ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення системи для керування блогами»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Валерій КОЛЯДА
(підпис)

Виконав: здобувач вищої освіти групи ПД-44

Валерій КОЛЯДА

Керівник: Володимир САДОВЕНКО
к. ф-м.н, доцент

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Коляді Валерію Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка програмного забезпечення системи для керування блогами»
керівник кваліфікаційної роботи к.ф-м.н., доцент Володимир САДОВЕНКО,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи розробки веб-застосунків, мова програмування Java,

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

1. Огляд та аналіз існуючих систем керування блогами, їх функціональних можливостей
2. Проєктування архітектури програмного застосунку.
3. Програмна реалізація ключових функціональних можливостей системи для керування блогами, включаючи управління публікаціями та механізми взаємодії.
4. Тестування системи.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Архітектура проєкту.
6. Діаграма класів.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання«25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих систем	05.03-13.03.2025	
3	Визначення основних вимог	14.03-16.03.2025	
4	Огляд та аналіз засобів реалізації системи для керування блогами	17.03-22.03.2025	
5	Проєктування архітектури застосунку	23.03-26.03.2025	
6	Програмна реалізація та тестування застосунку	27.03-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувач вищої освіти _____ Валерій КОЛЯДА
(підпис)

Керівник
кваліфікаційної роботи _____ Володимир САДОВЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: _53_ стор., __6_ табл., _21_ рис., _50_ джерел.

Мета роботи – Спрощення процесу взаємодії між авторами та читачами шляхом розробки системи для керування блогами.

Об'єкт дослідження – Процес створення, публікації та взаємодії з контентом блогу.

Предмет дослідження – Засоби та технології розробки веб-застосунків на базі Spring Boot і H2.

Короткий зміст роботи: В роботі проведено аналіз існуючих платформ для ведення блогів, визначено їхні ключові функціональні можливості, переваги та недоліки. На основі аналізу сформульовано функціональні та нефункціональні вимоги. Розроблено архітектуру програмного забезпечення, що базується на багаторівневій моделі та програмно реалізовані ключові функціональні можливості системи, зокрема: механізми реєстрації нових користувачів та їх автентифікації, створення, редагування, видалення та перегляд публікацій, можливість ставити та знімати "лайки" до постів, пошук публікацій за назвою та їх сортування за датою чи популярністю. Реалізовано розмежування прав доступу для звичайних користувачів та адміністраторів. В роботі використано мову програмування Java та фреймворк Spring Boot. В якості системи управління базами СУБД H2. Безпеку застосунку забезпечено засобами Spring Security..

Сферою використання застосунку є створення та ведення тематичних блогів

КЛЮЧОВІ СЛОВА: СИСТЕМА КЕРУВАННЯ БЛОГАМИ, ВЕБ-ЗАСТОСУНОК, SPRING BOOT, JAVA, SPRING DATA JPA, H2 DATABASE, THYMELEAF, SPRING SECURITY, УПРАВЛІННЯ КОНТЕНТОМ, БЛОГ-ПЛАТФОРМА.

ЗМІСТ

ВСТУП

У сучасному цифровому світі блоги стали не просто трендом, а невід’ємною та надзвичайно важливою частиною інформаційного простору. Важко уявити собі інтернет без них. Вони виконують роль багатогранних платформ для самовираження, надаючи можливість користувачам висловлювати власні ідеї, враження та досвід, досвідом та творчістю. Блоги є потужним інструментом для обміну знаннями, сприяючи навчанню та поширенню корисної інформації. Крім того, вони активно використовуються для ведення бізнесу, просування товарів та послуг, а також для формування та підтримки спільнот за інтересами, об'єднуючи людей зі схожими поглядами та захопленнями. Від затишних особистих щоденників, де автори діляться найпотаємнішим, до масштабних корпоративних ресурсів, що формують імідж компаній, та динамічних новинних порталів, які оперативно інформують про події у світі - блоги охоплюють неймовірно широкий спектр застосувань і залучають різноманітні аудиторії. Їхня гнучкість та доступність роблять їх універсальним інструментом комунікації та взаємодії.

Предметною галуззю даної дипломної роботи є саме процес створення, публікації та взаємодії з контентом у середовищі спеціалізованої блог-платформи. Це не просто набір окремих дій, а комплексний та багатоетапний процес, що охоплює весь життєвий цикл контенту. Він починається з моменту зародження ідеї в голові автора, проходить через етапи створення, редагування та публікації матеріалу, і завершується його споживанням читацькою аудиторією. Окрім того, невід'ємною частиною невід’ємним елементом цього процесу виступає управління платформою та технічна підтримка самої платформи, забезпечення її стабільної роботи та безпеки даних. Саме ефективність та зручність усіх цих етапів визначають успішність будь-якої блог-платформи.

Попри наявність широкого спектра вже реалізованих платформ для ведення блогів, від простих конструкторів до складних систем управління контентом, завжди існує простір для вдосконалення та створення спеціалізованих

інструментів, які б краще відповідали конкретним потребам користувачів. Розробка індивідуального рішення для адміністрування блог-контенту дозволяє не тільки глибоко зануритися у технології веб-розробки, але й створити продукт, що максимально точно враховує уявлення про затребуваний функціонал та користувацький досвід.

Метою даної роботи є спрощення процесу взаємодії між авторами контенту та їхніми читачами шляхом створення спеціалізованого програмного середовища для публікації та обговорення матеріалів. Йдеться про розробку інтуїтивно зрозумілої, функціональної та надійної платформи, яка б мінімізувала технічні бар'єри для авторів та забезпечила комфортне споживання контенту для читачів.

Об'єктом дослідження виступає сам процес створення, публікації та подальшої взаємодії з контентом у рамках блог-платформи. Це включає аналіз типових сценаріїв поведінки користувачів, таких як реєстрація, створення та редагування постів, оцінювання матеріалів та навігація по сайту.

Предметом дослідження є засоби та технології, що застосовуються для розробки сучасних веб-застосунків. Особлива увага приділяється стеку технологій на базі Spring Boot, зокрема таким його компонентам, як Spring Data JPA для взаємодії з базою даних, Spring Security для забезпечення механізмів автентифікації та авторизації, Thymeleaf для створення динамічних веб-сторінок, а також застосування вбудованої СУБД H2 як тимчасового сховища даних під час розробки та тестування. Вибір саме цих технологій обумовлений їхньою популярністю, надійністю та широкими можливостями для створення масштабованих та безпечних веб-додатків.

Щоб реалізувати поставлену мету цієї кваліфікаційної роботи, було окреслено такі ключові завдання::

– Дослідити існуючі аналоги у сфері блог-платформ. Визначити їхні ключові функціональні можливості, виявити сильні та слабкі сторони, а також оглянути основні технологічні стеки, що застосовуються для їх розробки;

– Сформулювати функціональні та нефункціональні вимоги до програмного рішення. Функціональні вимоги описуватимуть ключові можливості,

які система повинна надавати користувачам, а нефункціональні вимоги охоплюватимуть такі аспекти, як продуктивність системи, безпека і надійність;

– Здійснити вибір інструментальних засобів реалізації системи для керування блогами.

– На основі визначених вимог спроектувати архітектуру майбутнього застосунку;

– Розробити програмне забезпечення, що втілює основний запланований функціонал блог-платформи, включаючи користувацький інтерфейс та серверну логіку;

– Провести ручне тестування розробленої системи.

Прикладне значення цієї системи полягає у реалізації керування блогами, що охоплює весь необхідний базовий функціонал. Цей програмний продукт може стати основою для подальшої підтримки та впровадження нових функціональних можливостей блог-платформ і підвищення ефективності у застосуванні розробленої системи для вирішення реальних завдань веб-розробки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Історія розвитку блогів та їх роль у сучасному суспільстві

Слово «блог» є скороченням від «веблог» (поєднання слів «веб» та «лог»), спочатку блоги були просто онлайн-щоденником, де люди могли вести облік про своє повсякденне життя, але з того часу вони перетворилися на важливий форум для окремих осіб та підприємств, де вони могли ділитися інформацією та оновленнями [1].

Сьогодні блоги є невід'ємною частиною цифрового середовища. Вони виконують роль універсального інструменту для самовираження, просування продуктів, ведення просвітницької діяльності, а також побудови професійного іміджу. Окремі блогери набувають статусу лідерів думок, а компанії використовують блог-платформи як інструмент контент-маркетингу. Завдяки простоті створення, блоги стали доступними для широких верств населення, що сприяло їх масовому поширенню.

1.2 Цільова аудиторія блог-платформ

Блог-платформи орієнтовані на різні категорії користувачів, кожна з яких має свої потреби, інтереси й очікування щодо функціональності системи. Визначення цільової аудиторії має ключове значення для проєктування ефективного веб-застосунку, оскільки саме від розуміння потреб користувачів залежить структура інтерфейсу, набір функцій та загальна зручність користування.

Типовими користувачами систем керування блогами можна вважати три основні групи:

– Автори — користувачі, які безпосередньо створюють контент. Для них важливими є інтуїтивно зрозумілий редактор публікацій, можливість редагування

та управління власними записами, а також отримання фідбеку у вигляді лайків або переглядів;

– Читачі — кінцеві споживачі інформації. Їх цікавить зручна навігація, пошук матеріалів за ключовими словами, перегляд популярних або нових публікацій. Читачі також цінують можливість взаємодії з контентом, наприклад, через оцінювання публікацій чи підписку на конкретного автора;

– Адміністратори — особи, відповідальні за модерацію та стабільну роботу системи. Вони потребують доступу до інструментів для контролю та редагування контенту, моніторингу активності користувачів, а також підтримки безпеки платформи.

Крім того, до потенційної аудиторії можна віднести аналітиків, які можуть використовувати дані з платформи для досліджень, та розробників, що зацікавлені в її розширенні або інтеграції з іншими сервісами.

1.3 Обґрунтування необхідності спрощеної платформи

На ринку представлено чимало готових рішень для ведення блогів - WordPress, Blogger, Medium, Ghost та інші. За даними W3Techs, «WordPress використовують 61,3% усіх відомих нам веб-сайтів, система управління контентом яких нам відома. Це 43,6% усіх веб-сайтів» [2], що робить його беззаперечним лідером галузі. Ці платформи пропонують широкий функціонал, численні теми оформлення та величезний вибір плагінів. Проте саме ця універсальність часто стає їхнім недоліком: інтерфейс перевантажується опціями, а базові завдання можуть вимагати довгого налаштування.

Кастомізація під унікальні потреби проєкту чи організації нерідко вимагає залучення фахівців, а гнучке налаштування прав доступу чи інтеграція з корпоративними системами легко перетворюються на складний технічний виклик. Саме тому існує потреба в легкій, цілеспрямованій платформі, що фокусуватиметься на базовому функціоналі і водночас забезпечуватиме гарний

рівень безпеки та продуктивності. Така система повинна бути інтуїтивною для авторів і зручною для читачів, а побудова на сучасному стеку гарантуватиме надійність і простоту подальшого розвитку. Саме на вирішення цих завдань і спрямована дана кваліфікаційна робота.

1.4 Аналіз існуючих систем

Блог-платформи є спеціалізованим програмним забезпеченням, яке спрощує створення, редагування та управління контентом. Вони дозволяють користувачам з різним рівнем технічних знань створювати якісний контент і взаємодіяти з аудиторією. Найбільш відомими системами є:

WordPress є найпопулярнішою системою управління контентом у світі [3]. Вона використовується для створення блогів, вебсайтів і навіть інтернет-магазинів. Заснована у 2003 році Меттом Малленвегом і Майком Літтлом, WordPress починалася як форк платформи b2/cafelog, але швидко стала лідером завдяки своїй відкритості та гнучкості [4]. Сьогодні WordPress підтримує понад 43% усіх вебсайтів, що робить її беззаперечним лідером у сфері CMS [5].

Історія та розвиток

WordPress був створений для спрощення процесу блогінгу, дозволяючи користувачам без технічних знань створювати контент. У 2004 році з'явилася версія 1.2, яка ввела підтримку плагінів, що значно розширило можливості платформи [6]. З роками платформа еволюціонувала від простого інструменту для блогів до повноцінної CMS, що підтримує електронну комерцію, форуми та портали.

Основні функції

WordPress має багато можливостей:

– Створення, редагування та видалення постів і сторінок через зручний редактор, який зображено на рисунку 1.1;

– Підтримка тисяч тем і плагінів для кастомізації дизайну та функціоналу;

– Вбудована система коментарів і управління користувачами;

– Механізми оптимізації контенту для пошукових систем, що сприяє покращенню позицій у видачі [4];

– Можливість підключення до соціальних платформ і сервісів аналітики [6].

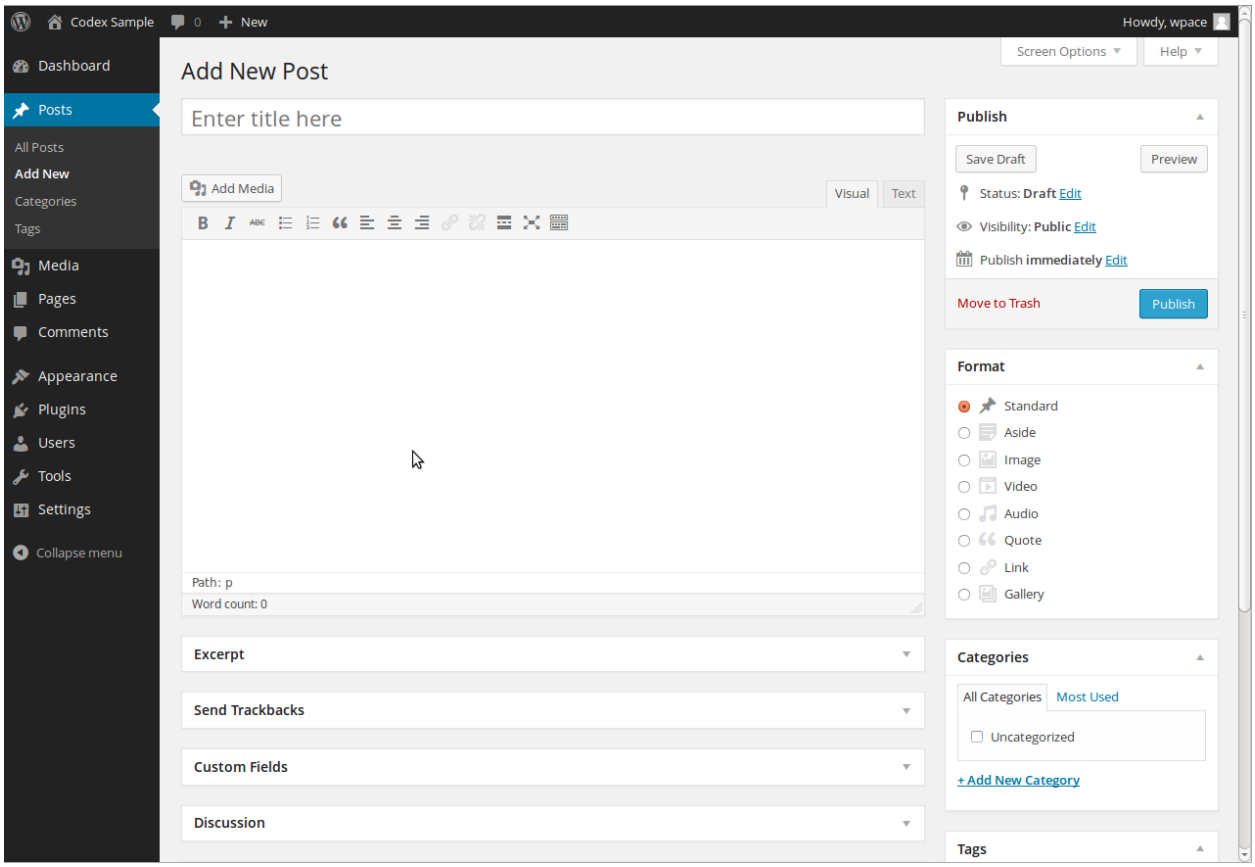


Рис. 1.1 Інтерфейс створення поста в WordPress

Переваги

– Через велику кількість плагінів і тем WordPress має можливість адаптувати до будь-яких потреб, від особистих блогів до корпоративних сайтів [3, 4];

– Спільнота: Велика спільнота розробників забезпечує постійні оновлення, підтримку та безліч ресурсів [7];

– Відкритий код: Як open-source платформа, WordPress дозволяє вільно модифікувати код, що сприяє інноваціям [4, 6].

Недоліки

– Складність для новачків: Велика кількість опцій і плагінів може видатись складною в розумінні для початківців;

– Через популярність є мішенню для хакерів, що вимагає регулярних оновлень і додаткових заходів безпеки;

– Продуктивність: Використання багатьох плагінів може уповільнити роботу сайту, якщо не оптимізувати сервер [8].

Технології

– PHP, що забезпечує легке розгортання на більшості хостингів [4, 6];

– HTML, CSS, JavaScript, з підтримкою сучасних фреймворків у темах [6].

Medium - це платформа для публікації статей, створена у 2012 році Еваном Вільямсом, співзасновником Twitter [9]. Вона орієнтована на авторів і читачів, які цінують якісний, аналітичний контент .

Medium з'явилася як відповідь на потребу в простій платформі для публікації довгих текстів без необхідності створення власного сайту. Medium швидко здобула популярність завдяки мінімалістичному дизайну.

Основні функції

– Інтуїтивний WYSIWYG-редактор для створення статей, який зображено на рисунку 1.2.

– Система "claps" (аналог лайків) для оцінки контенту;

– Можливість підписки на авторів і тематичні публікації;

– Пошук статей за ключовими словами та тегами;

– Підключення до соціальних платформ з метою розповсюдження матеріалів.

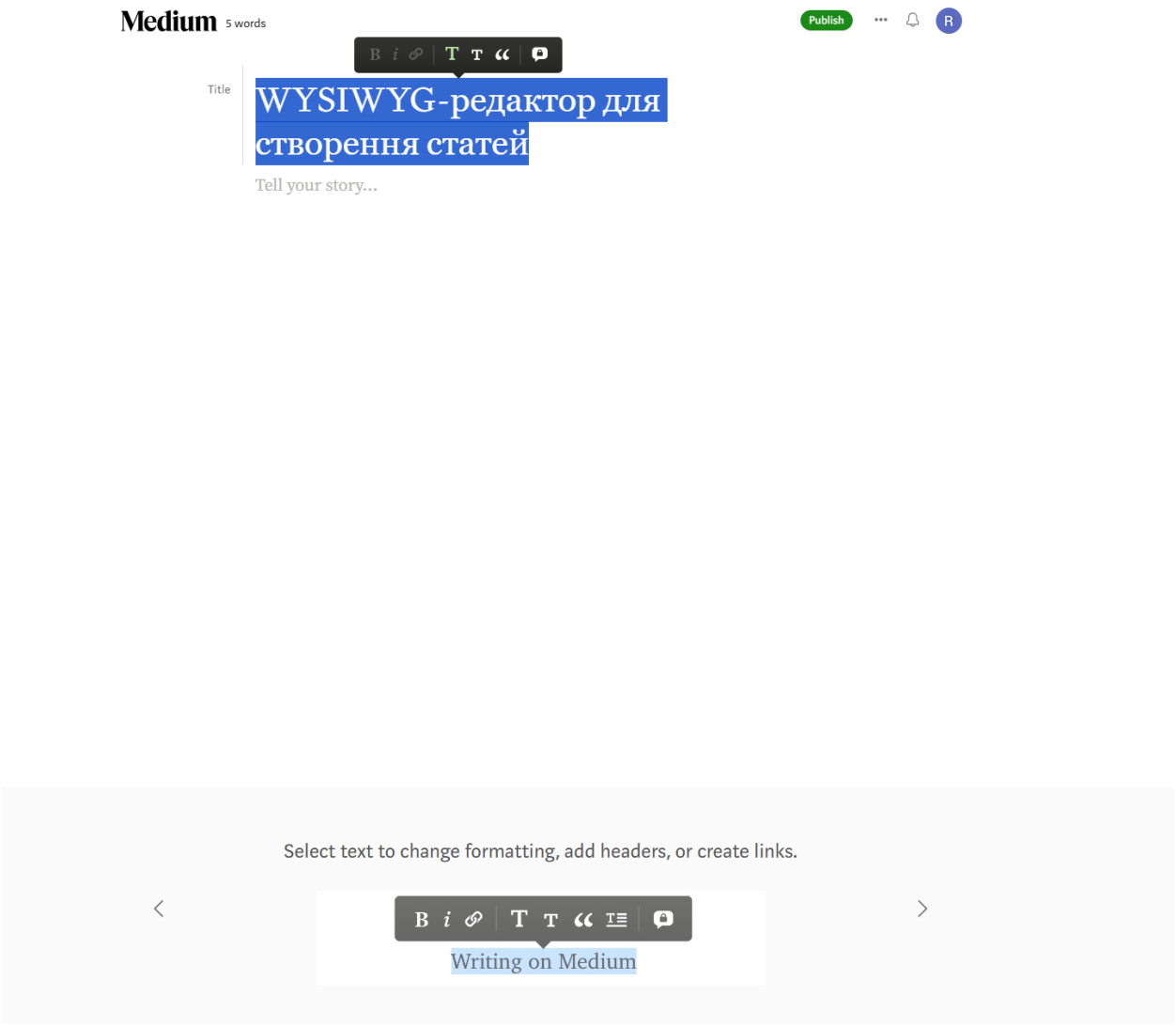


Рис. 1.2 Редактор статей у Medium

Переваги

– Простота: Інтерфейс зосереджений на контенті, що робить його ідеальним для новачків;

– Аудиторія: Велика база читачів сприяє швидкому поширенню статей;

– Монетизація: Партнерська програма дозволяє заробляти на популярних статтях.

Недоліки

– Обмежена кастомізація: Користувачі не можуть змінювати дизайн чи додавати власні функції;

– Контроль контенту: Medium володіє правами на контент, що може бути

проблемою для авторів;

– Платний доступ: Деякі статті доступні лише за підпискою, що може відштовхувати читачів.

Технології

– Серверна частина: Node.js, забезпечує швидку обробку запитів;

– База даних: MongoDB для гнучкого зберігання даних;

– Клієнтська частина: React для створення динамічного інтерфейсу.

Ghost - це сучасна платформа для блогінгу, створена у 2013 році Джоном О'Ноланом як альтернатива WordPress із фокусом на швидкість і простоту [11].

Історія та розвиток

Ghost почав як проєкт на Kickstarter. Платформа позиціонується як інструмент для професійних авторів і видавців, пропонуючи як самостійний хостинг, так і хмарний сервіс Ghost(Pro) [11].

Основні функції

– Редактор постів із підтримкою Markdown зображено на рисунку 1.3 [11];

– Підтримка тем для кастомізації дизайну;

– Система членства та підписки для монетизації [11];

– Вбудована аналітика та SEO-оптимізація [10, 11].

Переваги

– Швидкість: Ghost оптимізований для швидкого завантаження сторінок [11];

– Простий інтерфейс: Зосереджений на створенні контенту без зайвих функцій [11].

Недоліки

– Обмежена екосистема: Менше плагінів і тем порівняно з WordPress;

– Технічні вимоги: Самостійний хостинг потребує знань Node.js [10];

– Менша спільнота: Обмежена підтримка порівняно з WordPress.

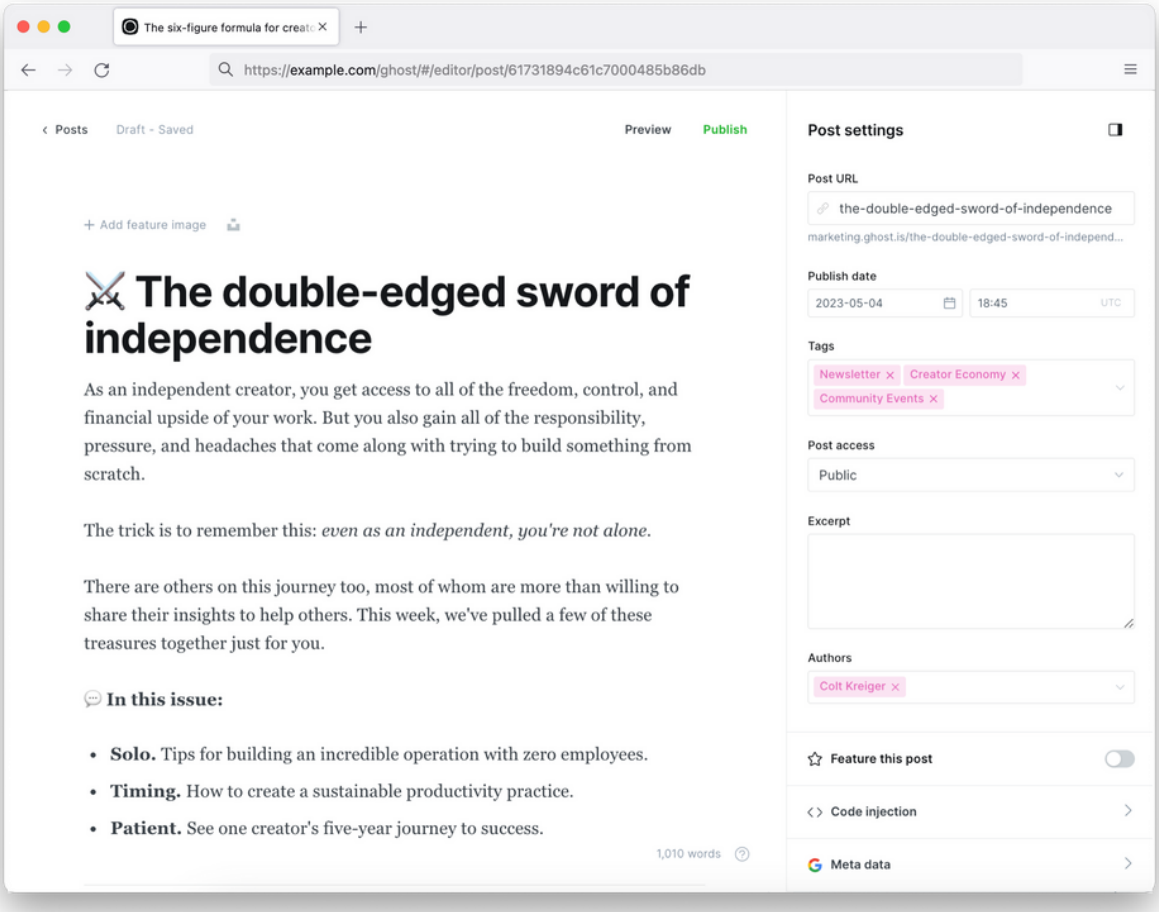


Рис. 1.3 Редактор постів у Ghost

Blogger - це безкоштовна платформа від Google, запущена у 1999 році компанією Pyra Labs і придбана Google у 2003 році [12].

Історія та розвиток

Сервіс Blogger увійшов до числа піонерів у сфері блогінгу, значно спростивши запуск власних веб-журналів для широкого загалу. Після його викупу корпорацією Google платформа отримала інтеграцію з їхніми сервісами, але з часом втратила популярність через застарілий інтерфейс.

Основні функції

– Створення та редагування постів через WYSIWYG-редактор зображено на рисунку 1.4;

– Підтримка шаблонів і кастомізації дизайну;

– Інтеграція з Google Analytics і AdSense;

– Можливість додавання віджетів для соціальних мереж.

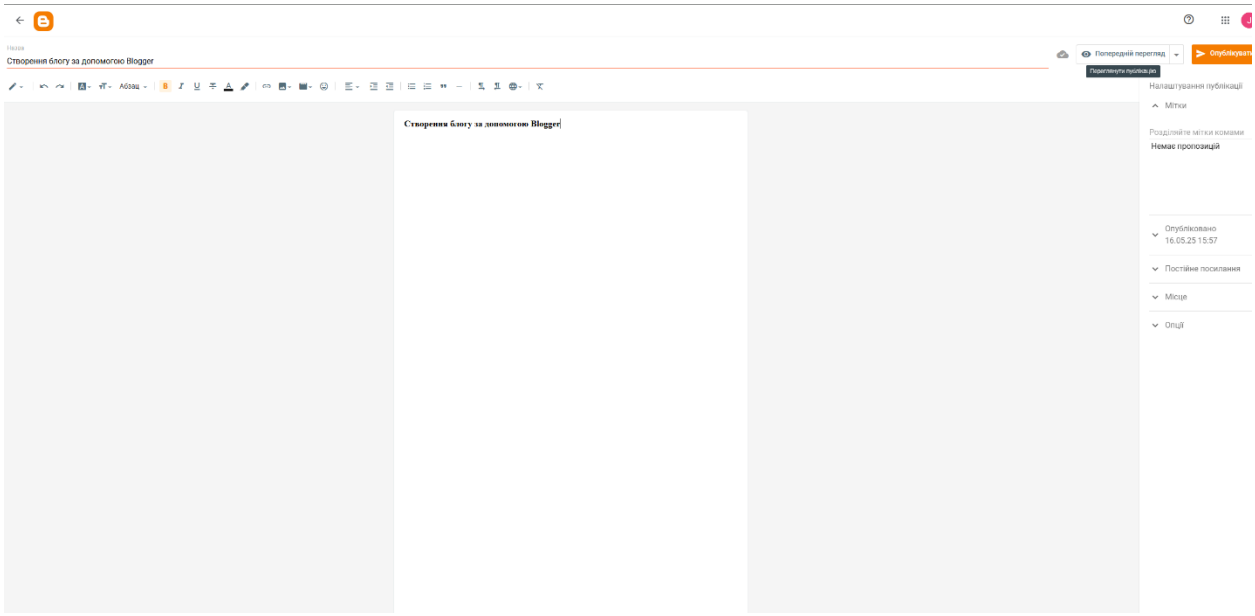


Рис. 1.4 Форма створення поста в Blogger

Переваги

– Безкоштовність: Хостинг надається Google без додаткових витрат;

– Простота: Ідеально для новачків, які не хочуть займатися технічним налаштуванням;

– Безпека: Google забезпечує захист і резервне копіювання [12].

Недоліки

– Застарілий інтерфейс: Дизайн і функціонал не відповідають сучасним стандартам;

– Обмежена кастомізація: Менше можливостей порівняно з WordPress чи Ghost [12];

– Відсутність вбудованих лайків: Соціальні функції потребують сторонніх віджетів.

Також слід згадати менш популярні платформи, але які також мають велику базу користувачів

Write.as (див. рисунок 1.5) – це платформа для написання та публікації блогів [10]. Ця платформа також передбачає відкритий вихідний код сервера (проект

WriteFreely на Go) і дозволяє публікувати без створення акаунту (забезпечуючи приватність). Серед недоліків - доволі простий інтерфейс без передових можливостей розмітки, а також відсутність вбудованого хостингу зображень: для них потрібно використовувати окремий сервіс (Snap.as) [13].

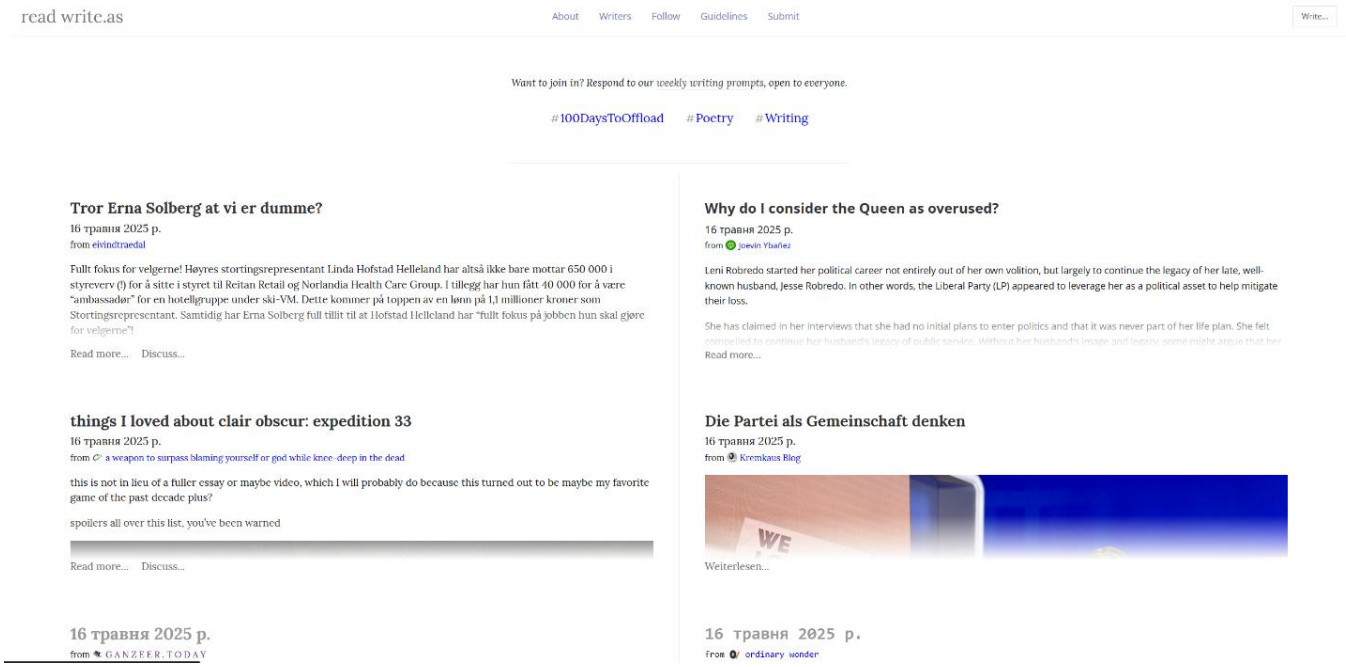


Рис. 1.5 Вигляд головної сторінки Write.as

Tumblr - соціальна мікроблогінг-платформа для коротких публікацій (рисунок 1.6), швидко дозволяє запустити блог завдяки готовим темам і містить вбудовану спільноту (користувачі можуть підписуватися, ставити «лайки» та робити реблоги) [14]. Це робить його зручним для креативних авторів та візуального контенту. Але для серйозного брендингу Tumblr не підходить: шаблони досить прості, можливості кастомізації обмежені, а блокування плагінів не має, крім того, налаштування SEO там також прості [14].

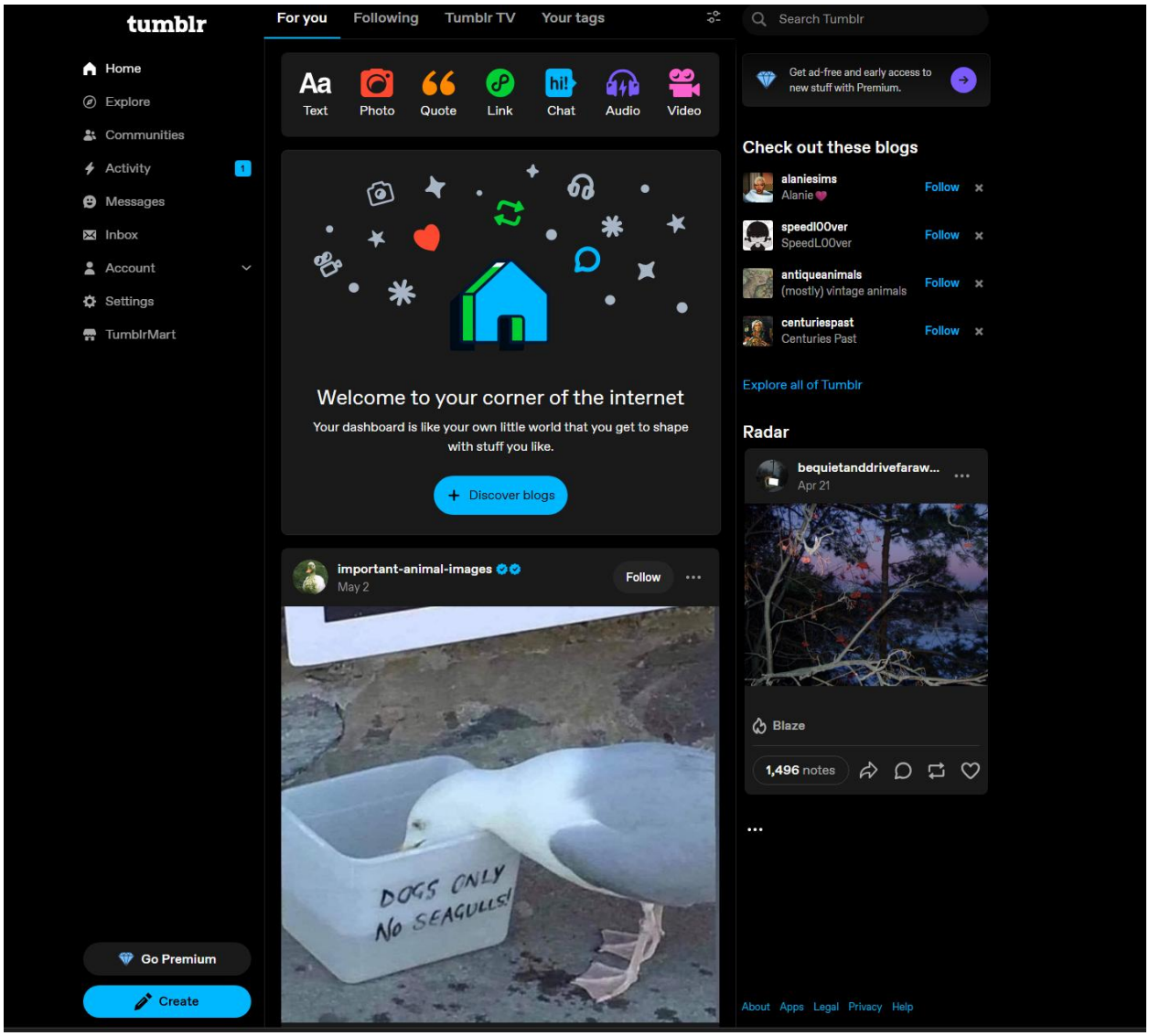


Рис. 1.6 Головна сторінка Tumblr

Svbtle - подібно до Write.as, розроблений для письменників, орієнтований на довготривале зберігання контенту (рисунок 1.7). Він надає дуже обмежений набір налаштувань оформлення (трохи кольорів і логотип) і не підтримує коментарі (використовується система «Kudos» як показник вподобань) [13]. Платформа позиціонується як «вечний» хостинг для вашого блогу - припинення підписки не вилучає пости з інтернету [13]. Недоліком є відсутність оновлень інтерфейсу і опцій, а також платний доступ (біля 75 $/рік за кожен блог) [13].

Рис. 1.7 Svbtle

Після ознайомлення з існуючими блог-платформами, зведемо дані в таблицю 1.1 для порівняльного аналізу та для формування вимог до розробляємого програмного забезпечення системи для керування блогами.

Таблиця 1.1

Результати аналізу популярних блог-платформ

Функція	WordPress	Medium	Ghost	Blogger	Write.as	Tumblr	Svbtle
Реєстрація	+	+	+	+	+	+	+
Публікація	+	+	+	+	+	+	+
Редагування	+	+	+	+	+	+	+
Вподобання	-	+	-	+	-	+	-
Пошук	+	+	+	-	-	+	-
Теми/Шаблони	+	-	+	+	-	-	-
Коментарі	+	+	-	+	-	+	-

У таблиці видно, що функції та напрямок систем реалізовані по-різному. Це дає змогу оцінити важливість наявності певного функціоналу.

1.5 Висновок аналізу

Проведений аналіз дав зрозуміти, що проаналізовані блог-платформи мають як переваги, так і обмеження. Це дало змогу сформувати вимоги до програмного забезпечення і зрозуміти, що деякі функції не важливі для користувачів, які ведуть особисті блоги. Зокрема, дослідження показало, що функціонал коментарів не є пріоритетним для даної кваліфікаційної роботи. Аналіз різних блогів продемонстрував, що пости від звичайних користувачів, які діляться особистими думками, емоціями та історіями, часто збирають значну кількість переглядів та вподобань, однак кількість коментарів залишається мінімальною, а частіше вони взагалі відсутні. Це свідчить про те, що основна взаємодія відбувається на рівні перегляду та вираження симпатії, а не активного обговорення під кожним постом.

Аналогічна ситуація спостерігається і з функціями форматування тексту. Якщо мова не йде про блоги великих компаній чи експертів, звичайні користувачі, які ведуть особисті щоденники та діляться власним досвідом, або використовують форматування мінімально (наприклад, для виділення кількох ключових слів), або ж взагалі не вдаються до нього. Це вказує на те, що для цільової аудиторії простота та швидкість викладу думок є більш пріоритетними, ніж розширені можливості стилізації тексту. Таким чином, при розробці програмного забезпечення системи для керування блогами доцільно зосередитися на базових функціях, які забезпечують зручність публікації контенту, а не на ускладненні інтерфейсу додатковими опціями.

1.6 Вимоги до програмного забезпечення

У результаті аналізу предметної області та існуючих аналогів були сформульовані функціональні та нефункціональні вимоги до розроблюваної системи для керування блогами.

Функціональні вимоги:

– Реєстрація та автентифікація користувачів. Система повинна забезпечувати створення нового облікового запису, вхід до системи за допомогою логіну та пароля та можливість виходу;

– Розмежування прав доступу. Має бути передбачено принаймні дві ролі - звичайний користувач і адміністратор - з відмінним рівнем доступу до функцій платформи;

– Управління публікаціями. Авторизовані користувачі можуть створювати нові дописи, а також редагувати й видаляти власні записи. Адміністратор отримує розширені права: змога змінювати та видаляти будь-які пости;

– Перегляд вмісту. На головній сторінці відображається перелік всіх публікацій із можливістю переходу до детального перегляду кожного допису;

– Взаємодія з контентом. Авторизовані користувачі можуть ставити й знімати «лайки» у записів; передбачено окремий розділ для перегляду вподобаних публікацій;

– Фільтрація та пошук. Користувачі можуть сортувати дописи за датою створення або за кількістю «лайків», а також шукати матеріали за заголовком;

– Перегляд публікацій автора. Забезпечується можливість виведення списку всіх постів, створених конкретним користувачем.

Нефункціональні вимоги передбачають продуктивність, а саме реакцію на дії користувача не більше 1 секунди, хешування паролів, сумісність із сучасними браузерами та зручність інтерфейсу (підказки та зміна теми).

2 ЗАСОБИ РЕАЛІЗАЦІЇ

Підбір набору інструментів та обраних технологій відіграє ключову роль у створенні будь-якого програмного продукту. Від того, які саме фреймворки, бібліотеки та середовища розробки будуть задіяні, залежить не лише швидкість написання коду та зручність роботи, а й загальна продуктивність готової системи, її здатність масштабуватися під зростаюче навантаження, а також стійкість до помилок і збоїв. У рамках системи для керування блогами було проведено аналіз сучасних технологічних стеків. З урахуванням особливостей поставлених завдань, визначених функціональних вимог, було вирішено зупинитись на мові програмування Java.

2.1 Мова програмування

Java - це об'єктно-орієнтована мова програмування загального призначення, яка з моменту свого створення в 1995 році стала однією з найпопулярніших мов у світі [15]. Її головними перевагами є платформа-незалежність, завдяки віртуальній машині Java (JVM), та багатий набір стандартних бібліотек. Java активно використовується для розробки веб-застосунків, мобільних застосунків, корпоративних систем та багато іншого.

2.1.1 Загальні переваги Java

Java здобула широку популярність завдяки низці властивостей, що роблять її ідеальним вибором для створення масштабованих і кросплатформених рішень:

Платформна незалежність: у Java код компілюється в байткод, який виконується на будь-якій системі, де встановлена JVM, це означає, що одна й та сама програма може працювати під Windows, Linux, macOS або будь-якою іншою ОС без додаткових змін у вихідному коді - надзвичайно важливий фактор для веб-додатків, які мають обслуговувати різноманітні клієнтські пристрої [19].

Висока продуктивність: сучасні реалізації JVM містять передові алгоритми оптимізації та динамічної компіляції , а також потужні механізми збору сміття, що дозволяють Java-застосункам ефективно використовувати ресурси навіть під навантаженням [20].

Багата екосистема: навколо Java сформована одна з найбільших спільнот розробників та найбільший набір бібліотек і фреймворків. Зокрема, Spring Boot забезпечує «опініоновану» конфігурацію, що дозволяє запустити серверний модуль усього за кілька рядків коду та уникає багатьох ручних налаштувань [21].

У березні 2024 року була випущена версія Java 22 [16], яка принесла низку нововведень та покращень, спрямованих на підвищення продуктивності, зручності розробки та розширення функціональних можливостей мови. Через ці покращення було вирішено використовувати саме цю версію.

2.1.2 Основні нововведення Java 22

Шаблони рядків: Ця функція дозволяє створювати рядки з вбудованими виразами, що спрощує форматування тексту та підвищує читабельність коду, шаблони рядків доповнюють існуючі літерали рядків та текстові блоки, дозволяючи поєднувати фіксований текст з динамічними виразами, особливо корисно при генерації SQL-запитів, HTML-коду або повідомлень користувачу [17].

Неіменовані змінні та шаблони: Java 22 вводить можливість використовувати символ підкреслення (_) як заміну для змінних, які не використовуються, що дозволяє уникнути оголошення зайвих змінних у лямбда-виразах, обробці винятків та інших конструкціях, де значення змінної не потрібне, сприяє написанню більш чистого та зрозумілого коду [17].

API для зовнішніх функцій та пам'яті: Цей API надає можливість безпечно та ефективно взаємодіяти з нативним кодом та пам'яттю поза JVM, що є сучасною альтернативою Java Native Interface, пропонуючи більш простий та продуктивний спосіб виклику функцій, написаних на інших мовах, та роботи з нативною пам'яттю, що значно полегшує інтеграцію Java-застосунків з бібліотеками, написаними на C або C++ [17].

Запуск багато файлових програм без попередньої компіляції: Java 22 дає змогу одразу виконувати набір .java файлів без окремої компіляції: ця можливість усуває стандартний цикл «компіляція-запуск» і скорочує час на підготовку та перевірку прототипів чи невеликих проєктів, що дозволяє швидко оцінювати робочі ідеї [18].

Покращення продуктивності: У Java 22 було впроваджено низку оптимізацій, спрямованих на підвищення продуктивності додатків. Зокрема, оновлення збирача сміття G1 дозволяє зменшити затримки при роботі з нативною пам'яттю, а також покращено загальну ефективність виконання коду [18].

2.2 Середовище розробки IntelliJ IDEA

Інструмент JetBrains IntelliJ IDEA створений як потужне середовище для професійної розробки на платформі JVM. Він відзначається широким набором можливостей: «розумним» автодоповненням та статичним аналізом коду, зручними засобами для проведення рефакторингу, а також гнучкою екосистемою плагінів, що дозволяє оперативно розширювати функціонал і оптимізувати роботу зі складними веб-проєктами [23].

IntelliJ IDEA надає контекстно-орієнтоване автодоповнення, яке враховує поточні імпорти, сигнатури методів та типи змінних, під час набору коду IDE підсвічує не лише стандартні класи Java, але й методи з активних бібліотек (Spring, Hibernate) та користувацькі API, завдяки чому розробник отримує релевантні підказки в режимі реального часу, що дозволяє знизити кількість синтаксичних помилок та пришвидшити написання коду [22].

IDE виконує «на льоту» перевірку коду на потенційні помилки: цілісність типів, порушення шаблонів безпеки та зайві імпорти [24]. Інспекції коду автоматично запускаються під час редагування і можуть бути налаштовані за пріоритетністю або зовсім відключені, завдяки цьому розробник бачить проблеми ще до компіляції, що підвищує якість кінцевого продукту [24].

IntelliJ IDEA підтримує понад 30 варіантів рефакторингу: перейменування класів і методів, екстракція інтерфейсів, зміна підпису методів, переміщення файлів і пакетів. Під час рефакторингу IDE аналізує всі виклики елементів у проєкті, виявляє залежності та пропонує автоматичне виправлення коду. Це дозволяє проводити навіть великі перетворення структури проєкту без ризику порушити роботу програми [25].

IntelliJ IDEA підтримує Java (включно з версією 22), Kotlin, Groovy та інші мови на JVM. Це означає, що в одному проєкті можна поєднувати кілька мов, наприклад, використовувати Kotlin для бізнес-логіки й Java для низькорівневих модулів, не виходячи з середовища розробки [23].

2.2.1 Інтеграція зі Spring

IDE автоматично розпізнає Spring Boot-проєкт та надає змогу:

– Переходити до оголошень Spring Beans та конфігураційних класів (@Configuration, @Bean);

– Переглядати залежності між бін-класами у вигляді діаграм;

– Автодоповнювати параметри в application.properties та application.yml;

– Пропонувати швидкі фікси для невірних анотацій (@Autowired, @Service) та налаштувань безпеки [26].

2.2.2 Маркетплейс плагінів

До Marketplace IntelliJ IDEA входить понад 2000 плагінів, серед яких:

– Docker і Kubernetes для взаємодії з контейнерами;

– Database Navigator та H2 Console для роботи з базами даних;

– CheckStyle-IDEA та SpotBugs для контролю якості коду;

– PlantUML Integration для вбудованого малювання UML-діаграм.

Можливість швидкого встановлення будь-якого плагіна розширює функціонал IDE відповідно до потреб команди [27].

2.2.3 Засоби налагодження та профілювання

IntelliJ IDEA надає:

– Умовні та логічні брейкпойнти з виразами;

– Перегляд стеку викликів та змінних у реальному часі;

– Інтеграцію з профілювачами (YourKit, VisualVM) для виявлення джерел витоку пам’яті та «вузьких місць» продуктивності [28].

2.2.4 Вбудована робота з базами даних

IDE містить редактор SQL та консоль для взаємодії з реляційними СУБД. Підключивши H2 Database, можна виконувати запити, переглядати схеми й результати у табличному вигляді без переходу в окремий клієнт [29].

2.2.5 Інтеграція з Git

Модуль інтеграції з Git забезпечує можливість:

– Виконувати коміти, створювати та перемикати гілки;

– Здійснювати merge та rebase з візуальним відображенням конфліктів;

– Переглядати історію змін і порівнювати версії файлів прямо в IDE [30].

2.3 Spring Framework та ключові Spring-модулі

Spring Framework - це всеохопна платформа для розробки Java- застосунків, яка надає гнучку інверсію контролю, аспектно-орієнтоване програмування та широкий набір модулів, що дозволяють реалізовувати веб, бекенд і корпоративні рішення [31]. Головна перевага Spring полягає в модульності: можна підключати лише необхідні компоненти, що зменшує складність і розмір проєкту, а також сприяє чіткому розподілу обов'язків між шарами програми [31].

2.3.1 Spring Boot

Spring Boot забезпечує розробникам комплекс інструментів, спрямованих на оптимізацію початкових налаштувань проєкту та оперативне розгортання

застосунку в експлуатацію, завдяки продуманій архітектурі автоконфігурації та готовим стартовим залежностям, робота над проєктом може розпочатися майже без зайвих кроків із налаштування [21].

Нижче наведено ключові компоненти та механізми Spring Boot, які дозволяють швидко перейти від ідеї до працюючого веб-застосунку:

– Автоконфігурацію «за найкращими практиками», що аналізує залежності в pom.xml і автоматично налаштовує відповідні бін-класи та сервіси.

– Інтегрований веб-сервер (наприклад, Tomcat чи Jetty) дає змогу розгортати програму у формі автономного JAR-файла без необхідності додаткового розміщення на зовнішньому сервері [21];

– Spring Boot Starters - набір готових залежностей (наприклад, spring-boot-starter-web, spring-boot-starter-security), які підключають потрібні бібліотеки у єдиному пакеті;

– Actuator - модуль моніторингу (за потреби), що додає REST-ендпойнти для діагностики, метрик і перевірки стану [21].

У версії 3.2.5 Spring Boot підтримує Java 22, віртуальні потоки (preview), спрощене створення Docker-образів та покращену інтеграцію з Kubernetes. Вбудовані інструменти для метрик, трасування та логування допомагають відстежувати стан сервісу в реальному часі [21].

2.3.2 Spring MVC, REST API та Thymeleaf

Spring MVC - це веб-фреймворк, що реалізує патерн Model-View-Controller і надає механізми маршрутизації HTTP-запитів до контролерів [32]. Основні можливості:

– Контролери з анотаціями @Controller і @RestController для обробки веб-сторінок та REST-запитів відповідно [32].

– Обробка форм і валідація через @RequestParam, @ModelAttribute, @Valid [32].

– Відображення у вигляді HTML за допомогою шаблонів (Thymeleaf), а також повернення JSON/XML для клієнтських SPA чи мобільних застосунків [32].

Thymeleaf - сучасний серверний шаблонізатор, який використовує «натуральні шаблони»: HTML-файли з атрибутами th:*, що залишаються валідними навіть без обробки сервером, що дозволяє дизайнерам і фронтенд-розробникам працювати з макетами як із звичайними HTML-сторінками, а розробникам - швидко впроваджувати бізнес-логіку через Spring Model [33].

REST API в Spring реалізується через @RestController, @RequestMapping і спеціальні анотації для HTTP-методів (@GetMapping, @PostMapping тощо), автоматичне перетворення між Java-об'єктами та JSON здійснює Jackson (або інший контейнер, налаштований у Starter) [31].

2.3.3 Spring Data JPA

Spring Data JPA значно спрощує роботу з реляційними базами даних. Розробник створює інтерфейс, що розширює JpaRepository<Entity, ID>, і отримує готовий набір CRUD-операцій, а також можливість оголошувати власні методи пошуку за іменем (наприклад, findByTitleContainingIgnoreCase(String title)) [34]. Додаткові можливості:

– Підтримка сторінок (pagination) і сортування через параметри запиту.
– Транзакційне управління через @Transactional.
– Підключення кастомних запитів через анотацію @Query [34].

2.3.4 Spring Security

Spring Security - ключовий модуль для забезпечення безпеки Java- застосунків [35]. Основні компоненти:

– Аутентифікація: підтримка форми логіну, Basic Auth, OAuth2, LDAP.
– Авторизація: обмеження доступу до URL та методів через ролі (@PreAuthorize, @Secured) чи вирази SpEL.
– CSRF Protection, CORS, налаштування HTTP-заголовків безпеки.
– Шифрування паролів: використання BCryptPasswordEncoder та наступних стандартів [35].

Для налаштування безпеки застосунок використовує конфігурацію Java (клас із @EnableWebSecurity та SecurityFilterChain), що дозволяє централізовано описати політики авторизації й аутентифікації.

2.4 Вбудована база даних H2

H2 Database Engine - це легка, вбудована реляційна СУБД, написана на мові Java і призначена для використання як у режимі in-memory, так і як файлове сховище, ідея H2 полягає в тому, щоб надати розробникам швидку і просту у налаштуванні базу даних, яка не потребує окремого серверу, є надзвичайно швидкою та повністю сумісна з JDBC і SQL стандартами [36].

Основні характеристики H2:

– Режим in-memory та файлове зберігання: у пам'яті (для швидкого прототипування та тестування) або на диску у вигляді єдиного файлу (для збереження даних між запусками);

– Повна підтримка SQL та JDBC: дозволяє використовувати знайомі запити та підключення без додаткових драйверів;

– Малий розмір: компактне ядро (менше 2 МБ), що робить H2 оптимальною для легких мікросервісів і тестових середовищ;

– Вбудована веб-консоль: графічний інтерфейс для виконання SQL-запитів, перегляду схем та даних прямо в браузері;

– Продуктивність: H2 вирізняється хорошими показниками швидкодії, зокрема під час роботи з великими наборами даних, що зумовлено використанням оптимізованих алгоритмів читання та запису [36].

Переваги використання H2 у розробці:

– Миттєве розгортання: не потрібно налаштовувати окремий сервер БД - достатньо додати залежність у pom.xml, і система готова до роботи;

– Ізольованість середовищ: кожен розробник або CI/CD процес може мати власну in-memory БД, що запобігає конфліктам даних і забезпечує чисту «чисту» базу для тестів;

– Зручне тестування: легко впровадити у тести Spring через автоматичну конфігурацію spring-boot-starter-data-jpa - при запуску контексту використовується H2, без необхідності зовнішніх ресурсів;

– Графічна консоль: дозволяє швидко перевіряти структуру таблиць і виконувати ad hoc запити без відходу з IDE чи командного рядка [36].

2.4.1 Інтеграція з Spring Data JPA

При використанні Spring Boot достатньо вказати драйвер і URL з jdbc:h2:mem:testdb (або інший шлях), щоб Spring Data JPA автоматично підключилась до H2.

Після цього всі репозиторії, що успадковують JpaRepository, працюватимуть з H2 без додаткових налаштувань.

2.4.2 Використання в проєкті

У системі керування блогами H2 застосовується як тимчасове сховище під час розробки та тестування. Це дозволяє:

Швидко перевіряти нові сутності і міграції без додаткових дій по налаштуванню БД.

Використовувати веб-консоль H2 (/h2-console) для ручного огляду даних під час налагодження.

2.2 Технології фронтенду: HTML, CSS та JavaScript

У сучасному веб-розробці **HTML**, **CSS** і **JavaScript** виступають базовим трикутником технологій, що визначають зовнішній вигляд, структуру та поведінку інтерфейсу користувача. Нижче наведено огляд кожної з цих складових із акцентом на їхні ключові можливості та роль у реалізації багатофункціональних веб-застосунків.

2.2.1 HTML

HTML є мовою розмітки, котра задає семантичну структуру документів у Всесвітній мережі [37]. З виходом **HTML5** стандарти значно розширилися, включивши:

- Семантичні елементи (<header>, <nav>, <article>, <section>, <footer>) для чіткого розподілу логічних блоків сторінки;
- Розширені форми із новими типами полів (email, date, range тощо) та вбудованою валідацією, що знижує потребу у додатковому JavaScript [38];
- Мультимедійні теги (<video>, <audio>, <canvas>, <picture>) для вбудованого відтворення відео, аудіо та роботи з графікою;
- API для офлайн-режиму (localStorage, sessionStorage, IndexedDB), що дозволяють зберігати дані на стороні клієнта та будувати прогресивні веб-застосунки.

HTML5 також закріпив внутрішню підтримку таких корисних можливостей, як перетягування елементів drag-and-drop і робота з геолокацією, що спрощують розробку інтерактивних функцій.

2.2.2 CSS

CSS [37] — це мова оформлення, що відповідає за візуальну презентацію розмічених HTML-елементів. Серед її сучасних можливостей варто виділити:

- Модульна архітектура: замість великих монолітних версій тепер існують окремі модулі (Layout, Flexbox, Grid, Animations), що швидко розвиваються та оновлюються [37];
- Гнучкі макети: Flexbox і CSS Grid дозволяють створювати адаптивні сіткові структури без використання «плаваючих» елементів або складних хитрощів;
- Анімації та переходи (@keyframes, transition) без застосування JavaScript, що дає змогу оживити інтерфейс і покращити користувацький досвід;
- Псевдокласи і псевдоелементи (:hover, :focus, ::before, ::after), які спрощують стилізацію взаємодій та декоративних вставок;

– Підтримка темізації: можливість реалізації світлої та темної тем завдяки медіа-запитам prefers-color-scheme та CSS-змінним.

CSS також оптимізує продуктивність за рахунок кешування зовнішніх стилів та розділення презентації і структури документа, що спрощує масштабування проєкту та підтримку.

2.2.3 JavaScript

JavaScript [38] — це мова сценаріїв, відповідальна за динамічну поведінку веб-сторінок. Основні напрямки застосування та ключові можливості:

– Маніпуляція DOM: динамічне створення, видалення та модифікація елементів сторінки на льоту, що дозволяє реалізувати інтерактивні інтерфейси;

– Сучасний синтаксис ES6+: класи, модулі import/export, стрілкові функції та деструктуризація об'єктів забезпечують чистіший та підтримуваний код.

– Подієво-орієнтована модель: обробники подій (click, submit, input), що реагують на дії користувача;

– Fetch API та AJAX: асинхронний обмін даними з сервером без перезавантаження сторінки, що лежить в основі SPA-застосунків;

– Promise та async/await: зручні конструкції для обробки асинхронних операцій з покращеним контролем потоку виконання;

– Підтримка широкого спектра бібліотек та фреймворків (React, Vue.js, Angular), що дозволяє вибрати оптимальний інструмент для побудови складного клієнтського коду.

3 ПРОЄКТУВАННЯ ТА РОЗРОБКА СИСТЕМИ

3.1 Варіанти використання системи для керування блогами

Перед описом конкретних сценаріїв взаємодії корисно звернутися до загального контексту моделювання систем. UML (Unified Modeling Language) - це стандартизована мова графічного моделювання, яка широко застосовується для графічного відображення, точного опису, проєктування та документування складових програмної системи. Діаграми в рамках UML дозволяють представити структуру системи (класи, компоненти), її динаміку (послідовності, стани) та організацію (розгортання) в уніфікованому форматі, що зрозумілий і технічним спеціалістам, і бізнес-зацікавленим сторонам. Використання UML сприяє поліпшенню комунікації в команді, виявленню потенційних проблем ще на стадії проєктування та забезпечує єдину "мову" для подальшої підтримки та розвитку продукту [39,40].

Під час створення ПЗ сценарії використання (use cases) використовують для для формального опису функціональних вимог із позиції кінцевого користувача. Вони фіксують послідовність кроків взаємодії між актором (користувачем або зовнішньою системою) і самим застосунком, чітко окреслюючи, які дії повинен виконувати програмний продукт у різних ситуаціях. Підхід use-case полегшує обговорення із зацікавленими сторонами, оскільки формулює вимоги у природній мові з мінімальною кількістю технічних деталей, і водночас сприяє перевірці повноти й однозначності очікуваної поведінки системи [41].

У системі керування блогами виокремлено три класи користувачів, кожен із яких має власні права та обмеження (див. рисунок 3.1):

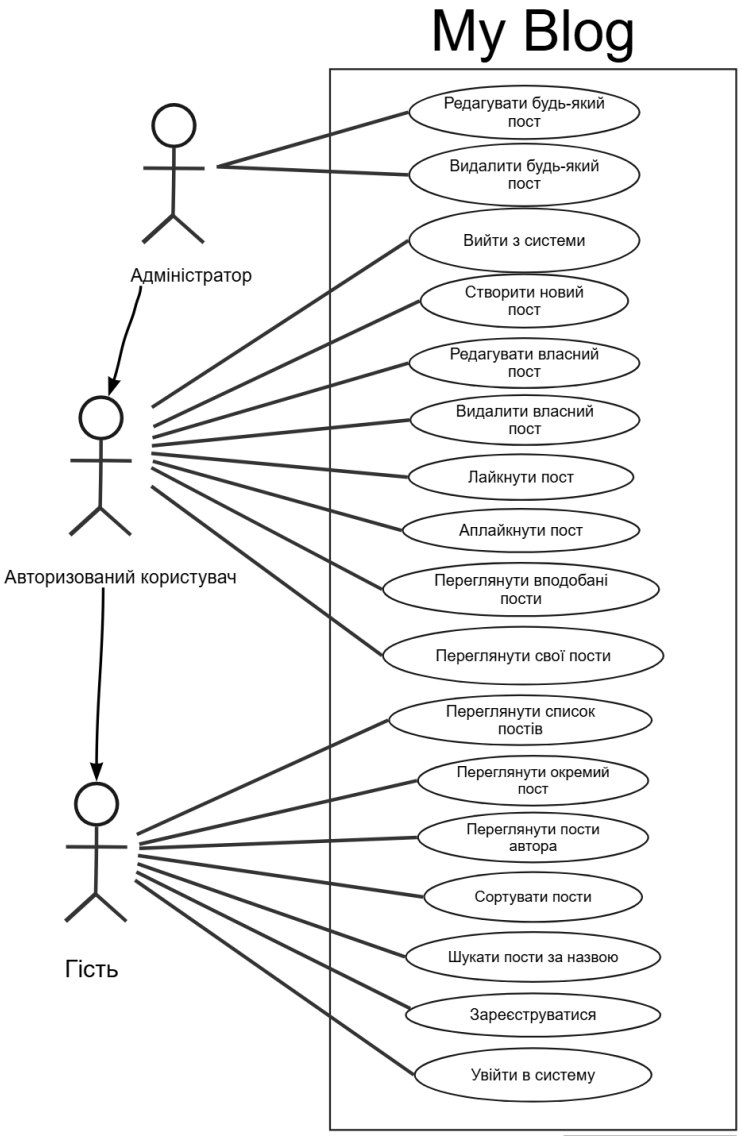


Рис. 3.1 Схематичне зображення сценаріїв взаємодії з системою

Гість (неавторизований користувач): Відвідувач платформи без облікового запису може:

– Переглядати список публікацій на головній сторінці;

– Читати вміст окремих дописів;

– Використовувати пошук за заголовками публікацій;

– Перейти на форму реєстрації чи входу для отримання розширених можливостей.

Авторизований користувач (ROLE_USER): Після успішної автентифікації користувач отримує всі права гостя та додаткові можливості:

– Створення власних постів;

– Редагування власних постів;
– Видалення власних постів;
– Позначення постів як "вподобані" (ставлення «лайків»);
– Перегляд списку своїх публікацій (через сторінку автора);
– Перегляд списку вподобаних матеріалів.

Адміністратор (ROLE_ADMIN): Користувач з розширеними правами, що включають всі можливості Авторизованого користувача, а також:

– Редагування будь-яких постів у системі;
– Видалення будь-яких постів у системі.

3.2 Основні сценарії (Use Cases)

Нижче наведено перелік ключових варіантів використання, які реалізовані в системі:

– Перегляд списку постів
– Перегляд окремого поста
– Пошук постів за назвою
– Сортування постів (за датою, за кількістю вподобань)
– Реєстрація нового користувача
– Вхід до системи
– Вихід із системи
– Створення нового поста
– Редагування власного поста
– Видалення власного поста
– Вподобання поста (лайк)
– Скасування вподобання поста (анлайк)
– Перегляд вподобаних постів
– Перегляд постів певного автора (включаючи власні)
– Редагування будь-якого поста (адміністратор)
– Видалення будь-якого поста (адміністратор)

3.2.1 Вхід до системи

Актор: Гість

Мета: Отримати статус авторизованого користувача для доступу до розширених функцій.

Передумови: Гість перебуває на сайті.

Основний сценарій:

1. Гість активує перехід до форми входу (натискає кнопку "Вхід").
2. Система відображає сторінку входу з полями для введення даних.
3. Гість вводить свою вводить свій e-mail та пароль у відведені поля.
4. Гість підтверджує введення даних (натискає кнопку "Увійти").
5. Здійснюється звірка наданих даних зареєстрованим користувачам у базі даних.
6. У разі успішної автентифікації система надає користувачеві доступ до функцій авторизованого користувача (можливість створювати пости, ставити лайки) та перенаправляє на стартову сторінку.

Інший можливий варіант сценарію (невдала автентифікація):

– Якщо на кроці 5 введені дані не відповідають жодному зареєстрованому користувачеві або пароль невірний, система виводить сповіщення про невдалу спробу та залишає користувача на сторінці входу для повторної спроби.

3.2.2 Створення нового поста

Актор: Авторизований користувач

Мета: Створити та опублікувати новий пост у блозі.

Передумови: Користувач успішно увійшов до системи.

Основний сценарій:

1. Користувач ініціює створення нового поста (натискає кнопку "Написати блог").
2. Система відображає форму для створення поста з полями для заголовка та основного тексту.

3. Користувач заповнює поля "Заголовок" та "Текст" своїм контентом.

4. Користувач підтверджує публікацію поста (натискає кнопку "Опублікувати").

5. Система зберігає новий пост у базі даних, пов'язуючи його з обліковим записом автора.

6. Після успішного збереження пост стає видимим для інших користувачів.

Альтернативний сценарій (помилка при створенні):

– Якщо під час збереження виникає помилка (наприклад, збій підключення до бази даних або інша непередбачена ситуація), система може відобразити загальну сторінку помилки (/pomilka).

3.2.3 Вподобання поста (Лайк)

Актор: Авторизований користувач

Мета: Висловити схвалення публікації, поставивши "лайк".

Передумови: Користувач переглядає пост та є авторизованим.

Основний сценарій (реалізація з окремими діями "лайк" / "анлайк"):

1. Користувач натискає кнопку "Лайк" (або аналогічну) біля поста.

2. Система реєструє дію "лайк" від користувача для даного поста, додаючи зв'язок між користувачем та постом у базі даних. Якщо користувач вже лайкнув цей пост, повторне натискання кнопки "Лайк" прибере його з поста

3. Лічильник лайків для поста оновлюється. Фронтенд може відобразити оновлену кількість лайків та змінений стан кнопки (Заповнене серце або стає пустим).

Сценарій скасування вподобання (Анлайк):

1. Якщо користувач раніше лайкнув пост, він може натиснути кнопку "Анлайк";

2. Система видаляє запис про "лайк" від цього користувача для даного поста;

3. Лічильник лайків оновлюється, і стан кнопки повертається до початкового.

3.2.4 Редагування поста адміністратором

Актор: Адміністратор

Мета: Внести зміни до існуючого поста або вилучити недоречний контент.

Передумови: Адміністратор авторизований у системі.

Основний сценарій:

1. Адміністратор знаходить пост, який потребує редагування (через головну сторінку, пошук або сторінку автора).
2. На сторінці поста адміністратор бачить та активує опцію "Редагувати" (доступну завдяки його правам).
3. Система відображає сторінку редагування поста, заповнену поточним вмістом цього поста (заголовок, текст).
4. Адміністратор вносить необхідні зміни до заголовка та/або тексту поста.
5. Адміністратор підтверджує внесення змін (натискає кнопку "Оновити").
6. Система зберігає оновлений контент поста в базі даних.
7. Після успішного оновлення система перенаправляє адміністратора на сторінку перегляду оновленого поста.

3.3 Архітектура та структура програмного застосунку

Розробка системи для керування блогами базувалася на необхідності створення надійного, підтримуваного та гнучкого програмного продукту. Досягнення цих цілей неможливе без ретельного проєктування архітектури та чіткої організації структури коду.

3.3.1 Загальна архітектурна модель системи

В основі програмної реалізації блог-платформи лежить випробувана часом багаторівнева архітектурна модель, що забезпечує логічний поділ функціональності системи на ізольовані шари, кожен з яких має чітко визначену зону відповідальності та взаємодіє з іншими шарами через стандартизовані інтерфейси [42]. Обрана для проєкту архітектура візуалізована на рисунку 3.2:

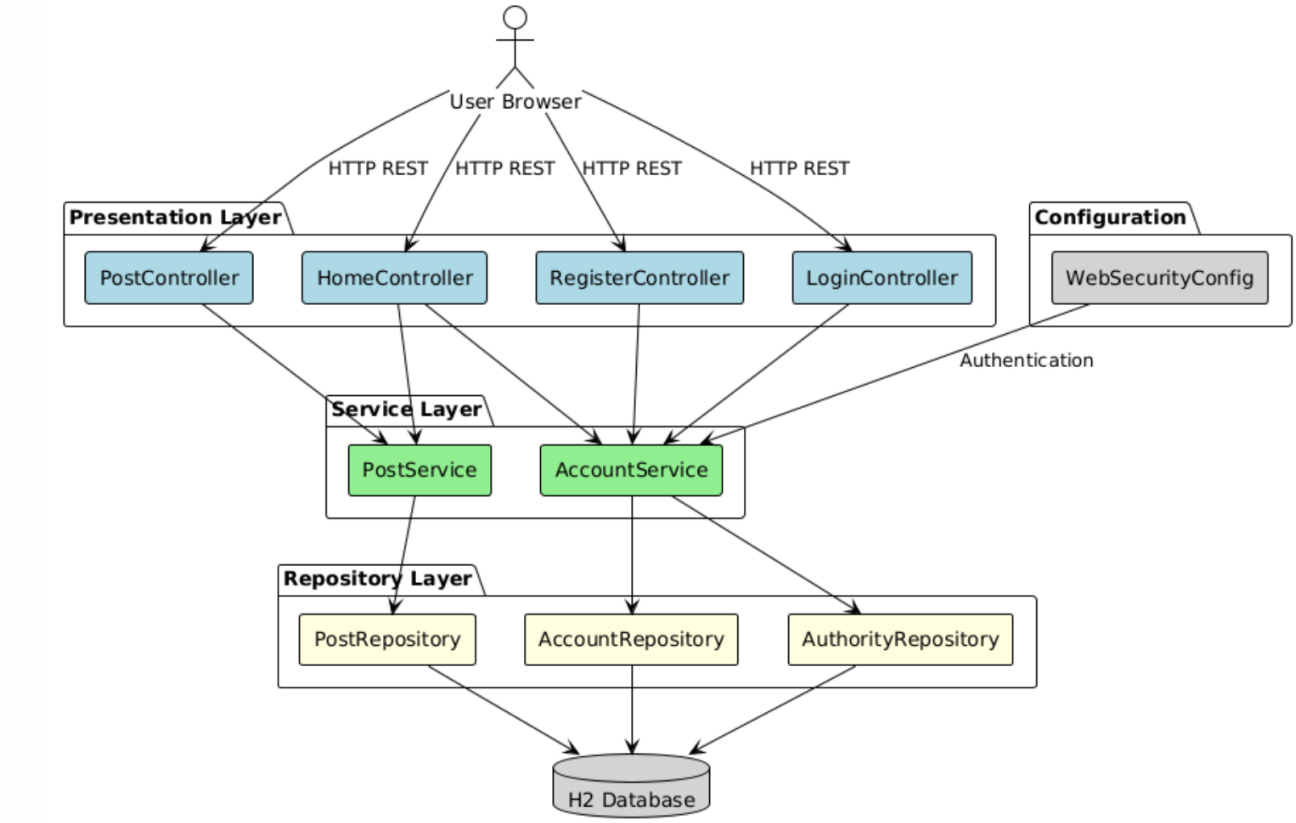


Рис. 3.2 Архітектурна схема системи

– Шар представлення (Presentation Layer): Виступає як інтерфейс між користувачем та системою, головне завдання якого – прийом HTTP-запитів, ініційованих діями користувача у браузері, їхня передача для подальшої обробки та коректне відображення отриманих результатів. У даному проєкті цей шар реалізований з використанням контролерів Spring MVC [42].

– Сервісний шар (Service Layer): виступає центральним осередком бізнес-логіки, де реалізуються операції та алгоритми предметної області,

виконуються перевірки й обробка даних, а також координується взаємодія між рівнем представлення й доступом до даних [42].

- Шар доступу до даних (Repository Layer): опікується низькорівневими операціями з базою, приховуючи складнощі SQL-запитів, та надає уніфікований інтерфейс для операцій збереження, пошуку й оновлення сутностей [42].
- Моделі Даних (Domain Entities): Хоча на архітектурній схемі (див. рисунок 3.2) вони не виділені в окремий прямокутний "шар", моделі даних (Account, Post, Authority) є фундаментальною частиною системи. Вони визначають структуру об'єктів предметної області, якими оперують сервіси та які зберігаються у базі даних за допомогою шару доступу.
- Шар конфігурації (Configuration Layer): Забезпечує налаштування та ініціалізацію ключових аспектів функціонування системи, таких як параметри безпеки, механізми автентифікації та авторизації, а також завантаження початкових даних при старті застосунку.

Вибір такої архітектури обґрунтований її перевагами: чітке розмежування функціональних обов'язків, що спрощує розробку та розуміння системи, підвищена тестованість компонентів, гнучкість у внесенні змін та можливість відносно легкої заміни або модернізації окремих шарів, а також закладений потенціал для подальшого масштабування [42].

3.3.2 Структурна організація програмних компонентів

Дотримання принципів модульності та чіткої структури є ключовим для підтримки та розвитку кодової бази. Вихідний код системи організований у пакунки відповідно до їх функціонального призначення. Детальна ієрархія каталогів та пакунків проєкту представлена на рисунку 3.3

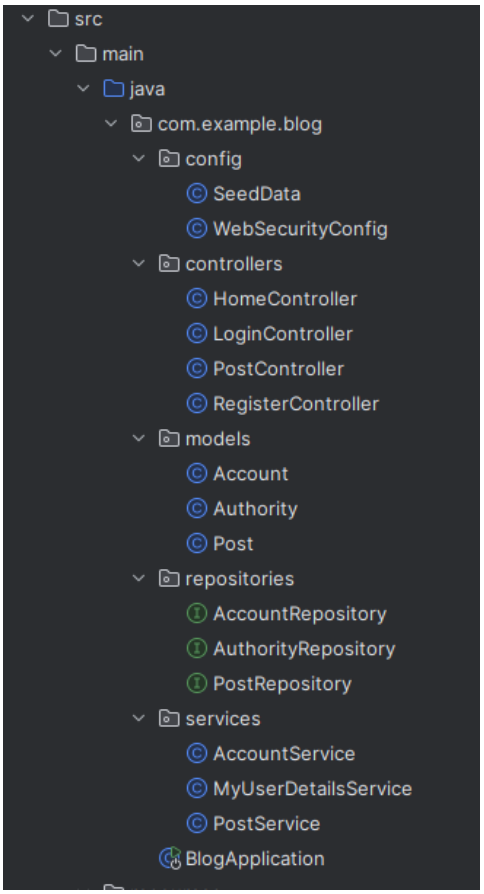


Рис. 3.3 Дерево каталогів проєкту

Ключові програмні компоненти розміщені у базовому пакунку com.example.blog та його спеціалізованих підпакунках:

– config: Містить класи, відповідальні за конфігураційні аспекти системи. Зокрема, SeedData забезпечує ініціалізацію бази даних початковим набором даних, а WebSecurityConfig визначає конфігурацію безпеки, включаючи правила доступу та механізми автентифікації.

– controllers: Реалізує шар представлення. Класи цього пакунку, такі як HomeController, PostController, RegisterController та LoginController, обробляють вхідні HTTP-запити.

– services: Інкапсулює бізнес-логіку. Сервіси, такі як PostService, AccountService, та MyUserDetailsService (для інтеграції зі Spring Security), виконують основні операції над даними та координують взаємодію з репозиторіями.

– repositories: Забезпечує доступ до даних. Інтерфейси репозиторіїв (PostRepository, AccountRepository, AuthorityRepository) використовують Spring Data JPA для взаємодії з базою даних.

– models: Визначає доменні сутності системи (Post, Account, Authority), які відображаються на таблиці бази даних.

Точкою входу для запуску всього застосунку є клас BlogApplication, анотований @SpringBootApplication.

3.3.3 Механізми взаємодії архітектурних компонентів

Послідовна та узгоджена взаємодія між шарами архітектури забезпечує коректну обробку користувацьких запитів та формування відповідей. Раніше представлена архітектурна схема системи, яка зображена на рисунку 3.2, також ілюструє основні потоки даних. Узагальнений цикл обробки запиту можна описати так:

1. Користувацька дія у веб-браузері ініціює HTTP-запит.
2. Контролер (Presentation Layer) приймає цей запит, аналізує його та передає необхідні дані відповідному сервісу.
3. Сервіс (Service Layer) виконує основну бізнес-логіку, за потреби звертаючись до одного або декількох репозиторіїв для операцій з даними.
4. Репозиторій (Repository Layer) взаємодіє з базою даних H2, виконуючи запити на читання або модифікацію даних.
5. Результат операцій повертається від репозиторію до сервісу, а потім до контролера.
6. Контролер формує HTTP-відповідь (HTML-сторінку) і надсилає її клієнту.

3.3.4 Аспекти конфігурації безпеки та ініціалізації даних

Забезпечення безпеки та належної підготовки середовища є критичними для функціонування будь-якого вебзастосунку. У розробленій системі ці задачі вирішуються наступними компонентами:

– Конфігурація безпеки, реалізована у класі WebSecurityConfig, визначає правила доступу до ресурсів, налаштовує процес автентифікації та авторизації користувачів, використовуючи можливості Spring Security.

– Сервіс MyUserDetailsService інтегрується з Spring Security для завантаження даних про користувачів (включаючи їхні ролі та повноваження) з бази даних під час автентифікації.

– Компонент SeedData відповідає за створення початкового набору даних при старті системи, що включає визначення базових ролей (ROLE_USER, ROLE_ADMIN) та, за потреби, тестових акаунтів і публікацій.

3.4 Опис ключових класів системи

Ключові класи проєкту, що відповідають за реалізацію доменної моделі, обробку користувацьких запитів та взаємодію з базою даних. Візуальне представлення структури класів та їхніх основних взаємозв'язків зображено на рисунку 3.4.

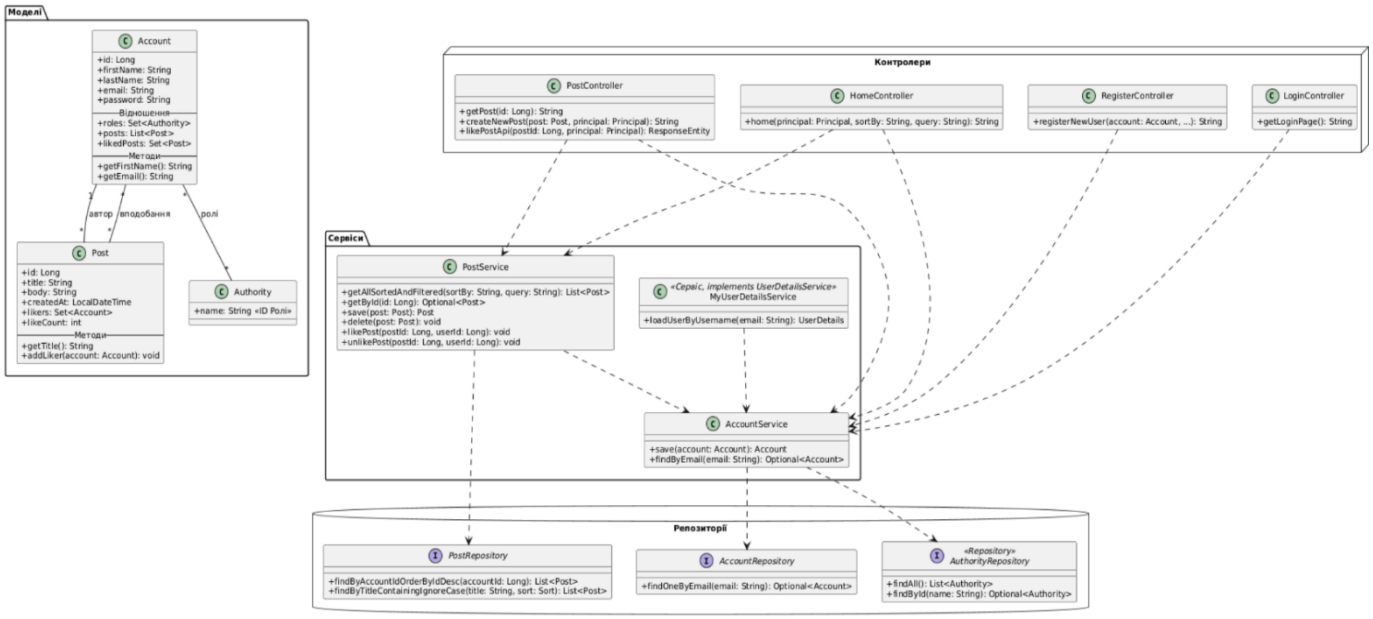


Рис. 3.4 UML-діаграма класів системи для керування блогами

3.4.1 Доменні сутності (Model)

Класи цього шару, розташовані в пакунку com.example.blog.models, визначають структуру даних, якими оперує система, та їхнє відображення на таблиці бази.

- Account: Представляє користувача системи. Містить поля id (унікальний ідентифікатор), firstName (псевдонім), lastName (ім'я), email (електронна пошта, використовується для автентифікації) та password (пароль, зберігається у хешованому вигляді).
- Зв'язок типу "багато-до-багатьох" з класом Authority визначає набір ролей користувача (ROLE_USER, ROLE_ADMIN), що використовуються для контролю доступу;
- Зв'язок типу "один-до-багатьох" з сутністю Post (через поле posts) відображає публікації, автором яких є даний користувач;
- Зв'язок типу "багато-до-багатьох" з сутністю Post (через поле likedPosts та відповідну проміжну таблицю post_like) реалізує механізм вподобання постів користувачами;
- Post: Центральна сутність, що представляє публікацію в блозі. Включає поля id, title (заголовок), body (основний текстовий вміст), createdAt (час створення) та modifiedAt (час останньої модифікації). Поле likedBy (тип Set<Account>) містить множину користувачів, які вподобали даний пост.
- Метод addLiker(Account account) дозволяє додати користувача до списку тих, хто вподобав пост, а removeLiker(Account account) - видалити;
- Поле likeCount, анотоване як @Formula("(select count(*) from post_like pl where pl.post_id = id)"), автоматично обчислює та надає актуальну кількість вподобань для поста безпосередньо з бази даних;
- Зв'язок типу "багато-до-одного" з сутністю Account (через поле account) однозначно визначає автора кожної публікації.
- Authority: Клас, що моделює роль користувача в системі. Містить єдине ключове поле name ("ROLE_USER", "ROLE_ADMIN"), яке слугує ідентифікатором

ролі. Сутності Authority використовуються Spring Security для визначення прав доступу до різних функцій та ресурсів системи.

3.4.2 Контролери (Controllers)

Контролери, розташовані в пакунку com.example.blog.controllers, є компонентами шару представлення та відповідають за обробку HTTP-запитів, що надходять від клієнта, делегування бізнес-логіки сервісному шару та формування відповіді.

- HomeController: Обробляє запити до головної сторінки. Метод home відповідає за відображення списку публікацій з урахуванням параметрів сортування (за датою, за кількістю вподобань) та пошукового запиту за назвою.
- PostController: Керує операціями, пов'язаними з публікаціями.
- Метод getPost забезпечує відображення сторінки перегляду окремого поста.
- Методи createNewPost та createNewPost обробляють створення нових постів.
- Виклики API для взаємодії з публікаціями likePostApi та unlikePostApi реалізують функціонал вподобання та скасування вподобання постів.
- RegisterController: Відповідає за процес реєстрації нових користувачів. Метод registerNewUser обробляє дані реєстраційної форми та ініціює створення нового облікового запису.
- LoginController: Метод getLoginPage() відображає сторінку автентифікації користувача.

Контролери взаємодіють із сервісним шаром для виконання бізнес-операцій та зазвичай повертають імена логічних представлень , які обробляються шаблонізатором (Thymeleaf) для генерації HTML-відповіді.

3.4.3 Сервіси (Services)

Сервісний шар, реалізований у класах пакунку com.example.blog.services, інкапсулює основну бізнес-логіку застосунку, забезпечуючи цілісність операцій та координацію взаємодії між контролерами та репозиторіями.

- PostService: Надає методи для управління публікаціями:
  - getAllSortedAndFiltered: Отримує список постів з можливістю сортування та фільтрації за пошуковим запитом.
  - getById(Long id): Здійснює пошук поста за його унікальним ідентифікатором.
  - save та delete: Виконують операції створення/оновлення та видалення постів.
  - likePost та unlikePost: Реалізують логіку додавання та видалення вподобань для постів.
- AccountService: Відповідає за бізнес-логіку, пов'язану з обліковими записами користувачів:
  - save: Забезпечує збереження нового або оновлення існуючого облікового запису, включаючи хешування пароля та перевірку унікальності email та псевдоніма.
  - findByEmail: Здійснює пошук користувача за його електронною поштою, що використовується, зокрема, під час автентифікації.
- MyUserDetailsService: Реалізація стандартного інтерфейсу Spring Security UserDetailsService. Метод loadUserByUsername(String email) завантажує дані користувача (включаючи його ролі) з бази даних за email та формує об'єкт UserDetails, необхідний для механізмів автентифікації та авторизації Spring Security.

3.4.4 Репозиторії (Repositories)

Пакунок com.example.blog.repositories містить інтерфейси, які розширюють JpaRepository від Spring Data JPA. Ці інтерфейси надають абстрактний шар для

взаємодії з базою даних, автоматично реалізуючись фреймворком для виконання CRUD-операцій та кастомних запитів.

- PostRepository: Надає методи для роботи з сутностями Post, включаючи findByAccountIdOrderByIdDesc для отримання постів конкретного автора та findByTitleContainingIgnoreCase для пошуку постів за фрагментом назви.
- AccountRepository: Забезпечує доступ до даних облікових записів, зокрема метод findOneByEmail для пошуку користувача за email.
- AuthorityRepository: Надає стандартні методи JpaRepository (findAll, findById(String name)) для управління сутностями Authority (ролями).

3.4.5 Зв’язки та залежності між компонентами

Діаграма класів та структури проєкту дозволяє виділити ключові залежності та потоки взаємодії між описаними компонентами:

- Контролери (Presentation Layer) використовують сервіси (Service Layer) для виконання бізнес-операцій. Ця залежність зазвичай реалізується через механізм впровадження залежностей (Dependency Injection).
- Сервіси, у свою чергу, оперують репозиторіями (Repository Layer) для здійснення доступу до даних та їх персистентності.
- Доменні моделі (Account, Post, Authority) є основними об'єктами передачі даних між шарами та використовуються контролерами (як параметри запитів або частини моделей для представлення), сервісами (для реалізації бізнес-логіки) та репозиторіями (для збереження та вибірки з бази даних).

Чітке розмежування функцій між шарами та компонентами сприяє прозорості архітектури, спрощує внесення змін і дозволяє розвивати окремі модулі, що в результаті полегшує підтримку та прискорює розгортання нових функцій.

4 ТЕСТУВАННЯ СИСТЕМИ

4.1 Важливість тестування

Однією з найважливіших фаз розробки програмного забезпечення є проведення тестів. Саме воно дозволяє переконатися в коректності реалізації функціональності, відповідності системи вимогам, стабільності роботи застосунку в різних умовах та на різних рівнях взаємодії. Навіть найбільш продумана система не може вважатися завершеною без належної перевірки її працездатності.

У випадку веб-застосунків тестування виконує такі основні завдання:

- виявлення логічних хиб у реалізації бізнес-правил;
- оцінка зручності й зрозумілості інтерфейсу з точки зору кінцевого користувача;
- перевірка коректності обробки та збереження даних у базі;
- аналіз реакції системи на неправильне або непередбачене введення;
- підтвердження того, що механізми контролю доступу працюють згідно з ролями користувачів.

Для забезпечення систематичного та всебічного підходу до перевірки функціональності розробленої системи для керування блогами було обрано метод мануального (ручного) тестування. Цей метод, хоч і вимагає безпосередньої участі тестувальника, дозволяє глибоко дослідити поведінку системи в реальних сценаріях використання, оцінити зручність користувацького інтерфейсу та гнучко реагувати на непередбачувані ситуації.

Ключовим інструментом для структурування процесу мануального тестування та документування його результатів є розробка тестових випадків (test case). Кожен тест-кейс формалізує окрему перевірку, визначаючи необхідні передумови, послідовність кроків для виконання, конкретні вхідні дані та очікуваний результат роботи системи.

4.2 Тестування основних функцій системи

Тест-кейси для перевірки найважливіших сценаріїв роботи платформи: реєстрації, входу, створення, редагування та видалення постів.

Таблиця 4.1

Тест кейс 1 реєстрації нового користувача

Елемент	Опис
Передумови	Відвідувач знаходиться на головній сторінці; база даних не містить дані про тестового користувача.
Вхідні дані	Псевдонім: testuser Ім’я: Валерій Email: test@example.com Пароль: Qwerty123!
Кроки тестування	1. Натиснути «Реєстрація» в навігаційному меню. 2. У формі ввести вхідні дані у відповідні поля. 3. Натиснути кнопку «Зареєструватися».
Очікуваний результат	Користувач спрямовується на сторінку входу до системи; в базі даних H2 з’являється новий запис у таблиці ACCOUNT із вказаним email.
Фактичний результат	Після натискання «Зареєструватися» відбулося перенаправлення на форму входу; в H2 виявлено новий рядок у таблиці ACCOUNT.
Статус	Успішно

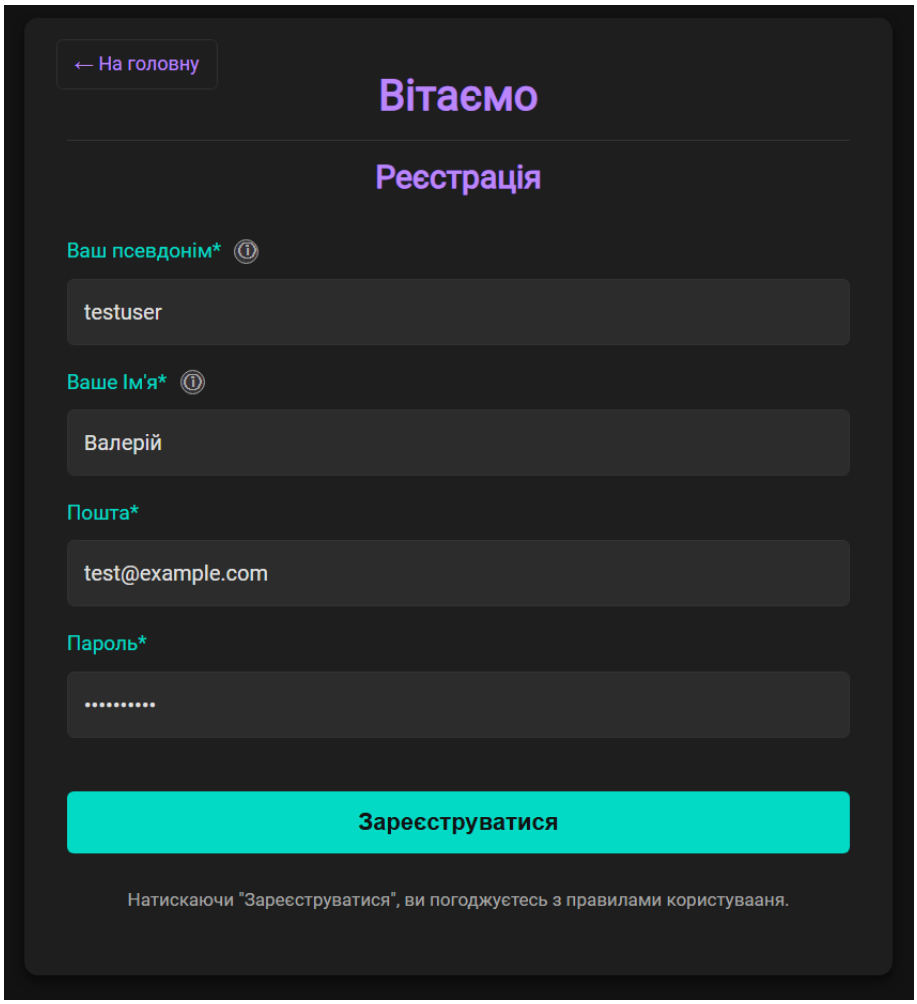


Рис. 4.1 Форма реєстрації користувача (заповнена)

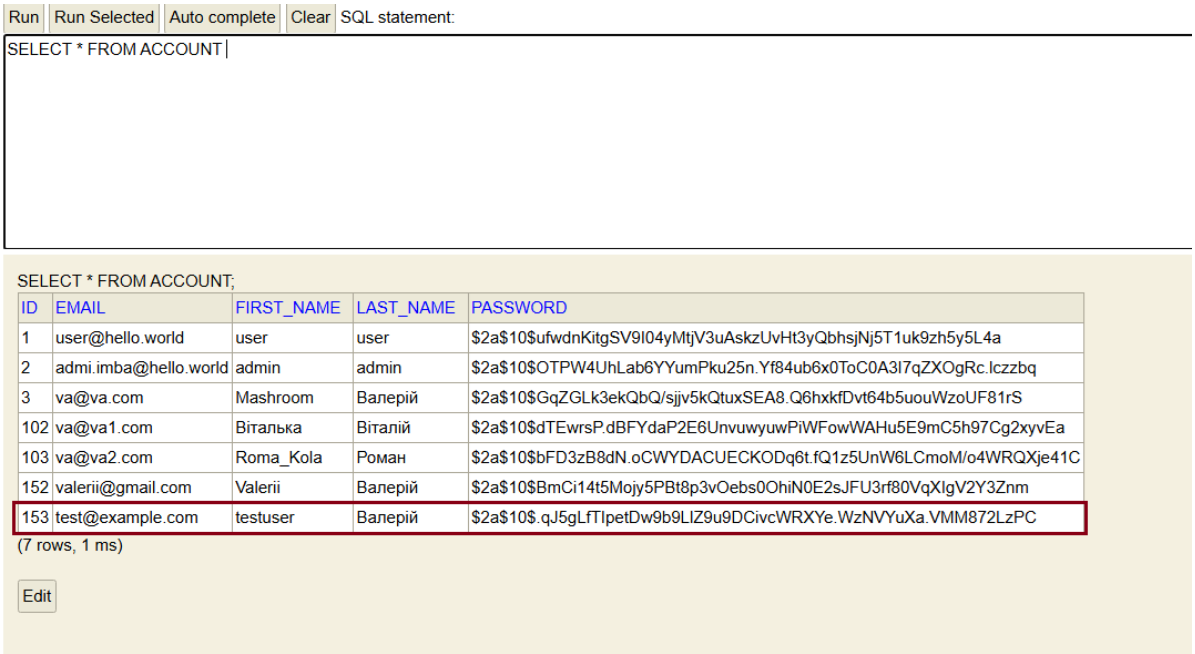


Рис. 4.2 Новий запис у таблиці ACCOUNT бази даних H2 після реєстрації

Таблиця 4.2

Тест кейс 2 Перевірка входу в систему

Елемент	Опис
Передумови	Запис про користувача з email test@example.com існує в таблиці ACCOUNT; користувач не авторизований.
Вхідні дані	Email: test@example.com Пароль: Qwerty123!
Кроки тестування	1. Натиснути «Увійти» в навігаційному меню. 2. Ввести email і пароль у відповідні поля. 3. Натиснути кнопку «Увійти».
Очікуваний результат	Користувач авторизується й спрямовується на головну сторінку з відображенням кнопки «Вихід» та можливістю створення постів.
Фактичний результат	Після введення даних і кліку «Увійти» відбулося перенаправлення на головну сторінку; у навігації з’явилася кнопка «Вихід» та опція «Написати блог».
Статус	Успішно

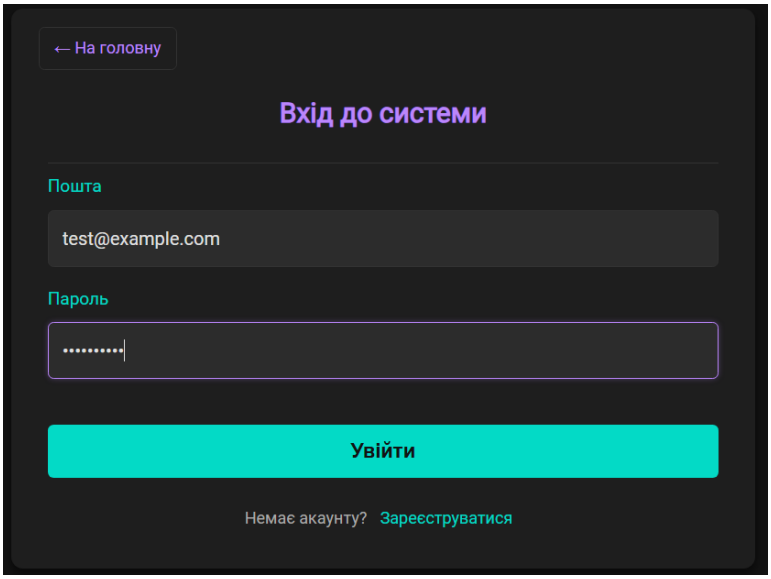


Рис. 4.3 Форма входу в систему

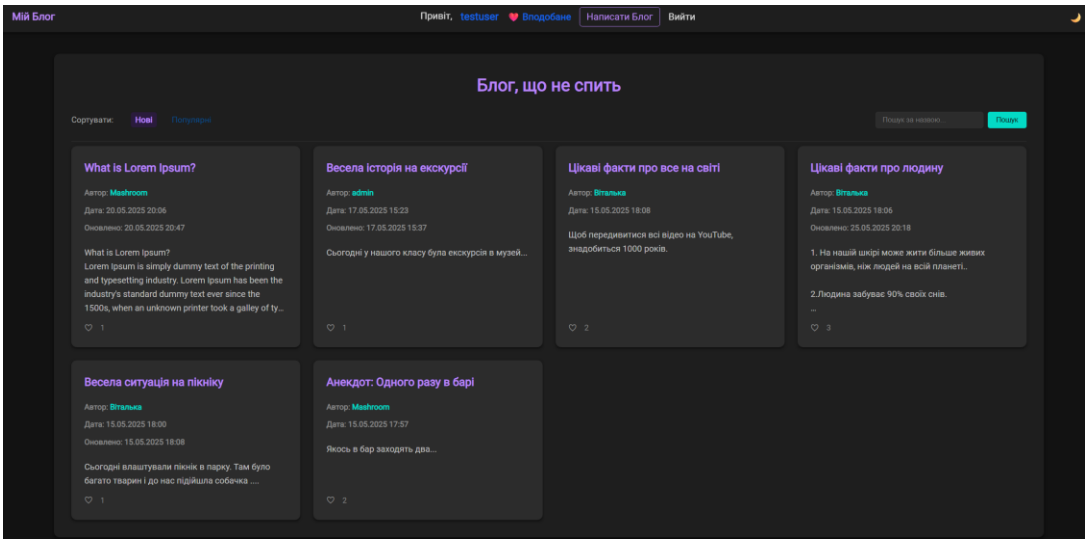


Рис.4.4 Успішний вхід до системи

Таблиця 4.3

Тест кейс 3 створення нового поста

Елемент	Опис
Передумови	Користувач авторизований
Вхідні дані	Заголовок: My First Blog Текст: Це тестовий допис.

Продовження таблиці 4.3

Тест кейс 3 створення нового поста

Елемент	Опис
Кроки тестування	1. Натиснути «Написати пост». 2. У формі ввести заголовок і текст.. 3. Натиснути «Опублікувати».
Очікуваний результат	Система зберігає пост і відображає його на головній сторінці;
Фактичний результат	Публікація створена, відображається на головній сторінці
Статус	Успішно

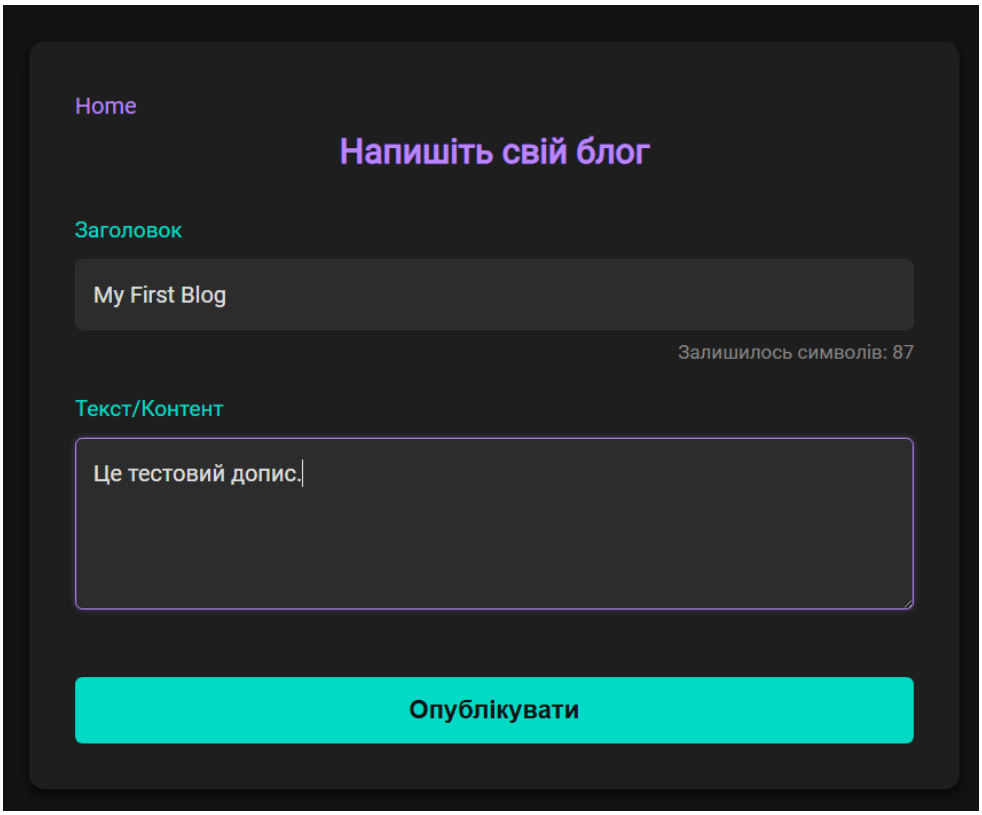


Рис. 4.5 Форма написання блогу

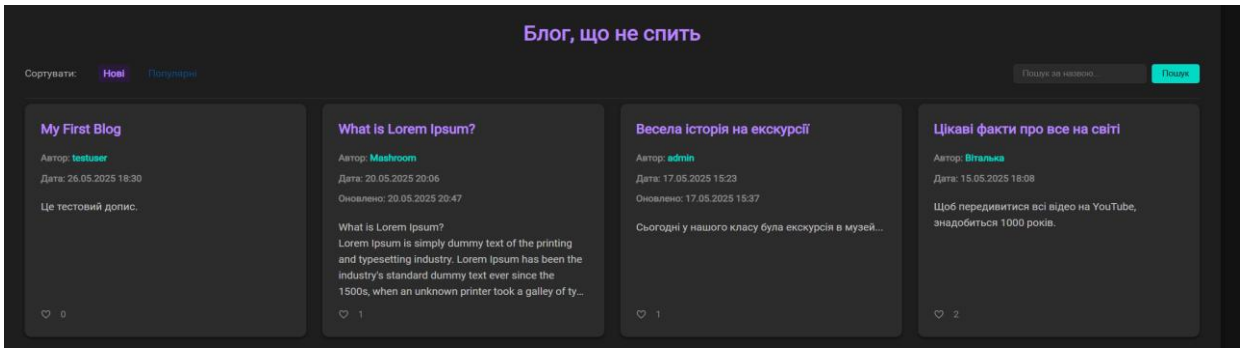


Рис. 4.6 Пост опублікувався та відображається на головній сторінці

Таблиця 4.4

Тест-кейс 4 редагування власного поста

Елемент	Опис
Передумови	Користувач авторизований; існує власний пост із заголовком My First Blog. Новий текст: Оновлений тестовий допис.
Кроки тестування	1. Перейти до сторінки списку власних постів. 2. Натиснути «Редагувати» біля публікації My First Blog. 3. Замінити заголовок і текст. 4. Зберегти.
Очікуваний результат	Система оновлює публікацію
Фактичний результат	Заголовок і вміст успішно змінені, відображення коректне.
Статус	Успішно

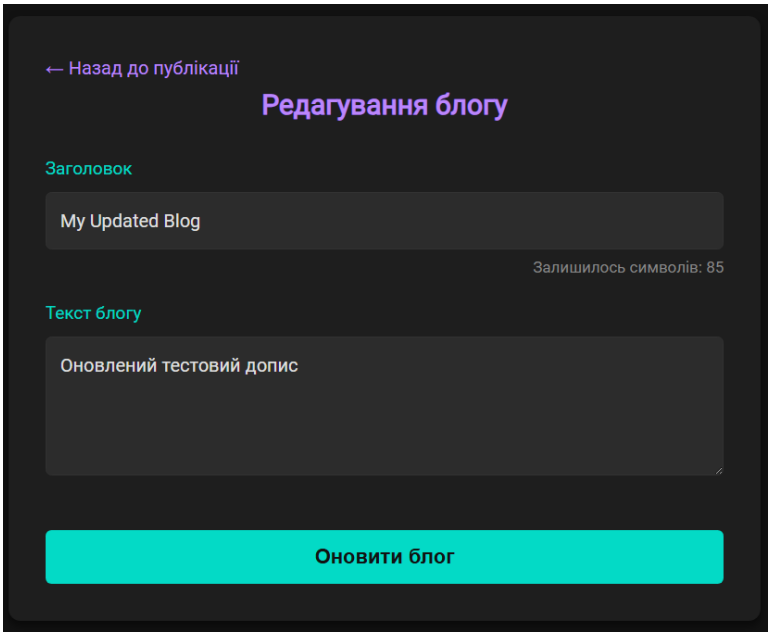


Рис. 4.7 Форма редагування поста

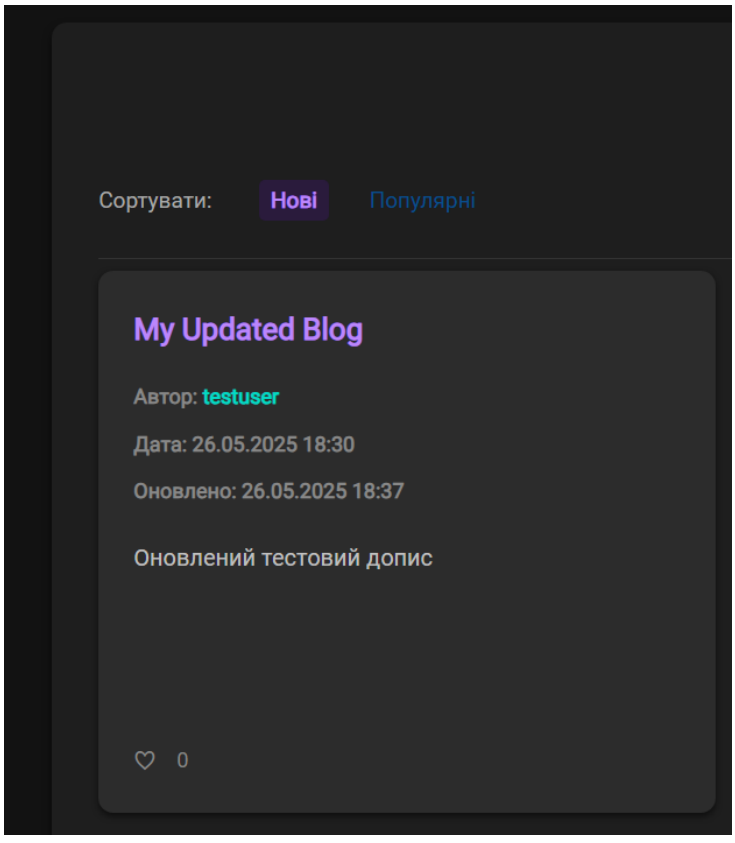


Рис. 4.8 Оновлений пост на головній сторінці

Таблиця 4.5

Тест-кейс 5 видалення власного поста

Елемент	Опис
Передумови	Користувач авторизований; існує власний пост із заголовком My Updated Blog
Вхідні дані	Заголовок: My First Blog Текст: Це тестовий допис.
Кроки тестування	1. Перейти на сторінку публікації. 2. Натиснути «Видалити»
Очікуваний результат	Запис видаляється з бази даних; на головній сторінці більше не відображається.
Фактичний результат	Публікацію успішно видалено, вона зникла зі списків.
Статус	Успішно

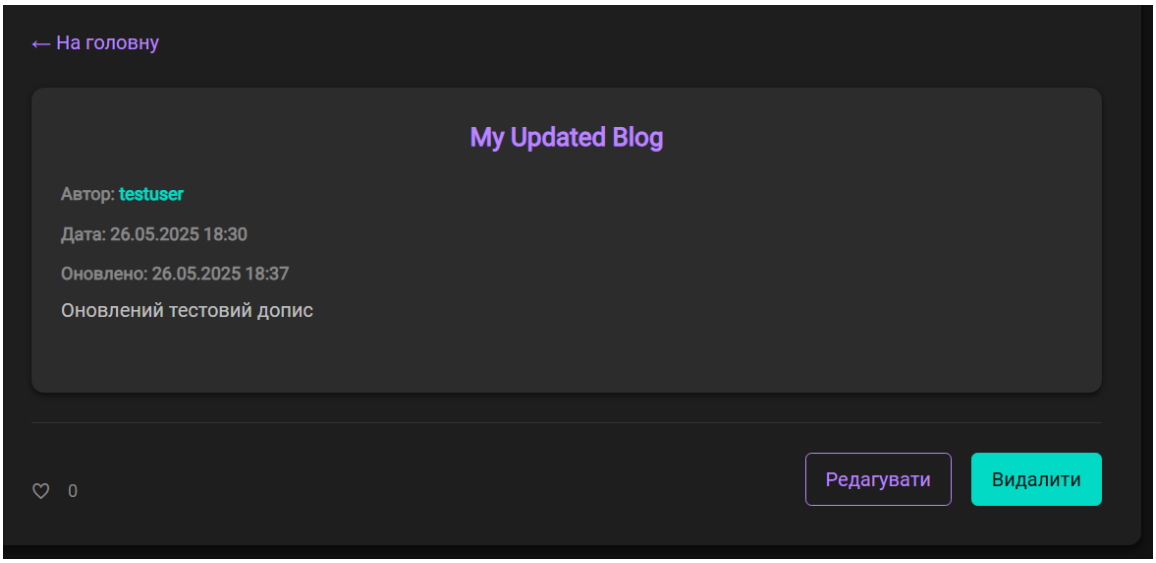


Рис. 4.9 Сторінка публікації та копка «Видалити»

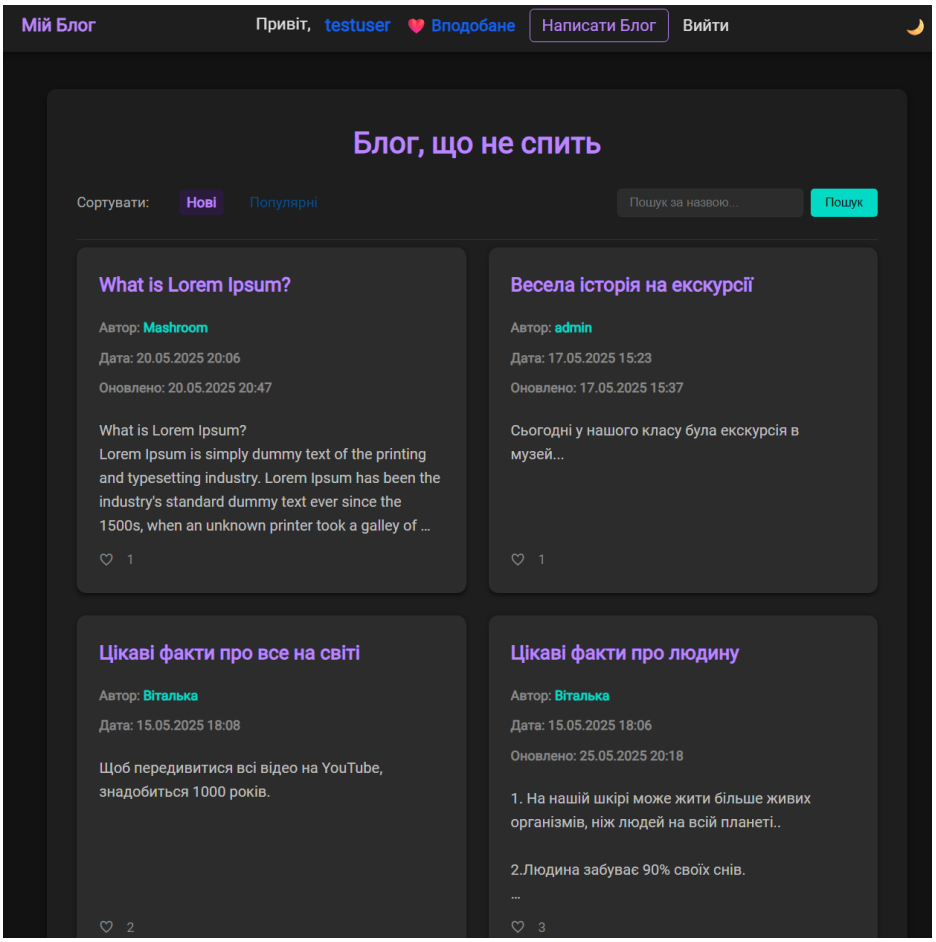


Рис 4.10 Запис видалено та не відображається на головній сторінці

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розглянуто низку важливих аспектів, що забезпечують повноцінну розробку та впровадження системи для керування блогами. Зокрема:

1. Досліджено ряд популярних блог-платформ (WordPress, Blogger, Medium, Ghost тощо) з метою виявлення їхніх сильних і слабких сторін. Визначено характерні функціональні можливості й обмеження, що дозволило сформувати чіткий орієнтир для власної розробки. Такий підхід дав змогу уникнути надлишкової складності та сконцентруватися на базових потребах користувачів.

2. Сформульовані вимоги до системи, на основі проведеного аналізу аналогів. З'ясовано, що функціональна частина охоплює ключові сценарії (автентифікація, CRUD-операції над постами, пошук, сортування, уподобання), а нефункціональні вимоги передбачають продуктивність, безпеку, сумісність із сучасними браузерами та зручність інтерфейсу.

3. Спроєктовано багатошарову архітектуру системи, що відповідає сучасним практикам розробки та дозволяє ізолювати компоненти, а також спрощує тестування й подальшу підтримку коду.

4. Реалізовано робочий прототип системи на базі Spring Boot, Spring Data JPA, Spring Security та Thymeleaf.

5. Проведено ручне тестування на основі тест-кейсів, що дало змогу перевірити коректність виконання основних функцій

Робота пройшла апробацію. За її результатами опубліковано тези:

1. Коляда В.С. Використання системи керування контентом блогу в освітньому середовищі. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 62-63.

2. Коляда В.С. Захист інформації у web-застосунку створення та керування блогами. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 286-287.

ПЕРЕЛІК ПОСИЛАНЬ

1. What is a blog? Definition, types, benefits and why you need one. *Wix*. URL: https://www.wix.com/blog/what-is-a-blog (дата звернення: 16.05.2025).
2. Usage statistics and market share of WordPress. *W3techs.com*. URL: https://w3techs.com/technologies/details/cm-wordpress (дата звернення: 16.05.2025).
3. Системи керування вмістом (CMS): порівняння популярних CMS, таких як WordPress, Drupal і Joomla. *Redstone*. URL: https://redstone.agency/blog/systemy-keruvannia-vmistom-cms-porivniannia-populiarnych-cms-takych-jak-wordpress-drupal-i-joomla/ (дата звернення: 15.05.2025).
4. The History of WordPress from 2003 – 2025 (with Screenshots). *Wpbeginner*. 04.03.2025. URL: https://www.wpbeginner.com/news/the-history-of-wordpress/ (дата звернення: 15.05.2025).
5. Usage statistics and market shares of content management systems. *W3techs*. URL: https://w3techs.com/technologies/overview/content_management (дата звернення: 15.05.2025).
6. The History of WordPress, its Ecosystem and Community. *Kinsta*. URL: https://kinsta.com/learn/wordpress-history/ (дата звернення: 15.05.2025).
7. History. *Wordpress*. URL: https://wordpress.org/about/history/ (дата звернення: 15.05.2025).
8. Олег Г. Переваги та недоліки CMS WordPress. *Esfirum*. URL: https://esfirum.com/blog/pros-and-cons-cms-wordpress/ (дата звернення: 15.05.2025).
9. Будував ідеальний медіасвіт, а вийшов Medium. Чому медіаутопія Ева Вільямса не спрацювала. *Vector*. URL: https://vctr.media/ua/istoriya-medium-149765/ (дата звернення: 21.05.2025).
10. 17 найкращих БЕЗКОШТОВНИХ блогів (2025). *Guru99*. URL: https://www.guru99.com/uk/best-free-blogging-platform.html (дата звернення: 16.05.2025).

11. Ghost CMS — платформа для створення лендінгів та блогів. *Komarov.design*. URL: https://www.komarov.design/ghost-cms-platforma-dlya-stvorennya-lendingiv-ta-blogiv/ (дата звернення: 16.05.2025).

12. Blogger review – everything about Google's blog platform. *Cybernews*. URL: https://cybernews.com/best-website-builders/blogger-review/ (дата звернення: 16.05.2025).

13. Minimalist Blogging Platforms for the Confused. *Magherallylens*. URL: https://www.magherallylens.com/minimalist-blogging-platforms/ (дата звернення: 16.05.2025).

14. Tumblr, Wordpress, Steemit and Medium: Pros & Cons.. *Medium*. URL: https://medium.com/@Love4aviation/tumblr-wordpress-steemit-and-medium-pros-cons-a0b41376d70d (дата звернення: 16.05.2025).

15. What is Java technology and why do I need it?. *Java*. URL: https://www.java.com/en/download/help/whatis_java.html (дата звернення: 24.05.2025).

16. JDK 22. *OpenJDK*. URL: https://openjdk.org/projects/jdk/22/ (дата звернення: 24.05.2025).

17. Redlich M. Java 22 Delivers Foreign Memory & Memory API, Unnamed Variables & Patterns, and Return of JavaOne. *InfoQ*. URL: https://www.infoq.com/news/2024/03/java22-released/ (дата звернення: 24.05.2025).

18. JDK 22: What is new in Java 22?. *Better LLMs for software development - Symflower*. URL: https://symflower.com/en/company/blog/2024/what-is-new-in-java-22/ (дата звернення: 24.05.2025).

19. Tyma P. Why are we using Java again?. *Communications of the ACM*. 1998. Т. 41, № 6. С. 38–42. URL: https://doi.org/10.1145/276609.276617 (дата звернення: 24.05.2025).

20. JVM optimization: An empirical analysis of JVM configurations for enhanced web application performance / D. Noetzold та ін. *SoftwareX*. 2024. Т. 28. С. 101933. URL: https://doi.org/10.1016/j.softx.2024.101933 (дата звернення:

24.05.2025).

21. Spring Boot Reference Documentation. *Spring*. URL: https://docs.spring.io/spring-boot/docs/3.2.5/reference/htmlsingle/ (дата звернення: 24.05.2025).

22. Code completion. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/auto-completing-code.html (дата звернення: 24.05.2025).

23. IntelliJ IDEA overview. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/discover-intellij-idea.html (дата звернення: 24.05.2025).

24. Code inspections. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/code-inspection.html (дата звернення: 24.05.2025).

25. Code refactoring. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/refactoring-source-code.html (дата звернення: 24.05.2025).

26. Spring. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/spring-support.html (дата звернення: 24.05.2025).

27. JetBrains Marketplace. *JetBrains Marketplace*. URL: https://plugins.jetbrains.com/ (дата звернення: 24.05.2025).

28. Debug code. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/debugging-code.html (дата звернення: 24.05.2025).

29. Connection to a database. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/connecting-to-a-database.html (дата звернення: 24.05.2025).

30. Git. *IntelliJ IDEA Help*. URL: https://www.jetbrains.com/help/idea/settings-version-control-git.html (дата звернення: 24.05.2025).

31. Spring Framework Overview. *Spring*. URL: https://docs.spring.io/spring-framework/reference/overview.html (дата звернення: 24.05.2025).

32. 17. Web MVC framework. *Spring | Home*. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html (дата звернення: 24.05.2025).

33. Thymeleaf. *Thymeleaf*. URL: https://www.thymeleaf.org/index.html (дата звернення: 24.05.2025).

34. Bonteanu A. M., Tudose C. Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA, Hibernate, Spring Data JPA. *Applied Sciences*. 2024. 42 с. URL: https://doi.org/10.3390/app14072743 (дата звернення: 24.05.2025).

35. Spring Security. *Spring*. URL: https://docs.spring.io/spring-security/reference/ (дата звернення: 24.05.2025).

36. Spring Boot with H2 Database. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/spring-boot-with-h2-database/ (дата звернення: 25.05.2025).

37. Sarpong R. N. Ultimate Responsive Web Design with HTML5 and CSS3: Create Visually Stunning, Responsive Websites Effortlessly with HTML5 and CSS3. Індія : Orange Education Pvt Ltd, 2024. – 324 с.

38. Dean J. Web Programming with HTML, CSS, and JavaScript. Jones & Bartlett Learning, 2025. – 722 с.

39. Types of UML Diagrams. *Lucidchart*. URL: https://www.lucidchart.com/blog/types-of-UML-diagrams (дата звернення: 25.05.2025).

40. Sheldon R. What is Unified Modeling Language (UML)?. *Techtarget*. URL: https://www.techtarget.com/searchsoftwarequality/definition/Unified-Modeling-Language (дата звернення: 25.05.2025).

41. Use Case. *Awork*. URL: https://www.awork.com/glossary/use-case (дата звернення: 25.05.2025).

42. GeeksforGeeks. Spring Boot - Architecture. *GeeksforGeeks*.

URL: https://www.geeksforgeeks.org/spring-boot-architecture/ (дата звернення: 26.05.2025).

43. Nature-inspired metaheuristic methods in software testing / N. Khoshniat та ін. *Soft Computing*. 2023. URL: https://doi.org/10.1007/s00500-023-08382-8 (дата звернення: 26.05.2025).

44. Developing a Web-Based System to Support the Operation of a Social Organisation / M. Stefanova та ін. *TEM Journal*. 2025. С. 695–706. URL: https://doi.org/10.18421/tem141-62 (дата звернення: 26.05.2025).

45. Heckler M. Spring Boot: Cloud-native Anwendungen mit Java und Kotlin erstellen. Dpunkt.Verlag GmbH, 2021.

46. Heckler M. Spring Boot: Up and Running. Сполучені Штати Америки : O'Reilly Media, 2021.

47. Puig F. M. Spring Boot 3.0 Cookbook: Proven Recipes for Building Modern and Robust Java Web Applications with Spring Boot. Німеччина : Packt Publishing.

48. Gutierrez F. Spring Data with Spring Boot. *Pro Spring Boot 3*. Berkeley, CA, 2024. С. 201–272. URL: https://doi.org/10.1007/978-1-4842-9294-5_5 (дата звернення: 26.05.2025).

49. Soni M. Advanced Java. Індія : Poorav Publications, 2024.

50. Blogging: The Ultimate Guide. Сполучені Штати Америки : Larsen and Keller Education, 2023.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення системи для керування блогами

Виконав студент 4 курсу
групи ПД-44
Коляда Валерій Сергійович
Керівник роботи
канд. фіз.-мат. наук, доцент, професор кафедри ІПЗ Садовенко Володимир Сергійович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: Спрощення процесу взаємодії між авторами та читачами шляхом розробки системи для керування блогами.

Об’єкт дослідження: Процес створення, публікації та взаємодії з контентом блогу.

Предмет дослідження: Засоби та технології розробки веб-застосунків на базі Spring Boot і H2.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Провести аналіз існуючих систем, визначити їх основні функції, переваги та недоліки.
2. Сформувати функціональні та нефункціональні вимоги до майбутньої системи.
3. Обрати відповідні технології та фреймворки для реалізації, спроектувати архітектуру застосунку відповідно до вимог.
4. Розробити програмне забезпечення з реалізацією основного функціоналу системи.
5. Провести тестування системи.

АНАЛІЗ АНАЛОГІВ

Функція	WordPress	Medium	Ghost	Blogger	Write.as	Svbtle	My Blog
Реєстрація	+	+	+	+	+	+	+
Публікація	+	+	+	+	+	+	+
Редагування	+	+	+	+	+	+	+
Вподобання	-	+	-	+	-	-	+
Пошук	+	+	+	-	-	-	+
Теми/Шаблони	+	-	+	+	-	-	-
Коментарі	+	+	-	+	-	-	-

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Реєстрація, вхід до системи та вихід.
2. Розмежування прав доступу (користувач, адміністратор).
3. Створення нових постів авторизованими користувачами.
4. Редагування та видалення власних постів(будь-яких адміністратором).
5. Відображення списку всіх постів на головній сторінці.
6. Перегляд детальної інформації окремого поста.
7. Можливість ставити й знімати "лайки" у авторизованих користувачів.
8. Перегляд списку постів, які користувач вподобав.
9. Перегляд усіх постів певного автора.
10. Сортування постів за датою або кількістю лайків.
11. Пошук постів за заголовком.

Нефункціональні вимоги

1. Швидкодія інтерфейсу та обробки запитів - не більше 1 с на дію користувача.
2. Хешування паролів.
3. Коректна робота в браузерах Chrom та Opera.
4. Зручність використання: інтерактивні підказки та підтримка світлої/темної теми.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Spring Boot

IntelliJ IDEA

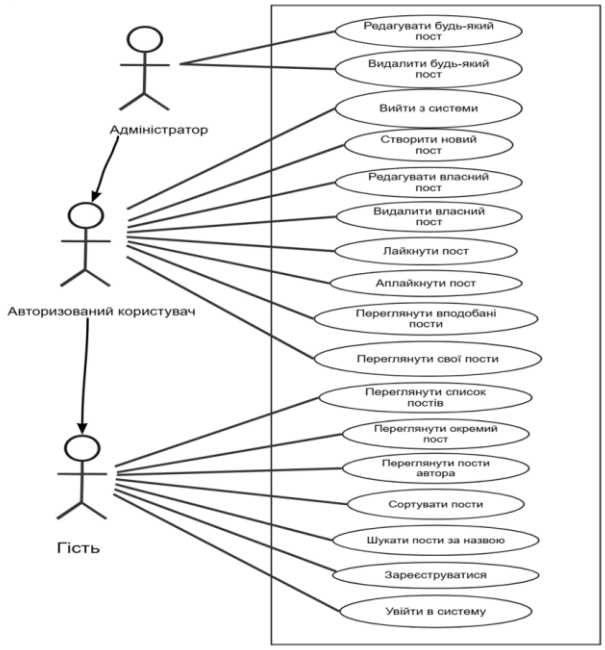
Spring Security

Spring Framework

H2 Database Engine

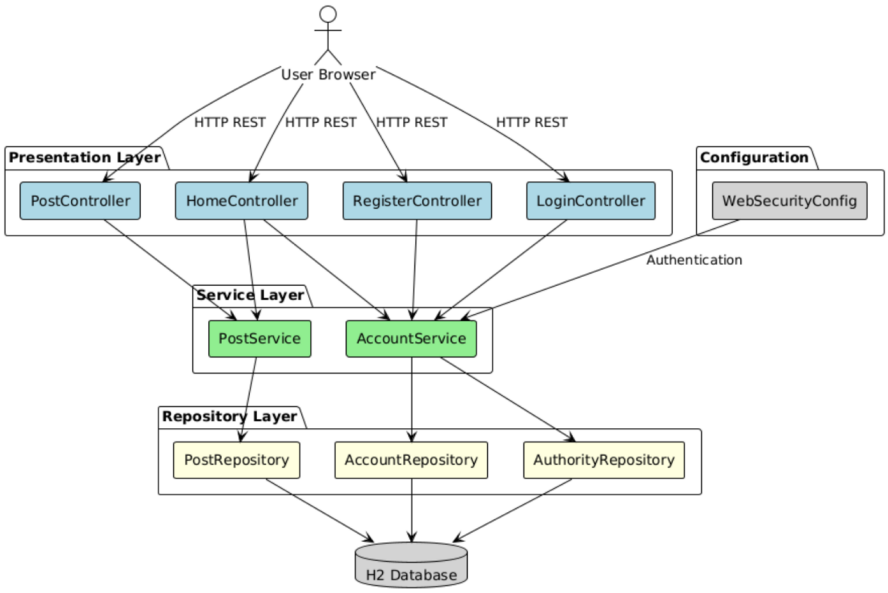
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



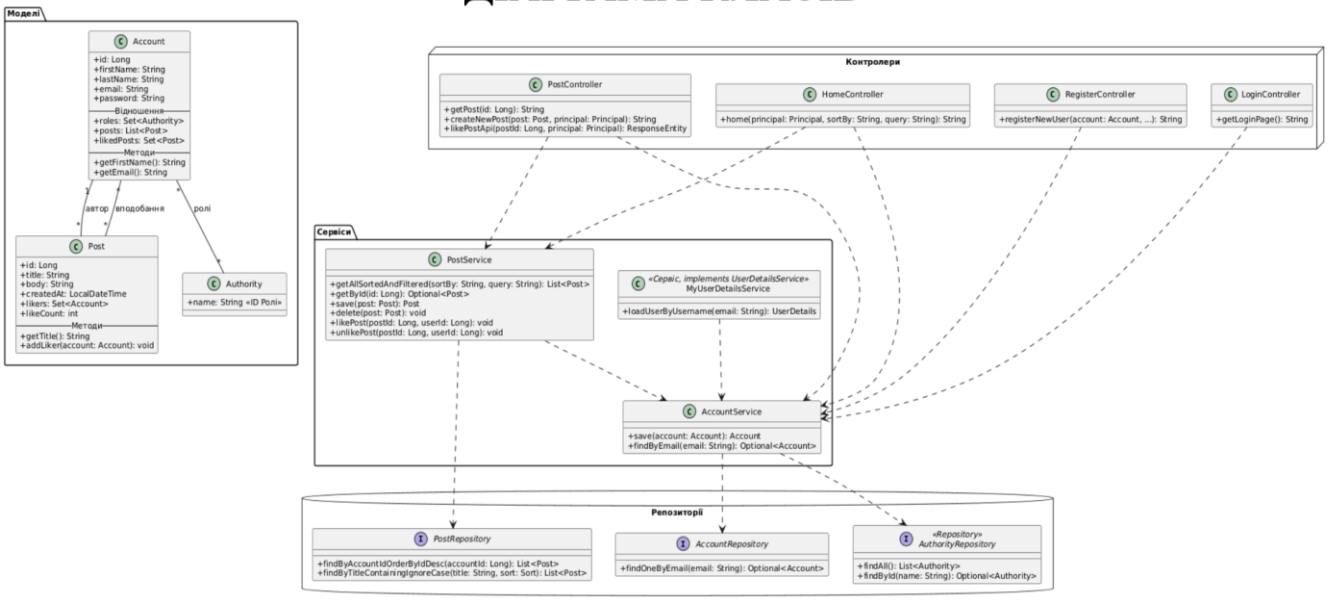
7

АРХІТЕКТУРА ПРОЄКТУ

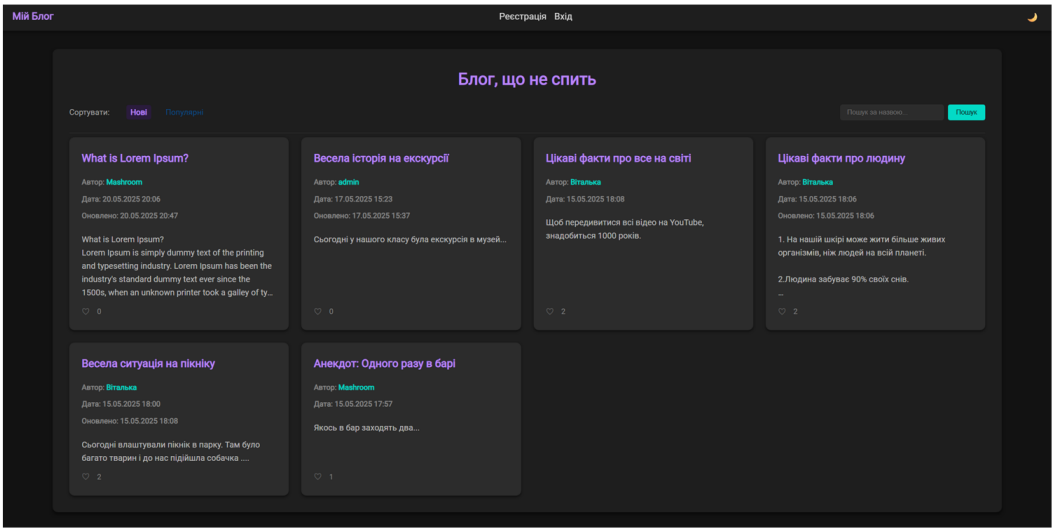


8

ДІАГРАМА КЛАСІВ

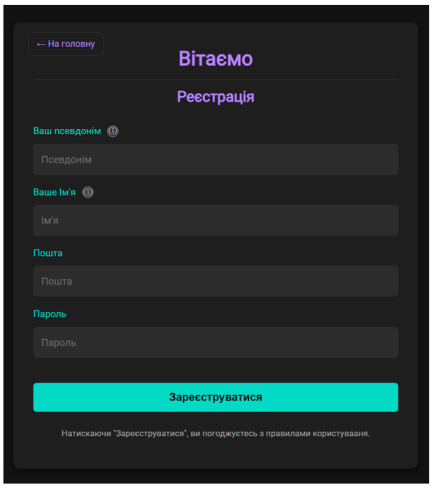


ЕКРАННІ ФОРМИ

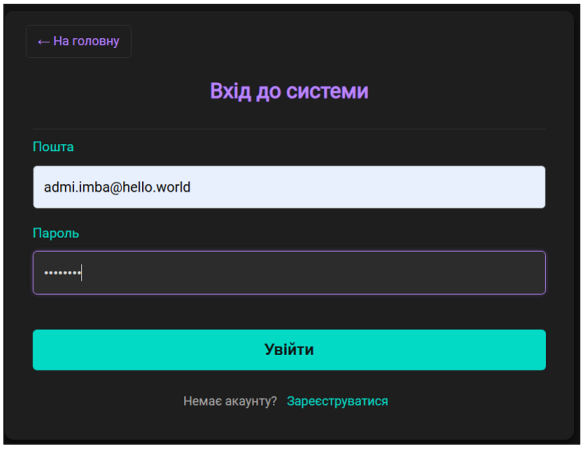


Головна сторінка

ЕКРАННІ ФОРМИ

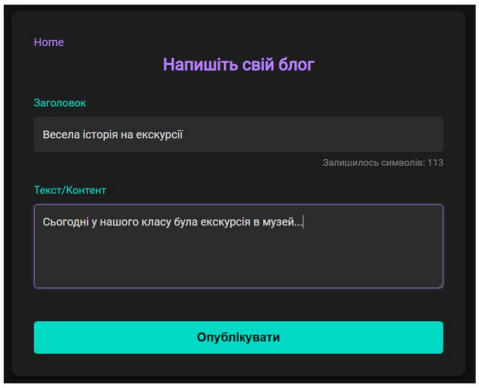


Реєстрація користувача

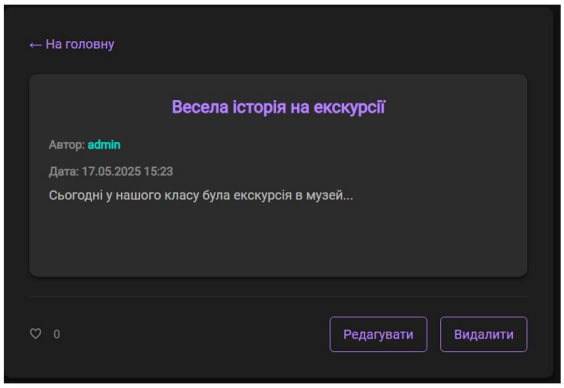


Вхід до системи

ЕКРАННІ ФОРМИ

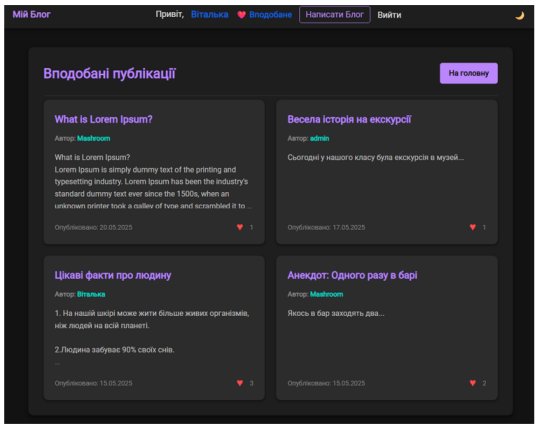


Написання блогу

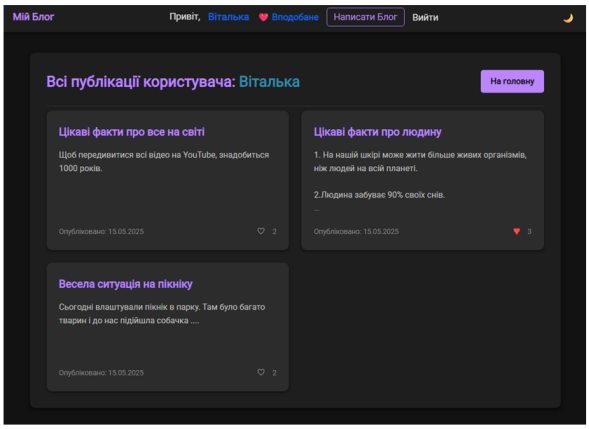


Сторінка блогу/посту

ЕКРАННІ ФОРМИ



Сторінка вподобаних публікацій



Публікації/блог конкретного автора

13

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Коляда В.С. Використання системи керування контентом блогу в освітньому середовищі. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 62-63.
2. Коляда В.С. Захист інформації у web-застосунку створення та керування блогами. Матеріали XII Всеукраїнської науково-практичної конференції молодих учених «Інформаційні технології – 2025», 15.05.25, КСУ імені Бориса Грінченка, Київ, Україна, С. 286-287.

14

ВИСНОВКИ

1. Досліджено відомі системи, визначено їх переваги, недоліки, а також характерні функціональні особливості. Це дало змогу окреслити орієнтири для реалізації власної системи керування блогами, уникнувши надмірної складності та зосередившись на основному функціоналі.
2. За результатами аналізу сформовано функціональні та нефункціональні вимоги до програмного забезпечення, що стали основою для проєктування системи.
3. Визначено технологічний стек, реалізовано багаторівневу архітектуру,що відповідає сучасним стандартам розробки веб-систем.
4. Створено робочий прототип, який підтримує авторизацію, публікацію й перегляд дописів, систему вподобань, пошук, фільтрацію постів.
5. Функціонал реалізовано відповідно до визначених вимог і протестовано на коректність виконання.

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

```
package com.example.blog.config;


import
org.springframework.boot.autoconfigure.security.servlet.PathR
equest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.method.config
uration.EnableMethodSecurity;
import
org.springframework.security.config.annotation.web.builders.H
ttpSecurity;
import
org.springframework.security.config.annotation.web.configurat
ion.EnableWebSecurity;
import
org.springframework.security.config.annotation.web.configurer
s.AbstractHttpConfigurer;
import
org.springframework.security.config.annotation.web.configurer
s.HeadersConfigurer;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordE
ncoder;
import
org.springframework.security.crypto.password.PasswordEncod
er;
import org.springframework.security.web.SecurityFilterChain;

import static
org.springframework.security.web.util.matcher.AntPathReques
tMatcher.antMatcher;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
public class WebSecurityConfig {

   @Bean
   public static PasswordEncoder passwordEncoder() {
      return new BCryptPasswordEncoder();
   }

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http)
throws Exception {

      http
            .csrf(AbstractHttpConfigurer::disable)//
Відключення CSRF (для H2 консолі
            .headers(headers -> headers

.frameOptions(HeadersConfigurer.FrameOptionsConfig::disabl
e)
            )
            .authorizeHttpRequests(auth -> auth
                  .requestMatchers(
                        antMatcher("/css/**"),
                        antMatcher("/js/**"),
                        antMatcher("/fonts/**"),
                        antMatcher("/webjars/**"),
                        antMatcher("/"),
                        antMatcher("/rss/**"),
                        antMatcher("/register/**"),
                        antMatcher("/login"),
                        antMatcher("/h2-console/**"),
                        antMatcher("/swagger-ui/**"), //
                        antMatcher("/v3/api-docs/**"), //
                        antMatcher("/posts/**"),
                        antMatcher("/pomilka"),

                        antMatcher("/author/**"),

                        PathRequest.toH2Console() //
Альтернативний спосіб дозволити H2
                  ).permitAll()

                  .anyRequest().authenticated()
            )
            .formLogin(form -> form
                  .loginPage("/login")
                  .loginProcessingUrl("/login")
                  .usernameParameter("email")
                  .passwordParameter("password")
                  .defaultSuccessUrl("/", true)
                  .failureUrl("/login?error")
                  .permitAll()
            )
            .logout(logout -> logout
                  .logoutUrl("/logout")
                  .logoutSuccessUrl("/?logout")
                  .permitAll()
            );

      return http.build();
   }
}

package com.example.blog.controllers;

import com.example.blog.models.Account;
import com.example.blog.models.Post;
import com.example.blog.services.AccountService;
import com.example.blog.services.PostService;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.RequestParam;

import java.security.Principal;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;

@Controller
```

```
public class HomeController {

    @Autowired
    private PostService postService;

    @Autowired
    private AccountService accountService;


    @GetMapping("/")
    public String home(Model model, Principal principal,

                       @RequestParam(name = "sort", defaultValue =
"date", required = false) String sortBy,

                       @RequestParam(name = "query", required =
false) String query) {


        List<Post> posts =
postService.getAllSortedAndFiltered(sortBy, query);


        Map<Long, Integer> likeCounts = new HashMap<>();
        Map<Long, Boolean> userLikedStatuses = new
HashMap<>();
        Account currentUser = null;
        Long currentUserId = null;
        if (principal != null) {
            Optional<Account> optCurrentUser =
accountService.findByEmail(principal.getName());
            if (optCurrentUser.isPresent()) {
                currentUser = optCurrentUser.get();
                currentUserId = currentUser.getId();
                model.addAttribute("currentUserAccount",
currentUser);
            }
        }
        for (Post post : posts) {
            likeCounts.put(post.getId(), post.getLikeCount());
            boolean liked = false;
            if (currentUserId != null) {
                liked = postService.hasUserLikedPost(post.getId(),
currentUserId);
            }
            userLikedStatuses.put(post.getId(), liked);
        }


        model.addAttribute("posts", posts);
        model.addAttribute("likeCounts", likeCounts);
        model.addAttribute("userLikedStatuses",
userLikedStatuses);
        model.addAttribute("currentSort", sortBy);

        model.addAttribute("searchQuery", query);

        return "home";
    }
}

package com.example.blog.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class LoginController {

    @GetMapping("/login")

    public String getLoginPage() {

        return "login";
    }
}

package com.example.blog.controllers;

import com.example.blog.models.Account;
import com.example.blog.models.Post;
import com.example.blog.services.AccountService;
import com.example.blog.services.PostService;
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import io.swagger.v3.oas.annotations.responses.ApiResponses;
import jakarta.persistence.EntityNotFoundException;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.core.Authentication;
import
org.springframework.security.core.context.SecurityContextHol
der;
import
org.springframework.security.core.authority.SimpleGrantedAut
hority;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.*;

import java.security.Principal;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;


@Controller
public class PostController {

    @Autowired
    private PostService postService;

    @Autowired
    private AccountService accountService;


    @PostMapping("/api/posts/{postId}/like")
    @PreAuthorize("isAuthenticated()")
    @ResponseBody
    public ResponseEntity<?> likePostApi(@PathVariable Long
postId, Principal principal) {
        Map<String, Object> response = new HashMap<>();
        try {

            Account currentUser =
accountService.findByEmail(principal.getName())
                    .orElseThrow(() -> new
EntityNotFoundException("Current user not found"));
            Long userId = currentUser.getId();
```

```
            postService.likePost(postId, userId);


            int newLikeCount = postService.getLikeCount(postId);

            response.put("success", true);
            response.put("likeCount", newLikeCount);
            response.put("userLiked", true);
            return ResponseEntity.ok(response);

        } catch (EntityNotFoundException e) {
            response.put("success", false);
            response.put("error", e.getMessage());
            return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(respons
e);
        } catch (Exception e) {

            response.put("success", false);
            response.put("error", "An unexpected error occurred: "
+ e.getMessage());
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERR
OR).body(response);
        }
    }


    @PostMapping("/api/posts/{postId}/unlike")
    @PreAuthorize("isAuthenticated()")
    @ResponseBody
    public ResponseEntity<?> unlikePostApi(@PathVariable
Long postId, Principal principal) {
        Map<String, Object> response = new HashMap<>();
        try {
            Account currentUser =
accountService.findByEmail(principal.getName())
                    .orElseThrow(() -> new
EntityNotFoundException("Current user not found"));
            Long userId = currentUser.getId();

            postService.unlikePost(postId, userId);

            int newLikeCount = postService.getLikeCount(postId);

            response.put("success", true);
            response.put("likeCount", newLikeCount);
            response.put("userLiked", false);
            return ResponseEntity.ok(response);

        } catch (EntityNotFoundException e) {
            response.put("success", false);
            response.put("error", e.getMessage());
            return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(respons
e);
        } catch (Exception e) {
            response.put("success", false);
            response.put("error", "An unexpected error occurred: "
+ e.getMessage());
            return
ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERR
OR).body(response);
        }
    }


    @GetMapping("/posts/liked")
    @PreAuthorize("isAuthenticated()")
    public String showLikedPosts(Model model, Principal
principal) {
        try {
            Account currentUser =
accountService.findByEmail(principal.getName())
                    .orElseThrow(() -> new
EntityNotFoundException("Current user not found"));
            Long userId = currentUser.getId();

            List<Post> likedPosts =
postService.getLikedPostsByUser(userId);


            Map<Long, Integer> likeCounts = new HashMap<>();
            Map<Long, Boolean> userLikedStatuses = new
HashMap<>();

            for (Post post : likedPosts) {
                likeCounts.put(post.getId(),
postService.getLikeCount(post.getId()));

                userLikedStatuses.put(post.getId(), true);
            }


            model.addAttribute("posts", likedPosts);
            model.addAttribute("likeCounts", likeCounts);
            model.addAttribute("userLikedStatuses",
userLikedStatuses);
            model.addAttribute("currentUserAccount",
currentUser);
            model.addAttribute("pageTitle", "Вподобані
публікації");

            return "liked_posts";

        } catch (EntityNotFoundException e) {

            return "pomilka";
        }
    }


    @GetMapping("/author/{authorId}")


    public String getPostsByAuthor(@PathVariable Long
authorId, Model model, Principal principal) {
        Optional<Account> optionalAuthor =
accountService.findById(authorId);
        if (optionalAuthor.isEmpty()) { /* ... обробка помилки ...
*/ return "pomilka"; }

        Account author = optionalAuthor.get();
        List<Post> posts =
postService.findPostsByAuthorId(authorId);


        Map<Long, Integer> likeCounts = new HashMap<>();
        Map<Long, Boolean> userLikedStatuses = new
HashMap<>();
        Account currentUser = null;
        Long currentUserId = null;

        if (principal != null) {
            Optional<Account> optCurrentUser =
accountService.findByEmail(principal.getName());
            if (optCurrentUser.isPresent()) {
```

```
            currentUser = optCurrentUser.get();
            currentUserId = currentUser.getId();
            model.addAttribute("currentUserAccount",
currentUser);
         }
      }

      for (Post post : posts) {
         likeCounts.put(post.getId(),
postService.getLikeCount(post.getId()));
         boolean liked = false;
         if (currentUserId != null) {
            liked = postService.hasUserLikedPost(post.getId(),
currentUserId);
         }
         userLikedStatuses.put(post.getId(), liked);
      }


      model.addAttribute("author", author);
      model.addAttribute("posts", posts);
      model.addAttribute("likeCounts", likeCounts);
      model.addAttribute("userLikedStatuses",
userLikedStatuses);

      return "author_posts";
   }



   @GetMapping("/posts/{id}")

   public String getPost(@PathVariable Long id, Model model,
Principal principal) {
      Optional<Post> optionalPost =
this.postService.getById(id);

      if (optionalPost.isPresent()) {
         Post post = optionalPost.get();
         model.addAttribute("post", post);

         boolean isOwner = false;
         boolean isAdmin = false;
         boolean userLiked = false;
         Account currentUser = null;


         if (principal != null) {
            String username = principal.getName();
            Optional<Account> optCurrentUser =
accountService.findByEmail(username);
            if (optCurrentUser.isPresent()) {
               currentUser = optCurrentUser.get();
               model.addAttribute("currentUserAccount",
currentUser);


               if (post.getAccount() != null &&
post.getAccount().getId().equals(currentUser.getId())) {
                  isOwner = true;
               }

               userLiked =
postService.hasUserLikedPost(post.getId(),
currentUser.getId());
            }


            Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();
            if (authentication != null &&
authentication.getAuthorities().contains(new
SimpleGrantedAuthority("ROLE_ADMIN"))) {
               isAdmin = true;
            }
         }


         model.addAttribute("canEditDelete", isAdmin ||
isOwner);
         model.addAttribute("likeCount",
postService.getLikeCount(post.getId()));
         model.addAttribute("userLiked", userLiked);

         return "post";
      } else {
         return "pomilka";
      }
   }


   @GetMapping("/posts/create")
   @PreAuthorize("isAuthenticated()")
   public String createNewPost(Model model, Principal
principal) {
      String authUsername = "anonymousUser";
      if (principal != null) {
         authUsername = principal.getName();
      }
      Optional<Account> optionalAccount =
accountService.findByEmail(authUsername);
      if (optionalAccount.isPresent()) {
         Post post = new Post();
         post.setAccount(optionalAccount.get());
         model.addAttribute("post", post);
         return "post_create";
      } else {
         return "pomilka";
      }
   }

   @PostMapping("/posts/create")
   @PreAuthorize("isAuthenticated()")

   public String createNewPost(@ModelAttribute Post post,
BindingResult result, Principal principal) {
      if (result.hasErrors()) {
         return "post_create";
      }
      String authUsername = principal.getName();
      Optional<Account> optionalAccount =
accountService.findByEmail(authUsername);
      if (optionalAccount.isPresent()) {
         if(post.getAccount() == null ||
!post.getAccount().getId().equals(optionalAccount.get().getId()
)) {
            post.setAccount(optionalAccount.get());
         }
      } else {
         return "pomilka";
      }
      postService.save(post);
      return "redirect:/posts/" + post.getId();
   }

   @GetMapping("/posts/{id}/edit")
   @PreAuthorize("isAuthenticated()")
```

```
    public String getPostForEdit(@PathVariable Long id, Model
model, Principal principal) {
        String authUsername = principal.getName();
        Optional<Post> optionalPost = postService.getById(id);
        if (optionalPost.isPresent()) {
            Post post = optionalPost.get();
            boolean isAdmin =
SecurityContextHolder.getContext().getAuthentication().getAu
thorities().stream()
                    .anyMatch(a ->
a.getAuthority().equals("ROLE_ADMIN"));
            if (post.getAccount() != null &&
(post.getAccount().getEmail().equalsIgnoreCase(authUsername
) || isAdmin)) {
                model.addAttribute("post", post);
                return "post_edit";
            } else { return "pomilka"; }
        } else { return "pomilka"; }
    }

    @PostMapping("/posts/{id}")
    @PreAuthorize("isAuthenticated()")
    public String updatePost(@PathVariable Long id,
@ModelAttribute Post post, BindingResult result, Model
model, Principal principal) {
        if (result.hasErrors()) {
            post.setId(id); model.addAttribute("post", post); return
"post_edit";
        }
        String authUsername = principal.getName();
        Optional<Post> optionalExistingPost =
postService.getById(id);
        if (optionalExistingPost.isPresent()) {
            Post existingPost = optionalExistingPost.get();
            boolean isAdmin =
SecurityContextHolder.getContext().getAuthentication().getAu
thorities().stream()
                    .anyMatch(a ->
a.getAuthority().equals("ROLE_ADMIN"));
            if (existingPost.getAccount() != null &&
(existingPost.getAccount().getEmail().equalsIgnoreCase(authU
sername) || isAdmin)) {
                existingPost.setTitle(post.getTitle());
                existingPost.setBody(post.getBody());
                postService.save(existingPost);
                return "redirect:/posts/" + existingPost.getId();
            } else { return "pomilka"; }
        } else { return "pomilka"; }
    }
    @GetMapping("/posts/{id}/delete")
    @PreAuthorize("isAuthenticated()")

    public String deletePost(@PathVariable Long id, Principal
principal) {
        String authUsername = principal.getName();
        Optional<Post> optionalPost = postService.getById(id);
        if (optionalPost.isPresent()) {
            Post post = optionalPost.get();
            boolean isAdmin =
SecurityContextHolder.getContext().getAuthentication().getAu
thorities().stream()
                    .anyMatch(a ->
a.getAuthority().equals("ROLE_ADMIN"));
            if (post.getAccount() != null &&
(post.getAccount().getEmail().equalsIgnoreCase(authUsername
) || isAdmin)) {
                postService.delete(post);
                return "redirect:/";
            } else { return "pomilka"; }
        } else { return "pomilka"; }
    }

}
```

```
package com.example.blog.controllers;

import com.example.blog.models.Account;
import com.example.blog.services.AccountService;
import jakarta.validation.Valid;
import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.dao.DataIntegrityViolationException;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.GetMapping;
import
org.springframework.web.bind.annotation.ModelAttribute;
import
org.springframework.web.bind.annotation.PostMapping;

@Controller
public class RegisterController {

    @Autowired
    private AccountService accountService;

    @GetMapping("/register")
    public String getRegisterPage(Model model) {
        if (!model.containsAttribute("account")) {
            model.addAttribute("account", new Account());
        }
        return "register";
    }

    @PostMapping("/register")
    public String registerNewUser(@Valid
@ModelAttribute("account") Account account,
                                  BindingResult bindingResult,
                                  Model model) {

        if (bindingResult.hasErrors()) {
            return "register";
        }

        try {
            accountService.save(account);
            return "redirect:/login?registered";
        } catch (IllegalArgumentException e) {
            String message = e.getMessage();
            if (message != null) {
                if
(message.startsWith(AccountService.MSG_PREFIX_EMAIL_
EXISTS)) {
                    String userMessage =
message.substring(AccountService.MSG_PREFIX_EMAIL_E
XISTS.length());
                    bindingResult.rejectValue("email",
"error.account.email", userMessage);
                } else if
(message.startsWith(AccountService.MSG_PREFIX_NICKNA
ME_EXISTS)) {
                    String userMessage =
message.substring(AccountService.MSG_PREFIX_NICKNA
ME_EXISTS.length());
                    bindingResult.rejectValue("firstName",
"error.account.firstName", userMessage);
```

```
            } else {

                model.addAttribute("registrationError", "Невідома
помилка валідації.");
            }
        } else {
            model.addAttribute("registrationError", "Сталася
помилка під час реєстрації.");
        }
        return "register";
    } catch (DataIntegrityViolationException e) {
        String rootCauseMessage =
e.getMostSpecificCause().getMessage().toLowerCase();
        boolean errorProcessed = false;

        if (rootCauseMessage.contains("account_email_key") ||
rootCauseMessage.contains("constraint violation for
public.account(email)")) {
            bindingResult.rejectValue("email", "error.db.email",
"Ця пошта вже зареєстрована (DB).");
            errorProcessed = true;
        }
        if
(rootCauseMessage.contains("account_firstname_key") ||
rootCauseMessage.contains("constraint violation for
public.account(firstname)")) {
            bindingResult.rejectValue("firstName",
"error.db.firstName", "Цей псевдонім вже використовується
(DB).");
            errorProcessed = true;
        }

        if (!errorProcessed) {
            model.addAttribute("registrationError", "Помилка
реєстрації: дані вже існують (DB).");
        }
        return "register";
    } catch (Exception e) {
        model.addAttribute("registrationError", "Сталася
неочікувана помилка.");
        return "register";
    }
  }
}
```

```
package com.example.blog.models;

import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.io.Serializable;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Account implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Long id;
  @NotBlank(message = "Пошта не може бути
порожньою")
  @Email(message = "Некоректний формат пошти")
  @Column(unique = true, nullable = false)
  private String email;

  @NotBlank(message = "Пароль не може бути порожнім")
  @Size(min = 8, message = "Пароль має містити
щонайменше 8 символів")
  private String password;

  @NotBlank(message = "Псевдонім не може бути
порожнім")
  @Size(min = 4, max = 20, message = "Псевдонім має
містити від 4 до 20 символів")
  @Column(unique = true, nullable = false) // Унікальність
для ніка (firstName)
  private String firstName;

  @NotBlank(message = "Ім'я не може бути порожнім")
  @Size(max = 20, message = "Ім'я має містити не більше
20 символів")
  private String lastName;

  // Звязок з постами написаними користувачем
  @OneToMany(mappedBy = "account", cascade =
CascadeType.ALL, orphanRemoval = true, fetch =
FetchType.LAZY)
  private List<Post> posts;


  @ManyToMany(fetch = FetchType.EAGER)
  @JoinTable(
      name = "user_authority",
      joinColumns = {@JoinColumn(name = "user_id",
referencedColumnName = "id")},
      inverseJoinColumns = {@JoinColumn(name =
"authority_name", referencedColumnName = "name")})
  private Set<Authority> authorities = new HashSet<>();



  @ManyToMany(mappedBy = "likedBy", fetch =
FetchType.LAZY)
  private Set<Post> likedPosts = new HashSet<>();



  @Override
  public String toString() {

    return "Account{" +
        "id=" + id +
        ", firstName='" + firstName + '\'' +
        ", lastName='" + lastName + '\'' +
        ", email='" + email + '\'' +
        '}';
  }



  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Account account = (Account) o;
    return id != null && id.equals(account.id);
  }
  @Override
  public int hashCode() {
```

```
        return id != null ? id.hashCode() :
System.identityHashCode(this);
    }
}

package com.example.blog.models;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.io.Serializable;

@Entity
@Setter
@Getter
@NoArgsConstructor
public class Authority implements Serializable {
    @Id
    @Column(length = 16)
    private String name;
    @Override
    public String toString() {
        return "Authority{" +
                "name='" + name + '\'' +
                '}';
    }
}

package com.example.blog.models;

import jakarta.persistence.*;
import jakarta.validation.constraints.NotNull;
import lombok.Getter;
import lombok.Setter;
import org.hibernate.annotations.Formula;

import java.time.LocalDateTime;
import java.util.HashSet;
import java.util.Set;

@Entity
@Getter
@Setter
public class Post {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
    private String title;
    @Column(columnDefinition = "TEXT")
    private String body;
    private LocalDateTime createdAt;
    private LocalDateTime modifiedAt;
    @NotNull
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "account_id",
referencedColumnName = "id", nullable = false)
    private Account account;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
            name = "post_like",
            joinColumns = @JoinColumn(name = "post_id"),
            inverseJoinColumns = @JoinColumn(name =
"account_id")
    )
    private Set<Account> likedBy = new HashSet<>();
    @Formula("(select count(*) from post_like pl where
pl.post_id = id)")
    private int likeCount;
    @Override
    public String toString() {
        return "Post{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", createdAt=" + createdAt +
                ", modifiedAt=" + modifiedAt +
                ", accountId=" + (account != null ? account.getId() :
null) +
                ", actualLikes=" + (likedBy != null ? likedBy.size() :
0) +
                ", calculatedLikeCount=" + likeCount +
                '}';
    }
    public void addLiker(Account account) {
        this.likedBy.add(account);
    }

    public void removeLiker(Account account) {
        this.likedBy.remove(account);
    }
}

package com.example.blog.repositories;

import com.example.blog.models.Account;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface AccountRepository extends
JpaRepository<Account, Long> {

    Optional<Account> findOneByEmail(String email);
    Optional<Account> findByFirstName(String firstName);
}

package com.example.blog.repositories;

import com.example.blog.models.Authority;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface AuthorityRepository extends
JpaRepository<Authority, String> {}

package com.example.blog.repositories;

import com.example.blog.models.Account;
import com.example.blog.models.Post;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Optional;
@Repository
public interface PostRepository extends JpaRepository<Post,
Long> {

    // для пошуку постів за автором
```

```
    List<Post> findByAccountIdOrderByIdDesc(Long
accountId);

    //  для лайків
    @Query("SELECT COUNT(p) > 0 FROM Post p JOIN
p.likedBy a WHERE p.id = :postId AND a.id = :accountId")
    boolean
existsByIdAndLikedByAccountId(@Param("postId") Long
postId, @Param("accountId") Long accountId);

    @Query("SELECT size(p.likedBy) FROM Post p WHERE
p.id = :postId")
    Integer findLikeCountByPostId(@Param("postId") Long
postId);

    @Query("SELECT p FROM Post p JOIN p.likedBy a
WHERE a.id = :accountId ORDER BY p.createdAt DESC")
    List<Post>
findPostsLikedByAccountId(@Param("accountId") Long
accountId);

    // пошук за назв
    List<Post> findByTitleContainingIgnoreCase(String title,
Sort sort);
}

package com.example.blog.services;

import com.example.blog.models.Account;
import com.example.blog.repositories.AccountRepository;
import
org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.crypto.password.PasswordEncod
er;
import org.springframework.stereotype.Service;
import
org.springframework.transaction.annotation.Transactional;
import java.util.Optional;
@Service
public class AccountService {

    @Autowired
    private AccountRepository accountRepository;

    @Autowired
    private PasswordEncoder passwordEncoder;

    public static final String MSG_PREFIX_EMAIL_EXISTS =
"ERROR_EMAIL_EXISTS:";
    public static final String
MSG_PREFIX_NICKNAME_EXISTS =
"ERROR_NICKNAME_EXISTS:";

    @Transactional
    public Account save(Account account) {
        if (account.getId() == null) {
            Optional<Account> existingByEmail =
accountRepository.findOneByEmail(account.getEmail());
            if (existingByEmail.isPresent()) {
                throw new
IllegalArgumentException(MSG_PREFIX_EMAIL_EXISTS +
"Пошта '" + account.getEmail() + "' вже використовується.");
            }

            Optional<Account> existingByFirstName =
accountRepository.findByFirstName(account.getFirstName());
            if (existingByFirstName.isPresent()) {
                throw new
IllegalArgumentException(MSG_PREFIX_NICKNAME_EXI
STS + "Псевдонім '" + account.getFirstName() + "' вже
використовується.");
            }

            if (account.getPassword() != null &&
!account.getPassword().isEmpty()) {

account.setPassword(passwordEncoder.encode(account.getPass
word()));
            }
        } else {
            Account existingAccountData =
accountRepository.findById(account.getId())
                    .orElseThrow(() -> new
RuntimeException("Акаунт для оновлення не знайдено: " +
account.getId()));

            if (account.getPassword() != null &&
!account.getPassword().isEmpty()) {
                if (!account.getPassword().startsWith("$2a$")) {

account.setPassword(passwordEncoder.encode(account.getPass
word()));
                }
            } else {

account.setPassword(existingAccountData.getPassword());
            }
        }
        return accountRepository.save(account);
    }

    @Transactional(readOnly = true)
    public Optional<Account> findByEmail(String email) {
        return accountRepository.findOneByEmail(email);
    }

    @Transactional(readOnly = true)
    public Optional<Account> findById(Long id) {
        return accountRepository.findById(id);
    }

    @Transactional(readOnly = true)
    public boolean existsById(Long id) {
        return accountRepository.existsById(id);
    }
}

package com.example.blog.services;

import com.example.blog.models.Account;
import lombok.RequiredArgsConstructor;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import
org.springframework.security.core.authority.SimpleGrantedAut
hority;
import org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsServi
ce;
import
org.springframework.security.core.userdetails.UsernameNotFo
undException;
import org.springframework.stereotype.Component;
```

```
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@Component("userDetailsService")
@RequiredArgsConstructor
public class MyUserDetailsService implements
UserDetailsService {
@Autowired
   private final AccountService accountService;
   @Override
   public UserDetails loadUserByUsername(String email)
throws UsernameNotFoundException {
      Optional<Account> optionalAccount =
accountService.findByEmail(email);
      if (!optionalAccount.isPresent()) {
         throw new UsernameNotFoundException("Акаунт не
знайдено");
      }
      Account account = optionalAccount.get();
      List<GrantedAuthority> grantedAuthorities = account
            .getAuthorities()
            .stream()
            .map(authority -> new
SimpleGrantedAuthority(authority.getName()))
            .collect(Collectors.toList());

      return new User(account.getEmail(),
account.getPassword(), grantedAuthorities);
   }
}

package com.example.blog.services;
import com.example.blog.models.Account;
import com.example.blog.models.Post;
import com.example.blog.repositories.PostRepository;
import jakarta.persistence.EntityNotFoundException;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import
org.springframework.transaction.annotation.Transactional;
import org.springframework.util.StringUtils;

import java.time.LocalDateTime;
import java.util.Collections;
import java.util.List;
import java.util.Optional;

@Service
public class PostService {

   @Autowired
   private PostRepository postRepository;

   @Autowired
   private AccountService accountService;


   public Optional<Post> getById(Long id) { return
postRepository.findById(id); }
   public Post save(Post post) { if (post.getId() == null) {
post.setCreatedAt(LocalDateTime.now()); }
post.setModifiedAt(LocalDateTime.now()); return
postRepository.save(post); }
   public void delete(Post post) { postRepository.delete(post); }
   public List<Post> findPostsByAuthorId(Long authorId) {
return
postRepository.findByAccountIdOrderByIdDesc(authorId); }
   @Transactional
   public void likePost(Long postId, Long userId) { Post post =
postRepository.findById(postId) .orElseThrow(() -> new
EntityNotFoundException("Post not found with id: " +
postId)); Account liker = accountService.findById(userId)
.orElseThrow(() -> new EntityNotFoundException("Account
not found with id: " + userId)); post.getLikedBy().add(liker); }
   @Transactional
   public void unlikePost(Long postId, Long userId) { Post post
= postRepository.findById(postId) .orElseThrow(() -> new
EntityNotFoundException("Post not found with id: " +
postId)); Account unliker = accountService.findById(userId)
.orElseThrow(() -> new EntityNotFoundException("Account
not found with id: " + userId));
post.getLikedBy().remove(unliker); }
   @Transactional(readOnly = true) public boolean
hasUserLikedPost(Long postId, Long userId) { if
(!postRepository.existsById(postId) ||
!accountService.existsById(userId)) { return false; } return
postRepository.existsByIdAndLikedByAccountId(postId,
userId); }
   @Transactional(readOnly = true) public int
getLikeCount(Long postId) { Integer count =
postRepository.findLikeCountByPostId(postId); return count
!= null ? count : 0; }
   @Transactional(readOnly = true) public List<Post>
getLikedPostsByUser(Long userId) { return
postRepository.findPostsLikedByAccountId(userId); }
   @Transactional(readOnly = true)
   public List<Post> getAllSortedAndFiltered(String sortBy,
String searchQuery) {
      Sort sort;

      if ("likes".equalsIgnoreCase(sortBy)) {
         sort = Sort.by(Sort.Direction.DESC, "likeCount",
"createdAt");
      } else {
         sort = Sort.by(Sort.Direction.DESC, "createdAt");
      }
      if (StringUtils.hasText(searchQuery)) {
         return
postRepository.findByTitleContainingIgnoreCase(searchQuery
.trim(), sort);
      } else {
         return postRepository.findAll(sort);
      }
   }


}
package com.example.blog;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;

@SpringBootApplication
public class BlogApplication {

   public static void main(String[] args) {
     SpringApplication.run(BlogApplication.class, args);
   }
```