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 *Гречкін Микола Олексійович*
(ПІБ)

академічної групи *122-19-2*
(шифр)

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

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

на тему: *Розробка веб-платформи для соціальної журналістики призначеної для розробників прогрманого забезпечення*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи				
розділів:				
спеціальний				
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 *122-19-2* (група) *Гречкін М.О.* (прізвище та ініціали)

тема кваліфікаційної роботи *Розробка веб-платформи для соціальної журналістики призначеної для розробників прогрманого забезпечення*

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 _____ _____
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ _____
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14.01.2023 р.*

Термін подання кваліфікаційної роботи до ЕК: *12.06.2023 р.*

РЕФЕРАТ

Пояснювальна записка: 91 с., 45 рис., 5 дод., 20 джерел.

Об'єкт розробки: розробка веб-платформи для соцільної журналістики призначеної для розробників.

Мета дипломного проекту: розробити платформу за допомогою якої розробники зможуть створювати статі, редагувати їх, пропонувати зміни до чужих статей та залишати до них реакції.

У вступі розглядається аналіз та сучасний стан проблеми, конкретизується мета кваліфікаційної роботи та галузь її застосування, наведено обґрунтування актуальності теми та уточнюється постановка завдання.

У першому розділі проведено аналіз предметної області, визначено актуальність завдання та призначення розробки, розроблена постановка завдання, задані вимоги до програмної реалізації, технологій та програмних засобів.

У другому розділі виконано аналіз існуючих рішень, обрано вибір платформи для розробки, виконано проектування і розробка програми, наведено опис алгоритму і структури функціонування системи, визначені вхідні і вихідні дані, наведені характеристики складу параметрів технічних засобів, описаний виклик та завантаження застосунку, описана робота програми.

В економічному розділі визначено трудомісткість розробленої інформаційної підсистеми, проведений підрахунок вартості роботи по створенню застосунку та розраховано час на його створення.

Актуальність даного програмного забезпечення визначається необхідністю користувачів планувати свою цілі та задачі, а також простежувати їх за допомогою зручного та зрозумілого інтерфейсу.

Список ключових слів: REACT, SPRING BOOT, TYPESCRIPT, CSS, HTML, DATABASE, REDUX, SAGA, POSTGRES, СОЦІАЛЬНА ЖУРНАЛІСТИКА, ВЕБ-ПЛАТФОРМА.

ABSTRACT

Explanatory note: 84 pages, 30 pics, 5 apps, 24 sources.

Object of development: development of a web platform for social journalism intended for developers.

The purpose of the diploma project: develop a platform with the help of which developers will be able to create articles, edit them, propose changes to other people's articles and leave reactions to them.

The introduction considers the analysis and the current state of the problem, specifies the purpose of the qualification work and the field of its application, provides a justification for the relevance of the topic and clarifies the problem.

In the first section the analysis of the subject area is carried out, the urgency of the task and purpose of development is defined, the statement of the task is developed, requirements to software realization, technologies and software are set.

The second section analyzes the existing solutions, selects the platform for development, performs design and development of the program, describes the algorithm and structure of the system, determines the input and output data, provides the characteristics of the parameters of hardware, describes the call and application load, describes the program.

In the economic section, the complexity of the developed information subsystem is determined, the cost of work on creating the application is calculated and the time for its creation is calculated.

The relevance of this software is determined by the need for users to plan their goals and objectives, as well as track them through a user-friendly interface.

List of keywords: REACT, SPRING BOOT, TYPESCRIPT, CSS, HTML, DATABASE, REDUX, SAGA, POSTGRES, SOCIAL JOURNALISM, WEB-PLATFORM.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

TS – мова програмування TypeScript.

API – інтерфейс програмування додатків.

HTML – Hyper Text Markup Language.

CSS – Cascading Styles Sheets.

MVC – Model View Controller.

CRUD – Create Read Update Delete.

JWT – Json Web Token.

HTTP – Hyper Text Transfer Protocol.

API – Application Programming Interface.

DTO – Data Transfer Object.

ORM – Object Relational Mapping.

ВСТУП

Кожен розробник має прагнути до самовдосконалення, бо сфера інформаційних технологій дуже динамічно розвивається, і якщо не займатися самоосвітою, то в один момент дуже швидко можна стати не затребуваним на ринку праці. Одним з основних джерел інформації є технологічні статті, які пояснюють якусь складну концепцію або знайомлять з новим технологіями. Саме тому віртуальні простори для соціальної журналістики, які надають можливість розробникам ділитися досвідом та розвиватися, стали настільки популярними.

У цьому контексті, розробка веб-платформи для написання статей може стати важливим інструментом за допомогою якого досвідчені спеціалісти зможуть ділитися своїм досвідом, а розробники, які знаходиться на старті своєї кар'єри, матимуть місце, де вони зможуть вчитися за допомогою досвіду інших. Ця кваліфікаційна робота ставить перед собою ціль створення такої веб-платформи.

Для реалізації проєкту було проведено аналіз існуючих веб-платформ для обміну інформацією, а також вивчено побажання та потреби розробників програмного забезпечення.

Платформа дозволить розробникам створювати та публікувати статті, редагувати їх, а також залишати реакції на статті інших авторів. Однією з основних особливостей цієї платформи є те, що кожен користувач, якщо його зацікавила якась інформація у статті, зможе зберегти її у вигляді цитати. Після чого він матиме змогу у будь-який час переглядати збережені цитати на спеціальній сторінці. У даній кваліфікаційна роботі описано процес розробки веб-платформи для соціальної журналістики, її функціональні можливості та особливості використання. Крім того, у роботі розглянуто архітектуру розробленої системи, технології, що використовувалися для її розробки, та проаналізовано результати її роботи.

Цей проєкт стане важливим кроком у напрямку розробки веб-платформ для обміну досвідом та інформацією в сфері програмування та стане корисним інструментом для розробників програмного забезпечення.

В рамках дипломної роботи також проведено дослідження ринку та аналіз конкурентних веб-платформ для соціальної журналістики для розробників програмного забезпечення. Це дозволить виявити переваги та недоліки існуючих рішень і використати цю інформацію для покращення розробленої веб-платформи. Також вивчені та використані найкращі практики веб-розробки, а також методи та алгоритми для забезпечення безпеки та стійкості. Результатом дипломної роботи є реалізація веб-платформи, яка функціонує та відповідає всім вимогам та стандартам, які були визначені для неї.

Окрім того, у роботі розглянуті питання проєктування та розробки інтерфейсу користувача з використанням сучасних підходів та дизайн-методологій. Візуальний аспект платформи відіграє важливу роль у залученні та утриманні користувачів, тому особлива увага приділена створенню зручного та естетичного інтерфейсу.

На підставі проведених досліджень та розробки веб-платформи можна зробити висновок, що дана робота виконана відповідно до поставлених цілей та завдань. Вона пропонує новий інструмент для розробників програмного забезпечення, що сприяє їхньому професійному розвитку, обміну досвідом та взаємній підтримці. Результатом дипломної роботи буде функціонуюча веб-платформа, яка забезпечує потреби розробників у доступі до якісного контенту. Крім того, платформа створює сприятливу середу для обміну думками, досвідом та взаємодопомогою між розробниками, що сприяє розвитку спільноти розробників.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості с предметної галузі

Соціальна журналістика для розробників – це підхід до журналістики, спрямований на розповідь і обмін історіями та знаннями зі світу програмування та розробки програмного забезпечення. Вона поєднує елементи журналістики та технічного спеціаліста. Соціальна журналістика для розробників сприяє обміну знаннями, стимулює інновації, сприяє розвитку розробницького співтовариства і допомагає залучати нових талановитих розробників до цієї галузі. Вона створює можливості для взаємодії, співпраці та вирішення спільних проблем, що сприяє поширенню знань і підвищенню рівня компетентності серед розробників.

Однією з основних переваг соціально новинних платформ є можливість швидкого та масштабного поширення інформації серед користувачів, а також можливість отримати зворотний зв'язок від аудиторії.

Розробка соціально новинної платформи для розробників програмного забезпечення є важливим кроком у розвитку технологій, що дозволяють комунікувати та співпрацювати між розробниками, а також ділитися знаннями та досвідом у роботі.

Соціальні мережі та інформаційні портали в сучасному світі є невід'ємною частиною нашого повсякденного життя. Вони надають нам можливість швидко та легко ділитись інформацією з іншими людьми, отримувати новини та розважальний контент, знайомитись з новими людьми та зберігати зв'язки зі старими друзями.

У сфері журналістики соціальні медіа стали важливим інструментом для швидкого та масового поширення новин та інформації. Завдяки соціальним мережам та інформаційним порталам, журналісти можуть отримувати доступ до новин з усього світу, відслідковувати тенденції та реагувати на них вчасно.

Розглянемо деякі популярні додатки, які використовуються для соціальної журналістики зараз.

Medium – платформа для електронних публікацій, створена співрозробником Blogger та співзасновником Twitter Еваном Вільямсом і запущена у серпні 2012 року. Доступна будь-яким користувачам на безкоштовній основі та відзначається можливістю простими засобами оформляти публікації на професійному дизайнерському рівні.

Еван Вільямс пояснював головне покликання Medium, з одного боку вийти за рамки обмежень у кількості знаків, характерних для Твіттер, а з іншого боку запропонувати вирішення проблеми засміченості мережі недостовірною та низькопрофесійною інформацією. Однак, такий підхід не обов'язково означає спрямованість на дуже довгі матеріали; на це натякає сама назва, що перегукується із гаслом платформи «*not to long, not too short, just medium*» («не надто довго, не надто коротко, середньо»). Особливістю Medium є зорієнтованість на поширення публікацій не серед кола друзів, а на ширший загал (рис. 1.1). За словами одного з співзасновників Medium Біза Стоуна, платформа є спробою еволюційного стрибка у видавничій справі, оскільки Medium задає високу планку якості матеріалів (рис. 1.2).

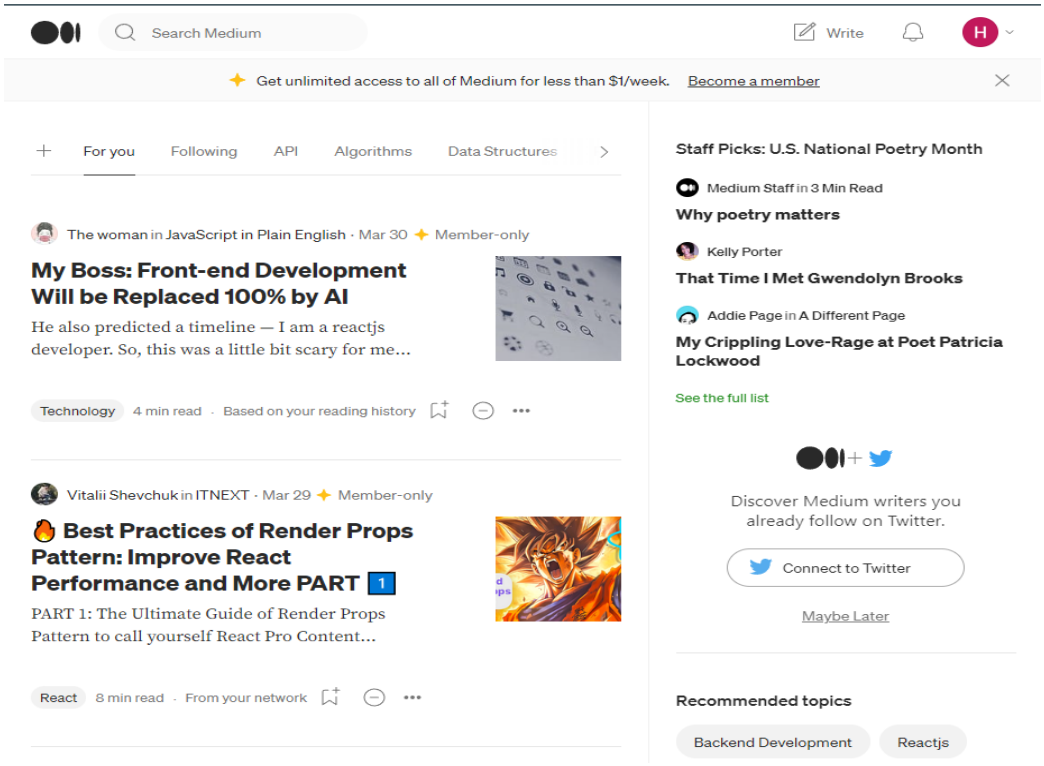


Рис. 1.1 – Головна сторінка порталу Medium

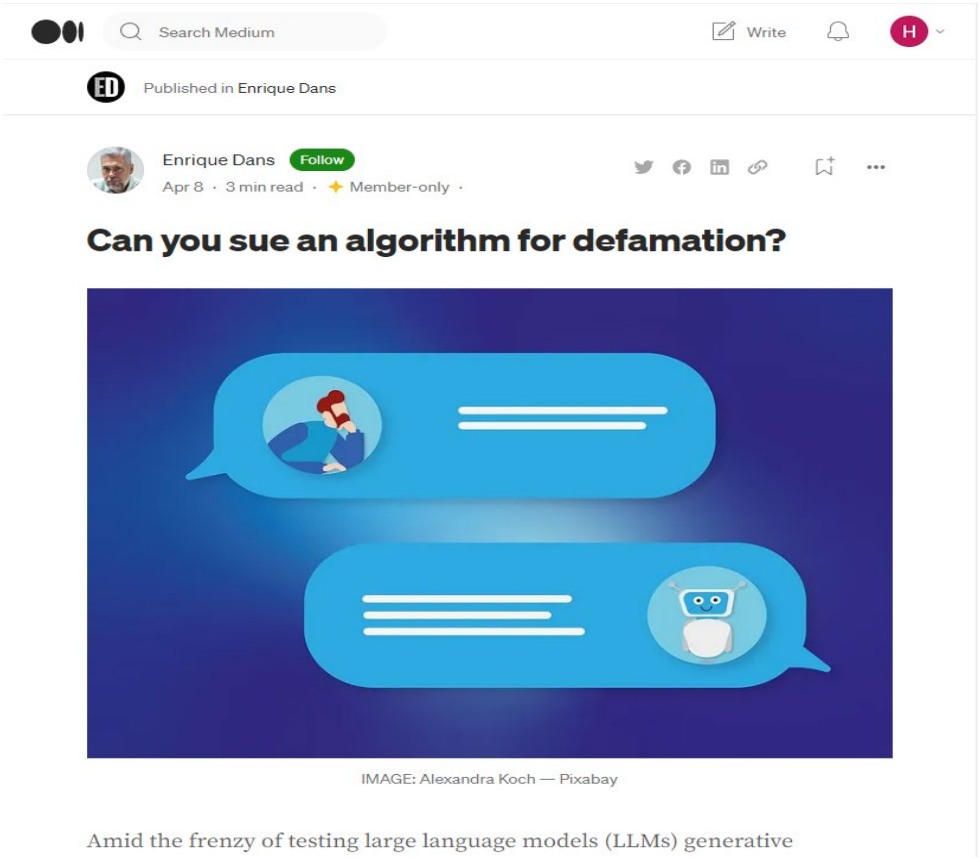


Рис. 1.2 – Стаття на порталі Medium

Основні переваги платформи Medium включають:

– спрощена публікація контенту: Medium пропонує інтуїтивно зрозумілий інтерфейс, який дозволяє додавати текст, зображення та відео до статті. Крім того, Medium має функції форматування, які дозволяють додавати заголовки, списки, цитати тощо;
– велика аудиторія: Medium має велику та зростаючу аудиторію, що дозволяє авторам контенту залучати більше читачів та отримувати більше впливу;
– легкий доступ до контенту: Medium дозволяє знайти та читати статті, навіть якщо ви не маєте акаунту на платформі;
– система підписок: Medium пропонує систему підписок, яка дозволяє читачам отримувати сповіщення про нові статті від обраних авторів;
– можливість заробітку: Medium пропонує авторам можливість заробляти на своєму контенті, що дозволяє їм отримувати винагороду за якісний та цікавий контент;

– легкий пошук контенту: Medium має досить потужну систему пошуку, яка дозволяє знайти контент на будь-яку тему, що вас цікавить;
– редагування та публікація з інших додатків: Medium дозволяє публікувати контент з інших додатків, таких як Google Docs або Microsoft Word, що дозволяє авторам зручно редагувати свій контент [1].

Dev.to – це веб-платформа для спільноти розробників програмного забезпечення, де вони можуть публікувати свої статті, ділитися досвідом, взаємодіяти з колегами та знайомитися з новими ідеями (рис. 1.3). Dev.to був створений в 2016 році та став популярним серед розробників, особливо тих, хто працює з відкритим кодом.

На платформі dev.to зібрано велику кількість тематичних статей про розробку програмного забезпечення, які можна знайти за допомогою пошуку або переглянути на сторінках тематичних тегів. Крім того, користувачі можуть долучатися до спільнот, обговорювати новини та події у своїй галузі, використовувати систему взаємодії, щоб отримувати зворотний зв'язок від інших розробників та взаємодіяти з ними.

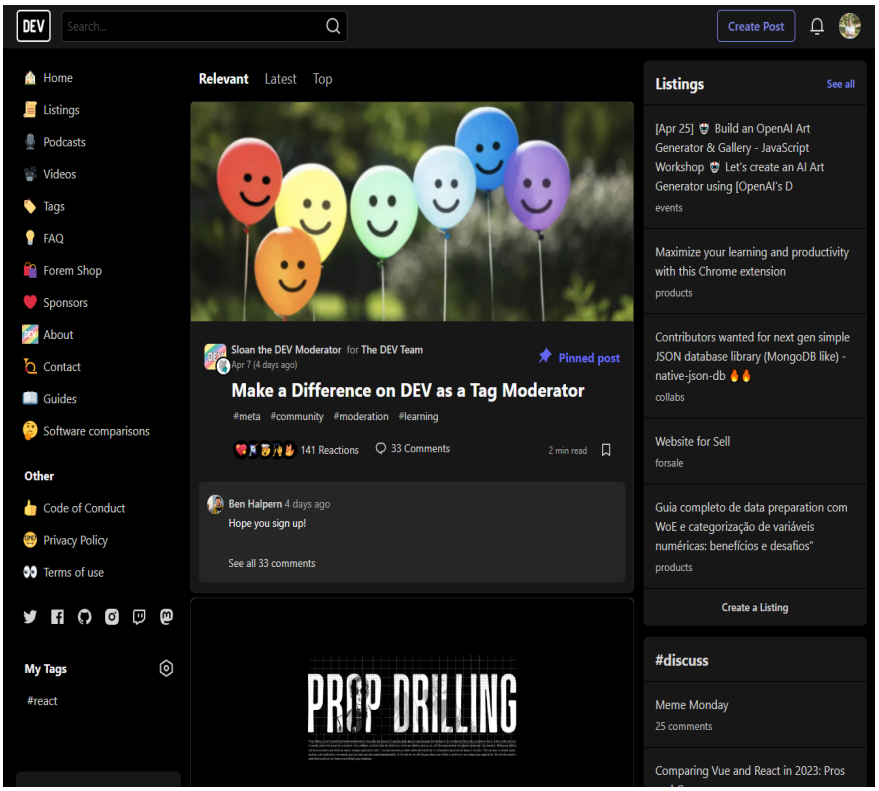


Рис. 1.3 – Головна сторінка платформи Dev.to

Dev.to також пропонує інструменти для ведення блогу, включаючи можливість зберігання чернеток, планування публікацій та надсилання повідомлень про нові статті підписникам. Крім того, на платформі регулярно проводяться віртуальні події та конференції, що дає можливість розробникам з усього світу зустрітися та обговорити теми, що їх цікавлять.

Основні переваги платформи dev.to для розробників та соціальних журналістів:

- спрощений інтерфейс: dev.to пропонує простий та зрозумілий інтерфейс, що дозволяє легко створювати та опубліковувати статті;
- відкритий код: dev.to має відкритий код, що дозволяє користувачам зробити свій внесок у розвиток платформи та створення нових функцій;
- підтримка різних форматів: на платформі dev.to можна публікувати не тільки текстові статті, але й відео, аудіо та інші формати контенту;
- спільнота розробників: dev.to – це активна спільнота розробників, яка взаємодіє між собою, обговорює теми та ділиться досвідом у сфері програмування та розробки ПЗ;
- широка аудиторія: на платформі dev.to зареєстровано більше 2 мільйонів розробників з усього світу, що забезпечує широку аудиторію для публікації своїх статей;
- відсутність реклами: на платформі dev.to відсутня реклама, що забезпечує користувачам чистий і безпечний контент.

1.2. Призначення розробки та галузь застосування

Темою бакалаврської дипломної роботи виступає: «Розробка веб-платформи для соціальної журналістики призначеної для розробників програмного забезпечення». Метою роботи є реалізація веб-платформи за допомогою якої розробники програмного забезпечення зможуть ділитися своїм досвідом через написання статей.

Головними критеріями розроблювального веб-додатку є:

- зручний інтерфейс користувача: додаток повинен мати зручний інтерфейс, який дозволить користувачам легко орієнтуватися в ньому та використовувати всі його функції.
- функціональність: додаток повинен мати достатньо функціональності для того, щоб користувачі могли легко використовувати платформу для своїх потреб.
- безпека: додаток повинен забезпечувати достатній рівень безпеки, щоб захистити користувачів від шкідливих впливів та зловмисників.

1.3. Підстава для розробки

Відповідно до ОКХ та ОПП, згідно навчального плану та графіків навчального процесу, в кінці навчання студент виконує кваліфікаційну роботу (проєкт). Тема роботи узгоджується з керівником проекту, випускаючою кафедрою, та затверджується наказом ректора.

Отже, підставами для розробки (виконання кваліфікаційної роботи) є:

- ОКХ та ОПП за напрямом підготовки 6.050101 «Комп’ютерні науки»;
- Графік навчального процесу та навчальний план;
- наказ ректора Національного технічного університету «Дніпровська політехніка» № 350-с від 16.05.2023 р;
- завдання на дипломний проект на тему «Розробка веб-платформи для соціальної журналістики призначеної для розробників програмного забезпечення».

1.4. Постановка завдання

Веб-додаток повинен реалізувати такі дії як:

- Збереження даних у базу даних з подальшою можливістю доступу до них.
- Можливість використовувати CRUD операції.
- Інтуїтивно зрозумілий інтерфейс додатку.

Для виконанная проєкту необхідно:

- Розробити схему бази даних додатку.
- Спланувати загальну архітектуру проєкту.
- Розробити front-end додатку.
- Розробити back-end додатку.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Розроблене програмне забезпечення, для того, щоб досягнути поставлених цілей, повинно підтримувати виконання таких дій:

- авторизація та аутентифікація користувачів;
- реакції на статті;
- рейтинг найпопулярніших статтей;
- можливість редагування створенної статті;
- можливість додавати статті у «Вибране»;
- функція збереження цитати зі статті;
- використання HTML чи Markdown при написанні тексту статті;

Для підтримки вище перераховані функцій у додатку має бути реалізовано:

- підтримка веб-браузера та доступ до програми через нього;
- програмна та апаратна сумісності;
- стандартна конфігурація яка дає змогу ввести застосунок в експлуатацію.

1.5.2. Вимоги до інформаційної безпеки

Для коректної роботи програми потрібно реалізувати:

- Можливість редагування даних.
- Можливість тривалої роботи протягом 24 годин.
- Контроль та обробка вхідних даних.

- Збереження цілісності даних у випадку збою системи.
- Локальне збереження даних для більшої захищеності.

1.5.3. Вимоги до складу та параметрів технічних засобів

Для забезпечення надійного функціонування програмного забезпечення необхідно, щоб обчислювальна машина, на якій буде експлуатуватися веб-додаток, мала такі характеристики:

- Маніпулятор “миша”.
- Клавіатура.
- Доступ до онлайн мережі.
- 1 Гб вільного місця на жорсткому диску.
- Процесор Intel Core i3 7100 з тактовою частотою 2.3 ГГц.
- Не менше ніж 3Гб оперативної пам’яті.
- Моніток з діагоналлю 14".

Вище наведені характеристики являють собою рекомендовані. Це означає, що при наявності характеристик не нижче зазначених, розроблений додаток буде функціонувати відповідно до вимог щодо надійності, безпеки та швидкості обробки даних.

1.5.4. Вимоги інформаційної та програмної сумісності

Вимоги інформаційної та програмної сумісності для сучасних веб-додатків включають ряд факторів, які впливають на їх ефективну роботу та взаємодію з іншими системами. Ці вимоги спрямовані на забезпечення сумісності та інтеграції додатків з різними платформами, протоколами та форматами даних.

Інформаційна сумісність передбачає, що веб-додаток повинен використовувати стандартні протоколи та формати даних для обміну інформацією з іншими системами. Наприклад, використання протоколу HTTP або HTTPS дозволяє передавати дані між клієнтом та сервером. Крім того, використання

популярних форматів даних, таких як JSON (Javascript Object Notation) або XML (eXtensible Markup Language), забезпечує зрозумілість та легкість обробки даних.

Програмна сумісність передбачає, що веб-додаток повинен бути розроблений з використанням сумісних програмних інтерфейсів (API) та бібліотек. Це дозволяє взаємодіяти з різними сервісами та компонентами системи. Наприклад, використання RESTful API (Representational State Transfer) дозволяє веб-додатку взаємодіяти з іншими додатками за допомогою стандартних HTTP-запитів.

Додатковою вимогою є забезпечення безпеки та конфіденційності даних. Веб-додаток повинен використовувати захисні механізми, такі як шифрування, аутентифікація та авторизація, для запобігання несанкціонованому доступу до даних.

Загалом, інформаційна та програмна сумісність важливі для забезпечення ефективної роботи та взаємодії сучасних веб-додатків з іншими системами та компонентами.

Для коректного функціонування програми необхідно, щоб програмне забезпечення обчислювальної машини, на якій буде експлуатуватися веб-додаток, відповідало наступним вимогам:

- Веб браузер Firefox або Google Chrome.
- Операційна система Windows 7/10.

Веб-орієнтована підсистема має бути реалізована на мові програмування TypeScript з використанням бібліотек React та Redux для рендеру веб-компонентів та зберігання даних. Для візуалізації було використано CSS який описує зовнішній вигляд сторінки.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1. Функціональне призначення системи

Під час виконання дипломної роботи було розроблено програму, а саме, веб-платформу для соціальної журналістики призначеної для розробників програмного забезпечення, головна мета якого надати можилвість розробникам ділитися своєю експертизою одинз одним.

Призначення розробленого додатку:

- Надання можливості створення онлайн середовища для створення статтей.
- Надання можливості редагування статей.
- Надання можливості залишати реакції до статей.
- Збереження всіх дій та даних в сховищі.
- Надання можливості зберігати цитати.

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 7/10, Mac OS, Linux.
- Мати підтримку сучасних браузерів Firefox, Safari, Edge або Google Chrome.

2.2. Опис застосованих математичних методів

Так як особливості предметної області розв'язуваної задачі не передбачають застосування математичних методів, при розробці веб-додатку для візуального планування задач математичні методи не використовувалися.

2.3. Опис використаних технологій та мов програмування

Веб-додаток реалізовано з використанням двох мов програмування. Серверна частина написана на мові Java та фреймворці SpringBoot. Для клієнтської частини використовувався TypeScript та фреймворки React та Redux [4 – 6]. Для стилізації сторінок застосувалась мова опису зовнішнього вигляду документа – CSS.

Опис мови програмування TypeScript

TypeScript - це мова програмування, яка є надмножиною мови JavaScript і використовується для розробки веб-додатків та Node.js застосунків. TypeScript підтримує статичну типізацію, що дозволяє програмістам перевіряти правильність використання типів під час компіляції коду.

TypeScript також має ряд особливостей, які роблять його корисним для розробки великих проектів, таких як інтерфейси, класи, абстрактні класи, модифікатори доступу, дженеріки та інші. Він забезпечує зручність у відладці коду завдяки відображенню помилок на етапі компіляції.

TypeScript є сумісним з JavaScript і може використовувати будь-яку JavaScript-бібліотеку чи фреймворк. Крім того, він підтримує сучасні стандарти JavaScript, такі як ES6, ES7 та ES8, тому розробники можуть використовувати нові функції мови JavaScript.

TypeScript компілюється в JavaScript і може бути використаний у браузерах або на серверній стороні за допомогою Node.js. Він підтримується Microsoft та має активну спільноту розробників, що робить його популярним вибором для розробки веб-додатків та Node.js застосунків.

Перед тим як браузер чи Node.js може виконати код TypeScript, його необхідно скомпілювати в JavaScript (рис. 2.1). TypeScript компілюється у чистий JavaScript за допомогою спеціального компілятора, який перетворює код TypeScript у код JavaScript [8].

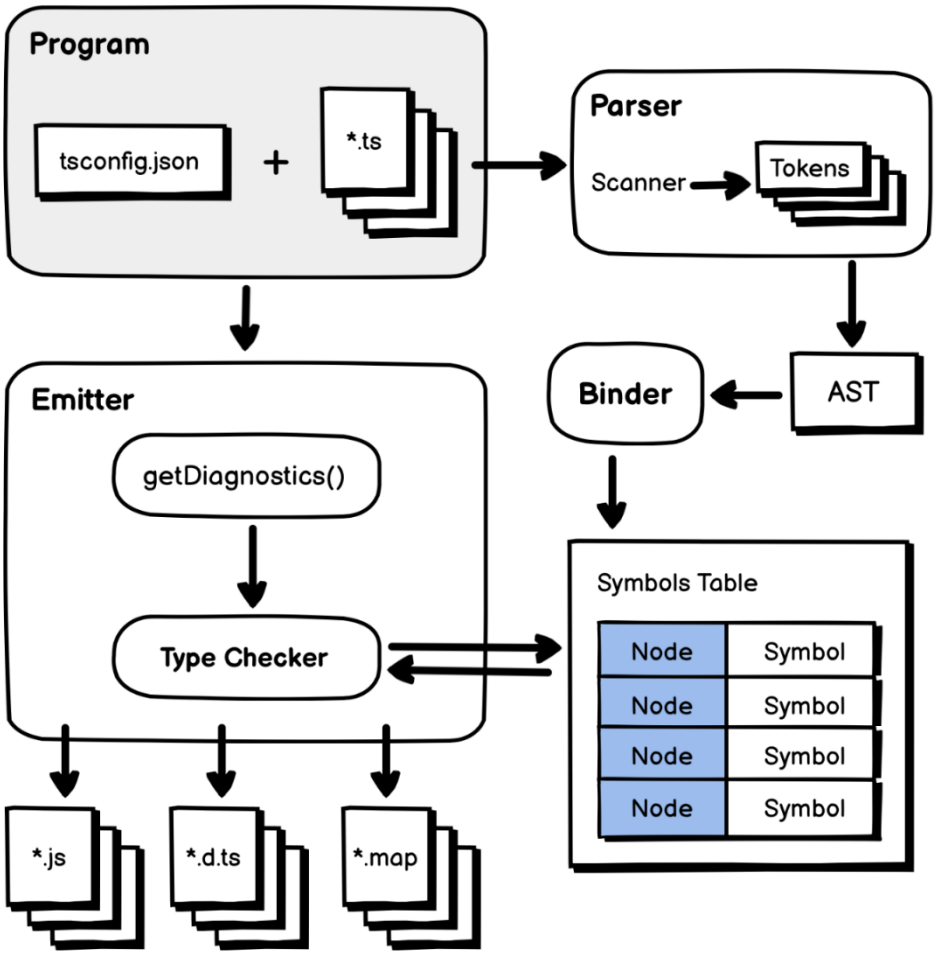


Рис. 2.1 – Компіляція TypeScript

Основний процес компіляції TypeScript складається з таких етапів:

- Лексичний та синтаксичний аналіз коду TypeScript. Цей етап відповідає за перетворення вихідного коду в абстрактне синтаксичне дерево (AST), що репрезентує синтаксис програми.
- Перевірка типів. На цьому етапі компілятор перевіряє типи всіх змінних, функцій та інших елементів програми, вказаних у коді.
- Генерація коду JavaScript. Після того як компілятор перевірив типи всіх елементів програми, він генерує код на чистому JavaScript, який може виконуватися в браузері чи на сервері.

Крім того, компілятор TypeScript підтримує декілька конфігураційних опцій, які дозволяють налаштувати процес компіляції залежно від потреб розробника.

Наприклад, можна включити підтримку новітніх функцій JavaScript, відключити певні перевірки типів або зберегти всі коментарі під час компіляції.

Опис мови програмування Java

Java – це об'єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems (пізніше придбана компанією Oracle). Вона є однією з найпопулярніших мов програмування у світі, використовується для створення різноманітних програм, веб-додатків, мобільних додатків та інших систем.

Java є мовою програмування з високим рівнем абстракції, що означає, що вона приховує від програміста деталі роботи з оперативною пам'яттю та іншими системними ресурсами. Вона підтримує генерацію коду для віртуальної машини Java (JVM), що дозволяє виконувати програми на різних платформах без необхідності перекомпіляції.

Java підтримує об'єктно-орієнтований підхід до програмування, що означає, що всі дані та функції в програмі представлені у вигляді об'єктів. Вона також підтримує багатопотоковість, що дозволяє програмам працювати з кількома потоками виконання одночасно.

Однією з ключових особливостей Java є її безпека. Java використовує механізми безпеки, які дозволяють запобігти виконанню шкідливого коду. Крім того, вона має вбудовану систему управління пам'яттю, яка допомагає запобігти витоку пам'яті та інших помилках.

Java має велику кількість стандартних бібліотек та фреймворків, що дозволяє швидко розробляти складні програми. Вона підтримує розробку веб-додатків з використанням технологій, таких як JavaServer Pages (JSP) та JavaServer Faces (JSF), а також мобільних додатків.

Однією зі значущих особливостей мови програмування Java є платформонезалежність. Код, написаний на Java, можна виконувати на будь-якій платформі, що підтримує відповідну машину (Java Virtual Machine або JVM). Це дає розробникам змогу створювати програми для різних операційних систем, не переписуючи код заново. Більше того, це дозволяє розробникам створювати

програми для різних платформ, включаючи мобільні пристрої та вбудовані системи.

Java також відома своєю високою масштабованістю. Це означає, що програми, написані на Java, можуть бути розширені для роботи з великими об'ємами даних або високою навантаженістю. Завдяки технології Java Enterprise Edition (Java EE), розробники мають змогу створювати потужні корпоративні застосунки, які можуть обробляти тисячі запитів в секунду.

Однією з інших особливостей Java є її вбудована система керування пам'яттю, яка автоматично вивільняє пам'ять, використану об'єктами, які більше не потрібні програмі. Це дозволяє розробникам уникнути багатьох типов помилок, пов'язаних з управлінням пам'яттю, таких як підтримка викидання винятків та невизначеність поведінки.

Крім того, Java має багатий стандартний набір бібліотек, які дозволяють розробникам швидко створювати програми різних типів, включаючи графічні інтерфейси, мережеві додатки та багато інших. Це дозволяє розробникам зосередитися на логіці програми, не витрачаючи час на написання власних бібліотек [8].

Опис Java бібліотеки Spring Boot

Spring Boot – це фреймворк для розробки веб-додатків на мові програмування Java. Він є одним з найпопулярніших фреймворків у світі Java, оскільки дозволяє розробникам швидко створювати високоякісні веб-додатки з мінімальною кількістю конфігурації.

Основні особливості Spring Boot:

- Автоматична конфігурація: Spring Boot дозволяє розробникам швидко створювати веб-додатки з мінімальною кількістю конфігурації. Завдяки автоматичній конфігурації, більшість налаштувань виконуються автоматично, що дозволяє значно скоротити час розробки.
- Вбудований веб-сервер: Spring Boot містить вбудований веб-сервер, що дозволяє запустити додаток без необхідності встановлення та налаштування окремого веб-сервера.

– Монолітна архітектура: Spring Boot дозволяє створювати монолітні веб-додатки, тобто додатки, які складаються з однієї складової. Це дозволяє простіше відлагоджувати та підтримувати додатки.
– Підтримка мікросервісної архітектури: Spring Boot також підтримує мікросервісну архітектуру, що дозволяє створювати веб-додатки, які складаються з багатьох дрібних складових.
– Багата екосистема: Spring Boot має багату екосистему, що дозволяє розробникам швидко та легко додавати нові функціональність до своїх додатків.

Архітектура Spring Boot (рис. 2.2) базується на паттерні Model-View-Controller (MVC) та інших принципах, що дозволяють забезпечити ефективне розроблення та масштабування веб-додатків.

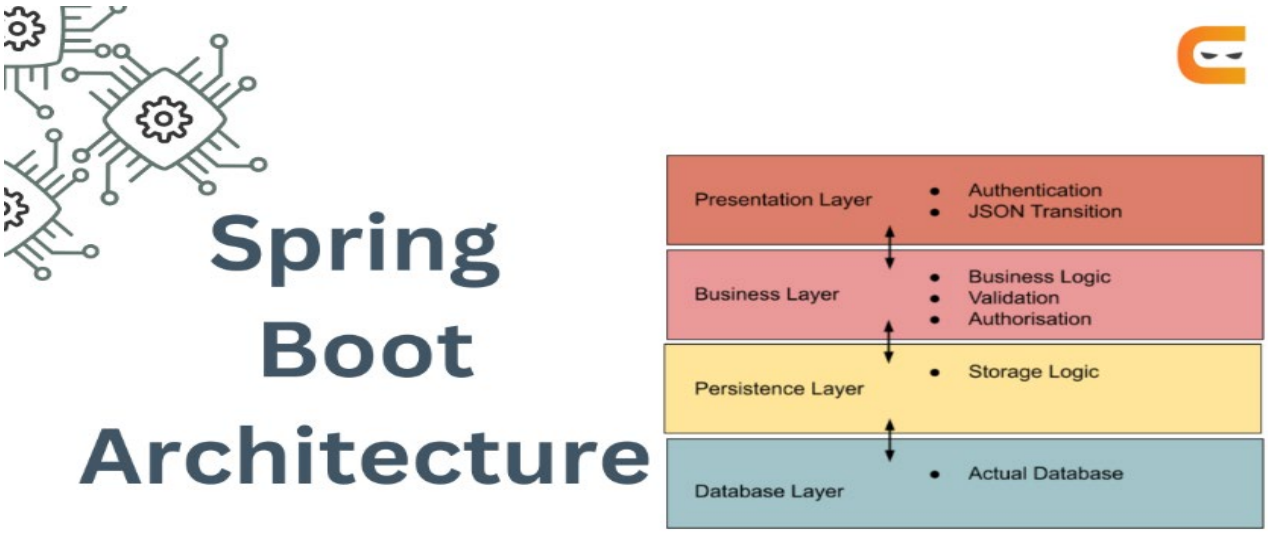


Рис. 2.2 – Архітектура Spring Boot

Spring boot architecture

Головні компоненти архітектури Spring Boot:

– Spring Core: це базовий компонент, який забезпечує основні функції, такі як інверсія керування (Inversion of Control, IoC) та управління залежностями. Він дозволяє забезпечити розширення функціональності за допомогою плагінів.

- Spring MVC: це компонент, який забезпечує реалізацію паттерну Model-View-Controller для веб-додатків.
- Spring Data: це компонент, який забезпечує доступ до баз даних через ORM-фреймворки, такі як Hibernate або JPA.
- Spring Security: це компонент, який забезпечує механізми аутентифікації та авторизації для веб-додатків.

Архітектура Spring Boot базується на конфігурації за замовчуванням, що дозволяє прискорити розробку та підтримку веб-додатків. Spring Boot також підтримує мікросервісну архітектуру, що дозволяє розробляти додатки з використанням невеликих та самодостатніх сервісів.

У цілому, архітектура Spring Boot дозволяє забезпечити ефективну та швидку розробку веб-додатків з використанням популярних фреймворків та бібліотек [8].

Опис фреймворку React

React – це відкрита JavaScript бібліотека, яка дозволяє розробляти користувацькі інтерфейси (UI) для веб-додатків. React був створений Facebook і перші версії були випущені в 2013 році. Однією з головних переваг React є його можливість працювати з великими та складними інтерфейсами, розбиваючи їх на компоненти, які можна легко керувати.

Основними поняттями, які використовуються в React, є компоненти, властивості та стан. Компонент - це самостійна частина UI, яка може містити HTML, CSS, JavaScript та інші компоненти. Властивості - це дані, які передаються в компонент ззовні, а стан - це дані, які контролюються внутрішньо компонентом.

Один з основних принципів React - це однонаправлений потік даних, що дозволяє легко розробляти та підтримувати код. React також дозволяє використовувати JSX - синтаксис, який поєднує HTML та JavaScript, що дозволяє легко створювати компоненти та використовувати їх у коді.

Окрім цього, React має велику кількість додаткових бібліотек та інструментів, які дозволяють розширювати його можливості та полегшувати розробку. Наприклад, Redux – це бібліотека, яка дозволяє керувати станом додатку, а React Router – це бібліотека для маршрутизації у веб-додатках на React.

Загалом, React є потужним інструментом для розробки користувацьких інтерфейсів у веб-додатках, який дозволяє швидко та ефективно створювати складні інтерфейси з великою кількістю функціональності.

React має ряд методів життєвого циклу, які дають можливість виконувати певні дії під час різних етапів життєвого циклу компонента (рис. 2.3).

1. Mounting - це етап, коли компонент вставляється в DOM. В цей момент відбувається створення компонента та ініціалізація його властивостей і стану.

   Методи життєвого циклу, що виконуються під час цього етапу:

   a) constructor(): викликається при створенні компонента та ініціалізує його властивості і стан.
   b) getDerivedStateFromProps(): викликається при оновленні властивостей компонента.
   c) render(): викликається для створення представлення компонента.

2. Updating – це етап, коли компонент оновлює свій стан або властивості. В цей момент відбувається оновлення компонента, тобто його властивостей і стану.

   Методи життєвого циклу, що виконуються під час цього етапу:

   a) getDerivedStateFromProps(): викликається при оновленні властивостей компонента.
   b) shouldComponentUpdate(): викликається для перевірки того, чи потрібно оновити компонент.
   c) render(): викликається для створення представлення компонента.
   d) getSnapshotBeforeUpdate(): викликається перед оновленням компонента і повертає значення, яке буде передане до методу componentDidUpdate().
   e) componentDidUpdate(): викликається після оновлення компонента.

3. Unmounting - це етап, коли компонент видаляється з DOM. В цей момент відбувається видалення компонента та звільнення ресурсів.

   Методи життєвого циклу, що виконуються під час цього етапу:

   a) componentWillUnmount() – викликається перед тим, як компонент буде видалений з DOM. Він дозволяє виконати необхідні очищення, зупинити

підписки на події або таймери, щоб уникнути підвисання програми або витоку пам'яті. Важливо не забувати видаляти усі підписки та зв'язки, щоб уникнути непередбачуваної поведінки.

b) componentDidCatch() – викликається, коли виникає помилка в будь-якому дочірньому компоненті. Він дозволяє компоненту обробити помилку і повернути альтернативний контент. Цей метод є корисним для відлагодження і покращення стабільності програми.

c) shouldComponentUpdate() – викликається, коли компонент отримує нові властивості або стан. Він дозволяє компоненту визначити, чи потрібно перерендерити компонент. Якщо метод повертає false, компонент не буде перерендерений, що може покращити продуктивність програми.

d) getDerivedStateFromError() – використовується, коли виникає помилка під час рендеринга компонента. Він дозволяє компоненту змінити стан, щоб відобразити помилку. Цей метод допомагає забезпечити стабільність програми та уникнути непередбачуваної поведінки [9 – 10].

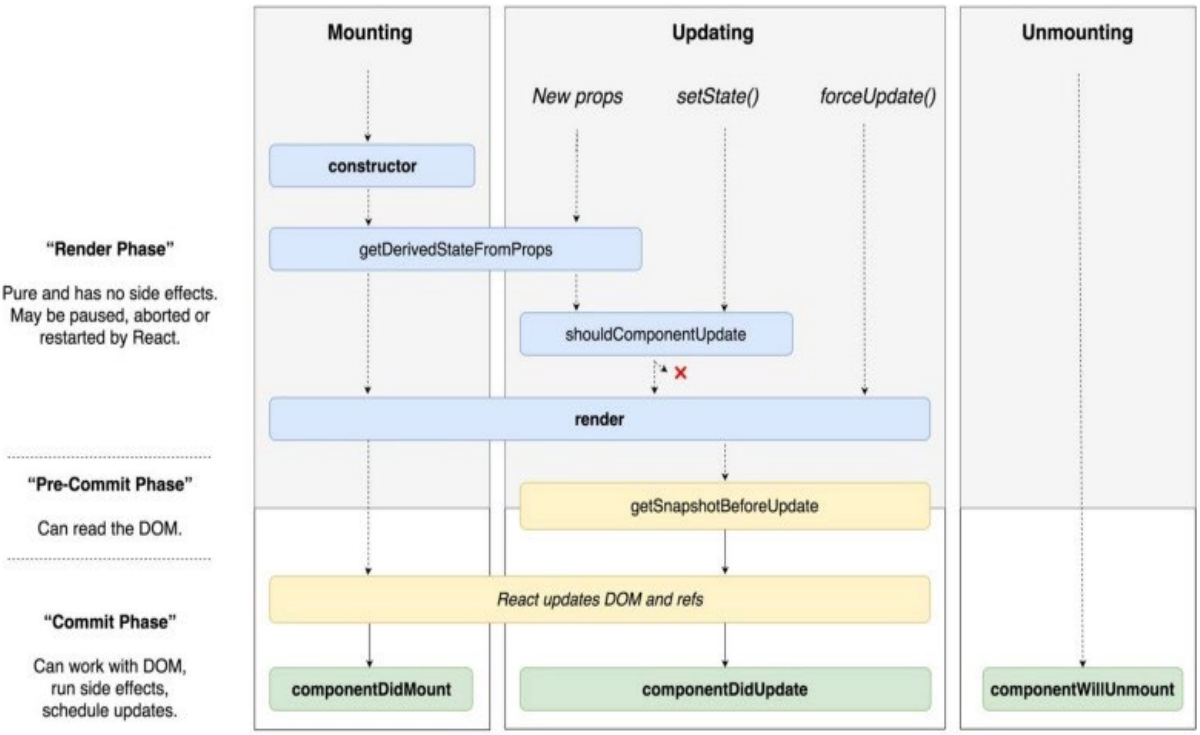


Рис. 2.3 – Діаграма методів життєвого циклу React

Опис бібліотеки Redux

Redux – це бібліотека для управління станом додатка в JavaScript-додатках з використанням архітектури Flux. Вона забезпечує зручний спосіб зберігати та оновлювати стан додатка, який може бути легко спільно використовуваний між компонентами та контейнерами.

Redux базується на трьох основних принципах: стор, дії та редуктори. Стор – це об'єкт, який містить увесь стан додатка. Дії – це об'єкти, які відображають події, що відбуваються в додатку, такі як взаємодія користувача з інтерфейсом. Редуктори – це функції, які приймають попередній стан та дію та повертають новий стан.

Redux також забезпечує механізм підписки на зміни стану додатка, що дозволяє компонентам оновлюватися автоматично при зміні стану.

Redux є популярним вибором для додатків з великою кількістю стану та/або великою кількістю компонентів, що взаємодіють зі станом. Він також дозволяє докладніше налаштувати спосіб, яким стан обробляється та оновлюється в додатку. Однак, він може бути важким для розуміння та використання для менших проєктів, де немає потреби в складному управлінні станом.

Store, Actions та Reducers (рис. 2.4) – три ключові складові бібліотеки Redux, які використовуються для управління станом додатка.

Store – це центральний об'єкт у Redux, який містить увесь стан додатка. Інші компоненти взаємодіють зі Store, щоб зчитувати та змінювати стан. Store зберігає стан в одному об'єкті, який неможливо змінити без використання Actions та Reducers.

Actions – об'єкти, які описують зміни стану додатка. Кожен Action містить тип та додаткові дані, які потрібні для зміни стану. Actions створюються за допомогою функцій, які називаються Action Creators.

Reducers – функції, які обробляють Actions та змінюють стан додатка. Кожен Reducer приймає поточний стан та Action, а потім повертає новий стан. Reducers не можуть змінювати поточний стан безпосередньо - вони повинні повернути новий стан.

Основна ідея за використанням Store, Actions та Reducers в Redux полягає в тому, що вони допомагають зробити стан додатка передбачуваним та контрольованим. Крім того, вони полегшують розробку складних додатків та роблять їх більш масштабованими [11].

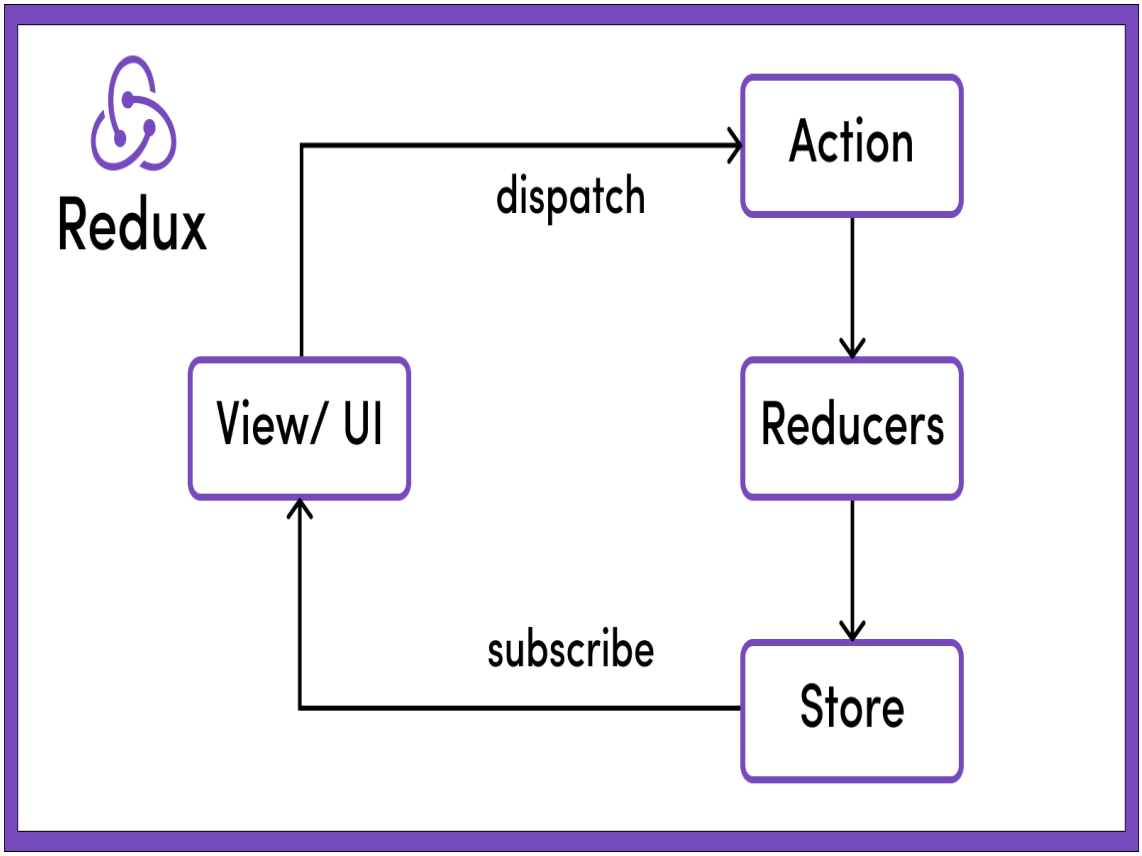


Рис. 2.4 – Ключові компоненти редакса: Store, Actions та Reducers

Опис CSS

CSS (Cascading Style Sheets) – це мова стилів, яка використовується для опису зовнішнього вигляду документів HTML, XHTML та XML. За допомогою CSS можна визначати колір, розмір, шрифт, відступи, рамки та інші властивості для різних елементів веб-сторінки.

CSS має ряд важливих особливостей, серед яких:

– Розділення змісту та вигляду: CSS дозволяє відокремлювати зовнішній вигляд документа від його змісту, що полегшує редагування та збереження коду.

- Каскадність: CSS забезпечує каскадність стилів, що означає, що стилі можуть успадковуватися від одного елементу до іншого. Це дозволяє зменшувати кількість коду, необхідного для задання стилів.
- Селектори: CSS дозволяє використовувати різні типи селекторів для вибору елементів, до яких будуть застосовуватися стилі.
- Пріоритет: CSS має систему пріоритетів, що дозволяє встановлювати, які стилі мають більший пріоритет над іншими.
- Розміщення: CSS можна використовувати для розміщення елементів на сторінці за допомогою властивостей позиціонування.

CSS є важливим інструментом для веб-розробки, оскільки дозволяє зберігати час і зусилля, які зазвичай потрібні для розташування та форматування вмісту веб-сторінки. Існує багато бібліотек та фреймворків CSS, які дозволяють використовувати готові стилі та компоненти для швидкої розробки веб-додатків.

2.4. Опис структури системи та алгоритмів її функціонування

Веб-застосунок побудований на основі MVC підходу.

MVC (Model-View-Controller) – це популярний архітектурний підхід в програмуванні, який розділяє програму на три головних компоненти: модель (Model), представлення (View) та контролер (Controller).

Модель відповідає за дані та логіку бізнес-процесів. Вона зберігає дані та інформацію про стан системи, а також містить методи для їх обробки та зміни.

Представлення відображає дані користувачу та забезпечує можливість взаємодії з системою. Вона може використовувати HTML, CSS, JavaScript та інші технології для створення інтерфейсу користувача.

Контролер взаємодіє з користувачем та моделлю. Він обробляє запити користувача, взаємодіє з моделлю для зберігання та отримання даних та відправляє відповідь користувачу.

MVC-архітектура (рис. 2.5) забезпечує розподіл відповідальностей між компонентами, що дозволяє легко змінювати кожен компонент окремо, не

змінюючи інші. Це зроблює код більш модульним та підтримуваним, що знижує витрати на розробку та підтримку програмного забезпечення [12 – 13].

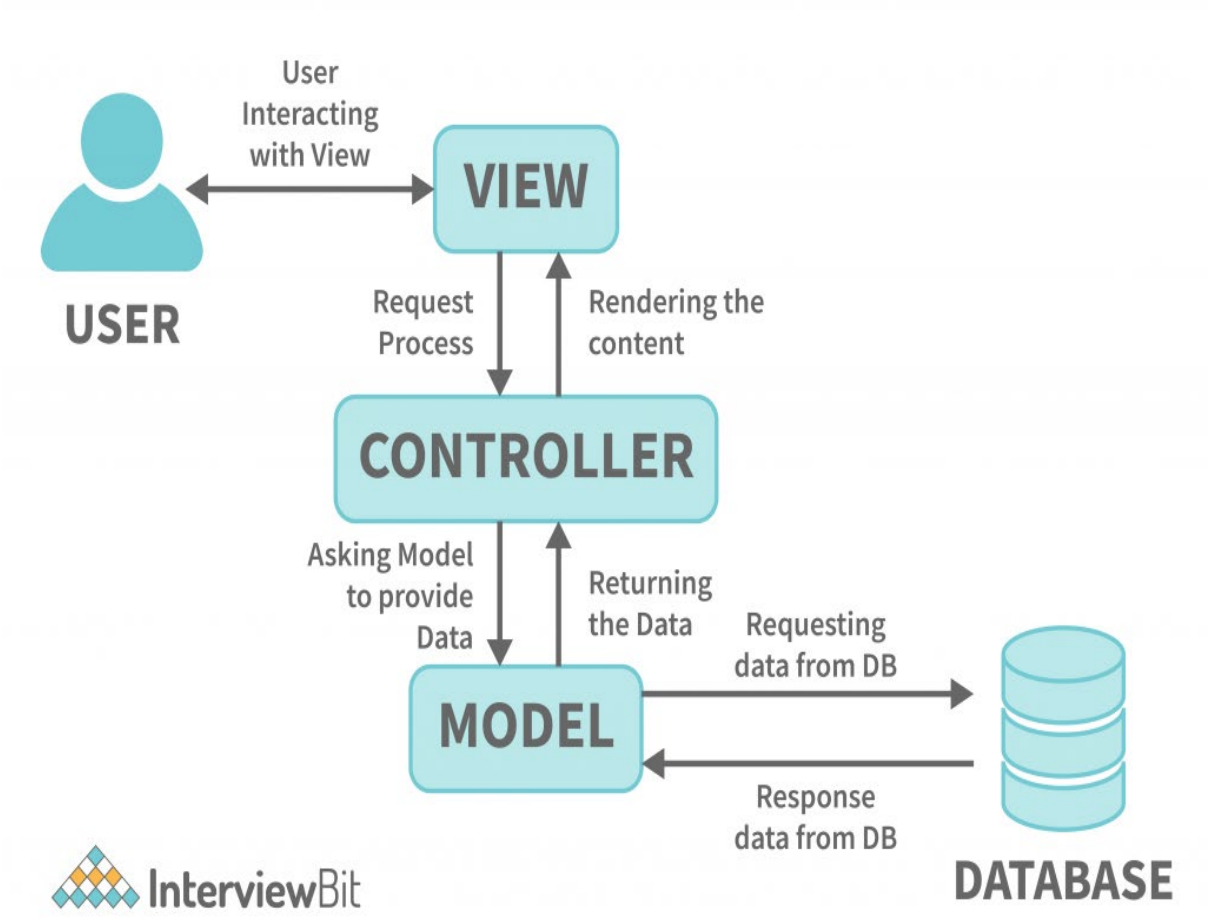


Рис. 2.5 – Архітектурний підхід MVC

Розглянемо серверну частину додатку.

Додаток реалізовано з використанням реляційної бази даних PostgreSQL, вона має ряд переваг:

– Вона використовує потужну та широко відому мову запитів SQL, яка дозволяє робити дуже специфічні та складні запити.
– Можливість для нормалізації, що дозволяє уникнути дублювання даних та зменшити розмір бази.
– Масштабованість. Реляційні бази даних дуже добре масштабуються вгору та вниз, що дозволяє легко розширювати та зменшувати розміри бази в залежності від потреб користувачів.

– Безпека даних. Реляційні бази даних надають можливості для забезпечення безпеки даних, такі як використання різних рівнів доступу, шифрування даних тощо [14 – 17].

Розробка веб-платформи для соціальної журналістики з використанням реляційної бази даних має кілька переваг, які роблять її кращим вибором ніж нереляційної.

Реляційна база даних дозволяє організовувати дані у вигляді структурованих таблиць з визначеними стовпцями і взаємозв'язками між ними. Це сприяє зручній організації і збереженню різних типів даних, які зустрічаються в соціальній журналістиці, таких як користувачі, статті, коментарі і т.д.

У соціальній журналістиці важливі зв'язки між різними об'єктами даних, такими як користувачі, статті, теги і т.д. Реляційна база даних дозволяє визначати та підтримувати ці зв'язки за допомогою зовнішніх ключів і виконувати операції з'єднання для отримання пов'язаних даних. Це дозволяє ефективно моделювати складні взаємозв'язки між даними.

Реляційна база даних надає потужну мову запитів, що дозволяє виконувати різноманітні операції над даними, такі як фільтрація, сортування, групування та об'єднання. У соціальній журналістиці часто потрібно виконувати складні запити для отримання агрегованих даних, наприклад, підрахунок кількості користувачів до статті або отримання найпопулярніших статей. Реляційна база даних дозволяє ефективно виконувати такі операції.

Схема бази даних зображена на рисунку 2.6.

Для організації даних використовуються 6 таблиць:

– User – зберігає всіх юзерів, які зареєстровані у додатку.
– Post – зберігає статті, які були опубліковані.
– Post_Reaction – зберігає дані про реакції, які користувачи залишили для статті.
– Highlight – зберігає цитати, які юзер виділив.
– Favorite – зберігає дані про те, які статті к додав в обрані.
– Tag – зберігає теги, які можуть бути обрані при створені статті [18].

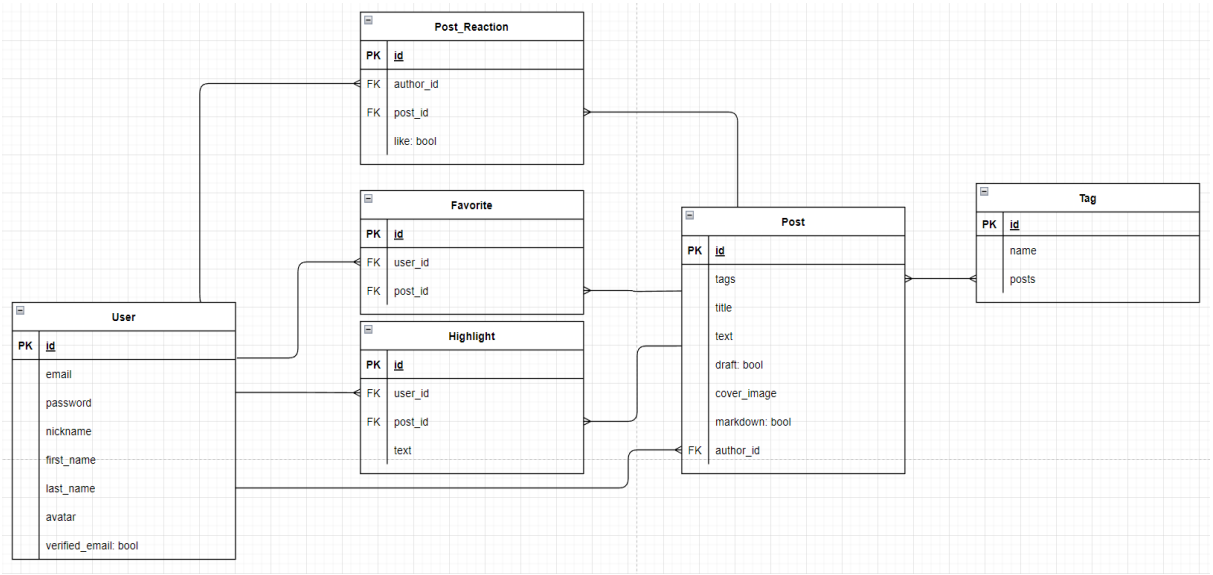


Рис. 2.6 – Схема бази даних

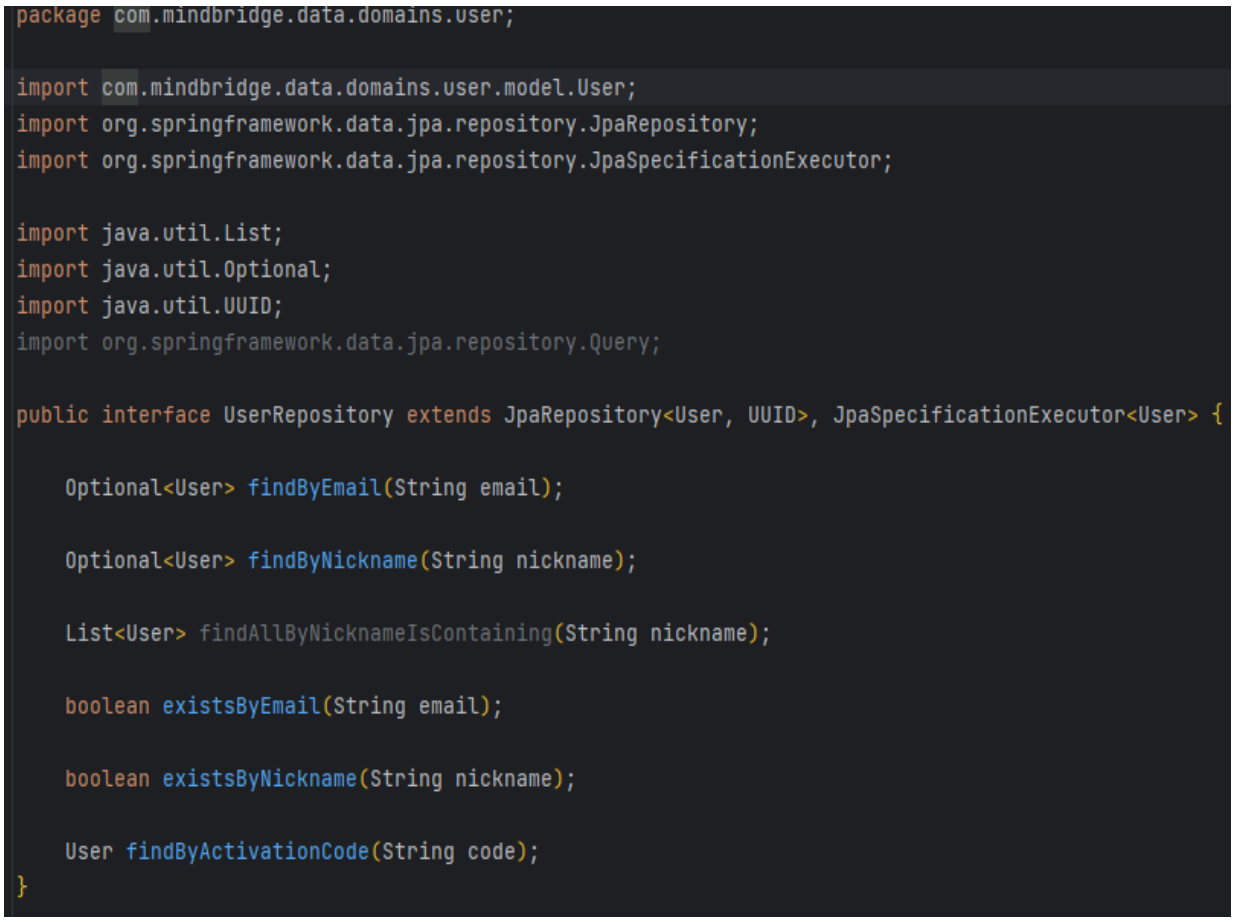
Для того, щоб розділити логіку доступу до даних від бізнес логіки використовується патерн репозиторій, що забезпечує легку та прозору роботу з даними та зменшення залежності від конкретної реалізації бази даних.

Репозиторії зазвичай містять основні CRUD-методи (створення, читання, оновлення та видалення даних) та додаткові методи, які пов'язані з конкретними операціями. Також є можливість писати свої запити до бази даних всередині об’єкта репозиторія. Для реалізації патерну була використана Java бібліотека JPA.

Java Persistence API (JPA) є стандартною бібліотекою для роботи з об'єктно-реляційним відображенням (ORM) в Java-додатках. Вона надає програмістам можливість взаємодіяти з реляційною базою даних, використовуючи об'єктно-орієнтовані підходи.

JPA спрощує розробку базових операцій з базою даних, таких як створення, оновлення, видалення і читання об'єктів. Вона визначає анотації, які можна застосовувати до класів та їх полів, щоб вказати відображення між об'єктами Java і таблицями бази даних. За допомогою JPA можна використовувати мову запитів JPQL (Java Persistence Query Language), яка надає можливість здійснювати запити до бази даних, використовуючи об'єктно-орієнтований підхід.

Приклад реалізації репозиторію зображений на рисунку 2.7.



```java
package com.mindbridge.data.domains.user;

import com.mindbridge.data.domains.user.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

import java.util.List;
import java.util.Optional;
import java.util.UUID;
import org.springframework.data.jpa.repository.Query;

public interface UserRepository extends JpaRepository<User, UUID>, JpaSpecificationExecutor<User> {

    Optional<User> findByEmail(String email);

    Optional<User> findByNickname(String nickname);

    List<User> findAllByNicknameIsContaining(String nickname);

    boolean existsByEmail(String email);

    boolean existsByNickname(String nickname);

    User findByActivationCode(String code);
}
```

Рис. 2.7 – Приклад реалізації UserRepository

Також для кожного репозиторія потрібно визначити Domain Entity – це об’єкт який представляє сутність домену додатку (наприклад, користувач, замовлення, продукт тощо). Ці об'єкти мають властивості, які описують їх характеристики, та методи, які дозволяють взаємодіяти з ними.

У паттерні репозиторій, об'єкти Domain Entity використовуються як об'єкти, з якими працює сам паттерн. Репозиторій дозволяє зберігати, зчитувати, оновлювати та видаляти об'єкти Domain Entity з бази даних (рис. 2.8).

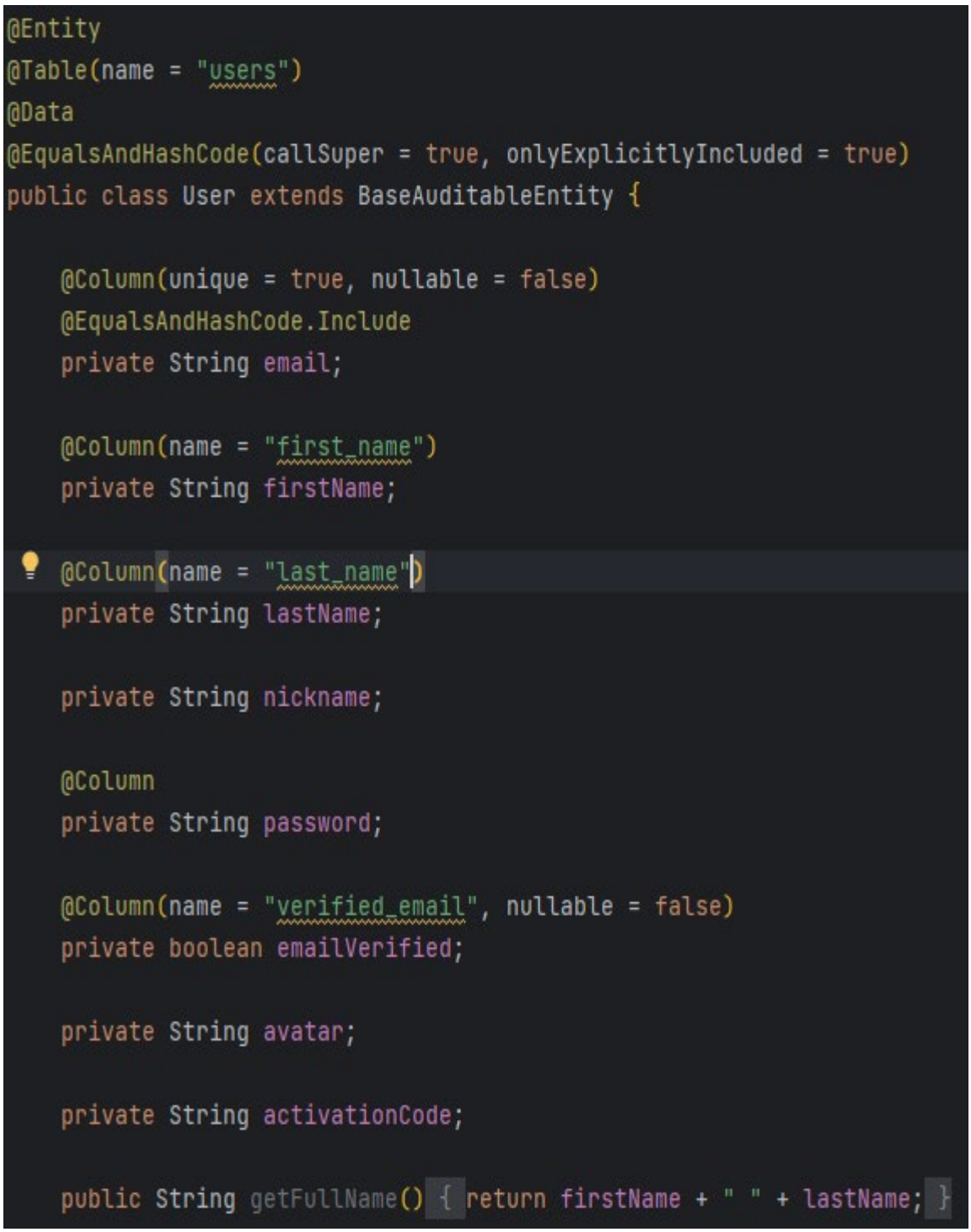


Рис. 2.8 – Domain Entity User

У додатку комопненти Model та Controller розділені по різним директоріям, тому репозиторії та їх домени, які відповідають за реалізацію компонента Model знаходяться у окремій директорії під назвою db (рис. 2.9).

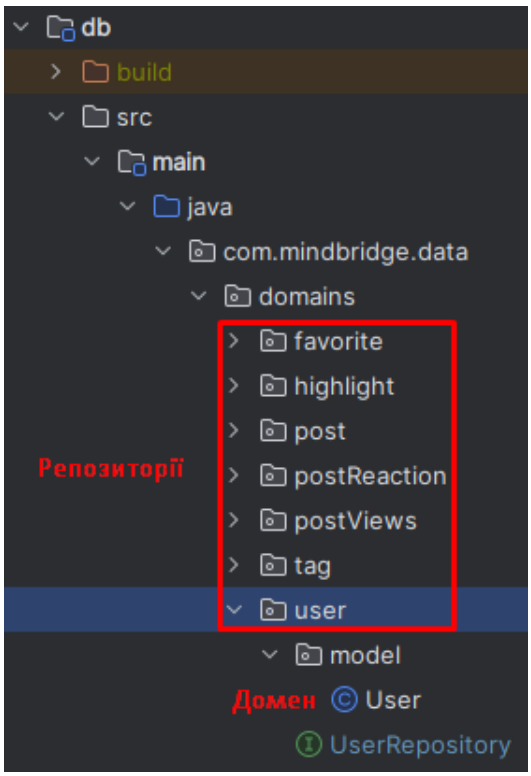


Рис. 2.9 – Ієрархія директорії db

Все, що стосується компонента Controller та усієї бізнес логіки додатку знаходиться у директорії app (рис. 2.10).

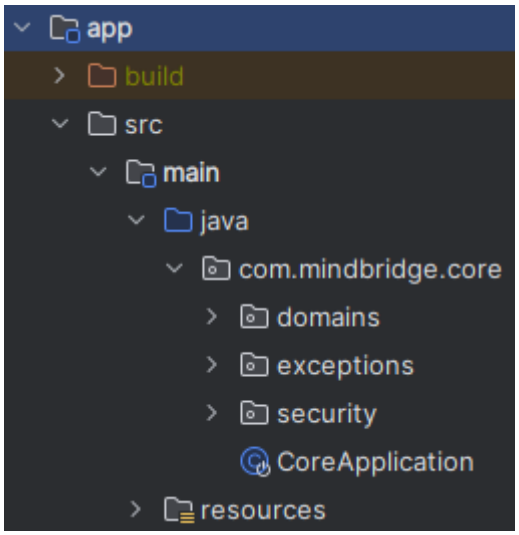


Рис. 2.10 – Ієрархія директорії app

У директорії domains знаходяться сутності, які використовуються у додатку. Кожна сутність реалізує конкретну таблицю у базі даних (рис 2.10).

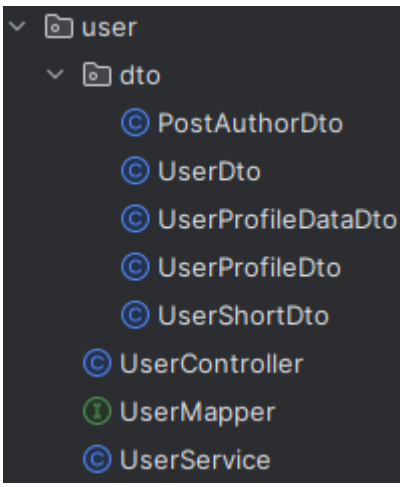


Рис. 2.11. – Директорія з DTO сутності User

В папці кожної сутності знаходиться три файли та папка в якій зберігаються DTO (Data Transfer Object). У патерні репозиторій зазвичай використовуються DTO (Data Transfer Object). Основна відмінність між DTO та Domain Entity полягає у тому, що Domain Entity представляє собою об'єкт з повною інформацією, яка зберігається в базі даних, тоді як DTO - це об'єкт, який містить лише ту інформацію, яка потрібна для передачі між окремими компонентами системи.

У файлі, який містить слово mapper, знаходяться інтерфейси, що реалізують процес мапінгу. Мапінг використовують для перетворення Domain Entity на DTO і навпаки. Наприклад, якщо потрібно вивести на екран інформацію про конкретного користувача, то потрібно взяти об'єкт User (Domain Entity) з бази даних та перетворити його на об'єкт UserDTO (DTO), який містить лише ті поля, які потрібні для відображення на екрані. Це дозволяє зменшити кількість запитів до бази даних та покращити продуктивність системи.

```java
package com.mindbridge.core.domains.user;

import com.mindbridge.core.domains.user.dto.*;
import com.mindbridge.data.domains.user.model.User;
import org.mapstruct.*;
import org.mapstruct.factory.Mappers;

@Mapper
public interface UserMapper {

    UserMapper MAPPER = Mappers.getMapper(UserMapper.class);

    @Mapping(target = "postsQuantity", ignore = true)
    @Mapping(target = "lastArticleTitles", ignore = true)
    @Mapping(target = "rating", ignore = true)
    @Mapping(target = "fullName", ignore = true)
    public abstract UserProfileDto userToUserProfileDto(User user);

    @Mapping(target = "postsQuantity", ignore = true)
    @Mapping(target = "rating", ignore = true)
    UserDto userToUserDto(User user);
}
```

Рис. 2.12 – Інтерфейс UserMapper

Компонент Controller являє собою реалізацію REST API.

API (Application Pro-gramming Interface) або прикладні програмні інтерфейси являються набором програмного коду, що забезпечує передачу даних між одним програмним продуктом та іншим. Вони також містять умови цього обміну даними та слугують інтерфейсами, що до-зволяють програмам спілкуватися одна з одною.

REST – це набір архітектурних обмежень, а не протокол чи стандарт. Розробники API можуть реалізовувати REST по-різному.

Коли клієнтський запит виконується через RESTful API, він передає представлення стану ресурсу запитувачеві або кінцевій точці. Ця інформація, або представлення, передається у одному з кількох форматів за допомогою протоколу HTTP: JSON (Javascript Object Notation), HTML, XML, Python, PHP або простий текст. JSON є найпопулярнішим форматом файлу, оскільки, незважаючи на свою

назву, він не залежить від конкретної мови програмування і може бути прочитаний як людьми, так і машинами.

Заголовки та параметри мають велике значення в HTTP-методах запиту RESTful API, оскільки вони містять інформацію про метадані запиту, авторизацію, уніфікований ідентифікатор ресурсу (URI), кешування, cookies та інше [19].

У файлі controller знаходяться кінцеві точки до яких може звертатися клієнтська частина з цілю отримання даних. Задача контролера обробити запит, дістати з нього дані та викликати необхідний сервіс. Дані можуть знаходитися або у шляху кінцевої точки або у тілі HTTP запиту. Після того, як сервіс завершить своє виконання, контролер має повернути дані (якщо це необхідно) та статус запиту – успішний або помилковий.

```java
@RestController
@RequestMapping("/user")
@Validated
public class UserController {

    private final UserService userService;

    @Autowired
    public UserController(UserService userService) { this.userService = userService; }

    @GetMapping("/{id}")
    public UserProfileDto getUserProfileInfo(@PathVariable UUID id, Principal principal) {
        return userService.getUserProfileInformation(id, principal);
    }

    @PostMapping("/update/{id}")
    public UserDto updateUserData(@PathVariable UUID id, @RequestBody UserProfileDataDto userProfileData) {
        return userService.updateUserById(id, userProfileData);
    }

    @PostMapping("/check/nickname")
    public boolean checkUserNickname(@RequestBody String nickname) {
        String nick = nickname.substring(0, nickname.length() - 1);
        return userService.checkNickname(nick);
    }

    @PostMapping("/check/password/{id}")
    public boolean checkUserPassword(@PathVariable UUID id, @RequestBody String password) {
        return userService.checkPassword(id, password);
    }

    @PostMapping("/update/avatar/{id}")
    public UserDto updateUserAvatar(@PathVariable UUID id, @RequestBody String url) {
        return userService.updateUserAvatarById(id, url);
    }
```

Рис. 2.13 – Клас UserController

Сервіси відповідають за бізнес-логіку додатку. Вони звертаються до репозиторіїв, коли необхідно отримати дані та можуть робити над ними різні операції.

```java
@Service
@Slf4j
public class UserService implements UserDetailsService {

    private final UserRepository userRepository;

    private final PostRepository postRepository;

    private final PasswordEncoder passwordEncoder;

    private final PostReactionRepository postReactionRepository;

    @Lazy
    @Autowired
    public UserService(UserRepository userRepository, PostRepository postRepository, PostReactionRepository postReactionRepository) {
        this.userRepository = userRepository;
        this.postReactionRepository = postReactionRepository;
        this.passwordEncoder = new PasswordConfig().passwordEncoder();
        this.postRepository = postRepository;
    }

    public UserProfileDto getUserProfileInformation(UUID userId, Principal principal) {
        var foundUser = userRepository.findById(userId)
                .orElseThrow(() -> new IdNotFoundException("User with id : " + userId + " not found."));
        var user = UserMapper.MAPPER.userToUserProfileDto(foundUser);
        var userReactions = postReactionRepository.getPostReactionByAuthorId(userId);
        List<Post> top5Posts = postRepository.getFirstPostTitles(userId, PageRequest.of( page: 0,  size: 5));
        user.setPostsQuantity(postRepository.countPostByAuthorId(userId));
        user.setUserReactions(userReactions.stream().map(UserReactionsDto::fromEntity).collect(Collectors.toList()));
        user.setLastArticleTitles(top5Posts.stream().map(PostTitleDto::fromEntity).collect(Collectors.toList()));
        user.setRating(calculateUserRating(userId));
        if (principal == null) {
            return user;
        }
        return user;
    }

    public long calculateUserRating(UUID userId) { return postReactionRepository.calcUserPostRating(userId); }
```

Рис. 2.14 – Клас UserService

Останній компонент серверної частини, який вартий уваги – автентифікація. До деяких частин додатку можуть мати доступ лише авторизовані користувач. Для того, щоб реалізувати це рівень захисту використовується підхід з Jwt ключами доступу та перезавантаження. JSON Web Token - це стандарт веб-токенів, який використовується для безпечної передачі інформації між двома сторонами. Він складається з трьох частин: заголовка, закодованого тіла та підпису. У закодованому тілі зберігається інформація про користувача, за допомогою якої його можна ідентифікувати. Автентифікація з використанням токену перезавантаження - це метод автентифікації, який дозволяє зберігати користувачів

у системі без необхідності постійного повторного введення логіну та паролю. При вхідній автентифікації, сервер видаватиме користувачу пару токенів: токен доступу та токен перезавантаження. Токен доступу містить обмежений термін дії, тоді як токен перезавантаження має довший термін дії та використовується для оновлення токена доступу без необхідності повторної аутентифікації користувача. При кожному оновленні токена доступу, сервер перевіряє валідність токена перезавантаження та видаватиме нову пару. Цей підхід дозволяє покращити безпеку, зручність та ефективність процесу аутентифікації в додатку.

Весь процес роботи серверної частини можна представити у вигляді діаграми:

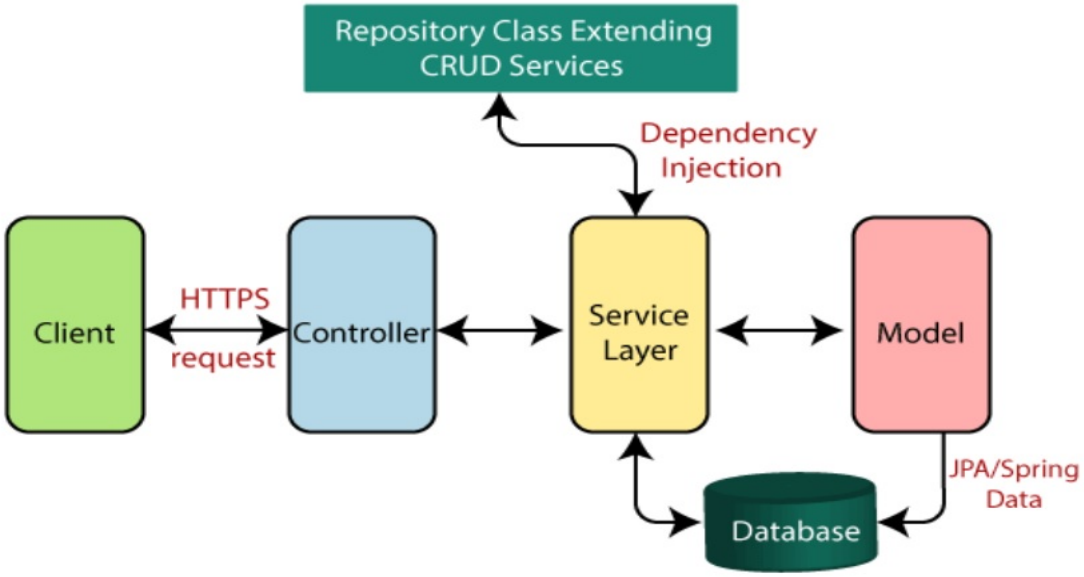


Рис. 2.15 – Алгоритм роботи серверної частини

Компонент View у системі представляє клієнтську частину додатку, він побудований з використанням Flux архітектури.

Flux це архітектурний паттерн запропонований компанією Facebook для побудови SPA (single-page-application). SPA – це додаток, який використовує єдиний HTML-документ як оболонку для усіх веб-сторінок, які взаємодіють з користувачем через динамічно завантажувані HTML, CSS, JavaScript. Тобто раніше, коли користувач хотів переключатися між сторінками одного веб-додатку,

кожного разу відкривалась нова вкладка і для неї підгружався весь необхідний код, на що витрачався час і ресурси, а користувач мав чекати. Тепер же всі переміщення по додатку відбуваються дуже швидко і користувачі витрачають мінімальну кількість часу на очікуванн, також це дуже сильно оптимізує роботу веб-додатку.

Flux пропонує розділити додаток на наступні частини:

- Store (Сховище);
- Dispatcher (Диспатчер);
- Views (Представлення);
- Action / Action Creators (Дія).

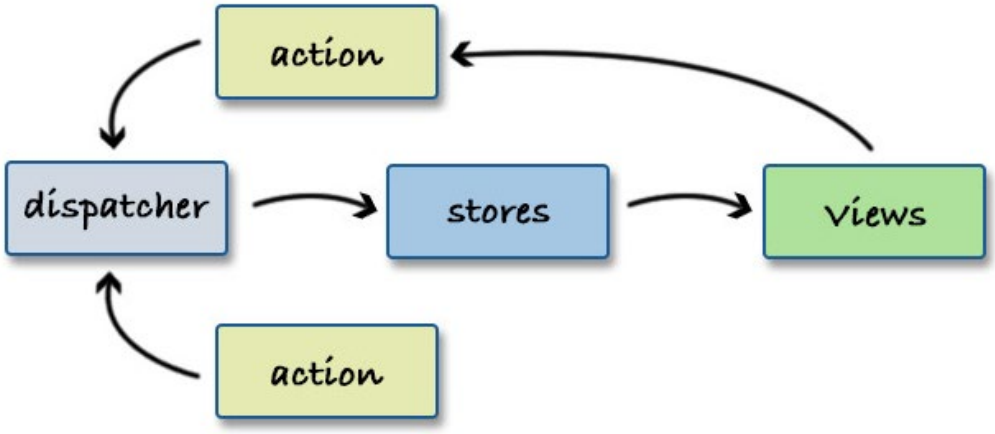


Рис. 2.16 – Представлення Flux архітектури

Store відповідає за зберігання стану додатку та за здійснення змін в ньому. Кожен Store містить деякий об'єм даних, який може бути запитаний компонентами (Views) або змінений через взаємодію з Actions.

Структура сховища базується на принципі "однонаправленого потоку даних". Всі зміни стану додатку повинні здійснюватись лише через диспетчера (Dispatcher). Диспетчер приймає дії від представлення та передає їх відповіднім сховищам. При зміні стану даних в сховище, він сповіщає про це відповідне представлення, який оновлює свій відображуваний стан.

Store також має підписників (Subscribers), які можуть бути повідомлені про зміну стану даних в ньому. Це дозволяє іншим частинам додатку, таким як інші сховища або сервіси, реагувати на зміни в даних та оновлювати свій власний стан.

Один з ключових аспектів сховища в Flux архітектурі - це те, що він повинен бути чистим (pure). Це означає, що він не повинен мати побічних ефектів, таких як мережеві запити чи збереження даних в браузері. Схлвище має бути лише контейнером для даних та логіки їхнього оновлення.

Загалом, сховище у Flux архітектурі є ключовим елементом, який дозволяє зберігати стан додатку та забезпечувати його централізоване керування. Це дозволяє забезпечувати стабільну та передбачувану роботу додатку, яка є особливо важливою в великих проектах з багатьма компонентами.

Диспатчер приймає дії, які відображають взаємодії користувача з додатком, та передає їх в сховище для обробки. Оскільки диспатчер є одним, він гарантує, що всі дії будуть оброблені в тому ж порядку, в якому вони були створені.

диспатчер також забезпечує можливість реєстрації обробників (callbacks), які викликаються при обробці кожної дії. Це дає змогу компонентам і сховищу підписатися на дії, які їх цікавлять, і виконувати відповідні дії при надходженні дії від користувача.

Важливо зазначити, що диспатчер не залежить від сховища та представлення. Він є незалежним шаром абстракції, який забезпечує механізми розподілу даних між ними. Це дозволяє легко замінювати або розширювати існуючі сховища та представлення без необхідності модифікувати сам диспатчер.

Також у додатку використовується бібіліотека Redux Saga. Її головне завдання покращення керування побічними ефектами (side effects) в Redux-додатка. Side effects – це дії, які не можуть бути прямо відтворені в чистій функціональній архітектурі Redux, такі як асинхронні запити до сервера, обробка даних з сервера, робота з локальним сховищем тощо [20].

Saga дозволяє відокремити бізнес-логіку від роботи з побічними ефектами, використовуючи генератори JavaScript для створення побічних задач (side task). Завдяки цьому розробник може більш просто і ефективно керувати асинхронними запитами, обробляти помилки та додавати логіку затримок, перетворення даних тощо.

Крім того, Saga дозволяє створювати більш складні функціональні послідовності, які можуть залежати від різних подій та станів додатка. Він дозволяє зв'язувати декілька побічних задач разом, здійснювати побічні дії до та після виконання дії, а також дозволяє зупиняти або скасовувати побічні задачі, якщо виникає необхідність.

Для того, щоб приєднати клієнтську частину до серверної, був сворений об’єкт під назвою api (рис. 2.17)

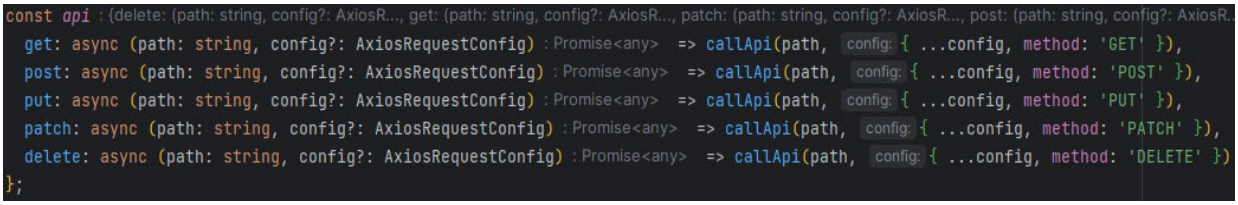


Рис. 2.17 – API об’єкт

Він містить у собі п'ять асинхронних функцій, вони делегують функції callApi операцію створення HTTP запиту.

Функція callApi (рис. 2.18) дістає токен доступу з локального сховища браузера та створює асинхронний запит до сервера за допомогою бібліотеки axios. Axios - це бібліотека JavaScript, яка використовується для виконання HTTP-запитів з веб-переглядача або з середовища Node.js. Вона надає простий та зручний спосіб взаємодії з веб-серверами. Також у функції callApi відбувається обробка події, коли у токена доступу закінчується строк дії і він має бути оновлений за допомогою токена перезавантаження.

Рис. 2.18 – Функція callApi

Ієрархія директорій клієнтської частини має наступний вигляд:

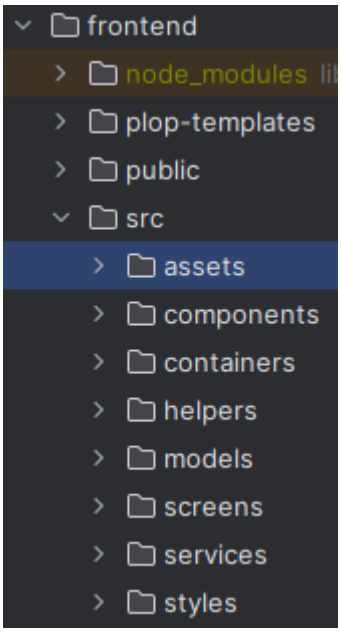


Рис. 2.19 – Ієрархія директорії frontend

У screens знаходяться директорії кореневих компонентів, які відображають головні сторінки додатку (створення статті, реєстрація, логін, і тд). Кожна з них всередині має ще одну ієрархію директорій.

Розглянемо роботу кореневого компонента на прикладі відображення всіх статтей на головній сторінці. Для отримання даних про всі наявні статті у компоненті FeedPage ми викликаємо функцію fetchData.

```
const handleLoadMorePosts = filtersPayload => {
  fetchData( arg: {
    params: filtersPayload,
    filter
  });
  setLoadMorePosts( arg: true);
};
```

Рис. 2.20 – Вигляд функції handleLoadMorePosts

В нижній частині комопнента ми підключаємо його до сховища Redux.

Функція mapStateToProps виконує мапінг потрібних частин стану Redux Store на властивості компонента. У ній ми можем бачити поле data, яке відповідає за зберігання постів. Це поле ми отримує зі сховища використовуючи шлях state.feedPageReducer.data.posts.

Об’єкт mapDispatchToProps виконує мапінг дій Redux на властивості компонента. У ньому знаходиться дія fetchData, яка відповідає за отримання всіх постів.

Функція connect поєднує функцію та об’єкт та створює високорівневий компонент, який має доступ до стану і дій Redux Store.

```
const mapStateToProps: (state: RootState) => IState = state : RootState  => ({
  data: state.feedPageReducer.data.posts,
  dataLoading: extractFetchDataLoading(state) || extractSearchPostsLoading(state),
  hasMore: state.feedPageReducer.data.hasMore,
  countResults: state.feedPageReducer.data.countResults,
  loadMore: state.feedPageReducer.data.loadMore,
  isAuthorized: state.auth.auth.isAuthorized,
  currentUser: state.auth.auth.user,
  userInfo: state.postPageReducer.data.profile,
  searchPosts: state.headerReducer.data.posts,
  allTags: state.postPageReducer.data.allTags
});

const mapDispatchToProps: IActions = {
  fetchData: fetchDataRoutine,
  setLoadMorePosts: addMorePostsRoutine,
  fetchUserProfile: fetchUserProfileRoutine,
  likePost: likePostRoutine,
  loadUser: loadCurrentUserRoutine,
  saveFavouritePost: saveFavouritePostRoutine,
  deleteFavouritePost: deleteFavouritePostRoutine,
  searchPostsByElastic: searchPostsRoutine,
  searchTitlesByElastic: searchPostsByElasticRoutine,
  loadCountResults: loadCountResultsRoutine,
  fetchTags: fetchTagsRoutine,
  fetchUserData: fetchUserRoutine
};

export default connect(mapStateToProps, mapDispatchToProps)(FeedPage);
```

Рис. 2.21 – Підключення компонента до сховища Redux

Після виклику дії fetchData, управління передається у Saga функцію, яка знаходиться у директорії sagas. Її задача викликати сервіс, який відповідає за відправку запиту на сервер для отримання даних про всі пости.

Сервіс постів виглядає наступним чином:

```
const feedPageService : {fetchAllUsersNumber: () => Promise<any>, getBestPosts: (filter: object) => Promise<an..., getData: (filter: object) => Prom
  getData: async (filter: object) : Promise<any>  => api.get( path: '/api/post/all',  config: { params: filter }),
  getHotPosts: async (filter: object) : Promise<any>  => api.get( path: '/api/post/hots',  config: { params: filter }),
  getBestPosts: async (filter: object) : Promise<any>  => api.get( path: '/api/post/bests',  config: { params: filter }),
  likePost: async (post: object) : Promise<any>  => api.put(
    path: '/api/postreaction',
    config: {
      data: post
    }
  ),
  searchPosts: async (filter: object) : Promise<any>  => api.get( path: '/api/search/list',  config: { params: filter }),
  loadCountResults: async (filter: object) : Promise<any>  => api.get( path: '/api/search/count',  config: { params: filter }),
  fetchAllUsersNumber: async () : Promise<any>  => api.get( path: '/api/user/count')
};

export default feedPageService;
```

Рис. 2.22 – Сервіс для відправки запитів для сторінки FeedPage

У ньому нас цікавить перша функція, яка звертається по шляху /api/post/all. Ця функція асинхронна, але нам не потрібно думати про те, як обробити її результат, бо це завдання Saga функції.

Таким чином виглядає Saga для отримання постів:

```
function* fetchData(filter) : Generator<CallEffect<any> | Pu...  {
  try {
    let response;

    switch (filter.payload.filter) {
      case 'hots':
        response = yield call(feedPageService.getHotPosts, filter.payload.params);
        break;
      case 'bests':
        response = yield call(feedPageService.getBestPosts, filter.payload.params);
        break;
      default:
        response = yield call(feedPageService.getData, filter.payload.params);
        break;
    }
    const postsList : {posts: any}  = { posts: response };
    yield put(fetchDataRoutine.success(postsList));
  } catch (error: any) {
    yield put(fetchDataRoutine.failure(error?.message));
    toastr.error( title: 'Error',  message: 'Loading failed!');
  }
}
```

Рис. 2.23 – Saga функція fetchData для отримання постів

Вона приймає стрічку filter, як аргумент. За допомогою нього вирішується до якого сервісу потрібно звернутися: отримання всіх постів або отримання лише найпопулярніших. Після успішного виконання запиту результат знаходиться у

змінній postsList, яка за допомогою команди yield put передається у сховище. Для того, щоб у сховищі змінити саме поле posts, ми використовуємо дію fetchDataRoutine.success(postsList). Ця стрічка коду означає, що зміна postsList має бути передана в редюсер, який відповідає за стан успіху. Якщо ж запит був неуспішний, то викличеться дія fetchDataRoutine.failure(error?.message) і у редьюсер передасться повідомлення про помилку.

Для компоненту FeedPage існуються також інші дії, всі вони визначаються у директорії routines.

```
import { createRoutine } from 'redux-saga-routines';

const createFeedPageRoutine = <T extends unknown>(actionName: string) => createRoutine<T>( typePrefix: `FEED_PAGE:${actionName}`);

export const fetchAllUsersNumberRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'FETCH_ALL_USERS_NUMBER');
export const fetchDataRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'FETCH_DATA');
export const resetDataRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'RESET_DATA');
export const addMorePostsRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'ADD_MORE_POSTS');
export const likePostRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'LIKE_POST_ROUTINE');
export const disLikePostRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'DISLIKE_POST_ROUTINE');
export const searchPostsRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'SEARCH_POSTS');
export const loadCountResultsRoutine : UnifiedRoutine<function(?=): A...  = createFeedPageRoutine( actionName: 'LOAD_COUNT_RESULTS');
```

Рис. 2.24 – Доступні дії для компонента FeedPage

Всі редьюсери компонента знаходяться у директорії reducers. Редьюсер відповідає за те, щоб зберегти дані у сховище і далі передати їх у комопонет представлення. Редьюсер для FeedPage виглядає наступним чином:

```
export interface IFeedPageReducerState {
  posts: IPostFeed[];
  hasMore: boolean;
  loadMore: boolean;
  countResults: number;
  numberOfAllUsers: number;
}

const initialState: IFeedPageReducerState = {
  posts: [],
  countResults: 0,
  hasMore: false,
  loadMore: false,
  numberOfAllUsers: 0
};

export const feedPageReducer : ReducerWithInitialState<IFeedP...  = createReducer(initialState, actionsMap: {
  [fetchDataRoutine.SUCCESS]: (state : WritableDraft<IFeedPageReducer... , { payload }: PayloadAction<IPostList>) : void  => {
    if (!state.loadMore) {
      state.posts = payload.posts;
    } else {
      payload.posts.map(post : IPostFeed  => state.posts.push(post));
    }
    state.hasMore = !isEmptyArray(payload.posts);
  },
  [addMorePostsRoutine.TRIGGER]: (state : WritableDraft<IFeedPageReducer... , { payload }: PayloadAction<boolean>) : void  => {
    state.loadMore = payload;
  },
```

Рис. 2.25 - Об’єкт редьюсер для комопонента FeedPage

Об’єкт initialState визначає початковий стан сховища при першому завантажені сторінки. Редьюсер створюється за допомогою виклику функції createReducer. Вона приймає об’єкт початкового стану та об’єкт, полями якого виступає стан якоїсь дії. Тобто такий код [fetchDataRoutine.SUCCESS] означає: якщо виклик дії був успішним, то виконується функція, яка визначена для цього поля. У випадку зі статтями, вони просто додаються у наявне сховище у поле posts, за допомогою коду *state.posts = payload*.

Після цього, це поле може бути отримане і викликане у комопненті FeedPage.

Всі комопоненти проходять через такий самий цикл обробки, різниця лише у діях та даних, які використовуються.

Весь алгоритм роботи клієнтської частини можна представити як діаграму:

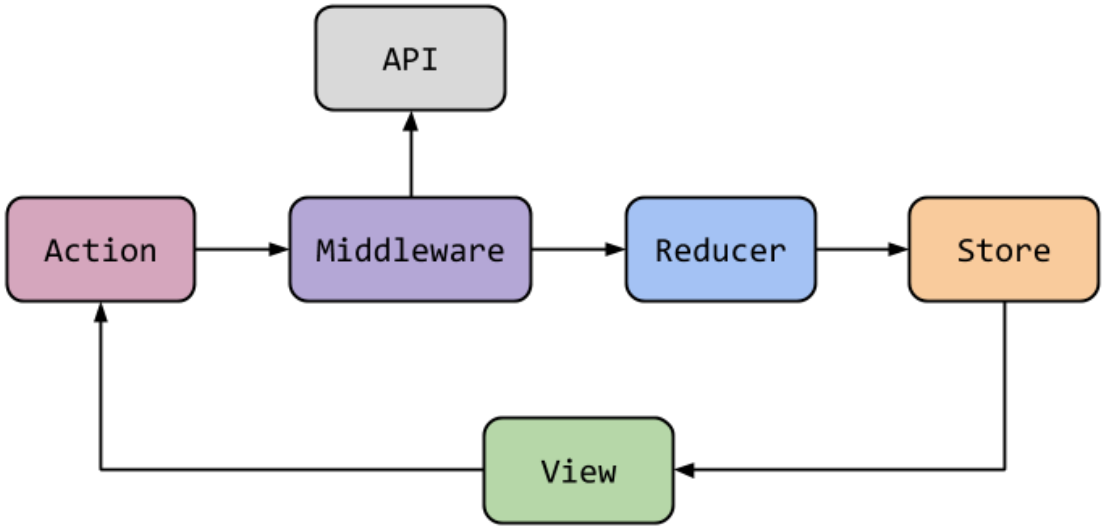


Рис. 2.26 – Алгоритм роботи клієнтської частини

2.5. Обгрунтування та організація вхідних та вихідних даних програми

Веб-додаток отримує вхідні данні с заповнених користувачем форм веб-компонентів, а також з місця розташування даних компонентів.

Вихідні данні представлені в виді:

– Веб-сторінок додатку.
– Інтерактивних компонентів системи.

2.6. Опис роботи розробленої системи

2.6.1. Використані технічні засоби

Так як вся інформація зберігається на стороні клієнта локально, то наступні рекомендовані характеристики обчислювальної машини наведені виключно для клієнтського обладнання:

- Процесор Intel(R) Core(TM) i3-2348M з тактовою частотою 2.3GHz.
- Оперативна пам’ять в розмірі 4ГБ пам’яті.
- Вільного місця на диску в розмірі 1ГБ пам’яті.
- Монітор
- Маніпулятор «миша».
- Клавіатура.
- Доступ до глобальної мережі.

Вище наведені характеристики являють собою рекомендовані. Це означає, що при наявності характеристик не нижче зазначених, розроблений додаток буде функціонувати відповідно до вимог щодо надійності, безпеки та швидкості обробки даних.

2.6.2. Використані програмні засоби

Веб-застосунок реалізований за допомогою мов програмування Typescript та Java з використанням бібліотек React, Redux та Spring Boot.

Необхідні програмні засоби на стороні клієнта:

- Веб браузер Firefox, Google Chrome.
- Операційна система Windows 7/10.

2.6.3. Виклик та завантаження програми

Розроблений додаток використовується онлайн, тому для його експлуатації необхідний веб-браузер з підтримкою JavaScript.

2.6.4. Опис інтерфейсу користувача

Робота з додатком розпочинається з головної сторінки, на якій знаходяться статті, які були створені іншими користувачами.

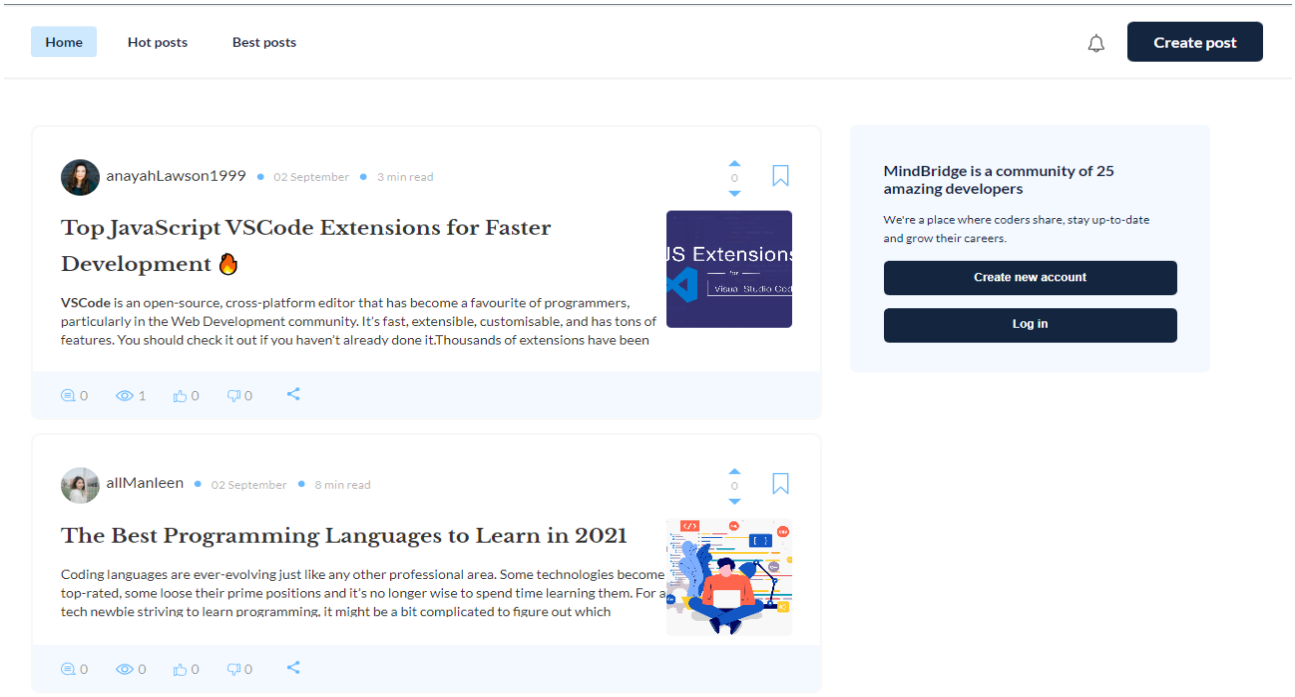


Рис. 2.27 – Головна сторінка додатку незареєстрованого користувача

Не зареєстрований користувач не має можливості створювати статі, додавати реакції до них, для того щоб ця можливість з'явилась потрібно зареєструвати акаунт.

Для того, щоб зареєструватись необхідно натиснути на кнопку «Create new account».

Після цього користувач потрапляє на сторінку реєстрації. Ця сторінка являє собою форму з полями персональної інформації, які обов’язково потрібно заповнити. Кожне поле має валідується, тому поки воно не буде заповнене, реєстрація не може бути виконана.

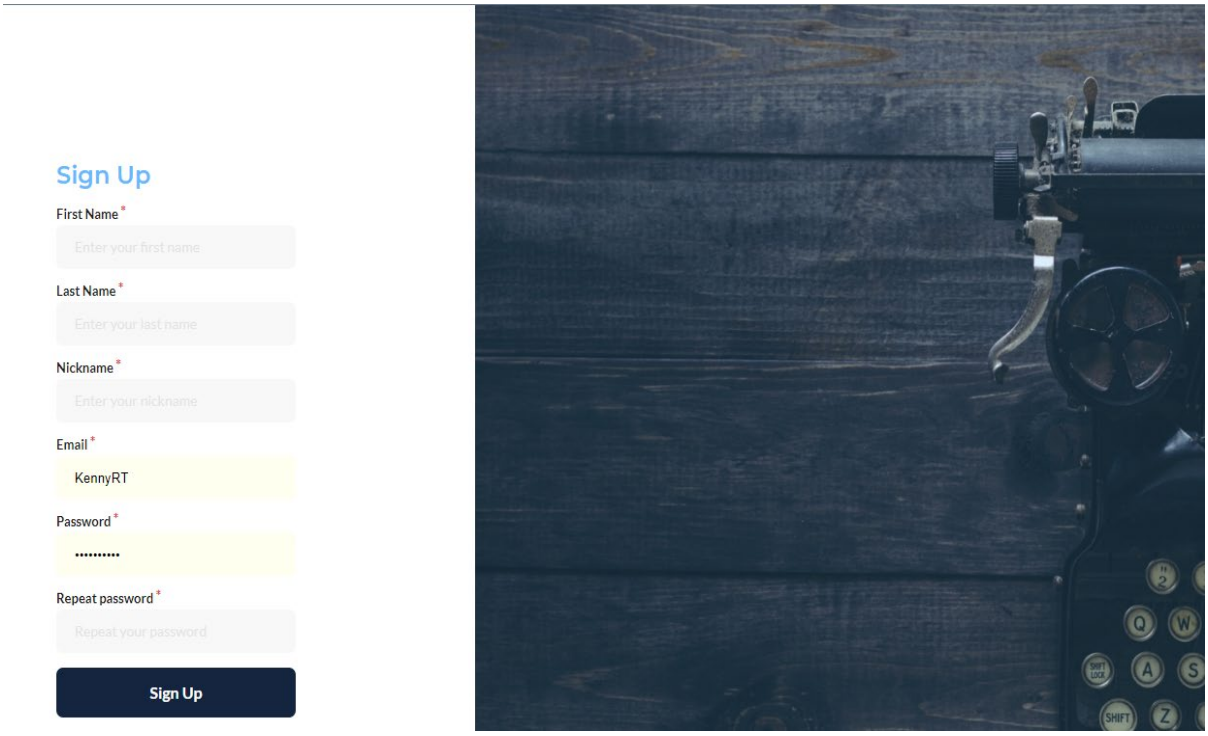


Рис. 2.28 – Сторінка реєстрації

Після успішної реєстрації користувач перенаправляється на головну сторінку. Як можна бачити, бокове меню тепер містить посилання на доступні сторінки.

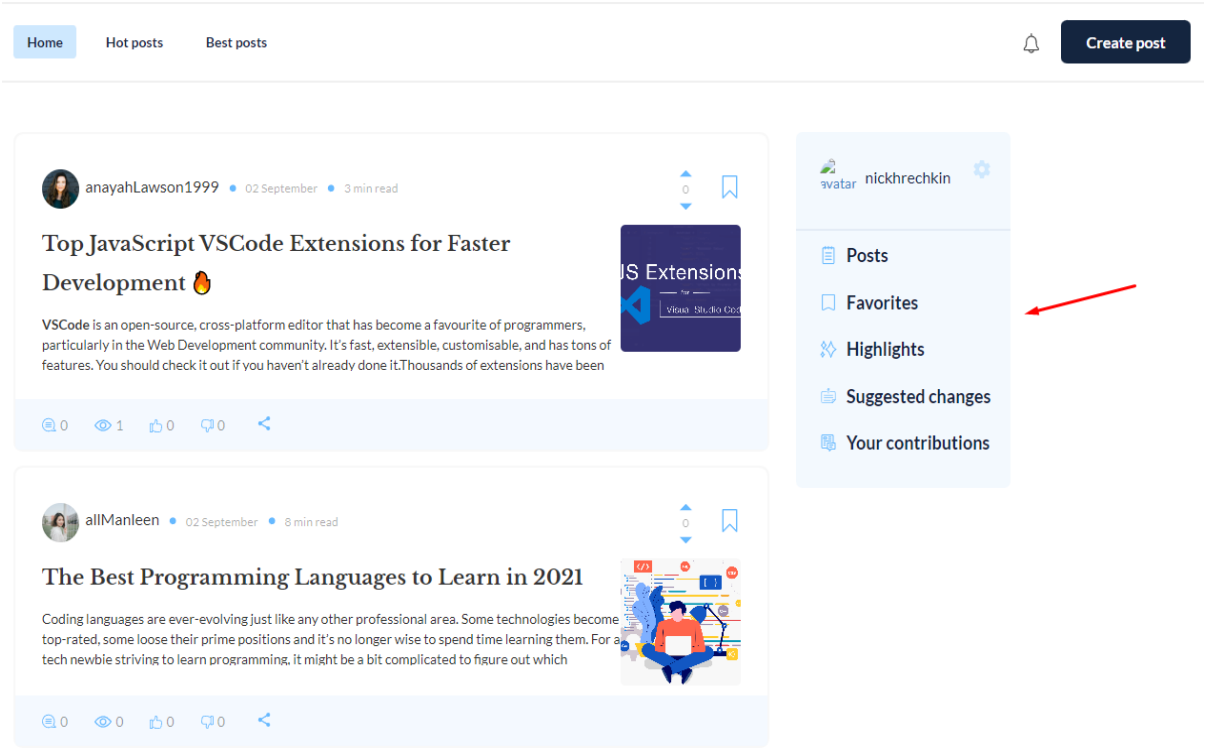


Рис. 2.29 – Головна сторінка додатку зареєстрованого користувача

Дані новоствореного акаунту можна змінити. Для цього потрібно натиснути на іконку шестерні, яка знаходиться біля імені та обрати опцію «Edit profile». При натисканні кнопки «Sign out» буде виконаний вихід із додатку.

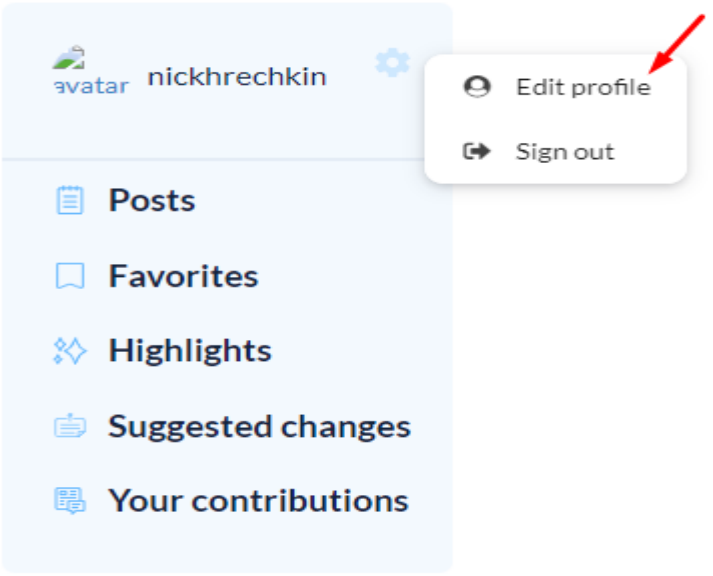


Рис. 2.30 – Меню для зміни інформаціх про користувача або виходу з додатку

Сторінка для зміни інформаціх про користувача виглядає наступним чином:

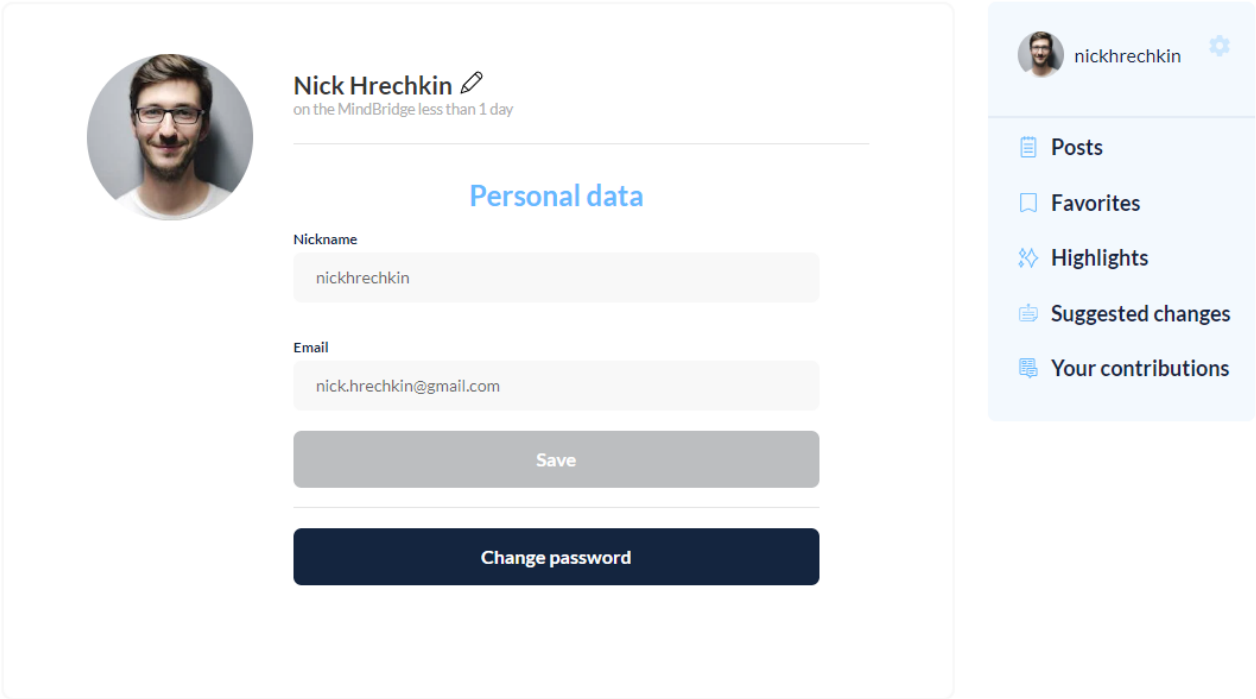


Рис. 2.31 – Сторінка зміни даних про користувача

На ній можна змінити ім’я, нікнейм та пошту.

Після реєстрації користувач має можливість стати автором статті. Для того щоб це зробити, натискаємо кнопку «Create post».

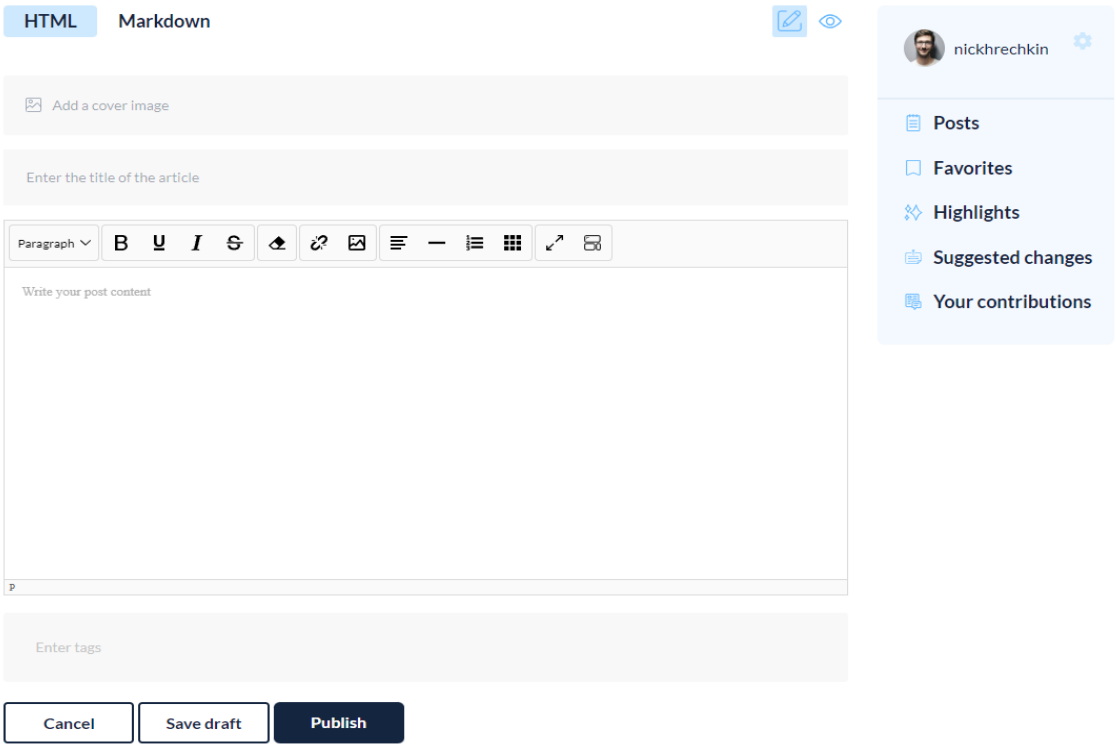


Рис. 2.32 – Сторінка створення статті

Користувачу доступні два варіанти як можна стилізувати статтю. Можна використати HTML розмітку або Markdown. Створимо її з використанням Markdown.

Так виглядає текст статті:

Рис. 2.33 – Сторінка створення статті з заповненими полями

Можна скористатися кнопкою «Show preview» і побачити який вигляд буде мати стаття після публікації.

Рис. 2.34 – Сторінка створення статті в режимі попереднього перегляду

Додаток підтримує можливість створення статті як чорновика – це стан у якому стаття буде збережена, але не опублікована. Зазвичай це робиться у випадках коли автор не має змогу завершити статтю до кінця і хоче повернутися до її написання через деякий час.

Всі опубліковані статті та чорновики можна знайти перейшовни за посиланням «Posts» у боковому меню.

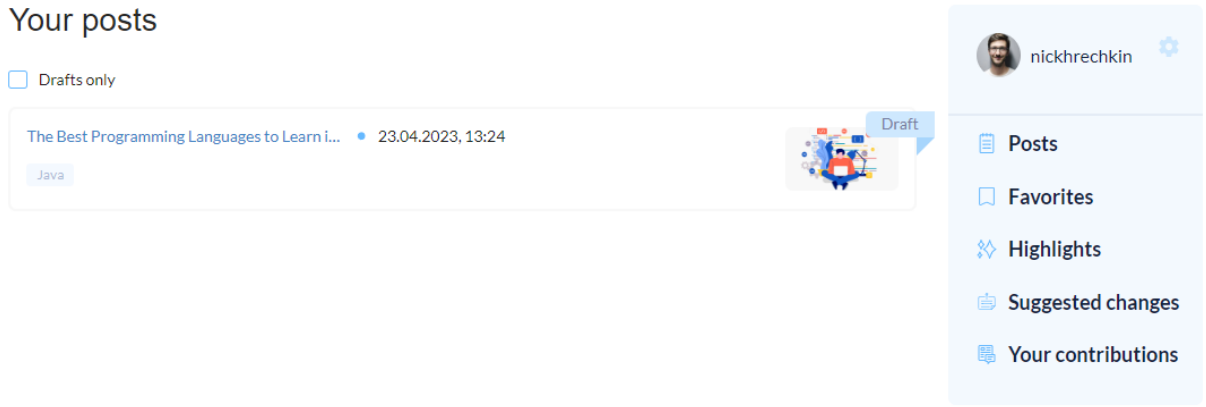


Рис. 2.35 – Сторінка де відображені всі опубліковані статті та чорновики

Якщо опублікованих статей багато, то можна скористатися кнопкою фільтрації і відображати лише статті, що є чорновиками.

Щоб переглянути статтю, потрібно натиснути на її назву і користувач буде перенаправленний на її сторінку.

Рис. 2.36 – Сторінка перегляду статті

Зліва присутні кнопки за допомогою яких можна оцінити статтю, додати її до улюблених, поділитися статею або редагувати. Якщо це чорновик, то кнопки додати в улюблені та поділитися вимкнені. Щоб опублікувати статтю потрібно перейти у меню редагування та натиснути кнопку «Опублікувати».

Після публікації можна перейти на головну сторінку і побачити там новостворену статтю.

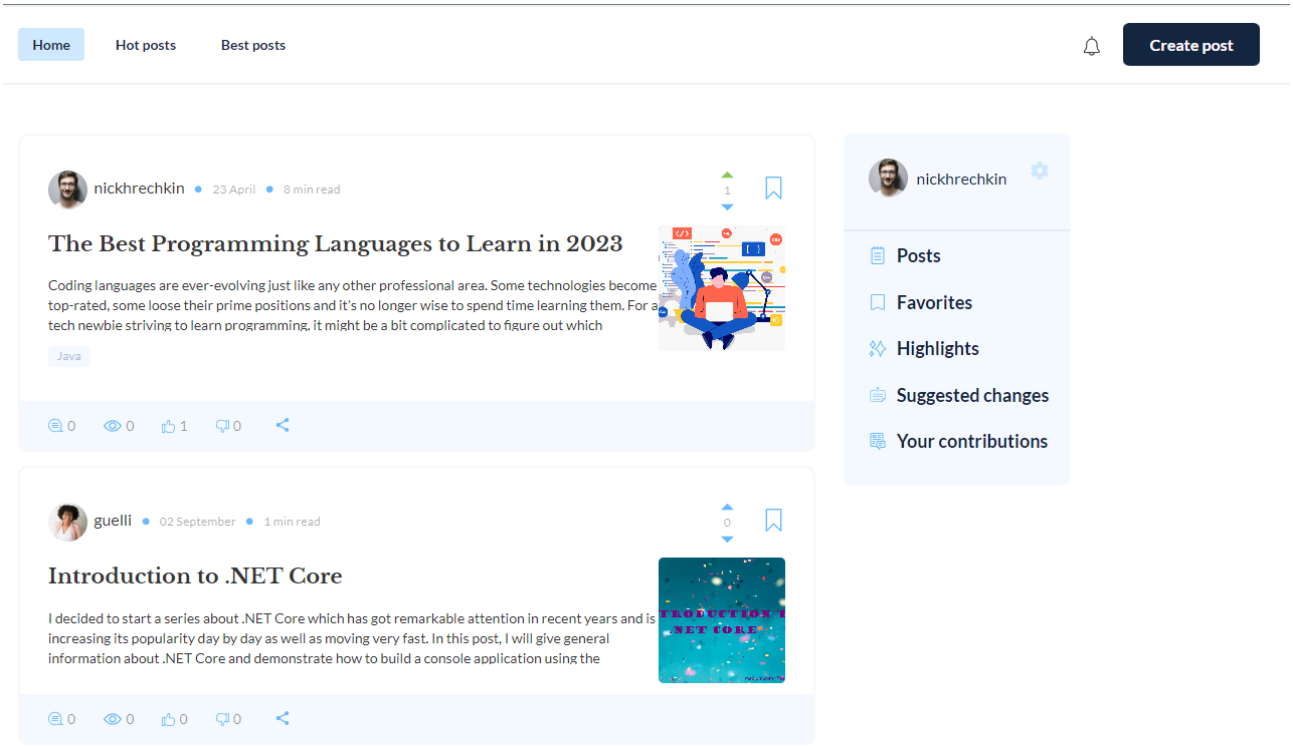


Рис. 2.37 – Головна сторінка додатку с новостворено статтею

Ще однію дуже важливою функцією в додатку є збереження цитат. Дуже часто траляються випадки, коли користувач читає статтю і знаходить якусь її частину дуже цікавою і хоче зберегти. Звісно він або вона може скопіювати її і збрегти у текстовий документ на комп'ютері або записати її ручкою чи олівцем але це незручно.

Тому зберегти цитати можна невиходячи з додатку, для цього потрібно виділити якусь частину тексту, з'явиться спливаюче меню на яке потрібно натиснути.

Рис. 2.38 – Процес збереження цитати із статті

Після цього цитата буде збережена і її можна переглянути на окремій сторінці.

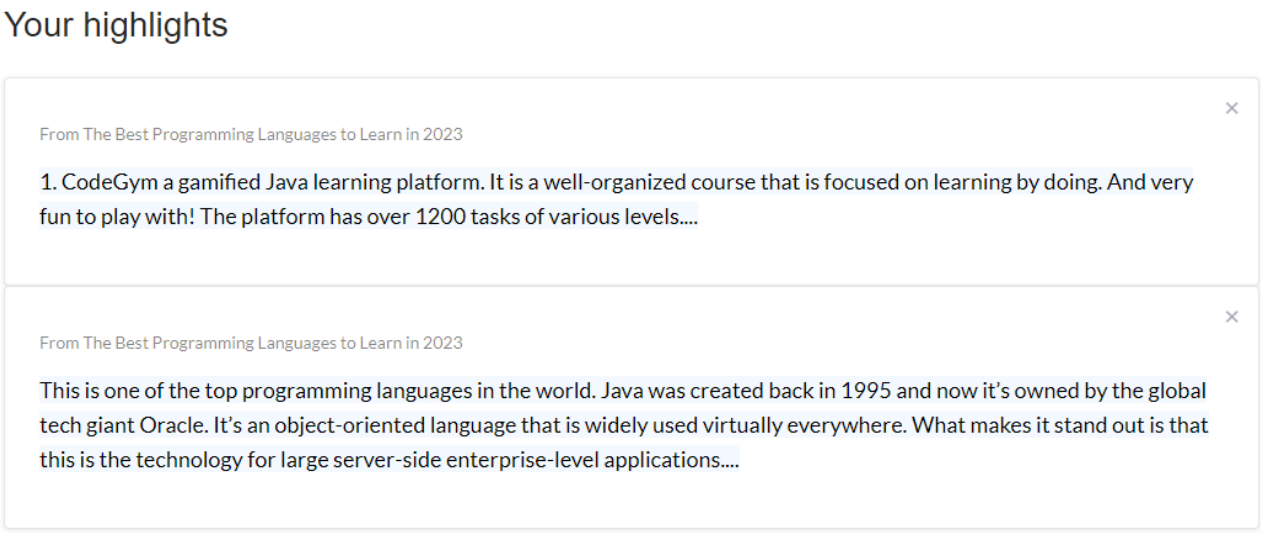


Рис. 2.39 – Сторінка збережених цитат

Якщо натиснути на блок з цитатою, то користувач буде перенаправлений на сторінку статті.

Останні дві функції, які підтримує додаток – реакції та додавання в обране.

Якщо натиснути на стрілочки, що знаходяться у правому верхньому кутку блока статті, можна залишити негативну або позитивну реакцію.

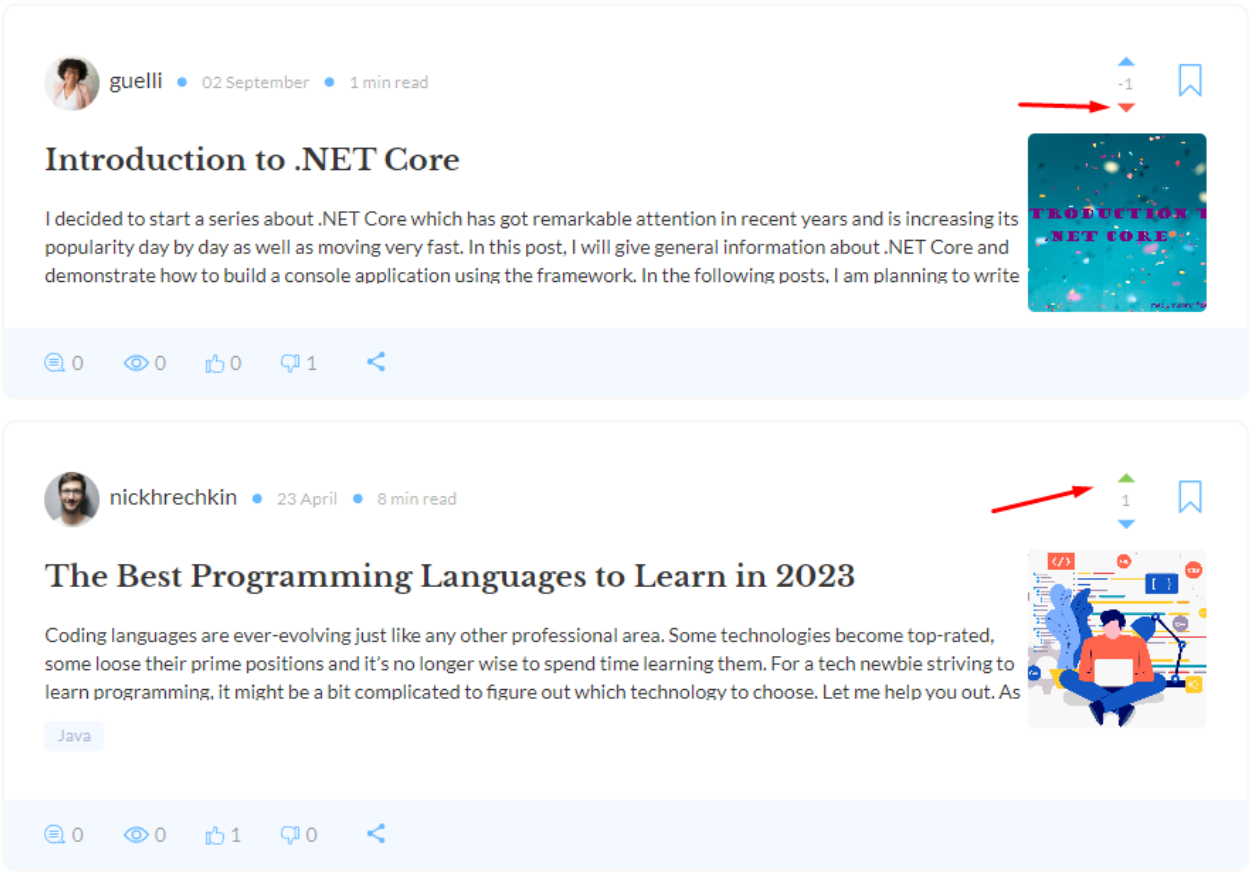


Рис. 2.40 – Головна сторінка додатку с новоствореною статтею

Якщо ж натиснути на іконку справа від реакцій, то стаття буде додана в обране.

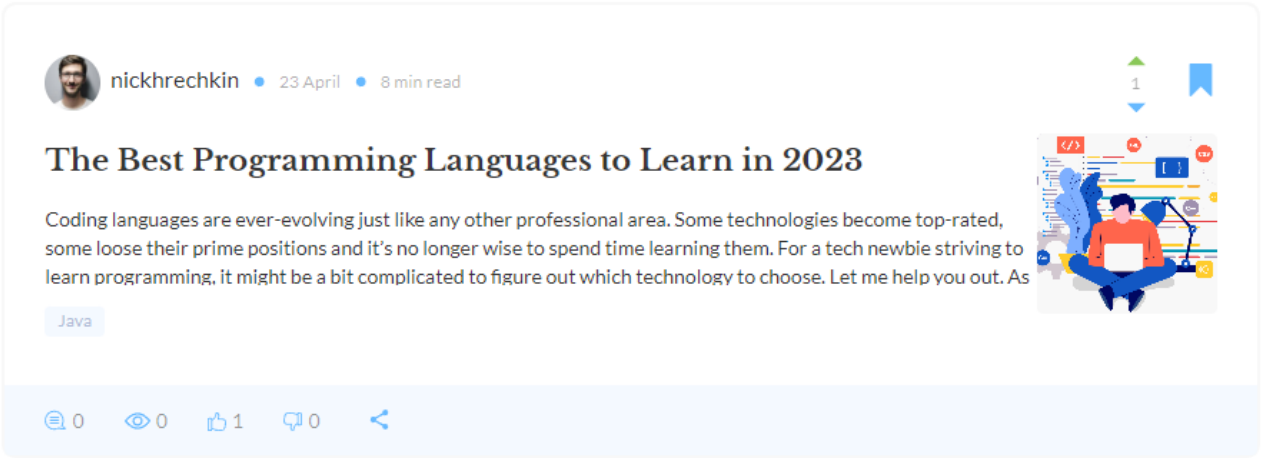


Рис. 2.41 – Додавання статті в обране

Обрані статті можна переглянути на сторінці «Favourites».

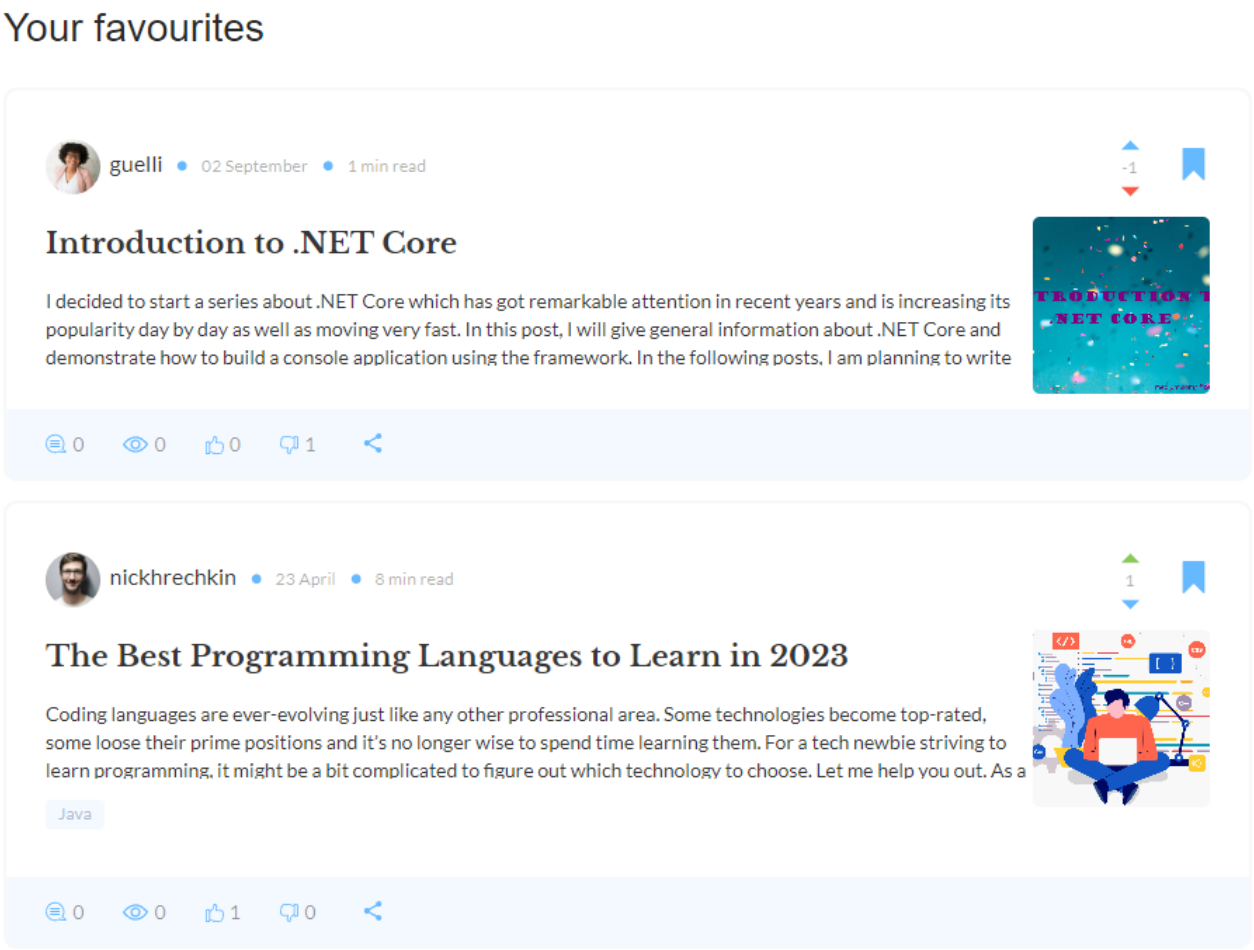


Рис. 2.42 – Сторінка з обраними статтями

РОЗДІЛ 3. ЕКОНОМІЧНА ЧАСТИНА

Під час розробки програмного забезпечення важливими етапами є визначення трудомісткості розробки і розрахунок витрат на створення програмного продукту.

3.1. Визначення трудомісткості розробки програмного забезпечення

Задані дані:

1. передбачуване число операторів – 900;
2. коефіцієнт складності програми – 2;
3. коефіцієнт корекції програми в ході її розробки – 0.15;
4. годинна заробітна плата програміста, грн/год – 101;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1.3;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,2;
7. вартість машино-години ЕОМ, грн/год – 20.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

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

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

t_u - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

t_n - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

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

де q – передбачуване число операторів;

C – коефіцієнт складності програми;

p – коефіцієнт корекції програми в ході її розробки.

$$Q = 900 \cdot 2 \cdot (1 + 0.15) = 2070$$

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

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

де B – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k – коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_u = \frac{2070 \cdot 1{,}3}{75 \cdot 1} = 33{,}6 \text{ людино-годин}$$

Витрати праці на розробку алгоритму рішення задачі:

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

$$t_a = \frac{2070}{20 \cdot 1} = 103{,}5 \text{ людино-годин}$$

Витрати на складання програми по готовій блок-схемі:

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

$$t_n = \frac{2070}{25 \cdot 1} = 82 \text{ людино-годин}$$

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

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

$$t_{отл} = \frac{2070}{5 \cdot 1} = 414 \text{ людино-годин}$$

– за умови комплексного налагодження завдання:

$$t^k_{отл} = 1{,}5 \cdot t_{отл}, \text{ людино-годин,} \quad (3.7)$$

$$t^k_{отл} = 1{,}5 \cdot 414 = 621 \text{ людино-годин.}$$

Витрати праці на підготовку документації:

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

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису;

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

$$t_{др} = \frac{2070}{20 \cdot 1} = 103{,}5 \text{ людино-годин.}$$

$t_{до}$ – трудомісткість редагування, печатки й оформлення документації:

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

$$t_{до} = 0{,}75 \cdot 103.5 = 77{,}625 \text{ людино-годин,}$$

$$t_{д} = 103.5 + 77{,}625 = 181 \text{ людино-годин.}$$

Тепер розрахуємо трудомісткість ПЗ:

$$t = 77{,}625 + 33{,}6 + 103{,}5 + 82 + 414 + 181 = 891{,}725 \text{ людино-годин.}$$

3.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ *Кпо* включають витрати на заробітну плату виконавця програми *Зз/п* і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$К_{по} = З_{зп} + З_{мв}, \text{ грн.} \quad (3.11)$$

Заробітна плата виконавців визначається за формулою:

$$З_{зп} = t \cdot C_{пр}, \text{ грн}, \quad (3.12)$$

де t – загальна трудомісткість, людино-годин;

$C_{пр}$ – середня годинна заробітна плата програміста, грн/година.

$$З_{зп} = 891{,}725 \cdot 101 = 90064, \text{ грн.}$$

Вартість машинного часу, необхідного для налагодження програми:

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

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

$C_{мч}$ – вартість машино-години ЕОМ, грн/год,

$C_{мч} = 15$, грн/год.

$$З_{мв} = 414 \cdot 20 = 8280, \text{ грн.}$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення ПЗ:

$$К_{по} = 49900 + 5197 = 98344, \text{ грн.}$$

Очікуваний період створення ПЗ:

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

де B_k – число виконавців (приймається 1),

F_p – місячний фонд робочого часу (40 годин на тиждень $F_p = 176$ годин).

$$T = \frac{891.725}{1 \cdot 176} = 5 \text{ міс.}$$

ВИСНОВКИ

У даній дипломній роботі була реалізована веб-платформа для соціальної журналістики, спрямованої на розробників програмного забезпечення. В роботі було використано такі технології та інструменти, як Spring Boot, Redux, React, Java та TypeScript.

Основні функції платформи включають публікацію статей, додавання статей в обране, можливість залишати реакції на статті та зберігати цитати зі статей. Ці функції сприяють активній взаємодії розробників, обміну знаннями та досвідом, а також створенню спільноти, яка сприяє взаємному розвитку.

В результаті дослідження та розробки було створено веб-платформу, яка задовольняє потреби розробників програмного забезпечення у спільному навчанні, обміні ідеями та публікації статей. Використанні технології дозволяють забезпечити ефективну та зручну роботу з платформою.

Платформа може бути використана як інструмент для саморозвитку розробників, покращення навичок програмування та обміну цінними знаннями. Додатково, вона створює можливість для збереження цитат зі статей, що допомагає розробникам зосередитися на найбільш цікавих та важливих аспектах.

Узагальнюючи, розроблена веб-платформа відповідає потребам розробників програмного забезпечення у соціальній журналістиці, дозволяючи їм спілкуватися, публікувати статті та підвищувати свою професійну компетентність.

Застосування бібліотек JavaScript зекономило довільну кількість часу та грошей. Тому час створення застосунку становить 5 місяців, а його оціночна вартість 98 тис. грн.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. І. Підгайний, В. Штагера Як приборкати Medium: покрокові поради 5с.
2. Сайт веб-платформи Medium - https://medium.com/.
3. Сайт веб-платформи Dev.to - https://dev.to/.
4. Шкарупа А. О., Стрельцов О.А. Аналіз особливостей сучасних веб-додатків 6с.
5. Є.Є. Малохвій, В.С. Бугай, Г.І. Молчанов, О.П. Черних Аналітичний огляд порівняння сучасних JavaScript рішень для роробки веб-додатків, 2021 58с.
6. The State of JavaScript. Front-end Frameworks Overview - https://2022.stateofjs.com/en-US/libraries/front-end-frameworks/ .
7. Документація по TypeScript - https://www.typescriptlang.org.
8. Гутьєррес Ф. Spring Boot 5: кращі практики для професіоналів, 2020 464с.
9. Марк Тілєнс Томас React in Action, 2018 360с.
10. Документація React - https://react.dev/.
11. Документація Redux - https://react.dev/.
12. Роберт С. Мартін Чиста Архітектура, 2019 416с.
13. Ерік Фрімен Патерни проєктування, 2018 672с.
14. М.Ю. Швець, Д.С. Заруба, Ю.В. Хохлов Порівняння SQL та NOSQL баз даних 5с.
15. М. Клеппман Високонавантажені додатки. Програмування, масштабування, підтримка. 640с.
16. Документація Postgresql - https://www.postgresql.org/.
17. Н. Є. Мамалига, А.І. Катаєва Система керування базами даних в сучасних умовах ІТ-індустрії 5с.
18. Д. Б. Буй, А.В. Пузікова Теорія нормалізації в реляційних базах даних: сучасний стан 10с.
19. М. В. Ліпчанський, О.О. Ільяшенко Порівняння підходів Code First та Design First в розробці API 4с.
20. Адама Бодуч Flux Architecture, 2016 352с.

ДОДАТОК А

КОД ПРОГРАМИ

Лістинг файлу screens/FeedPage/index.tsx

```
import React, { useEffect, useState } from 'react';
import { connect } from 'react-redux';
import styles from './styles.module.scss';
import PostCard from '@components/PostCard';
import { IBindingAction, IBindingCallback1 } from '@models/Callbacks';
import InfiniteScroll from 'react-infinite-scroll-component';
import { RootState } from '@root/store';
import { extractFetchDataLoading, extractSearchPostsLoading } from '@screens/FeedPage/reducers';
import {
  addMorePostsRoutine,
  fetchDataRoutine,
  likePostRoutine,
  loadCountResultsRoutine,
  searchPostsRoutine
} from '@screens/FeedPage/routines';
import { IPostFeed } from '@screens/FeedPage/models/IPostFeed';
import { ICurrentUser } from '@screens/Login/models/ICurrentUser';
import { loadCurrentUserRoutine } from '@screens/Login/routines';
import { useHistory } from 'react-router-dom';

import { fetchTagsRoutine, fetchUserProfileRoutine } from '@screens/PostPage/routines';
import { IUserProfile } from '@screens/PostPage/models/IUserProfile';
import { deleteFavouritePostRoutine, saveFavouritePostRoutine } from '@screens/FavouritesPage/routines';
import { useLocation } from 'react-use';
import NoResultsSvg from '@components/svgs/NoResultsSvg';
import SearchSvg from '@components/Header/svg/searchSvg';
import FoundPostsList from '@components/FoundPostsList';
import { useDebouncedCallback } from 'use-debounce';
import { IPost } from '@screens/Header/models/IPost';

import { searchPostsByElasticRoutine } from '@screens/Header/routines';
import { fetchUserRoutine } from '@screens/ProfilePage/routines';

export interface IFeedPageProps extends IState, IActions {
  isAuthorized: boolean;
  currentUser: ICurrentUser;
  userInfo: IUserProfile;
}

interface IState {
  data: IPostFeed[];
  dataLoading: boolean;
  hasMore: boolean;
  loadMore: boolean;
  searchPosts: IPost[];
  countResults: number;
  allTags: any[];
}

interface IActions {
  fetchData: IBindingCallback1<object>;
  likePost: IBindingCallback1<object>;
  fetchUserProfile: IBindingCallback1<string>;
  setLoadMorePosts: IBindingCallback1<boolean>;
```

```
  searchTitlesByElastic: IBindingCallback1<string>;
  loadUser: IBindingAction;
  saveFavouritePost: IBindingCallback1<object>;
  deleteFavouritePost: IBindingCallback1<object>;
  searchPostsByElastic: IBindingCallback1<object>;
  loadCountResults: IBindingCallback1<object>;
  fetchTags: IBindingAction;
  fetchUserData: IBindingCallback1<string>;
}

const params = {
  from: 0,
  count: 10,
  userId: ''
};

const ENTER_CHAR_CODE = 13;
const LOADING_PLACEHOLDERS = [...Array(3)];

const FeedPage: React.FC<IFeedPageProps> = (
  {
    data,
    fetchData,
    dataLoading,
    hasMore,
    setLoadMorePosts,
    loadMore,
    currentUser,
    userInfo,
    likePost,
    searchTitlesByElastic,
    countResults,
    searchPostsByElastic,
    searchPosts,
    loadCountResults,
    saveFavouritePost,
    deleteFavouritePost,
    fetchTags,
    allTags
  }
) => {
  const location = useLocation();
  const history = useHistory();
  const [isSearch, setIsSearch] = useState(false);
  const [isSearchInputFilled, setIsSearchInputFilled] = useState(false);
  const [elasticContent, setElasticContent] = useState('');
  const [tagsContent, setTagsContent] = useState([]);

  const filter = location.pathname.substring(1, location.pathname.length);

  useEffect(() => {
    fetchTags();
  }, [fetchTags]);

  useEffect(() => {
    params.from = 0;
    setLoadMorePosts(false);
    window.scrollTo(0, 0);
    const regex = /\\?tags=(.*)&query=(.*)/;

    if (regex.exec(location.search)) {
      const urlParams = new URLSearchParams(location.search);
```

```
    const query = urlParams.get('query');
    const tags = urlParams.get('tags');

    if (query === '' && tags === '') {
      history.push('/');
      return;
    }
    setIsSearch(true);
    setElasticContent(query);

    if (allTags.length !== 0 && tags !== '') {
      setTagsContent(tags.split(','));
    }

    loadCountResults({
      query,
      tags
    });
  } else {
    if (currentUser) {
      fetchData({
        from: 0,
        count: 10,
        userId: currentUser.id,
        filter
      });
    } else {
      fetchData({
        params,
        filter
      });
    }
    setIsSearch(false);
  }
}, [fetchData, location, currentUser, allTags]);

const handleLoadMorePosts = filtersPayload => {
  fetchData({
    params: filtersPayload,
    filter
  });
  setLoadMorePosts(true);
};

const handleSearchMorePosts = filtersPayload => {
  searchPostsByElastic({
    query: elasticContent,
    tags: tagsContent.toString(),
    params: filtersPayload
  });
};

const debounced = useDebouncedCallback(value => {
  searchTitlesByElastic(value);
}, 400);

const handleLikePost = postId => {
  if (currentUser.id) {
    const post = {
      postId,
      userId: currentUser.id,
      liked: true
    };
```

```
    likePost(post);
  } else {
    history.push('/login');
  }
};

const handleFavouriteAction = post => {
  if (!currentUser?.id) {
    history.push('/login');
    return;
  }
  if (!post.isFavourite) {
    saveFavouritePost({
      userId: currentUser.id,
      postId: post.id
    });
  } else {
    deleteFavouritePost({
      userId: currentUser.id,
      postId: post.id
    });
  }
};

const handleDisLikePost = postId => {
  if (currentUser.id) {
    const post = {
      postId,
      userId: currentUser.id,
      liked: false
    };
    likePost(post);
  } else {
    history.push('/login');
  }
};

const getMorePosts = () => {
  setLoadMorePosts(true);
  const {
    from,
    count
  } = params;
  params.from = from + count;
  if (isSearch) {
    handleSearchMorePosts(params);
  } else {
    handleLoadMorePosts(params);
  }
};

const handleLinkClick = () => {
  setIsSearchInputFilled(false);
  searchTitlesByElastic('');
  setElasticContent('');
};

const handleInputContent = (event: any) => {
  debounced(event.target.value);
  setElasticContent(event.target.value);
  if (event.target.value) {
    setIsSearchInputFilled(true);
  } else {
```

```
    setIsSearchInputFilled(false);
  }
};

const goToSearchPage = () => {
  setIsSearchInputFilled(false);
  history.push(`/search?tags=${tagsContent}&query=${elasticContent}`);
};

const handleBlur = (event: any) => {
  if (!event.relatedTarget) {
    setIsSearchInputFilled(false);
  }
};

if (dataLoading && !loadMore) {
  return (
    <div className={styles.feedPage}>
      <div className={styles.main}>
        {LOADING_PLACEHOLDERS.map(() => (
          <PostCard
            dataLoading={dataLoading}
            handleLikePost={handleLikePost}
            handleDisLikePost={handleDisLikePost}
            handleFavouriteAction={handleFavouriteAction}
            post={data[0]}
            userInfo={userInfo}
          />
        ))}
      </div>
    </div>
  );
}

const handleEnterDown = (event: any) => {
  if (event.keyCode === ENTER_CHAR_CODE) {
    goToSearchPage();
  }
};

return (
  <div className={styles.feedPage}>
    <div className={styles.searchTitle}>
      {isSearch && (
        <div className={styles.search_input} onBlur={handleBlur}>
          <input
            type="text"
            placeholder="Search..."
            onChange={handleInputContent}
            value={elasticContent}
            onKeyDown={handleEnterDown}
          />
          {isSearchInputFilled
          && <button type="button" className={styles.close_image} onClick={handleLinkClick}>✖</button>}
          <button type="button" onClick={goToSearchPage}>
            <SearchSvg />
          </button>
          {isSearchInputFilled
          && (
            <div className={styles.foundPosts}>
              <ul>
                {searchPosts[0]
                && searchPosts.map(post => (
```

```
                <FoundPostsList
                  linkClick={handleLinkClick}
                  key={post.sourceId}
                  post={post}
                />
              ))}
            </ul>
          </div>
        )}
        {isSearch && data && (
          <div className={styles.requestInfo}>
            <h4>
              {'Found '}
              <span className={styles.countPosts}>{countResults}</span>
              {countResults === 1 ? ' article.' : ' articles.'}
            </h4>
          </div>
        )}
      </div>
    )}
  </div>

  <div className={styles.main}>
    <InfiniteScroll
      style={{ overflow: 'none' }}
      dataLength={data.length}
      next={getMorePosts}
      hasMore={hasMore}
      loader={' '}
      scrollThreshold={0.9}
    >
      {data.length !== 0 ? (
        data.map(post => (
          <PostCard
            dataLoading={dataLoading && !loadMore}
            key={post.id}
            handleLikePost={handleLikePost}
            handleDisLikePost={handleDisLikePost}
            handleFavouriteAction={handleFavouriteAction}
            post={post}
            userInfo={userInfo}
          />
        ))
      ) : (
        <div className={styles.emptyList}>
          <NoResultsSvg width="35%" height="35%" />
          <p>
            No results were found for your request
          </p>
        </div>
      )}
    </InfiniteScroll>
  </div>
 </div>
 );
};

const mapStateToProps: (state: RootState) => IState = state => ({
  data: state.feedPageReducer.data.posts,
  dataLoading: extractFetchDataLoading(state) || extractSearchPostsLoading(state),
  hasMore: state.feedPageReducer.data.hasMore,
  countResults: state.feedPageReducer.data.countResults,
  loadMore: state.feedPageReducer.data.loadMore,
```

```
  isAuthorized: state.auth.auth.isAuthorized,
  currentUser: state.auth.auth.user,
  userInfo: state.postPageReducer.data.profile,
  searchPosts: state.headerReducer.data.posts,
  allTags: state.postPageReducer.data.allTags
});

const mapDispatchToProps: IActions = {
  fetchData: fetchDataRoutine,
  setLoadMorePosts: addMorePostsRoutine,
  fetchUserProfile: fetchUserProfileRoutine,
  likePost: likePostRoutine,
  loadUser: loadCurrentUserRoutine,
  saveFavouritePost: saveFavouritePostRoutine,
  deleteFavouritePost: deleteFavouritePostRoutine,
  searchPostsByElastic: searchPostsRoutine,
  searchTitlesByElastic: searchPostsByElasticRoutine,
  loadCountResults: loadCountResultsRoutine,
  fetchTags: fetchTagsRoutine,
  fetchUserData: fetchUserRoutine
};

export default connect(mapStateToProps, mapDispatchToProps)(FeedPage);
```

Лістинг файлу screens/FeedPage/reducer.ts

```
import { createReducer, PayloadAction } from '@reduxjs/toolkit';
import {
  addMorePostsRoutine,
  fetchAllUsersNumberRoutine,
  fetchDataRoutine,
  likePostRoutine,
  loadCountResultsRoutine, resetDataRoutine,
  searchPostsRoutine
} from '@screens/FeedPage/routines';
import { IPostFeed } from '@screens/FeedPage/models/IPostFeed';
import { IPostList } from '@screens/FeedPage/models/IPostList';
import { isEmptyArray } from 'formik';
import { deleteFavouritePostRoutine, saveFavouritePostRoutine } from '@screens/FavouritesPage/routines';

export interface IFeedPageReducerState {
  posts: IPostFeed[];
  hasMore: boolean;
  loadMore: boolean;
  countResults: number;
  numberOfAllUsers: number;
}

const initialState: IFeedPageReducerState = {
  posts: [],
  countResults: 0,
  hasMore: false,
  loadMore: false,
  numberOfAllUsers: 0
};

export const feedPageReducer = createReducer(initialState, {
  [fetchDataRoutine.SUCCESS]: (state, { payload }: PayloadAction<IPostList>) => {
    if (!state.loadMore) {
      state.posts = payload.posts;
    } else {
      payload.posts.map(post => state.posts.push(post));
    }
```

```
  state.hasMore = !isEmptyArray(payload.posts);
},
[addMorePostsRoutine.TRIGGER]: (state, { payload }: PayloadAction<boolean>) => {
  state.loadMore = payload;
},
[searchPostsRoutine.SUCCESS]: (state, { payload }: PayloadAction<IPostList>) => {
  if (!state.loadMore) {
    state.posts = payload.posts;
  } else {
    payload.posts.map(post => state.posts.push(post));
  }
  state.hasMore = !isEmptyArray(payload.posts);
},
[loadCountResultsRoutine.SUCCESS]: (state, { payload }: PayloadAction<number>) => {
  state.countResults = payload;
},
[likePostRoutine.SUCCESS]: (state, action) => {
  const { response, postId, reactionStatus } = action.payload;
  const post = state.posts.find(p => p.id === postId);
  if (reactionStatus === true) {
    if (response === null || response.isFirstReaction === true) {
      post.likesCount += action.payload.difference;
      post.postRating += action.payload.difference;
      post.reacted = action.payload.difference === 1;
      post.isLiked = action.payload.difference === 1;
    } else {
      post.disLikesCount -= action.payload.difference;
      post.postRating += action.payload.difference;
      post.postRating += action.payload.difference;
      post.likesCount += action.payload.difference;
      post.isLiked = true;
    }
  } else if (response === null || response.isFirstReaction === true) {
    post.disLikesCount += action.payload.difference;
    post.postRating -= action.payload.difference;
    post.reacted = action.payload.difference === 1;
    post.isLiked = action.payload.difference !== 1;
  } else {
    post.likesCount -= action.payload.difference;
    post.postRating -= action.payload.difference;
    post.disLikesCount += action.payload.difference;
    post.postRating -= action.payload.difference;
    post.isLiked = false;
  }
},
[resetDataRoutine.TRIGGER]: state => {
  state.posts = initialState.posts;
  state.hasMore = false;
  state.loadMore = false;
  state.countResults = 0;
},
[saveFavouritePostRoutine.SUCCESS]: (state, action) => {
  if (state.posts) {
    state.posts.find(post => post.id === action.payload).isFavourite = true;
  }
},
[deleteFavouritePostRoutine.TRIGGER]: (state, action) => {
  if (state.posts) {
    const favorite = state.posts.find(post => post.id === action.payload.postId);
    if (favorite) {
      favorite.isFavourite = false;
    }
  }
```

```
  },
  [fetchAllUsersNumberRoutine.SUCCESS]: (state, action) => {
    state.numberOfAllUsers = action.payload;
  }
});
```

Лістинг файлу screens/FeedPage/sagas.ts

```
import { all, call, put, takeEvery } from 'redux-saga/effects';
import feedPageService from '@screens/FeedPage/services/feedPage';
import { toastr } from 'react-redux-toastr';
import {
  disLikePostRoutine,
  fetchAllUsersNumberRoutine,
  fetchDataRoutine,
  likePostRoutine,
  loadCountResultsRoutine,
  searchPostsRoutine
} from '@screens/FeedPage/routines';
import { Routine } from 'redux-saga-routines';

function* fetchData(filter) {
  try {
    let response;

    switch (filter.payload.filter) {
      case 'hots':
        response = yield call(feedPageService.getHotPosts, filter.payload.params);
        break;
      case 'bests':
        response = yield call(feedPageService.getBestPosts, filter.payload.params);
        break;
      default:
        response = yield call(feedPageService.getData, filter.payload.params);
        break;
    }
    const postsList = { posts: response };
    yield put(fetchDataRoutine.success(postsList));
  } catch (error: any) {
    yield put(fetchDataRoutine.failure(error?.message));
    toastr.error('Error', 'Loading failed!');
  }
}

function* likePost(action) {
  try {
    const response = yield call(feedPageService.likePost, action.payload);
    const postReaction = {
      response,
      difference: response?.id ? 1 : -1,
      postId: action.payload.postId,
      reactionStatus: action.payload.liked
    };
    yield put(likePostRoutine.success(postReaction));
  } catch (error: any) {
    yield put(likePostRoutine.failure(error?.message));
    toastr.error('Error', 'Like post failed');
  }
}

function* disLikePost(action) {
```

```
  try {
    const response = yield call(feedPageService.likePost, action.payload);
    const post = {
      response,
      disLikeQuantity: response?.id ? 1 : -1,
      postId: action.payload.postId,
      reactionStatus: action.payload.liked
    };
    yield put(disLikePostRoutine.success(post));
  } catch (error: any) {
    yield put(disLikePostRoutine.failure(error?.message));
    toastr.error('Error', 'Dislike post failed');
  }
}

function* searchPosts({ payload }: Routine<any>) {
  try {
    const response = yield call(feedPageService.searchPosts,
      { query: payload.query, tags: payload.tags, from: payload.params.from, count: payload.params.count });
    yield put(searchPostsRoutine.success({ posts: response }));
  } catch (error: any) {
    yield put(searchPostsRoutine.failure(error?.message));
    toastr.error('Error', 'Search posts failed');
  }
}

function* loadCountResults({ payload }: Routine<any>) {
  try {
    const response = yield call(feedPageService.loadCountResults, { query: payload.query, tags: payload.tags });
    yield put(loadCountResultsRoutine.success(response));
  } catch (error: any) {
    yield put(loadCountResultsRoutine.failure(error?.message));
    toastr.error('Error', 'Load count of results failed');
  }
}

function* fetchAllUsersNumber() {
  try {
    const response = yield call(feedPageService.fetchAllUsersNumber);
    yield put(fetchAllUsersNumberRoutine.success(response));
  } catch (error: any) {
    yield put(fetchAllUsersNumberRoutine.failure(error?.message));
    toastr.error('Error', 'Load count of users failed');
  }
}

function* watchLoadCountResults() {
  yield takeEvery(loadCountResultsRoutine.TRIGGER, loadCountResults);
}

function* watchSearchPosts() {
  yield takeEvery(searchPostsRoutine.TRIGGER, searchPosts);
}

function* watchGetDataRequest() {
  yield takeEvery(fetchDataRoutine.TRIGGER, fetchData);
}

function* watchLikePost() {
  yield takeEvery(likePostRoutine.TRIGGER, likePost);
}

function* watchDisLikePost() {
```

```
  yield takeEvery(disLikePostRoutine.TRIGGER, disLikePost);
}

function* watchFetchAllUsersNumber() {
  yield takeEvery(fetchAllUsersNumberRoutine.TRIGGER, fetchAllUsersNumber);
}

export default function* feedPageSagas() {
  yield all([
    watchGetDataRequest(),
    watchLikePost(),
    watchDisLikePost(),
    watchSearchPosts(),
    watchLoadCountResults(),
    watchFetchAllUsersNumber()
  ]);
}
```

Лістинг файлу screens/FeedPage/reducers/index.ts

```
import { combineReducers } from 'redux';
import { RootState } from '@root/store';
import { reducerCreator } from '@helpers/reducer.helper';
import { feedPageReducer } from '@screens/FeedPage/containers/FeedPage/reducer';
import {
  addMorePostsRoutine,
  disLikePostRoutine,
  fetchDataRoutine,
  likePostRoutine,
  searchPostsRoutine,
  fetchAllUsersNumberRoutine
} from '@screens/FeedPage/routines';

const requests = combineReducers({
  /* PlopJS request placeholder. Do not remove */
  fetchAllUsersNumberRequest: reducerCreator([fetchAllUsersNumberRoutine.TRIGGER]),
  fetchDataRequest: reducerCreator([fetchDataRoutine.TRIGGER]),
  addMorePostsRequest: reducerCreator([addMorePostsRoutine.TRIGGER]),
  likePostRequest: reducerCreator([likePostRoutine.TRIGGER]),
  disLikePostRequest: reducerCreator([disLikePostRoutine.TRIGGER]),
  searchPostsRequest: reducerCreator([searchPostsRoutine.TRIGGER])
});

export default combineReducers({
  requests,
  data: feedPageReducer
});

const reqs = (state: RootState) => state.feedPageReducer.requests;

/* PlopJS request  extractor placeholder. Do not remove */
export const extractSearchPostsLoading = state => reqs(state).searchPostsRequest.loading;
export const extractFetchDataLoading = state => reqs(state).fetchDataRequest.loading;
```

Лістинг файлу screens/FeedPage/routines/index.ts

```
import { createRoutine } from 'redux-saga-routines';

const createFeedPageRoutine = <T extends unknown>(actionName: string) =>
createRoutine<T>(`FEED_PAGE:${actionName}`);

export const fetchAllUsersNumberRoutine = createFeedPageRoutine('FETCH_ALL_USERS_NUMBER');
export const fetchDataRoutine = createFeedPageRoutine('FETCH_DATA');
export const resetDataRoutine = createFeedPageRoutine('RESET_DATA');
export const addMorePostsRoutine = createFeedPageRoutine('ADD_MORE_POSTS');
export const likePostRoutine = createFeedPageRoutine('LIKE_POST_ROUTINE');
export const disLikePostRoutine = createFeedPageRoutine('DISLIKE_POST_ROUTINE');
export const searchPostsRoutine = createFeedPageRoutine('SEARCH_POSTS');
export const loadCountResultsRoutine = createFeedPageRoutine('LOAD_COUNT_RESULTS');
```

Лістинг файлу screens/FeedPage/sagas/index.ts

```
import { all } from 'redux-saga/effects';
import feedPagePageSagas from '@screens/FeedPage/containers/FeedPage/sagas';

export default function* feedPageSagas() {
  yield all([
    feedPagePageSagas()
  ]);
}
```

Лістинг файлу screens/FeedPage/services/feedPage.ts

```
import api from '@helpers/api.helper';

const feedPageService = {
  getData: async (filter: object) => api.get('/api/post/all', { params: filter }),
  getHotPosts: async (filter: object) => api.get('/api/post/hots', { params: filter }),
  getBestPosts: async (filter: object) => api.get('/api/post/bests', { params: filter }),
  likePost: async (post: object) => api.put(
    '/api/postreaction',
    {
      data: post
    }
  ),
  searchPosts: async (filter: object) => api.get('/api/search/list', { params: filter }),
  loadCountResults: async (filter: object) => api.get('/api/search/count', { params: filter }),
  fetchAllUsersNumber: async () => api.get('/api/user/count')
};

export default feedPageService;
```

Лістинг файлу backend/core/db/src/main/java/com/mindbridge/data/domains/post/PostRepository.java

```
package com.mindbridge.data.domains.post;

import com.mindbridge.data.domains.post.dto.PostsReactionsQueryResult;
import com.mindbridge.data.domains.post.model.Post;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.jpa.repository.Query;

import java.util.List;
```

```
import java.util.UUID;

public interface PostRepository extends JpaRepository<Post, UUID>, JpaSpecificationExecutor<Post> {

    @Query("SELECT new com.mindbridge.data.domains.post.dto.PostsReactionsQueryResult("
            + "(SELECT COALESCE(SUM(CASE WHEN pr.liked = TRUE THEN 1 ELSE 0 END), 0) FROM p.reactions pr
WHERE pr.post = p ), "
            + "(SELECT COALESCE(SUM(CASE WHEN pr.liked = FALSE THEN 1 ELSE 0 END), 0) FROM p.reactions pr
WHERE pr.post = p ))"
            + "FROM Post p where p.id = :id")
    PostsReactionsQueryResult getAllReactionsOnPost(UUID id);

    @Query("select p from Post p where p.deleted = false and p.draft = false order by p.createdAt desc ")
    List<Post> getAllPosts(Pageable pageable);

    @Query(value = "select p.* from Posts p, post_reactions pr " +
      "where p.deleted = false " +
      "and p.draft = false " +
      "and p.created_at >= current_date at time zone 'UTC' - interval '7 days' " +
      "group by p.id, p.created_at " +
      "order by (SELECT (COALESCE(SUM(CASE WHEN pr.liked = TRUE THEN 1 ELSE -1 END), 0) - " +
      "(0.1 * -(date_part('day',age(p.created_at, now())))) * " +
      "abs(COALESCE(SUM(CASE WHEN pr.liked = TRUE THEN 1 ELSE -1 END), 0)))) " +
      "FROM post_reactions pr WHERE pr.post_id = p.id) desc", nativeQuery = true)
    List<Post> getHotPosts(PageRequest pageable);

    @Query(value = "select p.* from Posts p " +
      "where p.deleted = false " +
      "and p.draft = false " +
      "order by (SELECT COALESCE(SUM(CASE WHEN pr.liked = TRUE THEN 1 ELSE -1 END), 0) FROM
post_reactions pr WHERE pr.post_id = p.id) desc", nativeQuery = true)
    List<Post> getBestPosts(PageRequest pageable);

    int countPostByAuthorId(UUID id);

    @Query("select p from Post p where p.author.id = :userId order by p.createdAt desc ")
    List<Post> getFirstPostTitles(UUID userId, Pageable pageable);

    @Query("select p.title from Post p where p.id = :id")
    String getTitleById(UUID id);

    @Query("SELECT p FROM Post p WHERE p.deleted = false AND p.author.id = :userId and p.draft = true order by
p.createdAt desc")
    List<Post> getDraftsByUser(UUID userId);

    @Query("select p from Post p where p.deleted = false and p.author.id = :userId order by p.createdAt desc")
    List<Post> getPostsByUser(UUID userId);

    @Query(value = "SELECT p.* " +
      "    FROM Posts p " +
      "        INNER JOIN Post2tag tg " +
      "            ON p.id = tg.post_id " +
      "        INNER JOIN Tags t " +
      "            ON tg.tag_id = t.id " +
      "    WHERE t.name in (:tags) and p.id != :id" +
      "    GROUP BY p.id " +
      "    ORDER BY count(tg.tag_id) DESC", nativeQuery = true)
    List<Post> getRelatedPostsByTags(UUID id, List<String> tags, Pageable pageable);
}
```

Лістинг файлу backend/core/db/src/main/java/com/mindbridge/data/domains/post/dto/PostsReactionsQueryResult.java

```
package com.mindbridge.data.domains.post.dto;

import com.mindbridge.data.domains.tag.model.Tag;
import com.mindbridge.data.domains.user.model.User;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.util.Date;
import java.util.*;
import java.util.UUID;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class PostsReactionsQueryResult {

    public long likeCount;

    public long disLikeCount;

}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/post/PostService.java

```
package com.mindbridge.core.domains.post;

import com.mindbridge.core.domains.helpers.DateFormatter;
import com.mindbridge.core.domains.post.dto.*;
import com.mindbridge.core.domains.postReaction.PostReactionService;
import com.mindbridge.core.domains.postReaction.dto.ReceivedPostReactionDto;
import com.mindbridge.core.domains.tag.dto.TagDto;
import com.mindbridge.core.domains.user.UserMapper;
import com.mindbridge.core.domains.user.UserService;
import com.mindbridge.data.domains.favorite.FavoriteRepository;
import com.mindbridge.data.domains.favorite.model.Favorite;
import com.mindbridge.data.domains.post.PostRepository;
import com.mindbridge.data.domains.post.dto.PostsReactionsQueryResult;
import com.mindbridge.data.domains.post.model.Post;
import com.mindbridge.data.domains.postReaction.PostReactionRepository;
import com.mindbridge.data.domains.postViews.PostViewsRepository;
import com.mindbridge.data.domains.tag.TagRepository;
import com.mindbridge.data.domains.tag.dto.TagDataDto;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Lazy;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;

import java.security.Principal;
import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;
```

```
@Service
@Slf4j
public class PostService {

    private final PostRepository postRepository;

    private final PostReactionService postReactionService;

    private final TagRepository tagRepository;

    private final FavoriteRepository favouriteRepository;

    private final PostReactionRepository postReactionRepository;

    private final UserService userService;

    private final PostViewsRepository postViewsRepository;

    @Lazy
    @Autowired
    public PostService(
        PostRepository postRepository,
        PostReactionService postReactionService,
        TagRepository tagRepository, UserService userService,
        PostReactionRepository postReactionRepository,
        FavoriteRepository favouriteRepository,
        PostViewsRepository postViewsRepository) {
      this.postRepository = postRepository;
      this.postReactionService = postReactionService;
      this.tagRepository = tagRepository;
      this.postReactionRepository = postReactionRepository;
      this.userService = userService;
      this.favouriteRepository = favouriteRepository;
      this.postViewsRepository = postViewsRepository;
    }

    public PostDetailsDto getPostById(Principal principal, UUID id) {
      var post = postRepository.findById(id);
      var postDetailsDto = post.map(PostMapper.MAPPER::postToPostDetailsDto).orElseThrow();


postDetailsDto.setAuthor(userService.setUserStatInformation(UserMapper.MAPPER.userToUserDto(post.orElseThrow().g
etAuthor()).getId()));
      List<String> tags = postDetailsDto.getTags().stream().map(TagDto::getName).collect(Collectors.toList());
      List<RelatedPostDto> relatedPostsDto = postRepository.getRelatedPostsByTags(id, tags, PageRequest.of(0, 3))
        .stream()
        .map(PostMapper.MAPPER::postToRelatedPostDto)
        .collect(Collectors.toList());

      relatedPostsDto.forEach(p -> p.setRating(postReactionService.calcPostRatingById(p.getId())));
      relatedPostsDto.sort(Comparator.comparingLong(RelatedPostDto::getRating).reversed());

      postDetailsDto.setRating(postReactionService.calcPostRatingById(id));
      postDetailsDto.setRelatedPosts(relatedPostsDto);

      if (principal == null) {
        return postDetailsDto;
      }
      var currentUser = userService.loadUserDtoByEmail(principal.getName());
      var reaction = postReactionRepository.getPostReaction(currentUser.getId(), postDetailsDto.getId());
      postDetailsDto.setReacted(reaction.isPresent());
      reaction.ifPresent(postReaction -> postDetailsDto.setIsLiked(postReaction.getLiked()));
      var favourite = favouriteRepository.getFavoriteByPostIdAndUserId(id, currentUser.getId());
```

```
        postDetailsDto.setIsFavourite(favourite.isPresent());
        postDetailsDto.setPostViewsNumber(postViewsRepository.countByPostId(id));
        return postDetailsDto;
    }

    public List<PostsListDetailsDto> getAllPosts(Principal principal, Integer from, Integer count) {
        var pageable = PageRequest.of(from / count, count);
        return postRepository.getAllPosts(pageable).stream().map(post -> mapPost(post,
principal)).collect(Collectors.toList());
    }

    public PostsListDetailsDto mapPost(Post post, Principal principal) {
        var postListDto = PostMapper.MAPPER.postToPostsListDto(post);
        postListDto.setAuthor(userService.setUserStatInformation(post.getAuthor().getId()));
        postListDto.setTags(post.getTags().stream().map(TagDataDto::fromEntity).collect(Collectors.toList()));
        PostsReactionsQueryResult postsReactionsQueryResult = postRepository.getAllReactionsOnPost(post.getId());
        postListDto.setLikesCount(postsReactionsQueryResult.likeCount);
        postListDto.setDisLikesCount(postsReactionsQueryResult.disLikeCount);
        postListDto.setPostRating(postsReactionsQueryResult.likeCount - postsReactionsQueryResult.disLikeCount);
        postListDto.setCreatedAt(DateFormatter.getDate(post.getCreatedAt(), "dd MMMM"));
        postListDto.setPostViewsNumber(postViewsRepository.countByPostId(post.getId()));
        if (principal == null) {
            return postListDto;
        }
        var currentUser = userService.loadUserDtoByEmail(principal.getName());
        var favouritePosts = favouriteRepository.getAllPostByUserId(currentUser.getId());
        setIfFavourite(favouritePosts, postListDto);
        var reaction = postReactionRepository.getPostReaction(currentUser.getId(), post.getId());
        postListDto.setReacted(reaction.isPresent());
        reaction.ifPresent(postReaction -> postListDto.setIsLiked(postReaction.getLiked()));
        return postListDto;
    }

    public void setIfFavourite(List<Favorite> favouritePosts, PostsListDetailsDto post) {
        var found = favouritePosts.stream().filter(favouritePost ->
favouritePost.getPost().getId().toString().equals(post.getId().toString())).findFirst();
        post.setIsFavourite(found.isPresent());
    }

    public List<PostsListDetailsDto> getHotPosts(Principal principal, Integer from, Integer count, UUID userId) {
        var pageable = PageRequest.of(from / count, count);
        var hotPosts = postRepository.getHotPosts(pageable).stream()
            .map(post -> mapPost(post, principal))
            .collect(Collectors.toList());
        var favouritePosts = favouriteRepository.getAllPostByUserId(userId);
        hotPosts.forEach(post -> setIfFavourite(favouritePosts, post));
        return hotPosts;
    }

    public List<PostsListDetailsDto> getBestPosts(Principal principal, Integer from, Integer count, UUID userId) {
        var pageable = PageRequest.of(from / count, count);
        var bestPosts = postRepository.getBestPosts(pageable).stream()
            .map(post -> mapPost(post, principal))
            .collect(Collectors.toList());
        var favouritePosts = favouriteRepository.getAllPostByUserId(userId);
        bestPosts.forEach(post -> setIfFavourite(favouritePosts, post));
        return bestPosts;
    }

    public UUID editPost(EditPostDto editPostDto) {
        var currentPost = postRepository.getOne(editPostDto.getPostId());
        currentPost.setTitle(editPostDto.getTitle());
        currentPost.setText(editPostDto.getText());
```

```
        currentPost.setMarkdown(editPostDto.getMarkdown());
        currentPost.setCoverImage(editPostDto.getCoverImage());
        currentPost.setDraft(editPostDto.getDraft());
        currentPost.setTags(new HashSet<>(tagRepository.findAllById(editPostDto.getTags())));

        return postRepository.save(currentPost).getId();
    }

    public UUID savePost(CreatePostDto createPostDto) {
        var post = PostMapper.MAPPER.createPostDtoToPost(createPostDto);
        var tags = new HashSet<>(tagRepository.findAllById(createPostDto.getTags()));
        post.setTags(tags);
        var savedPost = postRepository.save(post);
        postReactionService
                .setReaction(new ReceivedPostReactionDto(savedPost.getId(), createPostDto.getAuthor(), true));
        return savedPost.getId();
    }

    public String getTitleOfPost(UUID id) {
        return postRepository.getTitleById(id);
    }

    public List<DraftsListDto> getAllDrafts(UUID userId) {
        return postRepository.getDraftsByUser(userId).stream().map(PostMapper.MAPPER::postToDraftDto)
                .collect(Collectors.toList());
    }

    public List<DraftsListDto> getAllMyPosts(UUID userId) {
        return postRepository.getPostsByUser(userId).stream().map(PostMapper.MAPPER::postToDraftDto)
                .collect(Collectors.toList());
    }
}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/post/PostMapper.java

```
package com.mindbridge.core.domains.post;

import com.mindbridge.core.domains.post.dto.*;
import com.mindbridge.data.domains.post.model.Post;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.factory.Mappers;

@Mapper
public interface PostMapper {

    PostMapper MAPPER = Mappers.getMapper(PostMapper.class);

    @Mapping(target = "rating", ignore = true)
    @Mapping(target = "author", ignore = true)
    @Mapping(target = "isFavourite", ignore = true)
    @Mapping(target = "postViewsNumber", ignore = true)
    PostDetailsDto postToPostDetailsDto(Post post);

    RelatedPostDto postToRelatedPostDto(Post post);

    @Mapping(source = "author", target = "author.id")
    @Mapping(target = "tags", ignore = true)
    Post createPostDtoToPost(CreatePostDto createPostDto);
```

```
    DraftsListDto postToDraftDto(Post post);

    @Mapping(target = "author", ignore = true)
    @Mapping(target = "commentsCount", ignore = true)
    @Mapping(target = "likesCount", ignore = true)
    @Mapping(target = "disLikesCount", ignore = true)
    @Mapping(target = "postRating", ignore = true)
    @Mapping(target = "usersCount", ignore = true)
    @Mapping(target = "tags", ignore = true)
    @Mapping(target = "createdAt", ignore = true)
    @Mapping(target = "postViewsNumber", ignore = true)
    PostsListDetailsDto postToPostsListDto(Post post);
}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/post/PostController.java

```
package com.mindbridge.core.domains.post;

import com.mindbridge.core.domains.post.dto.*;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

import java.security.Principal;
import java.util.List;
import java.util.UUID;

@RestController
@RequestMapping("post")
@Validated
public class PostController {

    private final PostService postService;

    @Autowired
    public PostController(PostService postService) {
        this.postService = postService;
    }

    @GetMapping("/{id}")
    public PostDetailsDto getPost(@PathVariable UUID id, Principal principal) {
        return postService.getPostById(principal, id);
    }

    @PostMapping("/create")
    public UUID createPost(@RequestBody CreatePostDto post) {
        return postService.savePost(post);
    }

    @PutMapping("/edit")
    public UUID editPost(@RequestBody EditPostDto editPostDto) {
        return postService.editPost(editPostDto);
    }

    @GetMapping("/all")
    public List<PostsListDetailsDto> getAllPosts(@RequestParam(defaultValue = "0") Integer from,
                                                 @RequestParam(defaultValue = "10") Integer count, Principal principal) {
        return postService.getAllPosts(principal, from, count);
    }

    @GetMapping("/hots")
```

```
    public List<PostsListDetailsDto> getHotPosts(@RequestParam(defaultValue = "0") Integer from,
                                @RequestParam(defaultValue = "10") Integer count,
                                @RequestParam(defaultValue = "") UUID userId,
                                Principal principal) {
      return postService.getHotPosts(principal, from, count, userId);
    }

    @GetMapping("/bests")
    public List<PostsListDetailsDto> getBestPosts(@RequestParam(defaultValue = "0") Integer from,
                                @RequestParam(defaultValue = "10") Integer count,
                                @RequestParam(defaultValue = "") UUID userId,
                                Principal principal) {
      return postService.getBestPosts(principal, from, count, userId);
    }

    @GetMapping("/title/{id}")
    public String getTitle(@PathVariable UUID id) {
      return postService.getTitleOfPost(id);
    }

    @GetMapping("/drafts/{id}")
    public List<DraftsListDto> getAllDrafts(@PathVariable UUID id) {
      return postService.getAllDrafts(id);
    }

    @GetMapping("/allMy/{id}")
    public List<DraftsListDto> getAllMy(@PathVariable UUID id) {
      return postService.getAllMyPosts(id);
    }
}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/post/dto/PostsListDetailsDto.java

```
package com.mindbridge.core.domains.post.dto;

import com.mindbridge.core.domains.user.dto.UserDto;
import com.mindbridge.core.domains.user.dto.UserProfileDto;
import com.mindbridge.data.domains.tag.dto.TagDataDto;
import lombok.Data;

import java.util.*;

@Data
public class PostsListDetailsDto {

    private UUID id;

    private String title;

    private String text;

    private UserDto author;

    private String createdAt;

    private int commentsCount;

    private long likesCount;
```

```
    private long disLikesCount;

    private long postRating;

    private int usersCount;

    private List<TagDataDto> tags;

    private String coverImage;

    private Boolean markdown;

    private Boolean reacted;

    private Boolean isLiked;

    private Boolean isFavourite;

    private int postViewsNumber;
}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/highlight/HighlightController.java

```
package com.mindbridge.core.domains.highlight;

import com.mindbridge.core.domains.highlight.dto.HighlightsDetailsDto;
import com.mindbridge.core.domains.highlight.dto.SavaHighlightDto;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;

import java.util.*;

@RestController
@RequestMapping("highlight")
@Validated
public class HighlightController {

    private final HighlightService highlightService;

    @Autowired
    public HighlightController(HighlightService highlightService) {
        this.highlightService = highlightService;
    }

    @PostMapping("/save")
    public HighlightsDetailsDto saveHighlight(@RequestBody SavaHighlightDto highlightDto) {
        return highlightService.save(highlightDto);
    }

    @GetMapping("/all/{id}")
    public List<HighlightsDetailsDto> getHighlights(@PathVariable UUID id) {
        return highlightService.getAllHighlights(id);
    }

    @GetMapping("/infinite/{id}")
    public List<HighlightsDetailsDto> getHighlights(@PathVariable UUID id, @RequestParam(defaultValue = "0") Integer
from,
                                                    @RequestParam(defaultValue = "10") Integer count) {
        return highlightService.getAllHighlights(id, from, count);
```

```
    }

    @DeleteMapping("/delete/{id}")
    public UUID deleteHighlight(@PathVariable UUID id) {
      return highlightService.deleteHighlight(id);
    }
}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/highlight/HighlightMapper.java

```
package com.mindbridge.core.domains.highlight;

import com.mindbridge.core.domains.highlight.dto.HighlightsDetailsDto;
import com.mindbridge.core.domains.highlight.dto.SavaHighlightDto;
import com.mindbridge.data.domains.highlight.model.Highlight;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.factory.Mappers;

@Mapper
public interface HighlightMapper {

        HighlightMapper MAPPER = Mappers.getMapper(HighlightMapper.class);

        @Mapping(source = "authorId", target = "user.id")
        @Mapping(source = "postId", target = "post.id")
        Highlight saveHighlightDtoToHighlight(SavaHighlightDto savaHighlightDto);

        @Mapping(source = "user.id", target = "userId")
        @Mapping(source = "post.id", target = "postId")
        @Mapping(source = "post.title", target = "postTitle")
        HighlightsDetailsDto fromHighlightToHighlightDetailsDto(Highlight highlight);
}
```

Лістинг файлу backend/core/app/src/main/java/com/mindbridge/core/domains/highlight/HighlightService.java

```
package com.mindbridge.core.domains.highlight;

import com.mindbridge.core.domains.highlight.dto.HighlightsDetailsDto;
import com.mindbridge.core.domains.highlight.dto.SavaHighlightDto;
import com.mindbridge.data.domains.highlight.HighlightRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.UUID;
import java.util.stream.Collectors;

@Service
public class HighlightService {

    private final HighlightRepository highlightRepository;

    @Autowired
    public HighlightService(HighlightRepository highlightRepository) {
      this.highlightRepository = highlightRepository;
```

```
    }

    public HighlightsDetailsDto save(SavaHighlightDto highlightDto) {
        var highlight = HighlightMapper.MAPPER.saveHighlightDtoToHighlight(highlightDto);
        var savedHighlight = highlightRepository.save(highlight);
        return HighlightMapper.MAPPER.fromHighlightToHighlightDetailsDto(savedHighlight);
    }

    public List<HighlightsDetailsDto> getAllHighlights(UUID userId, Integer from, Integer count) {
        var pageable = PageRequest.of(from / count, count);
        var highlights = highlightRepository.getAllByUserId(userId, pageable);
        return
highlights.stream().map(HighlightMapper.MAPPER::fromHighlightToHighlightDetailsDto).collect(Collectors.toList());
    }


    public UUID deleteHighlight(UUID id) {
        highlightRepository.deleteById(id);
        return id;
    }

    public List<HighlightsDetailsDto> getAllHighlights(UUID userId) {
        var highlights = highlightRepository.getAllByUserId(userId);
        return
highlights.stream().map(HighlightMapper.MAPPER::fromHighlightToHighlightDetailsDto).collect(Collectors.toList());
    }
}
```

Лістинг файлу backend/core/db/src/main/java/com/mindbridge/data/domains/highlight/HighlightRepository.java

```
package com.mindbridge.data.domains.highlight;

import com.mindbridge.data.domains.highlight.model.Highlight;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.domain.Pageable;

import java.util.*;

public interface HighlightRepository extends JpaRepository<Highlight, UUID>, JpaSpecificationExecutor<Highlight> {

        @Query("SELECT h from Highlight h where h.deleted = false and h.user.id = :id order by h.createdAt desc")
        List<Highlight>getAllByUserId(UUID id, Pageable pageable);

        @Query("SELECT h from Highlight h where h.deleted = false and h.user.id = :id order by h.createdAt desc ")
        List<Highlight>getAllByUserId(UUID id);
}
```

Лістинг файлу backend/core/db/src/main/java/com/mindbridge/data/domains/highlight/model/Highlight.java

```
package com.mindbridge.data.domains.highlight.model;

import com.mindbridge.data.domains.post.model.Post;
import com.mindbridge.data.domains.user.model.User;
import com.mindbridge.data.model.BaseAuditableEntity;
import lombok.Data;
```

```
import lombok.EqualsAndHashCode;

import javax.persistence.*;
import javax.persistence.criteria.CriteriaBuilder;

@Entity
@Table(name = "highlights")
@Data
@EqualsAndHashCode(callSuper = true, onlyExplicitlyIncluded = true)
public class Highlight extends BaseAuditableEntity {

	private String text;

	@Column(name = "tag_name_start")
	private String tagNameStart;

	@Column(name = "tag_name_end")
	private String tagNameEnd;

	@Column(name = "index_start")
	private Integer indexStart;

	@Column(name = "index_end")
	private Integer indexEnd;

	@Column(name = "off_set_start")
	private Integer offSetStart;

	@Column(name = "off_set_end")
	private Integer offSetEnd;

	@ManyToOne(cascade = CascadeType.REFRESH, fetch = FetchType.LAZY)
	@JoinColumn(name = "user_id")
	private User user;

	@ManyToOne(cascade = CascadeType.REFRESH, fetch = FetchType.LAZY)
	@JoinColumn(name = "post_id")
	private Post post;

	@Override
	public String toString() {
		return "{\"_super\": " + super.toString() + ", " + "\"_class\":\"Highlight\", " + "\"text\":"
					+ (text == null ? "null" : "\"" + text + "\"") + ", " + "\"user_id\":"
					+ (user.getId() == null ? "null" : "\"" + user.getId() + "\"") + ", " + "\"post_id\":"
					+ (post.getId() == null ? "null" : "\"" + post.getId() + "\"") + ", " + "}";
	}

}
```

Лістинг файлу frontend/src/screens/HighlightsPage/containers/HighlightsPage/index.tsx

```
import React, { useEffect } from 'react';
import styles from './styles.module.scss';
import classNames from 'classnames';
import { connect } from 'react-redux';
import { IBindingCallback1 } from '@models/Callbacks';
import {
  addMoreHighlightsRoutine,
  deleteHighlightRoutine,
  fetchHighlightsRoutine
```

```
} from '@screens/HighlightsPage/routines';
import HighlightCard from '@screens/HighlightsPage/components/highlightCard';
import { ICurrentUser } from '@screens/Login/models/ICurrentUser';
import { IHighlight } from '@screens/HighlightsPage/models/IHighlight';
import InfiniteScroll from 'react-infinite-scroll-component';
import { extractHighlightsLoading } from '@screens/HighlightsPage/reducers';
import NotFoundContent from '@components/NotFoundContetn';
import { isEmptyArray } from 'formik';

export interface IHighlightsProps extends IState, IActions {
}

interface IState {
  highlights: IHighlight[];
  currentUser: ICurrentUser;
  hasMore: boolean;
  loadMore: boolean;
  dataLoading: boolean;
}

interface IActions {
  fetchHighlights: IBindingCallback1<object>;
  deleteHighlight: IBindingCallback1<string>;
  setLoadMoreHighlights: IBindingCallback1<boolean>;
}

const params = {
  from: 0,
  count: 10,
  user: ''
};

const HighlightsPage: React.FC<IHighlightsProps> = (
  { fetchHighlights, highlights, currentUser, deleteHighlight, hasMore,
    setLoadMoreHighlights, dataLoading }
) => {
  useEffect(() => {
    if (currentUser.id) {
      params.from = 0;
      setLoadMoreHighlights(false);
      fetchHighlights({ from: 0, count: 10, user: currentUser.id });
    }
  }, [currentUser, fetchHighlights]);

  const handleDeleteHighlight = id => {
    deleteHighlight(id);
  };

  const handleLoadMoreHighlights = filtersPayload => {
    fetchHighlights(filtersPayload);
  };

  const getMorePosts = () => {
    if (!dataLoading) {
      setLoadMoreHighlights(true);
      const { from, count } = params;
      params.from = from + count;
      params.user = currentUser.id;
      handleLoadMoreHighlights(params);
    }
  };

  return (
```

```
    <div className={classNames('content_wrapper', styles.container)}>
      <div className={styles.pageTitle}>
        Your highlights
      </div>
      {!isEmptyArray(highlights) && highlights ? (
        <InfiniteScroll
          style={{ overflow: 'none' }}
          dataLength={highlights ? highlights.length : 0}
          next={getMorePosts}
          hasMore
          loader={' '}
          scrollThreshold={0.9}
        >
          {highlights && highlights.map(highlight => (
            <HighlightCard
              key={highlight.id}
              highlight={highlight}
              handleDeleteHighlight={handleDeleteHighlight}
            />
          ))}
        </InfiniteScroll>
      ) : (
        <NotFoundContent description="Highlights list is empty" />
      )}
    </div>
  );
};

const mapStateToProps: (state) => IState = state => ({
  highlights: state.highlightsReducer.data.highlights,
  currentUser: state.auth.auth.user,
  hasMore: state.highlightsReducer.data.hasMore,
  loadMore: state.highlightsReducer.data.loadMore,
  dataLoading: extractHighlightsLoading(state)
});

const mapDispatchToProps: IActions = {
  fetchHighlights: fetchHighlightsRoutine,
  deleteHighlight: deleteHighlightRoutine,
  setLoadMoreHighlights: addMoreHighlightsRoutine
};

export default connect(mapStateToProps, mapDispatchToProps)(HighlightsPage);
```

ДОДАТОК Б

ВІДГУК

на кваліфікаційну роботу бакалавра
на тему:
«Розробка веб-платформи для соціальної журналістики призначеної для розробників програмного забезпечення»
студента групи 122-19-2 Гречкіна Миколи Олексійовича

ДОДАТОК В

РЕЦЕНЗІЯ

на кваліфікаційну роботу бакалавра
на тему:
«Розробка веб-платформи для соціальної журналістики призначеної для розробників програмного забезпечення»
студента групи 122-19-2 Гречкіна Миколи Олексійовича

ДОДАТОК Г

ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ

ДОДАТОК Д

ПЕРЕЛІК ФАЙЛІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файла	**Опис**
Пояснювальні документи	
Диплом.doc	Пояснювальна записка до дипломного проекту. Документ Word.
Диплом.pdf	Пояснювальна записка до дипломного проекту в форматі PDF
Програма	
Idea_board.zip	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація.ppt	Презентація дипломного проекту