МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ КОНТЕНТОМ ДЛЯ ВЕДЕННЯ БЛОГУ З ВИКОРИСТАННЯМ ANGULAR ТА NODE.JS»**

Виконав: студент 2 курсу, групи 8.1214-з
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
Г.Є. Жутовський
(ініціали та прізвище)

Керівник завідувач кафедри програмної інженерії, доцент, к.ф.-м.н. Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Жутовському Глібу Євгеновичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи управління контентом для ведення блогу з використанням Angular та Node.js

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 327-с

2. Строк подання студентом роботи 28.11.2025 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі, аналіз предметної області.
2. Проєктування.
3. Реалізація та тестування.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 14.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	28.03.2025	
2.	Збір вихідних даних.	18.04.2025	
3.	Обробка методичних та теоретичних	16.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	29.08.2025	
5.	Розробка третього розділу.	17.11.2025	
6.	Оформлення та нормоконтроль	24.11.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	12.12.2025	

Студент _____ Г.Є. Жутовський
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка системи управління контентом для ведення блогу з використанням Angular та Node.js»: 72 с., 42 рис., 1 табл., 20 джерел, 5 додатків.

ANGULAR, EXPRESS, NODE.JS, MODULE, MYSQL, REST API, RXJS, UML.

Об’єкт дослідження – системи управління контентом для ведення блогу.

Мета роботи – розробити систему управління контентом для ведення блогу з використанням Angular та Node.js.

Методи дослідження – аналітичний, моделювання, проєктування, програмний.

У роботі було розглянуто проєктування і реалізацію системи управління контентом для ведення блогу. Основний акцент зроблено на підтримці рольової моделі доступу, яка включає автора, читача та адміністратора, а також на забезпеченні мінімального порогу входження для будь-якого користувача, незалежно від його технічної підготовки. Архітектурно система побудована за клієнт-серверною моделлю, де взаємодія між компонентами здійснюється через REST API. Для моделювання структури та поведінки системи було розроблено набір UML-діаграм, включаючи діаграми варіантів використання, діяльності, послідовності та розгортання.

Клієнтська частина розроблена на Angular, що забезпечує реактивний та сучасний інтерфейс. Серверна частина виконана на Node.js із застосуванням фреймворку Express, а для зберігання даних використано MySQL. Інтерфейс користувача є простим та інтуїтивно зрозумілим за структурою, доповненою вбудованими підказками, що полегшує взаємодію.

У підсумку, розроблена система управління контентом не лише має зрозумілий та простий інтерфейс, що не вимагає від користувачів технічного досвіду, але і демонструє високу легкість масштабування та підтримки.

SUMMARY

Master’s qualifying paper “Development of a Content Management System for Blogging Using Angular and Node.js”: 72 pages, 42 figures, 1 table, 20 references, 5 supplements.

ANGULAR, EXPRESS, NODE.JS, MODULE, MYSQL, REST API, RXJS, UML.

The object of the study is content management systems for blogging.

The aim of the study is to develop a content management system for blogging using Angular and Node.js.

The methods of research are analytical, modeling, design, programming.

The paper considers the design and implementation of a content management system for blogging. The main focus is on supporting a role-based access model that includes the author, reader, and administrator, as well as ensuring a minimum entry threshold for any user, regardless of their technical training. Architecturally, the system is built on a client-server model, where interaction between components is carried out via REST API. To model the structure and behavior of the system, a set of UML diagrams was developed, including use case, activity, sequence, and deployment diagrams.

The client side is developed on Angular, which provides a responsive and modern interface. The server side is implemented on Node.js using the Express framework, and MySQL is used for data storage. The user interface is simple and intuitive in structure, complemented by built-in hints that facilitate interaction.

As a result, the developed content management system not only has a clear and simple interface that does not require technical experience from users, but also demonstrates high scalability and maintainability.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Технічне завдання та функціональні вимоги системи 10
1.1 Перелік умовних позначень, скорочень, термінів 10
1.2 Функціональні вимоги 11
1.2.1 Призначення і цілі створення системи 11
1.2.2 Загальні функціональні можливості системи 11
1.3 Нефункціональні вимоги 12
1.4 Опис предметної області 13
1.5 Опис системи 13
1.6 Огляд та порівняння аналогів 14
2 Проєктування 16
2.1 Використання UML під час розробки системи 16
2.2 Діаграма варіантів використання 17
2.2.1 Опис варіантів використання 20
2.3 Діаграма діяльності 29
2.4 Діаграма послідовності 31
2.5 Діаграма розгортання 33
3 Програмна реалізація та тестування 35
3.1 Опис інструментів розробки 35
3.2 Основні класи системи 35
3.3 Встановлення та налаштування оточення 37
3.4 Опис REST API 41
3.5 Тестування проєкту 43
3.6 Керівництво користувача 45

3.6.1 Вхід та реєстрація 45
3.6.2 Створення та редагування поста 48
3.6.3 Взаємодія з медіа..... 52
3.6.4 Пошук постів 53
Висновки 55
Перелік посилань..... 56
Додаток А Angular PostEditorComponent 58
Додаток Б Angular PostService 61
Додаток В Express Post Controller 62
Додаток Г Express Post Model 66
Додаток Д Unit-test PostEditorComponent 67

ВСТУП

Зі стрімким розвитком цифрових технологій та соціальних медіа, блоги стали невід'ємною частиною інформаціного простору. Вони є ключовим інструментом для обміну інформацією, просування ідей, налагодження комунікації з аудиторією та формування особистого або корпоративного бренду. Зростаюча потреба у швидкому та ефективному наповненні, редагуванні та публікації контенту вимагає наявності якісних та гнучких систем управління контентом.

Ринок вже має низку готових рішень для ведення блогів, але більшість з них часто є надто універсальними, що призводить до їхньої складності, перевантаженості функціоналом, що рідко використовується, або високої вартості ліцензування. Існуючі платформи можуть вимагати від користувачів певних технічних знань для налаштування, мати обмеження щодо кастомізації або мати високий поріг входження, особливо для авторів, чия основна увага має бути зосереджена на контенті, а не на технічних деталях.

Зважаючи на описані обмеження та постійний попит на зручні та спеціалізовані рішення, виникає обґрунтована необхідність у створенні легкої, адаптивної, безпечної та масштабованої CMS, орієнтованої саме на потреби блогінгу. Така система має мінімізувати технічний поріг входу, надаючи інтуїтивно зрозумілий інтерфейс для управління контентом, забезпечуючи при цьому високу швидкість роботи та надійність.

Актуальність роботи полягає у розробці сучасної, масштабованої та високопродуктивної системи управління контентом для ведення блогу, яка поєднує інтуїтивну простоту використання для кінцевих користувачів із потужністю та гнучкістю передових вебтехнологій.

З огляду на це, можна виділити наступні цілі і задачі дослідження.

Мета: розробити систему управління контентом для ведення блогу з використанням Angular та Node.js.

Досягнення поставленої мети передбачає реалізацію наступних завдань:

– сформулювати функціональні та нефункціональні вимоги до системи;
– проаналізувати та порівняти існуючі системи управління контентом;
– спроєктувати архітектуру системи, включаючи розробку необхідних UML-діаграм та REST API;
– реалізувати клієнтську частину на Angular та серверну частину на Node.js з Express, використовуючи MySQL як СКБД;
– протестувати розроблену систему та підготувати керівництво користувача.

Об’єкт дослідження: системи управління контентом.

Предмет дослідження: Angular, Node.js.

Методи дослідження: аналітичний, моделювання, проєктування, програмний.

У першому розділі описані функціональні й нефункціональні вимоги до системи, надано аналіз предметної області, проаналізовані існуючі аналоги та надано їхнє порівняння.

У другому розділі аргументовано вибір UML, спроєктовано архітектуру CMS, а також надані ключові діаграми та детально описані варіанти використання системи.

У третьому розділі наведено детальний опис інструментів розробки, процес встановлення та налаштування клієнтської та серверної частин платформи, описані основні API-endpoint, надані приклади unit-тестування та надано покрокове керівництво користувача.

1 ТЕХНІЧНЕ ЗАВДАННЯ ТА ФУНКЦІОНАЛЬНІ ВИМОГИ СИСТЕМИ

1.1 Перелік умовних позначень, скорочень, термінів

Angular – фреймворк для розробки SPA-застосунків із модульною архітектурою, реактивними формами та системою DI [1].

Angular Material – бібліотека UI-компонентів для Angular, що забезпечує сучасний та адаптивний інтерфейс [2].

CORS – механізм контролю доступу між доменами, який може блокувати запити під час локальної розробки.

Express.js – мінімалістичний серверний фреймворк для Node.js, призначений для маршрутизації та роботи з middleware [5].

JWT – токен авторизації у форматі JSON Web Token, що передається у заголовку Authorization [13].

Markdown-редактор – інструмент для створення та редагування постів у форматі Markdown із можливістю вставки медіа.

MySQL – реляційна система керування базами даних для зберігання користувачів, постів, категорій, медіа та ролей [10].

Node.js – серверна платформа для створення вебдодатків та REST API [11].

Proxy.conf.json – файл конфігурації Angular для проксування запитів і обходу обмежень CORS під час локальної розробки.

REST API – архітектурний стиль взаємодії в розподілених системах, що базується на HTTP-методах, URI та безстановій комунікації [14].

RxJS – бібліотека реактивного програмування, що працює на основі Observables та операторів [4, 15].

Slug – текстовий ідентифікатор поста, що використовується в URL та для пошуку.

SPA – односторінковий застосунок, у якому вміст оновлюється без повного перезавантаження сторінки.

Swagger UI – інструмент для документування та інтерактивного тестування REST API.

UML – уніфікована мова моделювання, яка використовується для опису структури та поведінки системи.

URI – уніфікований ідентифікатор ресурсу, що використовується у REST API [14].

ДВВ – діаграма варіантів використання.

ДД – діаграма діяльності.

ДП – діаграма послідовності.

ДР – діаграма розгортання.

Користувач – суб’єкт, що взаємодіє із системою.

1.2 Функціональні вимоги

1.2.1 Призначення і цілі створення системи

Функціональне призначення системи – надати авторам зручний інструмент для простого та швидкого створення публікацій блогу.

Експлуатаційне призначення – система експлуатується користувачами на різних рівнях: читачами для перегляду опублікованих матеріалів, авторами для створення та адміністрування власних публікацій, а також адміністраторами для керування користувачами, ролями та категоріями.

Мета створення системи – розробити систему управління контентом для ведення блогу.

1.2.2 Загальні функціональні можливості системи

Система має надавати авторам такі можливості:

– створювати/редагувати/видаляти/переглядати власні пости;

– змінювати статус/видимість поста;
– переглядати попередній вигляд поста.

Для адміністраторам система має надавати такі можливості:
– створення/перегляд/редагування/видалення користувачів;
– створення/перегляд/редагування/видалення постів;
– призначення/оновлення ролі користувача;
– створення/редагування/видалення категорій.

Система має надавати всім користувачам такі можливості:
– вхід/вихід до/з системи;
– реєстрація в системі;
– перегляд/оновлення профілю;
– пошук/фільтрація постів;
– перегляд публічних постів.

1.3 Нефункціональні вимоги

Інтерфейс користувача. Інтерфейс має бути інтуїтивно зрозумілим та адаптивним до різних типів пристроїв. Компоненти системи повинні бути побудовані за допомогою бібліотеки Angular Material.

Підтримка браузерів. Система повинна підтримувати та коректно працювати в актуальних версіях Chrome, Firefox, Safari.

Вимоги до продуктивності. Відкриття списку постів не повинно перевищувати 1 секунди за наявності 100+ записів. Створення/оновлення поста не довше за 1 секунду за умови стандартного навантаження.

Вимоги до безпеки. Паролі повинні зберігаютися у вигляді криптографічного хешу. Доступ до API повинен бути захищеним за допомогою JWT. Рольова модель повинна обмежувати несанкціонований доступ до серверних та клієнтських маршрутів. Система не повинна надавати доступ для незареєстрованих користувачів до приватних розділів.

Доступність та UX. Компоненти повинні відповідати базовим вимогам доступності, а також мати чітку валідацію форм та повідомлення про помилки.

1.4 Опис предметної області

Сфера ведення блогів вимагає швидкого та компактного циклу взаємодії з контентом, підтримки мультимедійного контенту та базового керування публічністю матеріалів. Традиційні CMS надають широкий спектр можливостей, але часто мають завищений поріг входу для авторів, платні функції, надмірні налаштування та складні робочі потоки.

Розроблювана система управління контентом для ведення блогу фокусується на мінімально необхідному функціоналі, а саме: створення постів із вбудованим Markdown-редактором і панелю інструментів для форматування, безпечне вбудовування медіа, категоризація та керування видимістю. Система зберігає користувачів, пости та категорії в базі даних, підтримує пошук/фільтрацію та забезпечує доступ лише відповідно до рольової моделі і статусу поста.

1.5 Опис системи

Система побудована за принципом архітектури «клієнт-сервер». Розглянемо складові частини.

Клієнтську частину реалізовано на Angular з компонентною бібліотекою Angular Material. Основними компонентами є форми входу/реєстрації, головна сторінка зі списком постів з фільтрацією/пошуком, сторінка перегляду поста, розділ «Мої пости» для автора, редактор постів із базовою панелю інструментів для форматування, інтеграція зображень та відео, вибір категорій.

Серверну частину створено засобами Node.js з фреймворком Express. Відповідає за REST API, маршрути для автентифікації із захистом JWT,

маршрути для постів за рольовою моделлю, маршрути для категорій з доступом тільки адміністраторам, валідація вхідних даних. Підтримується завантаження медіа і віддача статичних файлів.

База даних реалізована на MySQL. Має таблиці користувачів, ролей, постів, категорій та проміжні таблиці для зв’язків «багато-до-багатьох» між постами та категоріями.

Процес створення поста наступний: автор заповнює заголовок, додає зміст у Markdown-редакторі, вставляє медіа, обирає категорії та статус, попередньо переглядає пост та зберігає його. Після публікації пост з’являється у публічному списку, а чернетки доступні лише самому автору та адміністратору.

1.6 Огляд та порівняння аналогів

На ринку платформ для ведення блогу представлено як важкі та великі системи керування вмістом, так і легкі сервіси публікації. Більшість із них підтримують вставку мультимедійного контенту і мають базові механізми для організації матеріалів. Розглянемо подібні платформи.

WordPress є найпоширенішою системою керування вмістом із розгалуженою екосистемою тем і плагінів. Має широкі можливості щодо масштабування, налаштування ролей користувачів, керування категоріями/тегами, гнучкий редактор блоків та велику кількість інтеграцій. Але потребує налаштування, адміністрування та періодичних оновлень, що ускладнює вхід, особливо для початківців [20].

Ghost є сучасною платформою, орієнтованою на блогінг і публікації. Підтримує Markdown-редактор, вбудовування медіа, мінімалістичний інтерфейс і швидкодію. На відміну від WordPress, має менше вбудованих можливостей, але натомість має нижчий поріг входу для авторів [7].

Medium представляє собою хмарний сервіс публікацій із простим редактором і фокусом на читабельності. Надає мінімальні інструменти

форматування та вбудовування мультимедіа, але обмежує кастомізацію й контроль доступності на рівні окремих ролей/груп [9].

Substack є сервісом для розсилок і блогів, що направлений на спрощення публікацій, монетизації та підписки. Надає базовий набір для редагування і мультимедіа, але із недоліків можна виділити вузьку спеціалізацію та менше налаштувань структури контенту [16].

У таблиці 1.1 наведено порівняння описаних вище систем із системою, що розробляється.

Таблиця 1.1 – Порівняння аналогів

Платформа	**Вбудовування медіа**	**Категорії/теги**	**Керування доступністю публікацій**	**Простота створення та наповнення**	**Особливості**
WordPress	Так	Так	Гнучке (ролі/видимість, плагіни)	Середня (багато налаштувань)	Найбільша екосистема, плагіни, SEO
Ghost	Так	Так	Базове (публічно/приватно, членство)	Просте	Швидкий, Markdown-фокус, мінімалізм
Medium	Так	Обмежено	Переважно публічно	Дуже просте	Єдина стилістика, менше контролю
Substack	Так	Обмежено	Публічно/для підписників	Дуже просте	Підписки та розсилки за замовчуванням
Наша система	Так	Так	Публічно/чернетка + ролі	Дуже просте	Швидкий старт, простота використання

Аналіз показує, що наявні рішення або надто перевантажен функціоналом і вимагають додаткового адміністрування, або жорстко обмежують кастомізацію і мають платні плагіни. Запропонована система вирізняється простотою використання, сучасним інтерфейсом на Angular Material, достатньою функціональністю для блогу та можливістю мастабування.

2 ПРОЄКТУВАННЯ

2.1 Використання UML під час розробки системи

Використання UML під час розробки програмних систем є ключовим компонентом процесу моделювання, що дозволяє представити структуру, поведінку та взаємодію компонентів у зрозумілому візуальному форматі [18].

UML – це універсальна мова, яку однаково можуть інтерпретувати аналітики, архітектори, розробники та тестувальники. Такий підхід покращує комунікацію в команді, оптимізує обговорення вимог і рішень та сприяє спільному розумінню функціональності системи протягом усього процесу її розробки. Використання моделей UML полегшує виявлення логічних недоліків, розбіжностей і ризиків на ранній стадії, тим самим зменшуючи ймовірність виникнення критичних проблем під час впровадження [6, 18].

Основна перевага UML полягає в його здатності організовувати інформацію, пов’язану з системою, за допомогою різних діаграм, кожна з яких виконує окрему функцію. Структурні діаграми, включаючи діаграми класів і компонентів, полегшують опис статичної архітектури системи, охоплюючи її елементи, атрибути, методи та взаємозв’язки між цими елементами. Ця функція є особливо корисною на етапі проєктування, оскільки дозволяє розробникам зрозуміти логіку бізнес-об’єктів, залежностей та інтерфейсів [6, 18].

UML базується на системному підході до розробки, що охоплює всі етапи життєвого циклу: від аналізу та проєктування до впровадження, тестування та подальшої підтримки. На початкових етапах розробки UML допомагає чітко сформулювати вимоги, сприяючи досягненню взаєрозуміння між командою та клієнтом щодо функціональних можливостей. На етапі проєктування UML сприяє створенню логічної та фізичної архітектури системи, яка згодом служить основою для коду. На етапі впровадження моделі надають розробникам структуровану інформацію, допомагаючи уникнути зайвих залежностей та помилок у проєктуванні. На етапі тестування діаграми UML можуть слугувати

цінним джерелом для побудови тестових випадків, особливо в контексті сценаріїв використання або послідовностей взаємодії [6, 8 ,18].

Таким чином, використання UML під час розробки системи є гнучким інструментом, який забезпечує візуалізацію, структуризацію та чіткість технічних рішень. UML сприяє глибшому розумінню вимог, підвищенню якості архітектури та ефективній взаємодії між учасниками проєкту.

2.2 Діаграма варіантів використання

Діаграма варіантів використання є однією із базових діаграм UML, яка використовується для опису функціональності системи з точки зору користувачів та зовнішніх систем. Її призначенням є показ, що повинна робити система, а не як вона це робить технічно. На такій діаграмі зображаються актори та прецеденти, що є окремими функціями, які система надає цим акторам. Завдяки цьому діаграма варіантів використання є зручною для спілкування із замовниками та стейкхолдерами, які не обов’язково розуміють технічні деталі, але можуть оцінити, чи всі потрібні функції враховано [6, 8, 19].

Основними елементами діаграми варіантів використання є актори та прецеденти [6, 8, 19].

Актор – це зовнішній по відношенню до системи користувач або інша система, що взаємодіє з нею. Це може бути людина, наприклад «Клієнт», «Адміністратор», або зовнішній сервіс типу «Платіжна система».

Прецедент – це окрема функція або сценарій взаємодії, який приносить бізнес-користь актору, наприклад «Зареєструватися», «Увійти в систему», «Оформити замовлення», «Оплатити рахунок». Між акторами і прецедентами проводяться зв’язки, що показують, хто саме використовує яку функцію.

У діаграмі варіантів використання часто використовують додаткові типи зв’язків: «include» та «extend». Зв’язок «include» означає, що один варіант використання завжди включає виконання іншого. Наприклад, прецедент «Створення поста» може включати прецедент «Змінити статус». Це дозволяє не

дублювати однакові частини сценарію в кількох місцях. Зв’язок «extend» позначає розширення поведінки, тобто додатковий варіант використання, який виконується за певних умов. Наприклад, «Створення поста» може бути розширений варіантом «Попередній перегляд», який виконується не завжди, а лише за умови, що клієнт має відповідний статус. Таке структурування робить модель більш гнучкою й читабельною.

На рисунку 2.1 зображена ДВВ системи управління кнотентом для ведення блогу.

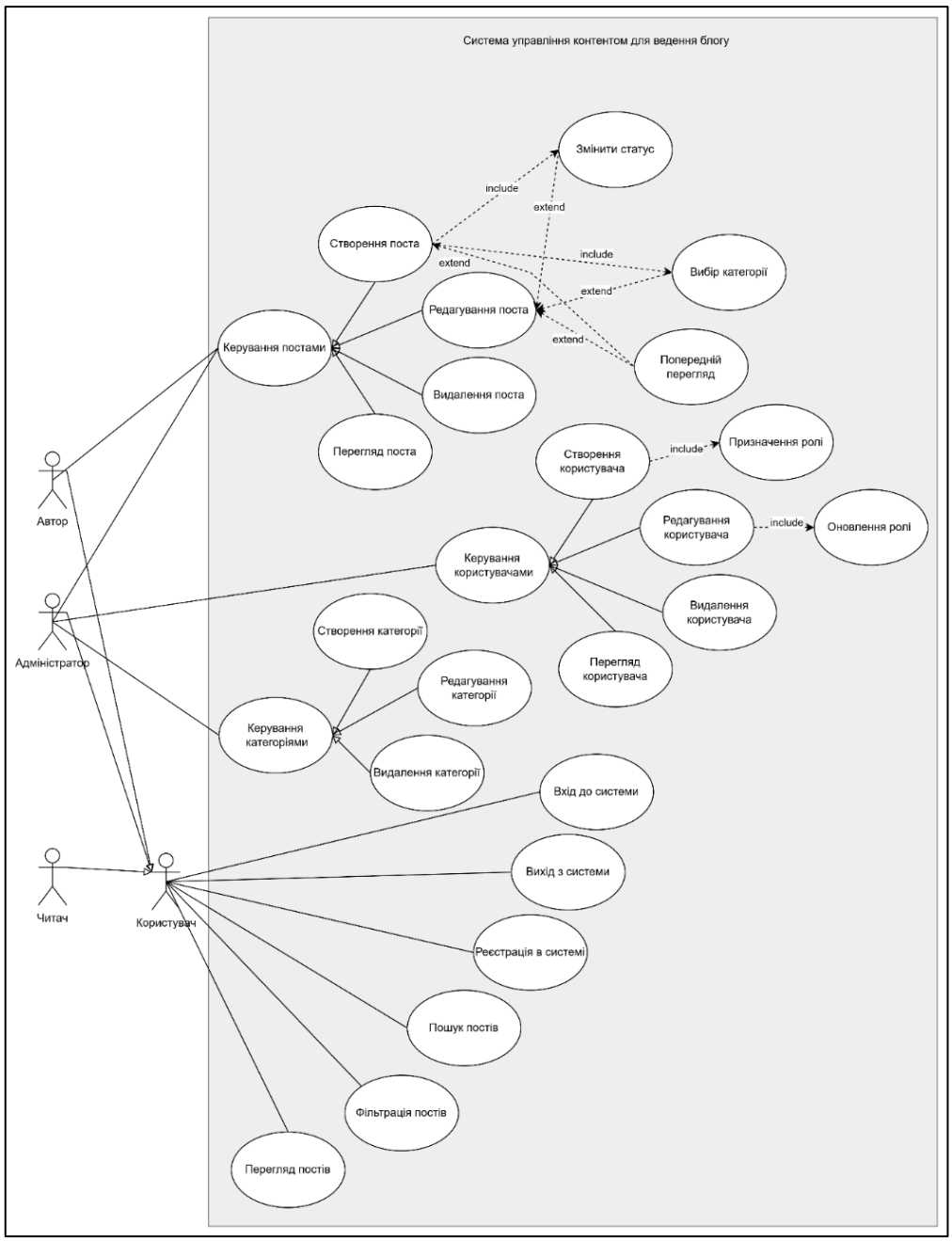


Рисунок 2.1 – Діаграма варіантів використання

Користувачі системи представлені наступними акторами: «Автор» «Читач» та «Адміністратор». Кожен з акторів взаємодіє з системою на різних рівнях відповідно до рольової моделі. Основним актором, який взаємодіє з постами є «Автор» (рис. 2.2).

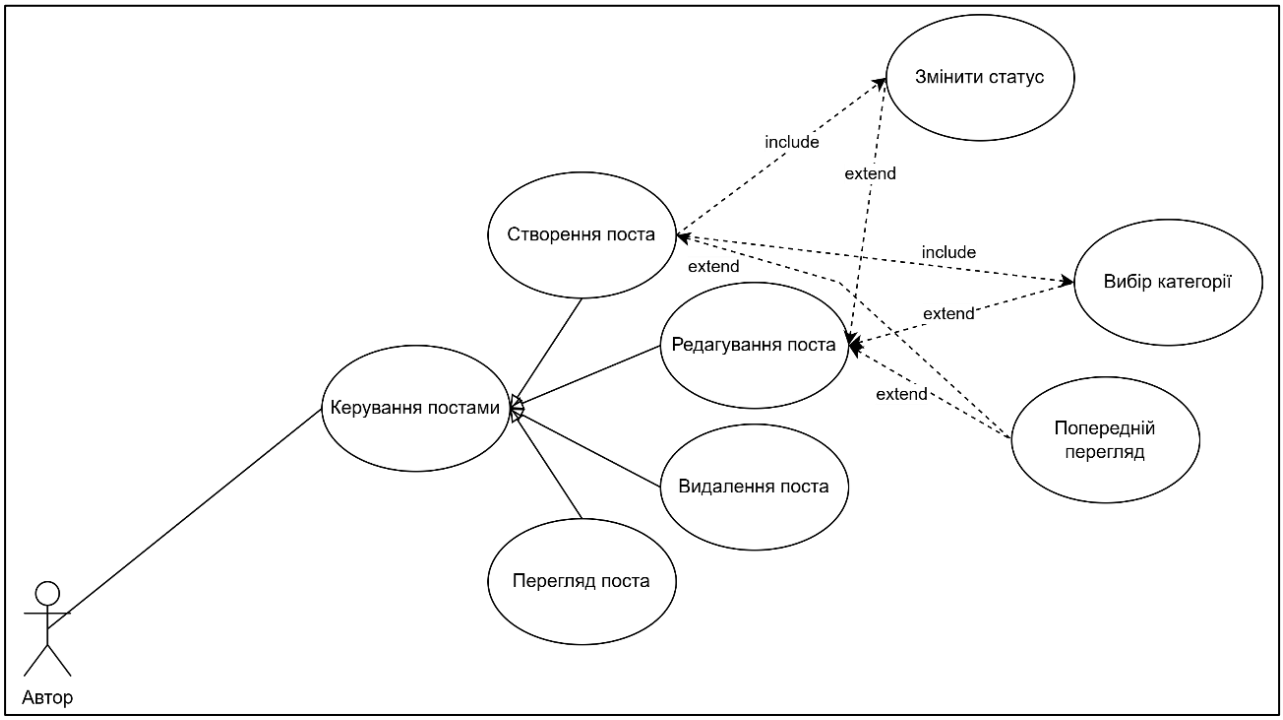


Рисунок 2.2 – ДВВ актора «Автор»

Адміністратор має права на керування постами, користувачами та категоріями.

Читач є базовим користувачем системи з правами на перегляд опублікованих постів.Актор «Користувач» об’єднує функціональні можливості всіх інших акторів (рис. 2.3).

Виділимо 1 з основних варіантів використання – «Створити пост». Після того, як користувач з правами «Автор» або «Адміністратор» натискає на кнопку «Новий пост», система відображає форму створення. Користувач заповнює відповідні поля форми та натискає на кнопку «Зберегти». Система створює новий запис та відображає сторінку з інформацією про пост. Також система валідує дані на всіх етапах, починаючи від перевірки полів форми на стороні клієнта і закінчуючи серверною перевіркою валідності та ролі.

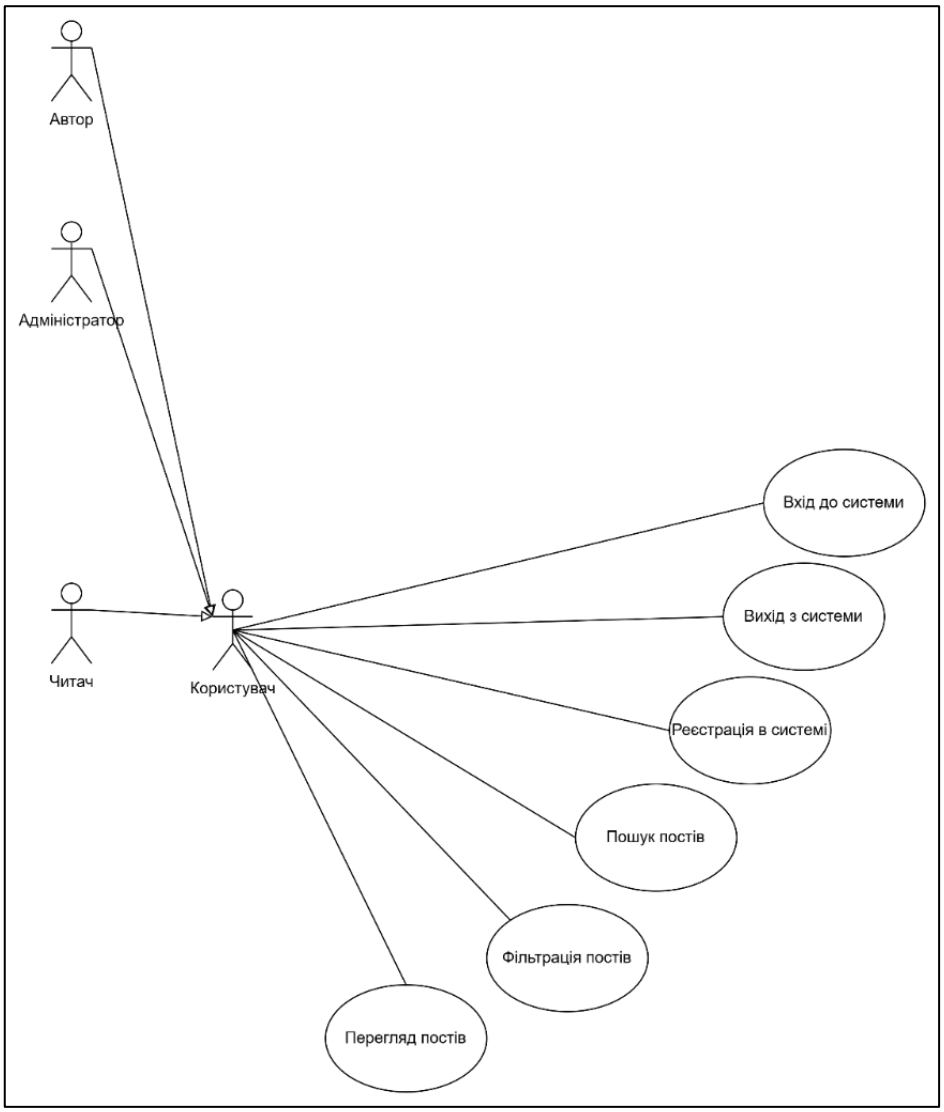


Рисунок 2.3 – ДВВ актора «Користувач»

2.2.1 Опис варіантів використання

Прецедент «Вхід до системи».

Призначення: даний варіант використання надає можливість користувачу автентифікуватися для доступу до особистого кабінету.

Передумови: користувач зареєстрований.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму входу, вводить email і пароль та натискає кнопку «Вхід». Система перевіряє облікові дані та надає доступ до розділів відповідно до ролі.

Виняткова ситуація 1: неправильні облікові дані – система виводить відповідне повідомлення про помилку авторизації.

Виняткова ситуація 2: сервер недоступний – система виводить відповідне повідомлення про технічну помилку та радить спробувати знов.

Прецедент «Вихід з системи».

Призначення: даний варіант використання надає можливість користувачу завершити сесію та вийти з системи.

Передумови: користувач увійшов в систему.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на кнопку «Вихід» в правому верхньому кутку. Система видаляє токен авторизації, очищує локальні дані сесії та перенаправляє на головну сторінку.

Виняткова ситуація 1: токен невалідний або протермінований – система очищує локальні дані та перенаправляє на головну сторінку.

Виняткова ситуація 2: сервер недоступний – система виконує локальний вихід і виводить відповідне повідомлення про помилку.

Прецедент «Реєстрація в системі».

Призначення: даний варіант використання надає можливість користувачу створити новий обліковий запис.

Передумови: користувач не зареєстрований у системі з вказаними обліковими даними.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму реєстрації, вводить email, пароль, роль і натискає кнопку «Зареєструватися». Система створює обліковий запис з відповідною роллю і входить до особистого кабінета.

Виняткова ситуація 1: email уже використовується – система виводить попередження та пропонує відновити доступ або змінити email.

Виняткова ситуація 2: невідповідність вимогам до пароля/валидації – система підсвічує поля та показує інструкції щодо виправлення.

Прецедент «Перегляд постів».

Призначення: даний варіант використання надає можливість користувачу переглянути список опублікованих постів.

Передумови: відсутні.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває головну сторінку або сторінку списку постів. Система завантажує перелік опублікованих постів з пагінацією та відображає їх.

Виняткова ситуація 1: сервер недоступний – система показує відповідне повідомлення про технічну помилку.

Виняткова ситуація 2: відсутні публікації – система показує порожній стан із підказкою.

Прецедент «Перегляд поста».

Призначення: даний варіант використання надає користувачу можливість переглянути повний вміст конкретного поста.

Передумови: пост опублікований або користувач має відповідні права доступу.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає пост зі списку. Система завантажує вміст поста і відображає детальну інформацію.

Виняткова ситуація 1: пост не знайдено або видалено – система показує повідомлення «Матеріал не знайдено» та повертає користувача до загального списку.

Виняткова ситуація 2: сервер недоступний – система показує відповідне повідомлення про технічну помилку та елементи повторного завантаження.

Прецедент «Пошук постів».

Призначення: даний варіант використання надає користувачу можливість пошуку постів за ключовими словами.

Передумови: користувач перебуває на сторінці списку постів.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить запит у поле пошуку та натискає на піктограму «Пошук» або натискає клавішу «Enter» на клавіатурі. Система застосовує пошуковий фільтр і відображає результати пошуку.

Виняткова ситуація 1: збій під час пошуку – система зберігає запит і пропонує повторити.

Виняткова ситуація 2: за запитом даних не знайдено – система відображає порожній результат із рекомендаціями змінити критерії пошуку.

Прецедент «Фільтрація постів».

Призначення: даний варіант використання надає користувачу можливість обмежити список постів за заданими критеріями.

Передумови: користувач перебуває на сторінці списку постів.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає критерії фільтрації. Система завантажує відфільтрований список і відображає результати фільтрації.

Виняткова ситуація 1: недійсні параметри фільтра – система ігнорує їх і повертає базовий список.

Виняткова ситуація 2: сервер недоступний – система показує повідомлення про помилку та пропонує скинути фільтри.

Виняткова ситуація 3: за фільтром даних не знайдено – система відображає порожній результат із рекомендаціями змінити критерії фільтрації.

Прецедент «Створення поста».

Призначення: даний варіант використання надає можливість автору створити новий пост.

Передумови: користувач увійшов у систему та має роль «Автор» або «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму створення, вводить заголовок, вміст, обирає категорії, задає статус та натискає «Зберегти». Система зберігає пост та відображає сторінку перегляду поста.

Виняткова ситуація 1: невалідні дані – система підсвічує помилки та блокує кнопку збереження.

Виняткова ситуація 2: помилка збереження на сервері – система показує відповідне повідомлення та пропонує повторити дію.

Прецедент «Редагування поста».

Призначення: даний варіант використання надає користувачу можливість змінити дані існуючого поста.

Передумови: користувач увійшов у систему, має роль «Автор» або «Адміністратор», пост належить автору або користувач має відповідні права.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває пост у режимі редагування, змінює дані і натискає кнопку «Зберегти». Система оновлює пост та відображає сторінку перегляду поста.

Виняткова ситуація 1: відсутні права редагування – система показує повідомлення про заборону операції.

Виняткова ситуація 2: конфлікт версій або помилка збереження – система повідомляє про помилку та пропонує оновити сторінку.

Прецедент «Видалення поста».

Призначення: даний варіант використання надає користувачу можливість видалити існуючий пост.

Передумови: користувач увійшов у систему, має роль «Автор» або «Адміністратор», пост належить автору або користувач має відповідні права.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на кнопку «Видалити» у меню поста та підтверджує дію. Система видаляє пост і оновлює список.

Виняткова ситуація 1: відсутні права – система показує повідомлення про заборону.

Виняткова ситуація 2: помилка на сервері – система показує відповідне повідомлення та не видаляє пост.

Прецедент «Змінити статус».

Призначення: даний варіант використання надає користувачу можливість змінити статус поста.

Передумови: користувач увійшов у систему, має роль «Автор» або «Адміністратор», має права на відповідний пост.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач у формі редагування поста обирає новий статус і підтверджує зміни. Система зберігає оновлений статус і відображає поточний стан поста.

Виняткова ситуація 1: невалідний перехід стану – система блокує зміну та показує пояснення.

Виняткова ситуація 2: помилка збереження – система показує відповідне повідомлення та залишає попередній статус.

Прецедент «Вибір категорії».

Призначення: даний варіант використання надає користувачу можливість призначити посту одну чи кілька категорій.

Передумови: користувач перебуває у формі створення або редагування поста.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває список категорій, обирає потрібні та підтверджує зміни. Система зберігає прив’язку поста до вибраних категорій.

Виняткова ситуація 1: категорії недоступні або не знайдені – система показує відповідне повідомлення та пропонує оновити список.

Виняткова ситуація 2: помилка збереження зв’язків – система показує відповідне повідомлення та не застосовує зміни.

Прецедент «Попередній перегляд».

Призначення: даний варіант використання надає користувачу можливість переглянути пост у режимі передпублікаційного відображення.

Передумови: користувач перебуває у формі створення або редагування поста.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на опцію «Попередній перегляд». Система рендерить вміст та відображає пост у вигляді, наближеному до опублікованого.

Виняткова ситуація 1: помилка рендерингу – система показує відповідне повідомлення та пропонує повернутися до редагування.

Виняткова ситуація 2: сервер недоступний – система показує відповідне повідомлення та пропонує повторити.

Прецедент «Створення користувача».

Призначення: даний варіант використання надає можливість адміністратору створити нового користувача.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор відкриває форму створення, вводить дані користувача та натискає «Зберегти». Система створює новий обліковий запис і відображає його у списку користувачів.

Виняткова ситуація 1: користувач з вказаним email уже існує – система відображає попередження і не створює обліковий запис.

Виняткова ситуація 2: помилка збереження – система показує відповідне повідомлення і пропонує повторити дію.

Прецедент «Перегляд користувача».

Призначення: даний варіант використання надає адміністратору можливість переглянути дані конкретного користувача.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор обирає користувача зі списку. Система завантажує та відображає дані його облікового запису.

Виняткова ситуація 1: користувача не знайдено – система відображає відповідне повідомлення та перенаправляє на сторінку зі списком користувачів.

Виняткова ситуація 2: сервер недоступний – система показує відповідне повідомлення та пропонує повторити.

Прецедент «Редагування користувача».

Призначення: даний варіант використання надає адміністратору можливість змінити облікові дані існуючого користувача.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор відкриває форму редагування, змінює дані й натискає «Зберегти». Система оновлює дані та відображає профіль користувача.

Виняткова ситуація 1: невалідні дані – система підсвічує помилки і не зберігає зміни.

Виняткова ситуація 2: конфлікт або помилка збереження – система показує відповідне повідомлення та пропонує оновити сторінку.

Прецедент «Видалення користувача».

Призначення: даний варіант використання надає адміністратору можливість видалити обліковий запис користувача.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор натискає «Видалити» для вибраного користувача та підтверджує дію. Система видаляє обліковий запис і оновлює список.

Виняткова ситуація 1: залежні дані/обмеження – система блокує видалення та відображає повідомлення з поясненнями.

Виняткова ситуація 2: помилка на сервері – система показує відповідне повідомлення та не видаляє обліковий запис.

Прецедент «Призначення ролі».

Призначення: даний варіант використання надає адміністратору можливість встановити роль користувача.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор у формі користувача обирає роль і натискає «Зберегти». Система застосовує нову роль і відображає профіль користувача.

Виняткова ситуація 1: спроба призначення невалідної ролі – система блокує дію та відображає повідомлення з поясненнями.

Виняткова ситуація 2: помилка оновлення – система показує відповідне повідомлення та не змінює роль.

Прецедент «Оновлення ролі».

Призначення: даний варіант використання надає адміністратору можливість змінити поточну роль користувача.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор відкриває профіль користувача, обирає нову роль і підтверджує. Система оновлює роль і відображає профіль користувача.

Виняткова ситуація 1: відсутні права у поточного адміністратора – система блокує оновлення та відображає повідомлення з поясненнями.

Виняткова ситуація 2: помилка на сервері – система показує відповідне повідомлення та залишає попередню роль.

Прецедент «Створення категорії».

Призначення: даний варіант використання надає можливість адміністратору створити нову категорію для постів.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор відкриває форму створення категорії, вводить назву і натискає на кнопку «Зберегти». Система створює категорію та відображає список категорій.

Виняткова ситуація 1: невалідні дані – система підсвічує помилки та блокує кнопку створення категорії.

Виняткова ситуація 2: вказана назва вже існує – система показує відповідне повідомлення та не створює дубль.

Виняткова ситуація 3: помилка збереження – система показує відповідне повідомлення та пропонує повторити.

Прецедент «Редагування категорії».

Призначення: даний варіант використання надає можливість адміністратору змінити дані існуючої категорії.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор відкриває форму редагування категорії, змінює дані та натискає на кпопку «Зберегти». Система оновлює категорію та відображає список категорій.

Виняткова ситуація 1: невалідні дані – система підсвічує помилки та блокує кнопку оновлення категорії.

Виняткова ситуація 2: помилка збереження – система показує відповідне повідомлення та пропонує повторити.

Прецедент «Видалення категорії».

Призначення: даний варіант використання надає адміністратору можливість видалити категорію.

Передумови: користувач увійшов у систему та має роль «Адміністратор».

Основний потік подій: даний варіант використання починає виконуватися, коли адміністратор натискає на кнопку «Видалити» для вибраної категорії та підтверджує дію. Система видаляє категорію та відображає список категорій.

Виняткова ситуація 1: категорія використовується в постах – система блокує видалення та пропонує спочатку змінити/видалити відповідні пости.

Виняткова ситуація 2: помилка на сервері – система показує відповідне повідомлення та не видаляє категорію.

2.3 Діаграма діяльності

Діаграма діяльності – тип діаграми UML, яка використовується для моделювання динамічних аспектів системи. Вона відображає послідовність дій, логіку процесу, переходи між станами, а також точки розгалуження та синхронізації. Діаграми активності є корисним інструментом для візуалізації бізнес-процесів, алгоритмів, сценаріїв користувачів та внутрішньої поведінки системи. Ці інструменти полегшують розуміння даних або потоків управління в рамках конкретного процесу [6, 8].

Основні компоненти діаграми активності включають дії, переходи, початкові та кінцеві стани, розгалуження, паралельні гілки та об’єкти, залучені до процесу. Використання діаграм активності підвищує прозорість процесів, полегшує виявлення потенційних проблем або можливостей для оптимізації та спрощує комунікацію між аналітиками, розробниками та зацікавленими сторонами [6, 8].

На рисунках 2.4 – 2.5 надано ДД прецеденту «Створити пост».

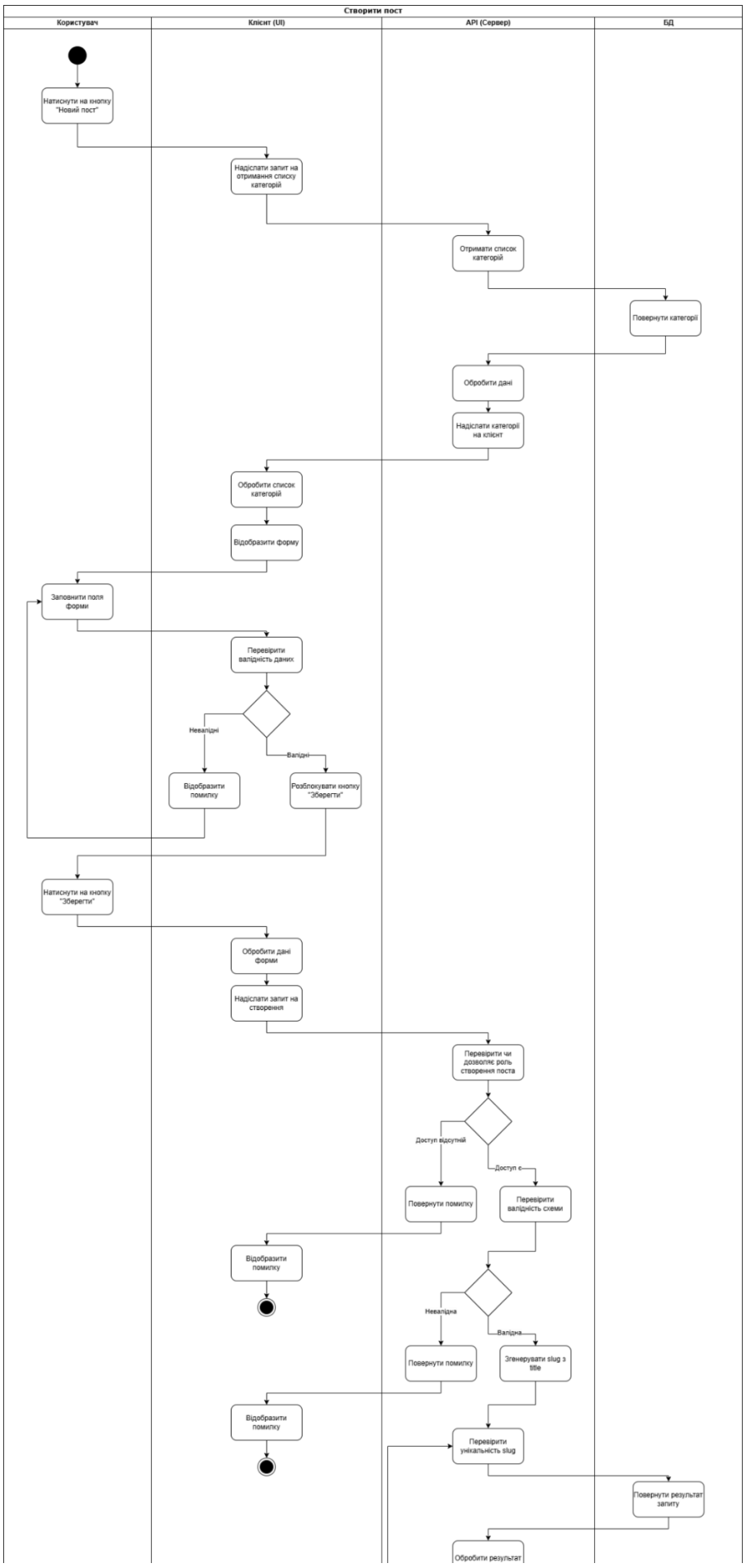


Рисунок 2.4 – ДД основного бізнес-процесу

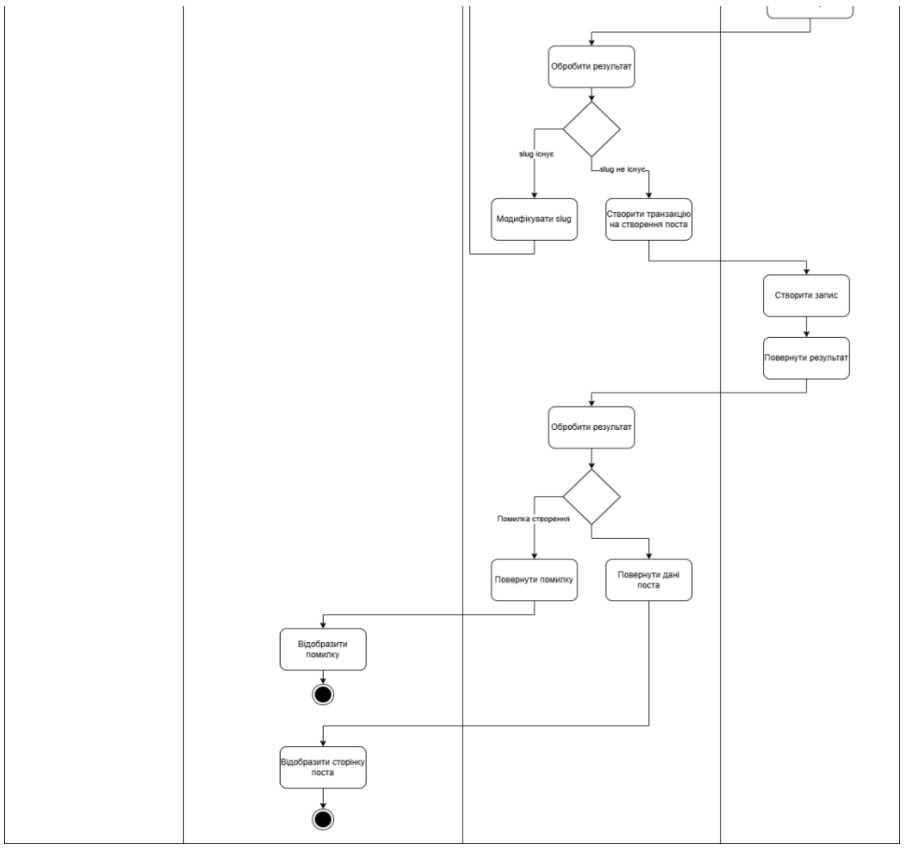


Рисунок 2.5 – ДД основного бізнес-процесу

ДД даного варіанту використання візуалізує однин із фундаментальних бізнес-процесів системи. Вона забезпечує розуміння послідовності та взаємозалежності операцій, а також демонструє точки прийняття рішень та можливі сценарії винятків [6 ,8].

2.4 Діаграма послідовності

Діаграма послідовності – тип діаграми, що використовується для моделювання взаємодій між об’єктами або компонентами системи в часі. Вона демонструє учасників комунікації, зміст обмінюваних повідомлень та послідовність їх передачі. Основний акцент робиться на хронологічній послідовності викликів методів, запитів, відповідей та сигналів, що полегшує розпізнавання робочого процесу системи від початку до завершення [6, 8].

На рисунку 2.6 наведено ДП для прецеденту «Створити пост».

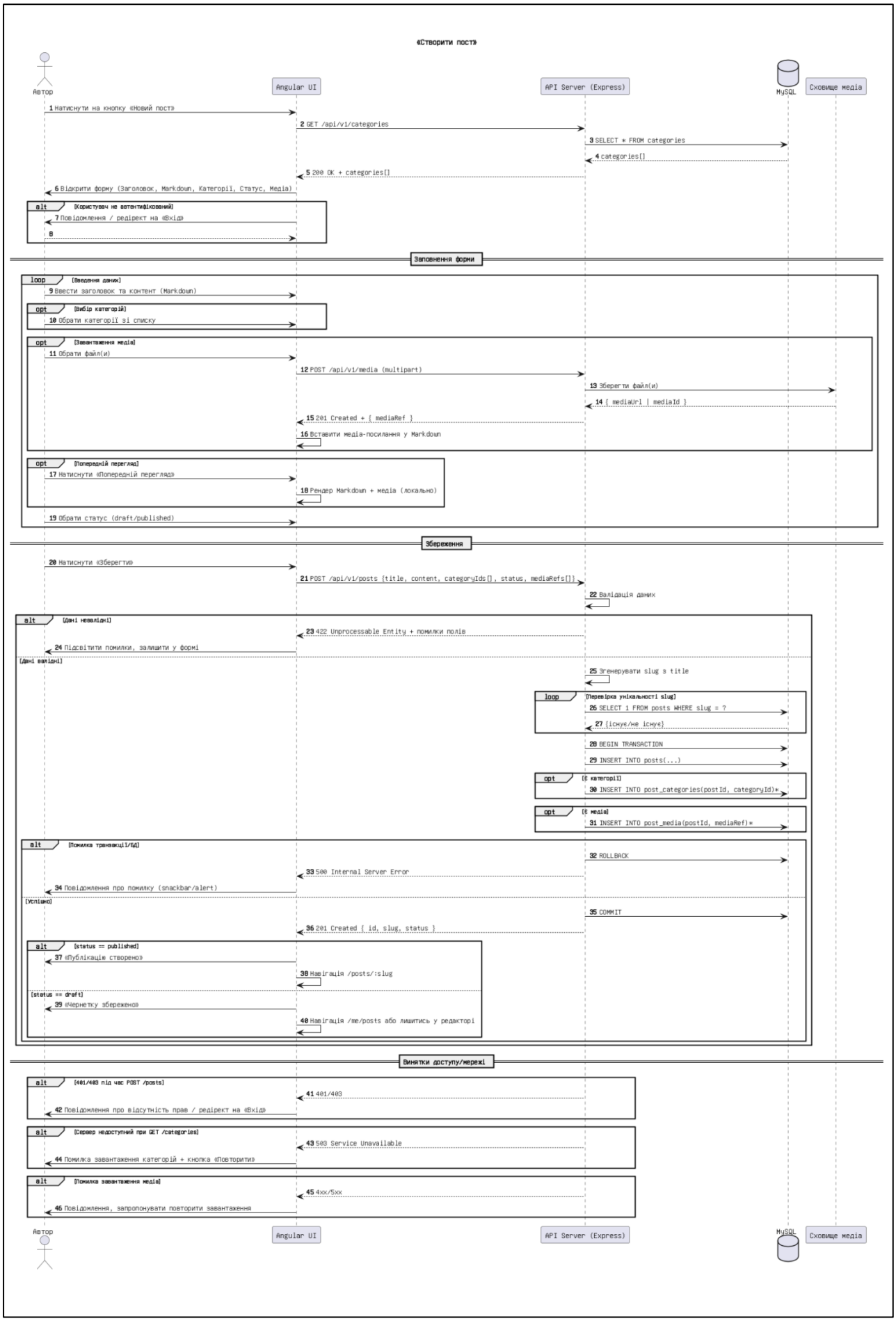


Рисунок 2.6 – ДП основного бізнес-процесу

Ця діаграма візуалізує логіку взаємодії з курсом як лінійний процес. Її ключова перевага полягає в тому, що вона чітко показує послідовність викликаних дій, а також відповідальних за їх виконання [6, 8].

Діаграма послідовності ілюструє учасників взаємодії у верхній частині, до яких можуть належати користувачі, підсистеми, служби або окремі об’єкти. Кожен учасник має вертикальну лінію життя, яка позначає його присутність у сценарії. Повідомлення розташовані у певному порядку, окресленому горизонтальними стрілками, які слугують для позначення викликів операцій, результатів повернення або обміну даними між учасниками [6, 8].

Крім того, діаграма може містити умовні блоки, цикли, опціональні фрагменти та інші комбіновані фрагменти, що дозволяють моделювати логіку розгалуження, повторювані дії та альтернативні сценарії поведінки системи в рамках одного діаграмного представлення [6, 8].

Діаграми послідовностей широко використовуються в сферах аналізу вимог, проєктування архітектури та документації випадків використання. Крім того, вони виявляють потенційні недоліки в логіці взаємодії компонентів до етапу впровадження [6, 8].

2.5 Діаграма розгортання

Діаграма розгортання – діаграма UML, яка показує фізичне розміщення програмних компонентів на апаратних вузлах системи. Вона відображає сервери, клієнтські пристрої, мережеві вузли, їхні з’єднання, а також програмні артефакти, які розгорнуті на цих вузлах. Така діаграма допомагає зрозуміти, як система працює в реальному середовищі [6, 8]. На рисунку 2.7 показано розгортання нашої системи.

Розділ обґрунтовує застосування UML для проєктування системи управління контентом блогу та представляє набір інтегрованих діаграм. Надані діаграми відображають повний цикл проєктування: від акторів/функціонала до

логіки взаємодії, часової послідовності та фізичного розгортання. Результатом є узгоджена основа для архітектури та реалізації, яка стане фундаментом для подальшої імплементації.

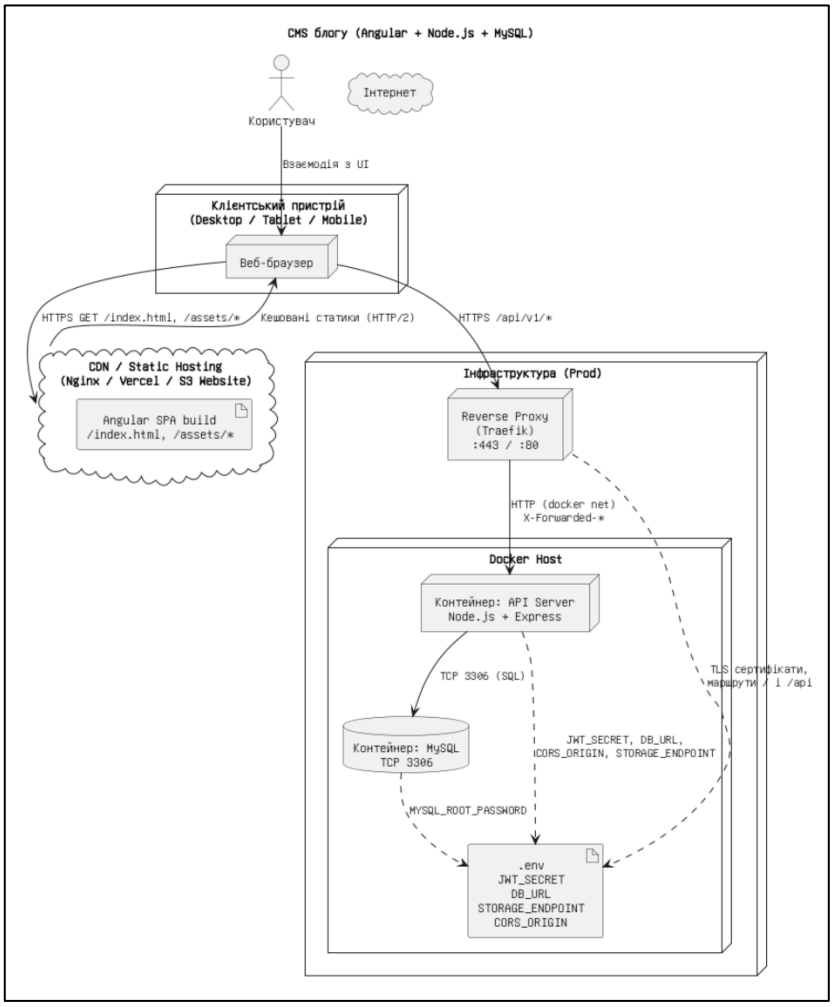


Рисунок 2.7 – Діаграма розгортання

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Опис інструментів розробки

Система реалізована за допомогою сучасного стеку технологій для створення вебплатформ публікації контенту урахуванням вимог та найкращих практик щодо масштабованості, гнучкості та продуктивності.

Angular – фреймворк для SPA з модульною архітектурою, реактивними формами та DI [1].

RxJS – бібліотека реактивного програмування на основі Observable та операторів [4, 15].

Node.js – серверна платформа для створення застосунків і REST API [11].

Express.js – мінімалістичний фреймворк маршрутизації та middleware [5].

MySQL – реляційна СКБД для зберігання користувачів, постів, категорій, медіа та ролей [10].

Angular Material – компонентна бібліотека [2].

3.2 Основні класи системи

Оскільки ключовим бізнес-процесом є створення та публікація постів, основні сутності будуть ті, що взаємодіють з постами, категоріями, користувачами та медіа.

Зі клієнтської сторони можна виділити наступні сервіси та компоненти.

Компонент PostEditorComponent та MarkdownEditorComponent відповідають за відображення форми створення/редагування поста з Markdown-редактором та вставкою медіа.

Компоненти PostListComponent та PostDetailsComponent відповідають за відображення відповідно списоку і карток постів з пошуком, фільтрами та пагінацією.

Компонент MyPostsComponent відповідає за особистий кабінет автора для керування власними постами.

Для обміну даними з REST API використовуються сервіси AuthService для авторизації, PostService для керування постами, CategoryService для взаємодії з категоріями і MediaService для завантаження та отримання даних.

За захист шляхів відповідають AuthGuard та RoleGuard.

Розглянемо ключові елементи серверу.

Каталог models/ містить класи, що є сутностями відповідних таблиць в базі даних, а також надають можливість серверної валідації схеми даних. Основним є клас Post, який відповідає за взаємодію з даними постів.

Каталог controllers/ містить класи, що оброблюють запити з клієнта та використовують відповідний обробник. Містять логіку щодо перевірки даних запиту, взаємодії з моделлю та відправки відповіді клієнту. За аналогією з класами моделей, основним можна назвати PostController, який відповідає за запити пов’язані з основною бізнес-логікою.

Нижче наведено приклад функції створення поста (рис. 3.1).

```
async function create(req, res) :Promise<any> {  Show usages
    const { title, body, status :string = 'DRAFT', categories :any[] = [] } = req.body;
    const slug :any = makeSlug(title);

    const exists :Post = await Post.findOne( options: { where: { slug } });
    if (exists) return res.status(409).json({ message: 'Post with same slug exists' });

    const payload :{…} = {
        title,
        slug,
        body,
        status,
        author_id: req.user.id,
        published_at: status === 'PUBLISHED' ? new Date() : null
    };

    const post :Post = await Post.create(payload);


    if (Array.isArray(categories) && categories.length) {
        const cats :Category[] = await Category.findAll( options: { where: { slug: categories } });
        await post.setCategories(cats);
    }

    return res.status(201).json(post);
}
```

Рисунок 3.1 – Функція створення поста

Ознайомитися з компонентом створення поста можна в додатку А, з сервісом у додатоку Б, а з контролером і моделлю в додатках В і Г.

3.3 Встановлення та налаштування оточення

Клієнтська та серверна частини працюють на мові програмування JavaScript, тому для початку необхідно встановити актуальну версію node і npm. Операційною системою для розробки нашого застос, унку є Linux Ubuntu, тому рекомендованим варіантом є використання команди nvm install –lts. Опціонально можна встановити пакети напряму, через команди sudo apt install nodejs та sudo apt install npm [11, 12].

Далі розгортаємо клієнтську сторону. Так к використовується Angular, то спочатку необхідно встановити Angular CLI за допомогою команди npm install -g @angular/cli [1, 3]. Додаток будується за архітектурою «клієнт сервер», тому створюємо спільну теку, яка буде містити код клієнтської та серверної частин. Переходимо до загальної теки і створюємо клієнтську частину за допмогою команди ng new blog-client –standalone –routing=false –style=scss (рис. 3.2) [1].

```
svarog@svarog:~/projects/blog-cms$ ng new blog-client --standalone --routing=false --style=scss
✔ Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)? No
CREATE blog-client/README.md (1473 bytes)
CREATE blog-client/.editorconfig (314 bytes)
CREATE blog-client/.gitignore (587 bytes)
CREATE blog-client/angular.json (2784 bytes)
CREATE blog-client/package.json (1042 bytes)
CREATE blog-client/tsconfig.json (915 bytes)
CREATE blog-client/tsconfig.app.json (424 bytes)
CREATE blog-client/tsconfig.spec.json (434 bytes)
CREATE blog-client/.vscode/extensions.json (130 bytes)
```

Рисунок 3.2 – Створення клієнтської частини

У проєкті використовується компонентна бібліотека Angular Material, тому переходимо до теки клієнтської частини і встановлюємо його за допомогою команди ng add @angular/material (рис. 3.3) [2].

```
svarog@svarog:~/projects/blog-cms$ cd blog-client
svarog@svarog:~/projects/blog-cms/blog-client$ ng add @angular/material

Would you like to share pseudonymous usage data about this project with the Angular Team
at Google under Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.dev/cli/analytics.

   No
Global setting: enabled
Local setting: disabled
Effective status: disabled
```

Рисунок 3.3 – Встановлення Angular Material

Наступним кроком створюємо структуру клієнтської частини орієнтуючись на features-підхід та особливості standalone-компонентів (рис. 3.4) [1, 3].

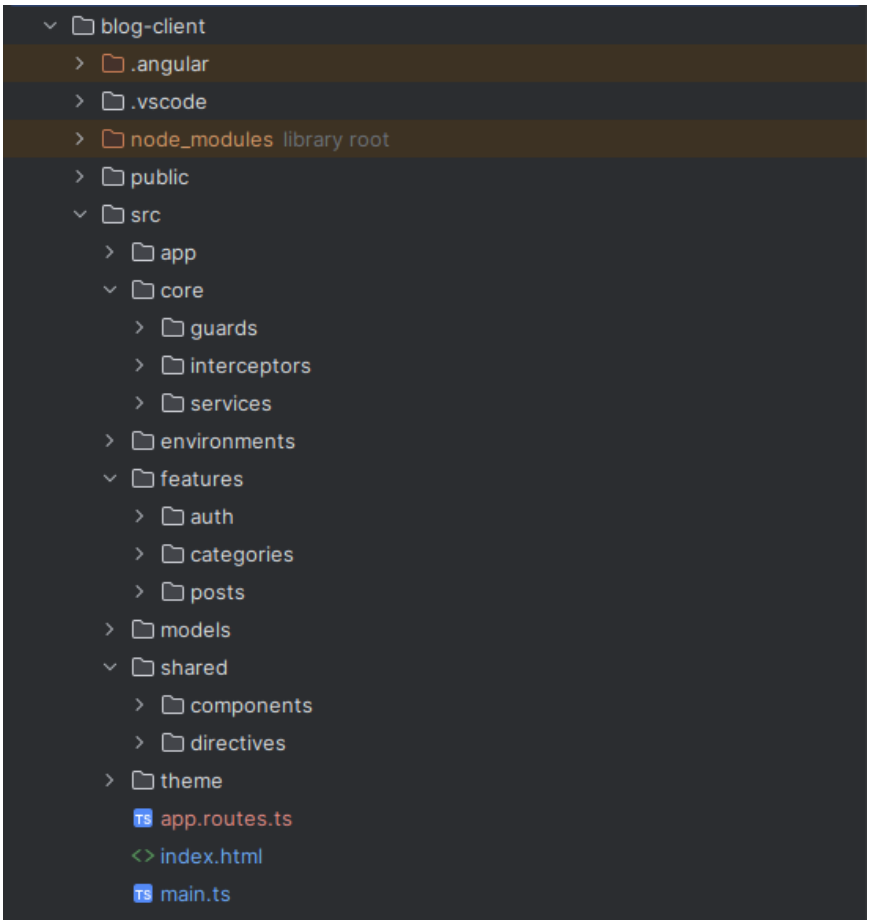


Рисунок 3.4 – Структура клієнтської частини

Далі створюємо у корені проєкту теку blog-backend, переходимо до неї та створюємо файл package.json за допомогою команди npm i -y (рис. 3.5) [12].

```
svarog@svarog:~$ npm init -y
```

Рисунок 3.5 – Створення файлу package.json

Для зручної та коректної роботи серверної чатини необхідно встановити залежності для роботи з CORS, медіа, використання самого фреймфорку express, підключення до mysql за допмогою sequelize, авторизацією черз jsonwebtoken та bycryptjs, валідацією з викопистанням express-validator та доступу до змінних оточення використовуючи dotenv. Залежності встановлюються за допомгою команди npm i package [12, 13]. Результат роботи команди можна переглянути у файлі package.json (рис. 3.6).



Рисунок 3.6 – Серверні залежності

Створюємо серверну структуру використовуючи адаптацію патерну MVC для express (рис. 3.7) [17].

Так як клієнтська частина буде надсилати як звичайні дані, так і медіа, то при локальні розробці часто виникає блокування зі сторони CORS, тому під час розробки створюємо proxy.conf.json, який буде слухати відповідні шляхи та перенаправляти на сервер (рис. 3.8) [1, 3].

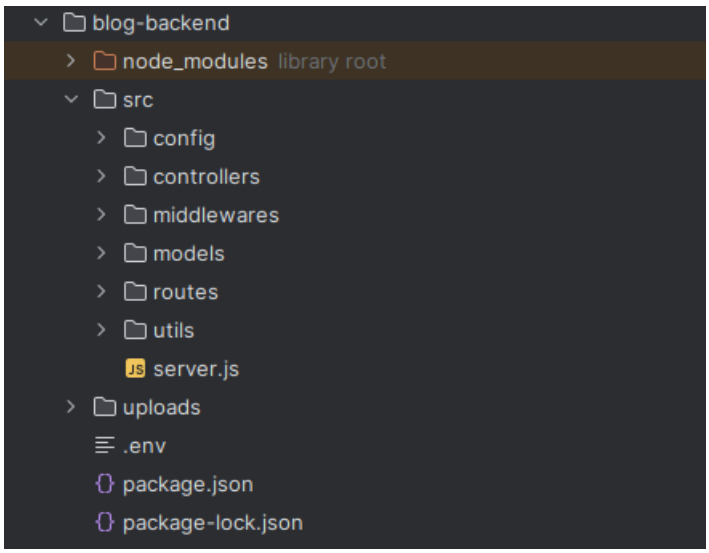


Рисунок 3.7 – Структура серверної частини

```
{
  "/api": { "target": "http://localhost:4000", "secure": false, "changeOrigin": true },
  "/uploads": { "target": "http://localhost:4000", "secure": false, "changeOrigin": true }
}
```

Рисунок 3.8 – Файл proxy.conf.json

Для серверної частини створюємо файл оточення .env у якому прописуємо константи для підключення до бази даних, створення адміністратора системи та взаємодії з JWT-авторизацією (рис. 3.9) [11].

```
PORT=4000
CORS_ORIGIN=http://localhost:4200


DB_HOST=localhost
DB_PORT=3306
DB_NAME=blogcms
DB_USER=root
DB_PASS=root


JWT_ACCESS_SECRET=c664db440ee73d55c42acba30c35df50
JWT_ACCESS_EXPIRES=15m
JWT_REFRESH_SECRET=05a730b507c6905a81729cc47c838aa8
JWT_REFRESH_EXPIRES=7d


ADMIN_EMAIL=admin@example.com
ADMIN_PASSWORD=Admin123!
```

Рисунок 3.9 – Файл .env

Для перевірки коректності роботи клієнтської частини, виконуємо команду ng serve –proxy-config proxy.conf.json з її корення (рис. 3.10) [1].



Рисунок 3.10 – Запуск клієнтської частини

Для перевірки коректності роботи серверної частини використовуємо мінімальний код, необхідний для запуску [11] і виконуємо команду npm run start (рис. 3.11) [11, 12].

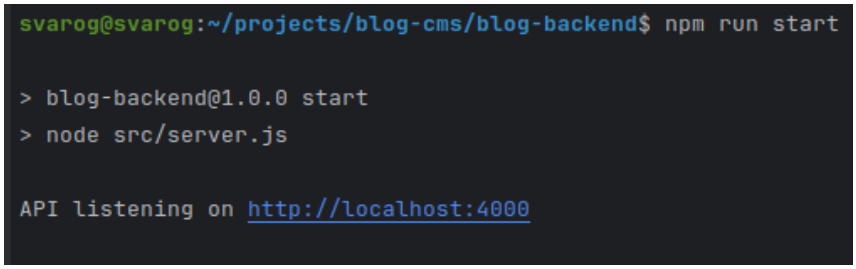


Рисунок 3.11 – Запуск серверної частини

3.4 Опис REST API

REST – це архітектурний стиль, що являє собою набір обмежень, які використовуються для розробки розподілених систем. Сервіс вважається «RESTful», якщо він дотримується таких ключових правил, як: клієнт-серверна модель, відсутність стану, можливість кешування, єдиний інтерфейс та шарова архітектура. У такій системі ресурси ідентифікуються за допомогою URI, а взаємодія здійснюється стандартними HTTP-методами, кожен з яких має чітко визначену семантику [14].

API системи побудований на принципах REST та використовує базовий шлях /api/v1 для всіх маршрутів. Обмін даними здійснюється виключно у форматі application/json. Для забезпечення безпеки використовується автентифікація Bearer JWT, де токен передається у заголовку Authorization [13]. Система підтримує рольову модель з трьома основними ролями: admin, author та reader, що визначають рівень доступу до функціонала. Для зручності роботи з великими обсягами даних реалізовано пагінацію та фільтри пошуку.

Розглянемо ключові API-ендпоінти нашої системи.

Шляхи **Auth** відповідають за управління доступом користувачів. Маршрут POST /auth/register використовується для реєстрації нового користувача, вимагаючи email, password та displayName.

Вхід до системи реалізується через POST /auth/login, який успішно повертає JWT-токен для подальшої роботи та об’єкт профілю користувача. Деталі поточного автентифікованого користувача можна отримати за допомогою GET /auth/me.

Шляхи **Posts** забезпечують повний функціонал для роботи з контентом. Публічний список постів доступний через GET /posts, який підтримує пагінацію, пошук та фільтри.

Отримати деталі конкретного поста можна за допомогою GET /posts/:id – це дозволено, якщо пост опублікований або якщо запит надсилає його власник чи адміністратор. Автори можуть переглядати власні пости (включно з чернетками) через маршрут GET /posts/mine.

Створення нового поста реалізується через POST /posts, а оновлення та видалення існуючого поста PUT /posts/:id, DELETE /posts/:id дозволено власнику поста або Адміністратору. Для зміни стану поста використовується PATCH /posts/:id/status.

Для опису нашого API використовується Swagger UI. Цей інструмент надає інтерактивний інтерфейс для ознайомлення з ендпойнтами, тестування запитів і верифікації вхідних параметрів.

Приклад опису API у Swagger наведено на рисунку 3.12.

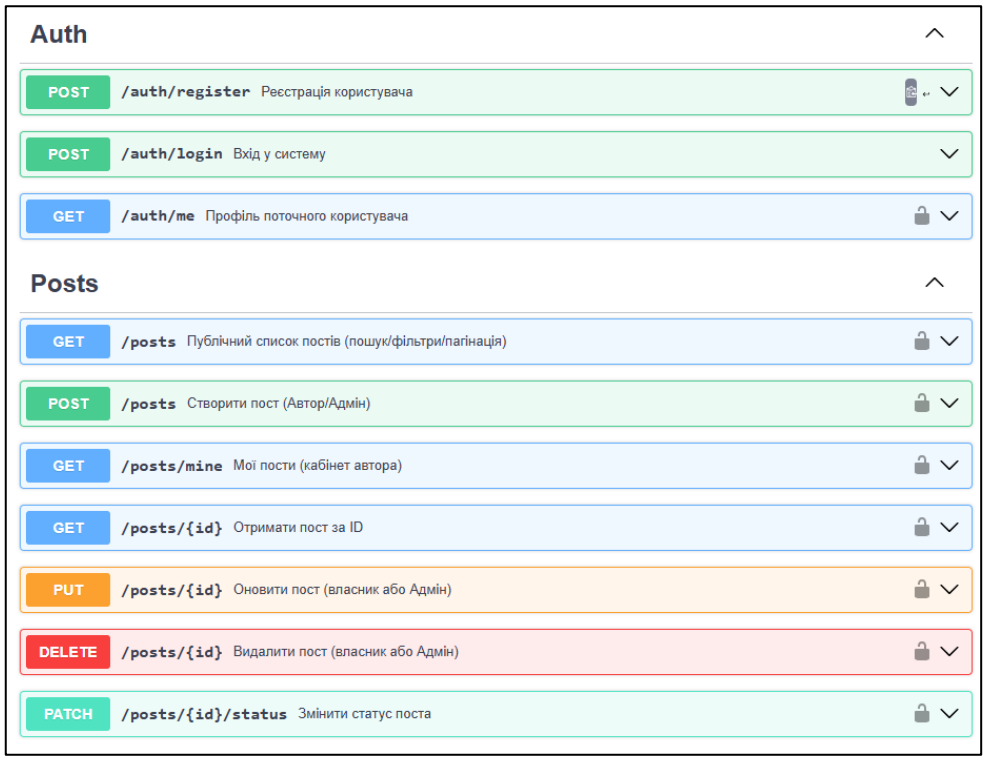


Рисунок 3.12 – Опис API у Swagger

3.5 Тестування проєкту

Angular CLI створює оточення для модульного тестування компонентів/сервісів: TestBed, хости, фікстури, async/await тощо [1, 3].

Для забезпечення якості та стабільності системи використовується комплексний підхід до тестування, що охоплює як клієнтську, так і серверну частину проєкту.

Клієнтська частина проєкту тестується за допомогою комбінації фреймворків [1, 3]:

– Jasmine – це BDD-фреймворк, незалежний від браузера, який надає функціонал для створення шпигунів, порівнювачів та підтримує асинхронне тестування;
– Karma – це тестовий раннер, який запускає вебсервер, підключає реальні браузери, виконує тести та відображає результати (це значно полегшує розробку через тестування та перевірку сумісності в різних браузерних середовищах).

На рисунку 3.13 наведено приклад тесту для PostEditorComponent.

```
it('створення: save() викликає create() з коректним DTO і робить navigate', () => {
  fixture.detectChanges();

  component.form.patchValue({
    title: 'My post',
    status: 'DRAFT',
    categorySlugs: ['tech', 'news'],
  });
  component.markdown = 'Hello **MD**';

  component.save();

  expect(md.toHtml).toHaveBeenCalledWith('Hello **MD**');
  expect(posts.create).toHaveBeenCalledTimes(1);

  const dto :any  = posts.create.calls.mostRecent().args[0];
```

Рисунок 3.13 – Приклад тесту для PostEditorComponent

Тести запускаються за допомогою команди ng test у корені клієнтської директорії. Результати виконання тестів автоматично відображаються у браузері за адресою http://localhost:9876, де індикатори пройдених/провалених специфікацій надають швидкий зворотний зв’язок (рис. 3.14) [1, 3].

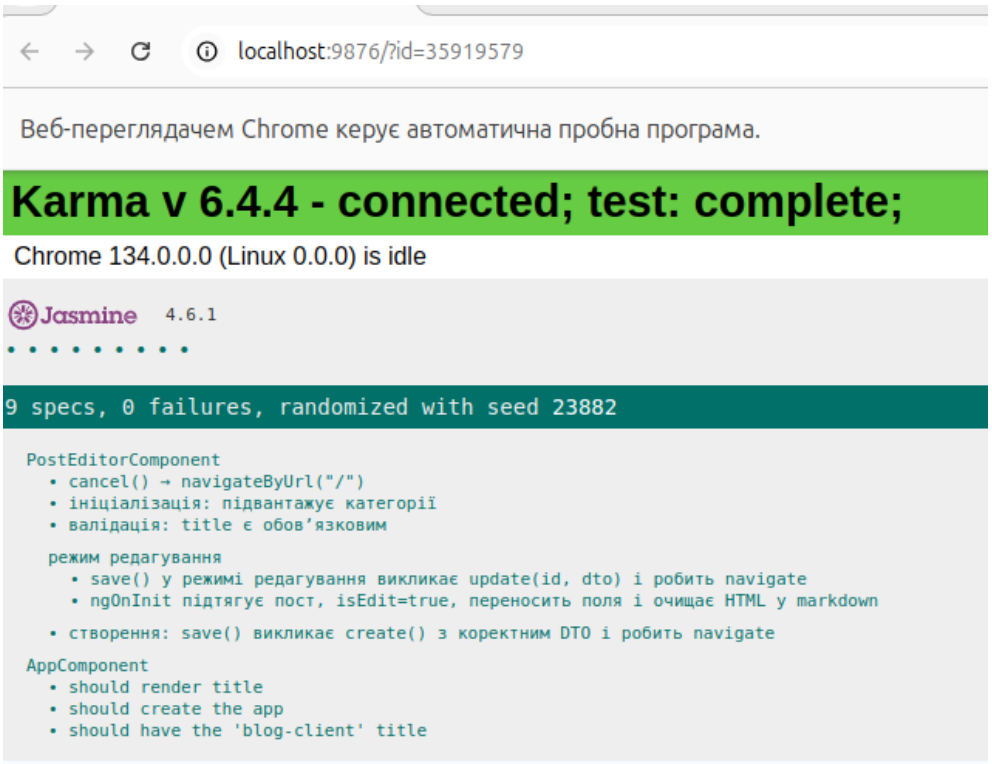


Рисунок 3.14 – Результат тестування

Тестування гарантує стабільність ключової бізнес-логіки, прогнозованість взаємодії з API та узгодженість усіх правил доступу та ролей [3].

Детально ознайомитися з тестами PostEditorComponent можна в додатку Д.

3.6 Керівництво користувача

3.6.1 Вхід та реєстрація

Після переходу на сторінку вебзастосунку, система відображає головну сторінку зі списком опублікованих постів (рис. 3.15).

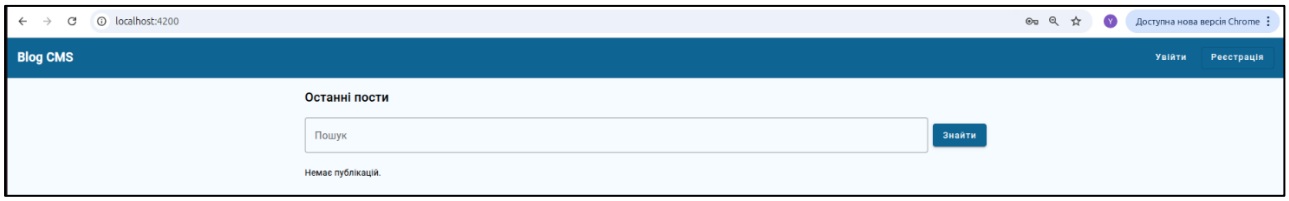


Рисунок 3.15 – Головна сторінка

За замовчуванням користувач може переглядати опубліковані пости та здійснювати пошук.

Для доступу до функцій створення та керування постами необхідно увійти або зареєструватися в системі. Якщо користувач не має облікового запису, то необхідно натиснути на кнопку «Реєстрація» у правому верхньому кутку верхньої панелі (рис. 3.16).

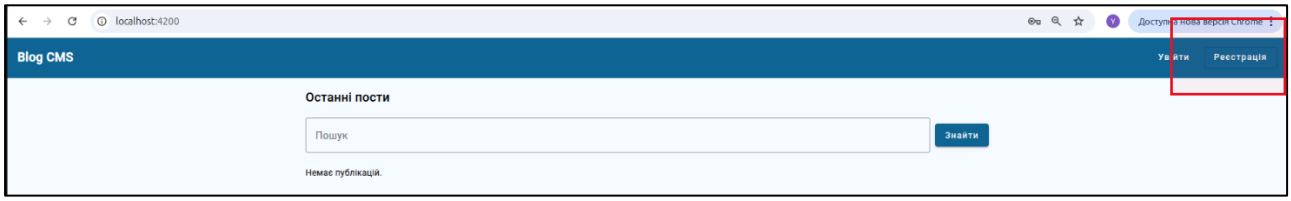


Рисунок 3.16 – Кнопка «Реєстрація»

Система відображає форму реєстрації (рис. 3.17).

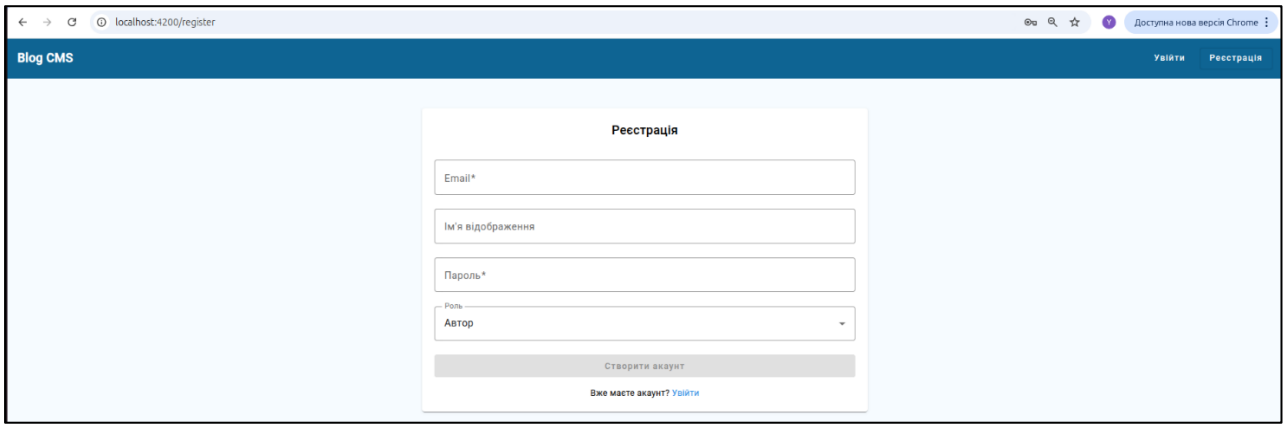


Рисунок 3.17 – Форма реєстрації

Користувач заповнює поля форми. Система автоматично застосовує правила валідації та повідомляє користувача про допущену помилку (рис. 3.18).

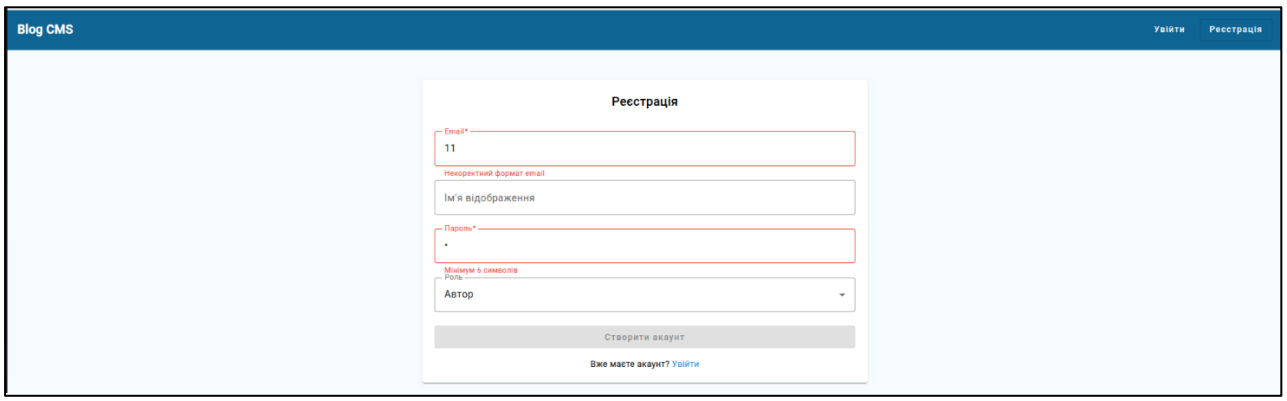


Рисунок 3.18 – Невалідні дані

Якщо поля заповнені валідними даними, то система розблоковує кнопку «Створити акаунт» (рис. 3.19).

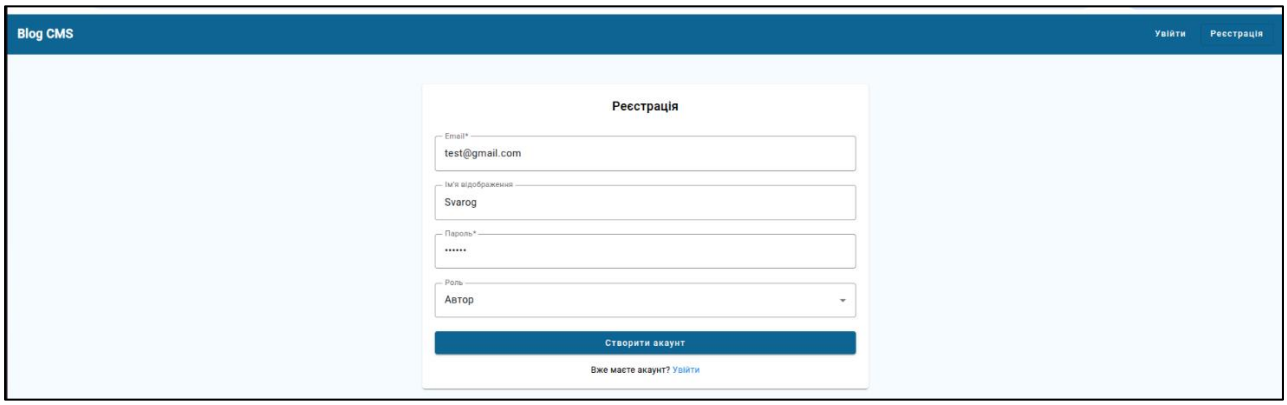


Рисунок 3.19 – Валідна форма

Користувач натискає на кнопку «Створити акаунт». Якщо користувача з вказаним email вже зареєстровано, то система повідомляє про це відповідним повідомленням (рис. 3.20).

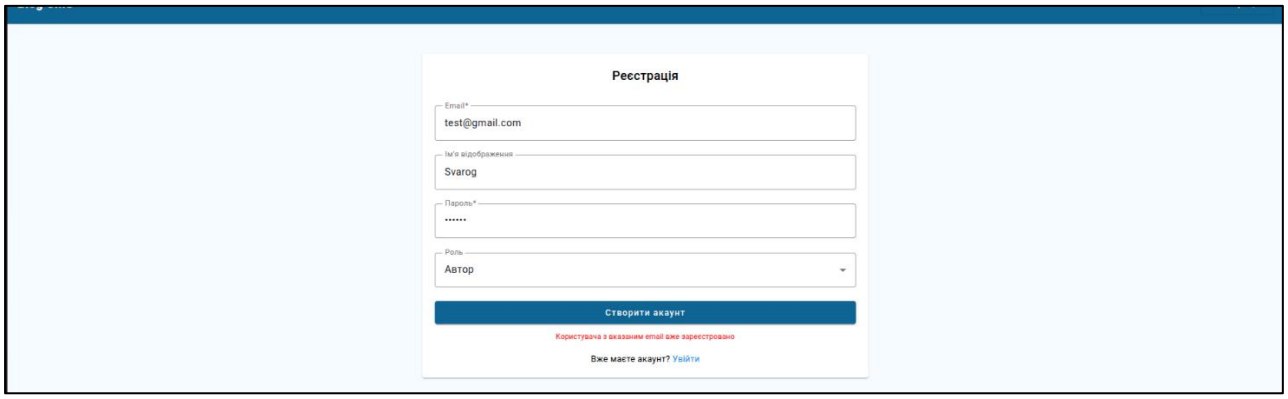


Рисунок 3.20 – Аккаунт вже існує

Після успішної реєстрації або у випадку, якщо обліковий запис вже існує, користувач натискає на кнопку «Увйти», система відображає форму входу для якої діють ті самі правила валідації та перевірки облікового запису (див. рис. 3.21 – 3.22).

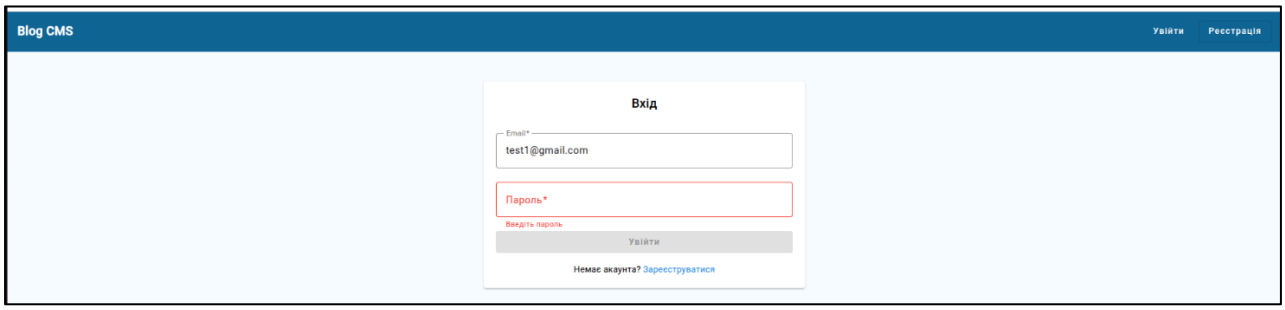


Рисунок 3.21 – Невалідна форма входу

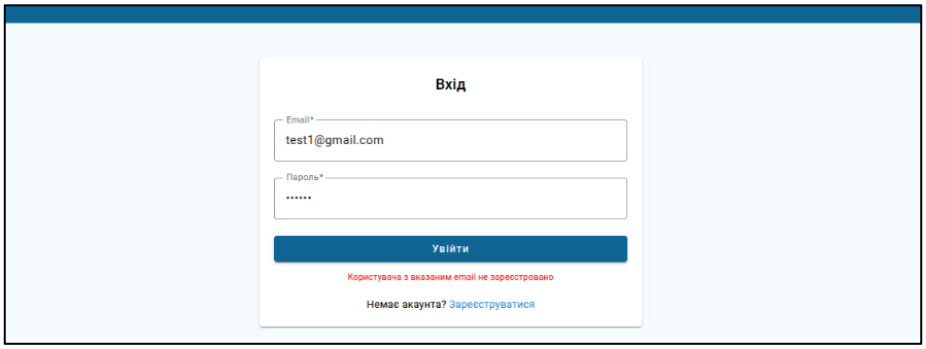


Рисунок 3.22 – Хибні облікові дані

Після успішного входу, система відображає кабінет користувача зі списком опублікованих постів, полем пошуку та функціональними можливостями відповідно до ролі (рис. 3.23).

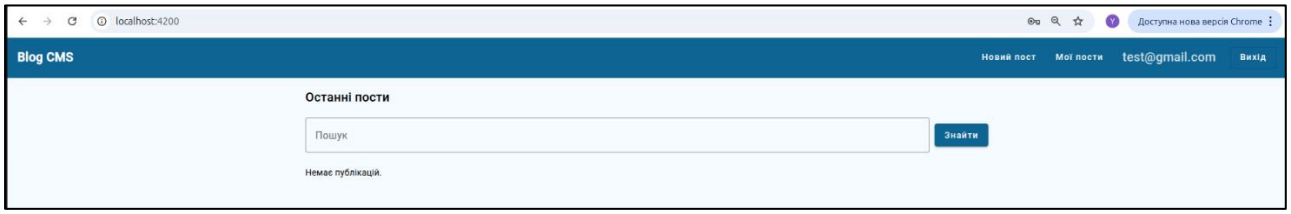


Рисунок 3.23 – Особистий кабінет

3.6.2 Створення та редагування поста

Після успішного входу в систему, користувач з роллю «Автор» отримує доступ до функціоналу керування власними публікаціями.

Для створення нового контенту необхідно з будь-якої частини застосунку натиснути на кнопку «Новий пост» (рис. 3.24).

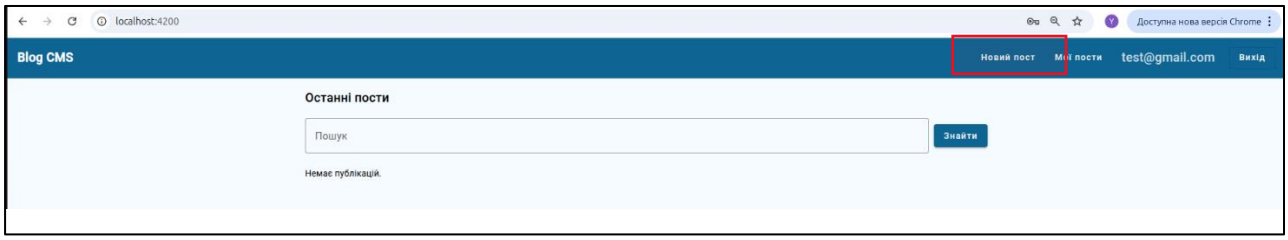


Рисунок 3.24 – Кнопка створення поста

Система відображає форму створення нового поста, де користувач має заповнити обов’язкові поля: заголовок та контент. Також можна обрати необхідні категорі. Для поста необхідно задати статус, обравши між «Чернетка» та «Опубліковано» (рис. 3.25).

Для форматування контенту використовується Markdown-редактор, що має панель інструментів над текстовою областю. Ця панель інструментів дозволяє швидко застосовувати основні стилі форматування:

– жирний, курсив, підкреслення;
– заголовки h1–h3;
– зміна кольорів тексту та тла;
– форматування цитат, нумерованих/маркованих списків;
– вставка коду;
– створення посилань;
– додавання медіаконтенту.

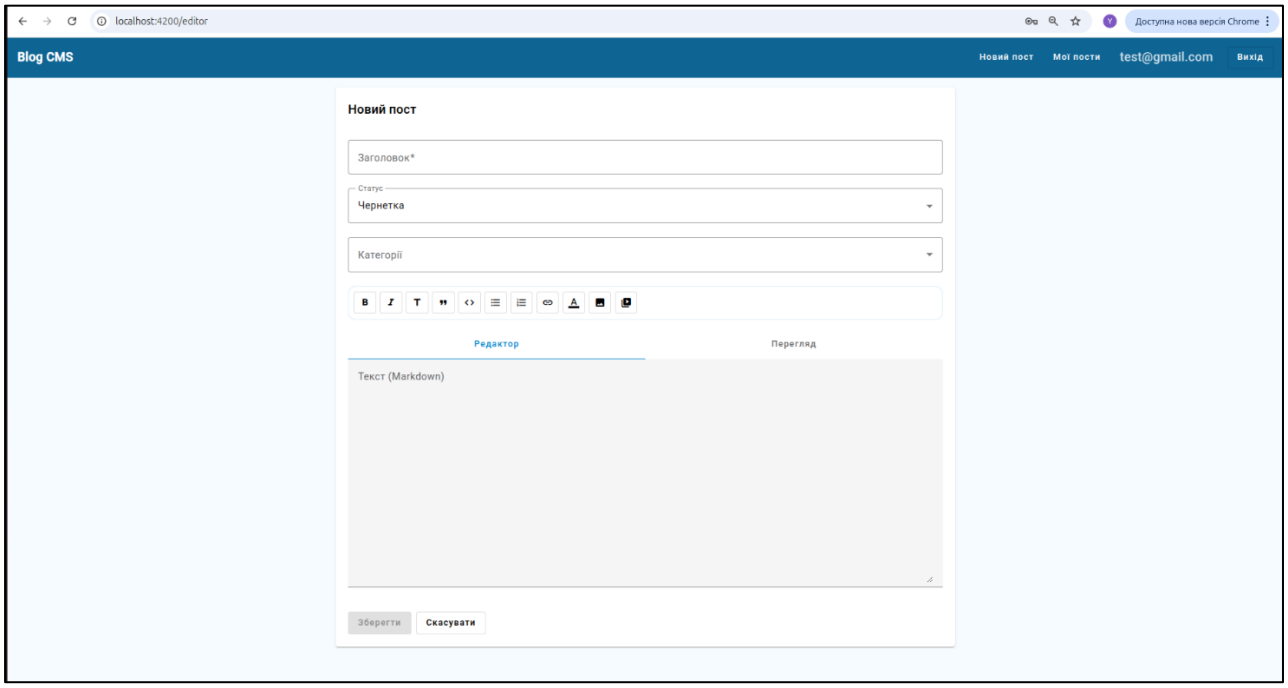


Рисунок 3.25 – Форма створення поста

У Markdown-редакторі стилі форматування застосовуються, як відповідні обгортки (рис. 3.26). Для перегляду приближенної до публікації версії, необхідно обрати функцію попереднього перегляду (рис. 3.27).

Після попереднього перегляду поста, користувач може натиснути на кнопку «Зберегти».

В залежності від обраного статусу пост або буде збережений, але залишиться невидимим для інших читачів, або з’явиться у загальній стрічці та буде доступний для пошуку. Система зберігає пост та перенаправляє на сторінку повного перегляду поста (рис. 3.28).

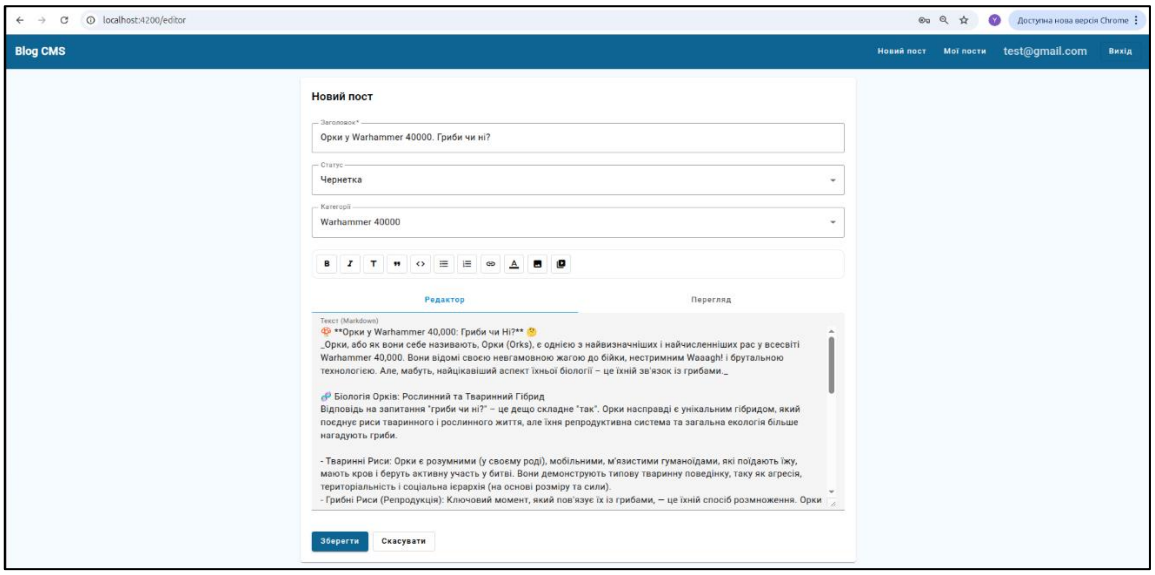


Рисунок 3.26 – Створення контента у редакторі

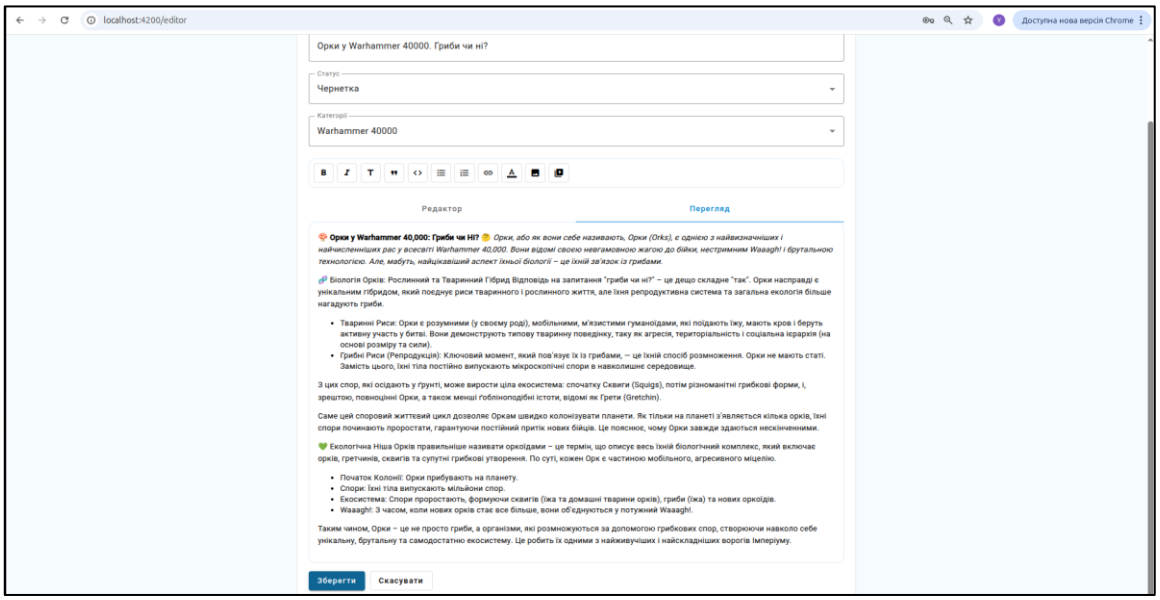


Рисунок 3.27 – Попередній перегляд

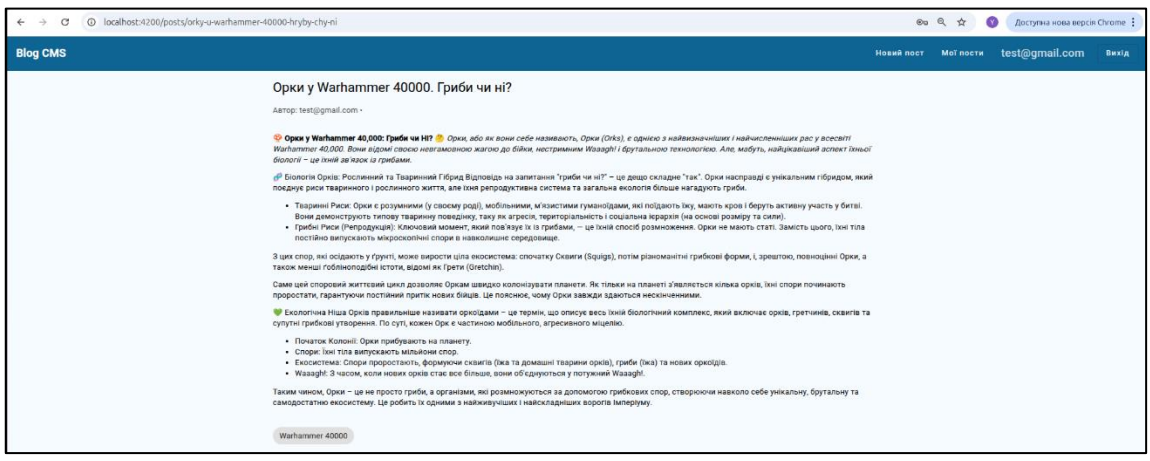


Рисунок 3.28 – Сторінка поста

Якщо пост було створено у статусі «Чернетка» або необхідно відкорегувати його вміст, то користувач натискає на кнопку «Мої пости». Система відображає сторінку зі списком постів користувача та елементами керування (рис. 3.29).

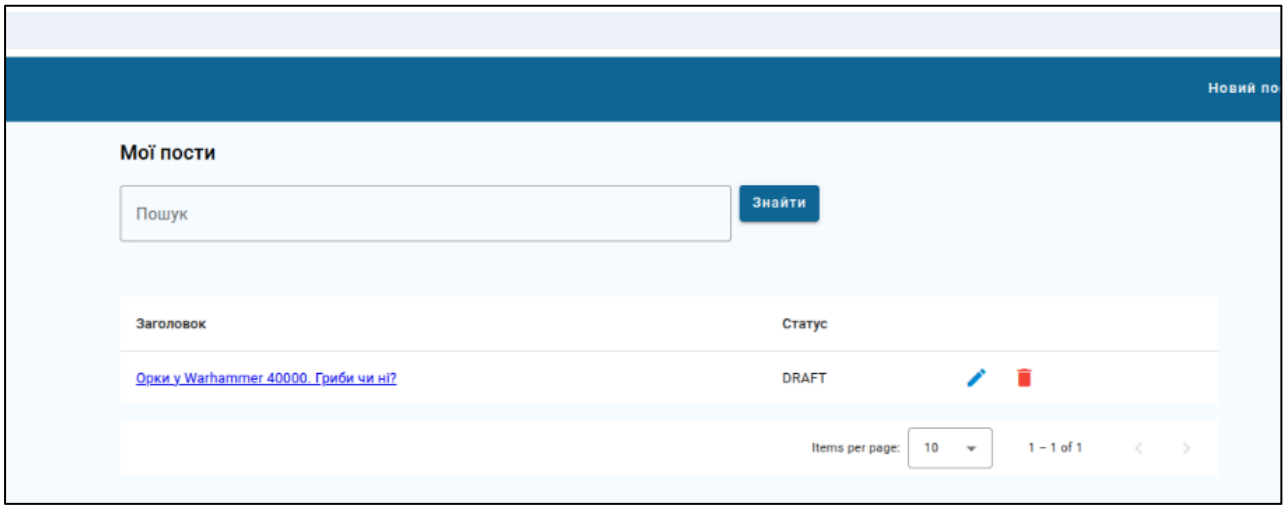


Рисунок 3.29 – Сторінка «Мої пости»

Для редагування користувачу необхідно натиснути на піктограми «Редагувати». Система відображає форму редагування (рис. 3.30).

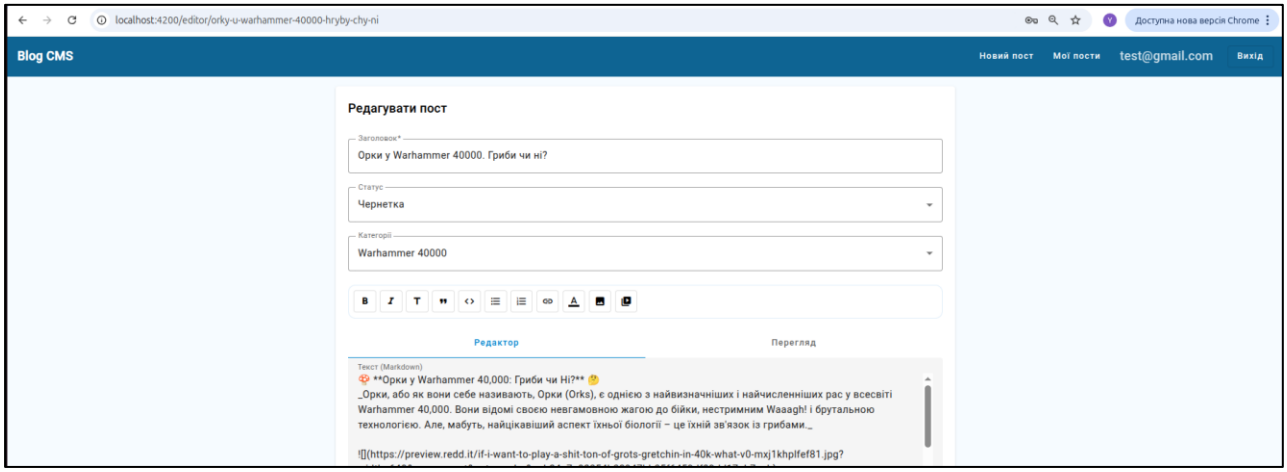


Рисунок 3.30 – Форма редагування поста

Після створення або редагування поста і за умови, що встановлено статус «Публікація», він стає доступним всім користувачам на сторінці списку постів (рис. 3.31).

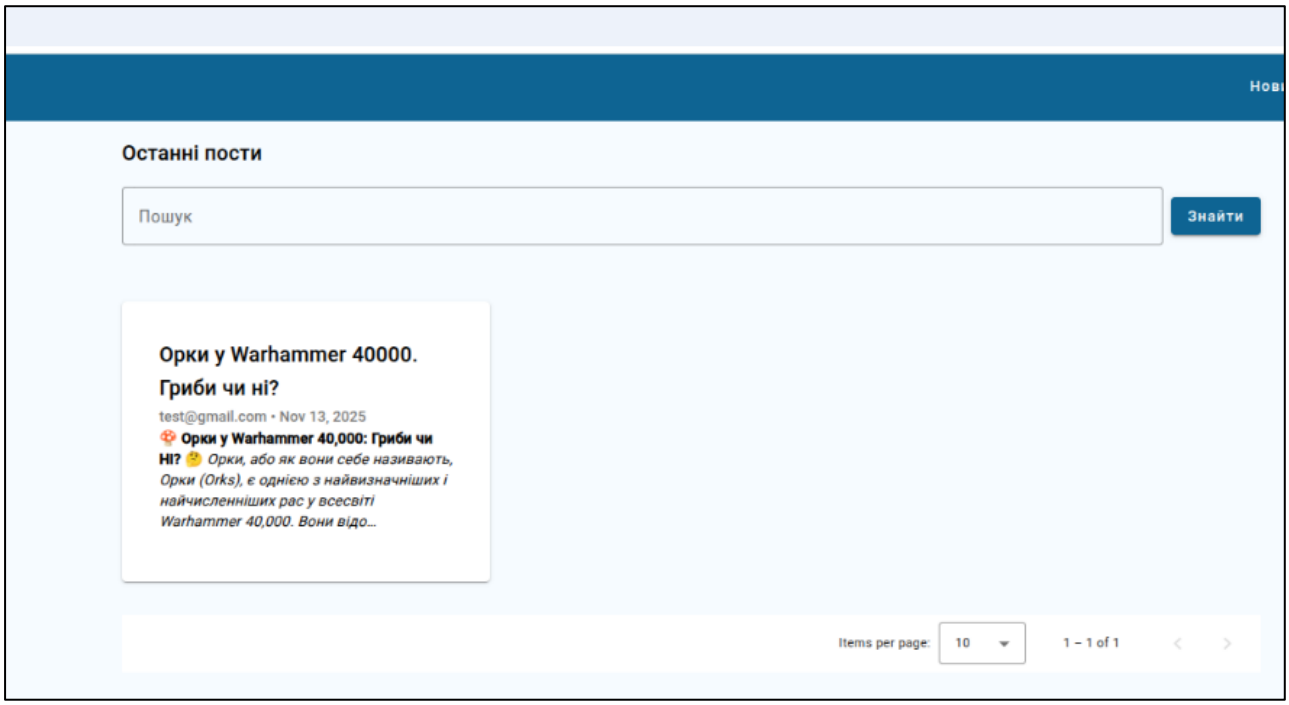


Рисунок 3.31 – Список опублікованих постів

3.6.3 Взаємодія з медіа

Для інтеграції зображень чи відео необхідно натиснути на одну з піктограм пов’язану з медіа. Після завантаження файлу, система повертає URL-адресу медіа-ресурсу. Ця URL-адреса вставляється у текстову область Markdown-редактора за відповідним шаблоном або за допомогою спеціального діалогового вікна (рис. 3.32 – 3.33).

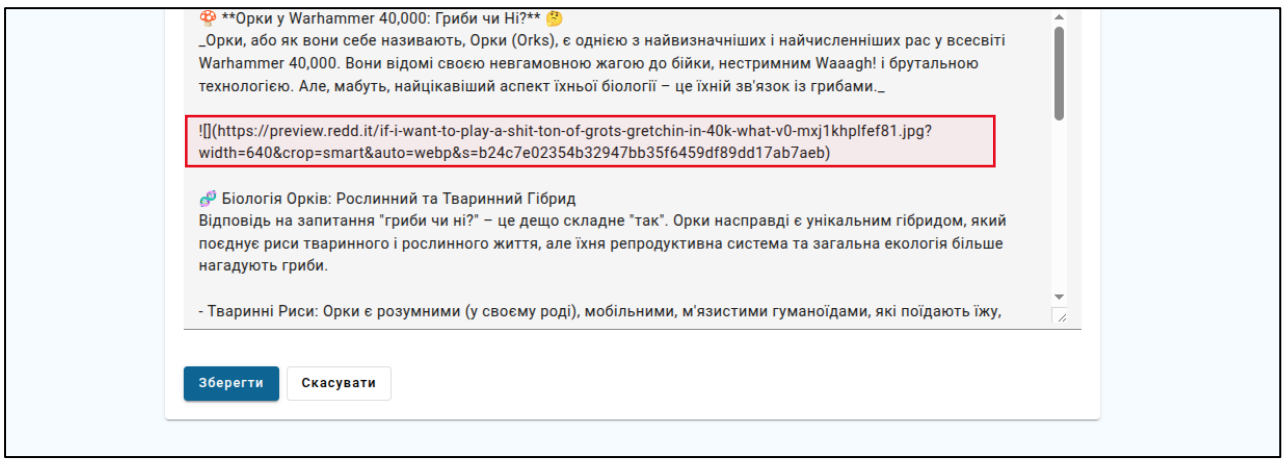


Рисунок 3.32 – Вставка медіа

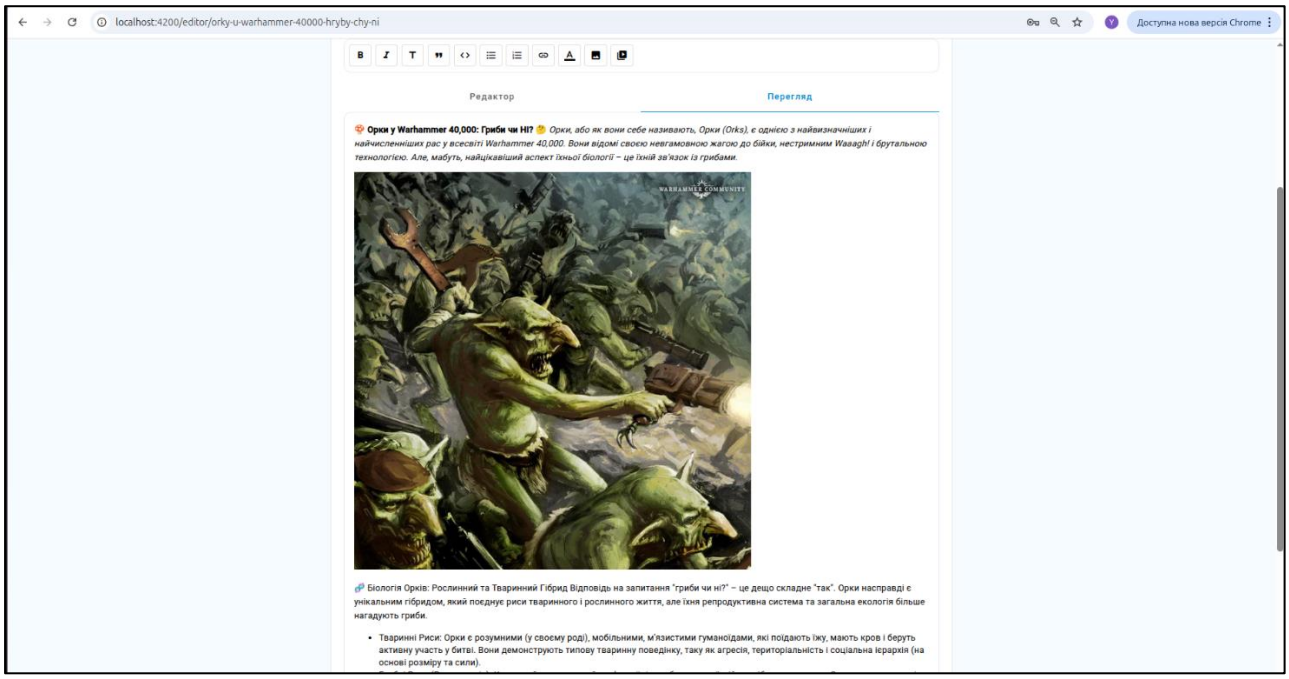


Рисунок 3.33 – Попередній перегляд поста з медіа

3.6.4 Пошук постів

Пошук постів здійснюється за назвою або slug. Для цього користувач на сторінці списку опублікованих постів або на сторінці «Мої пости» вводить відповідне значення у поле пошуку та натискає на кнопку «Знайти». В залежності від результату пошуку, система відображає або список постів, або напис «Немає публікацій» (рис. 3.34 – 3.35).

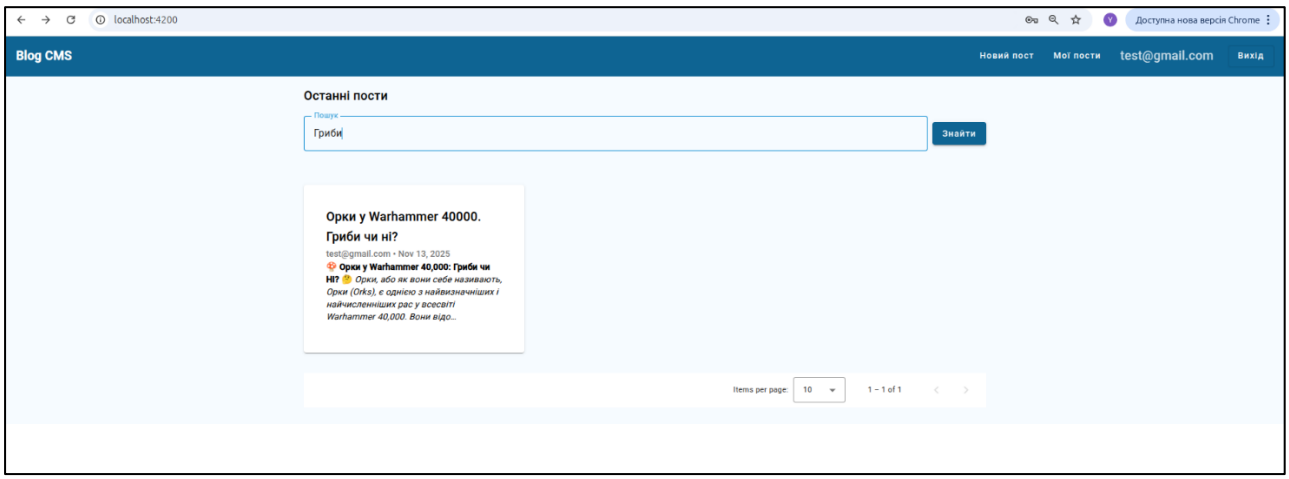


Рисунок 3.34 – Пошук з результатом

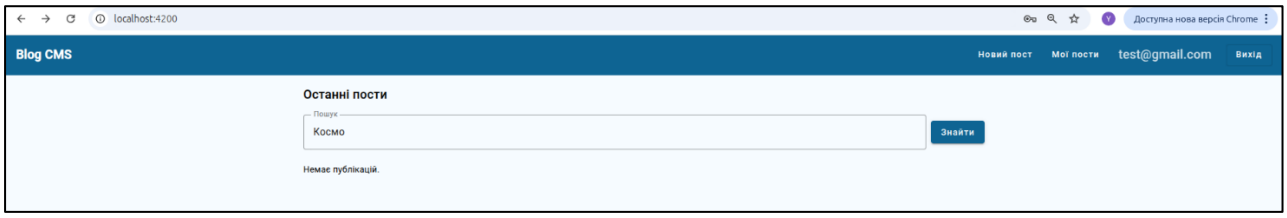


Рисунок 3.35 – Відсутні результати пошуку

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено систему управління контекстом для ведення блогу. Система надає можливість авторам легко та швидко створювати пости у форматі Markdown з попереднім переглядом, керувати категоріями та статусами, а адміністраторам керувати користувачами та їх ролями. Клієнт-серверна архітектура забезпечує гнучкість, масштабованість та підтримуваність системи.

У відповідності з метою кваліфікаційної роботи було розроблено систему управління контентом для ведення блогу з використанням таких технологій:

- Angular для реалізації клієнтської частини;
- Node.js з Express.js для реалізації серверної частини та REST API;
- MySQL для зберігання даних.

У відповідності з поставленими задачами були виконані наступні етапи створення платформи:

- сформульовано функціональні та нефункціональні вимоги до системи;
- проаналізовано та порівняно існуючі системи управління контентом;
- спроєктувано архітектуру системи, включаючи розробку необхідних UML-діаграм (варіантів використання, діяльності, послідовності, розгортання) та надано опис REST API;
- реалізувано клієнтську частину засобами Angular та серверну частину на Node.js з Express, для збереження даних використано MySQ;
- протестовано розроблену систему та надано керівництво користувача.

ПЕРЕЛІК ПОСИЛАНЬ

1. Angular Developer Documentation. URL: https://angular.io/docs/ (дата звернення: 21.08.2025).

2. Angular Material Guide. URL: https://material.angular.dev/guides (дата звернення: 02.09.2025).

3. Bampakos A., Deeleman P. Learning Angular: A no-nonsense guide to building web applications with Angular 15. Birmingham : Packt Publishing, 2023. 446 p.

4. Chebbi L. Reactive Patterns with RxJS for Angular: A practical guide to managing your Angular application’s data reactively and efficiently using RxJS 7. Birmingham : Packt Publishing, 2022. 224 p.

5. Express.js Guide. URL: https://expressjs.com/ (дата звернення: 14.08.2025).

6. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 2003. 208 p.

7. Ghost. URL: https://ghost.org/ (дата звернення: 01.07.2025).

8. Mrzyglocki F. UML Summarized: Key Concepts and Diagrams for Software Engineers, Architects, and Designers. Phillip Mrzyglocki, 2023. 59 p.

9. Medium. URL: https://medium.com/ (дата звернення: 01.07.2025).

10. MySQL Documentation. URL: https://dev.mysql.com/doc/ (дата звернення: 15.08.2025).

11. Node.js Documentation. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 14.08.2025).

12. npm Docs. URL: https://docs.npmjs.com/ (дата звернення: 14.08.2025).

13. npm jsonwebtoken. URL: https://www.npmjs.com/package/jsonwebtoken (дата звернення: 14.08.2025).

14. Postman. API documentation. URL: https://www.postman.com/api-platform/api-documentation/ (дата звернення: 14.08.2025).

15. RxJS Documentation. URL: https://rxjs.dev/api (дата звернення: 04.09.2025).

16. Substack. URL: https://substack.com/ (дата звернення: 01.07.2025).

17. Understanding the MVC Pattern with Node.js. URL: https://dev.to/abhishekjaiswal_4896/understanding-the-mvc-pattern-with-nodejs-1fpg (дата звернення: 14.08.2025).

18. Visual Paradigm. What is Unified Modeling Language (UML)? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/ (дата звернення: 07.08.2025).

19. Visual Paradigm. Use Case Diagram Guide. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення: 07.08.2025).

20. WordPress. URL: https://uk.wordpress.org/ (дата звернення: 01.07.2025).

ДОДАТОК А

Angular PostEditorComponent

```
import { Component, OnInit, inject } from '@angular/core';
import { FormBuilder, ReactiveFormsModule, Validators } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import {CommonModule, NgIf} from '@angular/common';
import { MatCardModule } from '@angular/material/card';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatSelectModule } from '@angular/material/select';
import { MatButtonModule } from '@angular/material/button';
import { PostService } from '../../core/services/post.service';
import { MarkdownService } from '../../core/services/markdown.service';
import { MarkdownEditorComponent } from '../../shared/components/markdown-editor.component';
import { Post } from '../../models/post';
import {Category} from '../../models/category';
import { CategoryService } from '../../core/services/category.service';

@Component({
  standalone: true,
  imports: [ReactiveFormsModule, CommonModule, MatCardModule, MatFormFieldModule,
MatInputModule, MatSelectModule, MatButtonModule, MarkdownEditorComponent],
  template: `
    <mat-card class="mt-4" style="max-width:1000px;margin:0 auto;">
      <h2>{{ isEdit ? 'Редагувати пост' : 'Новий пост' }}</h2>

      <form [formGroup]="form" (ngSubmit)="save()" class="mt-2" novalidate>
        <div class="form-grid">
          <mat-form-field appearance="outline">
            <mat-label>Заголовок</mat-label>
            <input matInput formControlName="title" required>
            <mat-error *ngIf="form.get('title')?.hasError('required')">Введіть заголовок</mat-error>
          </mat-form-field>

          <mat-form-field appearance="outline">
            <mat-label>Статус</mat-label>
```

```
          <mat-select formControlName="status">
            <mat-option value="DRAFT">Чернетка</mat-option>
            <mat-option value="PUBLISHED">Публікація</mat-option>
          </mat-select>
        </mat-form-field>
      </div>

      <mat-form-field appearance="outline" class="w-100">
        <mat-label>Категорії</mat-label>
        <mat-select formControlName="categorySlugs" multiple>
          <mat-option *ngFor="let c of categories" [value]="c.slug">{{ c.name }}</mat-option>
        </mat-select>
      </mat-form-field>
      <app-markdown-editor [(value)]="markdown"></app-markdown-editor>
      <div style="display:flex; gap:8px; flex-wrap: wrap;" class="mt-2">
        <button mat-raised-button color="primary" [disabled]="form.invalid ||
!markdown.trim()">Зберегти</button>
        <button mat-stroked-button type="button" (click)="cancel()">Скасувати</button>
      </div>
    </form>
  </mat-card>
  `
})
export class PostEditorComponent implements OnInit {
  private fb = inject(FormBuilder);
  private api = inject(PostService);
  private md = inject(MarkdownService);
  private route = inject(ActivatedRoute);
  private router = inject(Router);
  private categoryApi = inject(CategoryService);
  isEdit = false;
  post!: Post;
  markdown = '';

  categories: Category[] = [];


  form = this.fb.group({
    title: ['', Validators.required],
    status: ['DRAFT'],
```

```
    categorySlugs: this.fb.control<string[]>([])
  });


  ngOnInit() {
    this.categoryApi.list().subscribe(c => this.categories = c);
    const slug = this.route.snapshot.paramMap.get('slug');
    if (slug) {
      this.isEdit = true;
      this.api.getBySlug(slug).subscribe(p => {
        this.post = p;
        this.form.patchValue({
          title: p.title,
          status: p.status,
          categorySlugs: (p.categories || []).map(c => c.slug)
        });
        this.markdown = (p.body || '').replace(/<[^>]+>/g, '');
      });
    }
  }
  save() {
    const dto = {
      title: this.form.value.title!,
      body: this.md.toHtml(this.markdown || ''),
      status: this.form.value.status as 'DRAFT'|'PUBLISHED',
      categories: this.form.value.categorySlugs || []
    };
    if (this.isEdit) {
      this.api.update(this.post.id, dto).subscribe(p => this.router.navigate(['/posts', p.slug]));
    } else {
      this.api.create(dto).subscribe(p => this.router.navigate(['/posts', p.slug]));
    }
  }
  cancel() { this.router.navigateByUrl('/'); }
}
```

ДОДАТОК Б

Angular PostService

```
import { Injectable, inject } from '@angular/core';
import { HttpClient, HttpParams } from '@angular/common/http';
import { environment } from '../../environments/environment';
import { Paged } from '../../models/api';
import { Post } from '../../models/post';

@Injectable({ providedIn: 'root' })
export class PostService {
  private http = inject(HttpClient);
  private base = `${environment.apiUrl}/posts`;
  list(opts: { page?: number; limit?: number; search?: string; status?: 'PUBLISHED'|'DRAFT'; category?:
string } = {}) {
    let params = new HttpParams();
    Object.entries(opts).forEach(([k,v]) => { if (v !== undefined && v !== null && v !== '') params =
params.set(k, String(v)); });
    return this.http.get<Paged<Post>>(this.base, { params });
  }
  listMine(opts: { page?: number; limit?: number; search?: string; status?: 'PUBLISHED'|'DRAFT' } = {}) {
    let params = new HttpParams();
    Object.entries(opts).forEach(([k,v]) => { if (v !== undefined && v !== null && v !== '') params =
params.set(k, String(v)); });
    return this.http.get<Paged<Post>>(`${this.base}/mine`, { params });
  }
  getBySlug(slug: string) { return this.http.get<Post>(`${this.base}/${slug}`); }
  create(dto: { title: string; body: string; status?: 'DRAFT'|'PUBLISHED'; categories?: string[] }) {
    return this.http.post<Post>(this.base, dto);
  }
  update(id: number, dto: Partial<{ title: string; body: string; status: 'DRAFT'|'PUBLISHED'; categories:
string[] }>) {
    return this.http.patch<Post>(`${this.base}/${id}`, dto);
  }
  remove(id: number) { return this.http.delete<{ message: string }>(`${this.base}/${id}`); }
}
```

ДОДАТОК В

Express Post Controller

```
const { Op } = require('sequelize');
const Post = require('../models/Post');
const Category = require('../models/Category');
const User = require('../models/User');
const makeSlug = require('../utils/slug');


async function list(req, res) {
  const { page = 1, limit = 10, search = '', status = 'PUBLISHED', category } = req.query;
  const where = {};
  if (status) where.status = status;
  if (search) where.title = { [Op.substring]: search };


  const include = [
    { model: User, as: 'author', attributes: ['id', 'email', 'display_name'] },
    { model: Category, through: { attributes: [] } }
  ];


  if (category) {
    include[1].where = { slug: category };
  }


  const result = await Post.findAndCountAll({
    where,
    include,
    order: [['created_at', 'DESC']],
    limit: Number(limit),
    offset: (Number(page) - 1) * Number(limit)
  });


  return res.json({ total: result.count, items: result.rows });
```

```
}

async function getBySlug(req, res) {
  const { slug } = req.params;
  const post = await Post.findOne({
    where: { slug },
    include: [
      { model: User, as: 'author', attributes: ['id', 'email', 'display_name'] },
      { model: Category, through: { attributes: [] } }
    ]
  });
  if (!post) return res.status(404).json({ message: 'Post not found' });
  return res.json(post);
}


async function create(req, res) {
  const { title, body, status = 'DRAFT', categories = [] } = req.body;
  const slug = makeSlug(title);

  const exists = await Post.findOne({ where: { slug } });
  if (exists) return res.status(409).json({ message: 'Post with same slug exists' });

  const payload = {
    title,
    slug,
    body,
    status,
    author_id: req.user.id,
    published_at: status === 'PUBLISHED' ? new Date() : null
  };

  const post = await Post.create(payload);


  if (Array.isArray(categories) && categories.length) {
    const cats = await Category.findAll({ where: { slug: categories } });
    await post.setCategories(cats);
  }

  return res.status(201).json(post);
```

```
}

async function update(req, res) {
  const { id } = req.params;
  const { title, body, status, categories } = req.body;
  const post = await Post.findByPk(id, { include: [{ model: Category }] });
  if (!post) return res.status(404).json({ message: 'Post not found' });


  if (req.user.role !== 'ADMIN' && post.author_id !== req.user.id) {
    return res.status(403).json({ message: 'Forbidden' });
  }


  if (title) {
    post.title = title;
    post.slug = makeSlug(title);
  }
  if (body) post.body = body;
  if (status) {
    post.status = status;
    if (status === 'PUBLISHED' && !post.published_at) post.published_at = new Date();
    if (status === 'DRAFT') post.published_at = null;
  }


  await post.save();


  if (Array.isArray(categories)) {
    const cats = await Category.findAll({ where: { slug: categories } });
    await post.setCategories(cats);
  }


  return res.json(post);
}

async function remove(req, res) {
  const { id } = req.params;
  const post = await Post.findByPk(id);
```

```
  if (!post) return res.status(404).json({ message: 'Post not found' });
  if (req.user.role !== 'ADMIN' && post.author_id !== req.user.id) return res.status(403).json({ message:
'Forbidden' });
  await post.destroy();
  return res.json({ message: 'Deleted' });
}

async function mine(req, res) {
  const { page = 1, limit = 10, status, search = '' } = req.query;
  const where = { author_id: req.user.id };
  if (status) where.status = status;
  if (search) where.title = { [Op.substring]: search };

  console.log(req.query);

  const include = [
    { model: User, as: 'author', attributes: ['id','email','display_name'] },
    { model: Category, through: { attributes: [] } }
  ];

  const result = await Post.findAndCountAll({
    where, include, order: [['created_at','DESC']],
    limit: Number(limit), offset: (Number(page) - 1) * Number(limit)
  });

  return res.json({ total: result.count, items: result.rows });
}

module.exports = { list, getBySlug, create, update, remove, mine };
```

ДОДАТОК Г

Express Post Model

```
const { DataTypes, Model } = require('sequelize');
const { sequelize } = require('../config/db');
const User = require('./User');


class Post extends Model {}


Post.init({
   id: { type: DataTypes.INTEGER.UNSIGNED, primaryKey: true, autoIncrement: true },
   title: { type: DataTypes.STRING(191), allowNull: false },
   slug: { type: DataTypes.STRING(191), allowNull: false, unique: true },
   body: { type: DataTypes.TEXT('long'), allowNull: false },
   status: { type: DataTypes.ENUM('DRAFT', 'PUBLISHED'), allowNull: false, defaultValue: 'DRAFT' },
   published_at: { type: DataTypes.DATE, allowNull: true }
}, { sequelize, modelName: 'post' });


Post.belongsTo(User, { as: 'author', foreignKey: { name: 'author_id', allowNull: false }, onDelete:
'CASCADE' });
User.hasMany(Post, { as: 'posts', foreignKey: 'author_id' });


module.exports = Post;
```

ДОДАТОК Д

Unit-test PostEditorComponent

```
import { NO_ERRORS_SCHEMA, Component, Directive, EventEmitter, Input, Output } from
'@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ControlValueAccessor, NG_VALUE_ACCESSOR } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { CommonModule } from '@angular/common';
import { ReactiveFormsModule } from '@angular/forms';
import { of } from 'rxjs';

import { PostEditorComponent } from './post-editor.component';
import { PostService } from '../../core/services/post.service';
import { MarkdownService } from '../../core/services/markdown.service';
import { CategoryService } from '../../core/services/category.service';
import {provideHttpClient, withInterceptors} from '@angular/common/http';
import {authInterceptor} from '../../core/interceptors/auth.interceptor';

@Component({
  selector: 'app-markdown-editor',
  standalone: true,
  template: '',
})
class MarkdownEditorStubComponent {
  @Input() value = '';
  @Output() valueChange = new EventEmitter<string>();
}

@Directive({
  selector: 'mat-select',
  standalone: true,
  providers: [{ provide: NG_VALUE_ACCESSOR, useExisting: FakeMatSelectDirective, multi: true }],
})
class FakeMatSelectDirective implements ControlValueAccessor {
  private onChange = (_: any) => {};
  private onTouched = () => {};
  writeValue(_: any): void {}
```

```
  registerOnChange(fn: any): void { this.onChange = fn; }
  registerOnTouched(fn: any): void { this.onTouched = fn; }
  setDisabledState?(_: boolean): void {}
}


class PostServiceMock {
  create = jasmine.createSpy('create').and.callFake((dto: any) =>
    of({ id: 101, slug: 'hello-post', ...dto })
  );
  update = jasmine.createSpy('update').and.callFake((id: number, dto: any) =>
    of({ id, slug: 'updated-post', ...dto })
  );
  getBySlug = jasmine.createSpy('getBySlug').and.callFake((slug: string) =>
    of({
      id: 42,
      slug,
      title: 'Old title',
      status: 'PUBLISHED',
      body: '<p>Hi</p>',
      categories: [{ name: 'Tech', slug: 'tech' }],
    })
  );
}

class MarkdownServiceMock {
  toHtml = jasmine.createSpy('toHtml').and.callFake((md: string) => `<p>${md}</p>`);
}

class CategoryServiceMock {
  list = jasmine.createSpy('list').and.returnValue(
    of([
      { id: 1, name: 'Tech', slug: 'tech' },
      { id: 2, name: 'News', slug: 'news' },
    ])
  );
}

class RouterMock {
  navigate = jasmine.createSpy('navigate');
  navigateByUrl = jasmine.createSpy('navigateByUrl');
```

```
}


let routeSlug: string | null = null;
const ActivatedRouteStub = {
  snapshot: {
    paramMap: {
      get: (key: string) => (key === 'slug' ? routeSlug : null),
    },
  },
};

describe('PostEditorComponent', () => {
  let fixture: ComponentFixture<PostEditorComponent>;
  let component: PostEditorComponent;

  let posts: PostServiceMock;
  let md: MarkdownServiceMock;
  let cats: CategoryServiceMock;
  let router: RouterMock;

  beforeEach(async () => {
    routeSlug = null;

    await TestBed.configureTestingModule({
      imports: [
        PostEditorComponent,
        ReactiveFormsModule,
        CommonModule,
        MarkdownEditorStubComponent,
        FakeMatSelectDirective,
      ],
      providers: [
        { provide: PostService, useClass: PostServiceMock },
        { provide: MarkdownService, useClass: MarkdownServiceMock },
        { provide: CategoryService, useClass: CategoryServiceMock },
        { provide: Router, useClass: RouterMock },
        { provide: ActivatedRoute, useValue: ActivatedRouteStub },
        provideHttpClient(withInterceptors([authInterceptor]))
      ],
      schemas: [NO_ERRORS_SCHEMA],
```

```
  }).compileComponents();

  posts = TestBed.inject(PostService) as any;
  md = TestBed.inject(MarkdownService) as any;
  cats = TestBed.inject(CategoryService) as any;
  router = TestBed.inject(Router) as any;

  fixture = TestBed.createComponent(PostEditorComponent);
  component = fixture.componentInstance;
});

it('ініціалізація: підвантажує категорії', () => {
  fixture.detectChanges();
  expect(component).toBeTruthy();
  expect(cats.list).toHaveBeenCalled();
  expect(component.categories.length).toBe(2);
});

it('створення: save() викликає create() з коректним DTO і робить navigate', () => {
  fixture.detectChanges();

  component.form.patchValue({
    title: 'My post',
    status: 'DRAFT',
    categorySlugs: ['tech', 'news'],
  });
  component.markdown = 'Hello **MD**';

  component.save();

  expect(md.toHtml).toHaveBeenCalledWith('Hello **MD**');
  expect(posts.create).toHaveBeenCalledTimes(1);

  const dto = posts.create.calls.mostRecent().args[0];
  expect(dto).toEqual({
    title: 'My post',
    body: '<p>Hello **MD**</p>',
    status: 'DRAFT',
    categories: ['tech', 'news'],
  });
```

```
  expect(router.navigate).toHaveBeenCalledWith(['/posts', 'hello-post']);
});

it('валідація: title є обов’язковим', () => {
  fixture.detectChanges();
  component.form.patchValue({ title: '' });
  expect(component.form.invalid).toBeTrue();
});

it('cancel() → navigateByUrl("/")', () => {
  fixture.detectChanges();
  component.cancel();
  expect(router.navigateByUrl).toHaveBeenCalledWith('/');
});

describe('режим редагування', () => {
  beforeEach(() => {
    routeSlug = 'post-slug';
    fixture = TestBed.createComponent(PostEditorComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('ngOnInit підтягує пост, isEdit=true, переносить поля і очищає HTML у markdown', () => {
    expect(posts.getBySlug).toHaveBeenCalledWith('post-slug');
    expect(component.isEdit).toBeTrue();

    expect(component.form.value.title).toBe('Old title');
    expect(component.form.value.status).toBe('PUBLISHED');
    expect(component.form.value.categorySlugs).toEqual(['tech']);
    expect(component.markdown).toBe('Hi'); // '<p>Hi</p>' -> 'Hi'
  });

  it('save() у режимі редагування викликає update(id, dto) і робить navigate', () => {
    component.form.patchValue({
      title: 'Updated title',
      status: 'PUBLISHED',
      categorySlugs: ['news'],
    });
    component.markdown = 'Updated content';
```

```
    component.save();

    expect(md.toHtml).toHaveBeenCalledWith('Updated content');
    expect(posts.update).toHaveBeenCalledTimes(1);

    const [id, dto] = posts.update.calls.mostRecent().args;
    expect(id).toBe(42);
    expect(dto).toEqual({
      title: 'Updated title',
      body: '<p>Updated content</p>',
      status: 'PUBLISHED',
      categories: ['news'],
    });

    expect(router.navigate).toHaveBeenCalledWith(['/posts', 'updated-post']);
  });
 });
});
```