Міністерство освіти і науки України
Криворізький національний університет
Факультет інформаційних технологій
Кафедра автоматизації, комп'ютерних наук і технологій

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти – бакалавр
за освітньо-професійною програмою
«Комп'ютерні науки»
зі спеціальності
122 – Комп'ютерні науки

тема роботи:

«Розробка міні-блогу з модульною архітектурою на основі FastAPI»

Виконав студент гр. КН-21 _____ Чарієв Нєпєс.

Керівник _____ Харламенко В. Ю.

Нормоконтроль _____ Маринич І. А.

Завідувач кафедри _____ Рубан С. А.

Кривий Ріг – 2025

КРИВОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет: *інформаційних технологій*

Кафедра: *автоматизації, комп'ютерних наук і технологій*

Ступінь вищої освіти: *Бакалавр*

Спеціальність: *122 – Комп'ютерні науки*

ЗАТВЕРДЖУЮ

Зав. кафедрою: *к.т.н. Рубан С.А.*

_____

« *29* » *січня* 2025 р.

ЗАВДАННЯ

на кваліфікаційну роботу магістра

студентові групи КН-21 *Чарієву Нєпєсу*

1. Тема кваліфікаційної роботи: *«Розробка міні-блогу з модульною архітектурою на основі FastAPI»*

затверджено наказом по університету № *254с* від *13.05.2025 р.*

2. Термін здачі кваліфікаційної роботи: *06.06.2025 р.*

3. Склад кваліфікаційної роботи: *Пояснювальна записка обсягом 78с., додатки, презентація у Microsoft PowerPoint (13 слайдів) в електронному та друкованому вигляді*

4. Консультанти кваліфікаційної роботи:

Розділ 1-2 доц. Харламенко В. Ю.

Нормоконтроль доц. Маринич І. А.

5. Календарний план:

№	Етапи роботи	Термін виконання
1	*Вступ*	*01.03.25*
2	*Розділ 1*	*15.03.25*
3	*Розділ 2*	*21.04.25*
4	*Висновки*	*25.05.25*
5	*Оформлення кваліфікаційної роботи*	*30.05.25*
6	*Підготовка презентації та графічного матеріалу*	*01.06.25*
7	*Підготовка доповіді до захисту*	*05.06.25*

6. Дата видачі завдання: *28.01.2025р.*

Керівник _____ / Харламенко В. Ю./

7. Запевнення: *Я, Чарієв Нєпєс, запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.*

Із чинним Положенням про академічну доброчесність Криворізького національного університету ознайомлений.

Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі умисних порушень робота не допускається до захисту або оцінюється незадовільно.

Студент _____ / Чарієв Нєпєс/

АНОТАЦІЯ

Чарієв Нєпєс Розробка міні-блогу з модульною архітектурою на основі FastAPI : кваліфікаційна робота бакалавра : 122 – Комп’ютерні науки. Кривий Ріг. Криворізький національний університет, 2025. 63 с.

Метою даної роботи є створення міні-блогу з модульною архітектурою, який дозволяє користувачам реєструватися, автентифікуватися, створювати, редагувати та переглядати власні пости. Додаток має підтримувати зручну організацію контенту та забезпечувати захищений доступ до особистих даних користувачів.

У першому розділі здійснено комплексний аналіз сучасного розвитку блогів, а також визначено основні компоненти, необхідні для створення сучасного блогу. На основі проведеного аналізу було визначено основні інструменти, необхідні для реалізації міні-блогу. Такий підхід забезпечує цілісне бачення архітектури та функціональних можливостей веб-додатку для блогінгу, враховуючи як технічні, так і користувацькі вимоги.

У другому розділі було розроблено міні-блог із використанням модульної архітектури, що дало змогу чітко структурувати додаток та забезпечити гнучкість у його розширенні. Було визначено архітектурні рішення та функціональні можливості міні-блогу, включаючи управління контентом, реалізацію API для створення, отримання, видалення та зміни статусу блогів, а також підтримку тегів і фільтрів для зручного пошуку контенту. Було реалізовано ключові методи API, зокрема додавання нових блогів, отримання існуючих, видалення та управління статусом публікації. Також здійснено розробку інтерфейсу сторінок, реалізовано візуалізацію контенту та забезпечено коректне рендерування за допомогою FastAPI. На завершення було проведено тестування всіх функціональних блоків для перевірки їхньої коректності та стабільності роботи додатку.

АРХІТЕКТУРА ВЕБ-ДОДАТКУ, БАЗА ДАНИХ, БЛОГ, FASTAPI, ORM, PYTHON, API

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій та інтернет-комунікацій веб-додатки відіграють ключову роль у розвитку цифрових сервісів та поширенні інформації. Блоги та міні-блоги стали важливими інструментами для обміну думками, ведення особистих записів та професійної діяльності. З розвитком високошвидкісного інтернету та зростанням вимог користувачів до якості обслуговування виникає потреба у створенні продуктивних, безпечних і масштабованих веб-додатків.

Розробка веб-додатків потребує використання сучасних технологій, які забезпечують не лише швидкість виконання, а й легкість супроводу, модульність та підтримку новітніх стандартів безпеки. FastAPI - один із сучасних асинхронних веб-фреймворків для Python, який завдяки своїй простоті та високій продуктивності активно використовується для створення REST API. Поєднання FastAPI з ORM-бібліотекою SQLAlchemy (з асинхронною підтримкою через aiosqlite) дозволяє створювати ефективні та надійні додатки з роботою з базами даних. Використання Alembic для міграцій полегшує управління змінами в базі даних, а bcrypt та python-jose забезпечують безпечну аутентифікацію та управління користувачами. Markdown2 використовується для конвертації текстів у HTML, що спрощує написання контенту.

Актуальність роботи полягає у зростаючій потребі в легких і швидких веб-додатках із можливістю масштабування та розширення функціональності. Сучасні блоги та системи управління контентом часто потребують швидкого розгортання, інтеграції з різними сервісами та високої продуктивності при роботі з великою кількістю одночасних користувачів. Використання FastAPI як основного веб-фреймворку задовольняє ці вимоги завдяки своїй продуктивності, підтримці OpenAPI/Swagger для автоматичної документації та можливості інтеграції з асинхронними інструментами. Крім того, актуальним

є питання безпеки персональних даних та захисту авторизації, для чого використовуються сучасні засоби шифрування та управління токенами.

Метою даної роботи є створення міні-блогу з модульною архітектурою, який дозволяє користувачам реєструватися, автентифікуватися, створювати, редагувати та переглядати власні пости. Додаток має підтримувати зручну організацію контенту та забезпечувати захищений доступ до особистих даних користувачів. При розробці планується використати FastAPI як основний веб-фреймворк, SQLAlchemy у поєднанні з aiosqlite для взаємодії з базою даних, Alembic для управління міграціями, bcrypt для хешування паролів, python-jose для роботи з JWT-токенами та markdown2 для конвертації текстів у HTML.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. Проаналізувати існуючі інструменти та технології для розробки веб-додатків на основі FastAPI та обґрунтувати їх вибір для реалізації міні-блогу.

2. Розробити архітектуру додатку з урахуванням принципів модульності та масштабованості, яка забезпечить легке супроводження та розширення функціональності.

3. Реалізувати систему автентифікації та авторизації користувачів із використанням JWT-токенів, bcrypt та python-jose для забезпечення безпеки доступу.

4. Розробити модуль для створення та редагування публікацій із можливістю форматування тексту за допомогою Markdown та конвертації його у HTML.

5. Інтегрувати базу даних SQLite та ORM SQLAlchemy для збереження даних користувачів та постів, а також налаштувати Alembic для управління міграціями.

6. Розробити фронтенд додатку на базі HTML, CSS, JS та Jinja2 для рендерингу сторінок користувачів.

7. Забезпечити запуск додатку за допомогою Uvicorn та тестування його працездатності.

8. Провести тестування розробленої системи та оцінити її продуктивність і надійність.

Основною особливістю роботи є реалізація міні-блогу на основі модульної архітектури, яка дозволяє легко масштабувати та підтримувати проєкт. Також важливою особливістю є використання асинхронної обробки запитів, що підвищує продуктивність системи, особливо при обробці великої кількості одночасних запитів. Інтеграція Markdown-конвертера дозволяє користувачам зручно форматувати контент. Крім того, використання сучасних технологій для аутентифікації та управління користувачами робить додаток безпечним та зручним для кінцевих користувачів.

РОЗДІЛ 1
АНАЛІЗ РОЗВИТКУ БЛОГІВ ТА ВИБІР ІНСТРУТМЕНТІВ РОЗРОБКИ

1.1 Блог, його розвиток і особливості

Стрімкий розвиток інтернет-технологій зумовив виникнення нових форм комунікації, які нині є характерними ознаками інформаційного суспільства та важливими чинниками глобалізації. Серед таких форм особливу увагу привертають блоги, що створюють специфічний комунікативний та соціокультурний простір.

Блоги можуть мати індивідуальний характер, виступаючи вираженням особистісних поглядів та досвіду їх авторів, або набувати суспільного значення, інтегруючись у глобальну комунікаційну систему та справляючи вплив на культурний ландшафт сучасного світу. Завдяки своєму контенту блоги формують культурний контекст розвитку соціуму. Їхній розвиток характеризується динамічністю, оскільки вони приваблюють дедалі ширшу аудиторію та акцентують увагу на окремих аспектах повсякденного життя. Як сучасне явище, блоги є результатом трансформаційних процесів, що відбуваються як у технологічній, так і в соціокультурній сферах, водночас вони самі чинять вплив на ці процеси через діяльність віртуальних спільнот блогерів [10].

Інформаційні технології, які сприяли появі інтернету, призвели до формування нової форми взаємодії між людьми — інтернет-комунікації. У цьому контексті блоги стали невід’ємним елементом зазначеної взаємодії, створюючи унікальне інформаційно-комунікативне середовище, яке істотно впливає на процес глобалізації світового суспільства [23].

Існування блогів як засобу комунікації породжує нову модель взаємодії між авторами блогів та їхньою аудиторією. Спершу підписники виступають пасивними споживачами інформації, проте згодом перетворюються на

активних учасників цього процесу, висловлюючи власні оцінки й коментарі. Читачі сприймають публікації комплексно, беручи до уваги як авторський контент, так і коментарі інших користувачів. Таким чином, комунікаційний процес набуває творчого характеру, адже кожен учасник має можливість доповнювати інформаційний простір блогу та сприяти формуванню платформи для обговорень [3].

Такий тип міжособистісної взаємодії впливає на формування специфічної системи правил у віртуальному середовищі, урізноманітнюючи соціальні норми та визначаючи тематику, яка вважається актуальною для суспільства. У цьому контексті блогери, зважаючи на очікування своєї аудиторії, відіграють важливу роль у цьому процесі. Відсутність цензури робить блоги привабливими для користувачів, які шукають інформацію, здатну викликати емоційний відгук, а також впливати на їхні дії та ставлення [1].

Зростання популярності блогів, зокрема мікроблогів, призвело до значного збільшення кількості образливих коментарів і публікацій. Це явище спонукало одного з провідних ідеологів концепції Web 2.0 Тіма О’Рейлі запропонувати створення так званого «Кодексу блогера» [20].

У сучасних умовах дедалі більше компаній прагнуть створити власний блог і наповнити його високоякісним контентом. Провідні блогери сучасності користуються популярністю, порівнянною з популярністю зірок шоу-бізнесу, що підтверджується активним прагненням їхніх шанувальників до спілкування, зокрема у вигляді автографів та спільних фотографій. Блогери з великою аудиторією розглядаються як впливові медіа-актори, а їхні висловлювання набувають особливої ваги у публічному просторі; саме тому бренди активно залучають їх до співпраці в розробці нових продуктів. Таким чином, блоги перестають бути виключно онлайн-щоденниками, а завдяки розмаїттю платформ і форматів блогінгу трансформуються у потужний інструмент створення різноманітного контенту. Усе це свідчить про стійке зростання популярності блогінгу.

Блогосфера дедалі частіше виступає джерелом інформації, що користується довірою аудиторії, і ця тенденція, безперечно, посилюватиметься у майбутньому. Попри те що значна частина блогів має переважно щоденниковий характер і часто віддзеркалює повідомлення мас-медіа чи інших блогів, існують також ресурси, які пропонують аналітичний контент, підкріплений фактами або особистим досвідом авторів. Це свідчить про позитивну динаміку розвитку блогів як специфічної форми інтернет-комунікації [13].

Крім того, блоги виконують важливу комунікативну функцію, оскільки вони відображають культурний капітал як окремого автора, так і спільноти в цілому. Зокрема, кількість коментарів під дописами може слугувати індикатором культурного капіталу суспільства. Таким чином, блоги демонструють, як і чому виникають певні публікації та які наслідки вони мають.

1.2 Аналіз загальних компонентів блогу

Стандартна платформа для створення блогу зазвичай складається з кількох основних компонентів, кожен із яких виконує важливу функцію для ефективної організації та управління блогом. Розглянемо ці компоненти докладніше:

– *Content Management System (CMS, керування контентом):* є основою будь-якої блог-платформи та забезпечує створення, редагування, публікацію та організацію контенту. Вона надає користувачам зручний інтерфейс для роботи з текстами, зображеннями та іншими медіа-елементами без необхідності програмування. CMS також дозволяє структурувати контент за категоріями, тегами та іншими критеріями, що спрощує навігацію для читачів.

– *Дизайн та макет:* Цей компонент відповідає за візуальне оформлення блогу. Він включає шаблони сторінок, кольорову палітру, шрифти та інші елементи дизайну, які забезпечують привабливий і зручний для користувачів інтерфейс. Сучасні платформи дозволяють використовувати готові шаблони або створювати власні, адаптуючи їх до індивідуальних потреб та бренду блогу.

– *Управління користувачами:* Цей модуль дозволяє адміністратору блогу керувати ролями та правами доступу користувачів. Він забезпечує реєстрацію нових учасників, призначення ролей (автор, редактор, адміністратор) та модерацію контенту, створеного іншими користувачами. Це особливо важливо для колективних блогів або платформ із великою кількістю контрибуторів.

– *Медіа-контент:* Модуль медіа-контенту відповідає за зберігання, завантаження та управління мультимедійними файлами (зображення, відео, аудіо). Він дозволяє інтегрувати мультимедійні елементи у публікації, створювати галереї та забезпечує сумісність із різними форматами файлів. Такий компонент сприяє підвищенню візуальної привабливості блогу та його інтерактивності.

– *SEO-оптимізація:* Функціонал SEO-оптимізації допомагає підвищити видимість блогу у пошукових системах. Він включає інструменти для налаштування мета-тегів, описів, ключових слів, а також дозволяє керувати картами сайту та внутрішніми посиланнями. Це сприяє залученню нових користувачів і підвищує органічний трафік.

– *Коментарі та соціальна взаємодія:* Цей модуль забезпечує можливість обговорень між читачами та автором блогу через коментарі. Додатково можуть бути реалізовані функції соціальної взаємодії, такі як кнопки «Поділитися» у соцмережах, лайки, підписки тощо. Це створює спільноту навколо блогу, стимулює обговорення та підвищує залученість аудиторії.

– *Аналітика та статистика:* Компонент аналітики надає власникам блогів інструменти для відстеження відвідуваності, джерел трафіку, часу перебування на сторінці та інших важливих показників. Це дозволяє оцінювати ефективність контенту, виявляти популярні теми та адаптувати стратегію розвитку блогу відповідно до інтересів аудиторії.

Таким чином, наведені компоненти утворюють базову структуру платформи для створення блогу. Варто зазначити, що різні платформи можуть мати додаткові функції або відрізнятися гнучкістю налаштувань залежно від потреб користувачів та специфіки проекту.

1.3 Аналіз існуючих рішень створення блогу

У сучасному цифровому середовищі існує багато інструментів для створення та ведення блогів, кожен із яких має свої унікальні переваги та обмеження. Нижче представлено стислий огляд п’яти популярних платформ: Ghost, WordPress, Blogger, Wix та Squarespace - із виділенням їхніх сильних і слабких сторін, щоб допомогти обрати найкращий варіант для реалізації власних проектів.

1.3.1 Платформа Ghost

Найкраще підходить для професійних авторів, які цінують швидкість і простоту публікації текстового контенту та потребують монетизації через підписки.

До переваг платформи можна віднести:

- Мінімалістичний дизайн, зосереджений на контенті.
- Вбудовані інструменти монетизації (підписки, платний доступ).
- Швидкість роботи та чистий код (SEO-friendly).
- Ідеально підходить для професійних авторів та журналістів.

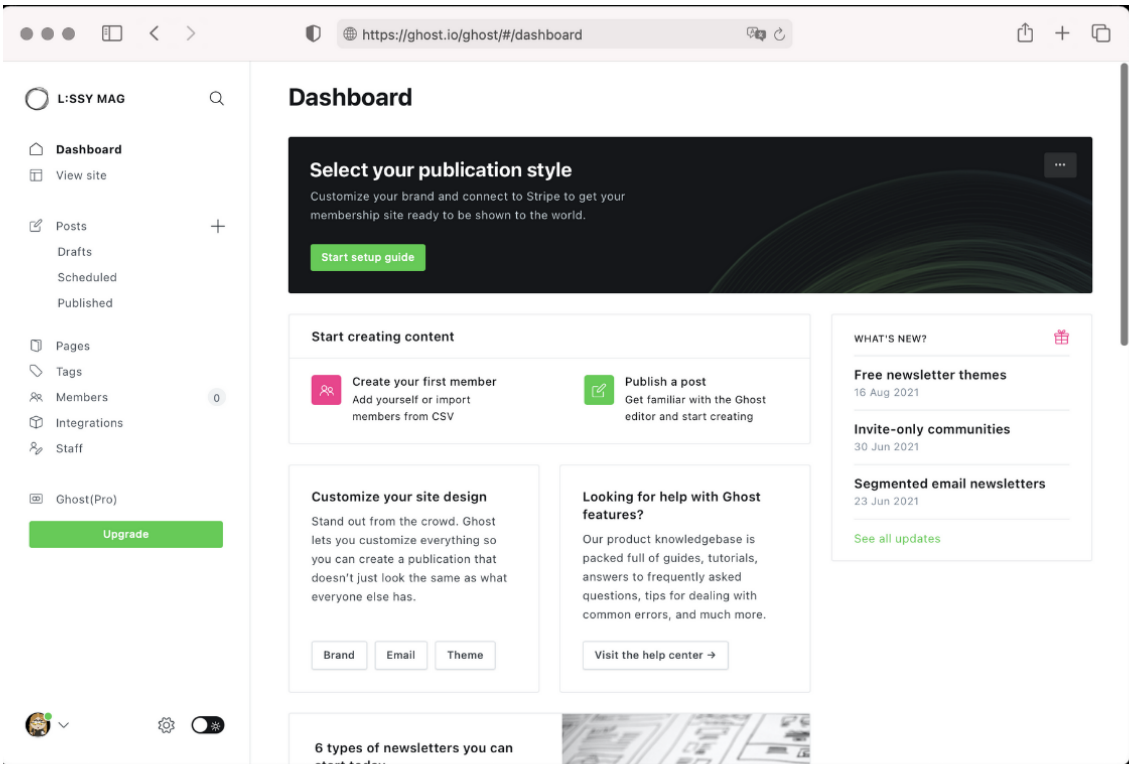


Рисунок 1.1 – Головний інтерфейс платформи Ghost

До недоліків платформи можна віднести:

– Потребує технічних знань для налаштування (якщо self-hosted).
– Обмежений вибір тем і плагінів у порівнянні з WordPress.
– Не підходить для складних сайтів із великою кількістю інтеграцій.

1.3.2 Платформа WordPress

Найбільш гнучкий варіант з величезною екосистемою плагінів і тем, підходить як для початківців, так і для великих проєктів.

До переваг платформи можна віднести:

– Найбільший вибір тем і плагінів.
– Підтримка будь-якого типу сайтів (від блогів до магазинів).
– Велика спільнота та потужна підтримка.
– Легко масштабувати і налаштовувати.

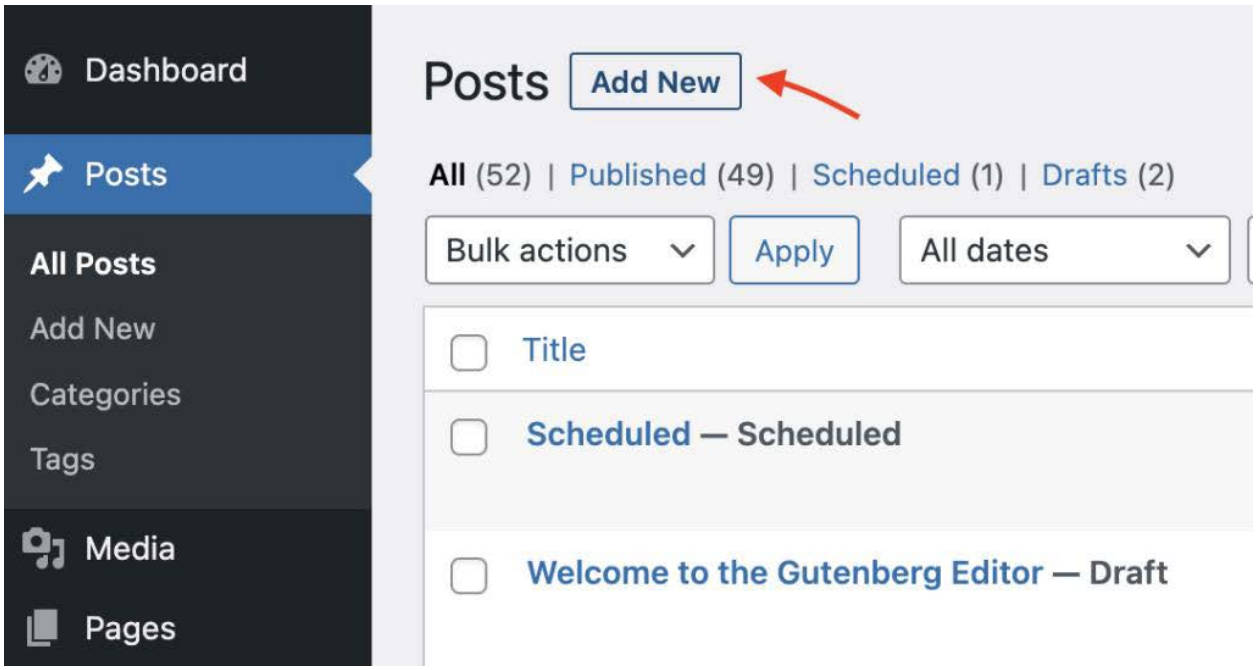


Рисунок 1.2 – Вигляд редактору WordPress

До недоліків платформи можна віднести:

– Може бути перевантаженим для простих блогів.
– Потребує регулярного обслуговування (оновлення, безпека).
– Деякі плагіни або теми можуть конфліктувати між собою.

1.3.3 Платформа Blogger

Найпростіший варіант для новачків без технічних знань, але з обмеженою функціональністю.

До переваг платформи можна віднести:

– Безкоштовний і простий у використанні.
– Інтеграція з Google-акаунтом.
– Надійний хостинг від Google.

До недоліків платформи можна віднести:

– Дуже обмежені можливості кастомізації.
– Немає сучасних SEO-інструментів.
– Виглядає дещо застаріло у порівнянні з іншими платформами.

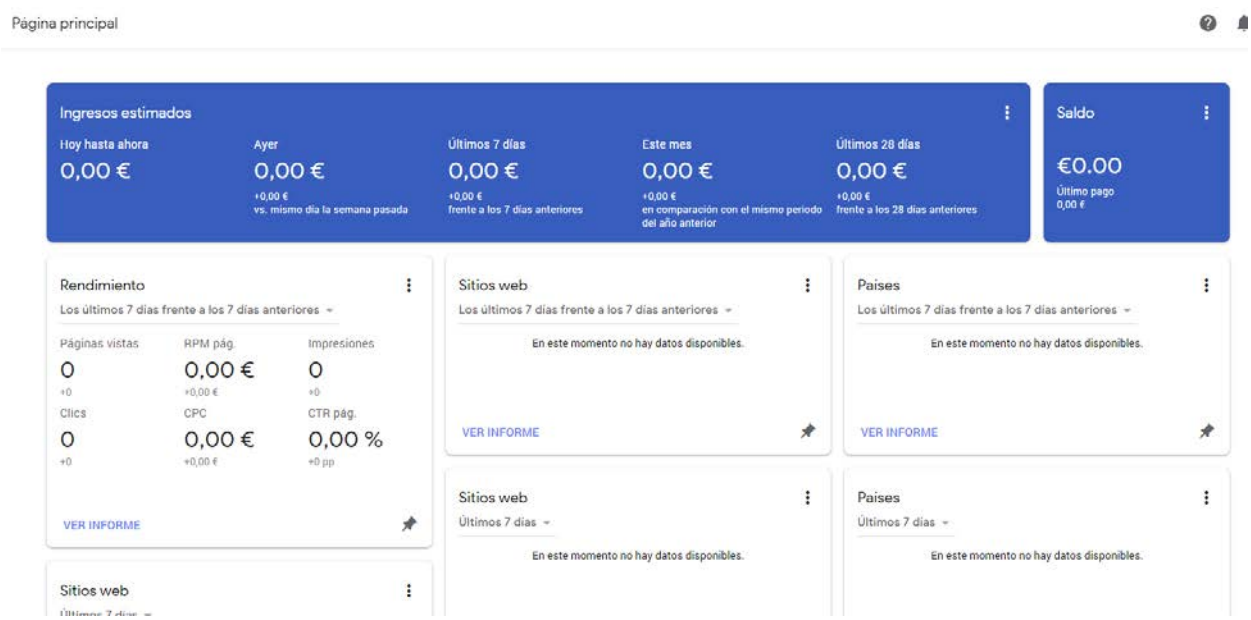


Рисунок 1.3 – Інтерфейс Blogger

1.3.4 Платформа Wix

Зручний drag-and-drop редактор для користувачів, яким потрібен блог як частина бізнес-сайту або портфоліо

До переваг платформи можна віднести:

– Drag-and-drop редактор — легко створювати сторінки без кодування.

– Великий вибір шаблонів та додатків.

– Інтеграція мультимедійного контенту.

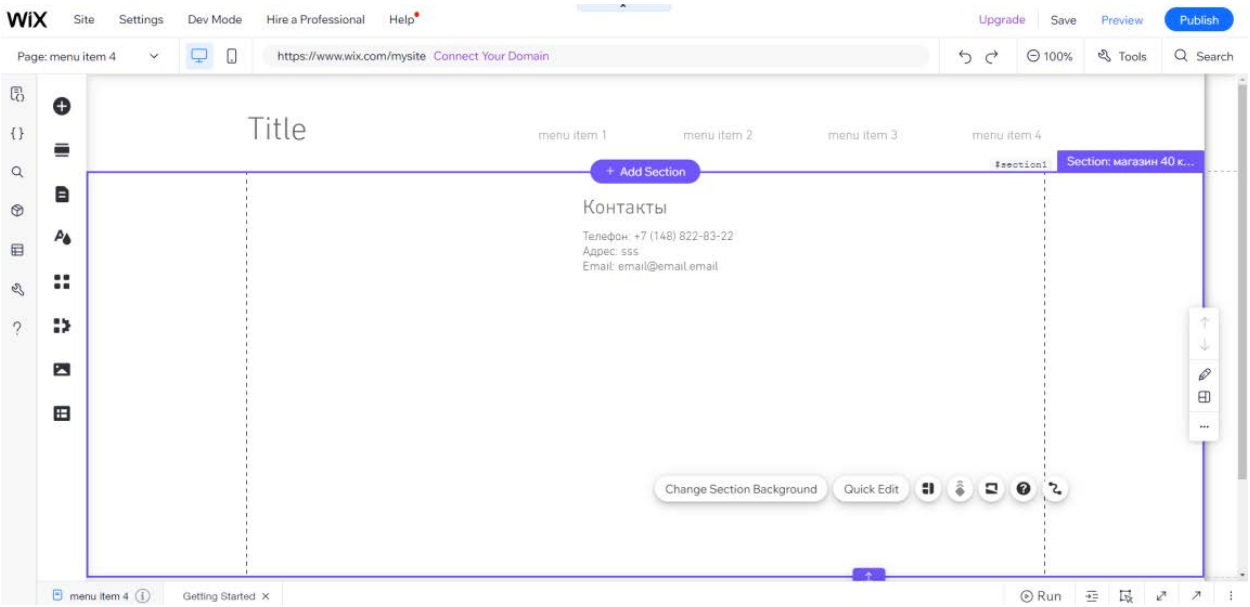


Рисунок 1.4 – Drag-and-Drop інтерфейс Wix

До недоліків платформи можна віднести:

– Деякі SEO-функції обмежені.
– Міграція на іншу платформу досить складна.
– Може бути дорожчим у довгостроковій перспективі.

1.3.5 Платформа Squarespace

Оптимальний для тих, хто хоче отримати професійний дизайн без зайвих налаштувань та складних конфігурацій

До переваг платформи можна віднести:

– Сучасний і професійний дизайн.
– Простий у використанні інтерфейс.
– Добра підтримка мультимедіа та e-commerce.

До недоліків платформи можна віднести:

– Обмежена гнучкість для складних проектів.
– Дорожчий порівняно з іншими платформами.
– Менше можливостей для SEO та інтеграцій, ніж у WordPress.

Для більш зручного представлення аналізу наведених платформ представимо його у вигляді таблиці 1.1.

Проаналізувавши існуючі платформи для ведення блогів, можна дійти висновку про необхідність вибору ключового функціоналу та додавання нового, який наразі відсутній у представлених рішеннях.

Отже, основний функціонал має включати:

– *Керування контентом.* Веб-додаток має передбачати наявність зручної та ефективної системи керування контентом (CMS), що дає змогу створювати, редагувати та управляти статтями, медіа-ресурсами та іншими елементами блогу.

– *Дизайн та налаштування*. Важливою є можливість широкого налаштування дизайну, включно з вибором шаблонів, кольорів, шрифтів, розміщенням елементів та іншими візуальними параметрами.

Таблиця 1.1 - Порівняльний аналіз популярних платформ для створення блогів

Параметр	Ghost	WordPress	Blogger	Wix	Squarespace
Модель використання	Open-source або хмарна підписка	Open-source або SaaS	Безкоштовний сервіс від Google	SaaS	SaaS
Простота використання	Орієнтований на професійних блогерів та розробників, потребує певних технічних знань	Універсальний: легкий старт для новачків, але підтримує складні функції	Дуже простий у використанні, підходить для початківців	Інтуїтивно зрозумілий drag-and-drop редактор	Простий в освоєнні drag-and-drop редактор, сучасний інтерфейс
Гнучкість дизайну	Мінімалістичний, переважно текстовий фокус; налаштування потребує знань у верстці	Величезний вибір тем, можливість кастомізації за допомогою плагінів та CSS	Обмежені шаблони, базовий рівень кастомізації	Багато шаблонів, гнучкі можливості кастомізації	Багато шаблонів, сучасні дизайн-інструменти, але менша гнучкість порівняно з WordPress
SEO-можливості	Вбудована SEO-оптимізація, але потребує додаткових налаштувань	Величезний вибір плагінів для SEO	Базова SEO, обмежені можливості	Вбудовані SEO-налаштування, але обмежені WordPress	Має базову SEO-підтримку
Можливості монетизації	Орієнтований на платні підписки та членства	Гнучкі можливості монетизації через плагіни	Обмежена монетизація через рекламу Google	Доступні вбудовані інструменти для продажу товарів	Орієнтований на малий бізнес: інтеграція електронної комерції
Підтримка мультимедіа	Підтримує зображення та відео, але немає широкого мультиме-дійного редактора	Широка підтримка мультимедіа, з можливістю інтеграції через плагіни	Обмежена підтримка мультимедіа (базові фото відео)	Розвинена мультимедійна підтримка, просте завантаження	Сильні мультимедійні можливості, інтеграція з Getty Images та Adobe
Ціна	Open-source - безкоштовно (потрібен хостинг); SaaS - від $9/міс	Open-source - безкоштовно (оплата хостингу та плагінів); SaaS - безкоштовно або від $4/міс	Безкоштовно	Безкоштовно (з рекламою) або від $16/міс	Від $16/міс

Крім основних можливостей, варто передбачити додатковий функціонал, якого бракує у наявних платформах. Зокрема, під час написання статті у популярних додатках для блогінгу кількість типів блоків є обмеженою: зазвичай це абзаци, заголовки, HTML-вставки, посилання, кнопки тощо. Натомість жодна з платформ наразі не пропонує можливість створювати власні кастомні блоки для розширення можливостей оформлення контенту.

1.4 Аналіз та вибір інструментів для реалізації міні-блогу

1.4.1 Вибір веб-фреймворку

Для розробки веб-застосунків на Python найпопулярнішими є такі фреймворки:

- Django - повноцінний фреймворк з ORM, системою аутентифікації, адміністраторською панеллю. Підходить для великих проектів, але менш гнучкий для модульної архітектури.
- Flask - легкий фреймворк, популярний для швидкого прототипування. Синхронний за замовчуванням, складніший для масштабування під високі навантаження.
- FastAPI - сучасний асинхронний фреймворк, що дозволяє швидко створювати RESTful API та автоматично генерує OpenAPI-документацію. Підтримує асинхронність через стандарт ASGI, легко інтегрується з іншими бібліотеками.

Таблиця 1.2 - Порівняння та вибір веб-фреймворку

Фреймворк	Переваги	Недоліки
FastAPI	Асинхронність, висока продуктивність, автоматична генерація OpenAPI.	Молодий, менше готових рішень.
Flask	Простий у використанні, велика спільнота.	Синхронний, менш масштабований.

Django	«Все в одному» (ORM, аутентифікація, міграції).	Монолітність, складно зробити легкий API.

Обираємо FastAPI для реалізації сучасного асинхронного веб-додатку, що легко інтегрується з іншими бібліотеками та відповідає вимогам до продуктивності та модульності.

1.4.2 Вибір ORM

- SQLAlchemy - найпопулярніший ORM у Python, підтримує як синхронний, так і асинхронний режим (через aiosqlite або інші драйвери). Гнучкий та розширюваний.
- Tortoise ORM - легкий ORM, орієнтований на асинхронну роботу. Підходить для простих застосунків, але менш гнучкий для складних моделей.
- Peewee - легкий ORM із простим API, але обмеженою підтримкою асинхронності.

Таблиця 1.3 - Порівняння та вибір ORM

ORM	Переваги	Недоліки
SQLAlchemy	Гнучкий, потужний, підтримує асинхронність.	Складніше налаштування.
Tortoise ORM	Простий у використанні, асинхронний з коробки.	Менша спільнота, обмежений функціонал.
Peewee	Легкий синтаксис.	Немає повноцінної асинхронності.

Вибираємо SQLAlchemy у поєднанні з aiosqlite, що забезпечує необхідну гнучкість, масштабованість та асинхронність.

1.4.3 Вибір бази даних

Для невеликого проєкту або MVP часто використовують SQLite як вбудовану базу даних . Для більш масштабних проектів зазвичай застосовують PostgreSQL або MySQL/MariaDB.

Таблиця 1.4 - Порівняння та вибір бази даних

СУБД	Переваги	Недоліки
SQLite	Простота налаштування, не потребує сервера.	Обмеження при високих навантаженнях.
PostgreSQL	Потужна СУБД, розширюваність.	Потребує окремого сервера та адміністрування.
MySQL/MariaDB	Швидка, популярна.	Потребує окремого сервера.

Вибираємо SQLite, як просту та вбудовану базу даних, що дозволяє швидко розгорнути проект, а також легко перейти на іншу СУБД у майбутньому.

1.4.4 Вибір засобу міграції

- Alembic - офіційний інструмент для SQLAlchemy, який дозволяє відслідковувати та застосовувати зміни у структурі бази даних.
- Django Migrations - тісно інтегрований із Django ORM, не підходить для FastAPI.

Таблиця 1.5 - Порівняння та вибір засобу міграції

Інструмент	Переваги	Недоліки
Alembic	Інтеграція з SQLAlchemy, можливість кастомізації.	Потребує налаштування для асинхронності.
Django Migrations	Автоматизація в Django.	Не підходить для FastAPI.

Вибираємо Alembic, оскільки він легко інтегрується з SQLAlchemy.

1.4.5 Вибір інструменту аутентифікації та авторизації

- bcrypt - популярний алгоритм хешування паролів.
- python-jose - бібліотека для створення та перевірки JWT-токенів.

Таблиця 1.6 - Порівняння та вибір інструменту аутентифікації та авторизації

Інструмент	Переваги	Недоліки
bcrypt	Високий рівень безпеки.	Повільніше для слабких пристроїв.
python-jose	Простота інтеграції.	Потребує налаштування для refresh-токенів.

Вибираємо bcrypt для паролів, python-jose для JWT.

1.4.6 Вибір рендеринг HTML

– Jinja2 - шаблонізатор, що легко інтегрується з FastAPI
– React/Vue - для SPA-проєктів, але потребують складної збірки.

Вибираємо Jinja2 обрано для легкого рендерингу HTML-шаблонів.

1.4.7 Запуск сервера

– Uvicorn - асинхронний сервер для ASGI-додатків.
– Gunicorn - традиційно використовується для WSGI, не підтримує асинхронність «з коробки».

Вибираємо Uvicorn, як нативний ASGI-сервер, обрано для використання з FastAPI.

1.4.8 Конвертація Markdown

– markdown2 - легка бібліотека для перетворення Markdown у HTML.
– Mistune - більш гнучка, але складніша у використанні.

Вибираємо markdown2 - просту та ефективну для реалізації блогу.

Проведений аналіз підтверджує доцільність використання обраних інструментів:

– FastAPI для асинхронного веб-фреймворку;

- SQLAlchemy з aiosqlite для роботи з базою даних;
- SQLite як легка та вбудована СУБД;
- Alembic для управління міграціями;
- bcrypt та python-jose для аутентифікації;
- Jinja2 для рендерингу HTML;
- Uvicorn як ASGI-сервер;
- markdown2 для конвертації Markdown.

Висновки до розділу:

У цьому розділі було здійснено комплексний аналіз сучасного розвитку блогів як комунікаційного та соціокультурного явища, а також визначено основні компоненти, необхідні для створення сучасного блогу. Розглянуто найпопулярніші платформи для ведення блогів: Ghost, WordPress, Blogger, Wix та Squarespace - з акцентом на їхні сильні та слабкі сторони, що дало змогу виявити відмінності між ними та оцінити їхню відповідність завданням розробки власного міні-блогу.

На основі проведеного аналізу було визначено основні інструменти, необхідні для реалізації міні-блогу, зокрема веб-фреймворк, ORM, базу даних, засіб міграції, систему аутентифікації та авторизації, механізм рендерингу HTML, а також засіб конвертації Markdown. Такий підхід забезпечує цілісне бачення архітектури та функціональних можливостей веб-додатку для блогінгу, враховуючи як технічні, так і користувацькі вимоги.

Отже, підсумовуючи результати розділу, можна зробити висновок, що вибір технологій та платформ має ґрунтуватися на глибокому аналізі сучасних інструментів та потреб користувачів, що дозволить створити інноваційний та конкурентоспроможний продукт.

РОЗДІЛ 2
РОЗРОБКА МІНІ-БЛОГУ З МОДУЛЬНОЮ АРХІТЕКТУРОЮ

2.1 Структура рішення розробки та можливості

Користувач має просто отримувати текст у форматі Markdown, який відображатиметься як оформлені сторінки (з посиланнями). Можна було б С використовувати для цієї задачі сервіс Telegraf , однак у Telegraf є багато обмежень: наприклад, не можна публікувати тексти у форматі Markdown - підтримується лише HTML, і то з обмеженим функціоналом та обмеження на довжину постів. Тому було вирішено реалізувати міні-блог, розроблений на базі FastAPI, з впровадженням системи реєстрації, аутентифікації та авторизації користувачів. Проект включає модульну архітектуру, гнучке логування через loguru та взаємодію з базою даних за допомогою SQLAlchemy. Аламбік використовується для управління міграціями.

Ключові особливості

Управління користувачами:

– Реєстрація користувача та вхід до системи.

– Безпечне хешування пароля за допомогою bcrypt.

– Авторизація за допомогою токенів JWT через бібліотеку python-jose.

– Захист кінцевих точок з ролями та залежностями.

Функціонал блогу:

– Додавання записів:

– Створіть новий блог із прикріпленим заголовком, текстом і тегами.

– Можливість зберегти запис у вигляді чернетки.

– Видалення записів:

– Автор може видаляти свої повідомлення.

- Переглянути записи:
  - Отримайте список усіх опублікованих блогів.
  - Перегляд певного запису.
  - Фільтруйте публікації за тегами.
- Дерево тегів:
  - Система тегів реалізована через зв'язок «багато до багатьох».
  - Автоматичне створення дерева тегів.

Логування:

- Використовуйте loguru для зручного керування журналами.

Модульна архітектура:

- Проект розбитий на модулі для спрощення підтримки та масштабування.

Стек технологій

- Веб-фреймворк: FastAPI
- ORM: SQLAlchemy з асинхронною підтримкою через aiosqlite
- Database:SQLite (може бути замінений будь-якою SQL DBMS)
- Міграції:Аламбік
- Аутентифікація: bcrypt для хешування паролів, python-jose для роботи з JWT токенами
- Фронтенд: HTML + CSS + JS (з використанням Jinja2)

Структура проекту

```
├── app/
│   ├── auth/              # Модуль авторизації й автентифікації
│   ├── api/               # Модуль API міни-блогу
│   ├── pages/             # Модуль фронтенд частини
│   ├── dao/               # Загальні DAO для додатку
│   ├── migration/         # Міграції бази даних
│   ├── static/            # Статичні файли додатку
│   ├── config.py          # Конфігурація додатку
```

```
│   ├── database.py          # Підключення до бази даних
│   ├── exceptions.py        # Виключення для обробки помилок
│   └── main.py              # Головний файл для запуску додатку
├── data/                    # Файли бази даних
├── .env                     # Конфігурація оточення
├── .gitignore               # Файли, ігноруємі у Git
├── alembic.ini              # Конфігурація Alembic
├── README.md                # Документація проекту
└── requirements.txt         # Злежності проекту
```

Представити взаємозв’язки модулів можна у вигляді діаграма архітектури наступним чином (рис. 2.1)

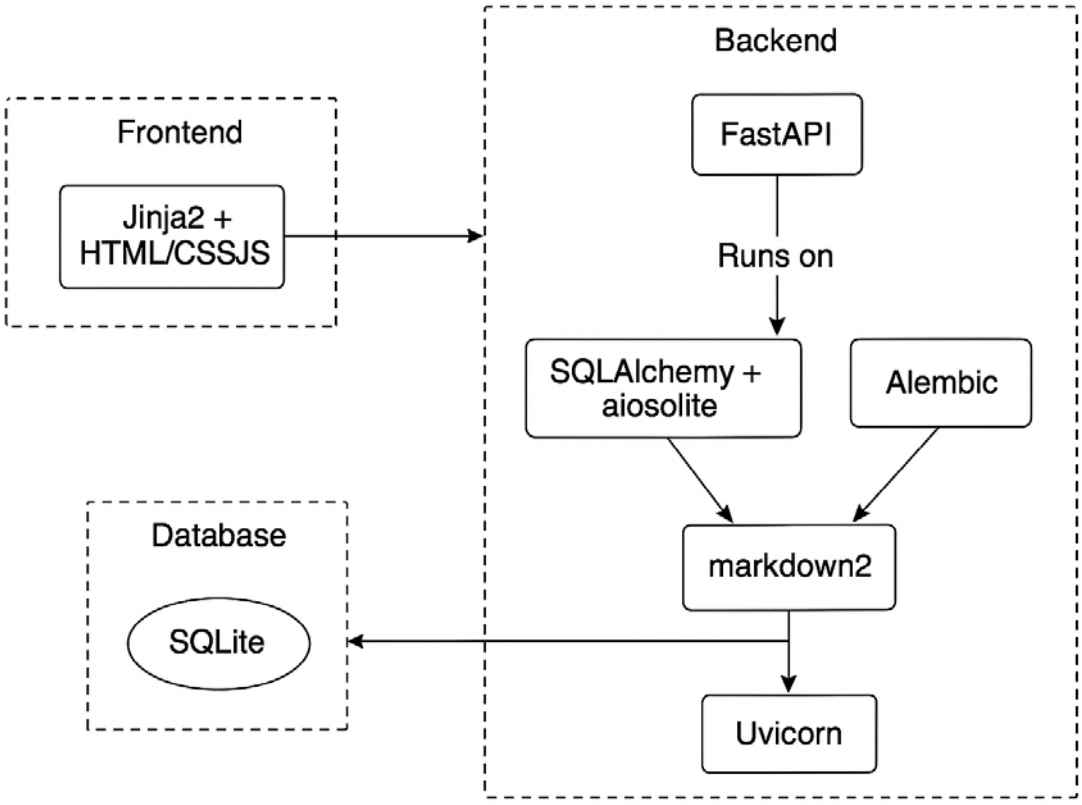


Рисунок 2.1 – Діаграма архітектури

2.2 Функціонал міні-блогу

Давайте розберемося, який функціонал буде у нашого міні-блогу. Незважаючи на те, що ми створимо єдиний застосунок, який включатиме як API-частину (логи), так і візуальну складову, проєкт буде розділений на декілька частин, прийнятих називати мікросервісами:

База даних і взаємодія з нею

Ми працюватимемо з SQLite, щоб уникнути витрат часу на налаштування PostgreSQL. У цьому нам допоможе асинхронний ORM SQLAlchemy 2, який дозволить описувати таблиці (моделі) і, що найголовніше, зручно взаємодіяти з базою даних через вбудовані механізми.

Щоб наші моделі таблиць перетворилися на повноцінні таблиці SQLite, ми будемо використовувати Alembic.

На рисунку 2.2 наведено ER (Entity-Relationship) діаграму, яка показує структуру таблиць у базі даних.

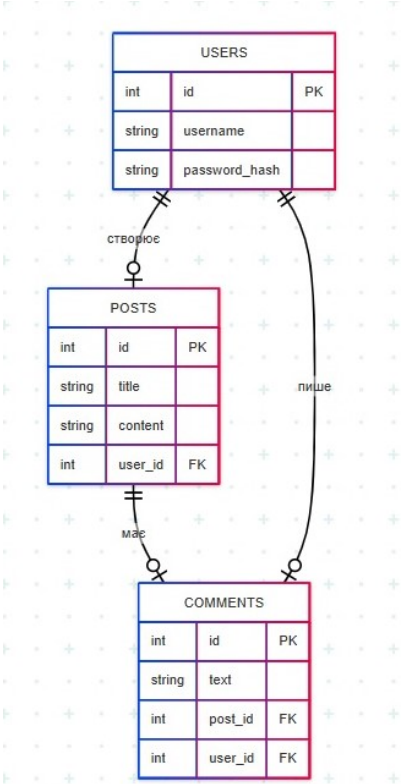


Рисунок 2.2 - Entity-Relationship діаграма

Реєстрація, авторизація та аутентифікація

У нашому блозі публікація статті буде неможливою без входу в систему. Отже, без реєстрації та авторизації нам не обійтися.

Для реалізації цієї логіки ми використовуватимемо систему JWT-токенів.

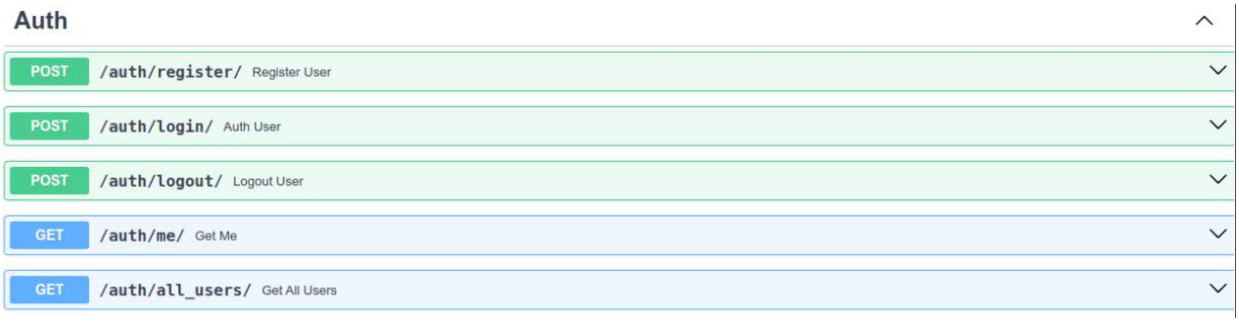


Рисунок 2.3 - Система JWT-токенів.

У межах цього блоку ми також детально розглянемо:

- Залежності (Dependencies) у FastAPI
- Ролі користувачів, їхнє керування та обмеження доступу

На рисунку 2.4 наведено діаграму взаємодії (Sequence Diagram), яка показує взаємозв'язки, наприклад як, наприклад, користувач авторизується, створює або переглядає пости.

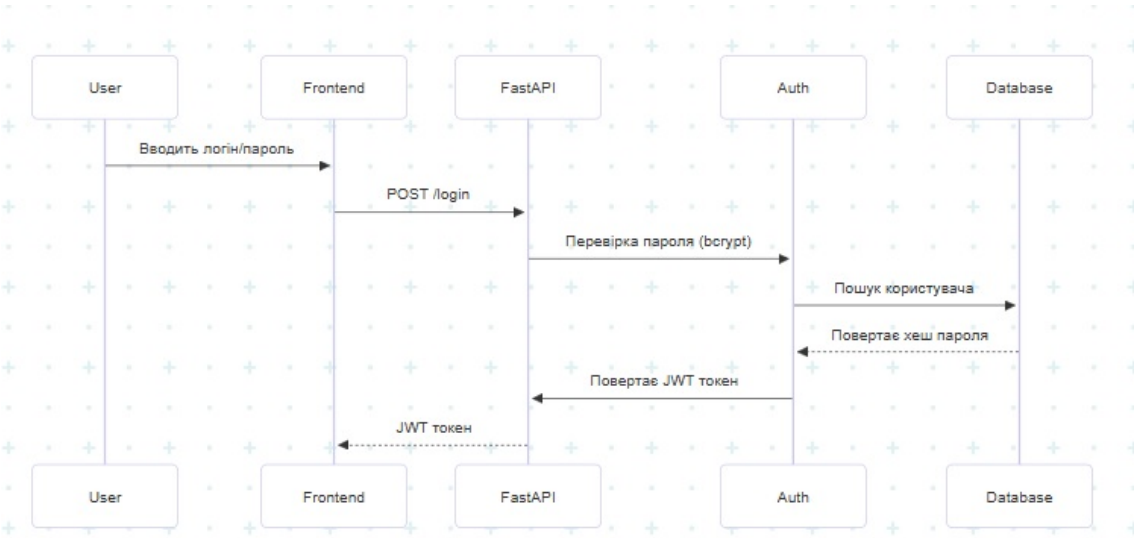


Рисунок 2.4 - Sequence Diagram

У результаті ми створимо набір API-методів, які забезпечать реєстрацію, авторизацію та аутентифікацію користувачів.

API-логіка

У цьому блоці ми опишемо такі методи:

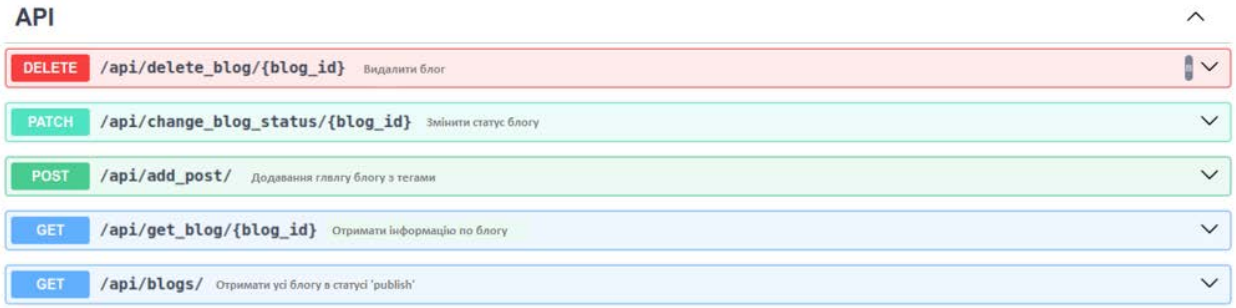


Рисунок 2.5 – Розглянуті API

- Видалення блогу
- Зміна статусу блогу (опубліковано / чернетка)
- Додавання нового блогу з тегами
- Отримання інформації про блог (читання блогу)
- Отримання всіх блогів зі статусом «Опубліковано»

Фронтенд

Для фронтенду ми реалізуємо два ендпойнти FastAPI:

- Рендеринг сторінки з блогом / сторінки «блог не знайдено»
- Рендеринг сторінки зі списком блогів (тут ми додатково ускладнимо реалізацію фільтрацією за автором і тегами)

2.3 Підготовка таблиць для API блогу

Базовий клас, на основі якого створюються всі моделі таблиць, має такий вигляд (рис.2.6):

```python
class Base(AsyncAttrs, DeclarativeBase):
    __abstract__ = True

    id: Mapped[int] = mapped_column(Integer, primary_key=True, autoincrement=True)
    created_at: Mapped[datetime] = mapped_column(TIMESTAMP, server_default=func.now())
    updated_at: Mapped[datetime] = mapped_column(
        TIMESTAMP,
        server_default=func.now(),
        onupdate=func.now()
    )

    @declared_attr
    def __tablename__(cls) -> str:
        return cls.__name__.lower() + 's'

    def to_dict(self) -> Dict[str, Any]:
        return {c.name: getattr(self, c.name) for c in self.__table__.columns}

    def __repr__(self) -> str:
        """ представлення об'єкта для відладки """
        return f"<{self.__class__.__name__}(id={self.id}, created_at={self.created_at}, updated_at={self.updated_at})>"
```

Рисунок 2.6 – Лістинг коду базового класу

Колонки id, created_at та updated_at будуть генеруватися автоматично, так само як і назви таблиць. Це зроблено для того, щоб не відволікати вас у блоці опису таблиць для нашого API блогу.

Створимо папку app/api і в ній файл models.py. Тепер можна переходити до опису моделей.

```python
class Blog(Base):
    # Заголовок статті
    title: Mapped[str_uniq]

    # Автор (зовнішній ключ на таблицю Users
    author: Mapped[int] = mapped_column(ForeignKey("users.id"), nullable=False)

    # Зміст статті у форматі Markdown
    content: Mapped[str] = mapped_column(Text)

    short_description: Mapped[str] = mapped_column(Text)

    # Статус статті
    status: Mapped[str] = mapped_column(default="published", server_default='published')
```

Рисунок 2.7 – Лістинг коду опису таблиці с блогами

На особливу увагу заслуговує поле author. Через ForeignKey ми пов’язуємо таблицю блогів із таблицею авторів. Наступна проста таблиця - для зберігання тегів. У ній ми будемо зберігати кожен тег як окремий запис:

```
class Tag(Base):
    # Назва тегу
    name: Mapped[str] = mapped_column(String(50), unique=True)
```

Тепер ускладнимо нашу модель. Логічно, що нам не завадить можливість отримувати всі статті автора окремим запитом, а також знати, хто є автором тієї чи іншої статті. Для цього ми можемо використати систему зв’язків SQLAlchemy - relationship.

Додамо до моделі Blog такий рядок:

```
# Посилання на об'кт користувача
user: Mapped["User"] = relationship("User", back_populates="blogs")
```

Тут ми вказали, що в одного блогу може бути тільки один автор, що цілком логічно. Тепер налаштуємо зворотний зв’язок, вказавши, що один користувач може бути автором одразу кількох блогів:

```
blogs: Mapped[list["Blog"]] = relationship(
    back_populates="user"    # Повинно співпадати з ім'ям в моделі Blogs
)
```

Тепер ми замкнули зв’язок. Завдяки цьому ми зможемо з легкістю отримувати як інформацію про автора блогу, так і всі статті конкретного автора. Тепер переходимо до зв’язку «багато до багатьох», який ми раніше ще не розглядали. Для реалізації зв’язку блогів і тегів використовуємо проміжну таблицю (рис.2.8):

```
class BlogTag(Base):
    # Зовнішні ключі для зв'язку блогів і тегів
    blog_id: Mapped[int] = mapped_column(ForeignKey("blogs.id", ondelete="CASCADE"), nullable=False)
    tag_id: Mapped[int] = mapped_column(ForeignKey("tags.id", ondelete="CASCADE"), nullable=False)

    # Унікальне обмеження для запобігання дублювання
    __table_args__ = (
        UniqueConstraint('blog_id', 'tag_id', name='uq_blog_tag'),
    )
```

Рисунок 2.8 – Лістнг коду створення проміжної таблиці

Розширимо моделі Blog і Tag для підтримки зв’язку «багато до багатьох» (рис.2.9).

Розберемо зв’язок «багато до багатьох» детальніше на прикладі блогів і тегів.

Уявіть, що у вас є блог про програмування. Одна стаття може мати кілька тегів - наприклад, #python, #backend, #sqlalchemy, а один тег може бути прив’язаний до багатьох різних статей. Наприклад, тег #python може бути у статтях про введення в мову, про фреймворки та про бази даних.

```
class Blog(Base):
    # ..... попередні поля ......

     # зв'язок з тегами через проміжну таблицю
    tags: Mapped[list["Tag"]] = relationship(
        secondary="blog_tags", # вказуємо проміжну таблицю
        back_populates="blogs"
    )

class Tag(Base):
    # ..... попередні поля ......

    # зворотній зв'язок з блогами
    blogs: Mapped[list["Blog"]] = relationship(
        secondary="blog_tags",
        back_populates="tags"
    )
```

Рисунок 2.9 – Лістинг коду розширення моделі Blog і Tag

Технічно зробити прямий зв’язок між блогами і тегами неможливо - потрібна проміжна таблиця blog_tags. Вона працює як своєрідний міст між блогами і тегами. Вона вирішує одразу кілька задач:

- Дозволяє створювати багато зв’язків між блогами і тегами
- Запобігає дублюванню записів
- Забезпечує цілісність даних

У нашому прикладі проміжна таблиця BlogTag містить лише два зовнішніх ключі: blog_id і tag_id. Це означає, що кожен запис у цій таблиці - це унікальний зв’язок між конкретним блогом і конкретним тегом.

Унікальне обмеження UniqueConstraint('blog_id', 'tag_id') гарантує, що ми не зможемо випадково додати один і той самий тег до одного блогу двічі.

Простіше кажучи, проміжна таблиця схожа на таблицю лайків в Instagram, яка зв’язує користувачів і їхні вподобання. Тільки в нашому випадку - це зв’язок блогів і тегів. Нам не доведеться напряму працювати з цією проміжною таблицею. Ми можете просто додавати теги до блогу через .tags.append() або отримувати всі теги блогу через .tags, і ORM сама подбає про коректне заповнення зв’язної таблиці.

2.3.1 Виконання міграції

На цьому етапі ми лише описали наші таблиці, але вони ще не створені в базі даних. Щоб ці таблиці з’явилися, необхідно виконати міграцію за допомогою Alembic.

Спочатку додамо нові імпорти у файл app/migration/env.py:

```
from app.api.models import Tag, Blog, BlogTag
```

Тепер, щоб створити файл з інструкціями для міграції, відкриємо термінал і у корені проєкту виконаємо команду:

```
alembic revision --autogenerate -m "add new tables"
```

Для застосування змін виконайте команду:

```
alembic upgrade head
```

Все пройшло успішно (рис. 2.10), ми бачимо нові таблиці у своїй базі даних.

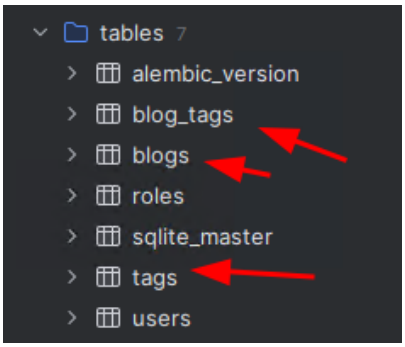


Рисунок 2.10 – Результат виконання міграції

Так як ми використовуємо підхід BaseDAO. Це означає, що існує певний клас, у якому зібрані універсальні методи для роботи з базою даних. Для цього скористаємося дочірніми класами, а точніше - розширимо їх.

У папці app/api створюємо файл dao.py і спочатку просто прив'яжемо кожну нову таблицю до базового класу BaseDAO (рис.2.11):

```python
from typing import Optional
from loguru import logger
from sqlalchemy import select, func
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import joinedload, selectinload
from app.api.schemas import BlogFullResponse
from app.dao.base import BaseDAO
from app.api.models import BlogTag, Tag, Blog


class TagDAO(BaseDAO):
    model = Tag


class BlogDAO(BaseDAO):
    model = Blog


class BlogTagDAO(BaseDAO):
    model = BlogTag
```

Рисунок 2.11 – Лістинг коду прив'язки нових таблиць до базового класу

Тепер у нас є можливість викликати універсальні методи для роботи з конкретною таблицею. Наприклад:

```python
blogs = await BlogDAO.find_all(session=session, filters=None)
```

2.3.2 Логіка додавання нових блогів та тегів

Почнемо з реалізації логіки додавання нового блогу. Нам достатньо просто використати метод додавання з BaseDAO.

```python
@classmethod
async def add_tags(cls, session: AsyncSession, tag_names: list[str]) -> list[int]:
    """
    Метод додавання тегів до бази даних.
    Приймає список рядків (тегів), перевіряє існують вони у базі даних,
    додає нові і повертає список ID тегів


    :param session: Сесія бази даних.
    :param tag_names: Список тегів у нижньому регістрі.
    :return: Список ID тегів.
    """

     tag_ids = []
     for tag_name in tag_names:
         tag_name = tag_name.lower()  # приводимо тег до нижнього регістру
          # шукаємо тег у базі даних
         stmt = select(cls.model).filter_by(name=tag_name)
         result = await session.execute(stmt)
         tag = result.scalars().first()

         if tag:
          # якщо тег знайдено додаємо його ID до списку
             tag_ids.append(tag.id)
         else:
          # якщо тег не знайдено, то створюємо новий тег
             new_tag = cls.model(name=tag_name)
             session.add(new_tag)
             try:
                 await session.flush() # створюється новий тег та отримується його ID
                 logger.info( f"Тег' {tag_name}    'додано до бази даних'    )
                 tag_ids.append(new_tag.id)
             except SQLAlchemyError as e:
                 await session.rollback()
                 logger.error(  f " помилка при додаванні тегу'  {tag_name}': {e}")
                 raise e

     return tag_ids
```

Рисунок 2.12 – Лістинг коду додавання блогів та тегів

А от із тегами ми реалізуємо динамічне створення тегів: якщо тега не існує - додаємо, потім створюємо зв'язок «блог ↔ теги» в проміжній таблиці (зв’язок багато-до-багатьох).

Послідовність така:

- Додаємо блог у таблицю блогів.
- Перевіряємо, чи всі теги, які вказав автор, є в таблиці тегів.
- Створюємо теги, яких немає.
- Пов'язуємо блог із тегами в проміжній таблиці.

Ця логіка забезпечує коректну роботу всіх закладених у систему зв'язків. У класі TagDAO опишемо метод:

Метод приймає список рядків (тегів). Далі ми ітеруємося по списку тегів, приводячи їх до нижнього регістру. Потім перевіряємо, чи існує тег у таблиці тегів. Якщо він є, додаємо його ID у вихідний список. Якщо тегу немає, створюємо запис у таблиці тегів і додаємо отриманий ID у список.

На виході отримуємо список ID усіх тегів, які вказав автор при додаванні блогу. Тепер нам потрібно виконати зв’язок ID нового блогу з ідентифікаторами тегів у проміжній таблиці. Для цього опишемо спеціальний метод у класі BlogTagDAO. Логіка схожа на попередню, за винятком того, що спочатку ми генеруємо інстанси (майбутні записи), а потім всі їх одразу фіксуємо в базі даних.

Таким чином, ми готові реалізувати метод для додавання нового блогу в таблицю блогів. Опишемо цей метод, а після повернемося до BlogDAO і додамо додаткові методи, які нам знадобляться для інших задач у системі.

2.4 Створення нового блогу

2.4.1 Створення API блогу

Для цього в папці app/api створюємо файл router.py. Тепер виконуємо імпорти і готуємо роутер (рис. 2.13):

```python
from fastapi import APIRouter, Depends, HTTPException, status, Query
from loguru import logger
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.exc import IntegrityError
from fastapi.responses import JSONResponse
from app.api.schemas import BlogCreateSchemaBase, BlogCreateSchemaAdd, BlogFullResponse, BlogNotFind
from app.auth.dependencies import get_current_user, get_blog_info
from app.auth.models import User
from app.dao.session_maker import SessionDep, TransactionSessionDep
from app.api.dao import BlogDAO, BlogTagDAO, TagDAO

router = APIRouter(prefix='/api', tags=['API'])
```

Рисунок 2.13 – Лістинг коду підготовки створення API блогу

Тепер опишемо перший API-метод для додавання блогу:

```python
@router.post("/add_post/", summary="додавання нового блогу з тегами" )
async def add_blog(
        add_data: BlogCreateSchemaBase,
        user_data: User = Depends(get_current_user),
        session: AsyncSession = TransactionSessionDep
):
    blog_dict = add_data.model_dump()
    blog_dict['author'] = user_data.id
    tags = blog_dict.pop('tags', [])

    try:
        blog = await BlogDAO.add(session=session, values=BlogCreateSchemaAdd.model_validate(blog_dict))
        blog_id = blog.id

        if tags:
            tags_ids = await TagDAO.add_tags(session=session, tag_names=tags)
            await BlogTagDAO.add_blog_tags(session=session,
                                           blog_tag_pairs=[{'blog_id': blog_id, 'tag_id': i} for i in tags_ids])

        return {'status': 'success', 'message': f'Блог с ID {blog_id} вдало додано з тегами.' }
    except IntegrityError as e:
        if "UNIQUE constraint failed" in str(e.orig):
            raise HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
                                detail=' "блог з таким заголовком вже існує." )
        raise HTTPException(status_code=status.HTTP_500_INTERNAL_SERVER_ERROR, detail='" помилка при додаванні блогу."
```

Рисунок 2.14 – Лістинг коду додавання блогу

Відразу почнемо розбір з аргументів ендпойнту. Першим аргументом іде add_data або, іншими словами, сама інформація про новий блог.

Як валідатор тут використовується схема Pydantic BlogCreateSchemaBase. Для цього у папці api створюємо файл schemas.py:

```python
from datetime import datetime
from typing import List
from pydantic import BaseModel, ConfigDict, computed_field, Field


class BaseModelConfig(BaseModel):
    model_config = ConfigDict(from_attributes=True)


class BlogCreateSchemaBase(BaseModelConfig):
    title: str
    content: str
    short_description: str
    tags: List[str] = []
```

Рисунок 2.15 – Лістинг коду файл schemas.py

Тут створено клас BaseModelConfig і прописано параметр from_attributes=True, просто щоб не прокидати це налаштування в кожну схему Pydantic. Сама схема BlogCreateSchemaBase описує поля нашого блогу: назву, контент, короткий опис та теги.

```python
user_data: User = Depends(get_current_user)
```

Цим рядком ми встановили чітку залежність між цим ендпоінтом та функцією:

```python
async def get_current_user(token: str = Depends(get_token), session: AsyncSession = SessionDep)
    try:
        payload = jwt.decode(token, settings.SECRET_KEY, algorithms=settings.ALGORITHM)
    except JWTError:
        raise NoJwtException

    expire: str = payload.get('exp')
    expire_time = datetime.fromtimestamp(int(expire), tz=timezone.utc)
    if (not expire) or (expire_time &lt; datetime.now(timezone.utc)):
        raise TokenExpiredException

    user_id: str = payload.get('sub')
    if not user_id:
        raise NoUserIdException

    user = await UsersDAO.find_one_or_none_by_id(data_id=int(user_id), session=session)
    if not user:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail='User not found')
    return user
```

Рисунок 2.16 – Лістинг коду залежностей

Перед виконанням ендпойнта для додавання блогу спочатку виконується функція перевірки, чи авторизований користувач у системі. Тут або ловиться помилка, або отримуються дані про користувача.

```
session: AsyncSession = TransactionSessionDep
```

Цей аргумент встановлює ще одну залежність - функцію для генерації сесії. Зверніть увагу: тут потрібно використовувати саме TransactionSessionDep, а не SessionDep. Справа в тому, що після завершення логіки ендпоінта add_post нам необхідно зберегти зміни в базі даних. SessionDep таких змін просто не зафіксує.

Далі, після всіх перевірок, починається основна логіка ендпоінта:

- Додаємо блог.
- Додаємо відсутні теги.
- Отримуємо список ID тегів.
- Зв’язуємо теги з блогом у проміжній таблиці.

Зареєструємо наш роутер у main-файлі:

```
from app.api.router import router as router_ap

app.include_router(router_api)
```

Тепер можна виконати запит на додавання поста:

Відкриваємо документацію:

```
http://127.0.0.1:8900/docs
```

Входимо у систему.

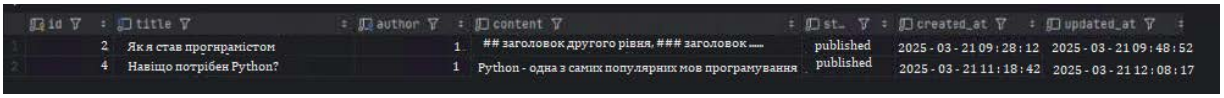
Виконуємо запит на додавання поста. Ось приклад такого запиту:

```
{
  "title": "Навіщо потрібен Python?",
  "content": "Python - одна з самих популярних мов програмування
  "short_description": Текст про те навіщо взагалі потрібно вивчати Python!"
  "tags": ["кодінг "]
}
```

Після додавання можна перевірити, чи з’явилася запис у базі даних. Має статися наступне:

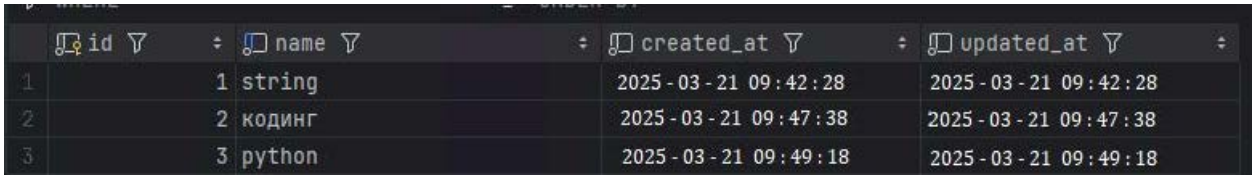
- Додасться сам блог.
- Додасться тег «кодинг».
- З’явиться зв’язок між тегом і блогом.

Ось як виглядають таблиці після додавання кількох постів з різними тегами і від різних авторів (рис.2.17 – 2.19).



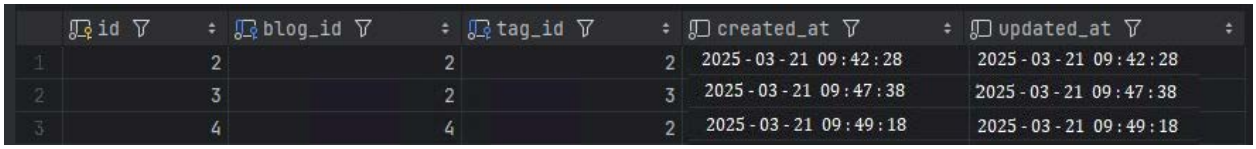
	id	title	author	content	st…	created_at	updated_at
1	2	Як я став програмістом	1	## заголовок другого рівня, ### заголовок	published	2025-03-21 09:28:12	2025-03-21 09:48:52
2	4	Навіщо потрібен Python?	1	Python - одна з самих популярних мов програмування	published	2025-03-21 11:18:42	2025-03-21 12:08:17

Рисунок 2.17 -Таблиця з постами



	id	name	created_at	updated_at
1	1	string	2025-03-21 09:42:28	2025-03-21 09:42:28
2	2	кодинг	2025-03-21 09:47:38	2025-03-21 09:47:38
3	3	python	2025-03-21 09:49:18	2025-03-21 09:49:18

Рисунок 2.18 -Таблиця с тегами



	id	blog_id	tag_id	created_at	updated_at
1	2	2	2	2025-03-21 09:42:28	2025-03-21 09:42:28
2	3	2	3	2025-03-21 09:47:38	2025-03-21 09:47:38
3	4	4	2	2025-03-21 09:49:18	2025-03-21 09:49:18

Рисунок 2.19 - Таблиця, що зв'язує пости і теги

2.4.2 Отримання блогу

Система передбачає можливість приховувати блоги у статусі чернетки. Логічно, що якщо блог у чернетці, його може переглядати лише автор блогу. Відповідно, потрібна логіка, яка перевірятиме, чи авторизований користувач на сайті, щоб визначити, чи є він автором цього блогу.

Але є й інший випадок: ми можемо захотіти, щоб опубліковані блоги бачили всі користувачі сайту.

Виходить, що нам потрібно або реалізувати два методи для перегляду блогу (наприклад, якщо я автор, то маю зайти за однією адресою, а якщо ні -

за іншою), або шукати більш універсальне рішення. Два методи - це незручно і призводить до дублювання коду. Тому ми виберемо інший шлях.

По-перше, реалізуємо окрему залежність. Вона працюватиме схожим чином на Depends(get_current_user), але при цьому дозволить повертати інформацію про блог навіть неавторизованим користувачам.

Для цього внесемо зміни у файл app/auth/dependencies.py. Почнемо з методу отримання JWT-токена користувача:

```
def get_token_optional(request: Request) -> str | None:
    return request.cookies.get('users_access_token')
```

Тут ми не отримаємо помилки, якщо токена немає, а просто отримаємо None, що нас цілком влаштовує.

Тепер додаємо новий метод для отримання інформації про користувача:

```
async def get_current_user_optional(
        token: str | None = Depends(get_token_optional),
        session: AsyncSession = SessionDep
) -> User | None:
    if not token:
        return None

    try:
        payload = jwt.decode(token, settings.SECRET_KEY, algorithms=settings.ALGORITHM)
    except JWTError:
        return None

    expire: str = payload.get('exp')
    expire_time = datetime.fromtimestamp(int(expire), tz=timezone.utc)
    if (not expire) or (expire_time < datetime.now(timezone.utc)):
        return None

    user_id: str = payload.get('sub')
    if not user_id:
        return None

    user = await UsersDAO.find_one_or_none_by_id(data_id=int(user_id), session=session)
    return user
```

Рисунок 2.20 – Лістинг коду отримання інформації про користувача

Цей метод продовжує традиції get_current_user і повертає або інформацію про користувача, або None.

Підготуємо метод, який дозволяє отримати повну інформацію про блог. Опишемо його в BlogDAO. Далі виконуємо знайомий запит для отримання однієї записи, у даному випадку - повної інформації про блог, або None, якщо блог не існує.

Якщо блог з вказаним ID не знайдений - повертаємо повідомлення про помилку. Якщо ж знайдений - перевіряємо, чи статус блогу «чернетка». Якщо так - повертаємо повідомлення, що переглянути блог може лише автор.

Якщо блог існує і користувач має право на перегляд - повертаємо повну інформацію. Тепер можна описати залежність для отримання блогу:

```
async def get_blog_info(
        blog_id: int,
        session: AsyncSession = SessionDep,
        user_data: User | None = Depends(get_current_user_optional)
) -> BlogFullResponse | BlogNotFind:
    author_id = user_data.id if user_data else None
    return await BlogDAO.get_full_blog_info(session=session, blog_id=blog_id, author_id=author_id)
```

На основі цієї залежності опишемо API-метод для отримання блогу:

```
@router.get('/get_blog/{blog_id}', summary = "Отримати інформацію по блогу "    )
async def get_blog_endpoint(
        blog_id: int,
        blog_info: BlogFullResponse | BlogNotFind = Depends(get_blog_info)
) -> BlogFullResponse | BlogNotFind:
    return blog_info
```

Цей метод працює завдяки механізму інспекції (reflection) у Python та анотаціям типів. FastAPI аналізує сигнатуру функції, бачить, що в залежності є параметр blog_id, і автоматично знаходить відповідний параметр у маршруті з таким самим ім’ям і типом.

Проще кажучи, фреймворк "вгадує" потрібний параметр за ім’ям і типом, незалежно від порядку оголошення. Завдяки цьому ми отримали досить лаконічний API-метод, що дозволяє отримати інформацію про блог будь-якому користувачу, який має право його переглядати, навіть якщо він не авторизований.

2.4.3 Метод видалення блогу

Тут ми не можемо дозволити видаляти блоги будь-якому авторизованому користувачу. Важливо перевірити: чи дійсно користувач є автором блогу. Тому метод для видалення блогу виглядає так:

```python
@router.delete('/delete_blog/{blog_id}', summary="видалити блог")
async def delete_blog(
        blog_id: int,
        session: AsyncSession = TransactionSessionDep,
        current_user: User = Depends(get_current_user)
):
    result = await BlogDAO.delete_blog(session, blog_id, current_user.id)
    if result['status'] == 'error':
        raise HTTPException(status_code=400, detail=result['message'])
    return result
```

Рисунок 2.21 – Лістинг коду API-ендпоінт видалення блогу

Це стандартний підхід до видалення блогу, з єдиною відмінністю: перед фактом видалення ми перевіряємо, чи має користувач право на це.

2.4.4 Метод зміни статусу блогу (чернетка / опубліковано)

Тепер опишемо метод для зміни статусу блогу: з "чернетка" на "опубліковано" або навпаки. Метод наведено у Додатку Б.

API-ендпоінт для зміни статусу виглядатиме так:

```python
@router.patch('/change_blog_status/{blog_id}', summary="Змінити статус блогу")
async def change_blog_status(
        blog_id: int,
        new_status: str,
        session: AsyncSession = TransactionSessionDep,
        current_user: User = Depends(get_current_user)
):
    result = await BlogDAO.change_blog_status(session, blog_id, new_status, current_user.id)
    if result['status'] == 'error':
        raise HTTPException(status_code=400, detail=result['message'])
    return result
```

Рисунок 2.22 – Лістинг коду API-ендпоінт зміни статусу

Цей ендпоінт акуратно реалізує всю описану вище логіку, роблячи процес зміни статусу блогу безпечним і чітко контрольованим.

2.4.5 Отримання списку блогів з фільтрами

Необхідно описати універсальний метод, який дозволяє:

- Отримувати всі блоги;
- Отримувати блоги з фільтрацією за автором;
- Отримувати блоги з фільтрацією за тегом;
- Генерувати пагінацію на сервері;
- Отримувати загальну кількість блогів, що відповідають критеріям пошуку.

Реалізація методу наведена у Додатку Б

Отримання списку опублікованих блогів з можливістю фільтрації та пагінації. Тепер опишемо API-метод:

```
@router.get('/blogs/', summary=  " Отримати усі блоги у статусі  'publish'")
async def get_blog_info(
        author_id: int | None = None,
        tag: str | None = None,
        page: int = Query(1, ge=1, description= "Номер сторінки"),
        page_size: int = Query(10, ge=10, le=100, description=  "Записів на сторінці" ),
        session: AsyncSession = SessionDep,
):
    try:
        result = await BlogDAO.get_blog_list(session=session, author_id=author_id, tapage=page,
                                             page_size=page_size)
        return result if result['blogs'] else BlogNotFind(message= "Блоги не знайдені",='error')
    except Exception as e:
        logger.error(f"  Помилка при отриманні блогу"{e}")
        return JSONResponse(status_code=500, content={"detail":" Помилка серверу" })
```

Рисунок 2.23 – Лістинг коду API-метод списку блогів

У межах нашого проєкту ми підемо нестандартним шляхом і самостійно візуалізуємо наш API за допомогою FastAPI та шаблонізатора Jinja2. Це дозволить нам створити повноцінний вебзастосунок без окремої команди фронтенд-розробників.

2.5 Створення сторінок

Перш ніж ми зв’яжемо API з фронтендом, підготуємо HTML+CSS шаблони сторінок. Всього буде 3 сторінки:

1. Сторінка «Блог не знайдено або недоступний».
2. Сторінка з конкретним блогом.
3. Сторінка зі списком блогів.

2.5.1 Створення сторінки блогу

Для її створення у папці app/templates створіть файл post.html з кодом, що наведено у Додатку В.

Звертаємо увагу: HTML-шаблон базується на синтаксисі Jinja2. Можна було реалізувати все лише на JavaScript, адже API вже готові. Але ми надаємо перевагу Python і уникаємо зайвого JS:

- У шаблон підставляються дані.
- Якщо користувач - автор блогу, йому дозволено змінювати статус або видаляти блог.

Рядок

```
<script src="/static/js/post.js"></script>
```

натякає, що нам доведеться написати JS-код, який ми потім імпортуємо до цього шаблону. Сам код лежатиме у файлі app/static/js/post.js.

```
function getCookie(name) {
    return document.cookie
        .split(';')
        .map(cookie =&gt; cookie.trim())
        .find(cookie =&gt; cookie.startsWith(`${name}=`))
        ?.split('=')[1] || null;
}
```

Рисунок 2.24 – Лістинг коду функції отримання cookie

```
async function sendRequest(url, method, body = null, jwtToken = null) {
    const headers = {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
    };

    if (jwtToken) {
        headers['Authorization'] = `Bearer ${jwtToken}`;
    }

    try {
        const response = await fetch(url, {
            method,
            headers,
            body: body ? JSON.stringify(body) : null,
        });

        if (!response.ok) {
            let errorData = {};
            try {
                errorData = await response.json();
            } catch (jsonError) {
                console.error('Ошибка парсинга JSON:', jsonError);
            }
            throw new Error(errorData.detail || `HTTP Error: ${response.status}`);
```

Рисунок 2.25 – Лістинг коду універсальної функції запиту до API

Код, який виконується під час завантаження сторінки, відповідає за обробку дій користувача з елементами блогу.

```
document.addEventListener('DOMContentLoaded', () =&gt; {
    const articleContainer = document.querySelector('.article-container');
    if (!articleContainer) {
        console.error("Елемент .article-container не знайдено.
        return;
    }

    const BLOG_DATA = {
        id: articleContainer.dataset.blogId,
        status: articleContainer.dataset.blogStatus,
        author: articleContainer.dataset.blogAuthor,
        jwtToken: getCookie('users_access_token'),
    };

    console.log('BLOG_DATA:', BLOG_DATA);

    const deleteButton = document.querySelector('[data-action="delete"]');
    if (deleteButton) {
        deleteButton.addEventListener('click', () =&gt; {
            if (confirm("Ви впевненні, що бажаєте видалити цей блог?") {
                deleteBlog(BLOG_DATA);
            }
        });
    }

    const statusButton = document.querySelector('[data-action="change-status"]');
```

Рисунок 2.26 – Лістинг коду ініціалізації на сторінці

```
async function deleteBlog({id, jwtToken}) {
    try {
        await sendRequest(`/api/delete_blog/${id}`, 'DELETE', null, jwtToken);
        alert( " блог вдало видалено. перенаправлення …." );
        window.location.href = '/blogs/';
    } catch (error) {
        console.error( "Невдалося видалити блог: ", error);
    }
}

async function changeBlogStatus({id, jwtToken}, newStatus) {
    try {
        const url = `/api/change_blog_status/${id}?new_status=${encodeURIComponent(newStatus)
        await sendRequest(url, 'PATCH', null, jwtToken);
        alert(  "Статус вдало змінено. Сторінка буде оновлена "        );
        location.reload();
    } catch (error) {
        console.error( "Невдалося змінити статус блогу :"    , error);
        alert(  Помилка при зміні статусу блогу. Спробуйте ще раз"          );
    }
}
```

Рисунок 2.27 – Лістинг коду функції для зміни статусу та видалення

Після видалення блогу користувач перенаправляється на сторінку зі списком блогів, а після зміни статусу - сторінка перезавантажується.

2.5.2 Візуалізація сторінок

Тепер реалізуємо шаблон сторінки «Блог не знайдено». Я назвав файл 404.html. Ось як він виглядає

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Блог ID {{ blog_id }} не знайдено </title>
    <link rel="stylesheet" href="/static/style/404.css">
</head>
<body>
<div class="message-container">
    <h1 class="message-title"> Блог не знайдено </h1>
    <p class="message-text">
            Нажаль, блог  <strong>с ID {{ blog_id }} </strong> не знайдено. Можливо автор
     видалив запис або перемістив його до чернетки
    </p>
    <a href="/blogs/" class="button">  Дивитись блоги, що опубліковані  </a>
</div>
</body>
</html>
```

Рисунок 2.27 – Лістинг коду візуалізації сторінки «Блог не знайдено»

Остання сторінка - зі списком блогів (posts.html). Тут, окрім використання шаблонізатора Jinja2, нічого складного немає. Єдине, що може викликати запитання - це блок із пагінацією. Проте, якщо уважно подивитися, це всього лише стандартний цикл for з Python, який обробляє дані, що повертає сервер.

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Блоги</title>
    <link rel="stylesheet" href="/static/style/posts.css">
</head>
<body>
<div class="content-container">
    <div class="page-header">
        <h1><a href="/blogs/">Всі блоги </a></h1>
    </div>

    <! --- Список статтей ---->
    <ul class="articles-list">
        {% for blog in article.blogs %}
        <li class="article-card">
            <h2><a href="/blogs/{{ blog.id }}">{{ blog.title }}</a></h2>
            <div class="article-meta">
                {% if blog.author_id %}
                <a href="?author_id={{ blog.author_id }}">{{ blog.author_name }}</a>
                {% else %}
        </li>
        {% endfor %}
    </ul>

    <!---- Пагінація ---->
    <div class="pagination">
        {% if article.page > 1 %}
        <a href="?page={{ article.page - 1 }}{% if filters.author_id %}&author_id={{ filters..
           class="pagination-link">←</a>
        {% endif %}
        {% for p in range(1, article.total_page + 1) %}
        <a href="?page={{ p }}{% if filters.author_id %}&author_id={{ filters.author_id }}{%
           class="pagination-link {% if p == article.page %}active{% endif %}">{{ p }}</a>
        {% endfor %}
        {% if article.page < article.total_page %}
        <a href="?page={{ article.page + 1 }}{% if filters.author_id %}&author_id={{ filters..
           class="pagination-link">→</a>
        {% endif %}
    </div>
</div>
</body>
</html>
```

Рисунок 2.28 – Лістинг коду візуалізації сторінки зі списком блогів

2.5.3 Підготовка FastAPI до рендерингу сторінок

Тепер, щоб наші сторінки почали відображатися у FastAPI, необхідно реалізувати відповідні методи API. Для цього створюємо папку app/pages, а в ній файл router.py. Імпорти для роботи з роутером:

```
from fastapi import APIRouter, Depends, Request
from fastapi.templating import Jinja2Templates
from sqlalchemy.ext.asyncio import AsyncSession

from app.api.dao import BlogDAO
from app.api.schemas import BlogFullResponse, BlogNotFind
from app.auth.dependencies import get_blog_info, get_current_user_optional
import markdown2

from app.auth.models import User
from app.dao.session_maker import SessionDep

router = APIRouter(tags=['ФРОНТЕНД'])

templates = Jinja2Templates(directory='app/templates')
```

Рисунок 2.29 – Лістинг коду файл router.py

Зверніть увагу: ми імпортували бібліотеку markdown2, яка знадобиться для правильного перетворення Markdown-текстів у HTML. Також ми налаштували шаблонізатор через рядок:

```
templates = Jinja2Templates(directory='app/templates')
```

Тепер можна використовувати шаблони у ендпойнтах FastAPI.

Ключові моменти:

– Аргумент request обов’язковий для передачі об’єкта запиту в шаблон.

– Валідація та перетворення даних блогу виконується через модель BlogFullResponse.

– Якщо блог не знайдено - рендериться 404.html.

– Якщо знайдено - контент перетворюється з Markdown у HTML і рендериться post.html.

```
@router.get('/blogs/{blog_id}/')
async def get_blog_post(
        request: Request,
        blog_id: int,
        blog_info: BlogFullResponse | BlogNotFind = Depends(get_blog_info),
        user_data: User | None = Depends(get_current_user_optional)
):
    if isinstance(blog_info, dict):
        return templates.TemplateResponse(
            "404.html", {"request": request, "blog_id": blog_id}
        )
    else:
        blog = BlogFullResponse.model_validate(blog_info).model_dump()
        #  Перетворення  Markdown в HTML
        blog['content'] = markdown2.markdown(blog['content'], extras=['fenced-code-blocks', '
        return templates.TemplateResponse(
            "post.html",
            {"request": request, "article": blog, "current_user_id": user_data.id if user_dat
        )
```

Рисунок 2.30 – Лістинг коду ендпоінт для перегляду блогу

Цей рядок:

```
blog['content'] = markdown2.markdown(blog['content'], extras=['fenced-code-blocks', 'tables'])
```

чітко показує, як легко в Python перетворювати Markdown у HTML прямо перед рендерингом сторінки.

У файлі main.py підключаємо роутер:

```
from app.pages.router import router as router_page
app.include_router(router_page)
```

Рисунок 2.31 – Лістинг коду реєстрації роутера

Переконаємось, що у нас є рядок для підключення статичних файлів:

```
app.mount('/static', StaticFiles(directory='app/static'), name='static')
```

Цей рядок повідомляє FastAPI: «Усі файли з папки app/static будуть доступні через URL, що починається з /static». Наприклад, зображення, збережене в app/static/logo.png, буде доступне за адресою: http://localhost:8000/static/logo.png

```python
@router.get('/blogs/')
async def get_blog_post(
        request: Request,
        author_id: int | None = None,
        tag: str | None = None,
        page: int = 1,
        page_size: int = 3,
        session: AsyncSession = SessionDep,
):
    blogs = await BlogDAO.get_blog_list(
        session=session,
        author_id=author_id,
        tag=tag,
        page=page,
        page_size=page_size
    )
    return templates.TemplateResponse(
        "posts.html",
        {
            "request": request,
            "article": blogs,
            "filters": {
                "author_id": author_id,
                "tag": tag,
            }
        }
    )
```

Рисунок 2.32 – Лістинг коду ендпоінт для списку блогів

2.6 Тестування блогу

Тепер безпосередньо можна перейти до локального тестування блогу навіть без його розгортання на хостингу. Таким чином можна перевірити працездатність всіх модулів перед його деплоєм.

При натисканні на автора нас перекидає на сторінку постів автора (/blogs/?author_id=2).

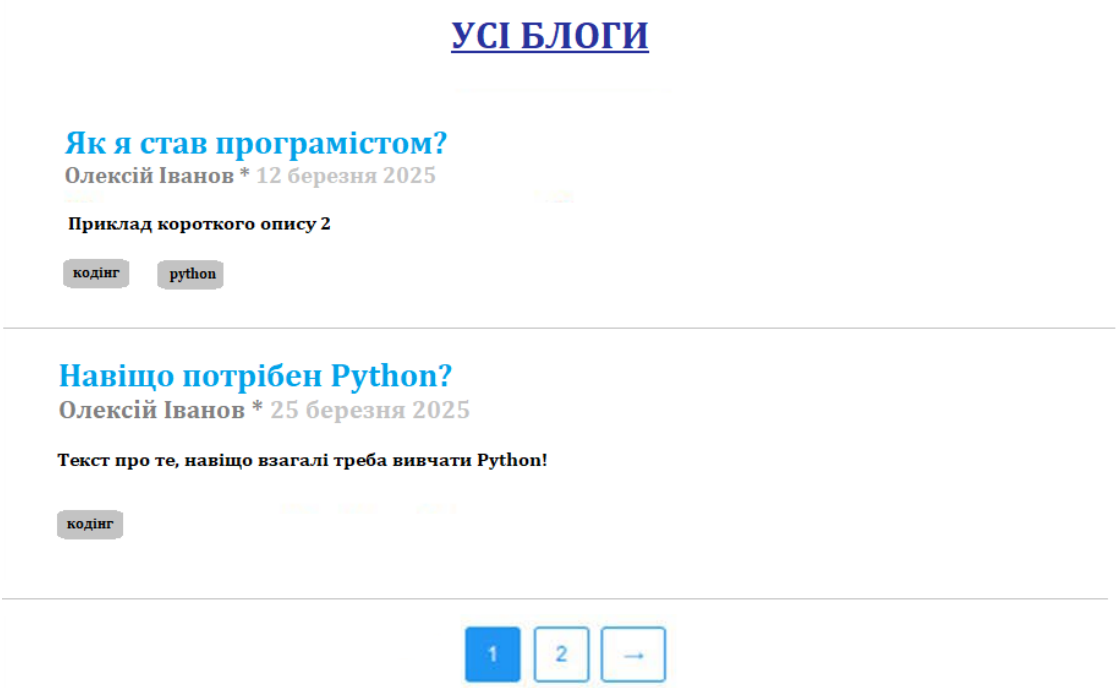


Рисунок 2.33 – Пости автору блогу

При натисканні на тег перекидає на сторінку з тегами (/blogs/?tag=веб-розробка). При натисканні на назву статті відкривається сама стаття.

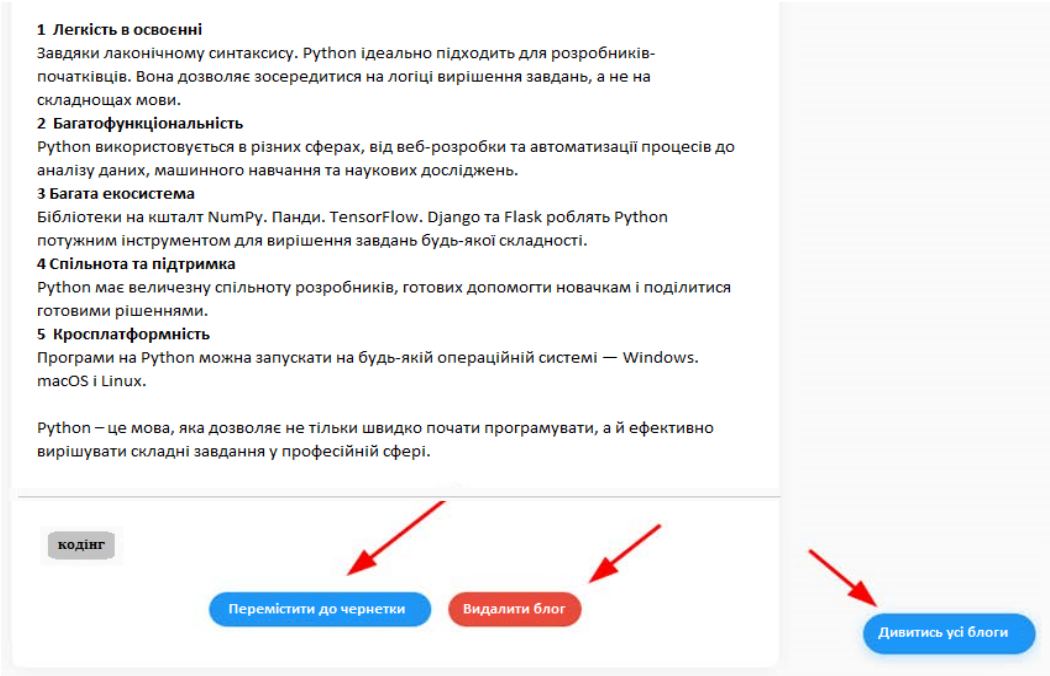


Рисунок 2.34 – Розгорнута стаття

Після авторизації через API-документацію і після відкриття статті за моїм авторством я отримую варіанти зі зміною статусу статті та її видаленням.

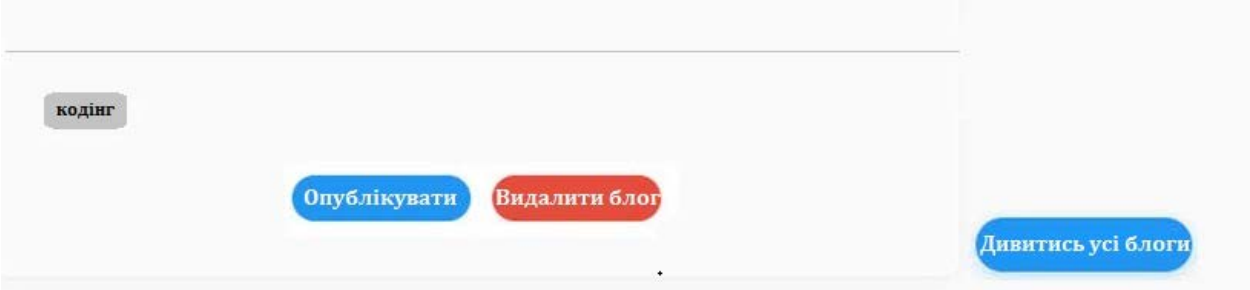


Рисунок 2.35 – Функції публікації та видалення

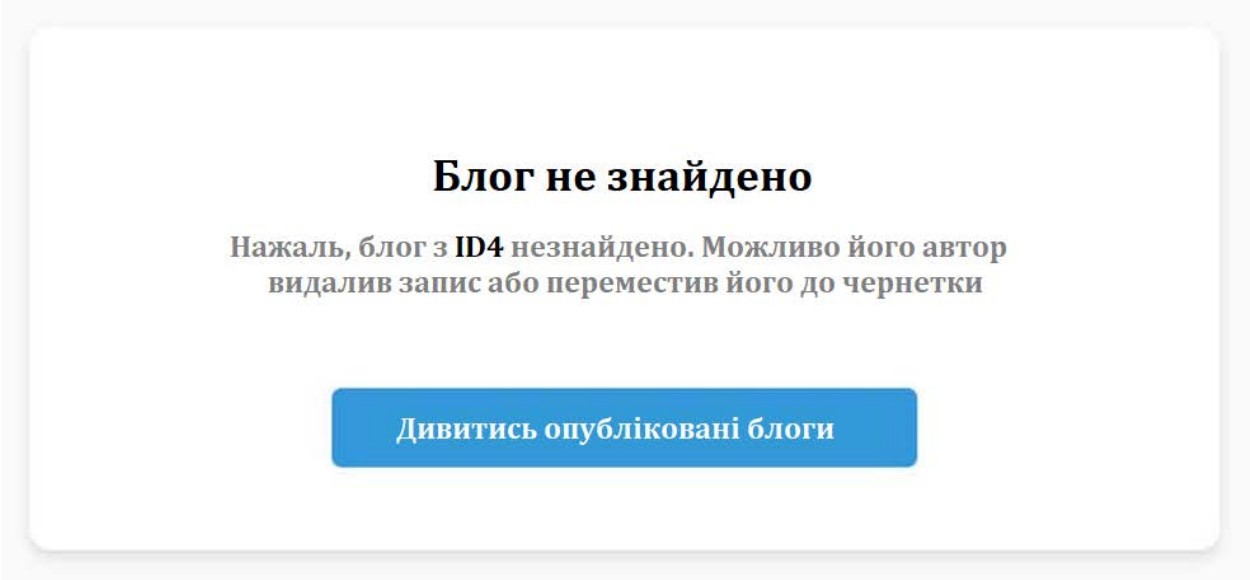


Рисунок 2.36 – Відкриття статті у режимі інкогніто

Після зміни статусу на «Опубліковано» стаття стає доступною навіть не авторизованим користувачам.

Висновки до розділу:

У цьому розділі було розроблено міні-блог із використанням модульної архітектури, що дало змогу чітко структурувати додаток та забезпечити гнучкість у його розширенні. Було визначено архітектурні рішення та функціональні можливості міні-блогу, включаючи управління контентом,

реалізацію API для створення, отримання, видалення та зміни статусу блогів, а також підтримку тегів і фільтрів для зручного пошуку контенту.

Здійснено розробку таблиць для зберігання даних, налаштовано механізми міграцій та виконано інтеграцію з ORM для забезпечення коректної роботи бази даних. Було реалізовано ключові методи API, зокрема додавання нових блогів, отримання існуючих, видалення та управління статусом публікації. Також здійснено розробку інтерфейсу сторінок, реалізовано візуалізацію контенту та забезпечено коректне рендерування за допомогою FastAPI.

На завершення було проведено тестування всіх функціональних блоків для перевірки їхньої коректності та стабільності роботи додатку. Таким чином, розділ демонструє повноцінний процес розробки та тестування міні-блогу, що відповідає поставленим завданням та технічним вимогам.

ВИСНОВКИ

У результаті виконання цієї роботи було досягнуто поставленої мети - створено міні-блог із модульною архітектурою, що забезпечує користувачам можливість реєструватися, автентифікуватися, створювати, редагувати та переглядати власні пости. Реалізований додаток відповідає сучасним вимогам безпеки, функціональності та зручності користування, підтримує захищений доступ до особистих даних користувачів та ефективне управління контентом.

На етапі теоретичного аналізу було досліджено сучасний розвиток блогів як соціокультурного явища та комунікаційного інструмента. Проаналізовано ключові компоненти сучасних блогів, зокрема: систему керування контентом (CMS), дизайн та налаштування, управління користувачами, медіа-контент, SEO-оптимізацію, коментарі та соціальну взаємодію, а також аналітику та статистику. Проведено порівняльний аналіз найпопулярніших платформ для ведення блогів (Ghost, WordPress, Blogger, Wix, Squarespace), що дозволило оцінити їхні сильні та слабкі сторони та визначити напрями для удосконалення власного веб-додатку.

У ході практичної частини було визначено архітектуру та набір інструментів, необхідних для реалізації міні-блогу: веб-фреймворк FastAPI, ORM, базу даних, механізм міграцій, засоби аутентифікації та авторизації, рендеринг HTML та конвертацію Markdown. Це дало змогу створити гнучкий, масштабований та легкий у розширенні додаток, що відповідає сучасним вимогам до веб-розробки.

Було розроблено ключові функціональні можливості блогу, зокрема API для створення, отримання, видалення та зміни статусу блогів, а також реалізовано підтримку тегів і фільтрів для зручного пошуку. Особливу увагу приділено структурі таблиць для зберігання даних, налаштуванню міграцій та інтеграції з ORM для забезпечення коректної роботи бази даних. Розроблено інтерфейс сторінок та реалізовано їх рендеринг за допомогою FastAPI.

На завершення проведено тестування всіх функціональних блоків, що підтвердило стабільність та правильність роботи додатку відповідно до технічних вимог.

Таким чином, робота демонструє комплексний підхід до аналізу, проєктування та реалізації сучасного веб-додатку для блогінгу. Досягнуто повного циклу розробки - від теоретичного обґрунтування до практичної реалізації та тестування. Це підтверджує ефективність обраної архітектури та інструментарію, а також відкриває можливості для подальшого вдосконалення та розширення функціоналу міні-блогу відповідно до потреб користувачів.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Антонова З. О., Жиловська Т. Л., Руденок А. І. Психологічні особливості блогінгу як різновиду віртуального емоційного лідерства. Теорія і практика сучасної психології. 2020. № 2. С. 8–13. URL: http://www.tpspjournal.kpu.zp.ua/archive/2_2020/3.pdf

2. Блейн Н. 10 найкращих платформ для ведення блогів у 2023 році. Webnus. URL: https://webnus.net/uk/best-blogging-platforms/ (Дата звернення 25.04.2025)

3. Блог. Офіційний сайт Вікіпедія. URL: https://uk.wikipedia.org/wiki/Блог (Дата звернення 25.04.2025)

4. Гавран І. А., Грабарчук О. М., Грубич К. В. Жанровотематична класифікація відеоблогінга: науковий підхід. Вісник Національної академії керівних кадрів культури і мистецтв : наук. журнал. 2021. № 3. С. 92–97.

5. Кастомізація інтерфейсу. Wikis.fandom. URL: https://wikis.fandom.com/uk/wiki/Кастомізація_інтерфейсу (Дата звернення 25.04.2025)

6. Класифікація блогів, основні види і характеристика, порівняльний аналіз. 2013. URL: https://www.referat911/Jurnalistika/klasifkacyablog vosnovnvidi/648011547849place2.htm

7. Codreanu Tatiana, C. Combe Celik. Vlogs, Video Publishing, and Informal Language Learning. Mark Dressman and Randall Sadler. The Handbook of Informal Language Learning, Wiley. 2019. pp. 153-168. URL: https://hal.science/hal-02194778/document (Дата звернення 25.04.2025)

8. Google AdSense: The Pioneering Online Advertising Platform. History Tools.. URL: https://www.historytools.org/companies/adsense-history (Дата звернення 25.04.2025)

9. Heather Armstrong Was the Original Influencer. The New York Times. URL: https://www.nytimes.com/2023/05/11/well/family/heather-armstrong-blog-dooce.html (Дата звернення 25.04.2025)

10. Jarvis J. The Man Is Gone, But Long Live The Blogosphere. NPR. URL: https://www.npr.org/2010/01/06/122277812/the-man-is-gone-but-long-live-the-blogosphere (Дата звернення 25.04.2025)

11. Myspace. Official web-site: Вікіпедія. URL: https://uk.wikipedia.org/wiki/Myspace (Дата звернення 25.04.2025)

12 Rozov, I. Comparative Analysis of Python Web Frameworks. Journal of Software Engineering.2023. p.178

13 Patterson, A. Building REST APIs with Flask. O'Reilly Media. 2022, 475p.

14 Smith, J. Modern Web Development with FastAPI. Packt Publishing. 2023, 275p.

15 Green, L.. Database Management with SQLAlchemy. Apress. 2021, 370p.

16 Martinez, K. Async Python and ORMs. Springer. 2022, 190p.

17 Taylor, D. SQLite for Small Projects. Medium Article. 2022. 410p.

18 Alembic Documentation url: https://alembic.sqlalchemy.org/en/latest/(Дата звернення 25.04.2025)

19 FastAPI Documentation. url: https://fastapi.tiangolo.com/ (Дата звернення 25.04.2025)

20 ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлення. Київ, ДП «УкрННЦ», 2015. 26с. (Інформація та документація).

21 ДСТУ 8302:2015. Бібліографічне посилання. Загальні вимоги та правила складання Київ, ДП «УкрННЦ», 2016. 16 с. (Інформація та документація).

22 ДСТУ 3582:2013. Бібліографічний опис. Скорочення слів і словосполучень в українській мові. Загальні вимоги та правила. Київ, ДП «УкрННЦ», 2013. 23 с. (Інформація та документація).

ДОДАТОК А

Лістинг коду методу для зміни статусу блогу: "черновик" - "опублікован"

```
@classmethod
async def change_blog_status(cls, session: AsyncSession, blog_id: int, new_status:
str, author_id: int) -> dict:
    """Метод для зміни статусі блогі. Зміна можлива тільки автором блогу.
    :param session: Асинхронна сесія SQLAlchemy
    :param blog_id: ID блога
    :param new_status: Новий статус блогу ('draft' або 'published')
    :param author_id: ID автора, що намагається змінити блог
    :return: Словник з результатами операцій"""
    if new_status not in ['draft', 'published']:
        return {
          'message': "Недопустимий статус. Використовуйте 'draft' або 'published'.",
         'status': 'error'
        }
    try:
        # Пошук блогу по ID
        query = select(cls.model).filter_by(id=blog_id)
        result = await session.execute(query)
        blog = result.scalar_one_or_none()
        if not blog:
            return {
                'message': f"Блог з ID {blog_id} не знайдено.",
                'status': 'error'
            }
        # Перевіряємо, є користувач автором блогу
        if blog.author != author_id:
            return {
                'message': "У вас немає прав на зміну статусу цього блогу.",
                'status': 'error'
            }
        # Якщо поточний статус співпадає з новим, повертаємо повідомлення без змін
        if blog.status == new_status:
            return {
                'message': f"Блог вже має статус '{new_status}'.",
                'status': 'info',
                'blog_id': blog_id,
                'current_status': new_status
            }
        # Змінюємо статус блогу
        blog.status = new_status
        await session.flush()
        return {
            'message': f"Статус блогу вдало змінено на '{new_status}'.",
            'status': 'success',
            'blog_id': blog_id,
            'new_status': new_status
        }
    except SQLAlchemyError as e:
        await session.rollback()
        return {
            'message': f" Виникла помилка при зміні статусу блогу: {str(e)}",
            'status': 'error'
        }
```

ДОДАТОК Б

Лістинг коду списку блогу

```
@classmethod
async def get_blog_list(cls, session: AsyncSession, author_id: Optional[int] = None,
tag: Optional[str] = None,
    page: int = 1, page_size: int = 10):
    """
    Отримуємо список блогів з можливістю фільтрації й пагонації.
    :param session: Асинхронна сесія SQLAlchemy
    :param author_id: ID автора для фільтрації (опційно)
    :param tag: Назва тегів для фільтрації (опційно)
    :param page: Номер сторінки (починая з 1)
    :param page_size: Кількість записів на сторінці (від 3 до 100)
    :return: Словник з ключами page, total_page, total_result, blogs
    """
    # Обмеження параметрів
    page_size = max(3, min(page_size, 100))
    page = max(1, page)
    # Початкова збірка базового запиту
    base_query = select(cls.model).options(
        joinedload(cls.model.user),
        selectinload(cls.model.tags)
    ).filter_by(status='published')
    # Фільтрація зв автором
    if author_id is not None:
        base_query = base_query.filter_by(author=author_id)
    # Фільтрація за тегом
    if tag:
        base_query =
base_query.join(cls.model.tags).filter(cls.model.tags.any(Tag.name.ilike(f"%{tag.lowe
r()}%")))
    # Підрахунок загадьної кількості записів
    count_query = select(func.count()).select_from(base_query.subquery())
    total_result = await session.scalar(count_query)
    # Якщо записів немає, повертаємо порожній результат
    if not total_result:
        return {
            "page": page,
            "total_page": 0,
            "total_result": 0,
            "blogs": []
        }
    # Розрахунок кількості сторінок
    total_page = (total_result + page_size - 1) // page_size
    # Застосування пагінації
    offset = (page - 1) * page_size
    paginated_query = base_query.offset(offset).limit(page_size)
    # Виконання запиту і отримання результату
    result = await session.execute(paginated_query)
    blogs = result.scalars().all()
    # Видалення дублікатів блогу за їх ID
    unique_blogs = []
    seen_ids = set()
    for blog in blogs:
        if blog.id not in seen_ids:
            unique_blogs.append(BlogFullResponse.model_validate(blog))
            seen_ids.add(blog.id)
    # Логування
    filters = []
```

```
    if author_id is not None:
        filters.append(f"author_id={author_id}")
    if tag:
        filters.append(f"tag={tag}")
    filter_str = " & ".join(filters) if filters else "no filters"
    logger.info(f"Page {page} fetched with {len(blogs)} blogs, filters:
{filter_str}")
    # Формування результату
    return {
        "page": page,
        "total_page": total_page,
        "total_result": total_result,
        "blogs": unique_blogs
    }
```

ДОДАТОК В

Лістинг коду створення сторінки блогу

```
<!DOCTYPE html>
<html lang="ru">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ article.title }}</title>
    <link rel="stylesheet" href="/static/style/post.css">
</head>
<body>
<article class="article-container"
         data-blog-id="{{ article.id }}"
         data-blog-status="{{ article.status }}"
         data-blog-author="{{ article.author }}">
    <h1 class="article-title">{{ article.title }}</h1>
    <div class="article-meta">
        <a href="/blogs?author_id={{ article.author }}">{{ article.author_name }}</a>
        {{ article.created_at.strftime('%d %B %Y') }}
    </div>
    <div class="article-content">
        {{ article.content|safe }}
    </div>
    <div class="tags-container">
        {% if article.tags %}
        <ul class="tags">
            {% for tag in article.tags %}
            <li><a href="/blogs?tag={{ tag.name }}" class="tag">{{ tag.name
}}</a></li>
            {% endfor %}
        </ul>
        {% else %}
        <p>Не має тегів для цієї статті.</p>
        {% endif %}
    </div>
    {% if current_user_id == article.author %}
    <div class="article-actions">
        {% if article.status == 'published' %}
        <button class="button status-button" data-action="change-status" data-new-
status="draft">
            Перемістити до чернеток
        </button>
        {% else %}
        <button class="button status-button" data-action="change-status" data-new-
status="published">
            Опублікувати
        </button>
        {% endif %}
        <button class="button delete-button" data-action="delete">Видалити
блог</button>
    </div>
    {% endif %}
</article>
<div class="view-blogs">
    <a href="/blogs" class="button view-blogs-button">Дивитись усі блоги</a>
</div>
<script src="/static/js/post.js"></script>
</body>
</htm
```