Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Кертиці Станіслава Володимировича*
(ПІБ)

академічної групи *122-19-1*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

освітньої програми *Комп'ютерні науки*
(назва освітньої програми)

на тему: Розробка соціальної мережі для оргранізації взаємодії між відвідувачами з використанням Nextjs, Nestjs та бази даних PostgreSQL

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	*доц. Спірінцев В.В.*			
розділів:				
спеціальний	*доц. Спірінцев В.В.*			
економічний	*проф. Вагонова О.Г.*			
Рецензент				
Нормоконтролер	*доц. Гуліна І.Г.*			

Дніпро
2023

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » 2023 року

ЗАВДАННЯ
на кваліфікаційну роботу

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *122-19-1* *Кертиці С.В.*
(група) (прізвище та ініціали)

тема кваліфікаційної роботи Розробка соціальної мережі для оргранізації взаємодії між відвідувачами з використанням Nextjs, Nestjs та бази даних PostgreSQL

затверджена наказом ректора НТУ «ДП» від *16.05.2023* № *350-с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів проєктно-технологічної практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі. Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2023 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки***	***27.05.2023 р.***

Завдання видав _____ *доц. Спірінцев В.В.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ *Кертиця С.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2023 р.*

Термін подання кваліфікаційної роботи до ЕК: *11.06.2023 р.*

РЕФЕРАТ

Пояснювальна записка: 86 сторінок, 45 рисунків, 22 джерела.

Об'єкт розробки: соціальна мережа для полегшення знайомств з людьми, обміну новин у вигляді громадських постів, коментування постів із можливістю ставити подобово та вести діалоги з різними людьми у приватних кімнатах.

Тема кваліфікаційної роботи: розробка соціальної мережі для взаємодії між відвідувачами з використанням Nextjs, Nestjs та бази даних PostgreSQL.

У вступі розглядається сучасний стан соціальних мереж, конкретизується мета сучасних соціальних мереж, актуальність та сфера їх застосування.

У першому розділі проаналізовано предметну галузь, визначено актуальність завдання та призначення розробки, сформульовано постановку задачі, зазначено вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі проаналізовано функціональне призначення системи. Описано використання технології та мову програмування. Описано структуру системи та алгоритми її функціонування. Обґрунтовано вхідні та вихідні дані програми. Перераховані технічні та програмні засоби. Дана інструкція з виклику та завантаження програми. Представлений інтерфейс користувача.

В економічному розділі визначено трудомісткість розробленої інформаційної системи, проведено підрахунок вартості роботи зі створення програми та розраховано час його створення.

Практичне значення полягає в організації зв'язку та спілкуванні між користувачами, що поліпшує соціальну сферу та взаємодію між ними.

Список ключових слів: СОЦІАЛЬНА МЕРЕЖА, ПОСТИ, ДРУЗІ, ПОВІДОМЛЕННЯ.

ABSTRACT

Explanatory note**:** 86 sides, 45 images, 22 sources.

Object of development: A social network for facilitating acquaintance with people, exchange of news in the form of public posts, commenting on posts with the possibility of posting daily and conducting dialogues with different people in private rooms.

The topic of the qualification work: development of a social network for interaction between visitors using Nextjs, Nestjs and the PostgreSQL database.

The introduction examines the current state of social networks, specifies the purpose of modern social networks, relevance and scope of their application.

In the first section, the subject area is analyzed, the relevance of the task and the purpose of the development is determined, the problem statement is formulated, and the requirements for software implementation, technologies and software tools are specified.

The second section analyzes the functional purpose of the system. The use of technology and the programming language are described. The structure of the system and algorithms of its functioning are described. The input and output data of the program are substantiated. Technical and software tools are listed. This is an instruction for calling and downloading the program. Presented user interface.

In the economic section, the labor intensity of the developed information system was determined, the cost of work on creating the program was calculated, and the time of its creation was calculated.

The practical significance lies in the organization of communication and communication between users, which improves the social sphere and interaction between them.

List of key words: SOCIAL MEMBERSHIP, FAST, FRIENDS, ANNOUNCEMENT.

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

TS – TypeScript;

БД – База Даних;

ПЗ – Програмне забеспечення;

ОС – Операційна Система;

UI – User Interface;

JWT – Json Web Token;

API - Application Programming Interface;

SRC – Source.

ЗМІСТ

РЕФЕРАТ..... 3
ABSTRACT..... 4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ..... 5
ВСТУП..... 8
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ..... 10
1.1. Загальні відомості з предметної області..... 10
1.2. Призначення розробки та область застосування..... 19
1.3. Підстава для розробки..... 19
1.4. Постановка завдання..... 19
1.5. Вимоги до програми або програмного виробу..... 20
1.5.1. Вимоги до функціональних характеристик..... 20
1.5.2. Вимоги до інформаційної безпеки..... 21
1.5.3. Вимоги до складу та параметрів технічних засобів..... 23
1.5.4. Вимоги до інформаційної та програмної сумісності..... 23
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ..... 24
2.1. Функціональне призначення системи..... 24
2.2. Опис застосованих математичних методів..... 24
2.3. Опис використаних технологій та мов програмування..... 25
2.4. Опис структури програми та алгоритмів її функціонування..... 31
2.5. Обґрунтування та організація вхідних та вихідних даних системи... 41
2.6. Опис розробленої системи……..... 46
2.6.1. Використані технічні засоби..... 46
2.6.2. Використані програмні засоби..... 46
2.6.3. Виклик та завантаження програми..... 46
2.6.4. Опис інтерфейсу користувача..... 47
РОЗДІЛ 3. ЕКОНОМІЧНИЙ РОЗДІЛ..... 60

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту..... 60
3.2. Рахунок витрат на створення програми..... 63
ВИСНОВКИ..... 66
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ..... 67
Додаток А. Код програми..... 70
Додаток Б. Відгук керівника економічного розділу..... 85
Додаток В. Перелік файлів на диску..... 86

ВСТУП

Соціальні мережі стосуються онлайн-платформ або програм, які дозволяють людям зв’язуватися та взаємодіяти з іншими, як правило, шляхом створення профілів, обміну вмістом і участі в різних формах спілкування. Ці платформи змінили те, як люди спілкуються, обмінюються інформацією та будують стосунки в епоху цифрових технологій.

Соціальні мережі надають людям можливість спілкуватися та залишатися на зв’язку з друзями, родиною, колегами та однодумцями по всьому світу.

Користувачі можуть ділитися особистими оновленнями, фотографіями, відео та іншим вмістом, що дозволяє їм висловлюватися та розповідати про свій досвід та інтереси. Соціальні мережі сприяють відкриттю нових ідей, тенденцій, новин і можливостей, об’єднуючи користувачів із різними поглядами та джерелами інформації. Соціальні мережі змінили спосіб спілкування людей, перейшовши до цифрової та онлайн-взаємодії як основного засобу залишатися на зв’язку.Новини та інформація швидко поширюються через соціальні мережі, що дозволяє отримувати оновлення в реальному часі та громадянську журналістику, але також викликає занепокоєння щодо дезінформації та фейкових новин. Мережі надали людям платформу для керування своїми цифровими персонажами та особистими брендами, впливаючи на самовираження, управління репутацією та онлайн-ідентичність.

Однак соціальні мережі також мають негативні аспекти. Вони можуть стати причиною залежності від віртуального світу, зниження фізичної активності, а також призвести до порушення приватності та поширення негативного контенту. Крім того, соціальні мережі можуть стати засобом маніпуляції та поширення дезінформації.У світлі цих результатів важливо розвивати критичне мислення та усвідомлене використання соціальних

мереж. Необхідно бути більш обізнаними щодо приватності та безпеки в онлайн-просторі, а також вміти фільтрувати та перевіряти інформацію, яку ми отримуємо. Крім того, соціальні мережі можуть бути ефективним інструментом для освіти, розвитку бізнесу та соціальної активності, тому важливо шукати та використовувати їх позитивний потенціал.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1. Загальні відомості з предметної області

За основу данної кваліфікаційної роботи був взятий сайт VK.COM. Це найкращий приклад того, як на сьогоднішній день має виглядати соціальная мережа.

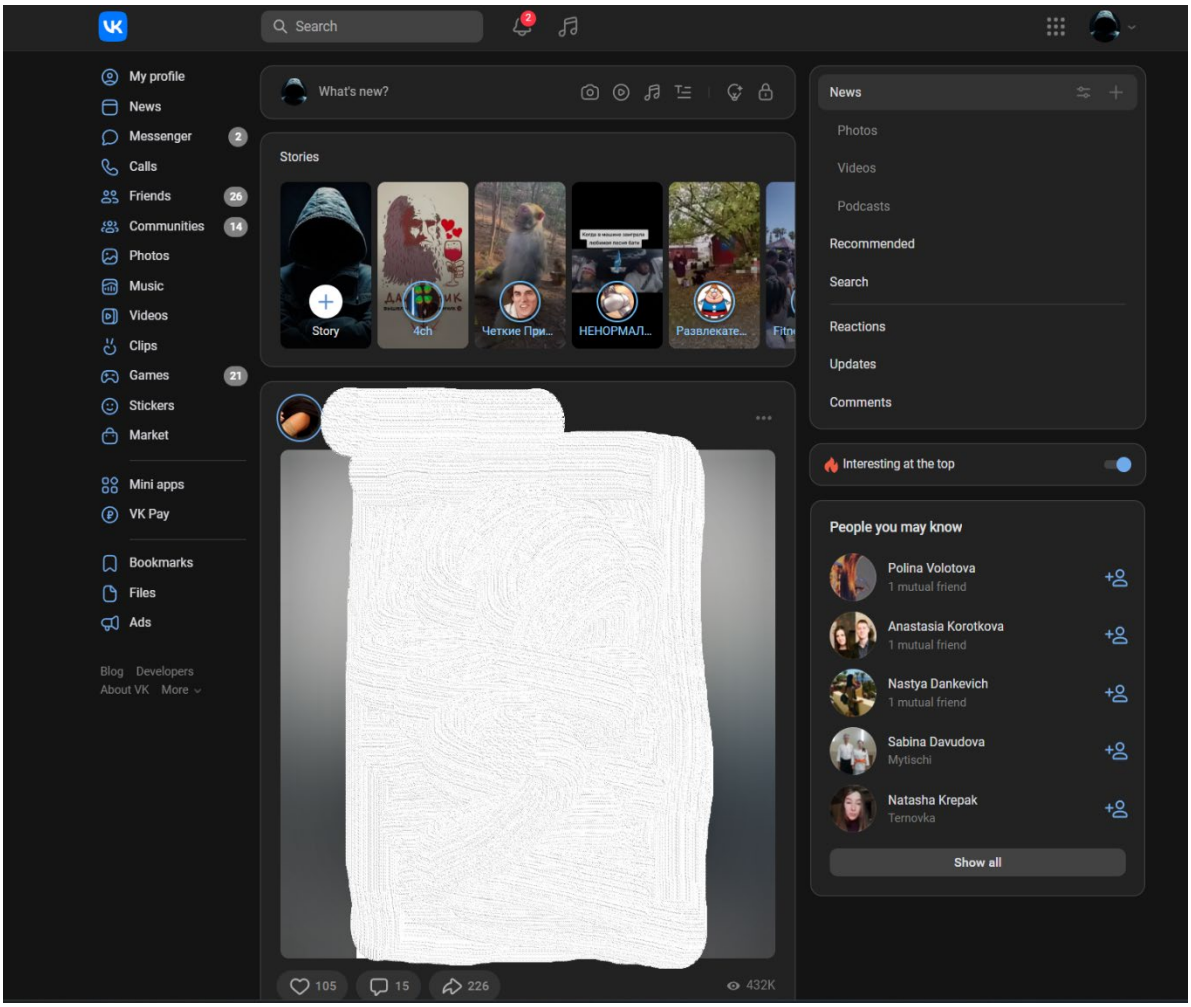

Рис. 1.1. Головна сторінка VK

Проаналізувавши соціальну мережу, зроблено висновок, що ця соціальна мережа дуже перевантажена зайвими сервісами, а саме:

- вбудовані дзвінки;
- вбудований додаток Messenger;
- власний фото/відео редактор;
- ігри;
- власний магазин;
- міні додатки.

Які не потрібні звичайним користувачам і багато коштують у розробці та підтримці функціоналу. І найголовніше, що цей сервіс не працює на території України, тому вирішено було прибрати весь зайвий функціонал та написати зручну соціальну мережу, яка функціонуватиме на території будь-якої держави.

Даний проект буде написаний з використанням клієнт-серверної архітектури див. рис. 1.2.

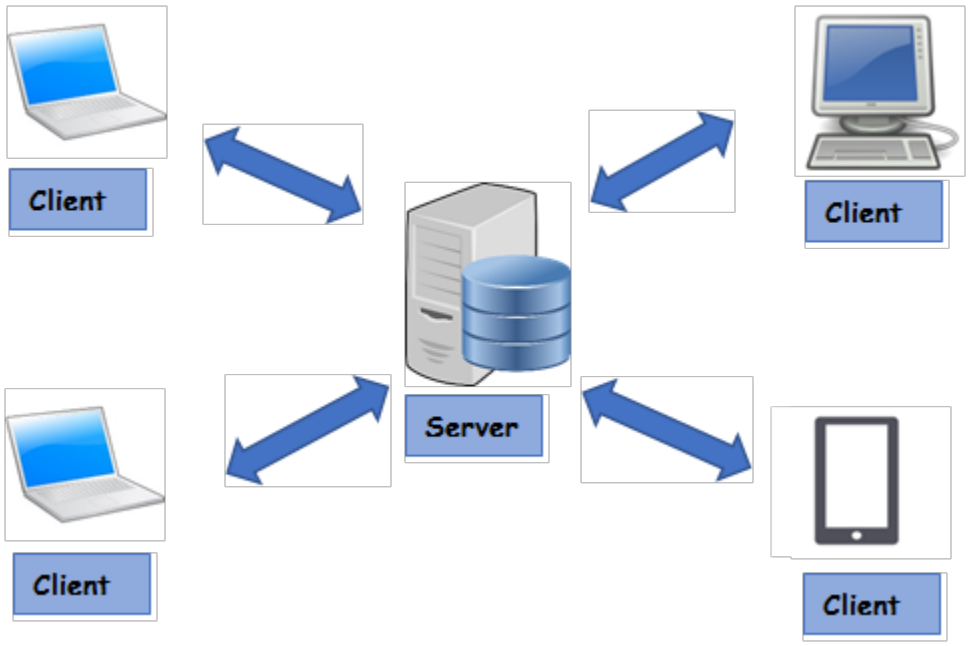

Рис. 1.2. Кліент-серверна архітектура

Клієнт-серверну архітектуру можна означити, як концепцію інформаційної мережі в якій основна частина її ресурсів зосереджена в серверах, обслуговуючих своїх клієнтів. Така архітектура визначає такі типи компонентів:

- набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
- набір клієнтів, які використовують сервіси, що надаються серверами;
- мережа, яка забезпечує взаємодію між клієнтами та серверами [1].

Правила взаємодії між клієнтом і сервером називаються протоколом обміну (протоколом взаємодії).

Модель клієнт-серверної взаємодії визначається перш за все розподілом обов'язків між клієнтом та сервером.

Логічно можна відокремити три рівні операцій:

- рівень представлення даних, який по суті являє собою інтерфейс; користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;
- прикладний рівень, який реалізує основну логіку застосунку і на якому здійснюється необхідна обробка інформації;

рівень управління даними, який забезпечує зберігання даних та доступ до них [2].

Роль - це функція сервера (наприклад поштовий, контролер домена). Один сервер може відігравати як одну так і декілька ролей одночасно.

Взалежності від ролі, сервісу який надається, розрізняють такі сервери:

Веб-сервер**.**

Сервер, що приймає HTTP-запити від клієнтів, зазвичай веб-браузерів, видає їм HTTP-відповіді, зазвичай разом з HTML-сторінкою, зображенням, файлом, медіа-потоком або іншими даними. Веб-сервер - основа Всесвітньої павутини.

Веб-сервером називають як програмне забезпечення, що виконує функції веб-сервера, так і комп'ютер, на якому це програмне забезпечення працює.

Клієнти дістаються веб-сервера за URL-адресою потрібної їм веб-сторінки або іншого ресурсу.

Сервер застосунків (Application Server).

Сервер, що виконує деякі прикладні програми. Термін також відноситься і до програмного забезпечення, що встановлено на такому сервері і забезпечує виконання прикладного ПЗ.

Сервери баз даних.

Сервери баз даних використовуються для обробки запитів користувача на мові SQL. При цьому СУБД знаходиться на сервері, до якого підключаються клієнтські додатки.

Файловий сервер (File Server).

Сервер що зберігає інформацію у вигляді файлів і представляє користувачам доступ до неї. Як правило файл-сервер забезпечує і певний рівень захисту від несакціонірованного доступу.

Поштовий сервер (Mail Server).

Дозволяє обслуговувати базові поштові скриньки ваших користувачів і дозволяє приймати і відправляти пошту з сервера. Вхідна пошта може зберігатися на сервері, а потім забиратися користувачем по протоколу POP3. Для ролі поштового сервера ви повинні мати: Активне з'єднання з інтернет Зареєстроване доменне ім'я Запис MX у провайдера для вашого поштового домен.

Термінальний сервер (Terminal Server).

Сервер, що надає клієнтам обчислювальні ресурси (процесорний час, пам'ять, дисковий простір) для вирішення завдань. Технічно термінальний сервер - надпотужний комп'ютер (або кластер), підключений до мережі з термінальними клієнтами - у котрих є, як правило, малопотужні або застарілі робочі станції або спецрішення для доступу до термінального сервера. Термінальний сервер служить для віддаленого обслуговування користувача з наданням робочого столу.

Remote Access/VPN Server.

Сервери віддаленого доступу і VPN надають точку входу в вашу мережу для віддалених користувачів. Використовуючи роль Remote Access / VPN Server, ви можете реалізувати протоколи маршрутизації для середовищ LAN і WAN. Ця роль підтримує модемні з'єднання і VPN через інтернет.

DHCP Server.

Сервер DHCP дозволяє клієнтам отримувати свій IP за потребою. Сервер DHCP також надає додаткову інформацію для конфігурації мережі - адреса серверів DNS, WINS і т.п.

Streaming Media Server.

Використовуються для управління і доставки мультимедійного контенту — потокового відео та аудіо — через інтранет або інтернет [3].

Переваги і недоліки клієнт - серверної архітектури:

Переваги:

- масштабованість: Клієнт-серверна архітектура дозволяє гнучко масштабувати систему. Сервери можуть бути налаштовані для обробки багатьох клієнтських запитів одночасно, що сприяє ефективному використанню ресурсів і забезпечує швидку обробку запитів;
- ментралізоване керування: Сервер відповідає за керування та зберігання даних, що дозволяє забезпечити їх централізоване управління і контроль. Це полегшує адміністрування системи та забезпечує збереження цілісності даних;
- безпека: Клієнт-серверна архітектура дозволяє встановлювати контрольований рівень доступу до ресурсів сервера. Сервер може використовувати механізми автентифікації та авторизації для перевірки прав доступу клієнтів. Це допомагає забезпечити безпеку і конфіденційність даних.

Недоліки:

- одночасність: Клієнт-серверна архітектура може винести обмеження на кількість одночасних з'єднань, які можуть бути оброблені сервером. Якщо навантаження стає дуже великим, сервер може стати недостатньою для обробки всіх запитів одночасно, що призводить до затримок або відмов у обслуговуванні;

- складність: Розробка та підтримка клієнт-серверних додатків може бути складною. Розробнику потрібно враховувати взаємодію між клієнтами та серверами, обробку помилок, синхронізацію даних та інші аспекти, що можуть бути складними;
- серверні вимоги: Клієнтам необхідне постійне з'єднання з сервером для обміну даними. Це може призвести до високих вимог щодо мережевого підключення, особливо для додатків, які вимагають великої пропускної здатності або мають велику кількість користувачів [4].

Для організації запитів клієнта до бази даних використовується RestAPI.

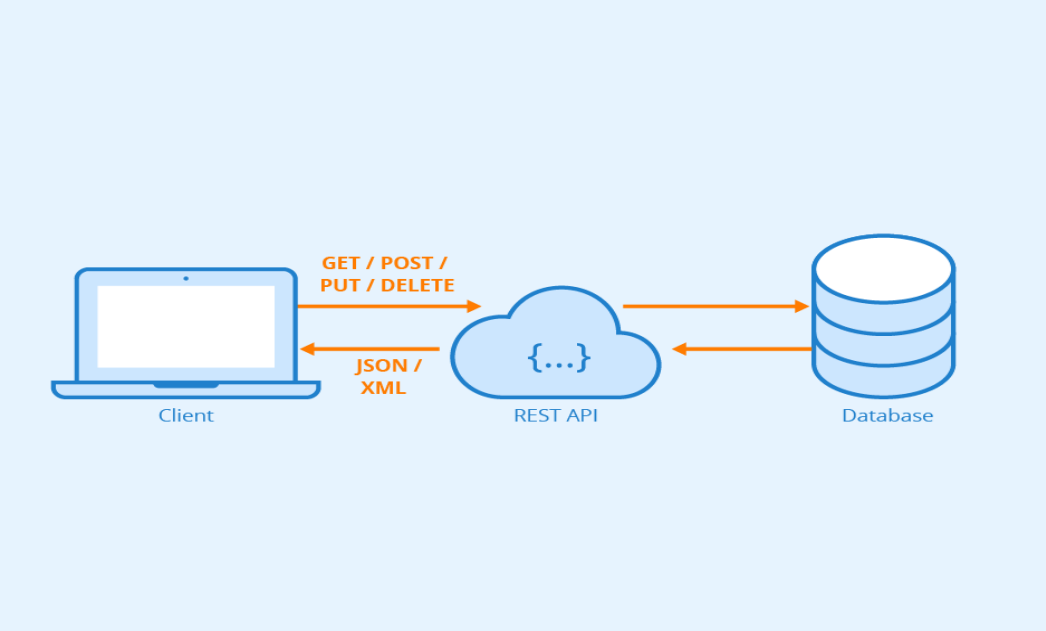

Рис. 1.3. Rest API

API побудовані на основі HTTP (протоколу передачі гіпертексту) і використовують стандартні методи HTTP, такі як GET, POST, PUT, PATCH і DELETE для виконання операцій над ресурсами. Ці ресурси ідентифікуються унікальними URL-адресами або URI (уніфікованими ідентифікаторами ресурсів). Кінцеві точки API представляють ці ресурси, і клієнти можуть взаємодіяти з ними, щоб отримати або маніпулювати даними [5].

Основні характеристики RESTful API включають:

Без стану: сервер не зберігає жодної інформації про клієнта між запитами. Кожен запит від клієнта повинен містити всю необхідну інформацію для його обробки сервером. Сервер відповідає запитуваними даними або відповідним кодом стану [6].

Уніфікований інтерфейс: API RESTful використовують стандартизований набір методів HTTP для взаємодії з ресурсами. Ці методи мають чітко визначену семантику: GET (отримати ресурс), POST (створити новий ресурс), PUT (оновити ресурс), PATCH (частково оновити ресурс) і DELETE (видалити ресурс).

На основі ресурсів: ресурси – це ключові об’єкти, доступні API. Кожен ресурс повинен мати унікальний ідентифікатор і представлення, яке може бути в різних форматах, таких як JSON, XML або HTML [7].

Для збереження даних користувачів був обраний тип бази даних Реляційний.

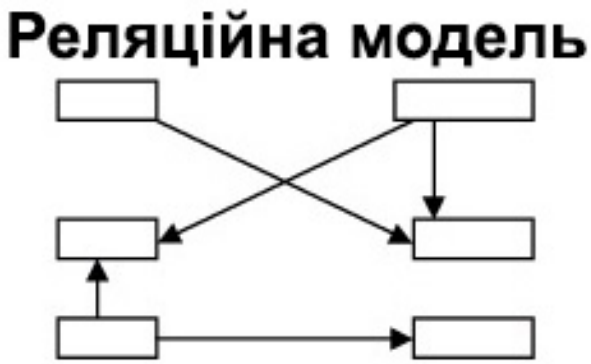

Рис. 1.4. Реляційна модель Бази Даних

Реляційна база даних – це структурована колекція даних, яка організована у вигляді таблиць. Кожна таблиця складається з рядків і стовпців, де рядки представляють конкретні записи, а стовпці визначають типи даних, які можуть бути збережені. Реляційна база даних використовує спеціальні зв’язки між таблицями, щоб увізуалізувати і зберігати залежності між даними [8].

Основні поняття реляційної бази даних.

Перед тим як розглядати деталі роботи з реляційною базою даних, давайте розглянемо основні поняття, які з нею пов’язані:

- таблиці: Реляційна база даних складається з однієї або декількох таблиць, де кожна таблиця має свою назву і структуру;
- рядки і стовпці: Рядок представляє окремий запис у таблиці, а стовпець визначає конкретний тип даних, який може бути збережений у цьому полі;
- ключі: Ключі використовуються для унікальної ідентифікації записів у таблиці. Первинний ключ визначає унікальний ідентифікатор запису, а зовнішній ключ встановлює зв’язок між таблицями;
- зв’язки між таблицями: Реляційна база даних використовує зв’язки між таблицями, щоб установити зв’язки між даними [9].

Найпоширенішим зв’язком є “один до багатьох”, де один запис у першій таблиці відповідає багатьом записам у другій таблиці.

Мова структуризованих запитів SQL.

Один із способів взаємодії з реляційною базою даних – це використання мови структуризованих запитів SQL (Structured Query Language). SQL надає набір команд, які можна використовувати для створення, модифікації і вибірки даних у реляційних базах даних.

Основні команди SQL включають:

- створення таблиць і внесення даних: за допомогою команд SQL можна створити таблиці з необхідною структурою і внести дані в ці таблиці;
- вибірка даних: запити SELECT використовуються для отримання певних даних з таблиць за певними умовами;
- зміна даних: команди UPDATE дозволяють змінювати вже існуючі дані у таблицях;
- видалення даних: команди DELETE використовуються для видалення даних з таблиць;

- інші команди: SQL також надає можливість створювати нові таблиці, змінювати структуру таблиць і виконувати різноманітні операції над даними [10].

Переваги реляційної бази даних.

Реляційна база даних має кілька переваг, які роблять її популярною в багатьох сферах:

- зручність використання: Реляційна модель даних є логічною і зрозумілою, що робить її зручною для розробників і користувачів;
- ефективність: Реляційна база даних дозволяє ефективно зберігати інформацію і проводити швидкий доступ до неї за допомогою оптимізованих запитів SQL;
- надійність: Реляційна база даних забезпечує цілісність даних за допомогою використання ключів і обмежень;
- масштабованість: Реляційна база даних може бути легко масштабована для обробки великих обсягів даних і великої кількості користувачів [11].

Недоліки реляційної бази даних:

- обмеження структури: Реляційні бази даних вимагають фіксованої структури, що може бути незручним у випадку, коли потрібно зберігати динамічні дані або дані зі змінюючоюся структурою;
- проблеми з продуктивністю при великих обсягах даних: При роботі з великими обсягами даних реляційні бази можуть стикатися з проблемами продуктивності, оскільки запити можуть вимагати значних ресурсів обчислювальної системи;
- складність моделювання зв’язків: У випадку складних залежностей між даними може виникати складність при моделюванні зв’язків між таблицями [12].

1.2. Призначення розробки та область застосування

Розробка соціальних мереж має на меті створення платформи, яка забезпечує зв'язок та взаємодію між користувачами через Інтернет. Основне призначення соціальних мереж полягає у спільноті та обміні інформацією між людьми, що мають спільні інтереси, думки або ділові зв'язки.

Розроблений проєкт повинен:

- відповідати сучасним стандартам;
- задовольняти вимоги продиктовані замовником;
- бути зручним для кінцевого користувача;
- мати інтуїтивно зрозумілий інтерфейс для користувача.

1.3. Підстава для розробки

Підставами для розробки (виконанням кваліфікаційної роботи) є:

- освітня програма спеціальності 122 «Комп’ютерні науки»;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» №350-с від 16.05.2023;
- завдання на кваліфікаційну роботу на тему «Розробка соціальної мережі для оргранізації взаємодії між відвідувачами з використанням Nextjs, Nestjs та бази даних PostgreSQL».

1.4. Постановка завдання

Завданням даної кваліфікаційної роботи є розробка кліент-серверного додатка, у вигляді REST Web Api.

Клієнт повинен мати змогу:

- створити аккаут зі своїми персональними данними;
- кастомізувати свій профіль: зміна аватарки, зміна обкладинки профілю;
- бачити весь список користувачів данної соціальної мережі та додавати їх до себе в друзі;
- створювати пости, редагувати та видаляти свої пости;
- лайкати пости;
- коментувати пости, редагувати та видаляти свої коментарі;
- обмінюватись повідомленнями с друзями.

1.6. Вимоги до програми або програмного виробу

В результаті Web - додаток повен відповідати наступним вимогам:

- швидке завантаження;
- обробка великої кількості запитів в одночас;
- зрозумілий та привабливий дизайн.

1.5.1. Вимоги до функціональних характеристик

Сервер повинен мати наступний функціонал:

- оптимізація запитів;
- реєстрація / авторизація користувачів з використанням JWT токенів;
- створення постів, редагування та їх видалення;
- лайк поста та його видалення;
- коментування поста, редагування та його видалення;
- створення повідомлень, їх зберігання, видалання та редагування.

1.5.2. Вимоги до інформаційної безпеки

Основні складові інформаційної безпеки дивитись на рис.1.5.

Рис.1.5. Складові інформаційної безпеки

Основними складовими інформаціної безпеки є доступність, конфіденційність та цілісність.

Доступність - це можливість за прийнятний час одержати необхідну інформаційну послугу.

Інформаційні системи створюються для отримання певних інформаційних послуг. Якщо за тими або іншими причинам надати ці послуги користувачам стає неможливо, це, очевидно, завдає збитку всім суб'єктам інформаційних відносин. Тому, не протиставляючи доступність решті аспектів, ми виділяємо її як найважливіший елемент інформаційної безпеки [13].

Особливо яскраво основна роль доступності виявляється в різного роду системах управління - виробництвом, транспортом тощо. Зовні менш драматичні, але також вельми неприємні наслідки - і матеріальні, і моральні - може мати тривала недоступність інформаційних послуг, якими користується велика кількість людей (продаж залізничних та авіаквитків, банківські послуги тощо).

Під цілісністю мається на увазі актуальність і несуперечність інформації, її захищеність від руйнування і несанкціонованої зміни.

Цілісність можна поділити на статичну (тобто незмінність інформаційних об'єктів) і динамічну (що відноситься до коректного виконання складних дій (транзакцій)). Засоби контролю динамічної цілісності застосовуються, зокрема, при аналізі потоку фінансових повідомлень з метою виявлення крадіжки, переупорядковування або дублювання окремих повідомлень [14].

Цілісність виявляється найважливішим аспектом ІБ в тих випадках, коли інформація служить керівництвом до дії.

Рецептура ліків, наказані медичні процедури, набір і характеристики комплектуючих виробів, хід технологічного процесу - все це приклади інформації, порушення цілісності якої може опинитися в буквальному розумінні смертельним. Неприємно і спотворення офіційної інформації, будь то текст закону або сторінка Web-сервера якої-небудь урядової організації.

Конфіденційність - це захист від несанкціонованого доступу до інформації.

Конфіденційність – найбільш опрацьований у нас в країні аспект інформаційної безпеки. На жаль, практична реалізація заходів по забезпеченню конфіденційності сучасних інформаційних систем натрапляє на серйозні труднощі. По-перше, відомості про технічні канали просочування інформації є закритими, так що більшість користувачів позбавлене можливості скласти уявлення про потенційні ризики. По-друге, на шляху призначеної для користувача криптографії як основного засобу забезпечення конфіденційності стоять численні законодавчі перепони і технічні проблеми [15].

Додаткою вимогою до інформаційної системи є захист від SQL injections.

SQL-ін'єкція (SQLi) - це вразливість веб-безпеки, яка дозволяє зловмиснику втручатися в запити, які додаток робить до своєї бази даних. Як правило, це дозволяє переглядати дані, які він зазвичай не може отримати. Це можуть бути інші користувачі, або будь-які інші дані, доступ до яких має сам додаток. У багатьох випадках зловмисник може змінювати або видаляти ці дані, викликаючи постійні зміни у вмісті або поведінці програми.

1.5.3. Вимоги до складу та параметрів технічних засобів

Мінімальні характеристики серверу на базі Linux або Windows:

- Процесор Core 2 Duo, Core i3, Core i5, Core i7 або аналогічні рішення процесорів AMD, бажано з тактовою частотою від 2,8 ГГц і вище;
- 4 Гб оперативної пам'яті й більше;
- Понад 20 Гб вільного місця на жорсткому диску для архівів бази даних.

Вимоги до інтернету:

- статична IP-адреса на серверному комп'ютері;
- дротове підключення 10-100-1000 Mbit/s.

1.5.4. Вимоги до інформаційної та програмної сумісності

Для локального старту коду необхідні такі компоненти:

- операційна система Windows/MacOS/Linux;
- пакетний менеджер Yarn;
- середовище Nodejs.

Для редагування коду необхідні такі компоненти:

- VScode, WebStorm, SublimeText, Atom і тд;
- фреймоворк Nextjs;
- фреймоворк Nestjs;
- Prisma з інтеграцією PostgreSQL.

Для коректної роботи системи слід користуватися такими браузерами:

- Google Chrome від 86 версії;
- Opera від 70 версії;
- Microsoft Edge від 85 версії.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

2.1. Функціональне призначення програми

Під час виконання кваліфікаційної роботи було розроблено веб-додаток у вигляді соціальної мережі для полегшення взаємодії людей, його головною метою є забезпечити зручний та якісний інструментарій для можливості користувачів дізнаватись новини інших без надобності зустрічатись.

Призначення розробленого додатку:

- створення постів;
- надання можливості маніпулювати створеними постами;
- Пошук та додавання людей в друзі;
- Збереження всіх даних в базу даних.

Для досягнення вище перерахованих функціональних можливостей розроблена програма повинна:

- Мати сумісність з стандартною апаратною конфігурацією обчислювальної машини;
- Мати підтримку програмного забезпечення, а саме операційних систем Windows 11/10;
- Мати підтримку сучасних браузерів.

2.2. Опис застосованих математичних методів

Дана розробка соціальної мережі використання спеціальних математичних методів не потребує, тому вони не будуть впроваджені в ході розробкі системи.

2.3. Опис використаних технологій та мов програмування

У ході розробці були використанні такі технології:

- фреймворк для React - Nextjs, для кліентської частини;
- препроцессор Sass для написання стилів;
- фреймворк для Nodejs - Nestjs, для серверної частини;
- postgreSQL база даних.

Next.js - це фреймворк, заснований на React, який дозволяє створювати веб-додатки з покращеною продуктивністю та покращеним користувальницьким досвідом за допомогою додаткових функцій попереднього рендерингу, таких як повноцінний рендеринг на стороні сервера (SSR) та статична генерація сторінок (SSG) [16].

Server Side Rendering. SSR дозволяє отримати доступ до всіх необхідних даних для побудови сторінки на сервері. Потім сторінка повністю відправляється назад у браузер і відразу відображається. SSR дозволяє веб-сторінкам завантажуватись за менший час і підвищує зручність роботи користувачів за рахунок підвищення швидкості відгуку.

SEO чи просто пошукова оптимізація. Використання SSR також дає вам перевагу в SEO, що допомагає вашому сайту займати вищі позиції на сторінках результатів пошукових систем. SSR підвищує рейтинг веб-сайтів для SEO, тому що вони завантажуються швидше та більше контенту сайту можна сканувати за допомогою трекерів SEO.

Переваги:

- Програми Next.js завантажуються значно швидше, ніж програми React завдяки вбудованому рендерингу на стороні сервера;
- Підтримує функцію експорту статичних сайтів;
- Автоматичне поділ коду для сторінок;
- Легко створювати внутрішні API-інтерфейси за допомогою вбудованих маршрутів API та створювати кінцеві точки API;

- Вбудована підтримка маршрутизації сторінок, CSS, JSX та TypeScript;
- Швидке додавання плагінів для налаштування Next.js відповідно до потреб вашої сторінки.

Недоліки:

Єдиним реальним недоліком Next.js є те, що це самостійний фреймворк, тобто він має певний метод і набір інструментів, які він хоче, щоб ви використовували для створення своїх додатків.

Однак можливості Next.js цілком підходять для більшості проектів [17].

React використовують для створення односторінкових та багатосторінкових програм, розробки великих сайтів.

Бібліотека призначена:

- для створення функціональних інтерактивних веб-інтерфейсів, працюючи з якими, не потрібно постійно оновлювати сторінку;
- швидкої та зручної реалізації окремих компонентів та сторінок цілком елементи в React легко використовувати повторно;
- легка розробка складних програмних структур — їх просто описувати якщо використовувати реалізований у React підхід;
- доопрацювання нової функціональності з будь-яким початковим стеком технологій: бібліотека не залежить від решти інструментарію і добре працюватиме, на чому б не було написано код;
- розробки односторінкових та багатосторінкових додатків (SPA та PWA). Це програми, які функціонують як програми та веб-сервіси та мають відповідний інтерфейс;
- роботи із серверною частиною сайту або розробки мобільних додатків. У таких випадках React використовують спільно з інструментами, що адаптують веб-технології для інших цілей [18].

Декларативність.

Декларативний стиль означає, що досить один раз описати, як виглядатимуть результати роботи коду — елементи у різних станах.

Йому не потрібно фокусуватися на способах досягнення результатів: більшість завдань виконає бібліотека. React.js автоматично оновлюватиме елементи в залежності від умов, головне завдання – грамотно описати їх. Зручний та зрозумілий підхід полегшує написання та налагодження коду.

Віртуальне DOM-дерево.

Будь-який веб-інтерфейс базується на HTML-документі та CSS-стилях, до яких підключений код JavaScript. Структура HTML-документа, точніше за його модель, називається DOM-деревом (DOM розшифровується як Document Object Mode, об'єктна модель документа). Це деревоподібна модель, в якій в ієрархічному вигляді зібрані всі елементи, що використовуються на сторінці [19].

Особливість React у тому, що він створює та зберігає в кеші віртуальне DOM-дерево – копію DOM, яка змінюється швидше, ніж реальна структура. Це потрібно, щоб швидко оновлювати сторінки. Якщо користувач виконає дію або настане будь-яка подія, DOM повинна змінитись, оскільки зміняться об'єкти на сторінці. Але реальна об'єктна модель може бути величезною, її оновлення є повільним процесом. Тому React працює не з нею, а з віртуальною копією у кеші, яка важить менше.

Коли відбувається подія, через яку код повинен оновити об'єкт, зміна швидко відображається у віртуальному DOM. Після цього оновлюється реальна об'єктна модель. Для користувача це означає, що зміни на сторінці відобразяться миттєво, а не після тривалого завантаження.

Оновлення DOM частинами.

Щоб покращити швидкодію, React оновлює DOM не повністю. Він зберігає в пам'яті дві полегшені копії: актуальну та попередню. Коли щось оновлюється, бібліотека порівнює версії між собою та змінює лише ту частину дерева, яка справді змінилася. Це потрібно, щоб не

перезавантажувати DOM повністю і не сповільнювати сторінку. Підхід здається складним, але важливий для оптимізації завантаження [20].

Можливість повторного використання компонентів.

React базується на компонентах – окремих елементах веб-інтерфейсу. Компоненти інкапсульовані, тобто самостійні: у кожному їх розміщені всі необхідні методи і дані.

Інкапсульовані самостійні компоненти можна використовувати повторно, розміщувати в іншому місці коду, іншому розділі або на іншій сторінці. Дані можна переносити по всьому додатку, використовувати поза межами DOM конкретної сторінки. Це прискорює розробку та скорочує кількість дій для створення функціонуючого інтерфейсу. Завдяки відсутності складних залежностей інкапсуляція також полегшує налагодження [21].

Синтаксис JSX.

JSX розшифровується як JavaScript XML. Це розширення мови JavaScript, яке допомагає описувати HTML-подібні елементи за допомогою коду React. За допомогою синтаксису на React створюють компоненти сторінки та гнучко керують ними.

Незважаючи на те, що елементи схожі на HTML, це, як і раніше, мова JavaScript з можливістю швидко і легко змінювати DOM за допомогою коду. І все ж таки JSX відтворюється як HTML: по суті розробник описує потрібний компонент на мові розмітки, а той залишається JavaScript-об'єктом з широкою функціональністю. Це зручно, спрощує програмування, але може заплутати початківців.

React Hooks.

У старих версіях керувати станами можна було за допомогою класів — спеціальних конструкцій, про які можна докладніше прочитати у статті про ОВП. Зараз у React.js є підтримка хуків — так називаються спеціальні функції-гачки, які чіпляються за стан елемента або за метод. Зміна стану або виклик методу «тягне» ці функції, і вони автоматично виконуються —

це допомагає уникнути використання класів, полегшує і спрощує написання коду [22].

Для написання стилів використанний препроцесор SASS.

Sass допомагає:

- зробити CSS-код зрозумілішим і простішим. Його легше масштабувати, оновлювати та підтримувати;
- розширити функціональність. За допомогою Sass можна використовувати CSS-константи, вбудовані функції, вкладені правила, домішки (змішані стилі), успадкування;
- уникнути багаторазового повторення однакових фрагментів коду. Це заощаджує час розробника, зменшує обсяг файлів стилів та прискорює обробку сторінок.

Найголовніше в NestJS – це модуль. Модуль являє собою абстрактну одиницю, яка інкапсулює в собі якийсь функціонал. Це може бути ваш API, робота з БД, домени, робота із зовнішніми сервісами, конфігурації та ін. Але найголовніше – це контекст, який створює модуль. Усередині модуля створюється контекст DI-контейнера і через нього дозволяються залежності (Injectable Services). В принципі, все те ж саме, що і в Angular 2+ за винятком, що замість параметра components у нас параметр controllers, в який можна додати наші контролери для RESTful API, рендерингу шаблонів або обробників повідомлень, і параметр providers, в який ми додаємо сервіси чи провайдери/фабрики для створення сервісів [23].

Контролери, сервіси, різні обробники влаштовані так само, як і, наприклад, Spring. Створюється клас, що позначається необхідним декоратором (Controller, Handler, Injectable). Методи класу теж позначаються декораторами (Get, Post, Put, Patch, Delete, Message тощо). Залежно вбудовуються через конструктор.

PostgreSQL не просто реляційна, а об'єктно-реляційна СУБД. Це дає йому деякі переваги над іншими базами даних SQL з відкритим вихідним кодом, такими як MySQL, MariaDB і Firebird.

Фундаментальна характеристика об'єктно-реляційної бази даних — це підтримка об'єктів користувача та їх поведінки, включаючи типи даних, функції, операції, домени та індекси. Це робить Постгрес неймовірно гнучким та надійним. Серед іншого, він вміє створювати, зберігати та отримувати складні структури даних. У деяких прикладах нижче ви побачите вкладені та складові конструкції, які не підтримуються стандартними РСУБД.

Існує великий список типів даних, які підтримує постгрес. Крім числових, з плаваючою точкою, текстових, булевих та інших очікуваних типів даних (а також безлічі їх варіацій), PostgreSQL може похвалитися підтримкою uuid, грошового, перерахованого, геометричного, бінарного типів, мережевих адрес, бітових рядків, текстового пошуку, xml, масивів, композитних типів та діапазонів, а також деяких внутрішніх типів для ідентифікації об'єктів та розташування логів. Заради справедливості варто сказати, що MySQL, MariaDB і Firebird теж мають деякі з цих типів даних, але тільки Постгрес підтримує їх усі.

Вся архітектура сайту пишется з використанням мови TypeScript.

З точки зору розробки програмного забезпечення, TypeScript пропонує багато переваг перед JavaScript:

- додаткова статична типізація. JavaScript — це мова з динамічними типами, що означає, що типи перевіряються, а помилки типів даних виявляються лише під час виконання. Це може бути дуже небезпечно та може призвести до помилок під час виробництва. TypeScript представляє необов’язкову стійку статичну типізацію: після оголошення змінна не змінює свій тип і може приймати лише певні значення;
- читабельність. Завдяки додаванню строгих типів та інших елементів, які роблять код більш самовиразним, ви можете побачити задум розробників, які спочатку написали код. Це особливо важливо для розподілених команд, які працюють над

одним проектом. Код, який говорить сам за себе, може компенсувати відсутність прямого спілкування між членами команди;

- підтримка IDE. Інформація про типи робить редактори та інтегровані середовища розробки (IDE) набагато кориснішими. Вони можуть запропонувати такі функції, як навігація кодом і автозаповнення, надаючи точні пропозиції;
- сила об'єктної орієнтації. TypeScript підтримує концепції об’єктно-орієнтованого програмування (ООП), такі як класи, інтерфейси, успадкування тощо. Парадигма ООП полегшує створення добре організованого масштабованого коду, і ця перевага стає більш очевидною, коли розмір і складність вашого проекту зростають.

2.4. Опис структури програми та алгоритми її функціонування

Схема використанної реляційної бази даних для розробки програми.

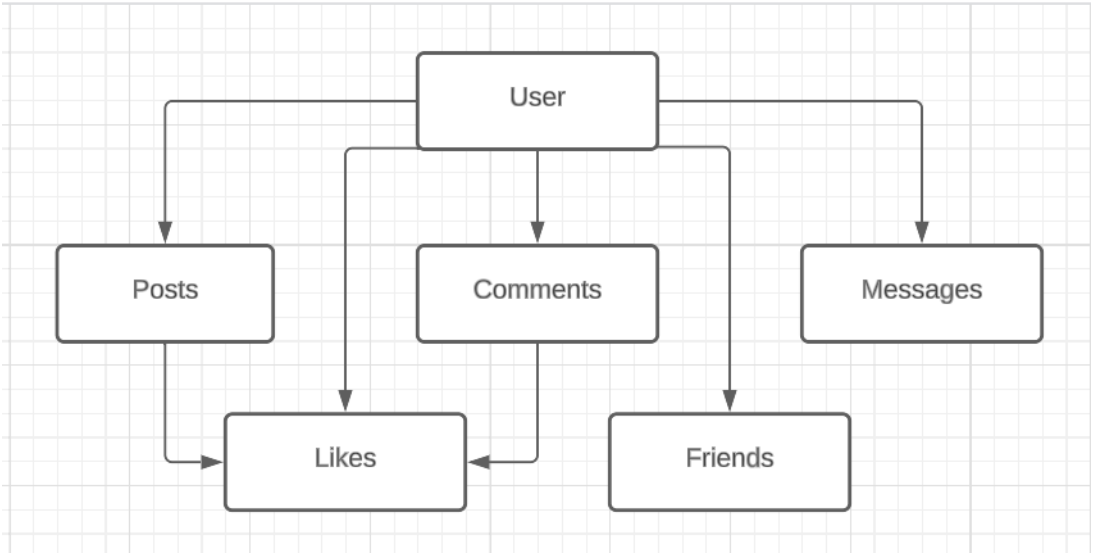

Рис.2.1. Схема бази даних

Використана база даних для розробки програми складається з таких сутностей:

- User;
- Posts;
- Comments;
- Messages;
- Friends;
- Likes.

Один користувач може мати багато постів, коментарів, повідомлень, друзів та лайок. Один пост має багато кометарів та лайок. Один комент має багато лайок.

Інформаційна система складається з двох основних частин, а саме серверної та клієнтської частини.

Почнемо із структури серверної частини. У пакеті dist зберігається готова версія сайту для публікації.

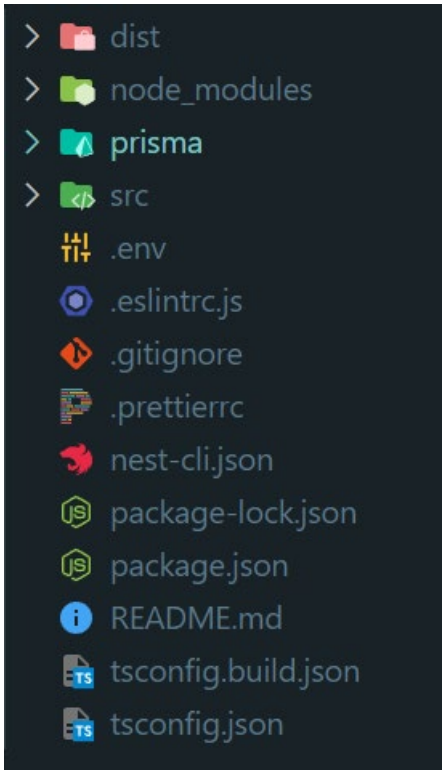

Рис. 2.2. Структура серверверної частини

Після ініціалізації проекту у папці має з'явитися файл package.json. package.json має залежності, які використовуються в проекті.

Файл .env використовується змінне оточення, у нього виноситься вся конфігурація.

Підключення до бази даних забезпечує файл prisma.service.ts.

```
@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  async onModuleInit() {
    await this.$connect();
  }

  async enableShutdownHooks(app: INestApplication) {
    this.$on('beforeExit', async () => {
      await app.close();
    });
  }
}
```

Рис. 2.3. Підключення до бази даних

Всі моделі можливих сутностей та їх зв’язки зберігаються в файлі schema.prisma.

```
model User {
  id         String   @unique @default(uuid())
  createdAt DateTime @default(now()) @map("created_at")
  updatedAt DateTime @updatedAt @map("updated_at")

  email    String @unique
  password String

  fullname   String
  avatarPath String @default("profile.png") @map("avatar_path")
  token      String

  posts    Post[]
  likes    Like[]
  comments Comment[]
}
```

Рис. 2.4. Структура моделі user

Вся основна логіка серверної частини знаходиться в папці source. А вхідний файл в программу – main.ts.

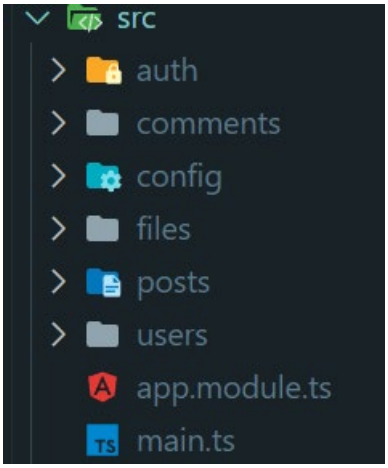

Рис. 2.5. Src та main.ts

Вся логіка поділенна на модулі, кожен мая відповідальність тільки за себе і навінь не знає про інші. Хоча є модулі які мають функції для повторного застосування в інших. Це зроблено щоб не дублювати зайвий код.

Кожен модуль ділиться на 3 частини:

- контроллер;
- коріний файл;
- сервіс.

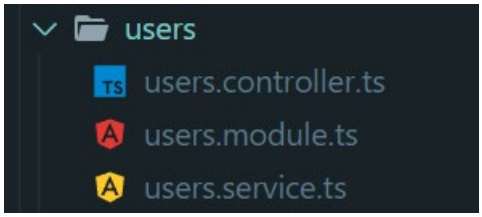

Рис. 2.6. Структура модуля

У вхідному файлі відбувається імпорт контроллера, сервіса. Можливий імпорт додаткових модулей, та експорт своїх.

```
@Module({
  providers: [UsersService],
  controllers: [UsersController],
  imports: [],
  exports: [UsersService],
})
export class UsersModule {}
```

Рис. 2.7. Структура корінного файла

У контролері зберігаються всі роути цього модуля. Контролери відповідають за обробку вхідних запитів (Request) та повернення відповідей (Response) клієнту.

Призначення контролера – приймати певні запити про програму. Механізм маршрутизації контролює, який контролер отримує запити. Часто кожен контролер має більше одного маршруту та різні маршрути можуть виконувати різні дії.

```
@Controller('users')
export class UsersController {
  constructor(private readonly users: UsersService) {}

  @Auth()
  @Get('profile')
  getProfile(@CurrentUser('id') userId: string) {
    return this.users.getProfile(userId);
  }
  @Auth()
  @Get()
  getAllUsers() {
    return this.users.getAllUsers();
  }
}
```

Рис. 2.8. Структура контроллера

У сервісі вже відбувається вся логіка взаємодії з базою даних. Отримання даних, додавання, видалення, чи зміннення.

```
@Injectable()
export class UsersService {
  constructor(private prisma: PrismaService) {}

  async getProfile(userId: string) {
    const user = await this.prisma.user.findUnique({
      where: { id: userId },
      include: {
        posts: true,
        likes: true,
      },
    });
    return this.returnUser(user, user.posts, user.likes);
  }

  async getAllUsers() {
    return this.prisma.user.findMany();
  }
```

Рис. 2.9. Структура сервіса

Аутентифікація є важливою частиною більшості програм. В данній роботі використанні JWT токени.

Вимоги до аутентифікації:

Дозволити користувачам аутентифікуватися за допомогою імейлу та паролю, повертаючи два JWT-токени (Acces та Refresh) для використання у наступних викликах захищених кінцевих точок API. Токен доступу живе як правило пару хвилин, коли сервер поверне помилку, її перехопить інтерсептор і запит на базу даних для оновлення токена доступу, відправивши токен для оновлення. Після вдалої операції сервер поверне нову пару згенерованих токенів.

Сама аутентифікація проходить наступним чином: сервер отримує мій пароль з фронту, спочатку порівнює имейлі, після вдалого парівніння бере пароль хешує його, та порівнює його з тим, що лежить у базі даних. Якщо хеші

співпали повертає у відповідь користувача.

```typescript
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    private configService: ConfigService,
    private prisma: PrismaService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: configService.get('jwt_secret'),
    });
  }

  async validate({ id }: Pick<User, 'id'>) {
    return this.prisma.user.findUnique({ where: { id } });
  }
}
```

Рис. 2.10. Структура аутентифікації

Тепер кліентська частина. В папці public зберігаються використанні іконки. Вся логіка зберігается в папці source.

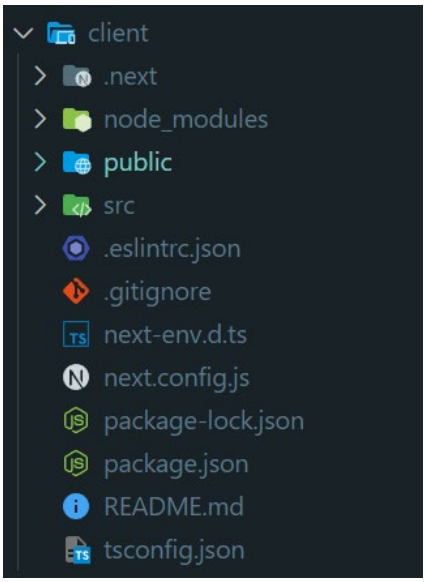

Рис. 2.11. Структура клінтської частини

Всі доступні маршрути сайту по яким можливо буде перейти користувач зберігаються в папці app. Вхідна сторінка – page.tsx.

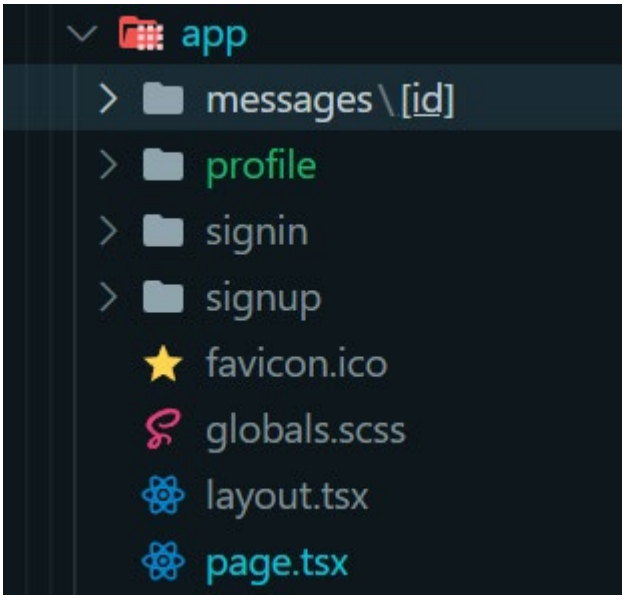

Рис. 2.12. Маршрути мережі

Так як технологія написання front-end була вибрана Nextjs, а це лише фреймворк для бібліотеки React, тому використовується підхід компонентної верстки. Коли кожна можлива функиція виноситься в новий компонент з можливістю використання його в будь-яких частинах сайту незалежно один від одного.

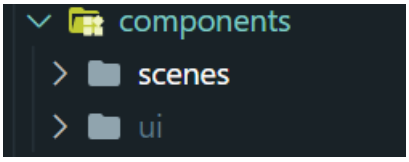

Рис. 2.13. Розділення сторінок та компонентів

В папці scenes зберігаються саме сторінки на які зможе перейти будь-який користувач.

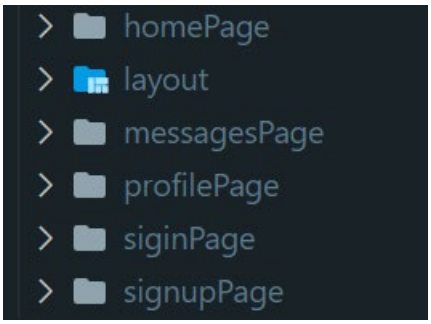

Рис. 2.14. Сторінки

В папці ui зберігаються саме компоненти які можна буде перевикористовувати в будь-якій частині сайту.

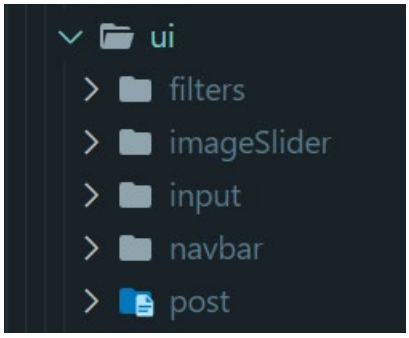

Рис. 2.15. Компоненти

Важливою частиною додатка є папка хуків, де зберігается взаємодія між клієнтом та сервером. Отправка даних на сервер, отримання даних з сервера, оновлення даних та їх видалення див.рис. 2.16.

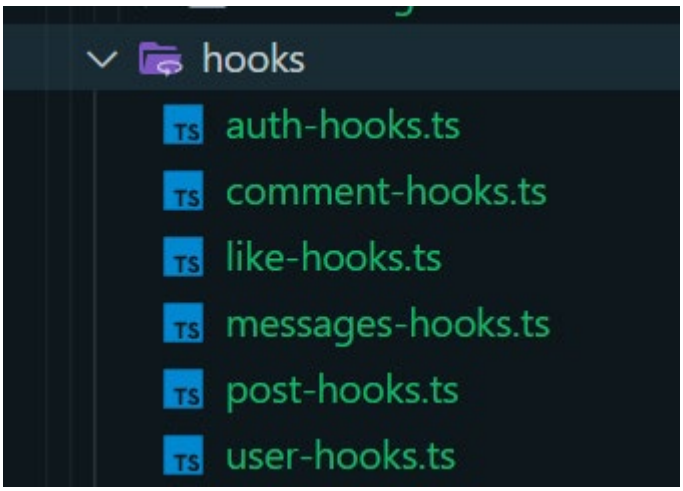

Рис.2.16. Хуки

Допоміжна папка сервісів где описані всі можливі кінцеві запити до сервера за домогою Axios,а саме GET, POST, PUT/PATCH, DELETE.

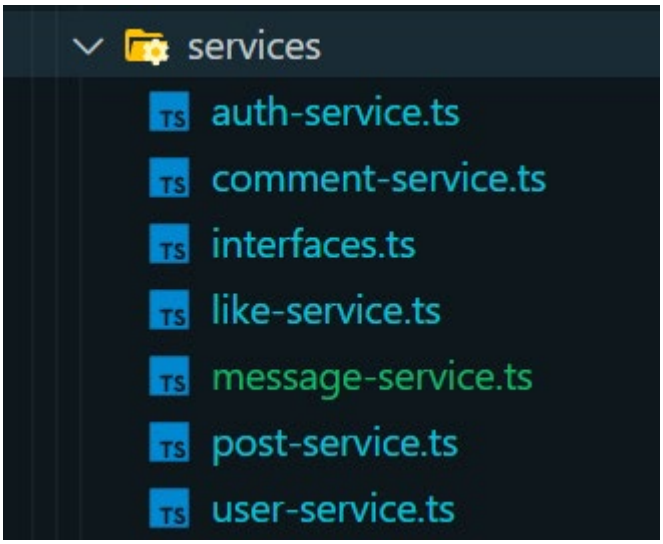

Рис.2.17. Сервіси

2.5. Обґрунтування та організація вхідних та вихідних даних програми

На вході програма буде отримувати такі дані:

для логіну:

- пошта користувача;
- пароль користувача.

для реэстрації:

- унікальна пошта пошта користувача;
- ім’я та прізвище користувача;

- унікальний номер користувача;
- пароль користувача.

для створення поста:

- текст;
- медіа файли (картинки / відео);
- та id користувача.

для створення коментаря:

- текст;
- id поста;
- id користувача.

На виході програма буде віддавати такі дані:

з вдалого логіну:

- данні користувача;
- токен доступу;
- токен оновлення.

з вдалої реэстрації:

- новий користувач;
- токен доступу;
- токен оновлення.

з вдалого створення поста:

- новий пост;
- автор створенного поста.

з вдалого створення коментаря:

- новий коментар;
- автор коментаря.

Тепер розглянемо вхідні та вихідні дані, та запити для кожної сутності бази даних детальніше.

Для логіну треба відправити Post запит який включає пошту та пароль на сервер з адресою http://localhost:5000/auth/signin

Рис.2.18. Вхідні дані логіну

Якщо пошта та пароль пройдуть всі перевірки, то повернеться користувач та пара токенів

Рис.2.19. Вихідні дані логіну

Для реєстрації треба відправити Post запит який включає пошту, пароль, ім’я, прізвище та номер телефону на сервер з адресою http://localhost:5000/auth/signup

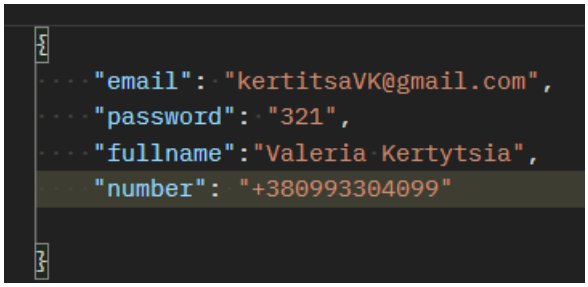

Рис.2.20. Вхідні дані реєстрації

Якщо пошта та номер телефону ще не заєстровані, то створюються новий користувач, два токени та повернуться у відповідь

Рис.2.21. Вихідні дані реєстрації

Для створення поста треба відправити Post запит який включає текст та медіа масив з фотографіями та відео, якщо ті були додані на сервер з адресою http://localhost:5000/post

Рис.2.22. Вхідні дані створення поста

Після вдалого запису медіа файлів на сервер, та створення самого поста повернеться створенний пост, масив кометарів, лайків та користувача, який створив пост.

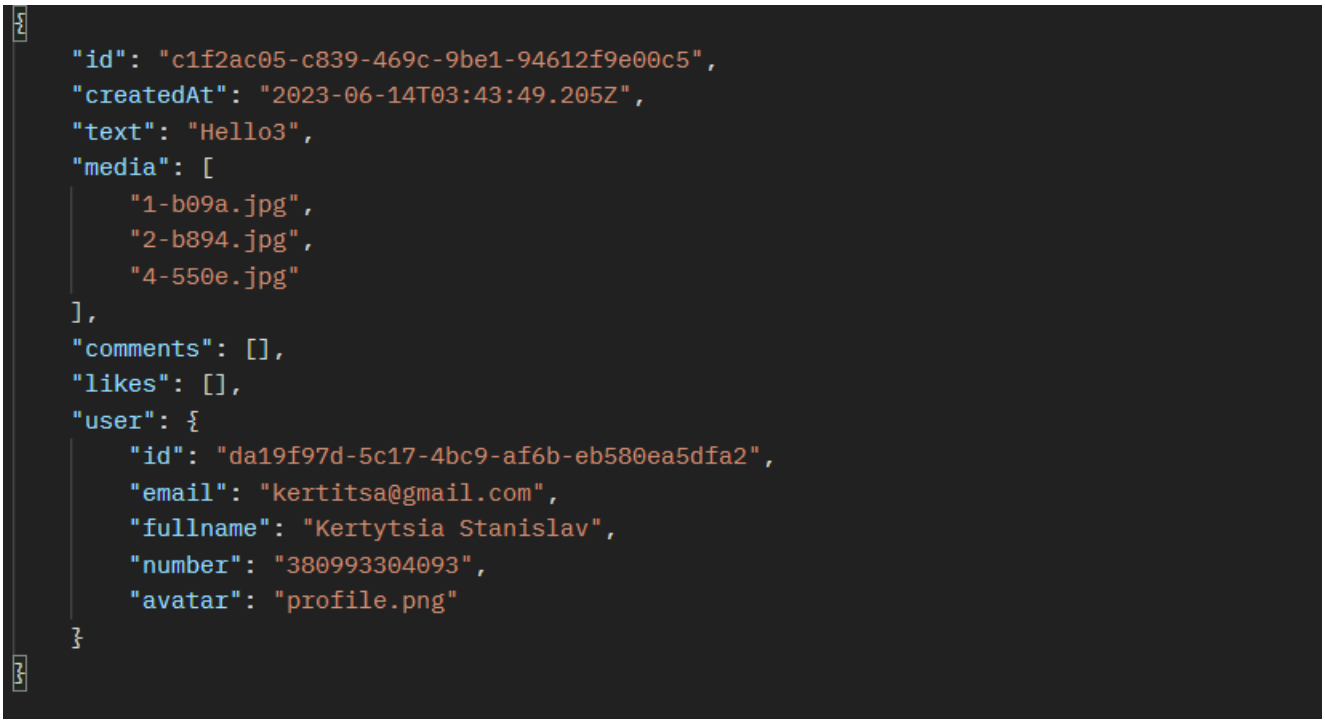

Рис.2.23. Вихідні дані створення поста

Для створення коментаря треба відправити Post запит який включає текст на сервер з адресою http://localhost:5000/comment/postid

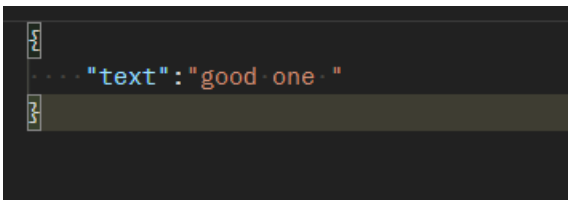

Рис.2.24. Вхідні дані коментаря

Після вдалого створення коментаря, сервер поверне створенний коментар, масив лайків та користувача який створив коментар.

Рис.2.25. Вихідні дані коментаря

2.6. Опис розробленої системи

2.6.1 Використані технічні засоби

Весь код був написаний з використанням ноутбука на базі процесора Intel i5 10300h, 16гб ОЗУ та відеокартою rtx2060 з Windows 11. Два додаткові монітори з діагоналю 27 дюймів, комп’ютерна миша та клавіатура.

2.6.2. Використані програмні засоби

Программа для написання коду VScode, программа для тестування запитів до сервера Postman, Prisma Web Client для обробки даних в БД, Vercel.com для розміщення клієнтської частини, та Railway.com для розміщення БД та серверної частини сайту.

2.6.3. Виклик та завантаження програми

Для локального завантаження проекту треба зайти в коріну папку, відкрити два будь-які можливі термінали та вписати:

- для першого – cd client; yarn install; yarn start:dev
- для другого – cd server; yarn install; yarn dev

Після ініціалізації файлів додаток відкриється у браузері на локальному сервері.

Для завантаження розміщенного проекту треба лише перейти за посиланням виданим сервісом де програма була розміщена.

2.6.4. Опис інтерфейсу користувача

Відкривши додаток вперше, користувач потрапляє на сторінку логіна.

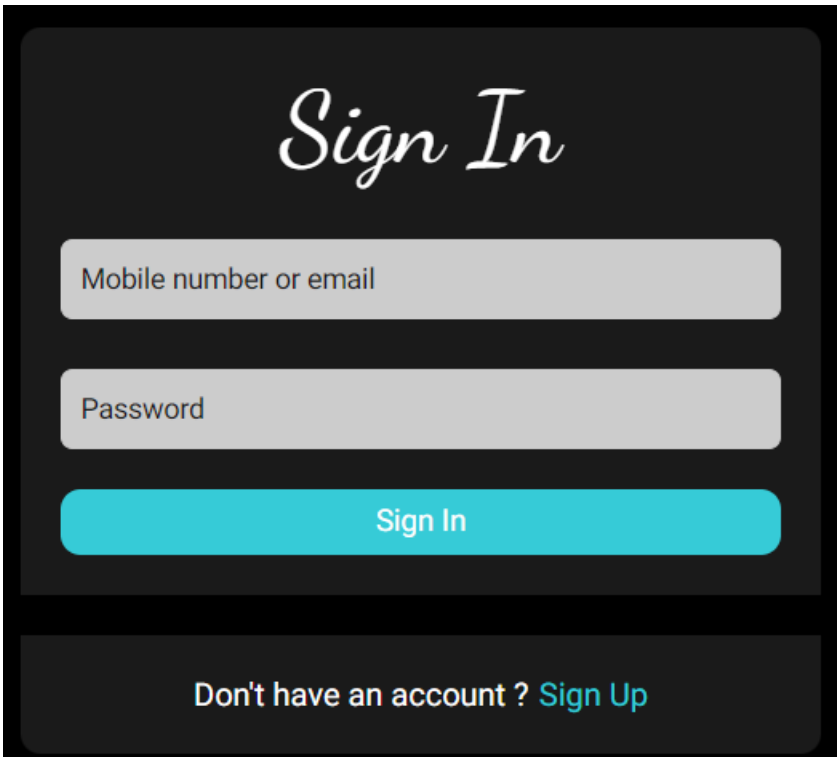

Рис.2.26. Форма логіну

Перевірка на заповнення всіх полів форми логіну.

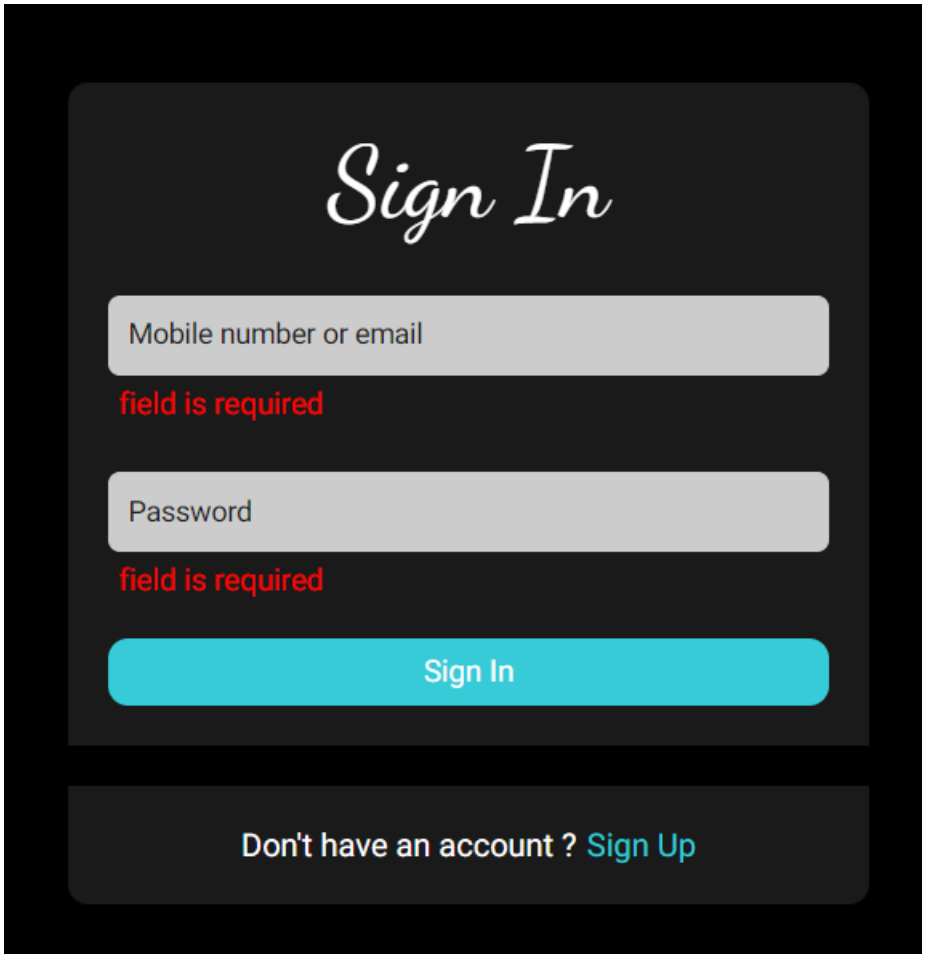

Рис.2.27. Перевірка заповнення

Якщо введені дані не пройшли перерівки зверху з’являється повідомлення, яке сповіщує, що дані введені не вірно.

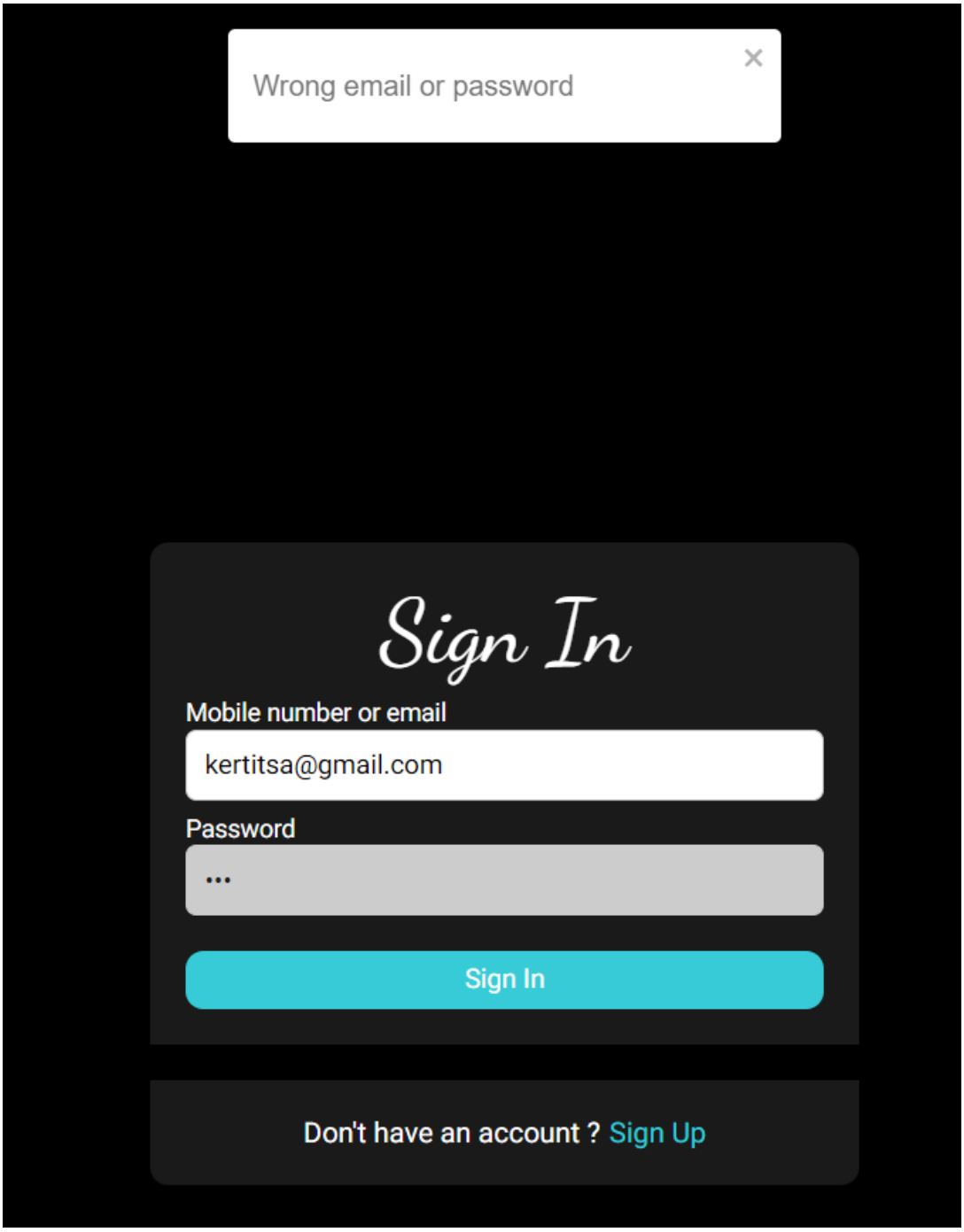

Рис.2.28. Неправильні дані

Зі сторінки логіну можна відразу же перейти на сторінку реєстрації.

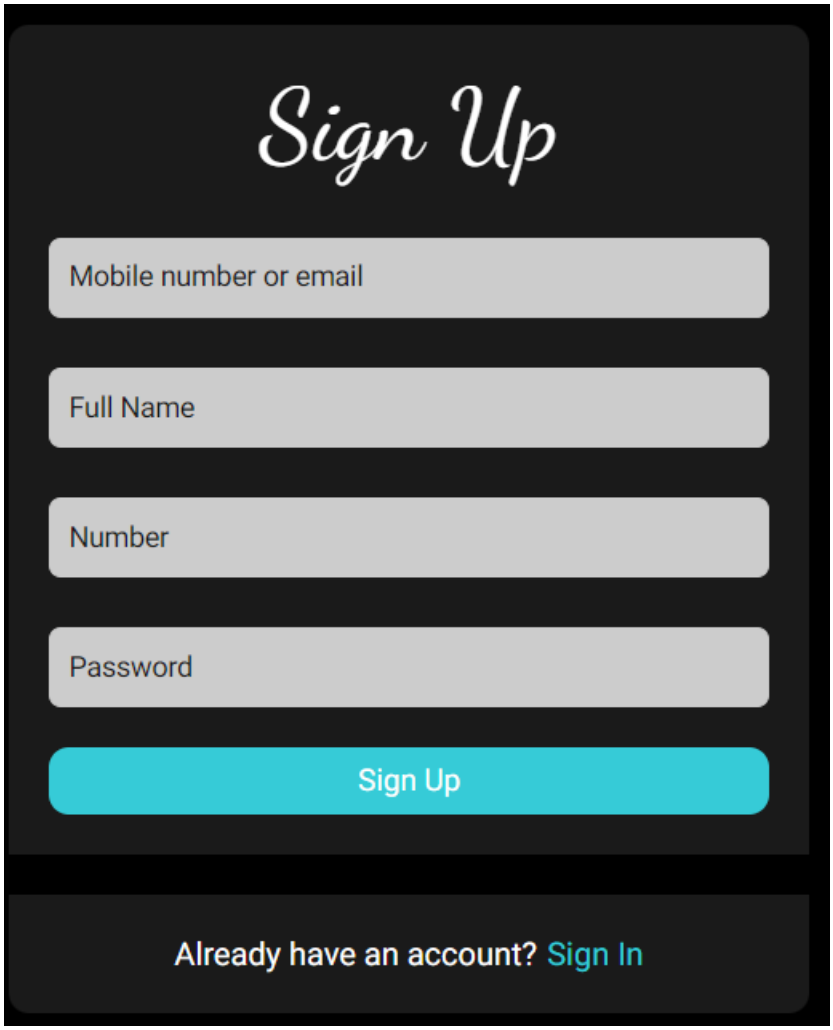

Рис.2.29. Форма реєстрації

Перевірка форми реєстрації на заповнення полів.

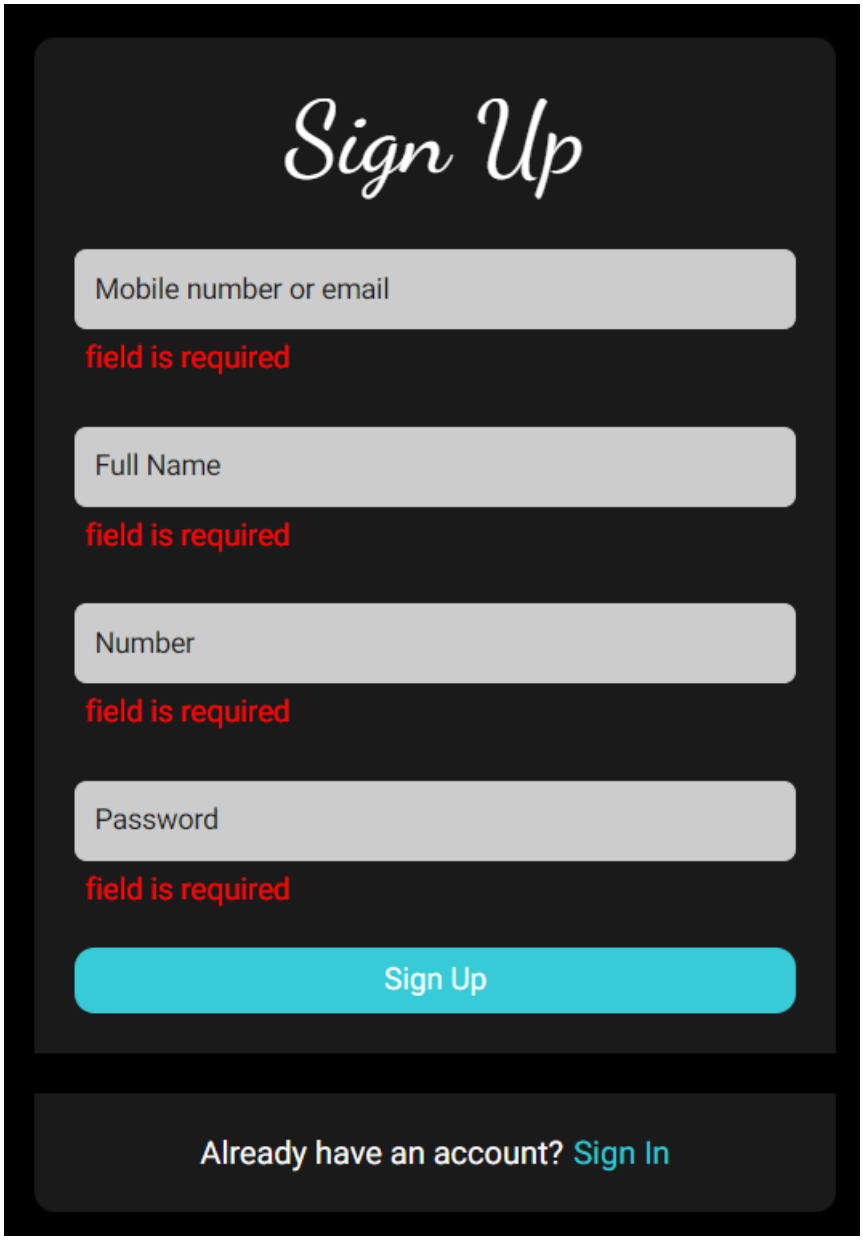

Рис.2.30. Заповнення полів реєстрації

Якщо при реєстрації введена пошта, що вже існує, то зверху з’явиться повідомлення, яке сповістить про те, що користувач с даною поштою вже зареєстрований.

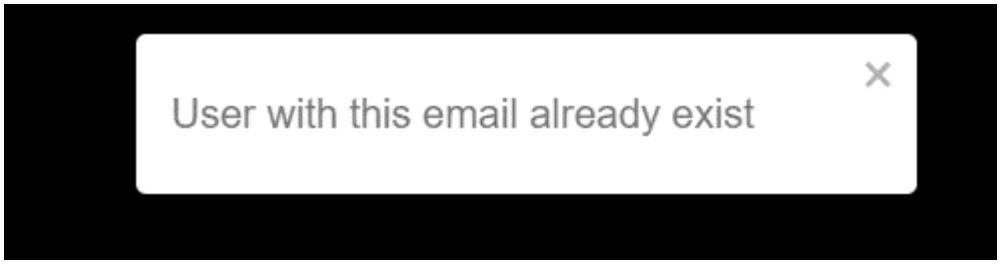

Рис.2.31. Введена пошта вже зареєстрована

Якщо при реєстрації введений номер, що вже існує, то зверху з’явиться повідомлення, яке сповістить про те, що користувач с даним номер вже зареєстрований.

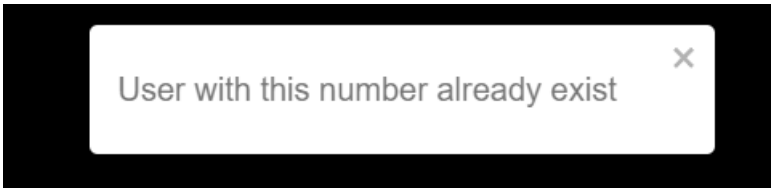

Рис.2.32. Введений номер вже зареєстрований

Після вдалого логіна або реєстрації користувач потрапляє на головну сторінку.

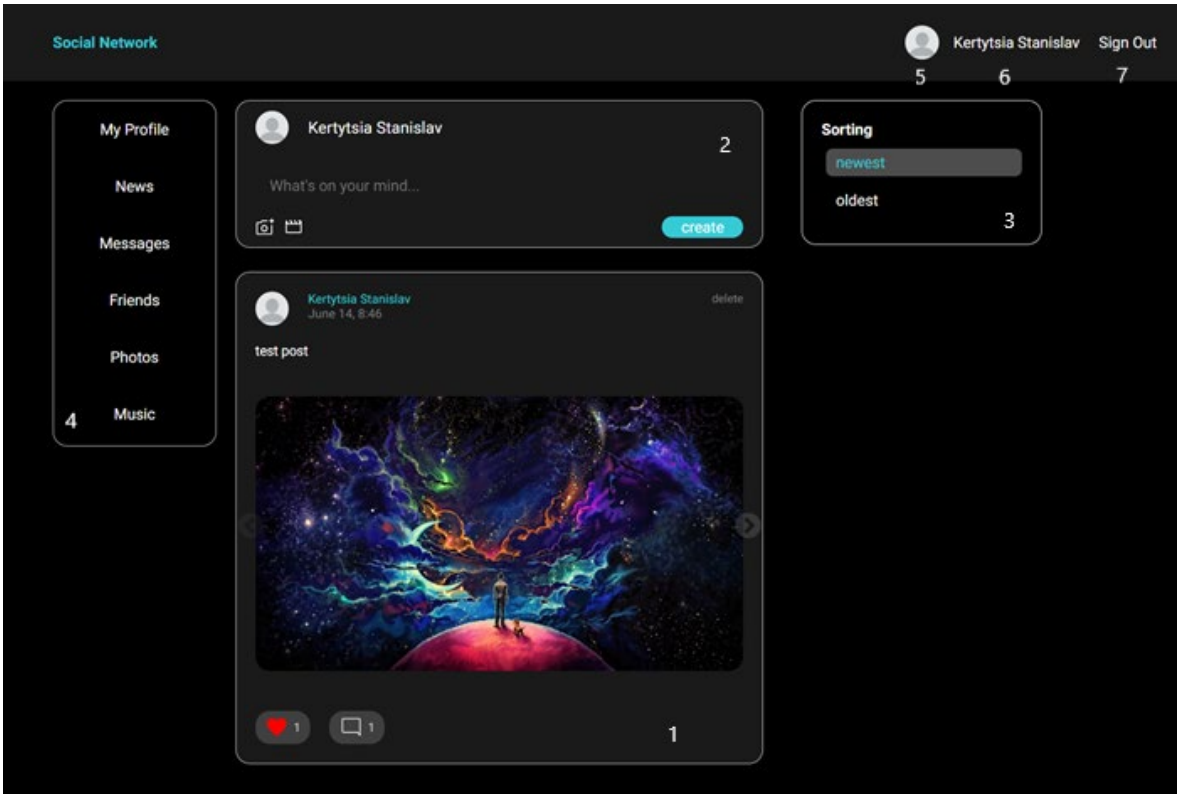

Рис.2.33. Головна сторінка

На головній сторінці користувач може побачити:

1. Пости.
2. Форму для створення постів.
3. Сортировка постів (спочатку старі/нові).
4. Навігаційна панель.
5. Аватар користувача.
6. Ім’я користувача.
7. Кнопка “виходу”.

Форма створювання постів складається з:

1. Додавання картинок.
2. Додавання відео.
3. Аватар користувача.
4. Ім’я користувача.
5. Кнопка “створити пост”.

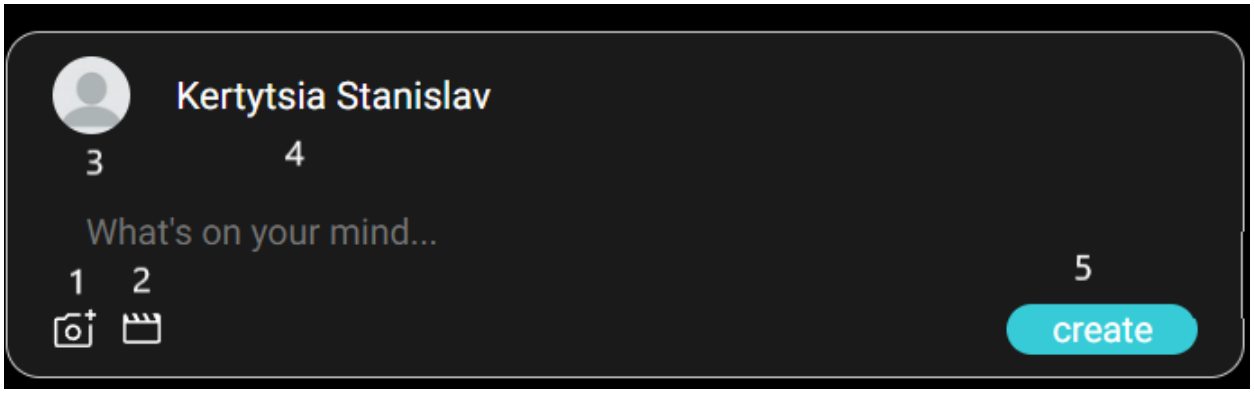

Рис.2.34. Форма створювання постів

Якщо користувач спробує створити пост не додавши тексту або медіа файлів, картинок чи відео, то при натисканні на кнопку Create з’явиться повідомлення, що не можна створити пустий пост.

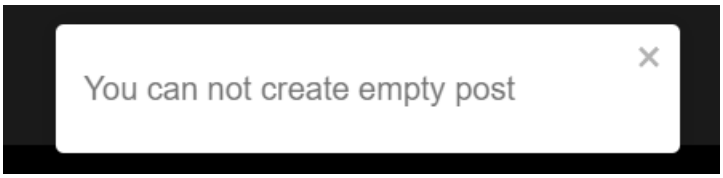

Рис.2.35. Створення пустого поста

Пост складається з:

1. Аватар користувача.
2. Ім’я користувача.
3. Дата створення поста.
4. Текст поста.
5. Кнопки далі/назад, якщо пост має більше 1 медіа файлу.
6. Кнопка лайку.
7. Відкрити коментарі.
8. Якщо користувач володар поста, кнопка видалити

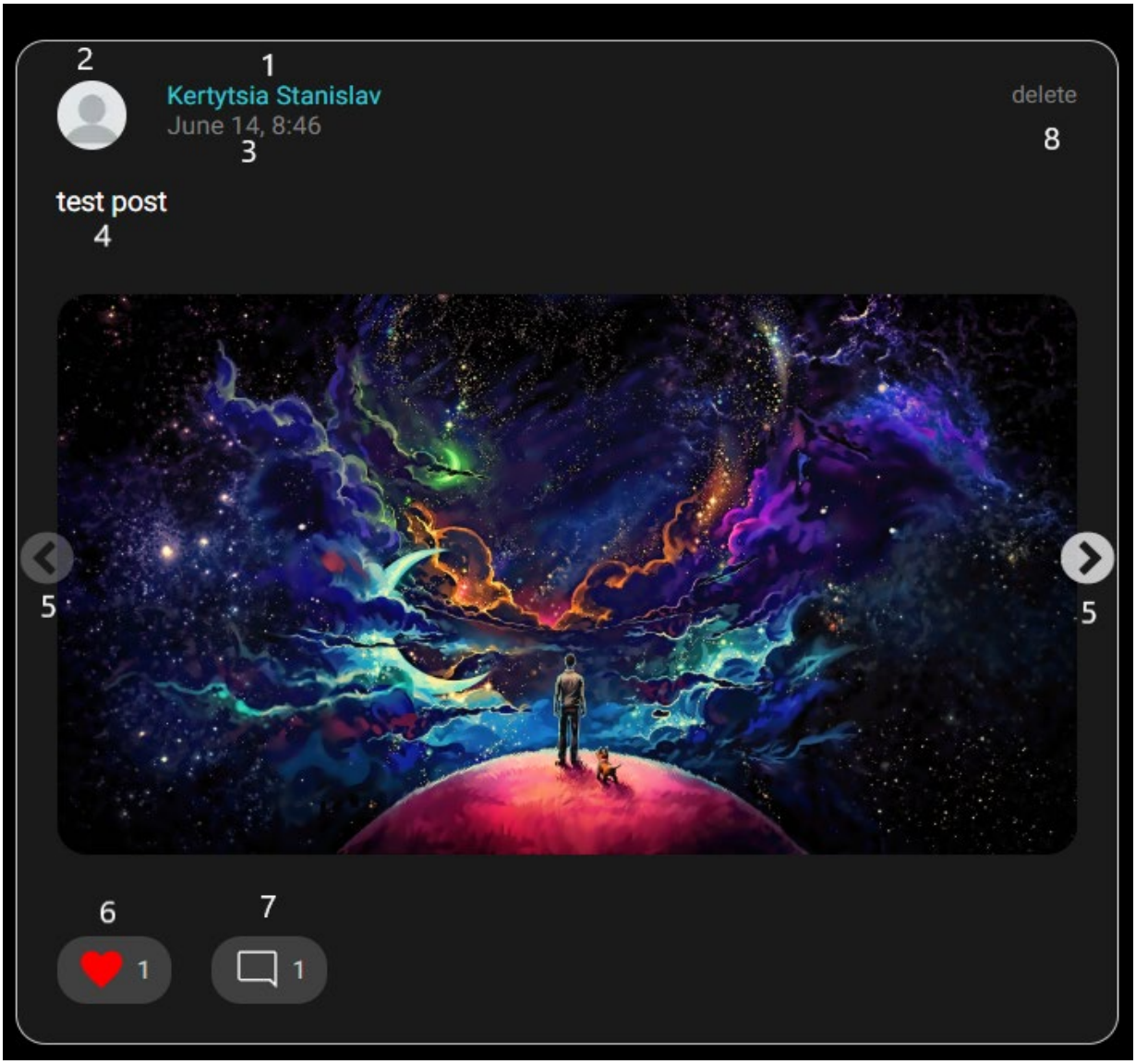

Рис.2.36. Складові поста

Частина коментарів складається з:

1. Аватар користувача.
2. Ім’я користувача.
3. Коментар.
4. Кнопка видалити, якщо ти володар коментаря.
5. Кнопка лайку
6. Форма створення коментарю.
7. Кнопка “закрити коментарі”.

Рис.2.37. Складові коментаря

Сторінка друзів виглядає наступним чином:

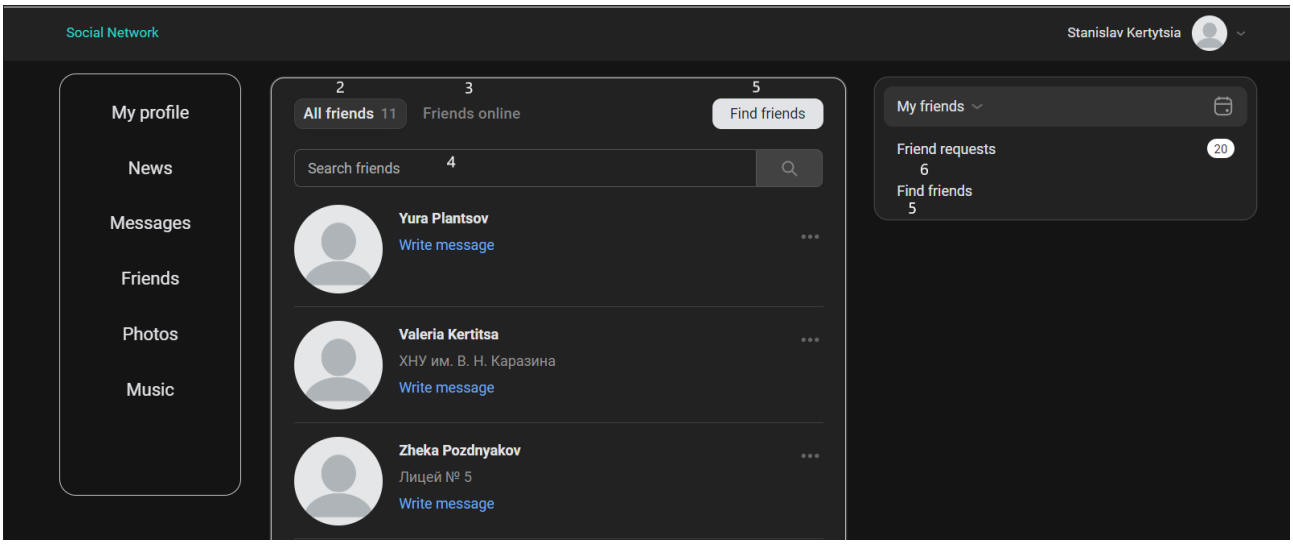

Рис.2.38. Сторінка друзів

На ній користувач побаче:

1. Список друзів
2. Кнопка для показу всіх друзів
3. Кнопка для показу тільки онлайн друзів
4. Форма для пошуку людини серед друзів
5. Кнопка “знайти друзів”
6. Запроси на додавання в друзі від інших людей

Сторінка профілю виглядає наступним чином:

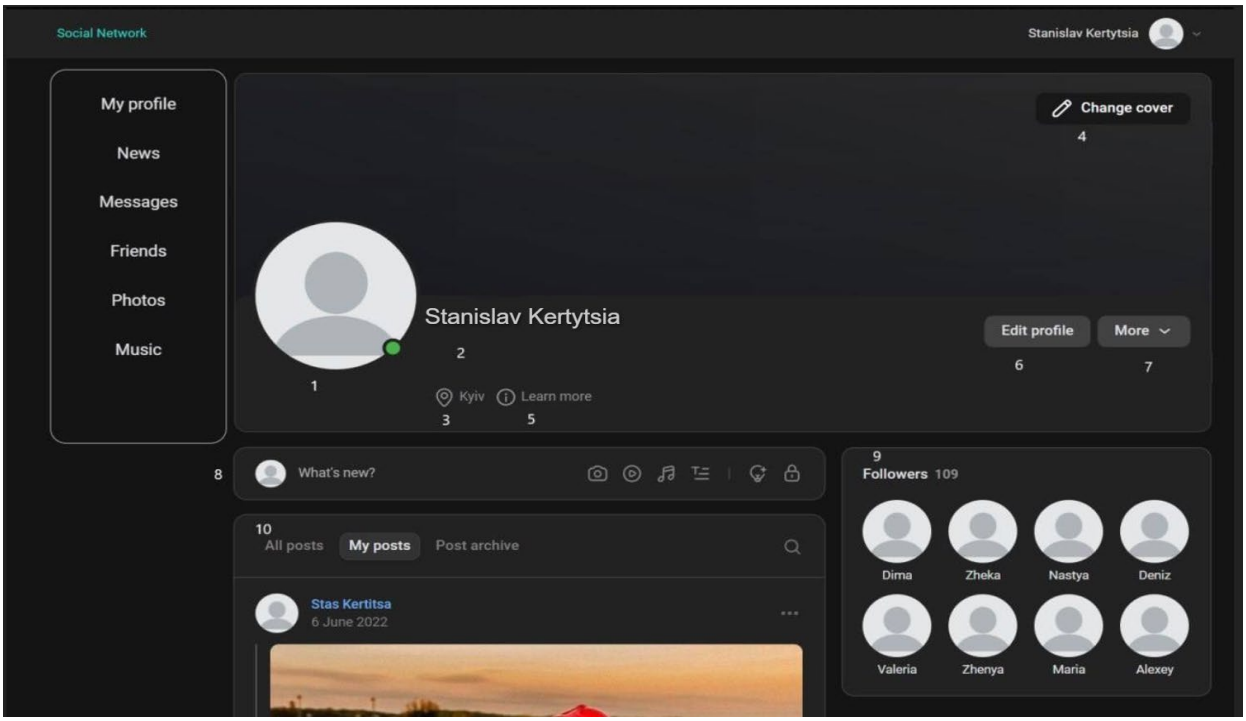

Рис.2.39. Сторінка профілю

На сторінці профілю користувач може побачити:

1. Аватар користувача.
2. Ім’я користувача.
3. Місце знаходження.
4. Кнопка змінити обкладинку.
5. Кнопка «показати додаткову інформацію».
6. Кнопка редагування профілю.
7. Кнопка “додаткові дії”.
8. Форма створення постів.
9. Друзі користувача.
10. Пости користувача.

Сторінка повідомлень виглядає наступним чином:

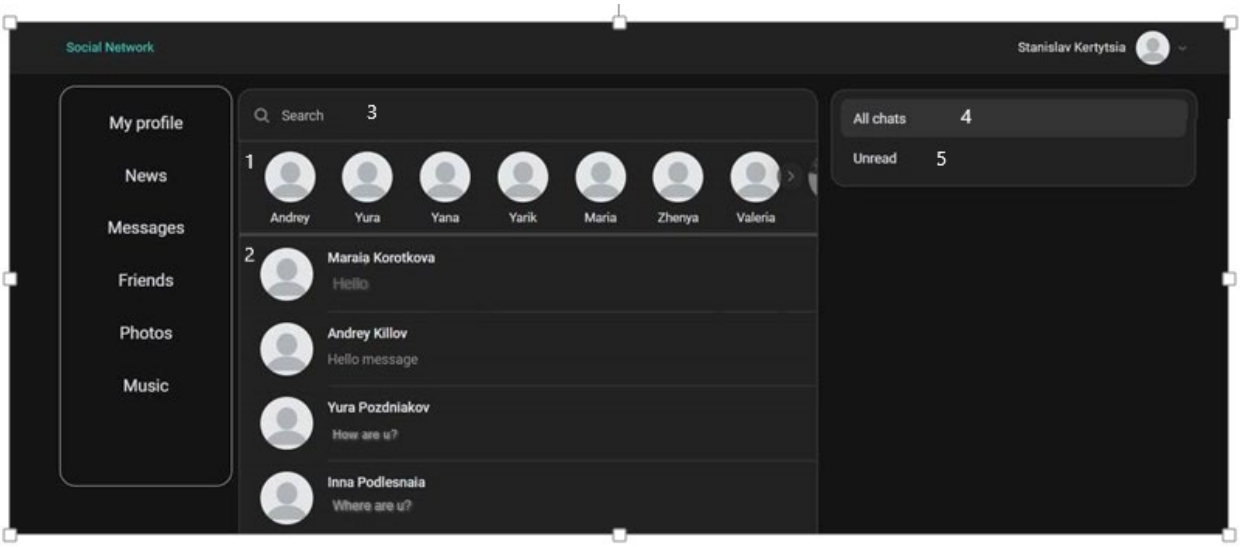

Рис.2.40. Сторінка повідомлень

На сторінці повідомлень користувач може побачити:

1. Список друзів
2. Список повідомлень
3. Форма пошуку друзів
4. Всі повідомлення
5. Тільки непрочитані повідомлення

РОЗДІЛ 3 ЕКОНОМІЧНИЙ РОЗДІЛ

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту

Початкові дані:

1. передбачуване число операторів програми – 2341;
2. коефіцієнт складності програми – 1,6;
3. коефіцієнт корекції програми в ході її розробки – 0,05;
4. годинна заробітна плата програміста – 228 грн/год;

Завдяки сайту (jobs.dou.ua) можна побачити, що програміст Junior Full Stack Web-developer заробляє приблизно 1000$ у місяць праці.

5. коефіцієнт збільшення витрат праці в наслідок недостатнього опису задачі – 1,4;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,2;
7. вартість машино-годин ЕОМ – 30 грн/год.

Трудомісткість розробки програмного забезпечення розраховується за наведеною формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д \text{, людино-годин} \qquad (3.1)$$

to – витрати праці на підготовку й опис поставленої задачі (60 людино-годин);

tu – витрати праці на дослідження алгоритму рішення задачі;

ta – витрати праці на розробку блок-схеми алгоритму;

tn – витрати праці на програмування по готовій блок-схемі;

tотл – витрати праці на налагодження ЕОМ;

tд – витрати праці на підготовку документації.

Умовне число операторів(підпрограм):

$$Q = q \cdot C \cdot (1 + p) \tag{3.2}$$

q – передбачуване число операторів (2341);

C – коефіцієнт складності програми (1,6);

p – коефіцієнт корекції програми в ході її розробки (0,05);

Розрахунок умовного числа операторів:

$$Q = 1{,}6 * 2341 * (1 + 0{,}05) = 3\ 932{,}88 \text{ людино-години},$$

Витрати праці на дослідження алгоритму рішення задачі:

$$t_u = \frac{Q \cdot B}{(75..85) \cdot k} \text{ , людино-годин} \tag{3.3}$$

B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності. При стажі роботи в два роки він складає 1.

З урахуванням коефіцієнта кваліфікації $k = 1{,}3$.

Отримуємо витрати праці на вивчення опису завдання:

$$t_u = (3932 \cdot 1.4) / (75 \cdot 1.3) = 56{,}45 \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі визначаються за формулою:

$$t_a = \frac{Q}{(20 \ldots 25) \cdot k} \text{ людино-годин} \tag{3.4}$$

Q – умовне число операторів програми;

k – коефіцієнт кваліфікації програміста.

$$t_a = 3932 / (20 \cdot 1{,}3) = 151{,}23 \text{ людино-годин.}$$

Витрати на програмування по завершеній блок-схемі:

$$t_n = \frac{Q}{(20 \ldots 25) \cdot k} \text{ , людино-годин} \qquad (3.5)$$

$$t_n = 3932 / (25 \cdot 1{,}3) = 120{,}98 \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ для автономного налагодження завдання:

$$t_{отл} = \frac{Q}{(4..5) \cdot k} \text{ , людино-годин} \qquad (3.6)$$

$$t_{отл} = 3932 / (5 \cdot 1{,}3) = 604{,}32 \text{ людино-годин.}$$

Витрати праці на підготовку документації можна визначити за формулою:

$$t_д = t_{др} + t_{до} \text{ , людино-годин} \qquad (3.8)$$

tдр – трудомісткість підготовки матеріалів і рукопису:

$$t_{др} = \frac{Q}{(15..20) \cdot k} \text{ , людино-годин} \qquad (3.9)$$

tдо – трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0{,}75 \cdot t_{др} \text{ людино-годин} \tag{3.10}$$

Підставляючи відповідні значення, отримаємо:

$t_{др}$ =3932 / (18 · 1.3) = 213,69 людино-годин.

$t_{до}$ = 0,75 · 213,69 = 160,27 людино-годин.

$t_{д}$ = 213,69 + 160,27 = 373,27 людино-годин.

Тепер визначимо повну трудомісткість розробки ПЗ:

t = 50 + 56,45 + 151.23*2 + 604,32 + 373,27 = 1386,5 людино-годин.

3.2. Розрахунок витрат на створення програми

Загальні витрати на створення ПЗ складаються з витрат машинного часу програми та витрат на заробітну плату робітнику:

$$K_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн} \tag{3.11}$$

Заробітна плата виконавців дорівнює:

$$З_{ЗП} = t \cdot C_{ПР}, \text{ грн} \tag{3.12}$$

де t – загальна трудомісткість, людино-годин;

Спр – середня годинна заробітна плата програміста, грн/година.

Отримуємо:

$$З_{ЗП} = 1386,5 \cdot 228 = 316\ 122 \text{ грн.}$$

Вартість машинного часу для налагодження програми на електронно обчислювальній машині:

$$З_{мв} = t_{отл} \cdot C_{мч}, \text{ грн,} \tag{3.13}$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год;

$C_{мч}$ - вартість машино-години ЕОМ, грн/год (12 грн/год).

Підставивши в формулу (3.14) відповідні значення, визначимо вартість необхідного для налагодження машинного часу:

$$З_{мв} = 906,28 \cdot 30 = 27\ 194,4 \text{ грн.}$$

Загальні витрати на створення програмного продукту будуть мати значення:

$$К_{ПО} = 316\ 122 + 27\ 194,4 = 343\ 316,4 \text{ грн.}$$

Очікуваний період створення програмного забезпечення розраховується за формулою:

$$T = \frac{t}{B_k \cdot F_p}, \text{ міс.} \tag{3.14}$$

Bk – число виконавців;

Fp – місячний фонд робочого часу (при 40 год. робочому тижні).

Витрати на створення програмного продукту:

$$T = 1386{,}5 / (1 \cdot 176) \approx 7{,}8 \text{ міс.}$$

Висновок: в процесі обчислення економічної частини кваліфікаційної роботи було отримано, що для розробки системи необхідно 7,8 місяців роботи одного виконавця. Вартість проекту складає 343 316 грн.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи на тему "Розробка соціальної мережі для оргранізації взаємодії між відвідувачами з використанням Nextjs, Nestjs та бази даних PostgreSQL" було проведено дослідницький та практичний аналіз різних аспектів створення та розвитку соціальних мереж. Метою роботи було вивчити процес розробки соціальної мережі та оцінити її потенціал для створення середовища, що сприяє комунікації та взаємодії користувачів.

В ході розробки додатку були використані наступні технології: Nextjs, Sass, Nestjs, TypeScript, PostreSQL.

В результаті дослідження було з'ясовано, що розробка соціальної мережі є складним та багатогранним процесом, що потребує глибокого розуміння потреб користувачів, аналізу конкурентного середовища та обліку останніх технологічних трендів. Продумана архітектура, інтуїтивний інтерфейс, ефективні механізми взаємодії та захист даних є ключовими складовими успішної соціальної мережі.

Однак, важливо зазначити, що розробка соціальної мережі – це лише початковий етап. Після запуску мережі необхідно продовжувати її розвиток, враховуючи зворотний зв'язок користувачів, впроваджуючи нові функції та покращуючи існуючі. Також слід звернути увагу на питання безпеки та захисту даних користувачів, щоб забезпечити їхню довіру та зберегти репутацію соціальної мережі.

На закінчення, розробка соціальної мережі є складним і динамічним процесом, який вимагає глибокого розуміння потреб користувача, використання передових технологій і постійного оновлення функціоналу. Успішна соціальна мережа може стати засобом комунікації, обміну інформацією та соціальною активністю, сприяючи розвитку співтовариств та підвищенню рівня взаємодії.

Додатково було визначено трудомісткість розробленої інформаційної системи (1386 людино-годин), проведений підрахунок вартості роботи по створенню програми (343 316 грн) та розраховано час на його створення (7.8 міс).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційна документація React.JS URL [Електронний ресурс] - Режим доступу до ресурсу: https://ru.reactjs.org/docs/gettingstarted.html (дата звернення: 20.05.2023).
2. About JavaScript URL [Електронний ресурс] - Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript (дата звернення: 20.05.2023).
3. Офіційний документація середовища розробки Visual Studio Code URL [Електронний ресурс] - Режим доступу до ресурсу: https://code.visualstudio.com (дата звернення: 27.03.2023).
4. HTML & CSS – W3C [Електронний ресурс] // Режим доступу до ресурсу: http://www.w3.org/standards/webdesign/htmlcss (дата звернення: 20.05.2023).
5. The State of JavaScript [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://2019.stateofjs.com/testing/. (дата звернення: 24.05.2023).
6. Node.js - Market Share & Web Usage Statistics [Електронний ресурс] //SimilarTech – Режим доступу до ресурсу: https://www.similartech.com/technologies/nodejs (дата звернення: 25.05.2023).
7. ECMAScript 2021 Language Specification [Електронний ресурс] – Режим доступу до ресурсу: https://tc39.es/ecma262/ (дата звернення: 25.05.2023).
8. Web-программист: средняя зарплата в Украине [Електронний ресурс]– Режим доступу до ресурсу: www.work.ua/ru/salary-web-программист/ (дата звернення: 27.03.2023).
9. Для чого потрібен React [Електронний ресурс] – Режим доступу до ресурсу: https://blog.skillfactory.ru/glossary/react/ (дата звернення: 25.05.2023).
10. Базовий захист даних на JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://academy.mediasoft.team/article/bazovaya-zashita-

dannykh-na-javascript/ (дата звернення: 26.05.2023).

11. Реляційна база даних [Електронний ресурс] – Режим доступу до ресурсу: https://ua5.org/database/189-reljacjjna-baza-danikh.html (дата звернення: 26.05.2023).

12. SQL-ін'єкції [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/articles/725134/ (дата звернення: 27.05.2023).

13. Основні поняття інформаційної безпеки [Електронний ресурс] – Режим доступу до ресурсу: https://www.miyklas.com.ua/p/informatica/9-klas/programne-zabezpechennia-ta-informatciina-bezpeka-327110/informatciina-bezpeka-327251/re-813d7f93-b124-47bb-92c6-261cf0cd2162 (дата звернення: 27.05.2023).

14. Основні складові інформаційної безпеки [Електронний ресурс] – Режим доступу до ресурсу: https://sites.google.com/site/informbezpekaosobu/osnovni-skladovi-informacijnoie-bezpeki (дата звернення: 27.05.2023).

15. Що таке RestAPI [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.itstep.org/blog_3/rest-api-what-is-it-restful-project-on-python-flask (дата звернення: 27.05.2023).

16. Global Time Tracking Software Market [Електронний ресурс] – Режим доступу до ресурсу: https://www.databridgemarketresearch.com/reports/global-time-tracking-software-market (дата звернення: 30.05.2023).

17. CodeTools & Tips, HTTP: Протокол, який повинен розуміти кожний веб-розробник (Частина 1) [Електронний ресурс] / Паван Поділа, 8 квітня 2013р. — Режим доступу до ресурсу: https://code.tutsplus.com/uk/tutorials/http-the-protocol-every-web-developer-mustknow-part-1--net-31177. (дата звернення: 30.05.2023).

18. What Is a Web Application? How It Works, Benefits and Examples [Електронний ресурс] / Indeed Editorial Team, November 10, 2021 — Режим

доступу до ресурсу:
https://www.indeed.com/career-advice/careerdevelopment/what-is-web-application. (дата звернення: 27.03.2023).
19. Material UI — бібліотека компонентів інтерфейсу React, реалізує Material Design від Google [Електронний ресурс] / Офіційне джерело розробників з поточною версією: 5.8.3 — Режим доступу до ресурсу: https://mui.com. (дата звернення: 27.03.2023).
20. Microsoft SQL Server 2019 Express, SQL, Database [Електронний ресурс] — Режим доступу до ресурсу: https://www.microsoft.com/enus/Download/details.aspx?id=101064 (дата звернення: 2.06.2023).
21. DOU / jobs, salaries, Junior Software Engineer, C#/.NET, JavaScript [Електронний ресурс] — Режим доступу до ресурсу: https://jobs.dou.ua/salaries/?period=2021-12&position=Middle%20SE (дата звернення: 2.06.2023).
22. МІНІСТЕРСТВО ЕНЕРГЕТИКИ УКРАЇНИ [Електронний ресурс] — Режим доступу до ресурсу: http://mpe.kmu.gov.ua/minugol/control/publish/article?art_id=245583544 (дата звернення: 2.06.2023).

ДОДАТОК А

КОД ПРОГРАМИ

Серверна частина.

```
@Controller('auth')
export class AuthController {
  constructor(private auth: AuthService) {}

  //signin
  @Post('signin')
  async login(@Body() dto: signinDto, @Res({ passthrough: true }) res) {
    const data = await this.auth.signin(dto);
    this.returnCookies(res, data.refreshToken);
    return data;
  }
  //signup
  @Post('signup')
  async signup(@Body() dto: signupDto, @Res({ passthrough: true }) res) {
    const data = await this.auth.signup(dto);

    this.returnCookies(res, data.refreshToken);
    return data;
  }
  //refresh
  @Get('refresh')
  async refreshTokens(@Res({ passthrough: true }) res, @Req() req) {
    const data = await this.auth.refreshTokens(req);
    this.returnCookies(res, data.refreshToken);
    return data;
  }
  //signout
  @Patch('signout')
  @Auth()
  async logout(@CurrentUser('id') userId, @Res({ passthrough: true }) res) {
    const data = await this.auth.signout(userId);
    res.cookie('refreshToken', ' ');
    return data;
  }
  returnCookies(res, token) {
    res.cookie('refreshToken', token, {
      maxAge: 1000 * 60 * 60 * 24 * 30,
      httpOnly: true,
    });
  }
}

@Injectable()
export class AuthService {
  constructor(
    private prisma: PrismaService,
    private user: UserService,
```

```
    private jwt: JwtService,
    private config: ConfigService,
  ) {}

  async signup(dto: signupDto) {
    const { email, fullname, password, number } = dto;

    const isEmailExist = await this.user.findUserByEmail(email);
    if (isEmailExist)
      throw new BadRequestException('User with this email already exist');
    const isNumberExist = await this.user.findUserByNumber(number);
    if (isNumberExist)
      throw new BadRequestException('User with this number already exist');

    const hashedPassword = await hash(password, 10);
    const newUser = await this.prisma.user.create({
      data: {
        email,
        number,
        fullname,
        password: hashedPassword,
      },
    });

    const { accessToken, refreshToken } = await this.generateTokens(newUser.id);
    const updatedUser = await this.prisma.user.update({
      where: { id: newUser.id },
      data: { token: refreshToken },
    });

    return {
      user: this.user.returnUser(updatedUser),
      accessToken,
      refreshToken,
    };
  }

  async signin(dto: signinDto) {
    const { email, password } = dto;

    const user = await this.user.findUserByEmail(email);
    if (!user) throw new BadRequestException('Wrong credentials');

    const isMatchPasswords = await compare(password, user.password);
    if (!isMatchPasswords) throw new BadRequestException('Wrong credentials');

    const { accessToken, refreshToken } = await this.updateTokens(user.id);
    return {
      user: this.user.returnUser(user),
      accessToken,
      refreshToken,
    };
```

```
}

async refreshTokens(req: Request) {
  const token = req.headers.cookie.replace('refreshToken=' || 'fake', '');
  if (!token) throw new UnauthorizedException('invalid refresh token');

  const result = await this.jwt.verifyAsync(token, {
    secret: this.config.get('jwt_secret'),
  });
  if (!result) throw new UnauthorizedException('invalid refresh token');

  const user = await this.prisma.user.findUnique({
    where: { id: result.id },
  });

  const { accessToken, refreshToken } = await this.updateTokens(user.id);

  return {
    user: this.user.returnUser(user),
    accessToken,
    refreshToken,
  };
}

async signout(userId: string) {
  return this.prisma.user.update({
    where: { id: userId },
    data: {
      token: ' ',
    },
  });
}

async generateTokens(userId: string) {
  const data = { id: userId };
  const accessToken = await this.jwt.signAsync(data, {
    expiresIn: this.config.get('jwt_access_expire'),
    secret: this.config.get('jwt_secret'),
  });
  const refreshToken = await this.jwt.signAsync(data, {
    expiresIn: this.config.get('jwt_refresh_expire'),
    secret: this.config.get('jwt_secret'),
  });

  return { accessToken, refreshToken };
}

async updateTokens(userId: string) {
  const { refreshToken, accessToken } = await this.generateTokens(userId);
  await this.prisma.user.update({
    where: {
      id: userId,
```

```
      },
      data: {
        token: refreshToken,
      },
    });
    return { refreshToken, accessToken };
  }
}

@Controller('comment')
export class CommentController {
  constructor(private comment: CommentService) {}

  //create
  @Auth()
  @Post(':postId')
  create(@Body() dto, @CurrentUser('id') userId, @Param('postId') postId) {
    return this.comment.create(dto, userId, postId);
  }
  //get all
  @Auth()
  @Get(':postId')
  getAll(@Param('postId') postId) {
    return this.comment.getAll(postId);
  }
  //delete
  @Auth()
  @Delete(':commentId')
  delete(@Param('commentId') commentId) {
    return this.comment.delete(commentId);
  }
  //edit
  @Auth()
  @Patch(':commentId')
  update(@Body() dto, @Param('commentId') commentId) {
    return this.comment.update(dto, commentId);
  }
}
import { Injectable } from '@nestjs/common';
import { Comment, Like, User } from '@prisma/client';

@Injectable()
export class CommentService {
  constructor(
    private prisma: PrismaService,
    private user: UserService,
    private like: LikeService,
  ) {}

  async create(dto: commentDto, userId: string, postId: string) {
    const { text } = dto;
```

```
  const createdComment = await this.prisma.comment.create({
    data: {
      text,
      postId: postId,
      userId: userId,
    },
    include: { likes: true, user: true },
  });

  return this.returnComment(
    createdComment,
    createdComment.user,
    createdComment.likes,
  );
}
async getAll(postId: string) {
  const foundComments = await this.prisma.comment.findMany({
    where: { postId: postId },
    include: { likes: true, user: true },
  });
  return foundComments.map((i) => this.returnComment(i, i.user, i.likes));
}

async delete(commentId: string) {
  const deletedComment = await this.prisma.comment.delete({
    where: { id: commentId },
    include: { likes: true, user: true },
  });
  return this.returnComment(
    deletedComment,
    deletedComment.user,
    deletedComment.likes,
  );
}
async update(dto: commentDto, commentId: string) {
  const { text } = dto;
  const editedComment = await this.prisma.comment.update({
    where: { id: commentId },
    data: {
      text,
    },
    include: { likes: true, user: true },
  });
  return this.returnComment(
    editedComment,
    editedComment.user,
    editedComment.likes,
  );
}

returnComment(comment: Comment, user: User, likes: Like[]) {
  return {
```

```
            id: comment.id,
            createdAt: comment.createdAt,
            text: comment.text,
            likes: likes.map((i) => this.like.returnLikes(i, user)),
            user: this.user.returnUser(user),
          };
        }
}
      import {
        Controller,
        Get,
        Param,
        Post,
        Res,
        UploadedFiles,
        UseInterceptors,
      } from '@nestjs/common';

      @Controller('files')
      export class FilesController {
        constructor(private readonly files: FilesService) {}

        @Auth()
        @Post()
        @UseInterceptors(
          FilesInterceptor('files', 6, {
            storage: diskStorage({
              destination: './uploads',
              filename: editFileName,
            }),
          }),
        )
        async uploadMultipleFiles(@UploadedFiles() files) {
          return this.files.upload(files);
        }

        @Get(':name')
        sendUploadedFiles(@Param('name') name, @Res() res) {
          res.sendFile(name, { root: './uploads' });
        }
}
      import { Injectable } from '@nestjs/common';

      @Injectable()
      export class FilesService {
        upload(files) {
          const response = [];
          files.forEach((file) => {
            const fileResponse = {
              originalname: file.originalname,
              filename: file.filename,
            };
```

```
      response.push(fileResponse);
    });
    return response.map((item) => item.filename);
  }
}

@Controller('like')
export class LikeController {
  constructor(private like: LikeService) {}

  //toggle
  @Auth()
  @Post('post/:itemId')
  toggleLikePost(@Param('itemId') itemId, @CurrentUser('id') userId) {
    return this.like.toggleLikePost(itemId, userId);
  }

  @Auth()
  @Post('comment/:itemId')
  toggleLikeComment(@Param('itemId') itemId, @CurrentUser('id') userId) {
    return this.like.toggleLikeComment(itemId, userId);
  }
  //get all
  @Auth()
  @Get()
  get() {
    return this.like.get();
  }
}



@Injectable()
export class LikeService {
  constructor(private prisma: PrismaService, private user: UserService) {}

  async toggleLikePost(itemId: string, userId: string) {
    const like = await this.prisma.like.findFirst({
      where: { userId: userId, postId: itemId },
    });
    if (like) {
      const deletedLike = await this.prisma.like.delete({
        where: { id: like.id },
        include: { user: true },
      });
      return this.returnLikes(deletedLike, deletedLike.user);
    }
    const createdLike = await this.prisma.like.create({
      data: {
        postId: itemId,
        userId: userId,
      },
```

```
      include: { user: true },
    });
    return this.returnLikes(createdLike, createdLike.user);
  }
  async toggleLikeComment(itemId: string, userId: string) {
    const like = await this.prisma.like.findFirst({
      where: { userId: userId, commentId: itemId },
    });
    if (like) {
      const deletedLike = await this.prisma.like.delete({
        where: { id: like.id },
        include: { user: true },
      });
      return this.returnLikes(deletedLike, deletedLike.user);
    }
    const createdLike = await this.prisma.like.create({
      data: {
        commentId: itemId,
        userId: userId,
      },
      include: { user: true },
    });
    return this.returnLikes(createdLike, createdLike.user);
  }
  async get() {
    const foundLikes = await this.prisma.like.findMany({
      include: { user: true },
    });
    return foundLikes.map((i) => this.returnLikes(i, i.user));
  }

  returnLikes(like: Like, user: User) {
    return {
      id: like.id,
      user: this.user.returnUser(user),
    };
  }
}


@Controller('post')
export class PostController {
  constructor(private post: PostService) {}

  //create
  @Auth()
  @Post()
  create(@Body() dto, @CurrentUser('id') userId) {
    return this.post.create(dto, userId);
  }
  //get all
  @Auth()
```

```
  @Get()
  getAll() {
    return this.post.getAll();
  }
  //get one
  @Auth()
  @Get('/one/:postId')
  getOne(@Param('postId') postId) {
    return this.post.getOne(postId);
  }

  //get user posts
  @Auth()
  @Get('user')
  getUserPosts(@CurrentUser('id') userId) {
    return this.post.getUserPosts(userId);
  }
  //delete
  @Auth()
  @Delete(':postId')
  delete(@Param('postId') postId) {
    return this.post.delete(postId);
  }
  //update
}

@Injectable()
export class PostService {
  constructor(
    private prisma: PrismaService,
    private user: UserService,
    private comment: CommentService,
    private like: LikeService,
  ) {}

  async create(dto: postDto, userId: string) {
    const createdPost = await this.prisma.post.create({
      data: {
        text: dto.text,
        media: dto.media,
        userId: userId,
      },
      include: { user: true, comments: true, likes: true },
    });
    return this.returnPost(
      createdPost,
      createdPost.user,
      createdPost.comments,
      createdPost.likes,
    );
  }
```

```
async getAll() {
  const foundPosts = await this.prisma.post.findMany({
    include: { user: true, comments: true, likes: true },
  });
  return foundPosts.map((i) =>
    this.returnPost(i, i.user, i.comments, i.likes),
  );
}

async getOne(postId: string) {
  const foundPost = await this.prisma.post.findUnique({
    where: { id: postId },
    include: { user: true, comments: true, likes: true },
  });
  return this.returnPost(
    foundPost,
    foundPost.user,
    foundPost.comments,
    foundPost.likes,
  );
}

async getUserPosts(userId: string) {
  const foundPosts = await this.prisma.post.findMany({
    where: { userId: userId },
    include: { user: true, comments: true, likes: true },
  });
  return foundPosts.map((i) =>
    this.returnPost(i, i.user, i.comments, i.likes),
  );
}

async delete(postId: string) {
  const deletedPost = await this.prisma.post.delete({
    where: { id: postId },
    include: { user: true, comments: true, likes: true },
  });

  return this.returnPost(
    deletedPost,
    deletedPost.user,
    deletedPost.comments,
    deletedPost.likes,
  );
}

returnPost(post: Post, user: User, comments: Comment[], likes: Like[]) {
  return {
    id: post.id,
    createdAt: post.createdAt,
    text: post.text,
    media: post.media,
```

```
      comments,
      likes: likes.map((i) => this.like.returnLikes(i, user)),
      user: this.user.returnUser(user),
    };
  }
}

@Controller('user')
export class UserController {
  constructor(private user: UserService) {}

  @Get('profile')
  @Auth()
  getUser(@CurrentUser('id') userId) {
    return this.user.getUser(userId);
  }

  @Get()
  @Auth()
  getUsers() {
    return this.getUsers();
  }
}

@Injectable()
export class UserService {
  constructor(private prisma: PrismaService) {}

  async getUser(userId: string) {
    const user = await this.prisma.user.findUnique({
      where: { id: userId },
    });
    return this.returnUser(user);
  }

  async getUsers() {
    const users = await this.prisma.user.findMany();
    return users;
  }

  returnUser(user: User) {
    return {
      id: user.id,
      email: user.email,
      fullname: user.fullname,
      number: user.number,
      avatar: user.avatar,
    };
  }

  findUserByEmail(email: string) {
    return this.prisma.user.findUnique({ where: { email: email } });
```

```
  }
  findUserByNumber(number: string) {
   return this.prisma.user.findUnique({ where: { number: number } });
  }
}
```

Кліентська частина.

```
import { AuthResponse } from "@/services/interfaces";
import axios from "axios";

const NEXT_PUBLIC_BASE_URL = process.env.NEXT_PUBLIC_BASE_URL;
const NEXT_PUBLIC_REFRESH_URL = "http://localhost:5000/auth/refresh";

export const instance = axios.create({
  withCredentials: true,
  baseURL: NEXT_PUBLIC_BASE_URL,
});
instance.interceptors.request.use((config) => {
  config.headers.Authorization = `Bearer ${localStorage.getItem("token")}`;

  return config;
});

instance.interceptors.response.use(
  (config) => config,
  async (error) => {
    const originalRequest = error.config;

    if (error.response.status == 401) {
      originalRequest._isRetry = true;

      const response = await
axios.get<AuthResponse>(NEXT_PUBLIC_REFRESH_URL, {
        withCredentials: true,
      });

      localStorage.setItem("token", response.data.accessToken);
      return instance.request(originalRequest);
    }
  }
);
export default axios;

type Inputs = {
  email: string;
  password: string;
};
const dancingFont = Dancing_Script({
  weight: ["400"],
  subsets: ["latin"],
});
export default function SigninPage() {
  const {
```

```
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<Inputs>();

  const signin = useSignin();
  const onSubmit: SubmitHandler<Inputs> = (data) => {
    signin.mutate(data);
  };
  return (
    <div className={s.container}>
      <form onSubmit={handleSubmit(onSubmit)} className={` ${s.form}`}>
        <div className={` ${dancingFont.className} ${s.title}`}>Sign In</div>
        <Input
          labelText={"Mobile number or email"}
          {...register("email", { required: true })}
          type={"text"}
          error={errors.email}
        />
        <Input
          labelText={"Password"}
          {...register("password", { required: true })}
          type={"password"}
          error={errors.password}
        />
        <div>
          <input type="submit" value="Sign In" className={s.submit} />
        </div>
      </form>

      <div className={s.signup}>
        <p> Don&amp;apos;t have an account ?</p>
        <Link href="/signup">Sign Up</Link>
      </div>

      <ToastContainer
        position={"top-center"}
        autoClose={5000}
        hideProgressBar={true}
        limit={1}
      />
    </div>
  );
}

type Inputs = {
  email: string;
  fullname: string;
  password: string;
  number: string;
};
const dancingFont = Dancing_Script({
```

```
  weight: ["400"],
  subsets: ["latin"],
});

export default function SignupPage() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<Inputs>();

  const signup = useSignup();

  const onSubmit: SubmitHandler<Inputs> = async (data) => {
    signup.mutate({
      email: data.email,
      fullname: data.fullname,
      password: data.password,
      number: data.number,
    });
  };
  return (
    <div className={s.container}>
      <form onSubmit={handleSubmit(onSubmit)} className={` ${s.form}`}>
        <div className={` ${dancingFont.className} ${s.title}`}>Sign Up</div>
        <div>
          <Input
            labelText={"Mobile number or email"}
            {...register("email", { required: true })}
            type={"text"}
            error={errors.email}
          />
          <Input
            labelText={"Full Name"}
            {...register("fullname", { required: true })}
            type={"text"}
            error={errors.fullname}
          />
          <Input
            labelText={"Number"}
            {...register("number", { required: true })}
            type={"number"}
            error={errors.fullname}
          />

          <Input
            labelText={"Password"}
            {...register("password", { required: true })}
            type={"password"}
            error={errors.password}
          />
          {signup.isLoading && <div>...Loading</div>}
```

```
            <div>
              <input type="submit" value="Sign Up" className={s.submit} />
            </div>
          </div>
        </form>
        <div className={s.signup}>
          <p>Already have an account?</p>
          <Link href="/signin">Sign In</Link>
        </div>
        <ToastContainer
          position={"top-center"}
          autoClose={5000}
          hideProgressBar={true}
          limit={1}
        />
      </div>
    );
}
    export default function HomePage() {
      const [sortBy, setSortBy] = useState<string>("newest");
      const sortPostsCallBack = useCallback(
        (posts: Post[]) => sortPostsHandler(posts, sortBy),
        [sortBy]
      );
      const posts = useQuery(["post"], postService.posts, {
        select: sortPostsCallBack,
      });
      return (
        <div className={s.container}>
          <div>
            <PostCreate />
            {posts.isLoading && <div>...Loading</div>}

            {posts.data?.map((post: Post) => (
              <PostItem key={post.id} post={post} />
            ))}
          </div>

          <SortWidget setSortBy={setSortBy} />
        </div>
      );
}
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу бакалавра

на тему:

«Розробка соціальної мережі для оргранізації взаємодії між відвідувачами з використанням Nextjs, Nestjs та бази даних PostgreSQL»

студента групи 122-19-1 Кертиці Станіслава Володимировича

Керівник економічного розділу

доцент каф. ПЕП та ПУ, к.е.н

Л. В. Касьяненко

ДОДАТОК В

ПЕРЕЛІК ФАЙЛІВ НА ДИСКУ

Ім’я файлу	**Опис**
Пояснювальні документи	
Kertytsia_Diploma.docx	Пояснювальна записка до кваліфікаційної роботи. Документ Word.
Kertytsia_Diploma.pdf	Пояснювальна записка до кваліфікаційної роботи в форматі PDF.
Програма	
Kertytsia_Diploma.rar	Архів. Містить файли програми.
Презентація	
Kertytsia_Diploma.pptx	Презентація кваліфікаційної роботи.