контексті веброзробки та комп’ютерного програмування означає API (Application Programming Interface), який надається сучасними веббраузерами для здійснення мережевих запитів з метою отримання ресурсів з сервера [28]. Це основний інструмент для створення динамічних вебдодатків, які взаємодіють з вебсерверами та API для отримання даних.

3.4.1 Основні характеристики

Promise-Based: Fetch побудовано на JavaScript Promises, що дозволяє легше керувати асинхронними операціями порівняно зі старими методами, такими як XMLHttpRequest [28]. Це означає, що операціями fetch можна керувати за допомогою методів .then() і .catch() або сучасного синтаксису async/await.

Спрощений синтаксис: синтаксис API fetch простіший і чистіший, що полегшує читання і написання коду мережевих запитів.

Перехресні запити: Fetch безпечніше обробляє перехресні запити (запити, зроблені до іншого домену). Він включає підтримку протоколу Cross-Origin Resource Sharing (CORS), який допомагає керувати дозволами на доступ до ресурсів, розміщених на різних серверах [28].

Потоки: Fetch підтримує читання даних у вигляді потоків, що дозволяє ефективніше обробляти великі обсяги даних, не завантажуючи весь ресурс в пам’ять.

Спільне налаштування Fetch функцій. Все POST функції мають однакові налаштування у поточному проєкті. Функція надсилає POST-запит на кінцеву точкуURL.

Запит налаштовано так, щоб обійти будь-який кеш (cache: «no-cache»), щоб гарантувати, що завжди будуть отримані свіжі дані. Заголовки включають Content-Type: «application/json», що означає, що тіло запиту у форматі JSON.

3.4.2 Запити до API

Функція UserAuth – це асинхронна JavaScript-функція, призначена для обробки автентифікації користувача за допомогою POST-запиту до вказаної кінцевої точки API. Функція приймає в якості параметрів об’єкт, що містить дані і пароль, які використовуються для аутентифікації користувача. Для коду статусу 500 (внутрішня помилка сервера) вона отримує і видає детальне повідомлення про помилку з JSON відповіді. Для інших статусів помилки вона виводить загальне повідомлення про помилку, що вказує на невдалу автентифікацію і включає код статусу. Якщо запит пройшов успішно, функція розбирає JSON відповіді. Вона отримує властивість користувача з JSON відповіді і повертає її.

Функція UserDelete – це асинхронна JavaScript-функція, призначена для видалення користувача шляхом надсилання запиту DELETE до вказаної кінцевої точки API. Функція отримує об’єкт, що містить ім’я користувача як параметр, і намагається видалити відповідн го користувача. Для коду статусу 500 (внутрішня помилка сервера) вона повертає false, що вказує на проблему на стороні сервера. Для інших статусів помилки вона видає помилку з повідомленням про те, що користувача не вдалося видалити, і включає код статусу. Якщо запит виконано успішно, функція аналізує і повертає відповідь у форматі JSON.

Функція UserCreate – це асинхронна функція JavaScript, призначена для створення нового користувача за допомогою POST-запиту до вказаної кінцевої точки API. Функція приймає об’єкт, що містить дані користувача, такі як ім’я користувача, пароль, перше_ім’я, останнє_ім’я, email і, за бажанням, pic (зображення), і відправляє ці дані на сервер для реєстрації нового користувача. Для коду статусу 500 (внутрішня помилка сервера) вона витягує і кидає детальне повідомлення про помилку з JSON відповіді. Для інших статусів помилки вона виводить загальне повідомлення про помилку, яке вказує на те, що не вдалося створити нового користувача, і включає код статусу. Якщо запит виконано успішно, функція аналізує і повертає

відповідь JSON.

Функція GetNews – це асинхронна JavaScript-функція, призначена для отримання новинних статей за допомогою POST-запиту до вказаної кінцевої точки API. Функція приймає об’єкт, що містить параметри page, limit та search, які керують пагінацією, кількістю статей для отримання та критеріями пошуку відповідно. Для коду статусу 500 (внутрішня помилка сервера) вона повертає false, що вказує на проблему на стороні сервера. Для інших статусів помилки вона видає помилку з повідомленням про те, що не вдалося отримати статті новин, і включає код статусу. Якщо запит виконано успішно, функція розбирає відповідь JSON. Вона ітераційно переглядає масив новин у відповіді, форматуючи дату створення кожної новини за допомогою бібліотеки моментів до певного формату ("ДД-ММ-РРРР ГГ:хх:сс").

Функція CreateNews – це асинхронна JavaScript-функція, призначена для створення нової новини за допомогою POST-запиту до вказаної кінцевої точки API. Функція приймає об’єкт, що містить title, body, ім’я користувача та image_data, які представляють деталі новинної статті, що створюється. Для коду статусу 500 (внутрішня помилка сервера) вона витягує і кидає детальне повідомлення про помилку з JSON відповіді. Для інших статусів помилки вона виводить загальне повідомлення про помилку, яке вказує на те, що не вдалося створити новину, і включає код статусу. Якщо запит виконано успішно, функція аналізує JSON відповіді. Вона повертає властивість з аналізованої відповіді, яка містить деталі новоствореної новини.

Функція LikedNewsGet – це асинхронна JavaScript-функція, призначена для отримання новин, які сподобалися певному користувачеві. Вона робить це за допомогою POST-запиту до вказаної кінцевої точки API. Функція приймає об’єкт, що містить ім’я користувача, сторінку, ліміт і пошук, які використовуються для фільтрації та пагінації результатів. Функція перевіряє, чи відповідь не є нормальною (!res.ok). Для коду статусу 500 (внутрішня помилка сервера) вона витягує і кидає детальне повідомлення про помилку з JSON відповіді. Для інших статусів помилки вона виводить загальне повідомлення про помилку, яке вказує на те, що не вдалося завантажити

вподобані статті новин, і включає код статусу. Якщо запит виконано успішно, функція розбирає JSON відповіді. Вона ітераційно переглядає масив новин у відповіді, форматуючи дату створення кожної новини за допомогою бібліотеки моментів до певного формату ("ДД-ММ-РРРР ГГ:хх:сс ").

Функція DislikedNewsGet – це асинхронна JavaScript-функція, яка отримує новини, що не подобаються певному користувачу. Це досягається за допомогою POST-запиту до визначеної кінцевої точки API. Функція приймає об’єкт, що містить такі параметри, як ім’я користувача, сторінка, ліміт і пошук для фільтрації та пагінації результатів. Для коду статусу 500 (внутрішня помилка сервера) вона витягує і кидає детальне повідомлення про помилку з JSON відповіді. Для інших статусів помилки вона виводить загальне повідомлення про помилку, яке вказує на те, що не вдалося завантажити новини, які вам не подобаються, і включає код статусу. Якщо запит виконано успішно, функція розбирає JSON відповіді. Вона ітераційно переглядає масив новин у відповіді, форматуючи дату створення кожної новини за допомогою бібліотеки моментів до певного формату ("ДД-ММ-РРРР ГГ:хх:сс ").

Функція GetOneNews – це асинхронна JavaScript-функція, призначена для отримання конкретної новини за її унікальним ідентифікатором. Ця функція надсилає POST-запит на вказану кінцеву точку API і очікує у відповідь одну новину. Функція отримує об’єкт з властивістю id, яка використовується для ідентифікації конкретної новони. Для коду статусу 500 (внутрішня помилка сервера) функція повертає false, що вказує на те, що новину не вдалося отримати. Для інших статусів помилки вона видає помилку з детальним повідомленням, витягнутим з JSON відповіді, що допомагає в налагодженні і забезпечує чіткий зворотній зв’язок. Якщо запит виконано успішно, функція розбирає JSON відповіді. Вона ітераційно переглядає масив новин у відповіді, форматуючи дату створення кожної новини за допомогою бібліотеки моментів до певного формату ("ДД-ММ-РРРР ГГ:хх:сс ").

Функція NewsEdit – це асинхронна JavaScript-функція, призначена для

оновлення існуючої статті новини. Вона надсилає POST-запит до вказаної кінцевої точки API, надаючи необхідні дані для оновлення статті. Функція приймає об’єкт з такими властивостями, як id, title, body і image_data, щоб полегшити процес редагування. Функція перевіряє, чи відповідь не є нормальною (!res.ok). Для коду статусу 500 (внутрішня помилка сервера) функція повертає false, що означає, що редагування новини не вдалося. Для інших статусів помилки вона видає помилку з повідомленням, яке містить код статусу, що вказує на неможливість редагування новини. Якщо запит виконано успішно, функція розбирає відповідь JSON, щоб витягти оновлені дані новини.

Функція DeleteOneNews – це асинхронна JavaScript-функція, призначена для видалення конкретної новини. Ця функція надсилає запит DELETE на вказану кінцеву точку API, використовуючи унікальний ідентифікатор статті для виконання видалення. Функція приймає об’єкт, що містить ідентифікатор новини, яку потрібно видалити. Для коду статусу 500 (внутрішня помилка сервера) вона витягує і кидає детальне повідомлення про помилку з JSON відповіді. Для інших статусів помилки вона виводить загальне повідомлення про помилку, яке вказує на те, що не вдалося видалити новину, і включає код статусу. Якщо запит виконано успішно, функція аналізує відповідь JSON, щоб отримати підтвердження видалення.

Функція NewsLike – це асинхронна JavaScript-функція, призначена для встановлення «like» для конкретної новини від конкретного користувача. Ця функція надсилає POST-запит на вказану кінцеву точку API з необхідними даними для запису «like». Вона приймає об’єкт, що містить ідентифікатор новини та ім’я користувача для ідентифікації новини та користувача, який її вподобав. Для коду статусу 500 (внутрішня помилка сервера) функція повертає false, що вказує на неможливість реєстрації. Для інших статусів помилки вона видає помилку з повідомленням, яке містить код статусу, що вказує на те, що не вдалося зареєструвати подібний запит. Якщо запит виконано успішно, функція розбирає і повертає відповідь у форматі JSON.

Функція NewsDislike – це асинхронна JavaScript-функція, призначена

для встановлення «dislike» конкретної новини від конкретного користувача. Ця функція надсилає POST-запит до вказаної кінцевої точки API, надаючи необхідні дані для фіксації «dislike». Функція приймає об’єкт, що містить ідентифікатор новини та ім’я користувача для ідентифікації новини та користувача, який її не вподобав. Для коду статусу 500 (Внутрішня помилка сервера) функція повертає false, що означає, що не вдалося зареєструвати дизлайк. Для інших статусів помилки вона видає помилку з повідомленням, яке містить код статусу, що вказує на те, що не вдалося зареєструвати вподобання. Якщо запит виконано успішно, функція розбирає і повертає відповідь у форматі JSON.

Функція GetGrades – це асинхронна JavaScript-функція, призначена для отримання кількості лайків і дизлайків для певної новини. Ця функція надсилає POST-запит до вказаної кінцевої точки API, надаючи необхідні дані для отримання оцінок. Функція приймає об’єкт, що містить ідентифікатор новинної статті, щоб визначити, оцінки якої статті потрібно отримати. Для коду статусу 500 (внутрішня помилка сервера) функція повертає false, що означає, що отримати оцінки не вдалося. Для інших статусів помилки вона видає помилку з детальним повідомленням, витягнутим з JSON відповіді. Якщо запит виконано успішно, функція аналізує JSON-відповідь, щоб витягти кількість лайків і дизлайків.

Функція CommentsLike – це асинхронна JavaScript-функція, призначена для реєстрації «like» для конкретного коментаря конкретного користувача. Ця функція надсилає POST-запит на вказану кінцеву точку API, надаючи необхідні дані для запису «like». Функція приймає об’єкт, що містить ідентифікатор коментаря та ім’я користувача для ідентифікації коментаря та користувача, який його вподобав. Для коду статусу 500 (Внутрішня помилка сервера) функція повертає false, що вказує на неможливість реєстрації. Для інших статусів помилки вона видає помилку з повідомленням, яке містить код статусу, що вказує на те, що не вдалося зареєструвати подібний запит. Якщо запит виконано успішно, функція розбирає і повертає відповідь у форматі JSON.

Функція CommentsDislike – це асинхронна JavaScript-функція, призначена для реєстрації «dislike» для конкретного коментаря конкретного користувача. Ця функція надсилає POST-запит на вказану кінцеву точку API, надаючи необхідні дані для запису «dislike». Функція приймає об’єкт, що містить ідентифікатор коментаря та ім’я користувача для ідентифікації коментаря та користувача, який його не вподобав. Для коду статусу 500 (Внутрішня помилка сервера) функція повертає false, що вказує на неможливість реєстрації. Для інших статусів помилки вона видає помилку з повідомленням, яке містить код статусу, що вказує на те, що не вдалося зареєструвати подібний запит. Якщо запит виконано успішно, функція розбирає і повертає відповідь у форматі JSON.

Функція GetComGrades – це асинхронна JavaScript-функція, призначена для отримання інформації про оцінки (лайки і дизлайки) конкретного коментаря. Ця функція надсилає POST-запит до вказаної кінцевої точки API. Вона приймає об’єкт, що містить ідентифікатор коментаря, оцінки якого потрібно отримати. Якщо статус відповіді 500 (внутрішня помилка сервера), функція повертає об’єкт з likes: 0 і dislikes: 0. Для інших статусів помилки функція генерує помилку з повідомленням про те, що не вдалося отримати оцінки коментарів, включаючи повідомлення про помилку з JSON відповіді. Якщо запит виконано успішно, функція аналізує JSON-відповідь для вилучення вподобань і невподобань. Потім функція повертає об’єкт, що містить лайки і дизлайки.

Функція NotiGet – це асинхронна JavaScript-функція, призначена для отримання сповіщень для конкретного користувача. Вона виконує POST-запит до вказаної кінцевої точки API. Функція приймає об’єкт, що містить ім’я користувача, ліміт, сторінку та пошук, які використовуються для фільтрації та пагінації сповіщень. Якщо статус відповіді 500 (внутрішня помилка сервера), фукція витягує з відповіді детальне повідомлення про помилку та повертає його. Для інших статусів помилки функція генерує помилку з повідомленням про те, що не вдалося отримати сповіщення, включно з кодом статусу. Якщо запит виконано успішно, функція розбирає

JSON відповіді, щоб витягти noti, count і сторінки.Потім функція форматує дату створення кожного повідомлення за допомогою бібліотеки моментів у певний формат ("ДД-ММ-РРРР ГГ:хх:сс ").

Функція NotiRead – це асинхронна JavaScript-функція, призначена для позначення певного повідомлення як прочитаного для даного користувача. Це досягається за допомогою POST-запиту до вказаної кінцевої точки API. Функція приймає об’єкт, що містить ідентифікатор повідомлення та ім’я користувача. Якщо статус відповіді 500 (внутрішня помилка сервера), функція не виконує ніяких конкретних дій, але може бути розширена для обробки цього випадку, якщо це необхідно. Для інших статусів помилки функція видає помилку з повідомленням про те, що не вдалося позначити повідомлення як прочитане, включно з кодом статусу. Якщо запит виконано успішно, функція розбирає відповідь JSON і повертає її.

Функція NotiDelete – це асинхронна JavaScript-функція, призначена для видалення конкретного повідомлення для даного користувача. Це досягається шляхом надсилання запиту DELETE до вказаної кінцевої точки API. Функція приймає об’єкт, що містить ідентифікатор сповіщення та ім’я користувача. Якщо статус відповіді 500 (внутрішня помилка сервера), функція повертає false.Для інших статусів помилки функція видає помилку з повідомленням про те, що не вдалося видалити сповіщення, включаючи код статусу. Якщо запит виконано успішно, функція розбирає відповідь JSON і повертає її.

3.4.4 Збереження станів

Для збереження деякої інформації використовувався менеджер станів zustand.

Менеджери станів є важливими інструментами в сучасній розробці програмного забезпечення, особливо для додатків зі складними

користувацькими інтерфейсами та взаємодіями. Вони надають систематизований спосіб обробки та управління станом програми, забезпечуючи узгодженість, передбачуваність і простоту обслуговування. Розуміння ролі та переваг менеджерів стану має вирішальне значення для розробників, які прагнуть створювати надійні та масштабовані додатки.

Менеджери станів, також відомі як бібліотеки або інструменти керування станами, призначені для керування станом програми. Під «станом» маються на увазі дані, які потрібні програмі для коректної роботи, зокрема дані, введені користувачем, відповіді сервера, кешовані дані та стан інтерфейсу користувача. Менеджери станів забезпечують структурований спосіб управління цими даними, гарантуючи, що стан є послідовним, а зміни передбачуваними.

Однією з основних функцій менеджерів стану є централізація стану в одному місці або сховищі. Така централізація спрощує відстеження та оновлення стану, оскільки всі зміни в ньому проходять єдиним, передбачуваним шляхом. Це полегшує налагодження та міркування про стан, зменшуючи ймовірність помилок, спричинених неузгодженістю стану.

Менеджери стану часто забезпечують односпрямований потік даних, коли зміни стану проходять через певні, контрольовані процеси. Такий підхід допомагає підтримувати цілісність стану, гарантуючи, що всі зміни є навмисними і відстежуваними. Популярні бібліотеки управління станом, такі як Redux, реалізують цей патерн, щоб сприяти ясності та передбачуваності.

Використовуючи менеджер станів, розробники можуть відокремити логіку, яка керує станом, від логіки, яка обробляє інтерфейс користувача та бізнес-правила. Таке розділення покращує організацію коду та модульність, що полегшує розробку, тестування та підтримку додатку.

Менеджери станів надають інструменти та методи, які роблять зміни станів передбачуваними та легшими для налагодження. Наприклад, дії, які змінюють стан, часто реєструються, і зміни стану можна відстежувати в часі.

У міру того, як додатки ускладнюються, керувати станом без

структурованого підходу стає все складніше. Менеджери станів добре масштабуються разом з додатком, надаючи механізми для ефективного управління станами у великих, складних додатках. Вони підтримують розширені функції, такі як налагодження в часі, гаряче перезавантаження та інтеграція з проміжним програмним забезпеченням, які мають вирішальне значення для підтримки продуктивності та продуктивності розробників у великих проєктах.

У додатках з багатьма взаємодіючими компонентами забезпечення узгодженості та актуальності стану кожного з них може бути складним завданням. Менеджери станів допомагають підтримувати цю узгодженість, надаючи єдине джерело істини для стану, до якого всі компоненти можуть мати доступ і оновлювати його контрольованим чином. Це зменшує ризик того, що компоненти не будуть синхронізовані один з одним.

Zustand – це бібліотека керування станами для JavaScript та TypeScript додатків, орієнтована, насамперед, на простоту та продуктивність. Назва «zustand» походить від німецького слова «стан», що відображає призначення бібліотеки – ефективно керувати станом додатку. Вона особливо добре підходить для додатків, створених за допомогою React, але є достатньо гнучкою для використання з іншими фреймворками або звичайним JavaScript.

Zustand розроблений для забезпечення мінімалістичного та простого підходу до управління станом. Він пропонує легке, але потужне рішення, яке дозволяє уникнути складності, що часто асоціюється з більш комплексними бібліотеками для управління станом, такими як Redux. Zustand робить акцент на простоті використання, мінімальному шаблонному коді та продуктивності.

Хоча Zustand можна використовувати з будь-яким JavaScript фреймворком, він особливо добре інтегрований з React. Zustand використовує API хуків React, щоб забезпечити плавний та ідіоматичний спосіб керування станом у React-додатках. Ця інтеграція гарантує, що Zustand працює природно в екосистемі React, що робить його популярним вибором для React-

розробників

Zustand є універсальним, оскільки може керувати як глобальним, так і локальним станом. Глобальне керування станом дозволяє розподіляти стан між декількома компонентами, в той час як локальне керування станом обмежує його до певних компонентів. Ця гнучкість дозволяє розробникам використовувати Zustand для широкого спектру потреб в управлінні станами, від невеликих локальних станів до складних станів в масштабі всього додатку.

Zustand оптимізовано для продуктивності, що гарантує ефективне оновлення станів та мінімізацію непотрібних повторних рендерингів. Бібліотека використовує неглибоке порівняння для виявлення змін, перемальовуючи компоненти лише за необхідності. Така оптимізація має вирішальне значення для підтримки продуктивності у великих і складних додатках.

Zustand розроблено з урахуванням гнучкості та розширюваності. Він надає ряд опцій для визначення та маніпулювання станом, включаючи дії, селектори та проміжне програмне забезпечення. Розробники можуть розширювати функціональність Zustand для задоволення конкретних потреб, інтегруючи його з іншими бібліотеками та інструментами за потреби.

3.5 Реалізація інтерфейсу

Реалізація зовнішнього вигляду відбувалася за рахунок використання React компонентів які використовувалися на статичних сторінках серверного рендерінгу.

Структура сторінка така що на сторінці можуть розміщатися функції для отримання або передачі даних, оболонка для відображення данника, а також компонент який у собі може містити вкладені компоненти та функції. Для прикладу розглянемо структури сторінка News.

На сторінці є функціональний компонент News який є основним експортом файлу, який відображає оболонку, що містить вкладенний компонент NewsList.

Компонент NewsList керує відображенням і функціональністю списку новин. Компонент містить функції а також оболонки для відображення.

Автентифікація: використовується служба автентифікації для отримання інформації автентифікованого користувача, зокрема імені користувача.

Управління станом: використувається модулі керування станом зміних, зокрема станом новини, відстежується загальна кількість новин, визначаються кількість новин, що відображаються на сторінці, відстежується номер поточної сторінки

Процес отримання даних відбувається, коли компонент монтується або коли змінюються залежності.

Є функції которі реалізовують отримання новин на основі поточного пошукового запиту, очищення пошукового запиту, оновлює кількість елементів на сторінці та повторно завантажує новини, оновлює поточну сторінку та повторно завантажує новини.

Візуальна частина сторінки реалізована за принципом блоків.

Заголовок: відображає заголовок і кнопку «Створити», якщо користувач автентифікований.

Компонент пошуку: керує функціями пошуку, дозволяючи користувачам здійснювати пошук і очищати пошуковий запит.

Пагінація: два екземпляри компонента Pagination використовуються, один у верхній і один у нижній частині списку новин, для навігації сторінками новин.

Список новин: відображає список новин за допомогою компонента NewsMap для кожної новини.

Стан завантаження: відображається повідомлення про завантаження, коли завантажуються новини.

3.6 Тестування

У проєкту було застосовано автоматичне застосування вебзастосунку за допомогою Mocha.js

Mocha.js – це широко використовуваний фреймворк для тестування JavaScript, що працює на Node.js і в браузері, розроблений для того, щоб зробити асинхронне тестування простим і ефективним [31]. Він забезпечує надійну та гнучку основу для написання, організації та запуску тестів, що робить його кращим вибором для багатьох розробників, які працюють з JavaScript додатками.

Поведінково-орієнтована розробка (Behavior-driven Development, BDD) та тестово-орієнтована розробка (Test-driven Development, TDD): Mocha підтримує як BDD, так і TDD інтерфейси, що дозволяє розробникам писати тести у стилі, який відповідає їхнім уподобанням [32]. У BDD тести пишуться в більш описовій манері з використанням таких ключових слів, як describe і it, в той час як TDD використовує suite і test.

Однією з найсильніших особливостей Mocha є її здатність витончено працювати з асинхронним тестуванням. Він надає прості у використанні механізми для роботи з асинхронними операціями, такими як зворотні виклики, обіцянки та синтаксис async/await. Це важливо для сучасних JavaScript-додатків, які часто покладаються на асинхронне отримання та обробку даних.

Mocha пропонує потужні хуки для встановлення та зняття умов до і після тестів. Основні хуки включають before, after, beforeEach та afterEach. Ці хуки безцінні для ініціалізації станів, очищення ресурсів та забезпечення запуску тестів в ізольованих середовищах.

Mocha є дуже розширюваним завдяки підтримці плагінів та користувацьких звітів. Розробники можуть інтегрувати його з іншими бібліотеками та інструментами, такими як Chai для тверджень, Sinon для шпигунів, імітацій та заглушок, а також різними генераторами звітів для створення тестових звітів у різних форматах (наприклад, JSON, HTML).

Вихідні дані Mocha за замовчуванням є чистими і читабельними, але він також дозволяє використовувати користувацькі репортери. Фреймворк підтримує різні звіти для задоволення різних потреб, від простого консольного виводу до складних конвеєрів CI/CD та звітів про охоплення.

Mocha надає гнучкі способи вказівки тестових файлів і каталогів, що полегшує організацію тестів у великих проєктах. Він може рекурсивно шукати тестові файли у вказаних каталогах, підтримуючи різні формати файлів та угоди про імена.

Mocha сумісний з широким спектром середовищ. Він працює на Node.js і в браузері, що робить його придатним для тестування бекенд-сервісів, фронтенд-додатків або навіть спільних кодових баз, які працюють в обох середовищах.

Mocha має сильну та активну спільноту, що сприяє його постійному вдосконаленню та інтеграції з іншими інструментами. Розгалужена екосистема навколо Mocha включає плагіни, навчальні посібники та підтримку спільноти, гарантуючи, що розробники можуть легко знайти допомогу та ресурси.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи бакалавра було створено вебсервіс новин з елементами соціальної мережі, що пропонує користувачам отримувати та створювати контент на основі їх вподобано. Також було реалізовано зручний інтерфейс користувача.

Для реалізації вебсервісу було проведено аналіз вимог до програмного забезпечення та сформовано мету, завдання, а також були сформовані вимоги до програмного забезпечення дипломного проєкту бакалавра.

Побудована модель клієнтської частини вебсервісу, описані специфікації варіантів використання та створені UML діаграми для полегшення проєктування і в подальшому реалізації. Було виконано проєктування архітектури бази даних та серверної частини застосунку, було обрано модель «клієнт-сервер».

Був проведений аналіз можливих засобів для реалізації вебсервісу. Для реалізації бази даних було обрано SQL, а у якості система керування базою даних PostgreSQL.

Для клієнтської частини застосунку використовувався Next.js та React. Для написання стилів було обрано css фреймворк tailwind.

Для серверної частини додатку було обрано Node.js.

Було проведено тестування створеного вебсервісу з використання автоматичних технологій тестування, а такаж було виконанно тестування серної частини за допомогою.

У майбутньому ми планується розвивати вебсервіс, розширювати його функціонал, покращувати дизайн та підвищувати зручність користування на основі відгуків реальних користувачів.

Користувачам була надана можливість отримати класичний досвіт використання засобів масової інформаціх у новому вигляді. Для більш сучасних користувачів функціонал надає можливість додаткової взаємодії.

ПЕРЕЛІК ПОСИЛАНЬ

1. What’s the Difference Between Frontend and Backend in Application Development? URL: http://surl.li/udvcr (дата звернення: 12.04.2024).
2. Next js Introduction. URL: http://surl.li/udvcv (дата звернення: 12.04.2024).
3. What is Node.js (Node)? URL: http://surl.li/udvcz (дата звернення: 12.04.2024).
4. About Node.js. URL: http://surl.li/udvdf (дата звернення: 12.04.2024).
5. Legacy Reactjs. URL: http://surl.li/udvdl (дата звернення: 13.04.2024).
6. Overview of React.js. URL: http://surl.li/udvdq (дата звернення: 13.04.2024).
7. What is TypeScript?. URL: http://surl.li/udvdw (дата звернення: 13.04.2024).
8. TypeScript for JavaScript Programmers. URL: http://surl.li/udveb (дата звернення: 13.04.2024).
9. TypeScript Introduction. URL: http://surl.li/udvej (дата звернення: 13.04.2024).
10. TypeScript – Overview. URL: http://surl.li/udvep (дата звернення: 13.04.2024).
11. 7 Top React State Management Libraries. URL: http://surl.li/udvew (дата звернення: 14.04.2024).
12. Zustand Introduction. URL: http://surl.li/udvfb (дата звернення: 14.04.2024).
13. Rapidly build modern websites without ever leaving your HTML. URL: http://surl.li/udvgz (дата звернення: 14.04.2024).
14. Introduction to Tailwind CSS. URL: http://surl.li/udvhf (дата звернення: 14.04.2024).
15. What Is a Database? URL: http://surl.li/udvib (дата звернення: 14.04.2024).
16. Taylor S. Database. URL: http://surl.li/ufmhr (дата звернення: 14.04.2024).
17. Database. URL: http://surl.li/ufmic (дата звернення: 14.04.2024).
18. What is PostgreSQL? URL: http://surl.li/udvlo (дата звернення: 14.04.2024).
19. About PostgreSQL. URL: http://surl.li/udvmm (дата звернення: 14.04.2024).

20. Web Application Architecture in 2024: Moving in the Right Direction. URL: http://surl.li/udvnm (дата звернення: 28.04.2024).
21. Web Application Architecture: Definition, Models, Types, and More. URL: http://surl.li/udvom (дата звернення: 28.04.2024).
22. Guide to Web Application Architecture. URL: http://surl.li/udvpc (дата звернення: 28.04.2024).
23. What is Architecture Diagramming? URL: http://surl.li/udvqc (дата звернення: 28.04.2024).
24. What is Unified Modeling Language. URL: http://surl.li/ufmlp (дата звернення: 28.04.2024)
25. What is Client-Server Architecture? Everything You Should Know. URL: http://surl.li/udvqt (дата звернення: 30.04.2024).
26. What is Client-Server Architecture? Explained in Detail. URL: http://surl.li/udvqx (дата звернення: 30.04.2024).
27. HTTP response status codes. URL: http://surl.li/udvrc (дата звернення: 30.04.2024).
28. Fetch API. URL: http://surl.li/udvrl (дата звернення: 30.04.2024).
29. Using the Fetch API. URL: http://surl.li/udvrz (дата звернення: 30.04.2024).
30. Fetch info. URL: http://surl.li/udvsj (дата звернення: 30.04.2024).
31. Mocha info. URL: http://surl.li/udvsu (дата звернення: 12.05.2024).
32. Getting Started with Node.js and Mocha. URL: http://surl.li/udvth (дата звернення: 12.05.2024).
33. What Is Ubuntu? A Quick Beginner’s Guide URL: http://surl.li/ufmoa (дата зверення: 12.05.2024).

ДОДАТОК А

Дизайн застосунку

Рисунок А.1 – Головна сторінка

Рисунок А.2 – Сторінка реєстрації

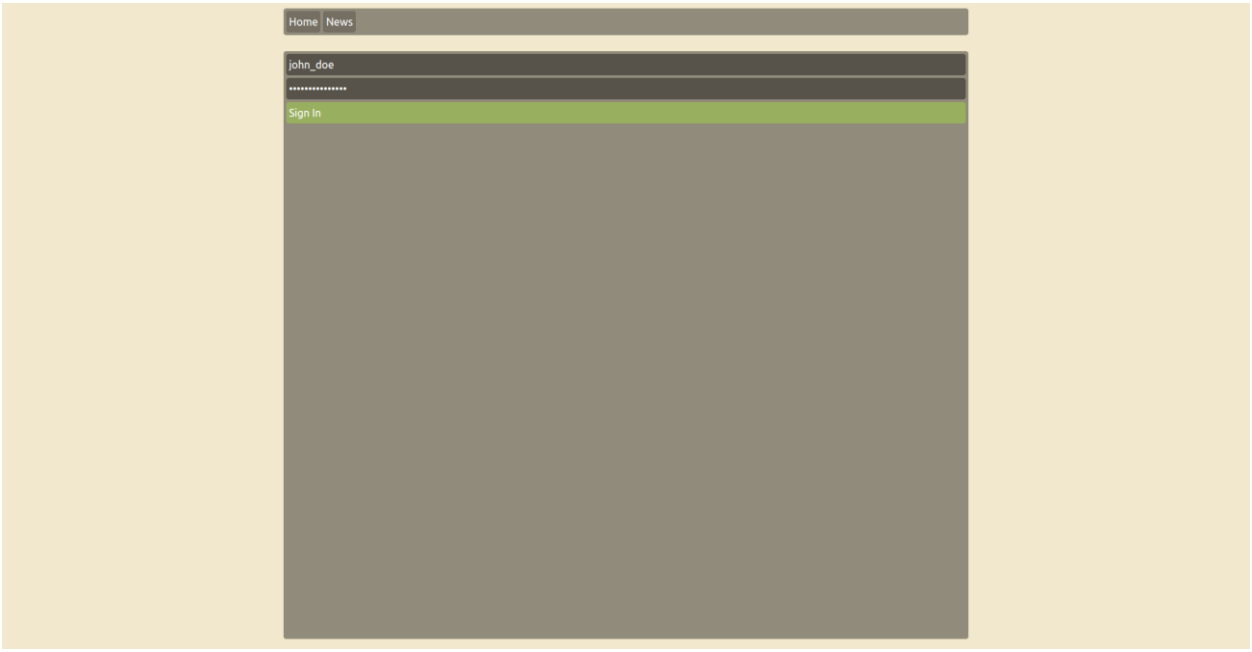

Рисунок А.3 – Сторінка авторизації

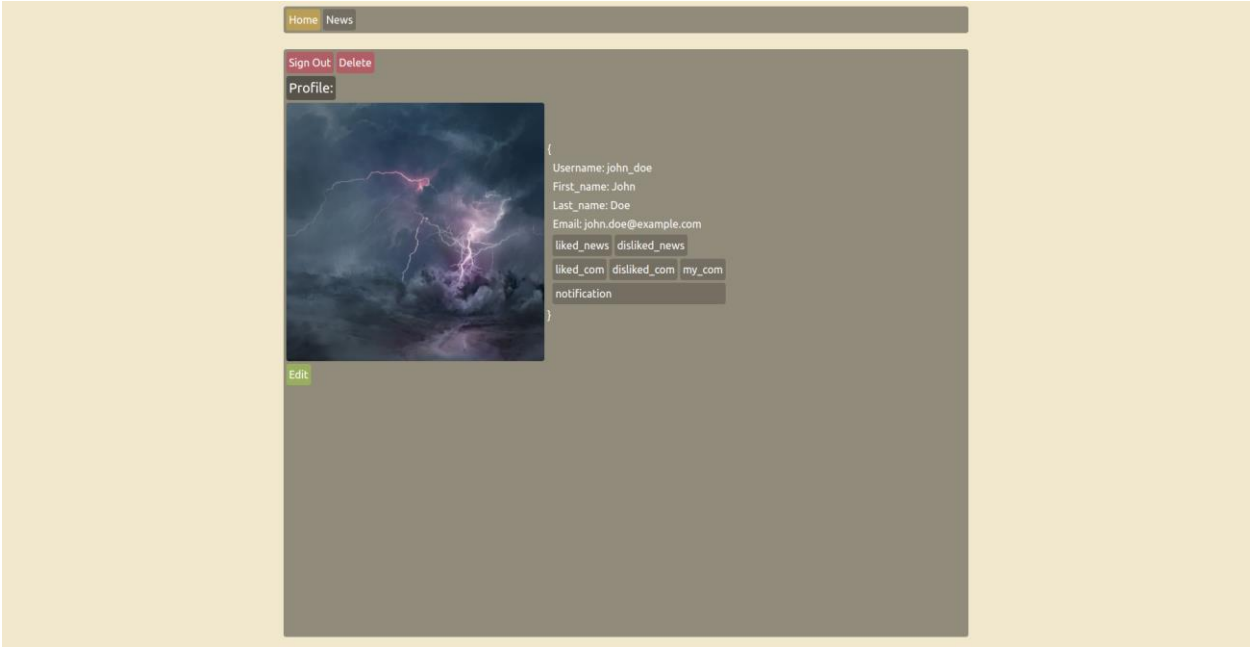

Рисунок А.4 – Головна сторінка з компонентом профіля

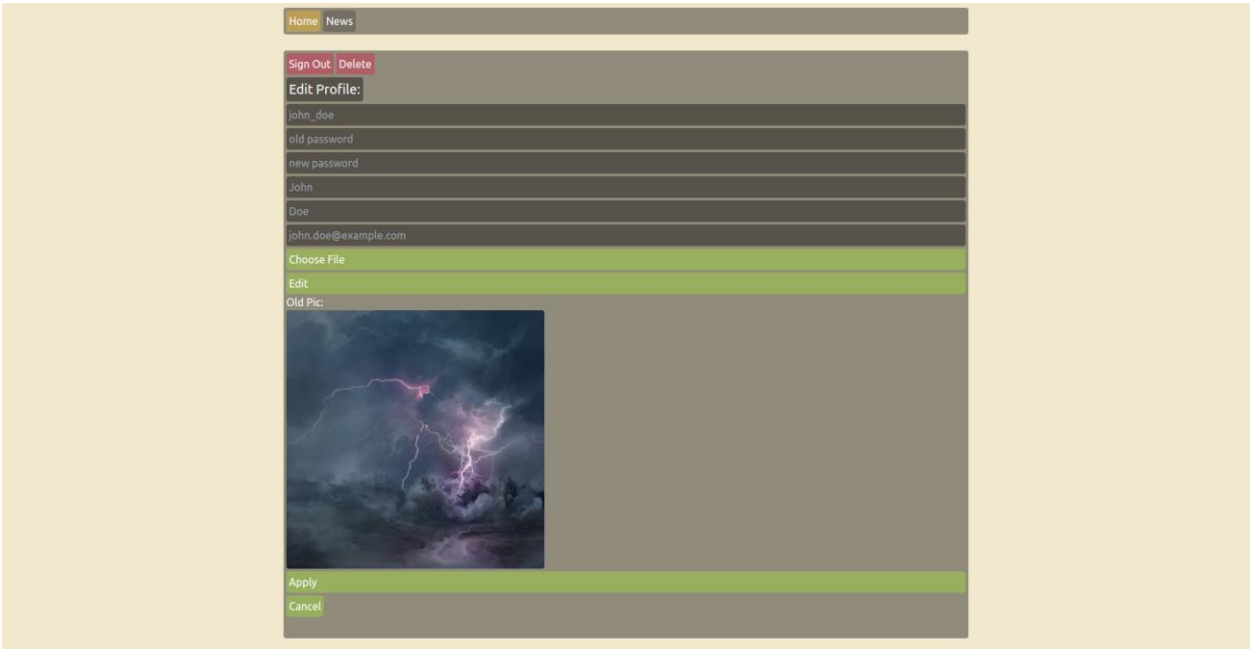

Рисунок А.5 – Головна сторінка з компонентою кастомізації користувача

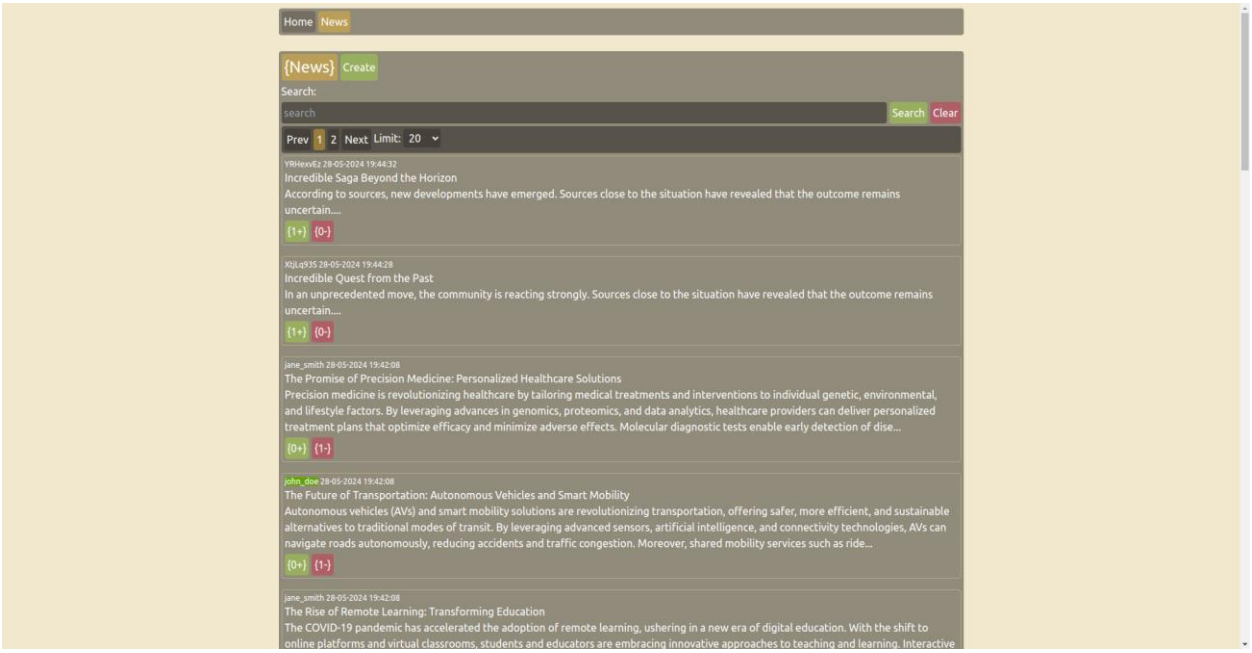

Рисунок А.6 – Сторінка новин

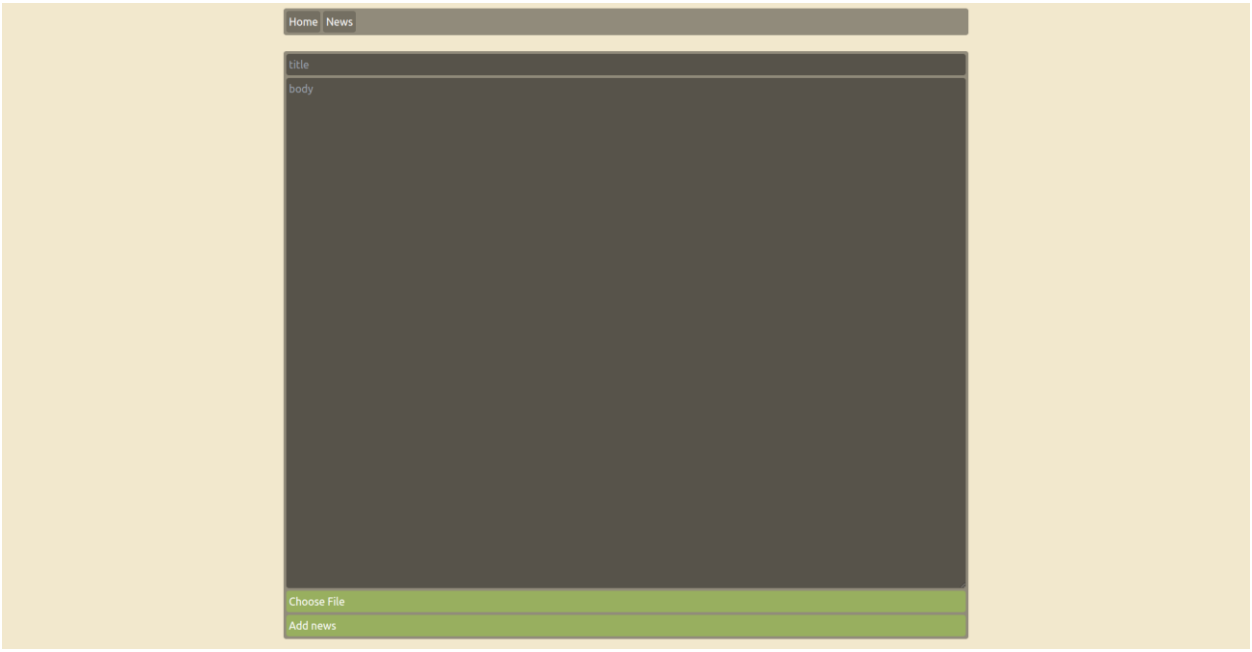

Рисунок А.7 – Сторінка створення новини

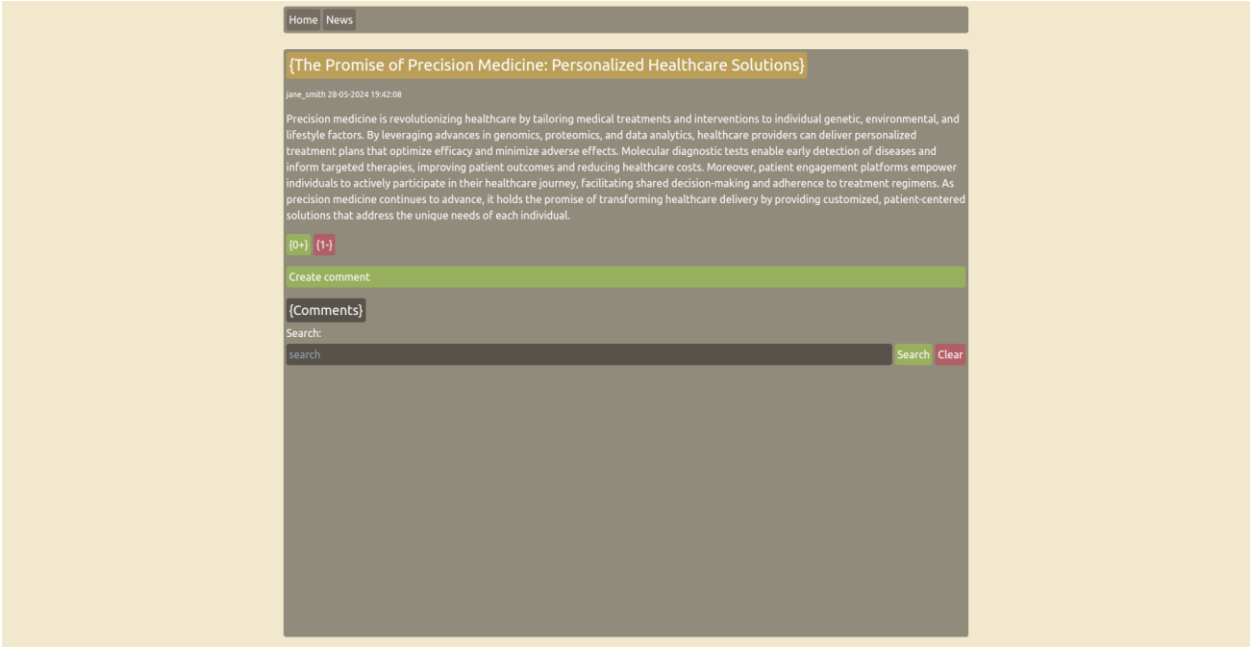

Рисунок А.8 – Сторінка перегляду новини

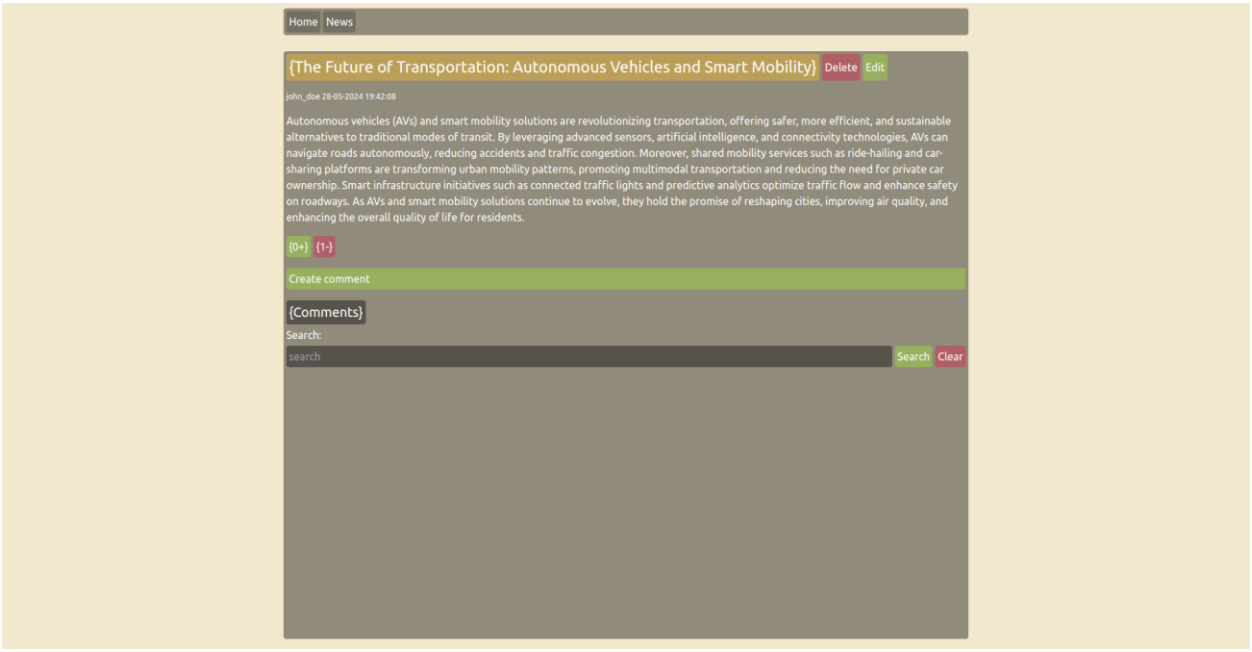

Рисунок А.9 – Сторінка перегляду власної новини

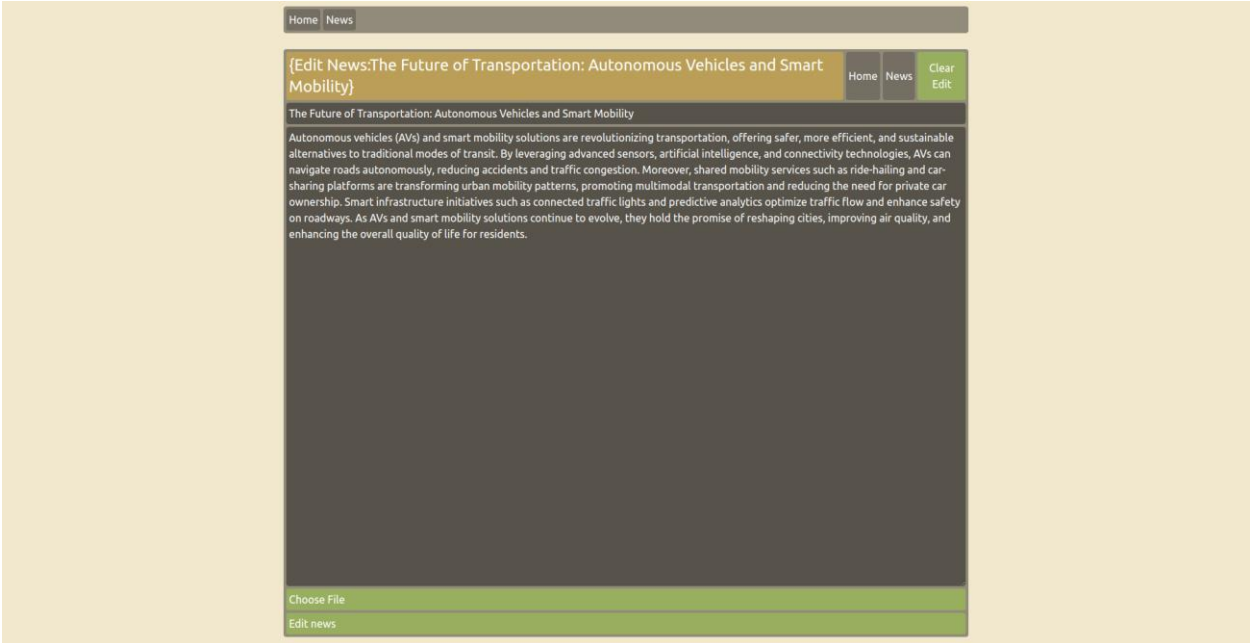

Рисунок А.10 – Сторінка змінення власної новини

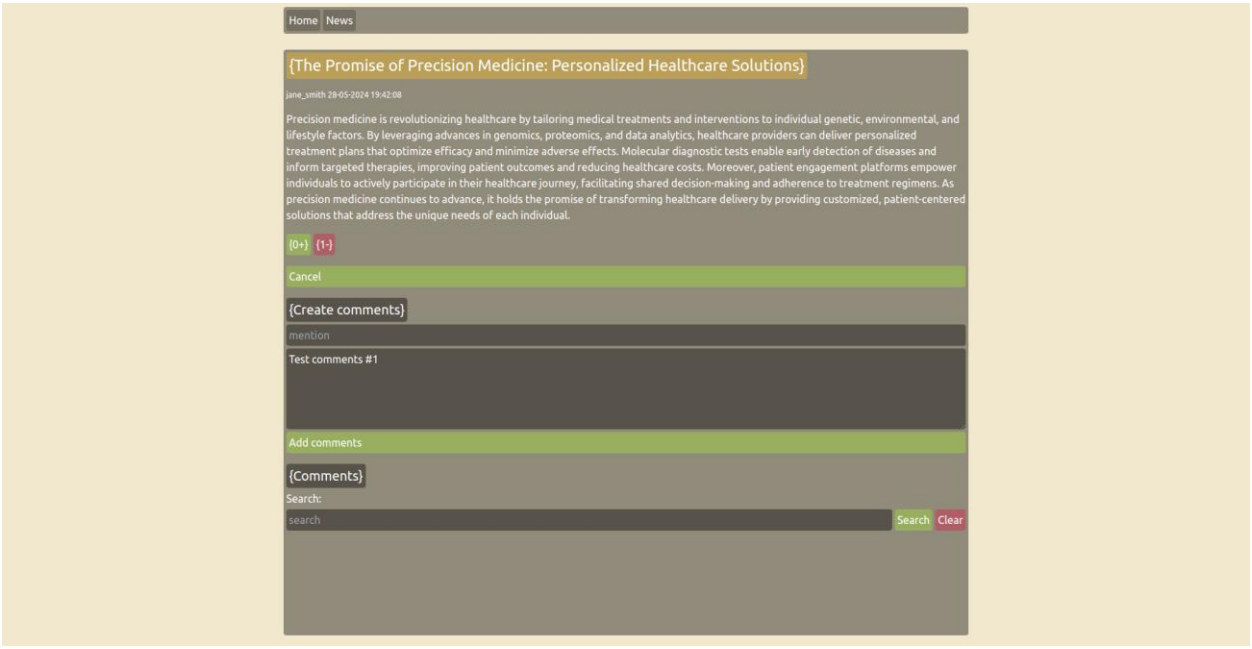

Рисунок А.11 – Сторінка перегялду новини з відкритою формою коментарів

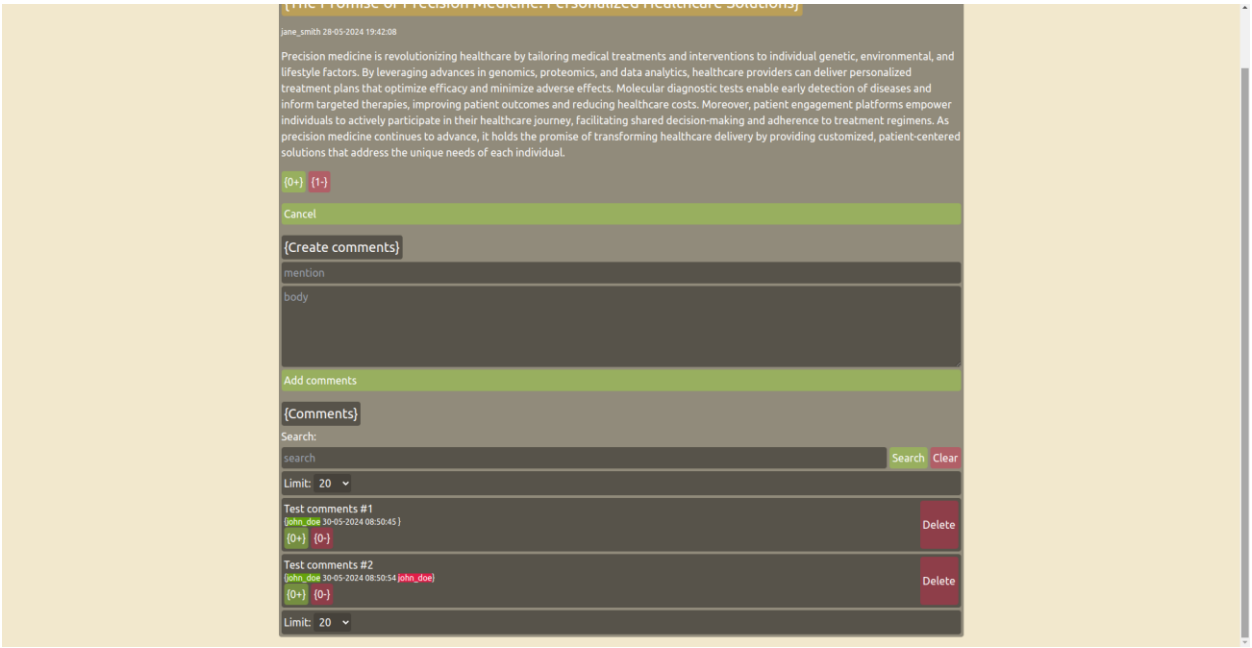

Рисунок А.12 – Сторінка перегляду новини зі створеними коментарями

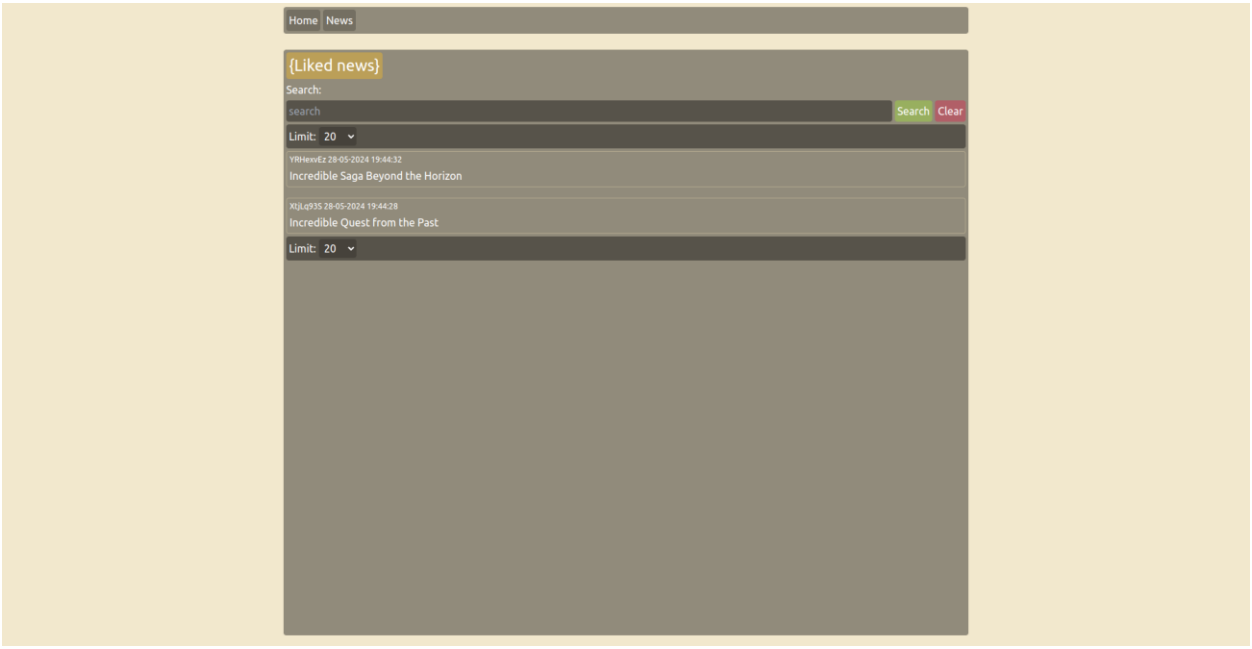

Рисунок А.13 – Сторінка з новинами які сподобались користувачу

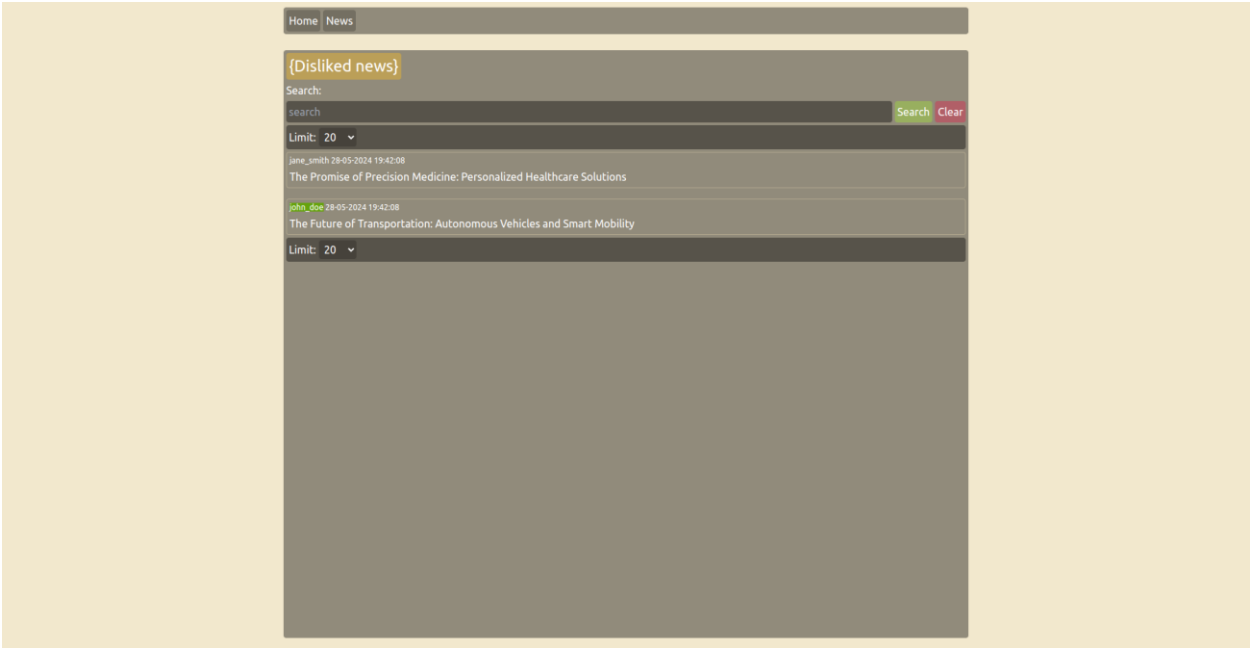

Рисунок А.14 – Сторінка з новинами які не сподобалися користувачу

Рисунок А.15 – Сторінка з коментарями які сподобалися користувачу

Рисунок А.16 – Сторінка з коментарями які не сподобалися користувачу

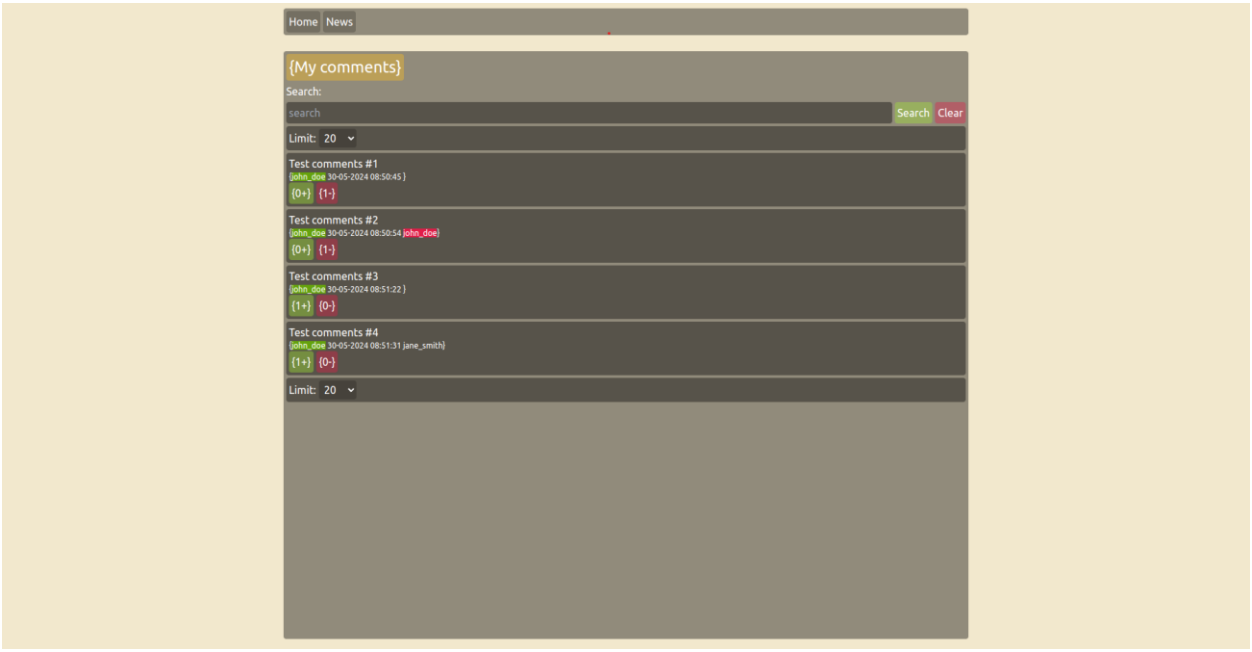

Рисунок А.17 – Сторінка з коментарями користувача

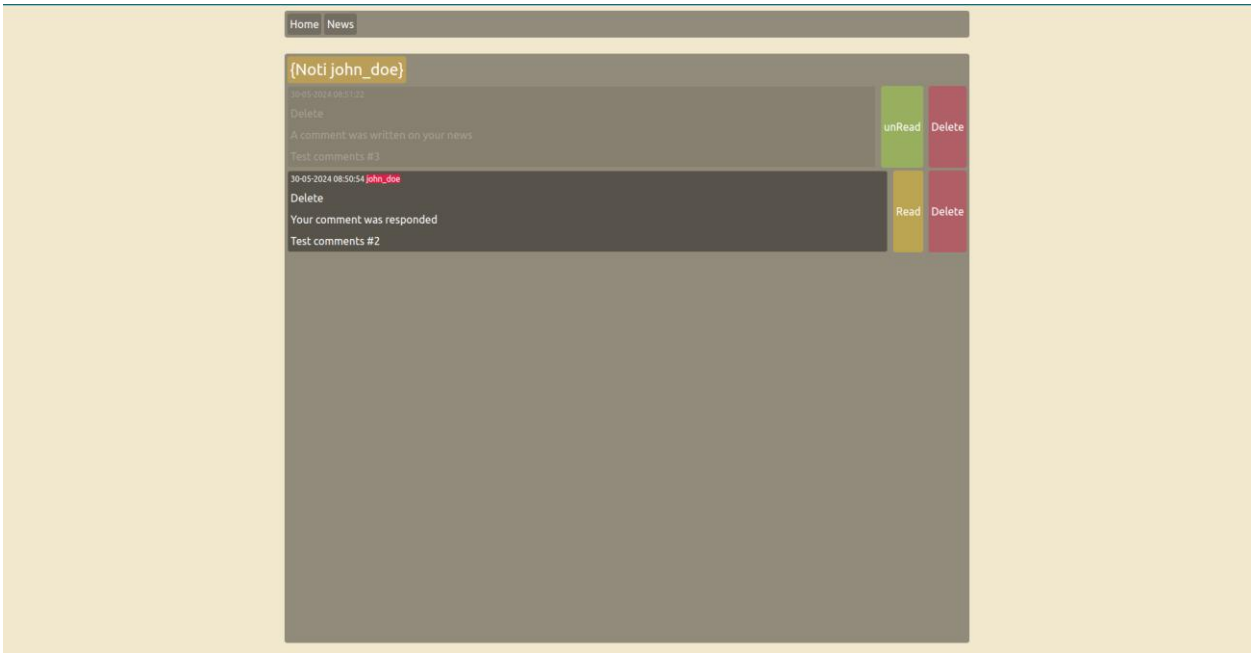

Рисунок А.18 – Сторінка з повідомленнями користувача

ДОДАТОК Б

Лістинг коду

Б.1 UserAuth

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function GET(req: NextRequest) {
    try {
        const users = await db.query("SELECT * FROM users");
        if (users.rowCount)
            return NextResponse.json({ users: users.rows }, { status: 200 });
        return NextResponse.json({ error: `Users doesn't exist` }, { status:
500 });
    } catch (error) {
        console.log({ error: `${error}` }, { status: 400 });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
export async function POST(req: NextRequest) {
    try {
        const body = await req.json();
        const checkMail = await db.query(
            `SELECT * FROM users WHERE email = '${body.data}' AND password =
'${body.password}';`
        );
        if (checkMail.rowCount)
            return NextResponse.json({ user: checkMail.rows[0] }, { status:
200 });
        const checkUsername = await db.query(
            `SELECT * FROM users WHERE username = '${body.data}' AND password
= '${body.password}';`
        );
        if (checkUsername.rowCount)
            return NextResponse.json(
                { user: checkUsername.rows[0] },
                { status: 200 }
            );
        return NextResponse.json({ error: `User doesn't exist` }, { status:
500 });
    } catch (error) {
        console.log({ error: `${error}` }, { status: 400 });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}

export async function DELETE(req: NextRequest) {
    try {
        const body = await req.json();
        const user = await db.query(
            `DELETE FROM users * WHERE username='${body.username}';`
        );
        if (user.rowCount)
            return NextResponse.json(
                { user: `${body.username} was deleted` },
```

```
                { status: 200 }
            );
        return NextResponse.json({ error: `User doesn't exist` }, { status:
500 });
    } catch (error) {
        console.log({ error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.2 UserCreate

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";
import {UserCheck} from "@/service/User/UserCheck";

export async function POST(req: NextRequest) {
    try {
        const { username, email, password, pic, first_name, last_name } =
            await req.json();
        const data = username || email;
        const check = await UserCheck({ data, password });
        if (check.status === 200 && check.exists == true) {
            return NextResponse.json(
                { error: `User already exist` },
                { status: 500 }
            );
        }
        let createUser;
        if (pic) {
            createUser = await db.query(
                `INSERT INTO users (username, password, first_name,
last_name, email, image_data)
    VALUES ($1,$2,$3,$4,$5,$6);`,
                [username, password, first_name, last_name, email, pic]
            );
        } else {
            createUser =
                await db.query(`INSERT INTO users (username, password,
first_name, last_name, email)
    VALUES ('${username}', '${password}', '${first_name}', '${last_name}',
'${email}@');`);
        }
        if (createUser.rowCount)
            return NextResponse.json({ new_user: true }, { status: 200 });
        return NextResponse.json({ new_user: false }, { status: 500 });
    } catch (error) {
        console.log({ error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.3 UserCheck

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const { data, password } = await req.json();
        const checkPass = await db.query(
            `SELECT EXISTS (SELECT 1 FROM users WHERE (username = '${data}'
OR email = '${data}') AND password = '${password}');`
        );
        if (checkPass.rows[0].exists)
            return NextResponse.json(
                { exists:true },
                { status: 200 }
            );
        return NextResponse.json({ exists: false }, { status: 500 });
    } catch (error) {
        console.log({ error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.4 UserEdit

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const {
            username,
            password,
            first_name,
            last_name,
            email,
            pic,
            old_username,
        } = await req.json();
        let edit;
        console.log({pic})
        if (pic) {
            edit = await db.query(
                `
            UPDATE users
            SET
                username = $1,
                password = $2,
                first_name = $3,
                last_name = $4,
                email = $5,
                image_data = $6
            WHERE username = $7;
            `,
                [username, password, first_name, last_name, email, pic,
```

```
old_username]
            );
        } else {
            edit = await db.query(
                `
        UPDATE users
        SET
            username = $1,
            password = $2,
            first_name = $3,
            last_name = $4,
            email = $5
        WHERE username = $6;
        `,
                [username, password, first_name, last_name, email,
old_username]
            );
        }
        if (edit.rowCount) return NextResponse.json(true);
        return false;
    } catch (error) {
        console.log({ error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.5 NewsGet

```
import db from "@/server/db";
import {NextRequest, NextResponse} from "next/server";

export async function POST(req: NextRequest) {
    try {
        const body = await req.json();
        const {page, limit, search, username} = await body;
        let news;
        let request = `SELECT * FROM news `;
        let options = ``;
        let order = ``;

        if (
            typeof username != "undefined" &&
            typeof username != undefined &&
            username.length > 0
        ) {
            options = `WHERE username ILIKE '%${username}%'`;
        } else if (typeof search != "undefined") {
            options = `WHERE title ILIKE '%${search}%'
            OR body ILIKE '%${search}%'
            OR username ILIKE '%${search}%'`;
        }

        if (page && limit) {
            order = ` ORDER BY id DESC LIMIT ${limit} OFFSET ${(page - 1) *
limit};`;
        } else {
            order = ` ORDER BY id DESC;`;
        }
```

```
        const count = Number(
            (await db.query(`SELECT COUNT(*) FROM news
${options}`)).rows[0].count
        );
        news = await db.query(request + options + order);

        const pages = Math.ceil(count / limit) || 1;

        return NextResponse.json(
            {
                news: news.rows,
                // count,
                pages,
            },
            {status: 200}
        );


    } catch (error) {
        console.log({"news.error": error});
        return NextResponse.json({error: `${error}`}, {status: 400});
    }
}
```

Б.6 NewsCreate

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const { title, body, username, image_data } = await req.json();
        // console.log({ title, body, username, image_data });

        let create_news;

        if (image_data) {
            create_news = await db.query(
                `INSERT INTO news (title, body, username, image_data) VALUES
($1,$2,$3,$4);`,
                [title, body, username, image_data]
            );
        } else {
            create_news = await db.query(
                `INSERT INTO news (title, body, username) VALUES
($1,$2,$3);`,
                [title, body, username]
            );
        }
        if (create_news.rowCount)
            return NextResponse.json({ new_news: true }, { status: 200 });
        return NextResponse.json({ new_news: false }, { status: 500 });
    } catch (error) {
        console.log({ error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.7 NewsLike

```
import { NextRequest, NextResponse } from "next/server";
import {
    LikeNews,UndislikeNews,UnlikeNews,CheckDislike,CheckLike
} from "../service/grades";

export async function POST(req: NextRequest) {
    try {
        const body = await req.json();
        const news_id = Number(body.news_id);
        const username = body.username;

        const like_check = await CheckLike(news_id, username);
        const dislike_check = await CheckDislike(news_id, username);

        if (!like_check) {
            if (dislike_check) {
                await UndislikeNews(news_id, username);
            }
            const like = await LikeNews(news_id, username);
            if (like) return NextResponse.json({ res: true }, { status: 200
});
        }
        await UnlikeNews(news_id, username);
        return NextResponse.json({ res: false }, { status: 200 });
    } catch (error) {
        console.log({ "like_news.error": error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.8 NewsDislLike

```
import {NextRequest, NextResponse} from "next/server";
import {
     UndislikeNews, DislikeNews, CheckLike, UnlikeNews, CheckDislike
} from "../service/grades";

export async function POST(req: NextRequest) {
    try {
        const body = await req.json();
        const news_id = Number(body.news_id);
        const username = body.username;

        const dislike_check = await CheckDislike(news_id, username);
        const like_check = await CheckLike(news_id, username);

        if (!dislike_check) {
            if (like_check) {
                await UnlikeNews(news_id, username);
            }
            const dislike = await DislikeNews(news_id, username);
            if (dislike) return NextResponse.json({res: true}, {status:
200});
        }
```

```
        await UndislikeNews(news_id, username);
        return NextResponse.json({res: false}, {status: 200});
    } catch (error) {
        return NextResponse.json({error: `${error}`}, {status: 400});
    }
}
```

Б.9 OneNewsGet

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const {id} = await req.json();
        const news = await db.query(`SELECT * FROM news WHERE id =
'${id}';`);
        if (news.rowCount)
            return NextResponse.json({ news:news.rows[0] }, { status: 200 });
        return NextResponse.json(
            { error: `News ${id} doesn't exist` },
            { status: 500 }
        );
    } catch (error) {
        console.log({ "news[id].error": error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.10 OneNewsEdit

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const { id,title, body, image_data } = await req.json();

        let edit_news;

        if (image_data) {
            edit_news = await db.query(
                `UPDATE news SET title=$1, body=$2, image_data=$3 WHERE
id=$4;`,
                [title, body, image_data,id]
            );
        } else {
            edit_news = await db.query(
                `UPDATE news SET title=$1, body=$2 WHERE id=$3;`,
                [title, body, id]
            );
        }
        if (edit_news.rowCount)
```

```
            return NextResponse.json({ edit_news: true }, { status: 200 });
        return NextResponse.json({ edit_news: false }, { status: 500 });
    } catch (error) {
        console.log({ error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.11 OneNewsDelete

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function DELETE(req: NextRequest) {
    try {
        const {id} = await req.json();
        const delete_news = await db.query(`DELETE FROM news WHERE id =
${id};`);
        if (delete_news.rowCount)
            return NextResponse.json({ delete_news: true }, { status: 200 });
        return NextResponse.json(
            { error: `News ${id} doesn't exist` },
            { status: 500 }
        );
    } catch (error) {
        console.log({ "news[id].delete.error": error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.12 GradesGet

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const {id} = await req.json();
        const news = await db.query(`SELECT * FROM news WHERE id =
'${id}';`);
        if (news.rowCount)
            return NextResponse.json({
likes:news.rows[0].likes,dislikes:news.rows[0].dislikes }, { status: 200 });
        return NextResponse.json(
            { error: `Grades of news #${id} doesn't exist` },
            { status: 500 }
        );
    } catch (error) {
        console.log({ "news[id].error": error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.13 CommentsGet

```
import db from "@/server/db";
import { NextResponse } from "next/server";

export async function POST(req: NextResponse) {
    try {
        let com;
        const { id, username, page, limit, search } = await req.json();

        let request = `SELECT * FROM comments `;
        let options = ``;
        let order: string;
        if (id) options = `WHERE news_id = ${id}`;

        if (
            typeof username != "undefined" &&
            typeof username != undefined &&
            username.length > 0 &&
            id
        ) {
            options = options + ` AND username ILIKE '%${username}%'`;
        } else if (
            typeof username != "undefined" &&
            typeof username != undefined &&
            username.length > 0 &&
            !id
        ) {
            options = `WHERE username ILIKE '%${username}%'`;
        }
        if (
            typeof search != "undefined" &&
            typeof search != undefined &&
            search.length > 0
        ) {
            options =
                options +
                ` AND body ILIKE '%${search}%' OR mention ILIKE
'%${search}%'`;
        }

        if (page && limit) {
            order = ` ORDER BY id LIMIT ${limit} OFFSET ${(page - 1) *
limit}`;
        } else {
            order = ` ORDER BY id`;
        }

        const count = (await db.query(`SELECT COUNT(*) FROM comments
${options}`))
            .rows[0].count;

        com = await db.query(request + options + order);

        const pages = Math.ceil(count / limit) || 1;
        return NextResponse.json(
            { comment: com.rows, count, pages },
```

```
            { status: 200 }
        );

    } catch (error) {
        console.log({ comment_get: error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.14 CommentsPOST

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const { news_id, mention, username, data } = await req.json();

        let comment;
        if (mention) {
            comment = await db.query(
                `INSERT INTO comments (news_id, mention, username, body)
VALUES (${news_id},'${mention}', '${username}', '${data}');`
            );
            try {
                const notification = await db.query(
                    `INSERT INTO notifications (body, mention, com_body,
news_id, username) VALUES ('Your comment was
responded','${username}','${data}',${news_id},'${mention}');`
                );
            } catch (error) {
                console.log({ "notification.error": error });
                return NextResponse.json({ error: `${error}` }, { status: 400
});
            }
        } else {
            comment = await db.query(
                `INSERT INTO comments (news_id, username, body) VALUES
(${news_id},'${username}', '${data}');`
            );
            try {
                const { username: news_username } = (
                    await db.query(`SELECT * FROM news WHERE id =
'${news_id}';`)
                ).rows[0];

                if (news_username) {
                    const notification = await db.query(
                        `INSERT INTO notifications (body, com_body, news_id,
username) VALUES ('A comment was written on your
news','${data}',${news_id},'${news_username}');`
                    );
                }
            } catch (error) {
                console.log({ "notification.error": error });
                return NextResponse.json({ error: `${error}` }, { status: 400
});
            }
```

```
        }
        if (comment.rowCount)
            return NextResponse.json({ comment: true }, { status: 200 });
    } catch (error) {
        console.log({ comment: error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.15 CommentsLike

```
import { NextRequest, NextResponse } from "next/server";
import {
    LikeCom,UndislikeCom,UnlikeCom,CheckComLike,CheckComDislike
} from "../service/ComGrades";

export async function POST(req: NextRequest) {
    try {
        const { com_id, username } = await req.json();


        const like_check = await CheckComLike({com_id, username});

        const dislike_check = await CheckComDislike({com_id, username});

        if (!like_check) {
            if (dislike_check) {
                await UndislikeCom(com_id, username);
            }
            const like = await LikeCom(com_id, username);
            if (like) return NextResponse.json({ res: true }, { status: 200
});
        }
        await UnlikeCom(com_id, username);
        return NextResponse.json({ res: false }, { status: 200 });
    } catch (error) {
        console.log({ com_like: error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.16 CommentsDislike

```
import { NextRequest, NextResponse } from "next/server";
import {
    DislikeCom,UndislikeCom,CheckComLike,CheckComDislike,UnlikeCom
} from "../service/ComGrades";

export async function POST(req: NextRequest) {
    try {
        const { com_id, username } = await req.json();

        const like_check = await CheckComLike({com_id,username});
```

```
        const dislike_check = await CheckComDislike({com_id, username});

        if (!dislike_check) {
            if (like_check) {
                await UnlikeCom(com_id, username);
            }
            const dislike = await DislikeCom(com_id, username);
            if (dislike) return NextResponse.json({ res: true }, { status:
200 });
        }
        await UndislikeCom(com_id, username);
        console.log({like_check,dislike_check});
        return NextResponse.json({ res: false }, { status: 200 });
    } catch (error) {
        console.log({ com_dislike: error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.17 CommentsGradesGet

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const {id} = await req.json();
        const com = await db.query(`SELECT * FROM comments WHERE id =
'${id}';`);
        if (com.rowCount)
            return NextResponse.json(
                { likes: com.rows[0].likes, dislikes: com.rows[0].dislikes },
                { status: 200 }
            );
        return NextResponse.json(
            { error: `Grades of comment #${id} doesn't exist` },
            { status: 500 }
        );
    } catch (error) {
        console.log({ "news[id].error": error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.18CommentsDelete

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function DELETE(req: NextRequest) {
    try {
        const { id } = await req.json();
        console.log({id});
```

```
        const deleted = await db.query(`DELETE FROM comments WHERE id =
${id};`);
        console.log({deleted});
        if (deleted.rowCount)
            return NextResponse.json({ deleted: true }, { status: 200 });
        return NextResponse.json({ deleted: false }, { status: 500 });
    } catch (error) {
        console.log({ comment_delete: error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.19 NotiGet

```
import db from "@/server/db";
import { NextRequest, NextResponse } from "next/server";

export async function POST(req: NextRequest) {
    try {
        const { username, page, limit, search } = await req.json();
        let noti;
        let request = `SELECT * FROM notifications `;
        let options = `WHERE username ILIKE '%${username}%' `;
        let order = ``;

        if (
            typeof search != "undefined" &&
            typeof search != undefined &&
            search.length > 0
        )
            options =
                options +
                `AND (body ILIKE '%${search}%' OR mention ILIKE '%${search}%'
OR com_body ILIKE '%${search}%')`;

        if (page && limit) {
            const offset = page > 1 ? (page - 1) * limit : 0;
            order = ` ORDER BY id DESC LIMIT ${limit} OFFSET ${offset};`;
        } else {
            order = ` ORDER BY id DESC;`;
        }

        noti = await db.query(request + options + order);
        const count = Number(
            (await db.query(`SELECT COUNT(*) FROM notifications
${options}`)).rows[0]
                .count
        );
        const pages = Math.ceil(count / limit) || 1;

        if (noti.rowCount)
            return NextResponse.json(
                { noti: noti.rows, count, pages },
                { status: 200 }
            );
        return NextResponse.json(
            { error: `Notification doesn't exist` },
            { status: 500 }
```

```
        );
    } catch (error) {
        console.log({ "news.error": error });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.20 NotiRead

```
import {NextRequest, NextResponse} from "next/server";
import db from "@/server/db";

export async function POST(req: NextRequest) {
    try {
        let read
        const { id,username } = await req.json();
        const users = await db.query(`SELECT * FROM notifications where
id='${id}' AND username ILIKE '%${username}%'`);
        if (users.rowCount)
            read = await db.query(`UPDATE notifications SET is_read = NOT
is_read WHERE id = '${id}'`);
            if(read.rowCount) return NextResponse.json(true);
        return false;
    } catch (error) {
        console.log({ error: `${error}` }, { status: 400 });
        return NextResponse.json({ error: `${error}` }, { status: 400 });
    }
}
```

Б.21 NotiDelete

```
import {NextRequest, NextResponse} from "next/server";
import db from "@/server/db";

export async function DELETE(req: NextRequest) {
    try {
        let read
        const {id, username} = await req.json();
        const users = await db.query(`SELECT * FROM notifications where
id='${id}' AND username ILIKE '%${username}%'`);
        if (users.rowCount)
            read = await db.query(`DELETE FROM notifications WHERE id =
'${id}'  AND username ILIKE '%${username}%'`);
        if (read.rowCount) return NextResponse.json(true);
        return false;
    } catch (error) {
        console.log({error: `${error}`}, {status: 400});
        return NextResponse.json({error: `${error}`}, {status: 400});
    }
}
```

Б.21 grades.ts

```
import db from "@/server/db";

export const CheckDislike = async (news_id: number, username: string) => {
    try {
        const user = await db.query(
            `SELECT * FROM users WHERE username = '${username}'`
        );
        return user.rows[0].dislikes.includes(news_id);
    } catch (error) {
        console.log({error})
        throw new Error(`${error}`);
    }
};
export const CheckLike = async (news_id: number, username: string) => {
    try {
        const user = await db.query(
            `SELECT * FROM users WHERE username = '${username}'`
        );
        return user.rows[0].likes.includes(news_id);
    } catch (error) {
        console.log({error})
        throw new Error(`${error}`);
    }
};
export const DislikeNews = async (news_id: number, username: string) => {
    try {
        const news_dislikes = await db.query(`UPDATE news
    SET dislikes = dislikes + 1
    WHERE id = ${news_id};`);
        if (news_dislikes.rowCount) {
            const user_dislikes = await db.query(
                `UPDATE users SET dislikes = array_append(dislikes,
${news_id}) WHERE username = '${username}';`
            );
            if (user_dislikes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({error})
        throw new Error(`${error}`);
    }
};
export const LikeNews = async (news_id: number, username: string) => {
    try {
        const news_likes = await db.query(`UPDATE news
    SET likes = likes + 1
    WHERE id = ${news_id};`);
        if (news_likes.rowCount) {
            const user_likes = await db.query(
                `UPDATE users SET likes = array_append(likes, ${news_id})
WHERE username = '${username}';`
            );
            if (user_likes.rowCount) {
                return true;
            }
```

```
        }
        return false;
    } catch (error) {
        console.log({error})
        throw new Error(`${error}`);
    }
};
export const UndislikeNews = async (news_id: number, username: string) => {
    try {
        const news_undislikes = await db.query(
            `UPDATE news SET dislikes = dislikes - 1 WHERE id = ${news_id};`
        );
        if (news_undislikes.rowCount) {
            const user_undislikes = await db.query(
                `UPDATE users SET dislikes = array_remove(dislikes,
${news_id}) WHERE username = '${username}';`
            );
            if (user_undislikes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({error})
        throw new Error(`${error}`);
    }
};
export const UnlikeNews = async (news_id: number, username: string) => {
    try {
        const news_unlikes = await db.query(
            `UPDATE news SET likes = likes - 1 WHERE id = ${news_id};`
        );
        if (news_unlikes.rowCount) {
            const user_unlikes = await db.query(
                `UPDATE users SET likes = array_remove(likes, ${news_id})
WHERE username = '${username}';`
            );
            if (user_unlikes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({error})
        throw new Error(`${error}`);
    }
};
```

Б.22 GetUser.ts

```
import db from "@/server/db";

export const GetUser = async ({data}: { data: string }) => {
    try {
        const usersByName = await db.query(
            `SELECT * FROM users WHERE username = '${data}';`
        );
        if (usersByName.rowCount) return usersByName.rows[0];
```

```
        const usersByMail = await db.query(
            `SELECT * FROM users WHERE email = '${data}';`
        );
        if (usersByMail.rowCount) return usersByMail.rows[0];
        return {user:[]};
    } catch (error) {
        console.log({error});
        throw new Error(`${error}`);
    }
}
```

Б.23 ComGrades

```
import db from "@/server/db";

export const CheckComDislike = async (
    {
        com_id,
        username,
    }: {
        com_id: number;
        username: string;
    }) => {
    try {
        const dislike_check = (
            await db.query(`SELECT EXISTS (
      SELECT
      FROM users
      WHERE username = '${username}'
      AND ${com_id} = ANY(com_dislikes)
  );`)
        ).rows[0].exists;
        return dislike_check;
    } catch (error) {
        console.log({error});
        throw new Error(`${error}`);
    }
};
export const CheckComLike = async (
    {
        com_id,
        username,
    }: {
        com_id: number;
        username: string;
    }) => {
    try {
        const like_check = (
            await db.query(`SELECT EXISTS (
      SELECT
      FROM users
      WHERE username = '${username}'
      AND ${com_id} = ANY(com_likes)
  );`)
        ).rows[0].exists;
        return like_check;
    } catch (error) {
        console.log({error});
```

```
        throw new Error(`${error}`);
    }
};
export const DislikeCom = async (news_id: number, username: string) => {
    try {
        const news_dislikes = await db.query(`UPDATE comments
    SET dislikes = dislikes + 1
    WHERE id = ${news_id};`);
        if (news_dislikes.rowCount) {
            const user_dislikes = await db.query(
                `UPDATE users SET com_dislikes = array_append(com_dislikes,
${news_id}) WHERE username = '${username}';`
            );
            if (user_dislikes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({error});
        throw new Error(`${error}`);
    }
};
export const LikeCom = async (id: number, username: string) => {
    try {
        const news_likes = await db.query(`UPDATE comments
    SET likes = likes + 1
    WHERE id = ${id};`);
        if (news_likes.rowCount) {
            const user_likes = await db.query(
                `UPDATE users SET com_likes = array_append(com_likes, ${id})
WHERE username = '${username}';`
            );
            if (user_likes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({"like_com.error": error});
        throw new Error(`${error}`);
    }
};
export const UndislikeCom = async (news_id: number, username: string) => {
    try {
        const news_undislikes = await db.query(
            `UPDATE comments SET dislikes = dislikes - 1 WHERE id =
${news_id};`
        );
        if (news_undislikes.rowCount) {
            const user_undislikes = await db.query(
                `UPDATE users SET com_dislikes = array_remove(com_dislikes,
${news_id}) WHERE username = '${username}';`
            );
            if (user_undislikes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({error});
        throw new Error(`${error}`);
    }
};
```

```
export const UnlikeCom = async (news_id: number, username: string) => {
    try {
        const news_unlikes = await db.query(
            `UPDATE comments SET likes = likes - 1 WHERE id = ${news_id};`
        );
        if (news_unlikes.rowCount) {
            const user_unlikes = await db.query(
                `UPDATE users SET com_likes = array_remove(com_likes,
${news_id}) WHERE username = '${username}';`
            );
            if (user_unlikes.rowCount) {
                return true;
            }
        }
        return false;
    } catch (error) {
        console.log({error});
        throw new Error(`${error}`);
    }
};
```

Б.24 Функція UserAuth

```
export type UserAuthProps = {
    data: string;
    password: string;
};

const UserAuth = async ({ data, password }: UserAuthProps) => {
    const res = await fetch(`http://localhost:3000/api/UserAuth`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ data, password }),
    });
    if (!res.ok) {
        if (res.status === 500) {
            throw new Error(
                `${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        } else {
            throw new Error(`Failed to authenticate. Error code:
${res.status}`);
        }
    }

    const userJSON = await res.json();
    return await userJSON.user;
};
export {UserAuth}
```

Б.25 Функція UserDelete

```
const UserDelete = async ({ username }: {username:string}) => {
    const res = await fetch(`http://localhost:3000/api/UserAuth`, {
        cache: "no-cache",
        method: "DELETE",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ username }),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(`Failed to delete user. Error code:
${res.status}`);
        }
    }
    return await res.json();
};

export {UserDelete}
```

Б.26 Функція UserCreate

```
export type CreateUserProps = {
    username: string;
    password: string;
    first_name: string;
    last_name: string;
    email: string;
    pic?: string[];
};

const UserCreate = async (
    {
        username,
        password,
        first_name,
        last_name,
        email,
        pic,
    }: CreateUserProps) => {

    if (pic == undefined) {
        pic = []
    }

    const res = await fetch(`http://localhost:3000/api/UserCreate`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({
            username,
            password,
            first_name,
```

```
            last_name,
            email,
            pic,
        }),
    });


    if (!res.ok) {
        if (res.status === 500) {
            throw new Error(
                `${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        } else {
            throw new Error(`Failed to create new user: ${res.status}`);
        }
    }
    return res.json();
};

export {UserCreate}
```

Б.27 Функція GetNews

```
import moment from "moment/moment";
import {GetNewsProps} from "@/service/News/NewsType";

const GetNews = async ({page, limit, search}: GetNewsProps) => {
    // console.log({ page, limit, search });
    const res = await fetch(`http://localhost:3000/api/NewsGet`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({page, limit, search}),
    });
    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(`Failed to get news. Error code: ${res.status}`);
        }
    }

    const data = await res.json();

    data.news.forEach((item: { generation: any }) => {
        item.generation = moment(item.generation).format("DD-MM-YYYY
HH:mm:ss");
    });
    // data.count = Number(data.count)

    // console.log({ data });

    return data;
};
```

```
export {GetNews}
```

Б.28 Функція CreateNews

```
type CreateNewsProps = {
    title: string;
    body: string;
    username: string;
    image_data?: string[];
};

const CreateNews = async (
        {
            title,
            body,
            username,
            image_data,
        }: CreateNewsProps) => {
        const res = await fetch(`http://localhost:3000/api/NewsCreate`, {
            cache: "no-cache",
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({
                title,
                body,
                username,
                image_data,
            }),
        });
        if (!res.ok) {
            console.log(res);
            if (res.status === 500) {
                throw new Error(
                    `${await res.json().then((r_error) => {
                       return r_error.error;
                    })}`
                );
            } else {
                throw new Error(
                    `Error during create news request. Error code:
${res.status}`
                );
            }
        }
        const data = await res.json();
        return data.new_news;
    }
;

export {CreateNews}
```

Б.29 Функція LikedNewsGet

```
import {GetNewsProps} from "@/service/News/NewsType";
import moment from "moment";

type RatedNewsGetProps = GetNewsProps & {
    username: string;
};

const LikedNewsGet = async (
    {
        username,
        page,
        limit,
        search,
    }: RatedNewsGetProps) => {
    // console.log({ "s_news.liked_news": true, username, page, limit, search
});
    const res = await fetch(`http://localhost:3000/api/LikedNewsGet`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({
            username,
            page,
            limit,
            search,
        }),
    });

    if (!res.ok) {
        console.log(res);
        if (res.status === 500) {
            throw new Error(
                `${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        } else {
            throw new Error(
                `Error during liked news download request. Error code:
${res.status}`
            );
        }
    }

    const data = await res.json();

    data.news.forEach((item: { generation: any }) => {
        item.generation = moment(item.generation).format("DD-MM-YYYY
HH:mm:ss");
    });

    return data;
};

export {LikedNewsGet}

export type {RatedNewsGetProps}
```

Б.30 Функція DislikedNewsGet

```
import moment from "moment";
import {RatedNewsGetProps} from "@/service/News/LikedNewsGet";

const DislikedNewsGet = async (
    {
        username,
        page,
        limit,
        search,
    }: RatedNewsGetProps) => {
    // console.log({ "s_news.liked_news": true, username, page, limit, search
});
    const res = await fetch(`http://localhost:3000/api/DislikedNewsGet`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({
            username,
            page,
            limit,
            search,
        }),
    });

    if (!res.ok) {
        console.log(res);
        if (res.status === 500) {
            throw new Error(
                `${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        } else {
            throw new Error(
                `Error during liked news download request. Error code:
${res.status}`
            );
        }
    }

    const data = await res.json();

    data.news.forEach((item: { generation: any }) => {
        item.generation = moment(item.generation).format("DD-MM-YYYY
HH:mm:ss");
    });

    return data;
};

export {DislikedNewsGet}
```

Б.31 Функція GetOneNews

```
import moment from "moment";
```

```
const GetOneNews = async ({ id }: { id: number }) => {
    const res = await fetch(`http://localhost:3000/api/OneNewsGet`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({id}),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(
                `Failed to get news: ${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        }
    }

    const data = (await res.json()).news;

    // console.log({data});

    data.generation = moment(data.generation).format("DD-MM-YYYY HH:mm:ss");

    return data;
};

export {GetOneNews}
```

Б.32 Функція DeleteOneNews

```
const DeleteOneNews = async ({ id }: { id: number }) => {
    const res = await fetch(`http://localhost:3000/api/OneNewsDelete`, {
        cache: "no-cache",
        method: "DELETE",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({id}),
    });
    if (!res.ok) {
        console.log(res);
        if (res.status === 500) {
            throw new Error(
                `${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        } else {
            throw new Error(
                `Error during delete news request. Error code: ${res.status}`
            );
        }
    }
```

```
    const data = await res.json();
    return data.delete_news;
};

export {DeleteOneNews}
```

Б.33 Функція NewsLike

```
const NewsLike = async ({ news_id, username }:{news_id:number,
username:string}) => {
    const res = await fetch(`http://localhost:3000/api/NewsLike`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({
            news_id,
            username,
        }),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(`Failed to like news: ${res.status}`);
        }
    }

    return await res.json();
};

export {NewsLike}
```

Б.34 Функція NewsDislike

```
const NewsDislike = async ({ news_id, username }:{news_id:number,
username:string}) => {
    const res = await fetch(`http://localhost:3000/api/NewsDislike`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({
            news_id,
            username,
        }),
    });

    if (!res.ok) {
        if (res.status === 500) {
```

```
            return false;
        } else {
            throw new Error(`Failed to dislike news: ${res.status}`);
        }
    }

    return await res.json();
};

export {NewsDislike}
```

Б.35 Функція GetGrades

```
type GradesProps = {
    dislikes: number;
    likes: number;
};

type GetGradesProps = {
    id:number;
};

const GetGrades = async ({ id }: GetGradesProps) => {
    const res = await fetch(`http://localhost:3000/api/GradesGet`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({id}),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(
                `Failed to get news: ${await res.json().then((r_error) => {
                    return r_error.error;
                })}`
            );
        }
    }

    const  { likes, dislikes } = (await res.json());

    return { likes, dislikes };
};

export {GetGrades}
export type { GetGradesProps,GradesProps };
```

Б.36 Функція CommentsLike

```
const CommentsLike = async ({ com_id, username }:
{com_id:number,username:string}) => {
    const res = await fetch(`http://localhost:3000/api/CommentsLike`, {
```

```
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ com_id, username }),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(`Failed to like comment: ${res.status}`);
        }
    }

    return await res.json();
};

export {CommentsLike}
```

Б.37 Функція CommentsDislike

```
const CommentsDislike = async ({ com_id, username }:
{com_id:number,username:string}) => {
    const res = await fetch(`http://localhost:3000/api/CommentsDislike`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ com_id, username }),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(`Failed to dislike comment: ${res.status}`);
        }
    }

    return await res.json();
};

export {CommentsDislike}
```

Б.38 Функція GetComGrades

```
import {GetGradesProps} from "@/service/Grades/GradesGet";

type ComGradesProps = {
    likes: number;
```

```
    dislikes: number;
};

type ComGradesListProps = {
    username: string;
    page?: number;
    limit?: number;
    search?: string;
};

const GetComGrades = async ({id}: GetGradesProps) => {
    const res = await fetch(`http://localhost:3000/api/CommentsGradesGet`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({id}),
    });

    if (!res.ok) {
        if (res.status === 500) {
            // return false;
            return {likes: 0, dislikes: 0};
        } else {
            throw new Error(
                `Failed to get comment: ${await res.json().then((r_error) =>
{
                    return r_error.error;
                })}`
            );
        }
    }

    const {likes, dislikes} = await res.json();

    return {likes, dislikes};
};

export {GetComGrades}

export type {ComGradesProps, ComGradesListProps};
```

Б.39 Функція NotiGet

```
import {GetNotiProps, GetNotiRes} from "@/service/Noti/NotiType";
import moment from "moment";

const NotiGet = async (
    {
        username,
        limit,
        page,
        search,
    }: GetNotiProps) => {
    const res = await fetch(`http://localhost:3000/api/NotiGet`, {
        cache: "no-cache",
        method: "POST",
```

```
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({username, limit, page, search}),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return {noti: [], count: 0, pages: 0};
        } else {
            throw new Error(
                `Failed to check notification. Error code: ${res.status}`
            );
        }
    }
    const {noti, count, pages}: GetNotiRes = await res.json();
    noti.forEach((item: { generation: any }) => {
        item.generation = moment(item.generation).format("DD-MM-YYYY
HH:mm:ss");
    });
    return {noti, count, pages};
};

export {NotiGet}
```

Б.40 Store

```
import { create } from "zustand";

type UserState = {
    username: string | null;
    first_name: string | null;
    last_name: string | null;
    email: string | null;
    Auth: boolean;
    setUsername: (username: string | null) => void;
    setFirstName: (first_name: string | null) => void;
    setLastName: (last_name: string | null) => void;
    setEmail: (email: string | null) => void;
    setAuth: (Auth: boolean) => void;
};

export const useUser = create<UserState>((set) => ({
    username: null,
    first_name: null,
    last_name: null,
    email: null,
    Auth: false,
    setUsername: (username) => set((state) => ({ username })),
    setFirstName: (first_name) => set((state) => ({ first_name })),
    setLastName: (last_name) => set((state) => ({ last_name })),
    setEmail: (email) => set((state) => ({ email })),
        setAuth: (Auth) => set((state) => ({ Auth })),
}));
```

Б.41 Функція NotiRead

```
const NotiRead = async ({id,username}: {id:number,username:string}) => {
    const res = await fetch(`http://localhost:3000/api/NotiRead`, {
        cache: "no-cache",
        method: "POST",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({id,username}),
    });

    if (!res.ok) {
        if (res.status === 500) {
        } else {
            throw new Error(
                `Failed to check notification. Error code: ${res.status}`
            );
        }
    }
    return await res.json();
};

export {NotiRead}
```

Б.42 Функція NotiDelete

```
const NotiDelete = async ({ id,username }: {id:number,username:string}) => {
    const res = await fetch(`http://localhost:3000/api/NotiDelete`, {
        cache: "no-cache",
        method: "DELETE",
        headers: {
            "Content-Type": "application/json",
        },
        body: JSON.stringify({ id,username }),
    });

    if (!res.ok) {
        if (res.status === 500) {
            return false;
        } else {
            throw new Error(`Failed to delete user. Error code:
${res.status}`);
        }
    }
    return await res.json();
};
export {NotiDelete}
```