МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНЕ НЕКОМЕРЦІЙНЕ ПІДПРИЄМСТВО "ДЕРЖАВНИЙ УНІВЕРСИТЕТ "КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ""
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Мікросервісний вебзастосунок «Блог» для асинхронної обробки запитів із використанням Django, Celery і RabbitMQ»

Виконавець: Костянтин ЛАГОДА

Керівник: к.т.н., доцент Вікторія СИДОРЕНКО

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2024

ДЕРЖАВНЕ НЕКОМЕРЦІЙНЕ ПІДПРИЄМСТВО "ДЕРЖАВНИЙ УНІВЕРСИТЕТ "КИЇВСЬКИЙ АВІАЦІЙНИЙ ІНСТИТУТ""

Факультет *комп'ютерних наук та технологій*

Кафедра *комп'ютерних інформаційних технологій*

Спеціальність *122 «Комп'ютерні науки»*

Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
Завідувач кафедри
_____Аліна САВЧЕНКО
(підпис)
«___»_____2024 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи

Лагоди Костянтина Дмитровича
(ПІБ випускника)

1. Тема кваліфікаційної роботи: «Мікросервісний вебзастосунок «Блог» для асинхронної обробки запитів із використанням Django, Celery і RabbitMQ» затверджена наказом ректора № 1782/ст від 06.09.2024р.

2. Термін виконання роботи: з 26 серпня 2024 року по 03 грудня 2024 року.

3. Вихідні дані до роботи: застосунок на мовах програмування Python, JavaScript з використанням технологій Django, Celery і RabbitMQ для демонстрації мікросервісного веб-додатку «Блог».

4. Зміст пояснювальної записки: 1. Огляд предметної області. 2. Проектування веб-додатку 3. Розробка та презентація веб-додатку.

5. Перелік обов'язкового ілюстративного матеріалу: 1. Історія веб-блогів. 2. Проблема монолітних веб-додатків без асинхронного виконання завдань. 3. Методи вирішення проблеми. 4. Вибір та обґрунтування інструментів проектування. 5. Розробка серверної та клієнтської частин додатку. 6. Презентація основного функціоналу

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1	Огляд та аналіз предметної області. Написання 1 розділу, представлення керівнику.	26.08.2024-20.09.2024	
2	Вибір та опис використаних технологій. Розробка вебзастосунку. Написання 2 розділу, представлення керівнику.	23.09.2024-10.10.2024	
3	Написання 3 розділу, представлення керівнику.	11.10.2024-14.11.2024	
4	Загальне редагування та друк пояснювальної записки.	15.11.2024-19.11.2024	
5	Проходження нормоконтролю, перепліт пояснювальної записки.	20.11.2024-25.11.2024	
6	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації	26.11.2024-03.12.2024	

7. Дата видачі завдання 26.08.2024р.

Керівник кваліфікаційної роботи _____ Вікторія СИДОРЕНКО
(підпис керівника)

Завдання прийняв до виконання _____ Костянтин ЛАГОДА
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Мікросервісний вебзастосунок «Блог» для асинхронної обробки запитів із використанням Django, Celery і RabbitMQ» містить: 102 сторінки, 44 рисунки, 18 інформаційних джерел, 1 додаток.

Об'єкт дослідження – процес асинхронної обробки даних.

Предмет дослідження – методи, засоби та технології розробки веб-додатків для асинхронної обробки запитів.

Мета кваліфікаційної роботи – розробка мікросервісного вебзастосунку «Блог» для асинхронної обробки запитів з використанням сучасних фреймворків, бібліотек та інструментів.

Методи дослідження – логічний, алгоритмічний аналіз, порівняльний, аналіз інформаційних джерел, моделювання та симуляція.

Результати кваліфікаційної роботи можуть бути використані для розробки масштабованих і надійних веб-додатків, які потребують ефективної асинхронної обробки запитів. Зокрема, реалізована архітектура мікросервісного блогу з використанням фреймворку Django, Celery та RabbitMQ може бути адаптована для інших проектів, що вимагають високої продуктивності, зниження навантаження на сервери та покращення якості користувацького досвіду. Отримані результати також можуть бути корисними для розробників, які займаються створенням систем з розподіленою обробкою задач і високими вимогами до надійності та масштабованості.

ВЕБ-ДОДАТОК, БЛОГ, PYTHON, JAVASCRIPT, DJANGO, CELERY, RABBITMQ, REDIS, WEB.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

HTML	–	Hyper Text Markup Language – мова гіпертекстової розмітки
CSS	–	Cascade Style Sheet – каскадні таблиці стилів
API	–	Application Programming Interface - інтерфейс прикладного програмування
JSON	–	JavaScript Object Notation – формат запису об’єктів JavaScript
HTTP	–	Hyper Text Transfer Protocol – протокол передачі гіпертексту
URL	–	Uniform Resource Locator - Уніфікований покажчик інформаційного ресурсу
ORM	–	Object-Relational Mapping – об'єктно-реляційне відображення, метод доступу до бази даних через об'єкти
Django	–	високорівневий веб-фреймворк на мові Python для швидкої розробки веб-додатків
Celery	–	асинхронна черга завдань для розподіленої обробки задач у фоновому режимі
RabbitMQ	–	брокер повідомлень для обміну даними між різними частинами системи
Redis	–	інструмент для кешування даних та брокер для черг завдань у Celery
MVC	–	Model-View-Controller – архітектурний шаблон для організації коду в додатках
IP	–	Internet Protocol – протокол передачі даних у мережі
Broker	–	посередник, що передає повідомлення між продюсером і консюмером у системах на

		основі черг
Producer	–	компонент, який відправляє повідомлення в чергу
Consumer	–	компонент, який отримує повідомлення з черги та виконує обробку
Worker	–	процес або потік, який виконує завдання Celery
Beat	–	компонент Celery, що відповідає за планування періодичних завдань

ВСТУП

У сучасному світі цифрові технології відіграють ключову роль у розвитку інформаційного простору, комунікацій та підприємництва. Одним із найпопулярніших засобів взаємодії між людьми є веб-додатки, що дають можливість користувачам обмінюватися інформацією, взаємодіяти один з одним, висловлювати свої думки і творчі ідеї. Важливе місце серед таких веб-додатків посідають блоги, які надають платформу для публікації особистих чи професійних статей, обміну досвідом, думками, відгуками та новинами. Блоги стали невід'ємною частиною сучасної інтернет-культури, пропонуючи користувачам можливість створювати унікальний контент, а також залучати й утримувати аудиторію.

З кожним роком збільшується попит на швидкодію і масштабованість веб-додатків. Зокрема, важливо забезпечити ефективну обробку великої кількості користувацьких запитів і даних, що поступають у реальному часі. Для цього необхідно використовувати сучасні підходи до архітектури програмного забезпечення, зокрема мікросервісну архітектуру, яка дозволяє розділяти систему на окремі незалежні сервіси.

Актуальність теми кваліфікаційної роботи «Мікросервісний вебзастосунок «Блог» для асинхронної обробки запитів із використанням Django, Celery і RabbitMQ» зумовлена необхідністю розробки сучасних веб-додатків, здатних обробляти великі обсяги інформації без значних затримок. Враховуючи постійне зростання кількості користувачів веб-платформ і контенту, який вони генерують, важливо забезпечити стабільну та швидку роботу системи. Для цього використовуються інструменти, такі як Celery для управління асинхронними задачами і RabbitMQ для чергування повідомлень, які дозволяють оптимізувати процес обробки запитів і підвищити ефективність роботи додатка.

Ціль розробки мікросервісного веб-додатку для блогу полягає у створенні платформи, яка здатна забезпечити масштабовану, надійну та асинхронну обробку запитів користувачів. Такий підхід дозволить знизити навантаження на сервери та прискорити час відгуку системи, що, у свою чергу, підвищить якість

користувацького досвіду. Створення такого додатку можливе з використанням фреймворку Django для бекенду, Celery для обробки асинхронних задач і RabbitMQ для чергування задач.

Об'єктом дослідження кваліфікаційної роботи є веб-додаток для блогу, побудований на основі мікросервісної архітектури. Предметом дослідження є процес асинхронної обробки користувацьких запитів із використанням технологій Django, Celery та RabbitMQ.

Метою кваліфікаційної роботи є розробка мікросервісного вебзастосунку «Блог», що підтримує асинхронну обробку запитів, для забезпечення масштабованості та ефективності системи при великій кількості користувачів.

Для досягнення цієї мети було визначено наступні завдання:

-проаналізувати існуючі рішення для асинхронної обробки запитів у веб-додатках;

- розробити архітектуру мікросервісного додатку, яка базується на Django, Celery і RabbitMQ;

- розробити веб-додаток, включаючи асинхронну обробку задач і черги повідомлень як складові його архітектури;

- провести тестування та налаштування зручності для кінцевих користувачів.

Наукова новизна полягає в удосконаленні існуючих підходів до розробки блог-платформи, за рахунок поєданння мікросервісної архітектури з асинхронною обробкою даних, для підвищення швидкості обробки великих масивів даних в умовах інтенсивного використання.

Практичне значення отриманих результатів полягає в можливості застосування розробленого мікросервісного веб-додатку для створення масштабованих і високопродуктивних платформ, які здатні ефективно обробляти велику кількість асинхронних запитів користувачів. Окрім того, рішення може бути використане як основа для подальшого розвитку подібних платформ.

РОЗДІЛ 1
ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Історія веб-блогів

Історія виникнення перших веб-блогів бере свій початок з кінця 1990-х років. Перші блоги були простими особистими сторінками, де люди ділилися своїми думками, новинами та враженнями. Одним із перших веб-сайтів, який можна вважати прототипом сучасного блогу, є «Links.net», створений у 1994 році Джастіном Голлом (Justin Hall), американським студентом. Цей сайт складався із персональних записів, що включали коментарі до посилань на інші ресурси в Інтернеті [1]. Як виглядає даний сайт можна побачити на рис. 1.1.

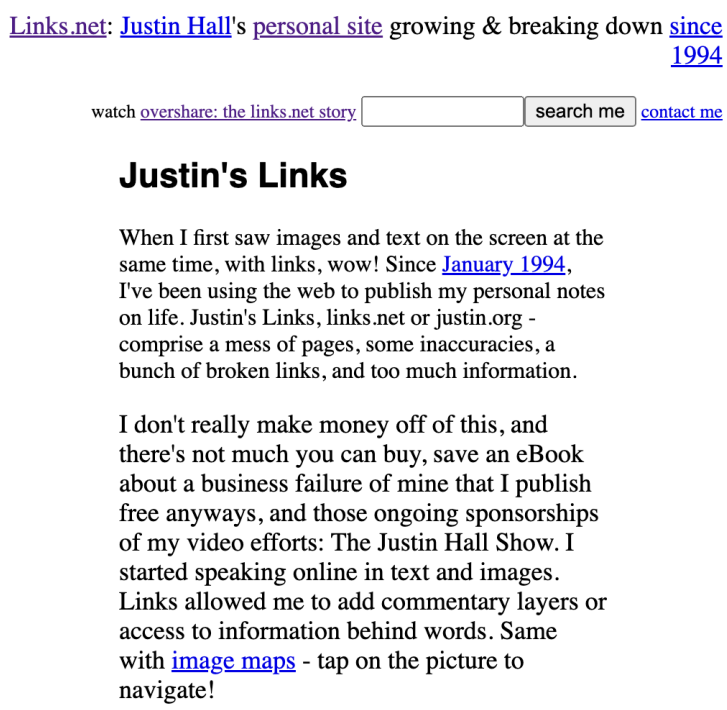

Рис. 1.1. Links.net, який вважається прототипом сучасного блогу

Саме слово "блог" з'явилося пізніше, у 1997 році, коли програміст і письменник Джорн Баргер (Jorn Barger) почав використовувати термін "weblog" для опису процесу ведення інтернет-журналу або «журналу подій в Інтернеті» («log»). Баргер створив свій блог "Robot Wisdom", де збирав і коментував посилання на цікаві статті та ресурси в Інтернеті. У 1999 році Пітер Мерхольц (Peter Merholz) скоротив термін "weblog" до "blog", що швидко стало популярним терміном для позначення персональних онлайн-щоденників(рис. 1.2).

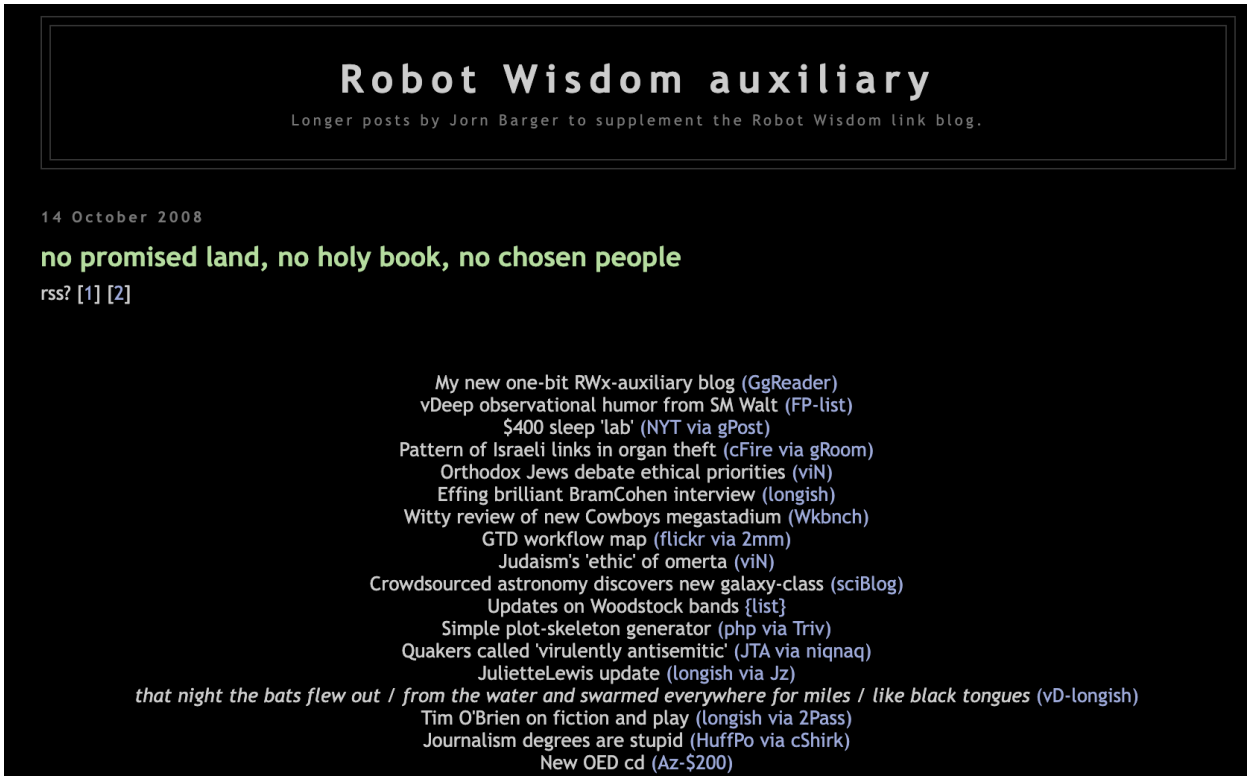

Рис. 1.2. “Robot Wisdom” Джорна Баргера

Завдяки таким ресурсам, як «Blogger», який був заснований компанією Pyra Labs у 1999 році і пізніше куплений Google у 2003 році, процес створення блогів значно спростився. Blogger став першою платформою, що дозволяла користувачам створювати й публікувати свої власні блоги без потреби в глибоких технічних знаннях(рис. 1.3). Цей крок сприяв масовій популяризації блогів серед звичайних користувачів і став початком нового етапу в розвитку інтернет-контенту.

Рис. 1.3. Вигляд, який мав сервіс «Blogger» в 2003 році

Веб-блоги стали одним із ключових чинників, що сприяли глобальній трансформації Інтернету, перетворивши його з простого засобу обміну інформацією на інклюзивну та децентралізовану платформу для самовираження, комунікації та розповсюдження новин. Від свого зародження наприкінці 1990-х років блоги швидко стали впливовим інструментом демократизації інформації та суттєво змінили те, як люди взаємодіють з контентом та одне з одним в онлайні.

Одна з реально найважливіших ролей блогів у розвитку Інтернету полягає в тому, що вони допомогли децентралізувати процес створення і поширення інформації. До появи блогів більшість медіа-контенту контролювали централізовані організації, такі як газети, журнали, телеканали або радіостанції, які мали виключні можливості диктувати порядок денний і вибирати, які новини чи думки будуть доступні широкій аудиторії. Це обмежувало доступ до медіа для широкого загалу, оскільки процес публікації був складним, дорогим і регульованим редакціями та видавцями.

Перші блог-платформи, такі як «Blogger», «LiveJournal» і «WordPress», зробили створення блогу надзвичайно простим завданням, яке не вимагало від користувача знань програмування чи великих фінансових витрат. Кожен охочий отримав змогу створити персональну платформу, що стала інструментом для самовираження. Сам вигляд, який мав LiveJournal в 1999 році, можна побачити на рис. 1.4.

Рис. 1.4. Як виглядав LiveJournal в 1999 році

Ця відкритість сприяла створенню величезної кількості контенту, що висвітлював найрізноманітніші теми. Інтернет наповнився інформацією з різних сфер — від особистих історій до спеціалізованих тем на кшталт технологій, політики, науки, мистецтва тощо. Відтак блогери стали альтернативним джерелом інформації.

Однією з найбільших переваг блогів стало те, що вони надавали

можливість висвітлювати теми, які часто ігнорувалися або перекручувалися.

Блогери отримали змогу писати про суспільно важливі питання, висвітлювати події з різних точок зору та ділитися альтернативними думками, які не завжди знайшли б місце у великих медіакомпаніях. Це дало можливість створювати альтернативний медіапростір, в якому могли бути почуті голоси незалежних авторів.

Відомими прикладами цього стали блоги, що висвітлювали політичні події, соціальні проблеми або конфлікти у різних куточках світу. Так, у періоди воєнних дій, революцій або криз блогери часто ставали першими джерелами інформації, іноді передаючи на світовий рівень новини про події, про які мовчали великі ЗМІ через цензуру або корпоративні інтереси. Наприклад, під час Арабської весни (2010–2012 рр.) блогери й активісти соціальних медіа відіграли важливу роль у поширенні інформації про протестні рухи та залученні міжнародної аудиторії до цих подій.

Крім того, блоги стали потужним інструментом для журналістики даних і незалежних розслідувань. Багато журналістів, що працювали поза великими виданнями, використовували свої блоги для публікації аналітичних матеріалів, досліджень і викриттів. Це сприяло розвитку «громадянської журналістики», де звичайні люди мали змогу стати репортерами і висвітлювати важливі для них події.

Блоги не лише стали засобом поширення інформації, але й місцем, де люди могли будувати спільноти навколо певних інтересів, ідей або цінностей. Кожен блог міг стати точкою тяжіння для людей, які поділяли схожі погляди або інтереси, що сприяло формуванню активних онлайн-спільнот. Ці спільноти стали просторами для обговорення тем, обміну досвідом та ідеями, а також підтримки соціальних зв’язків між їх учасниками.

Це зробило Інтернет більш «живим» і динамічним середовищем для взаємодії, порівняно з традиційними медіа, де зворотний зв’язок з аудиторією був обмеженим.

Як результат, блоги стали одними із основних формуючих факторів для

появи такого Інтернету, яким ми його знаємо сьогодні і чимало особливостей тогочасних блогів перейшли в новітні соціальні мережі.

1.2. Сучасний стан блогосфери

Попри поширення відеоконтенту, подкастів та соціальних мереж, текстові блоги залишаються важливою частиною інтернет-культури і продовжують відігравати значну роль у сучасній інформаційній екосистемі. Текстові блоги зберігають актуальність завдяки глибині аналізу, зручності сприйняття інформації та можливості оптимізації для пошукових систем (SEO). У цьому розділі розглянемо сучасний стан текстових блогів, їхні тенденції та значення у цифрову епоху.

Сучасні текстові блоги акцентують увагу на якості контенту та глибокому аналізі тем. Відвідувачі таких блогів часто шукають докладну інформацію з різних тем: від технологій і бізнесу до подорожей, психології чи здоров'я. Один із прикладів такого блогу — "Wait But Why"[2], автор якого Тім Урбан пише довгі аналітичні статті про складні теми(рис. 1.5).

Рис. 1.5. Сайт-блог "Wait But Why" Тіма Урбана

Завдяки можливості текстових блогів надавати більш деталізовані матеріали, вони часто стають джерелом аналітичної інформації для професіоналів різних галузей. Наприклад, блоги на платформах на кшталт Medium чи Substack часто публікують глибокі статті з економіки, політики чи наук.

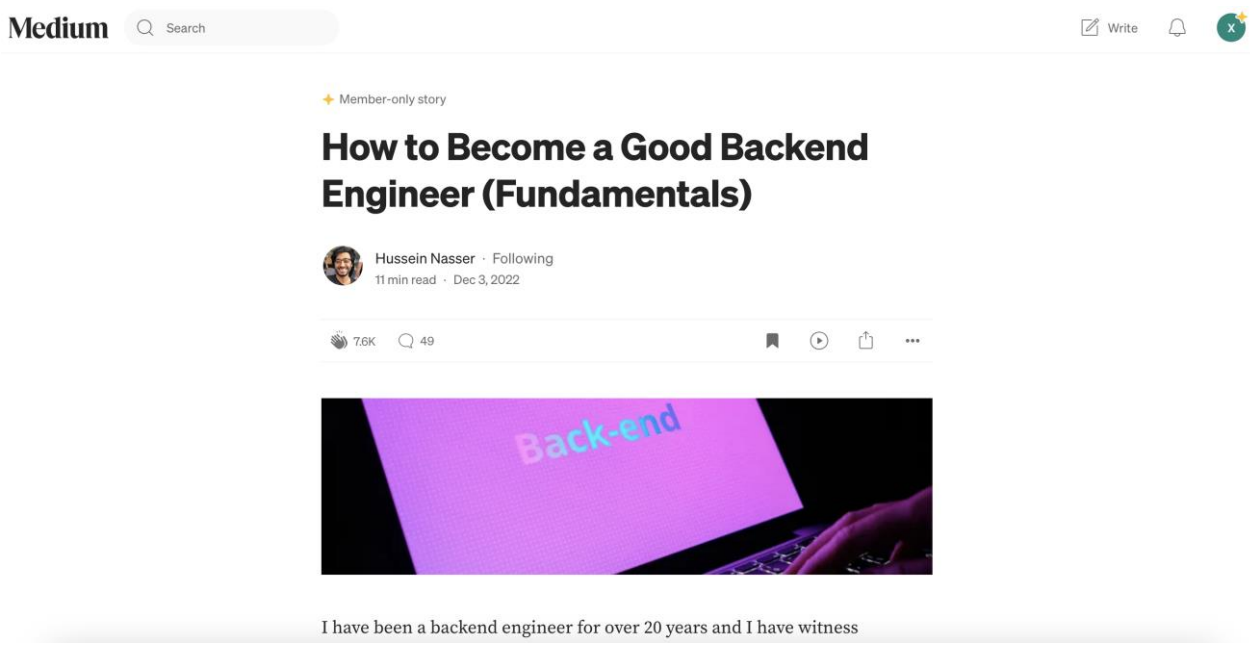

Рис. 1.6. Приклад статті на Medium

Текстові блоги мають одну з головних переваг перед іншими форматами контенту — їхня оптимізація для пошукових систем. Відповідність SEO-критеріям дозволяє блогерам досягати високих позицій у пошукових запитах і залучати органічний трафік. Наприклад, блоги на платформі WordPress дозволяють авторам використовувати інструменти для оптимізації заголовків, ключових слів, метаданих, що сприяє кращій видимості у Google та інших пошукових системах.

Перевага текстових блогів полягає також у тому, що їхні матеріали зберігаються в інтернеті довгий час і можуть постійно приносити трафік з пошукових запитів. Тоді як відео чи подкасти часто потребують додаткового просування через соціальні мережі, текстові блоги мають стійкий потік

відвідувачів завдяки пошуковій оптимізації, зберігаючи актуальність і потребуючи мінімум(відносно інших типів контенту) зусиль.

Для багатьох спеціалістів ведення блогу стало важливим елементом формування особистого бренду. Написання статей на професійні теми дозволяє експертам показати свою компетентність і залучити потенційних клієнтів або роботодавців.

Наприклад, багато IT-фахівців ведуть технічні блоги, де діляться своїм досвідом, пишуть навчальні матеріали та публікують огляди нових технологій. Такі платформи, як LinkedIn та Medium, дозволяють спеціалістам публікувати свої статті на професійні теми, створюючи свій образ авторитетного експерта у галузі.

Текстові блоги мають кілька основних шляхів монетизації:

- Реклама: Багато блогів отримують дохід від розміщення рекламних оголошень через такі мережі, як Google AdSense. Це дозволяє блогерам отримувати дохід від кількості переглядів їхніх статей.
- Партнерські програми: Блогери можуть заробляти, розміщуючи посилання на продукти чи послуги і отримуючи комісію з продажів. Наприклад, блоги про технології часто містять огляди пристроїв із посиланням на магазини.
- Платні підписки та пожертви: Платформи на зразок Substack дозволяють блогерам створювати ексклюзивний контент для підписників, які платять за доступ до статей. Також існують сервіси на зразок Patreon, де читачі можуть фінансово підтримувати авторів.

Текстові блоги, попри розвиток відео та подкастів, залишаються важливим джерелом інформації та аналізу в сучасній блогосфері. Завдяки своєму фокусу на якості контенту, можливості оптимізації для пошукових систем та монетизації, текстові блоги продовжують розвиватися і адаптуватися до сучасних вимог аудиторії.

1.3. Основні складові всіх веб-блогів

Текстові блоги є структурованими цифровими платформами, які

включають певні елементи, що роблять контент зручним для сприйняття та забезпечують позитивний досвід користувачів. Оскільки блоги є засобом комунікації між автором і читачем, кожен блог зазвичай складається з декількох важливих елементів. Основні складові будь-якого блогу можна розділити на кілька категорій: структурні компоненти, навігаційні елементи, інформаційні блоки та інтерактивні функції.

Загалом, можна виділити 10 таких складових:

1. Заголовок (назва блогу) та логотип — це основні елементи, що формують перше враження про блог і допомагають ідентифікувати його серед інших. Вони розташовуються у верхній частині сторінки і надають можливість користувачам одразу зрозуміти тематику або стиль блогу. Наприклад, у блозі про фінансову грамотність заголовок може включати слово "Фінанси", а логотип відображати символи, пов'язані з грошима чи інвестиціями. Це допомагає відвідувачам швидко орієнтуватися та вирішити, чи є цей ресурс корисним для них.

2. Меню навігації — це одна з ключових складових будь-якого блогу, яка допомагає користувачам переміщатися між різними розділами та статтями. Меню може містити посилання на головну сторінку, сторінки з тематичними категоріями, сторінку про автора, контактну інформацію або архіви. Деякі блоги також додають до меню кнопки соціальних мереж або підписки на оновлення. Меню навігації забезпечує зручний досвід користування сайтом, дозволяючи читачам швидко знаходити необхідну інформацію, що особливо важливо для блогів із великою кількістю контенту.

3. Головна сторінка — це центральний елемент блогу, що часто служить вітриною, яка відображає останні публікації, популярні статті чи рекомендований контент. Структура головної сторінки може відрізнятися залежно від тематики блогу та обраного стилю дизайну. На головній сторінці зазвичай відображаються короткі анонси статей або списки категорій. Це дозволяє читачам швидко ознайомитися з новими матеріалами та вирішити, що їх цікавить. У блогах новинного типу головна сторінка може виглядати як стрічка

новин, тоді як у особистих блогах вона може містити акцент на актуальну статтю чи історію.

4. Статті — це основний контент, що наповнює будь-який блог. Кожна публікація має структуру, що складається із заголовка, основного тексту, зображень чи медіа, і часто закінчується закликом до дії або пропозицією залишити коментар. У текстових блогах статті можуть бути оглядами, інструкціями, аналітичними матеріалами або особистими роздумами. Заголовок статті повинен зазвичай є привабливим і відображає суть матеріалу. У блогах, присвячених складним темам, статті можуть містити графіки, таблиці або ілюстрації для наочності.

5. Сайдбар або бокова панель — це додатковий елемент інтерфейсу, що розташовується збоку від основного контенту і містить різну додаткову інформацію. Бокова панель може включати віджети, категорії, популярні публікації чи кнопку підписки на оновлення блогу. Сайдбар особливо важливий для поліпшення зручності навігації.

6. Інтерактивні елементи - сучасні блоги активно використовують інтерактивні елементи, такі як розділ коментарів, де читачі можуть залишати відгуки, обговорювати публікації та ділитися своїми думками. Коментарі сприяють створенню спільноти навколо блогу та забезпечують зворотній зв’язок для автора. Окрім коментарів, інтерактивність також може досягатися через кнопки "Поділитися" в соціальних мережах, що дозволяє читачам поширювати статті, а також форми для підписки на оновлення для отримання повідомлень про нові публікації.

7. Реклама є важливою складовою більшості блогів, які прагнуть монетизувати свій контент, вони можуть розташовуватися в різних частинах блогу, наприклад, у заголовку, боковій панелі або між абзацами у статтях. Блоги зазвичай співпрацюють із рекламними мережами, такими як Google AdSense.

8. Сторінка "Про нас" - в більшості випадків містить інформацію про автора, його професійний досвід, інтереси та мету створення блогу.

9. Архіви зазвичай відображають статті, організовані за датами, що

дозволяє читачам переглядати більш ранні матеріали. Категорії ж групують контент за темами, полегшуючи пошук статей на конкретні теми.

10. Підписка на оновлення - багато блогів пропонують читачам підписатися на оновлення через електронну пошту. Це дозволяє постійним читачам отримувати повідомлення про нові публікації і залишатися в курсі оновлень контенту.

1.4. Основні проблеми монолітних веб-блогів без асинхронної обробки запитів

Монолітна архітектура для веб-блогів, яка не включає асинхронну обробку запитів, має обмеження в ефективності, масштабованості та стійкості до високих навантажень. Використання веб-фреймворків без додавання асинхронних інструментів, таких як Celery і RabbitMQ, створює специфічні проблеми для обробки запитів у реальному часі, особливо в умовах зростаючої аудиторії. Нижче наведені основні проблеми, які часто виникають у монолітних веб-блогах, побудованих без асинхронної обробки запитів, а також приклади їх впливу на функціональність.

- Монолітні веб-блоги без асинхронної обробки запитів зазвичай використовують синхронну обробку HTTP-запитів, що означає, що кожен запит виконується послідовно, і сервіс не може обробляти новий запит до завершення поточного. Якщо аудиторія блогу зростає, зростає і кількість одночасних запитів, що може призвести до блокування та тривалого часу відповіді для користувачів.
- У монолітній архітектурі без асинхронних інструментів на зразок Celery та RabbitMQ довготривалі завдання, такі як обробка зображень, відправка електронних листів або перевірка контенту на наявність шкідливого коду, блокують основний процес. Це призводить до значних затримок для користувачів, оскільки такі операції виконуються в рамках основного запиту.
- Монолітна архітектура не оптимізована для виконання ресурсомістких завдань, таких як аналітика, синхронізація даних або резервне копіювання,

оскільки всі процеси виконуються в межах одного сервера, і це може призвести до перевантаження та навіть краху сервера під час пікових навантажень.

- Монолітна архітектура обмежена в можливостях горизонтального масштабування. Щоб обробляти більше запитів, часто необхідно збільшувати апаратні ресурси сервера, на якому розгорнуто додаток. Це менш ефективно, ніж горизонтальне масштабування, яке стає можливим із впровадженням асинхронних процесів.

Реальні приклади таких проблем:

- Hacker News(рис. 1.7) — популярний серед розробників сайт, спершу створили як простий моноліт, де всі дії користувачів виконувалися послідовно. Це інколи спричиняло затримки. Коли популярність ресурсу почала швидко зростати, сервери періодично не справлялися з навантаженням, особливо в години пікового трафіку. Щоб вирішити цю проблему, додали асинхронну обробку для перевірки спаму та оновлення рейтингів, використовуючи Celery.

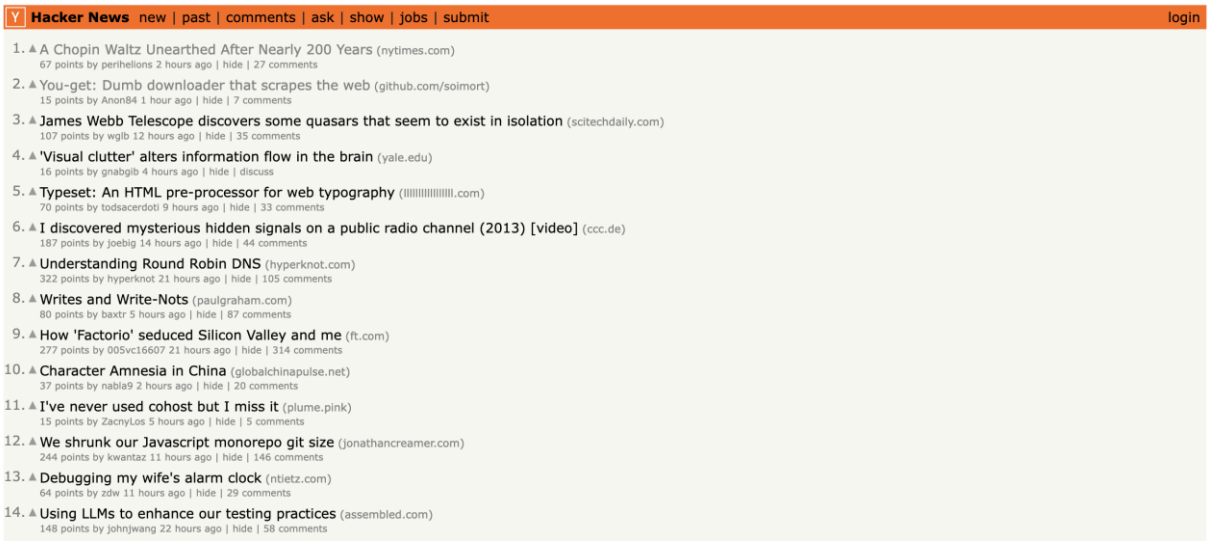

Рис. 1.7. Домашня сторінка Hacker News

- TechCrunch також стикався із труднощами, коли резонансні статті чи ексклюзивні новини притягували величезну кількість читачів одночасно. Сервери не завжди могли впоратися з таким напливом відвідувачів, особливо в

моменти серйозних навантажень.

- Unsplash Blog, який створений для фотографів і публікує статті про фотографію та обробку зображень(рис 1.8), мав схожі проблеми. Серйозні навантаження на сервер виникали під час завантаження фотографій, оскільки це потребувало значної обробки(кожне фото треба було підготувати у кількох розмірах, що займало ресурси і блокувало доступ іншим користувачам). Через монолітну структуру сайту без асинхронної обробки завантаження зображень сповільнювали інші процеси. Для вирішення цієї проблеми Unsplash почав використовувати Celery - це дозволило розподілити навантаження на сервери та вивільнити ресурси для інших запитів. Тепер користувач може завантажити зображення, а сайт опрацьовує його у фоновому режимі, забезпечуючи плавність взаємодії.

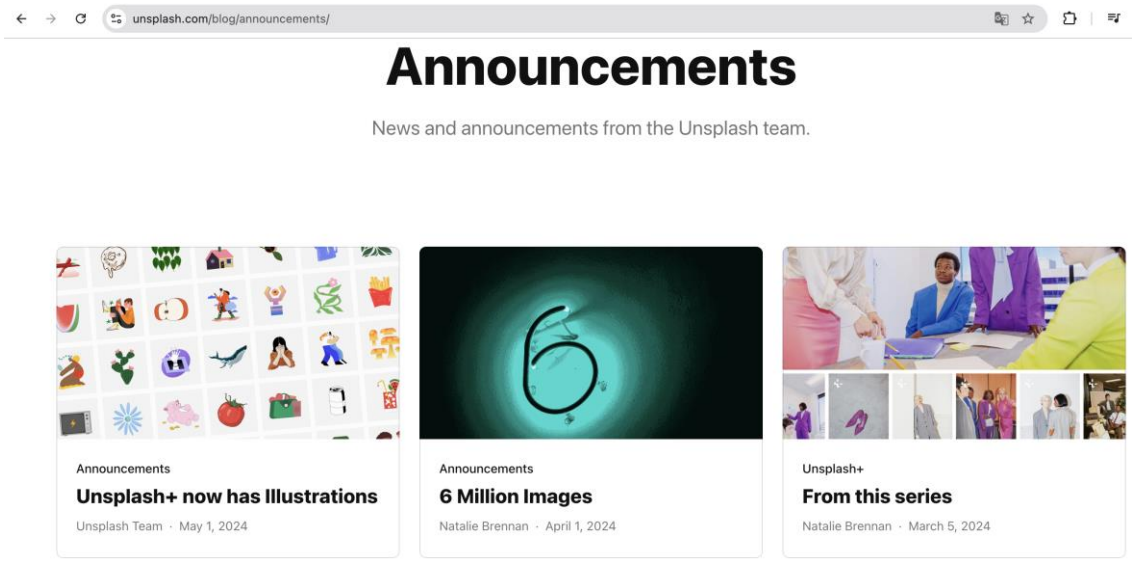

Рис. 1.8. Одна із сторінок Unsplash Blog

- Medium, платформа для текстових блогів і наукових статей, також зіткнулася з необхідністю обробляти велику кількість запитів на коментарі, реакції та рейтинг статей. Коли система стикалася з високим навантаженням, її продуктивність різко знижувалася, адже всі операції виконувались у головному потоці. Приплив нових користувачів і активне коментування під популярними

статтями лише погіршували ситуацію. Затримки виникали особливо часто під час взаємодій користувачів, таких як лайки, коментарі чи оновлення рейтингу. Щоб вирішити цю проблему, Medium перейшов на асинхронну архітектуру, і завдяки використанню Celery і RabbitMQ, всі завдання тепер обробляються у фоновому режимі. Це означає, що користувачі можуть залишати коментарі й ставити лайки без необхідності чекати, поки операція завершиться.

1.5. Методи вирішення таких проблем

Веб-додатки, які не підтримують асинхронне опрацювання запитів, часто стикаються з труднощами під час обслуговування великої кількості користувачів. Особливо помітні такі затримки на популярних платформах, де взаємодія включає коментарі, оцінки та оновлення рейтингів. Такі проблеми можуть негативно вплинути на досвід уже активних користувачів і знизити привабливість платформи для нових.

Для забезпечення швидкої та безперебійної роботи необхідно впроваджувати технології, які дозволяють ефективно обробляти запити навіть під великим навантаженням. Одним із рішень є оптимізація роботи додатку шляхом розподілу завдань. Завдання, які не потребують миттєвої обробки, можна виконувати у фоновому режимі. Наприклад, це стосується оновлення рейтингів, розсилки сповіщень, обробки зображень, перевірки на спам чи генерації рекомендацій. Такий підхід дозволяє зменшити навантаження на основний потік і забезпечити плавну взаємодію з користувачами.

Чим менший час потрібен сторінці для повернення контролю користувачеві, тим зручніше з ним працювати і тим легше він буде витримувати велику кількість одночасних користувачів/операцій.

Окремий сервіс для фонової обробки задач дозволяє значно зменшити навантаження на основний сервер, адже будь-які важкі обчислення та запити до бази даних можна відкласти на певний час і виконувати незалежно.

Ключовим у таких рішеннях є використання розподіленої архітектури с кількома сервісами(які мають свої потужності, бази даних і т.д.), яка дозволяє обробляти складні завдання асинхронно.

Асинхронна система обробки запитів дозволяє рівномірно розподіляти завдання між серверами, забезпечуючи їхню швидку обробку без затримок. Завдяки тому, що кожне завдання виконується в окремому процесі, це не впливає на загальну продуктивність системи. Під час пікових періодів, коли трафік значно зростає, асинхронний підхід дозволяє платформі працювати ефективно: основний сервер залишається вільним для критично важливих операцій, що запобігає зниженню швидкодії.

Окрім технічних переваг, цей підхід є також економічно вигідним. Платформу можна масштабувати поступово, не інвестуючи одразу в потужне обладнання для роботи в пікові моменти. Такий гнучкий розподіл ресурсів дозволяє підтримувати значну кількість користувачів, забезпечуючи стабільну роботу й швидкий відгук системи.

Асинхронні процеси відкривають нові можливості для розвитку функціоналу платформи. Наприклад, у фоновому режимі можна реалізувати логіку рекомендацій, з аналізом дій та уподобань користувачів, або проводити моніторинг коментарів для швидкого реагування на конфлікти. Це дозволяє адміністраторам оперативно втручатися, не перевантажуючи основний сервер. Завдяки розподілу завдань платформи можуть виконувати навіть складну аналітику без шкоди для швидкодії і без помітних змін в швидкодії для користувачів. Основний сервер при цьому зосереджується на ключових запитах, забезпечуючи стабільну взаємодію з ними. У сучасних умовах це є доволі серйозним аргументом для платформ, які прагнуть залишатися ефективними, популярними та привабливими для своєї аудиторії.

1.6. Постановка завдання

У ході вибору теми для дипломного проєкту було обрано реалізацію сучасного веб-додатку «Блог», який зможе обробляти запити користувачів

асинхронно, забезпечуючи стабільну продуктивність, хорошу швидкодію та розширені функції. Основна мета — спроектувати мікросервісний додаток, який дозволить забезпечити масштабованість та надійність завдяки асинхронному підходу до виконання завдань. Основні завдання, які необхідно виконати в процесі розробки:

- обрати та налаштувати технології для розподілу завдань між мікросервісами та ефективної асинхронної обробки, зокрема для керування повідомленнями між сервісами та виконанням фонових задач;
- розробити архітектуру та інтерфейс користувача для блогу, з можливістю створення і редагування статей, категоризації, реєстрації та адміністрування;
- забезпечити інтеграцію технологій для асинхронної обробки, таких як генерація щотижневих звітів про активність, автоматичне створення резервних копій і відправка повідомлень на електронну пошту при публікації нових постів;
- забезпечити ефективне кешування для прискорення відгуку користувачам під час роботи з контентом блогу;
- протестувати роботу веб-додатку для перевірки надійності виконання фонових завдань, коректності взаємодії мікросервісів та продуктивності під навантаженням.

Проєкт також спрямований на інтеграцію системи повідомлень між окремими компонентами додатка, що забезпечить безперебійність обробки запитів на відправку електронних листів.

1.7. Висновок до розділу 1

У першому розділі розглянуто предметну область, зокрема історію розвитку текстових веб-блогів, їх роль у становленні Інтернету, а також сучасний стан блогосфери. Проаналізовано основні структурні складові блогів та виклики, з якими стикаються монолітні веб-блоги.

Розвиток текстових блогів у кінці 1990-х – на початку 2000-х років став важливим етапом у популяризації та глобалізації Інтернету, надаючи

користувачам змогу легко створювати та поширювати контент, обговорювати актуальні теми та впливати на суспільну думку. Поступово блоги почали відігравати значну роль у формуванні цифрової культури, створюючи майданчики для особистого самовираження, обміну інформацією та налагодження онлайн-спільнот. Цей феномен сприяв розширенню доступності інформації, активізувавши розвиток соціальних мереж та медіа.

Сучасний стан блогосфери характеризується різноманіттям платформ та інструментів для ведення блогів, що дозволяють користувачам вести блоги незалежно від технічного досвіду. Багато текстових блогів зберегли свою актуальність, доповнюючи інші формати контенту, такі як відео та соціальні мережі. Вони продовжують забезпечувати читачів аналітичними матеріалами.

Водночас монолітні веб-блоги без асинхронної обробки запитів стикаються з рядом обмежень, зокрема з труднощами в обробці великої кількості запитів та ресурсомісткістю при виконанні таких операцій, як надсилання сповіщень або генерування звітів. Як наслідок, це може знижувати продуктивність - подібні виклики спостерігалися в низці відомих платформ, які на початку свого шляху використовували монолітну архітектуру.

Отже, розглянуті аспекти демонструють актуальність використання сучасних підходів до розробки веб-додатків для підвищення ефективності їх роботи, що створює підґрунтя для обґрунтування вибору технологій у наступних розділах.

РОЗДІЛ 2
ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ «БЛОГ»

2.1. Вибір та обґрунтування інструментів проектування

2.1.1. Середовище розробки

Для реалізації даного проєкту було обрано середовище розробки PyCharm Professional, яке надає потужний набір інструментів для ефективної роботи з Python. Це інтегроване середовище розробки (IDE), спеціально оптимізоване для створення веб-додатків на базі фреймворків, таких як Django, що робить його ідеальним для розробки сучасного блогу з мікросервісною архітектурою та асинхронною обробкою запитів.

PyCharm Professional містить вбудовані інструменти, які суттєво спрощують процес написання, тестування та налагодження коду. Однією з ключових переваг є повноцінна підтримка Django, що включає автоматичне заповнення коду, підсвітку синтаксису, зручну навігацію по проєкту та готові шаблони для структурування додатку. Це значно пришвидшує розробку складних компонентів веб-додатка, таких як інтеграція Celery для асинхронних задач і RabbitMQ для обміну повідомленнями між сервісами.

Але окрім цього є й безліч додаткових інструментів та бібліотек, які можна встановити для зручної роботи, наприклад, з CSS, JavaScript, Docker чи більшістю інших популярних технологій або мов програмування. Це відкриває безліч можливостей і робить процес розробки значно швидшим та зручнішим, без необхідності використовувати якісь окремі програми для цього.

Ну і не будемо забувати про бази даних - наявність інтегрованого інструменту для роботи з ними спрощує процес керування/оптимізації запитів.

Вигляд середовища розробки PyCharm Professional можна побачити на рис. 2.1.

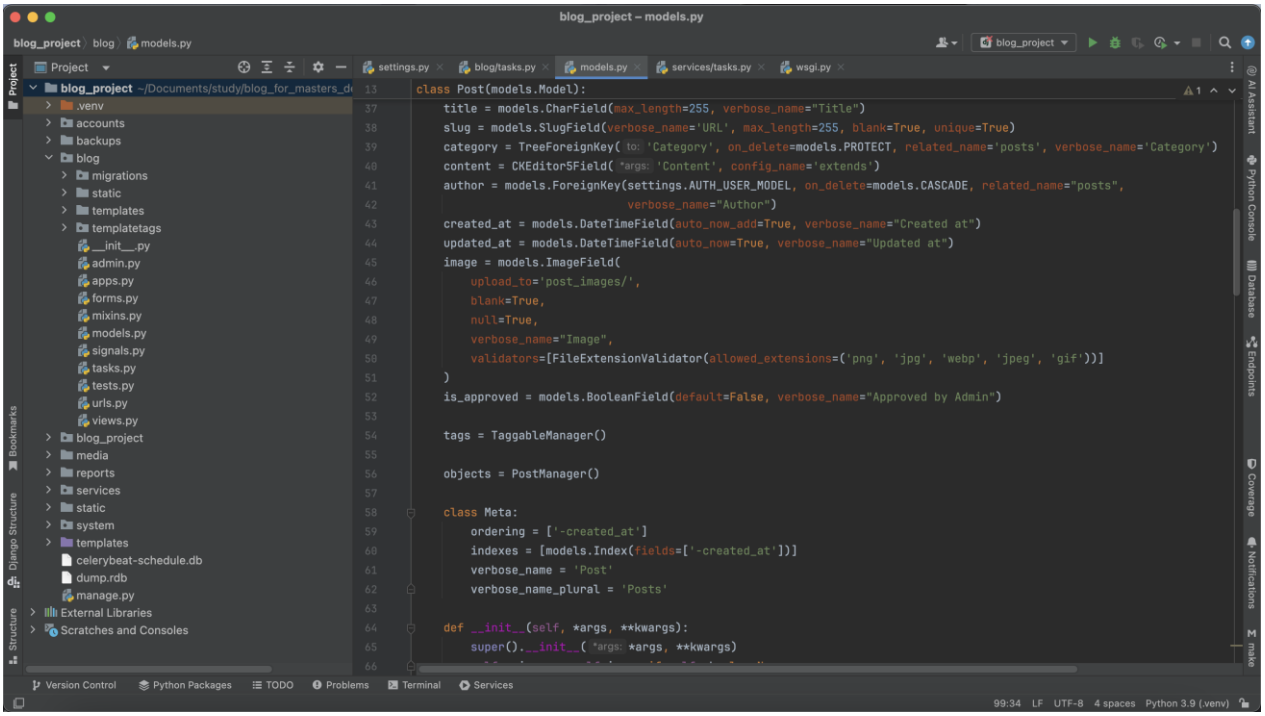

Рис. 2.1. Середовище розробки PyCharm Professional

2.1.2 Мова програмування Python

Для реалізації дипломного проєкту було обрано мову програмування Python. Цей вибір зумовлений рядом переваг, які роблять Python ідеальним для розробки сучасних веб-додатків, зокрема блогів із складною логікою, асинхронною обробкою запитів та інтеграцією мікросервісів.

По-перше, Python є високорівневою мовою, що дозволяє зосередитися на функціональності додатка, не витрачаючи зайвий час на низькорівневі деталі.

З його простим синтаксисом та багатою бібліотекою стандартних модулів Python значно пришвидшує процес написання коду. Це важливо для проєкту, який охоплює розробку блог-системи, адміністрування, управління контентом, а також функції інтеграції з іншими сервісами для відправлення повідомлень та виконання фонових завдань.

Python забезпечує інтеграцію з великою кількістю бібліотек та

фреймворків, що полегшують роботу з веб-розробкою та обробкою запитів. Серед них – Django, один із найпопулярніших фреймворків для розробки веб-додатків, який оптимізує структуру програми, роблячи її стійкою, масштабованою та зручною для підтримки. Використання Django дозволяє з легкістю створювати динамічні веб-сторінки та надає всі необхідні засоби для швидкого побудови моделі, контролерів і представлень, що є важливим у створенні інтерфейсу блогу, системи реєстрації користувачів та управління контентом[3].

Крім того, для обробки фонових задач у проєкті застосовуються інструменти, що працюють у зв'язці з Python, такі як Celery, який використовується для асинхронної обробки завдань на бекенді. Це дозволяє виконувати трудомісткі процеси у фоновому режимі, не блокуючи основну функціональність веб-сайту. Наприклад, відправка електронних листів або генерація звітів можуть виконуватись асинхронно, що підвищує зручність користування додатком та оптимізує використання ресурсів сервера.

Python також володіє потужними засобами для обробки даних та баз даних. Використання бібліотек для роботи з реляційними базами даних, таких як PostgreSQL, дозволяє зберігати та обробляти великі обсяги даних, забезпечуючи оптимальну продуктивність додатку. У поєднанні з ORM (Object-Relational Mapping) фреймворку Django, цей підхід дозволяє розробникам працювати з базою даних на рівні Python-коду, зменшуючи час на написання SQL-запитів та зменшуючи ризик помилок при роботі з даними.

Приклад простої функції для виводу кількості голосних букв в слові, яка написана на Python, наведено на рис. 2.2.

```python
def count_vowels(text):
    vowels = "aeiou"
    count = 0
    for char in text.lower():
        if char in vowels:
            count += 1
    return count


print(count_vowels("Hello World"))  # Output: 3
```

Рис. 2.2. Приклад коду Python

2.1.3 Мова програмування JavaScript

Конкретно в даному проєкті JavaScript активно використовується для роботи з API та асинхронного обміну даними через технологію AJAX, що дозволяє оновлювати вміст сторінок без повного перезавантаження. Це особливо корисно для таких функцій, як завантаження нових коментарів, пошук статей або оновлення переліку категорій - опис їх реалізації буде в наступному розділу.

Використання асинхронних запитів покращує ефективність роботи сайту, знижує навантаження на сервер та робить користувацький інтерфейс більш швидким і приємним у користуванні. З точки зору сучасної фронтенд-розробки, JavaScript має велику екосистему бібліотек і фреймворків, таких як jQuery, Vue.js, React, що значно розширюють можливості взаємодії з веб-сторінкою та спрощують процес розробки інтерфейсу[4].

У даному проєкті було обрано базовий підхід із використанням звичайного JavaScript, що дозволило уникнути зайвих залежностей та зберігати високу швидкість завантаження сторінок, однак за потреби, звичайно, можлива інтеграція додаткових бібліотек.

Вигляд, який має одна з функцій, яка написана з використанням JavaScript, наведено на рис. 2.3.

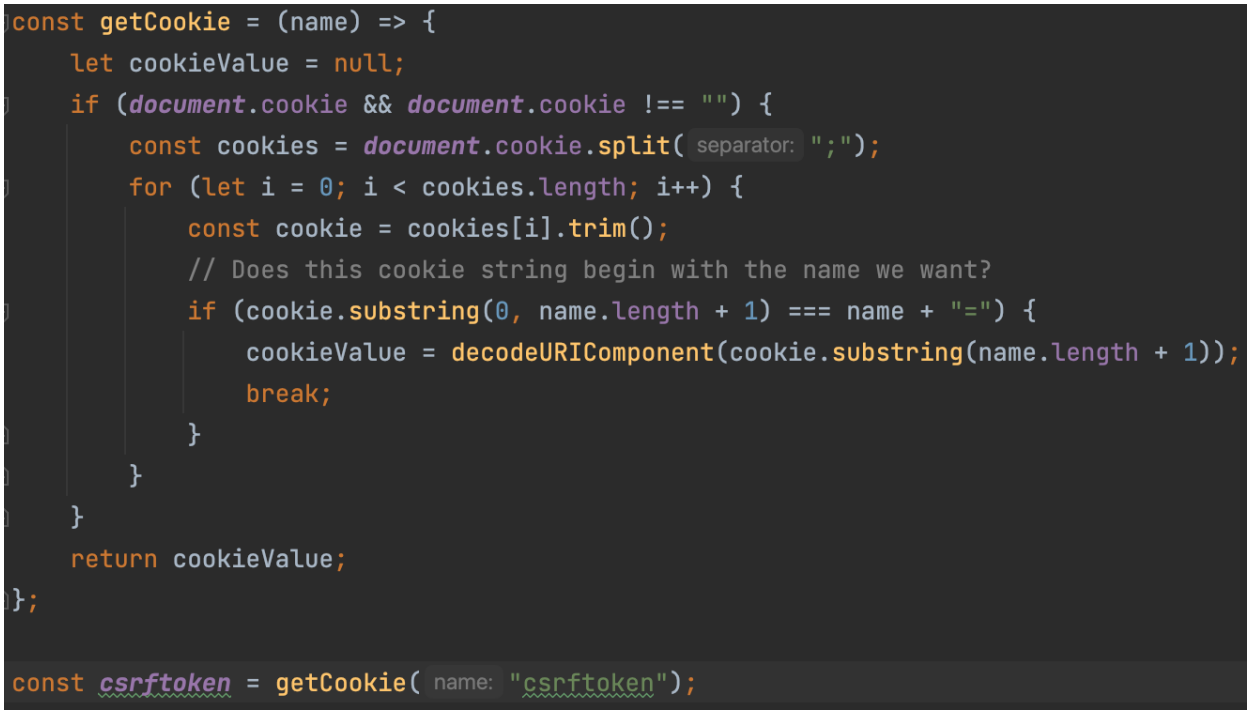

Рис. 2.3. Приклад синтаксису JavaScript

2.1.4 Мова гіпертекстової розмітки HTML

HTML (HyperText Markup Language) є основним стандартом для створення та структурування веб-сторінок, що робить його незамінним у будь-якому проєкті, пов'язаному з розробкою веб-додатків. У даному проєкті HTML використовується як головний засіб для визначення структури веб-блогу, включаючи статті, навігаційні меню, форму реєстрації користувачів, коментарі, категорії та інші основні елементи інтерфейсу. HTML забезпечує логічну організацію елементів сторінки, що полегшує їх подальше стилізування та оживлення за допомогою CSS і JavaScript. Однією з основних причин вибору HTML є його універсальність і підтримка всіма сучасними браузерами та пристроями. У структурі HTML-документа було визначено основні блоки, такі як заголовки, абзаци, списки, таблиці, форми та посилання, що відповідають вимогам до зручної навігації та інтуїтивного використання. У блозі HTML використовується для створення та структурування таких розділів, як сторінка

окремої статті, головна сторінка з переліком останніх дописів, сторінка профілю користувача та панель адміністратора, де відбувається управління статтями, коментарями та категоріями.

HTML також має критичне значення для доступності (accessibility), що є важливим аспектом сучасних веб-додатків. Наприклад, за допомогою спеціальних атрибутів (alt-тексти для зображень, aria-label для кнопок та інших елементів) HTML забезпечує доступ до контенту користувачам із порушеннями зору, що підвищує загальну доступність блогу для різних категорій відвідувачів. Крім того, логічна структура HTML сприяє оптимізації пошукових систем (SEO), що дозволяє блогу краще індексуватися пошуковими системами та отримувати більше відвідувачів із пошукових запитів.

HTML також забезпечує легке використання CSS для стилізації елементів сторінки. У даному проєкті HTML-коди сторінок були побудовані з урахуванням принципів семантичного HTML, де кожен елемент має логічну ієрархію та чітке значення (наприклад, використання <header>, <footer>, <nav>, <main> тощо), що дозволяє полегшити розуміння структури як розробникам, так і браузерам та іншим програмам, які взаємодіють із сайтом.

Також важливо відзначити, що HTML використовується для створення адаптивного макету сторінок. За допомогою різних контейнерів та елементів HTML структура блогу підтримує адаптивність, що дозволяє йому відображатися коректно на екранах різних розмірів — від мобільних пристроїв до великих моніторів. Це допомагає забезпечити зручність використання для всіх користувачів, незалежно від того, який пристрій вони використовують для доступу до блогу.

Вигляд частини сторінки профіля користувача, яка написана на HTML, можна побачити на рис. 2.4.

```
<div class="col-lg-7">
  <div class="card">
    <div class="card-body">
      <div class="row mb-3">
        <div class="col-sm-3">
          <h6 class="mb-0">Email</h6>
        </div>
        <div class="col-sm-9 text-secondary">
          <input id="email" type="text" class="form-control" value="{{ form.instance.email }}" readonly>
        </div>
      </div>
      <div class="row mb-3">
        <div class="col-sm-3">
          <h6 class="mb-0">First Name</h6>
        </div>
        <div class="col-sm-9 text-secondary">
          <input id="first_name" type="text" class="form-control" value="{{ form.instance.first_name }}">
        </div>
      </div>
      <div class="row mb-3">
        <div class="col-sm-3">
          <h6 class="mb-0">Last Name</h6>
        </div>
        <div class="col-sm-9 text-secondary">
          <input id="last_name" type="text" class="form-control" value="{{ form.instance.last_name }}">
        </div>
      </div>
```

Рис. 2.4. Приклад синтаксису HTML

2.1.5 Мова каскадних таблиць стилів CSS

CSS (Cascading Style Sheets) є ключовим інструментом для візуального оформлення веб-сторінок, що забезпечує привабливий, естетичний вигляд та зручність інтерфейсу блогу. У даному проєкті CSS використовується для стилізації структури, створеної за допомогою HTML, надаючи сторінкам чіткий і сучасний дизайн, що відповідає потребам користувачів. Використання CSS дозволяє відокремити візуальне представлення від структурних елементів, що полегшує подальше підтримання і розширення веб-додатка.

Однією з головних причин вибору CSS є його здатність легко адаптувати дизайн до різних розмірів екрану та пристроїв за допомогою медіа-запитів.

У блозі це означає, що сторінки зручно відображаються на мобільних телефонах, планшетах та комп’ютерах. Наприклад, для блогу було створено гнучку сітку, де статті, бічні панелі та інші елементи автоматично

підлаштовуються під розмір екрану.

CSS також допомагає у створенні чіткого і логічного макету. Використовуючи CSS Flexbox та CSS Grid, можна легко структурувати розташування різних елементів сторінки, таких як заголовки, меню навігації, контентні блоки та панель адміністратора. Це забезпечує послідовний і гармонійний вигляд усіх сторінок блогу, роблячи сайт візуально привабливим та функціональним. Наприклад, панель навігації розміщена таким чином, щоб забезпечити інтуїтивний доступ до різних категорій блогу, а кнопки, форми та інші інтерактивні елементи отримали стиль, що сприяє їх легкому знаходженню і використанню.

Завдяки CSS можливо також досягти динамічності та інтерактивності інтерфейсу без необхідності використання JavaScript. Наприклад, за допомогою псевдокласів CSS, таких як :hover та :focus, створені візуальні ефекти для кнопок і посилань, які роблять взаємодію з елементами більш приємною і зрозумілою для користувача. Це включає зміну кольору, тіні чи анімації, що підкреслюють обраний елемент і підвищують зручність взаємодії з контентом блогу.

Ще однією перевагою CSS є його здатність підвищувати загальну продуктивність і швидкість завантаження сайту. Завдяки збереженню стилів у зовнішньому файлі CSS, браузери можуть кешувати його, зменшуючи обсяг даних, які потрібно завантажити повторно під час переходів між сторінками. Це особливо важливо для блогів, де користувачі можуть переглядати велику кількість сторінок, переходячи від однієї статті до іншої. Додатково, оптимізація стилів та використання мінімалістичного підходу зменшують загальний обсяг CSS-файлу.

Окрім функціональних стилів, CSS сприяє створенню корпоративної айдентики веб-додатка. У поточному проєкті колірна схема, шрифти та іконки відображають унікальний стиль блогу, роблячи його впізнаваним серед інших. Основна кольорова палітра створює відповідну атмосферу на сайті, полегшує читання статей і допомагає фокусувати увагу на ключових елементах. Вибір

відповідних шрифтів і використання CSS для налаштування інтерліньяжу, відступів та розмірів тексту також робить контент приємним для читання.

У підсумку, CSS є невід'ємною частиною дизайну блогу, яка дозволяє створити адаптивний, привабливий та оптимізований інтерфейс, що забезпечує позитивний користувацький досвід. Завдяки CSS блог виглядає сучасно, доступно і привабливо, що сприяє залученню та утриманню користувачів, забезпечуючи їм комфорт під час взаємодії з контентом.

На рис. 2.5 можна побачити частину стилів, які були додані для категорій в цьому проєкті.

```css
/* Categories */
.category-list {
    list-style: none;
    padding: 0;
    margin: 0;
}

.category-item {
    padding: 0.5rem 0.75rem;
    border-bottom: 1px solid #f1f1f1;
}

.category-item a {
    color: #333;
    text-decoration: none;
}

.sub-category-list {
    list-style: none;
    padding-left: 1.5rem;
    margin: 0;
}
```

Рис. 2.5. Приклад синтаксису CSS

2.1.6 Веб фреймворк Django

Django є популярним фреймворком для розробки веб-додатків на Python, і він чудово підходить для реалізації функціональності блогу з асинхронною обробкою запитів завдяки своїм особливостям, зокрема вбудованим інструментам, багатофункціональній архітектурі, простоті інтеграції та високій продуктивності.

Однією з головних причин вибору Django є його архітектура «все в одному». Django надає потужний ORM (Object-Relational Mapping) для роботи з базами даних, систему маршрутизації для створення URL-структури, підтримку шаблонів, потужну панель адміністратора та засоби для управління користувачами і групами. Це особливо важливо для блогу, де необхідні авторизація користувачів, адміністрування контенту, управління категоріями, статтями та коментарями. Вбудована адміністративна панель дозволяє легко керувати всіма об'єктами контенту, а також забезпечує інтеграцію з системами автентифікації користувачів[5].

Ще однією значною перевагою Django є його модульність і можливість гнучкої інтеграції з іншими технологіями, що робить його ідеальним для асинхронних додатків. Використання Django спрощує взаємодію з системами для чергування завдань, необхідних для асинхронної обробки, наприклад, для періодичних завдань (на кшталт генерації звітів) і відправки email-сповіщень. Django без проблем інтегрується з фреймворками для обробки завдань, що дозволяє делегувати трудомісткі операції, зберігаючи продуктивність основного додатка. Зокрема, завдяки підтримці асинхронних функцій Django може направляти ресурсомісткі запити в чергу для асинхронної обробки, звільняючи сервер від затримок і дозволяючи користувачам отримувати швидку відповідь.

Django також чудово підходить для роботи з REST API, що є важливим для багатьох сучасних веб-додатків.

Це дозволяє блогу взаємодіяти з іншими сервісами та додатками, обмінюватися даними або забезпечувати інтеграцію з іншими платформами, такими як мобільні додатки або системи аналітики. Django REST Framework, який легко інтегрується з фреймворком, надає можливості для створення API-інтерфейсів.

Важливим фактором є і те, що Django має великий набір вбудованих інструментів для захисту додатків. Фреймворк містить засоби захисту від найпоширеніших веб-атак, таких як XSS, CSRF, SQL-ін'єкції. Для блогу це критично, оскільки захист користувацьких даних та даних самого додатку має

пріоритетне значення, особливо при роботі з асинхронними задачами, де зростає ризик навантаження на сервер.

Завдяки підтримці масштабування Django дозволяє легко збільшити можливості додатка, що важливо для розвитку блогу. З часом трафік на сайті може зрости, і гнучкість масштабування дозволяє адаптувати додаток до нових потреб. Окрім цього, у Django доступна розвинена система кешування, яка прискорює роботу додатку за рахунок збереження результатів обчислень та повторного використання готових відповідей.

Додатковою перевагою Django є велика спільнота користувачів і багата документація, що значно полегшує процес розробки, налагодження і підтримки проєкту. Сотні готових плагінів і розширень для Django дозволяють розширювати функціональність додатку без необхідності реалізовувати всі компоненти з нуля, що економить час та ресурси. Це також означає, що будь-які нові функції, наприклад, автоматизацію резервного копіювання або надсилання сповіщень, можна легко додати, використовуючи існуючі бібліотеки.

У підсумку, Django забезпечує надійний фундамент для побудови динамічного блогу, здатного задовольняти потреби як розробників, так і кінцевих користувачів.

2.1.7 Celery

Celery – це інструмент, який виконує завдання в асинхронному режимі в Python-додатках, тобто з використанням даного інструменту, додаток виступає в ролі посередника, забезпечуючи паралельне виконання операцій, не забираючи при цьому цінні ресурси основного потоку виконання, і не змушуючи користувача чекати на завершення процесів.

Вперше її було випущено ще в 2009 року, і відтоді вона стала одним із найпопулярніших інструментів для асинхронного опрацювання завдань у Python. Використовується в широкому спектрі проєктів, від невеликих веб-додатків(яким є даний проєкт) до великих корпоративних систем. Важливим моментом в історії цієї програми є її здатність ефективно впроваджуватися в різноманітні проекти,

надаючи програмістам можливість зосередитися на розв’язанні конкретних завдань, а не на організації асинхронного виконання — наприклад, в Django налаштування та підключення Celery заключається в додаванні конфігурації всього в два файли(celery.py і settings.py), що займає кілька хвилин, після чого уже можна користуватися всіма можливостями, які надають асинхронні задачі[6].

2.1.8 RabbitMQ

RabbitMQ — це інструмент для обміну повідомленнями між різними частинами програми або навіть між різними програмами. Найчастіше його порівнюють із поштовим кур’єром - адже одна частина(додаток-блог) системи пише "лист" (повідомлення), відправляє його через RabbitMQ, а інша частина(додаток-надсилач листів в поточному випадку) отримує і обробляє його[7].

RabbitMQ допомагає цим мікросервісам "спілкуватися" один з одним, не перевантажуючи основний функціонал і працює за принципом "відправник-черга-отримувач". Це значить, що відправник передає повідомлення в чергу, а отримувач забирає ці повідомлення і обробляє їх. І завдяки RabbitMQ можна виконувати задачі асинхронно.

Знову ж таки, конкретно в цьому проєкті він використовується для відправки email у фоновому режимі, не змушуючи користувача чекати - це робить систему швидшою, надійнішою і більш гнучкою.

2.1.9 Redis

Redis — це інструмент для швидкого зберігання та отримання даних, який працює прямо в оперативній пам'яті, а не на диску, як традиційні бази даних. Для спрощення можна сказати, що це як супер-швидкий блокнот, у якому програма може записувати або читати дані миттєво. Дуже часто його використовують для зберігання сесій користувачів(даних про їхній вхід) в веб-додатках - тут Redis ідеально підходить для цього, бо він може зберігати дані в пам'яті та дуже швидко до них звертатися. Інша доволі стандартна сфера його застосування –

кешування[8].

Саме для цієї цілі він був доданий до цього проєкту — щоб не слати сотні запитів щохвилини для отримання категорій чи перевірки на те, чи користувач онлайн, можна просто отримати дані з пам’яті, адже це не той тип даних, який часто змінюється або надто важливий. І замість того, щоб щоразу робити запити до бази даних, ці дані зберігаються у Redis і просто "витягуються" звідти. Можливо це не дуже помітно при локальному тестуванні, але в реальних умовах, це має величезний вплив на швидкість завантаження сторінок, навантаження на базу даних та заощадження ресурсів сервера. Redis підтримує різні типи даних: рядки, списки, множини, хеші тощо. Наприклад, ти можна зберігати просте значення (чи користувач в мережі, чи ні), список елементів (категорі), або ж, якщо потрібно, структуру з ключами та значеннями. Ще одна перевага Redis — він може не лише зберігати дані, а й організовувати обмін ними між різними частинами системи. В даному проєкті даний функціонал не використовується, але можливість його додати є.

2.2. Опис файлової структури веб-блогу

Оскільки даний проєкт написаний з використанням веб-фреймворку Django, він має чітко визначену структуру, що забезпечує легкість підтримки та масштабування, адже кожна функціональність розбивається на окремі “додатки” (apps), які розташовуються в окремих папках. Кожен додаток містить свої файли, такі як urls.py для маршрутизації, models.py для опису таблиць бази даних, views.py для логіки обробки запитів і відповіді, а також файли для міграцій, що відповідають за зміни в базі даних.

Коренева директорія кожного з мікросервісів містить конфігураційні файли, зокрема settings.py, де налаштовуються параметри бази даних, середовища виконання, підключення сторонніх додатків і middleware. Окрім них у кореневих директоріях знаходяться файли urls.py для глобальної маршрутизації, wsgi.py та asgi.py для налаштування серверу та файл manage.py, що слугує інструментом командного рядка для управління проєктом(через нього

ми будемо запускати обидва мікросервіси).

У моєму випадку структура проєкту також включає додаткові компоненти, такі як папка для статичних файлів (static) і шаблонів (templates) — саме там зберігаються CSS, JavaScript і зображення, тоді як шаблони відповідають за рендеринг HTML-сторінок.

2.2.1. blog_project/ – коренева директорія Django-проєкту

- __init__.py – ініціалізаційний файл, що дозволяє Python розпізнавати папку як модуль.
- asgi.py – файл для налаштування ASGI-сервера.
- celery.py – файл для конфігурації Celery, якщо проєкт використовує його для управління фоновими задачами.
- settings.py – головний файл налаштувань, де зберігаються всі конфігурації проєкту (наприклад, бази даних, додатки, середовища).
- urls.py – основний файл маршрутизації
- wsgi.py – файл для налаштування WSGI-сервера, використовується для обробки запитів у продакшн-середовищі.

2.2.2. accounts/ – додаток для управління користувачами

- migrations/ – папка з міграціями, які застосовуються до бази даних.
- templates/ – шаблони, пов'язані з користувацькими сторінками (наприклад, сторінки входу, реєстрації, особиста сторінка користувача).
- admin.py – налаштування адміністративної панелі для моделі користувачів
- apps.py – конфігураційний файл для додатка accounts.
- forms.py – файли для створення форм Django для різних сторінок (форма входу, реєстрації, зміни даних користувача).
- middleware.py – середовищний файл, де можуть бути додані середовища для обробки запитів та логіка модифікації запитів до моменту, коли

будуть “перехвачені” логікою в views.py.

- models.py – визначення моделей користувачів.
- tests.py – тести для додатка accounts.
- urls.py – маршрутизація URL для додатка accounts.
- views.py – представлення для обробки логіки користувачів (наприклад, функції для входу, реєстрації, відображення та зміни даних користувача).

2.2.3. blog/ – основний додаток блогу

- migrations/ – міграції для блогу.
- static/ – статичні файли, пов'язані з блогом (стилі, зображення).
- templates/ – шаблони для відображення постів блогу.
- templatetags/ – кастомні теги для шаблонів блогу.
- forms.py – форми для створення і редагування постів блогу.
- models.py – визначення моделей для зберігання інформації про пости та дотичних до моделей постів(категорії, рейтинги, коментарі тощо).
- views.py – представлення для обробки запитів, що пов'язані з блогом (наприклад, перегляд постів, створення нового поста, редагування існуючого тощо).
- admin.py – налаштування адміністративної панелі для моделі постів та суміжних моделей.
- apps.py – конфігураційний файл для додатка blog.
- signals.py – сигнали, що автоматизують певні процеси (наприклад, надсилання повідомлень після створення поста).
- mixins.py – допоміжні класи для перевикористання логіки в різних представленнях.

2.2.4. system/ – додаток для логіки з фідбеками і сторінок-помилок.

- migrations/ – міграції для моделі фідбеку.
- templates/ – шаблони для відображення форми фідбеку.

- models.py – визначення моделі фідбеку, який може відправляти користувач.
- urls.py – маршрут для створення фідбеку
- views.py – представлення для створення фідбеку
- forms.py – форма створення фідбеку.
- admin.py – налаштування адміністративної панелі для моделі фідбеку.
- apps.py – конфігураційний файл для додатка system.
- tests.py – тести для додатка system.

2.2.5. services/ – додаток для спеціальних утиліт і команд

- management/ – папка, в якій містяться налаштування для команди, яка буде створювати бекапи з БД.
- __init__.py – ініціалізаційний файл, що дозволяє Python розпізнавати папку як модуль.
- forms.py – форми для створення і редагування постів блогу.
- models.py – визначення моделей для зберігання інформації про пости та дотичних до моделей постів(категорії, рейтинги, коментарі тощо).
- tasks.py – файл, в якому містяться Celery-завдання, які можуть або запускатися з певним інтервалом, або по потребі(в залежності від налаштування і типу завдання).
- utils.py – допоміжні методи і класи, які не прив’язані до певного додатку, і можуть використовуватися в декількох з них.
- weekly_report.py – файл, що містить в собі клас і логіку для генерації щотижневого репорту про активність користувачів в веб-блозі(запускається інтервально з допомогою Celery).

2.2.6. static/ – статичні файли, які використються на фронтенді

- bootstrap/ – папка, в якій містяться налаштування та файли, які потрібні для коректної роботи Bootstrap4, який значно полегшує.

- custom_css/ - папка, в якій знаходяться два файли — sidebar.css, який визначає стилі для бокової панелі, і styles.css, де усі стилі, які були прописані спеціально під елементи на сторінках блогу.
- custom_js/ - папка, в якій знаходиться JavaScript, який використовується в проєкті.
- images/ – папка із зображенням по-замовчуванню для постів і користувача.

2.2.7. reports/ – папка із щотижневими .pdf репортами

2.2.8. backups/ – папка із щоденними .json бекапами бази данних

2.2.9. media/ – папка із файлами, які загружають користувачі

- post_images/ – зображення, які були додані для постів.
- profile_images/ – аватари, які були додані для користувачів.
- uploads/ – файли, які були загружені із форми ckeditor 5.

2.2.10. templates/ – загальні html-template для рівня всього проєкту

- base.html – базовий шаблон для проєкту, структуру якого наслідують інші сторінки.
- header.html – шаблон для хедера сайту, який використовується в base.html і не дублюється на інших сторінках
- latest_comments.html — шаблон для підрозділу в сайдбарі, який показує кілька останніх коментарів, які залишали користувачі.
- messages.html — шаблон для повідомлень, які отримують користувачі від системи.
- pagination.html — шаблон для пагінації.
- sidebar.html — шаблон для бокової панелі.

2.2.11(емейл-мікросервіс). email_sender/ – основний додаток

- migrations/ — файли міграцій для управління структурою бази даних цього додатку;
- templates/ - містить шаблони для електронних листів у форматі HTML, які використовуються для формування тексту повідомлень. Там наявні три шаблони — емейл підтвердження посту, емейл-фідбек, емейл-повідомлення для підписників, якщо в певного автора вийшов пост;
- admin.py – налаштування відображення моделей цього додатку в Django Admin;
- apps.py — файл конфігурації додатку email_sender;
- models.py — містить модель-лог для зберігання інформації про відправлені листи.
- tasks.py — файл, з функціями, які запускаються асинхронно для відправки електронних листів через RabbitMQ. Celery використовує ці завдання для обробки черг на відправлення листів.

2.2.12 (емейл-мікросервіс). email_service/ – загальна конфігурація

- __init__.py — позначає директорію як Python-пакет.
- asgi.py — файл конфігурації для ASGI-сервера, необхідний для запуску асинхронних задач.
- celery.py — налаштування Celery для асинхронної обробки задач, таких як відправка листів. Цей файл містить конфігурацію RabbitMQ та інші параметри Celery.
- settings.py — основні налаштування Django-проекту, включаючи підключення до бази даних, конфігурацію RabbitMQ, налаштування для відправки електронної пошти та інші системні параметри.
- urls.py — конфігурація URL для додатку email_service, але містить лише мінімальні налаштування(лише роут для адмін-панелі), оскільки цей сервіс функціонує, як мікросервіс.
- wsgi.py — файл конфігурації для WSGI-сервера, використовується для

розгортання сервісу у продакшн-середовищі.

2.3. Проектування та розробка вебзастосунку

Розробка серверної частини веб-блогу — це не просто створення базової логіки. Вона охоплює цілий ряд завдань - потрібно також реалізувати механізми автентифікації та авторизації користувачів, налаштувати “спілкування” мікросервісів та запуск асинхронних задач.

Серверна частина, побудована на базі Django, охоплює кілька важливих функцій, таких як управління аутентифікацією користувачів, обробка публікацій, збереження даних і забезпечення безпеки. Завдяки архітектурі MVC (Model-View-Controller), яку підтримує Django, логіка кожної частини системи чітко розподілена між окремими модулями. Це дозволяє не тільки підвищити гнучкість, а й забезпечити масштабованість проєкту, що робить його зручним для подальшого розвитку та розширення.

2.3.1. Модуль(додаток) accounts

Додаток accounts відповідає за управління користувачами у веб-блозі і все, що з ними зв’язано — саме він реалізує функціонал реєстрації, аутентифікації та авторизації користувачів, а також забезпечує можливість оновлення профілю. У додатку є моделі для зберігання даних користувачів, форми, шаблони, маршрути та представлення.

- models.py

В даному файлі визначено класи та методи для кастомної моделі користувача BlogUser. Клас BlogUserManager є менеджером для цієї моделі та містить методи create_user та create_superuser. Метод create_user створює звичайного користувача, перевіряючи наявність обов’язкових полів email і nickname, а create_superuser створює адміністратора(Дод. А, рис. А.1.1).

Модель BlogUser, що успадковується від AbstractBaseUser та PermissionsMixin, містить основні поля для зберігання інформації про користувача(Дод. А, рис. А.1.2).

До них належать email та nickname, які є унікальними, а також profile_image для зображення профілю та інші текстові поля для імені та прізвища. Поля is_active та is_staff визначають статус користувача, а date_joined зберігає час реєстрації. Користувачі можуть вказати свої імена користувачів у соціальних мережах через відповідні поля. Поле followers встановлює зв’язок для відстеження підписок між користувачами.

Модель також має низку методів - save автоматично заповнює slug на основі nickname, якщо він не вказаний. Метод __str__ повертає nickname як текстове представлення об'єкта, а get_absolute_url генерує URL для профілю користувача. Додаткові методи follow, unfollow та is_following забезпечують функціональність підписок. Метод is_online визначає, чи є користувач активним в системі.

- views.py

У файлі views.py визначено функції для обробки запитів, пов’язаних з реєстрацією, аутентифікацією, управлінням профілем та підписками користувачів.

Функція register обробляє процес реєстрації, перевіряє дані форми BlogUserCreationForm, створює нового користувача, виконує його вхід у систему та перенаправляє на сторінку профілю з повідомленням про успішну реєстрацію(Дод. А, рис. А.1.3).

Функція login_view відображає форму входу при GET-запиті. При POST-запиті перевіряє дані за допомогою LoginForm, аутентифікує користувача, виконує його вхід та перенаправляє на профіль з повідомленням про успішний вхід. Функція logout_view здійснює вихід користувача з системи і перенаправляє на сторінку входу(Дод. А, рис. А.1.4).

Функція profile відображає профіль користувача, дозволяючи редагувати його для поточного користувача, який є авторизованим(за допомогою форми BlogUserChangeForm), та переглядати інші профілі(Дод. А, рис. А.1.5).

Функції follow та unfollow дозволяють авторизованим користувачам підписуватися або відписуватися від інших користувачів через POST-запити,

повертаючи JSON-відповідь з результатом операції – робиться це для того, щоб можна було оновити дані на сторінці без перезавантаження самої сторінки(Дод. А, рис. А.1.6).

Функція user_list відображає список усіх зареєстрованих користувачів, отриманих з моделі BlogUser(Дод. А, рис. А.1.7).

- forms.py

Форма LoginForm використовується для обробки даних під час входу користувача. Вона містить поля email для введення електронної пошти, password для введення пароля та поле recaptcha для захисту від автоматичних запитів за допомогою ReCAPTCHA(Дод. А, рис. А.1.8).

Форма BlogUserCreationForm розширює стандартну форму UserCreationForm та дозволяє створювати нового користувача. Вона містить основні поля, такі як email, nickname, first_name та last_name, а також поле recaptcha для захисту від спаму. Модель, з якою пов’язана ця форма, — BlogUser(Дод. А, рис. А.1.9).

Форма BlogUserChangeForm використовується для редагування профілю користувача. Вона містить ті поля, які користувач має право змінювати - тобто лише емейл, нікнейм, ім’я та прізвище та поля соц. мереж (Дод. А, рис. А.1.10).

- routes.py

Файл routes.py визначає маршрути для обробки запитів, які пов’язані з додатком користувачів(Дод. А, рис. А.1.11).

Кожен маршрут відповідає за певну дію в даному додатку - наприклад, маршрут register/ веде до функції register, яка обробляє реєстрацію нового користувача. Шлях login/ направляє на login_view для входу користувача, а logout/ — на logout_view, щоб вийти з системи. Шлях profile/<int:user_id>/ дозволяє переглянути профіль конкретного користувача, а маршрути follow/<int:user_id>/ і unfollow/<int:user_id>/, в свою чергу, забезпечують функціональність підписки та відписки. Наприкінці, маршрут users/ відображає список усіх користувачів.

- admin.py

Файл admin.py налаштовує адміністративний інтерфейс Django для моделі BlogUser(Дод. А, рис. А.1.12).

Використовується кастомний клас BlogUserAdmin, який розширює базовий клас UserAdmin, щоб зручно відображати й редагувати інформацію про користувачів у панелі адміністратора.

Поле model вказує модель BlogUser, а add_form і form визначають форми для створення та редагування користувачів (BlogUserCreationForm і BlogUserChangeForm). Властивість list_display визначає поля, які відображаються в списку користувачів: email, nickname, first_name, last_name, is_staff, is_active. list_filter дозволяє фільтрувати користувачів за статусом is_staff і is_active.

Секції fieldsets і add_fieldsets визначають структуру полів на сторінках редагування й створення користувачів. Поля search_fields і ordering задають критерії пошуку та сортування в списку користувачів.

- middleware.py

Файл middleware.py містить клас ActiveUserMiddleware, який розширює MiddlewareMixin, він відстежує активність аутентифікованих користувачів і зберігає час їхньої останньої активності в кеші, щоб, наприклад, показувати, коли користувач був останній раз онлайн. Метод process_request виконується для кожного запиту, що надходить до сервера. Якщо користувач автентифікований (це в даному випадку перевіряється через request.user.is_authenticated) і має активну сесію (request.session.session_key), система формує унікальний ключ кешу у форматі last-seen-{user_id}. Якщо в кеші немає даних про останню активність користувача, оновлюється поле last_login моделі BlogUser для цього користувача до поточного часу (timezone.now()). Потім, для збереження ресурсів системи, час останньої активності користувача записується в кеш з таймаутом у 300 секунд(Дод. А, рис. А.1.13).

- login.html

Шаблон login.html відображає сторінку входу на сайт. У блоці content розташована форма входу. Форма відправляється методом POST і включає

захист від CSRF-атак за допомогою спеціалізованого тегу {% csrf_token %}(без нього форму не відправить і буде відповідна помилка).

Вміст форми ({{ form.as_p }}) автоматично рендерить всі поля форми, визначені в LoginForm. Вигляд, який має темплейт(шаблон) авторизації користувача, зображено в Додатку А на рис. А.1.14.

- profile_edit.py

Шаблон profile_edit.html у Django надає інтерфейс для редагування профілю користувача, де можна змінити персональні дані та керувати публікаціями.

У лівій колонці розташовано профільну інформацію користувача, включаючи фото, нікнейм, кількість підписників, дату приєднання та статус онлайну. Якщо профільне зображення відсутнє, використовується зображення за замовчуванням. Є можливість завантаження нового фото, приховане поле для вибору файлу відкривається при натисканні на зображення (Дод. А, рис. А.1.15).

Під профільною інформацією відображаються соціальні посилання (GitHub, Twitter, Instagram, Facebook), кожне з яких має іконку та скорочену версію імені користувача. Натискання на ім'я відкриває поле для введення повного імені, що дозволяє змінити його.

У правій колонці представлені текстові поля для редагування особистих даних: email, ім'я та прізвище. Поле для email доступне лише для перегляду. Поруч розміщена кнопка "Save Changes" для збереження змін.

У секції scripts JavaScript додає функціональність для редагування імен у соціальних профілях: натискання на соціальне ім'я дозволяє відобразити його повністю у полі для введення, сховавши попереднє значення (Дод. А, рис. А.1.16).

- register.html

Шаблон register.html є нічим іншим, як сторінкою створення нового користувача(Дод. А, рис. А.1.17).

Він, як і всі інші основні шаблони даного проєкту, наслідує основний шаблон base.html, що забезпечує загальне оформлення сторінки, а також

інтеграцію статичних файлів і основної структури вебсайту.

Секція content містить форму, яка надсилає дані POST-методом для створення нового профілю. У формі також передбачено завантаження файлів(для завантаження профільного зображення в даному випадку). Поле {% csrf_token %} додає захист від CSRF-атак, забезпечуючи безпечну передачу даних.

Кожне поле форми (наприклад, ім'я, електронна пошта, пароль) відображається всередині блоку form-group. Якщо у певного поля є помилки введення, до цього блоку додається клас form-group-error, щоб виділити його для користувача.

Для полів із помилками під полем відображається список повідомлень про помилки. Кожна помилка виводиться як елемент списку.

У нижній частині форми знаходиться кнопка "Create Profile", яка надсилає заповнену форму на сервер для обробки та створення нового профілю користувача.

- profile_view.html

Цей Django-шаблон відповідає за відображення профілю користувача і розділений на дві основні частини: одна містить інформацію про самого користувача, а інша — його публікації.

У першій частині відображається аватарка користувача, яка змінюється в залежності від того, чи є зображення профілю. Якщо воно є, воно показується в круглій рамці, якщо ні — використовується зображення за замовчуванням. Нижче зображення виводяться ім’я користувача (нікнейм), кількість підписників та дата реєстрації. Крім того, відображається статус користувача (онлайн чи офлайн), а також кнопка для підписки або відписки, яка змінюється в залежності від того, чи підписаний поточний користувач на цього(Дод. А, рис. А.1.18).

У наступному блоці йде список соціальних мереж, де користувач може вказати свої профілі. Для кожної соцмережі відображається відповідна іконка та ім’я користувача, яке можна обрізати до 14 символів, якщо воно занадто довге.

Друга частина шаблону виводить інформацію про контактні дані

користувача, включаючи електронну пошту, ім’я та прізвище (якщо вони вказані). Ця інформація представлена текстовими полями з підписами та значеннями.

Останній блок в списку(але не по важливості) відображає публікації користувача. Для кожного поста вказується його основна інформація, а також кнопка для переходу до сторінки детальної інформації посту.

В шаблоні також є JavaScript, який відповідає за зміну статусу підписки на користувача. Кнопка "Follow/Unfollow" викликає функцію toggleFollow, яка, в свою чергу, надсилає запит до сервера для оновлення підписки і відповідно змінює текст кнопки та кількість підписників(Дод. А, рис. А.1.19). Як результат, шаблон має інтуїтивно зрозумілий дизайн.

- user_list.html

Цей темплейт просто відображає список користувачів. Він також розширює базовий шаблон, отримуючи загальний каркас для сторінки, і завантажує статичні файли, які потрібні для покращення зовнішнього вигляду сторінки.

Основним контентом сторінки є контейнер, в якому розміщений список користувачів. Кожен користувач є елементом списку і включає його основну інформацію - картку з фотографією, нікнеймом, кількістю підписників та постів. Якщо профільне зображення користувача наявне, воно відображається, інакше використовується дефолтне зображення(Дод. А, рис. А.1.20).

Для кожного користувача є кнопка, що веде на його профільну сторінку. У випадку, якщо список користувачів порожній, виводиться повідомлення "No users found". Крім того, шаблон також дозволяє зручно переходити до інших профілів, зберігаючи функціональність кнопок та коректне відображення статистики для кожного користувача.

2.3.2. Модуль(додаток) blog

Додаток blog реалізує функціонал створення та редагування постів, а також додає коментарі, категорії та рейтинги до постів. У додатку наявні всі файли, які

потрібні для роботи з постами.

- models.py

У файлі models.py даного додатку визначено одразу кілька моделей, що відповідають основним сутностям блогу: постам, категоріям, коментарям, оцінкам та переглядам.

Модель Post має в собі менеджер PostManager(Дод. А, рис. А.2.1), який дозволяє фільтрувати пости, наприклад, за статусом затвердження і має поля заголовку, унікальний URL (slug), категорію, зміст, зображення та автора(Дод. А, рис. А.2.2). Вона також має прапорець для затвердження поста адміністраторами(без нього поста не зможе побачити ніхто окрім адмінів та автора). Також, дана модель містить має методи для отримання URL поста та стиснення зображень при збереженні - для економії місця на сервері(Дод. А, рис. А.2.3).

Модель Category, як зрозуміло із назви, відповідає за категорії постів і підтримує ієрархію завдяки використанню MPTTModel. Вона містить назву, опис і можливість вказувати батьківську категорію(Дод. А, рис. А.2.4). Модель дає можливість будувати структуру категорій у вигляді дерева, що, в свою чергу, дає змогу ефективно організувати пости за тематикою. В ній також є мета-клас, який визначає назву даної моделі в адмін-панелі та метод, який визначає точних шлях до поточного посту(Дод. А, рис. А.2.5).

Модель Comment є моделлю коментарів до постів і так само, як Category, підтримує деревоподібну структуру для організації відповідей на коментарі через використання MPTTModel. Коментарі мають текст, автора і дату створення, а також поле для позначення, чи був коментар відредагований(Дод. А, рис. А.2.6).

Модель Rating дозволяє користувачам ставити лайки або дизлайки до постів і зберігає інформацію про пост, користувача, оцінку та IP-адресу(для уникнення подвійних оцінок з одного пристрою). Всі оцінки зберігаються разом із метаданими про дату та час створення(Дод. А, рис. А.2.7).

Модель ViewCount зберігає інформацію про перегляди постів, включаючи

IP-адресу користувача та дату і час перегляду. Це дозволяє рахувати, скільки разів був переглянутий конкретний пост, а також відстежувати перегляди за певний період(Дод. А, рис. А.2.8).

Ці моделі в комбінації забезпечують багатий функціонал для блогу і дозволяють ефективно управляти контентом і мати повний функціонал для оптимальної роботи з постами.

- views.py

У файлі views.py додатку блогу реалізовано ряд класів, що відповідають за різні функції відображення та обробки запитів. Клас PostListView відображає список всіх постів - при цьому він підтримує пагінацію(до п'яти постів на сторінку). А в методі get_context_data додається заголовок для шаблону списку постів(Дод. А, рис. А.2.9).

Клас PostSearchResultView надає можливість пошуку постів за ключовими словами. Пошук здійснюється через комбінований вектор пошуку для змісту та заголовка поста. Запити фільтруються та сортуються за релевантністю, а в результаті користувач отримує сторінку з постами, що відповідають запиту. Заголовок сторінки змінюється відповідно до введеного запиту(Дод. А, рис. А.2.10).

PostByTagListView відображає пости, що мають певний тег. Тег отримується з параметрів URL, після чого уже відбувається фільтрацію по тому тегові, при цьому “відкидаються” пости, які не були підтверджені адміністрацією(Дод. А, рис. А.2.11).

PostByCategoryListView, в свою чергу, показує пости, що належать певній категорії. Категорія постів також, як і в випадку з тегами, передається через параметри URL, і потім пости фільтруються за цією категорією. Клас підтримує пагінацію і динамічно змінює заголовок в шаблоні відповідно до обраної категорії(Дод. А, рис. А.2.12).

Клас PostDetailView надає детальну інформацію про конкретний пост. Крім основного змісту, цей клас дозволяє відображати схожі пости, використовуючи теги поточного поста. У методі get_similar_posts вибираються

пости, що мають спільні теги, а потім вони випадковим чином перемішуються для більшої варіативності. Контекст сторінки також включає форму для створення коментарів(Дод. А, рис. А.2.13).

PostCreateView дозволяє створювати нові пости, але лише авторизованим користувачам(в іншому випадку буде перенаправлення на сторінку авторизації). Клас використовує форму створення поста, а також перенаправляє на особисту сторінкуавтора після успішного створення поста(Дод. А, рис. А.2.14).

PostUpdateView дозволяє користувачам редагувати свої пости. Також є логіка, яка дозволяє лише авторам постів та адміністраторам оновлювати їх(Дод. А, рис. А.2.15).

PostDeleteView відповідає за видалення постів. Після видалення користувач також перенаправляється на свій профіль, а процес видалення супроводжується повідомленням про успішну операцію(Дод. А, рис. А.2.16).

Для створення та обробки коментарів реалізовано клас CommentCreateView. Він дає можливість користувачам залишати коментарі до постів, підтримуючи можливість відповіді на інші коментарі. Конкретно в цього представлення важливою особливістю є підтримка асинхронних запитів, що дозволяє додавати коментарі без перезавантаження сторінки(Дод. А, рис. А.2.17).

Клас RatingCreateView дозволяє користувачам ставити оцінки (тобто лайки чи дизлайки) до постів. Оцінки зберігаються разом з інформацією про користувача та IP-адресу, щоб запобігти “махінаціям” з оцінками. Якщо користувач вже поставив оцінку, він може змінити її або видалити, а результат миттєво оновлюється через JSON-відповідь без оновлення сторінки(Дод. А, рис. А.2.18).

Кожен з цих класів відповідає за конкретні функції у блозі та забезпечує зручний інтерфейс для користувачів, а також гнучкість для адміністраторів при управлінні контентом. Також тут беруть участь міксини, які обмежують в правах доступу до створення, редагування чи видалення тих чи інших об’єктів,

користувачів, які або не авторизовані, або не мають мати доступу, бо не є авторами.

- routes.py

Файл routes для додатку blog визначає маршрути, що відповідають за обробку різних запитів до сторінок та функцій, пов'язаних з постами, коментарями та рейтингами у блоговій системі. Він містить основні маршрути до функція із списками постів, пошуком, фільтрацією за тегами та категоріями та ін. Кожен шлях пов'язаний з відповідним класом представлення, який обробляє логіку запиту.

Маршрут 'posts/' веде до PostListView, що відображає список всіх постів на сторінці. Шлях 'posts/tags/<str:tag>/' використовує PostByTagListView для відображення постів, відфільтрованих за конкретним тегом, що дозволяє користувачам переглядати публікації з відповідними тегами. Шлях 'posts/categories/<str:slug>/' аналогічно до попереднього використовує PostByCategoryListView, тільки уже для фільтрації постів за категорією.

Для пошуку постів у нас є маршрут 'posts/search/', який викликає PostSearchResultView для пошуку постів за заданим ключовими словами. Шлях 'post/<str:slug>/', в свою чергу, відповідає за показ деталей конкретного поста, включаючи подібні записи.

Створення нового поста виконується за допомогою маршруту 'create_post/', пов’язане з PostCreateView, що дозволяє авторизованим користувачам додавати публікації до блогу. Оновлення і видалення постів виконуються через 'post/<str:slug>/update/' та 'post/<str:slug>/delete/', які використовують PostUpdateView та PostDeleteView відповідно. Ці три маршрути доступні тільки для авторів постів та адміністраторів.

Коментарі до постів додаються, користуючись маршрутом 'post/<int:id>/add_comment/', який використовує CommentCreateView. Також є маршрут 'post/<int:id>/rating/', що пов'язаний з RatingCreateView і відповідає за оцінки, які ставлять користувачі постам.

Як результат, цей файл встановлює основу для роботи з постами, забезпечуючи маршрутизацію для ключових дій користувача та відображення інформації про них(Дод. А, рис. А.2.19).

- forms.py

Даний файл містить форми для створення та оновлення постів, а також для додавання коментарів до них. Тобто, у ньому містяться форми-класи для управління основними полями постів та коментарів з можливістю додаткової валідації.

Клас PostCreateForm, як зрозуміло із назви, працює з моделлю Post і визначає поля, що підлягають заповненню: title, category, tags, content та image. У методі __init__ встановлюються додаткові атрибути для полів content, tags та category, додаючи CSS-класи для налаштування їх вигляду у формі. Поля title та content визначаються як необов’язкові, але валідація, описана в методах clean_content та clean_title, все ж вимагає їх заповнення перед відправкою форми. Метод clean_content використовує BeautifulSoup для перевірки, що content містить текст, а не лише HTML-теги(Дод. А, рис. А.2.20).

Форма PostUpdateForm наслідує форму PostCreateForm та просто повторно використовує її поля без додаткової логіки, що дозволяє редагувати пост із тими ж вимогами до полів, що й під час створення(Дод. А, рис. А.2.21).

Клас CommentCreateForm є формою для створення коментарів до постів і також наслідується від forms.ModelForm, працюючи з моделлю Comment. Форма містить додаткове поле parent, яке є прихованим полем для встановлення ієрархії коментарів, а також поле content, де користувач вводить текст коментаря. Поле content додатково має налаштування стилів - за допомогою віджетів для встановлення розмірів текстової області та CSS-класів для стилізації(Дод. А, рис. А.2.22).

Як результат, цей файл додає зручний спосіб валідації і обробки даних з форм для постів та коментарів і забезпечує додаткову кастомізацію(стилі) та перевірку введених даних перед їх збереженням.

- admin.py

Файл admin.py для додатку blog відповідає за представлення моделей у панелі адміністратора Django, спрощуючи роботу з об'єктами моделей, при цьому даний додаток має більше налаштувань в адмін-панелі, ніж інші додатки, так як здебільшого саме в цьому розділі адміністрація працює(підтвердження постів, перегляд коментарів тощо).

Для моделі Category використовується клас CategoryAdmin, який наслідується від DraggableMPTTAdmin. Робиться це для того, щоб створити зручний інтерфейс для ієрархічних моделей(тобто категорії можуть мати вкладену структуру). Поля, що відображаються в списку, включають tree_actions та indented_title - для візуального відображення структури, а також id, title та slug. Slug автоматично генерується на основі title, і вони вдвох використовуються як посилання. Поля поділяються на розділи "Main Info" (основна інформація) і "Description" (опис) — так було зроблено для зручності(Дод. А, рис. А.2.23).

Для моделі Comment використовується клас CommentAdminPage, який дозволяє представити коментарі в ієрархічному форматі. Поля, що відображаються в списку, включають tree_actions та indented_title для ієрархії, а також post, author, created_at, is_updated. Коментарі можуть фільтруватися за датою створення та оновлення, автором і записом(Дод. А, рис. А.2.24).

Клас PostAdmin використовується для управління моделлю Post. Поле slug автоматично заповнюється на основі title. Метод get_readonly_fields забезпечує, що поле author стає лише для читання після створення об'єкта(Дод. А, рис. А.2.25).

Для моделі ViewCount створюється клас ViewCountAdmin, який не має додаткових налаштувань, але забезпечує можливість керувати записами моделі ViewCount через адмін-панель(Дод. А, рис. А.2.26).

Таким чином, admin.py у цьому додатку забезпечує зручне та інтуїтивне управління ієрархічними категоріями, коментарями, постами та кількістю переглядів через адмін-панель Django.

- mixins.py

Даний файл містить клас ViewCountMixin — міксин для підрахунку

переглядів публікацій.

Він наслідує метод get_object — за допомогою нього він отримує пост. Потім, для визначення унікального перегляду, використовується IP-адреса клієнта(потрібна для уникнення “махінацій” з переглядами), яка отримується через виклик функції get_client_ip, яка в свою чергу, імпортована з модуля services.utils. Після цього викликається метод get_or_create для моделі ViewCount, який або створює новий запис перегляду для відповідного об’єкта і IP-адреси, або повертає вже наявний запис, якщо перегляд з цього IP-адреса вже зареєстрований. Завдяки цьому механізму міксин допомагає відстежувати унікальні перегляди кожної публікації, повертаючи об'єкт публікації після оновлення інформації про перегляди(Дод. А, рис. А.2.27).

- signals.py

Файл signals.py містить в собі сигнали, які реагують на зміни статусу моделі Post, зокрема перевірку і відправку сповіщень про схвалення публікації. Спочатку створюється словник old_values, щоб зберігати старі значення поля is_approved для об'єктів Post перед збереженням змін.

Функція store_old_is_approved використовує сигнал pre_save для зберігання попереднього значення поля is_approved перед тим, як зберегти зміни, приц цьому вона також перевіряє наявність первинного ключа (pk) об'єкта - щоб переконатися, що йдеться про оновлення наявного запису, а не про створення нового. Після цього функція зберігає значення is_approved в словник old_values з використанням pk як ключа.

Функція post_approved_handler, є прив’язаною до сигналу post_save, тобто виконується після збереження об'єкта Post. Якщо публікація була створена (created=True), функція завершується без подальших дій, бо ми повинні реагувати саме на процес підтвердження посту, який не має бути в процесі створення, а в процесі модифікації(в адмін-панелі). Якщо публікація вже існувала і її статус is_approved змінився з False на True, викликається завдання trigger_approval_email_message_task, яке надсилає автору публікації лист про схвалення. Потім, якщо у автора є підписники, завдання

trigger_new_post_email_message_task надсилає сповіщення всім підписникам з електронною поштою та ім'ям публікації. В кінці функція видаляє старий запис значення is_approved з old_values, щоб зберегти актуальні значення тільки для змінюваних об'єктів(Дод. А, рис. А.2.28).

- tasks.py

Файл tasks.py має функції для відправки асинхронних завдань Celery. Функція trigger_approval_email_message_task(завдання призначене для відправки електронного листа автору про схвалення його публікації) отримує адресу електронної пошти одержувача і назву публікації. Ця функція використовує current_app.send_task, щоб надіслати завдання до сервісу email_sender.send_approval_email_message_task, передаючи йому одержувача та назву публікації як аргументи. Потім ці аргументи будуть використані в мікросервісі-надсилачі.

Функція trigger_new_post_email_message_task(задача відправки листів підписникам про новий пост) приймає список електронних адрес підписників, ім'я автора та назву публікації. Виклик current_app.send_task передає ці дані до завдання email_sender.send_new_post_email_message_task(Дод. А, рис. А.2.29).

- create_post.html

Файл create_post.html — це шаблон для створення нової публікації в блозі. У верхній частині шаблону підключаються статичні файли та теги для використання бібліотеки crispy_forms. У блоці extra_styles додається спеціальний файл(create_post.css) для стилізації сторінки.

Основний контент сторінки міститься в блоці content, де розміщена HTML-форма для створення нової публікації. Форма містить елементи для заповнення заголовку, обрання категорії, додавання тегів (через текстове поле для введення і через кому), написання контенту і завантаження зображення для публікації, яке буде потім відображене на сторінці списку постів.

Якщо ж у формі є помилки, вони будуть відображатися у вигляді червоного повідомлення з переліком всіх полів, що містять помилки. Для кожного поля, яке

має помилку, виводиться його назва та текст помилки. Також відображаються будь-які загальні помилки форми, не пов’язані з конкретним полем.

Кожне поле форми представлено через тег form-group, що забезпечує структуроване відображення з підписами (label) для кожного елемента вводу(Дод. А, рис. А.2.30).

- delete_post.html

Основний контент сторінки знаходиться в блоці content, де представлена картка з повідомленням про підтвердження видалення.

У верхній частині шаблону є заголовок "Deletion of post", який відразу дає зрозуміти мету сторінки. У формі(яка працює через метод POST), є повідомлення з попередженням про видалення, яке відображається всередині блока з класом alert alert-warning(виведеною через {{ post.title }}), і яке запитує підтвердження дії. Нижче повідомлення-попередження розміщені дві кнопки. Кнопка “Delete it” червоного кольору є кнопкою підтвердження видалення, яка надсилає форму для остаточного видалення публікації. Кнопка “No, leave it” синього кольору відміняє процес видалення посту та повертає користувача на його особисту сторінку(Дод. А, рис. А.2.31).

- post_details.html

Шаблон post_details.html розширює основний шаблон base.html і відображає деталі конкретного посту блогу. Вміст сторінки знаходиться в основному контейнері, поділеному на дві колонки: широку основну колонку для самого посту та вузьку колонку для бічної панелі (sidebar).

Основна колонка починається з картки, яка містить заголовок посту та посилання на профіль автора(Дод. А, рис. А.2.32). Нижче заголовка розміщено основний текст посту, який відображається без фільтрації(для використовується спеціальний тег safe), що дозволяє виводити HTML-теги, запобігаючи екрануванню контенту — це нам потрібно для коректного відображення особливих символів, які міг ввести користувач через кастомізовану форму вводу, яка використовується у нас для постів. Якщо пост має теги, вони відображаються під текстом посту у вигляді клікабельних бейджів.

Далі йде блок кнопок для взаємодії з постом, включаючи кнопки "Like" та "Dislike", а також відображення загальної кількості оцінок посту. З правого боку блоку кнопок показано дату публікації посту та лічильник переглядів, що відображає загальну кількість переглядів.

- posts_list.html

Шаблон posts_list.html відповідає за відображення списку постів у блозі з можливістю пошуку та взаємодії з постами. У верхній частині сторінки можна побачити форму пошуку, що дозволяє користувачам шукати пости за їхнім вмістом. Якщо пошуковий запит активний, кнопка змінюється на "Clear Search", що дає можливість очистити пошук і повернутися до повного списку постів(Дод. А, рис. А.2.33).

Далі після цього йде цикл, що відображає кожен пост з базовою інформацією, такою як заголовок, категорія, теги та дата публікації. Для кожного посту відображається зображення, яке взято з поля image посту. Якщо ж такого немає, то використовується зображення за замовчуванням. Заголовок посту відображається як посилання, яке веде на сторінку деталей посту. Категорія посту показується як бейдж, що дає користувачам можливість відразу відфільтрувати пости за вподобаною категорією.

Якщо пост має теги, вони виводяться під заголовком у вигляді посилань, що дозволяють переходити до списку постів, які мають цей тег. Під кожним постом є кнопки для взаємодії, зокрема для "Like" і "Dislike", а також для перегляду загальної кількості оцінок посту. Поряд з цими кнопками відображається кількість переглядів посту.

У кінці сторінки підключається скрипт, який відповідає за обробку кнопок для рейтингу постів. Бічна панель підключається за допомогою шаблону sidebar.html, що дозволяє користувачам побачити додаткову інформацію або рекомендації на сайті.

- update_post.html

Шаблон update_post.html створений для редагування існуючого поста в блозі і базується на формі для редагування поста - заголовку, вмісту, тегів,

категорій. Спочатку відображається форма з усіма необхідними полями.

Якщо ж форма містить помилки, вони відображаються в спеціальному блоці(Дод. А, рис. А.2.34).

В полі для введення тегів спеціально пояснюється, що теги повинні бути введені через кому. Лише того, як користувач заповнить або відредагує всі поля, він зможе надіслати форму(натиснувши на кнопку "Submit").

Форма надає можливість завантажити зображення, для цього в неї наявний атрибут enctype="multipart/form-data" - він потрібен для обробки файлів, таких як зображення. Кожен блок форми стилізований за допомогою класів Bootstrap, а також, окрім нього, є додаткові стилі для покращення вигляду сторінки.

- blog_tags.py

Файл blog_tags.py містить три спеціальні теги для шаблонів Django, що взаємодіють з моделями блогу(Дод. А, рис. А.2.35).

Перший з них, popular_tags, використовує модель Tag з пакету taggit для отримання найбільш популярних тегів. Він використовує функцію annotate, щоб порахувати, скільки разів кожен тег з'являється в постах, і повертає список тегів разом з кількістю використань і їхніми слагами.

Другий тег, popular_posts, як зрозуміло із назви, повертає популярні пости на основі кількості переглядів. Він порівнює перегляди постів за останній тиждень і перегляди, здійснені саме сьогодні. Для цього використовуються анотації та фільтри на основі дати. Тег повертає список з п'яти постів, відсортованих спочатку за загальною кількістю переглядів, а потім за кількістю переглядів, здійснених сьогодні.

Третій тег, show_latest_comments, включає блок з останніми коментарями до постів. Він отримує певну кількість(але по замовчуванню п’ять) останніх коментарів, відсортованих за датою створення, і передає їх у шаблон для відображення.

Ці теги дають можливість інтегрувати динамічний контент, такий як популярні теги, пости й коментарі, в шаблони веб-сайту і при цьому допомагають уникнути дублювання коду з перевикористанням цих тегів в різних

шаблонах.

- custom_filters.py

Файл custom_filters.py містить лише один фільтр(add_class) для шаблонів, який додає CSS клас і приймає два параметри: перший — це поле форми, а другий — сам клас, який потрібно додати до цього поля. Також він використовує вбудований метод as_widget, щоб змінити атрибути елемента, додаючи в нього аргумент-клас. Як результат, цей фільтр дозволяє кастомізувати вигляд полів форм без необхідності вручну додавати клас до кожного елемента в шаблоні(Дод. А, рис. А.2.36).

2.3.3. Модуль(додаток) system

- models.py

Модель Feedback зберігає інформацію про відгуки користувачів. Поле subject визначає тему відгуку і має обмеження в 255 символів, тоді як content містить основний текст відгуку. Поле created_at автоматично фіксує час створення відгуку, а поле ip_address зберігає IP-адресу. Поле user пов’язує відгук з конкретним користувачем, тобто автором даного фідбеку, використовуючи модель користувача. Внутрішній клас Meta вказує на те, що відгуки будуть сортуватися за датою створення у порядку спадання. Метод __str__ просто повертає текстовий опис, який містить електронну адресу відправника - додано для зручності перегляду в адмін-панелі(Дод. А, рис. А.3.1).

- forms.py

Форма FeedbackCreateForm забезпечує інтерфейс для створення об’єктів моделі Feedback. Вона включає два основних поля — subject та content, які відповідають за тему та зміст відгуку відповідно. Поле recaptcha, реалізоване за допомогою ReCaptchaField і віджета ReCaptchaV2Checkbox, додає захист від спаму за допомогою Google ReCAPTCHA. Конструктор __init__ застосовує до кожного текстового поля (окрім recaptcha) CSS-клас form-control для стилізації та вимикає автозаповнення, налаштовуючи атрибути class і autocomplete для полів(Дод. А, рис. А.3.2).

- urls.py

Файл urls.py має в собі лише один маршрут для обробки відгуків, і спрямовує користувача до представлення, яке відповідає за створення нового відгуку. Для того, щоб можна було легко посилатися на даний маршрут в шаблонах і в коді та забезпечити централізовану точку доступу для функції відгуків у додатку, даному маршруту було присвоєно ім'я 'feedback'(Дод. А, рис. А.3.3).

- views.py

У файлі views.py ми використовуємо клас FeedbackCreateView для створення нового відгуку від зареєстрованого користувача. Він наслідується від LoginRequiredMixin, щоб доступ був тільки для авторизованих користувачів, і SuccessMessageMixin - щоб показувати повідомлення про успіх після відправки форми. В класі використовується модель Feedback, форма FeedbackCreateForm та шаблон 'system/feedback.html'. В функції для встановлення додаткового контексту ми додаємо заголовок для шаблону - "Feedback Form". Після успішної відправки, користувач перенаправляється на сторінку профілю, а IP-адреса та користувач додаються до збереженого відгуку(Дод. А, рис. А.3.4).

Тут важливий момент - також викликається асинхронне завдання trigger_contact_email_message_task, яке надсилає електронний лист з даними відгуку, а клієнту відразу повертається контроль над сторінкою, тобто він не очікує завершення даного процесу.

Окрім цього, є три функції для обробки помилок: tr_handler404, tr_handler500 і tr_handler403. Як стає зрозуміло із їх назв, відповідають вони за відображення сторінок з помилками 404, 500 і 403 статусів відповідно, використовуючи спільний шаблон 'system/errors/error_page.html'(Дод. А, рис. А.3.5).

- admin.py

У файлі admin.py ми додаємо відображення в адміністративній панелі моделі Feedback. Клас FeedbackAdmin, який наслідується із вбудованого класу admin.ModelAdmin, налаштовує відображення полів ip_address та user у списку

елементів — ці поля є посиланнями, що дозволяють перейти до детального перегляду конкретного запису, забезпечуючи зручний доступ до основної інформації про відгук, включаючи IP-адресу та автора відгуку(Дод. А, рис. А.3.6).

- feedback.html

Шаблон feedback.html відповідає за відображення форми для відправки відгуку, він розширює основний шаблон base.html і використовує специфні стилі, завантажені з файлів статики. Форма, як і більшість інших форм даного додатку, надсилає дані методом POST, включаючи токен CSRF для безпеки. Поля форми рендеряться за допомогою {{ form.as_p }}, що відображає кожне поле у вигляді абзацу. Після полів форми розміщено кнопку "Send email" темного кольору, яка подає форму на обробку. Після успішного надсилання фідбеку, користувача буде перенаправлено на його особисту сторінку(Дод. А, рис. А.3.7).

- error_page.html

Шаблон error_page.html показує користувачам повідомлення про помилку у вигляді червоного сповіщення з класом alert alert-danger. Це сповіщення має кнопку закриття і з’являється з анімацією fade-in, створюючи інтерактивний вигляд. У повідомленні відображається змінна error_message, яка містить текст помилки, а поруч є посилання для повернення на сторінку зі списком постів(Дод. А, рис. А.3.8).

2.3.4. Модуль services

- weekly_report.py

Файл weekly_report.py — це модуль для створення PDF-звіту про активність користувачів за останній тиждень.

Метод header встановлює заголовок "Weekly Report" вгорі кожної сторінки, а footer додає номер сторінки внизу. Для кожного розділу методи chapter_title та chapter_body задають стилі і форматування тексту.

Метод create_table дозволяє створювати таблиці з певними заголовками та даними, застосовуючи ширину колонок і автоматичне вирівнювання

тексту(Дод. А, рис. А.4.1).

Функція generate_weekly_report збирає дані за останній тиждень, такі як нові пости, нові користувачі і загальна кількість переглядів. Для нових постів та користувачів створюються окремі таблиці, де відображаються всі основні дані, які можуть знадобитися для відображення інформації про ці сутності — назва, автор, кількість переглядів, ім’я, прізвище тощо(Дод. А, рис. А.4.2).

При цьому, на окремій сторінці підраховується загальна кількість нових постів, користувачів і переглядів — це потрібно для швидкого перегляду результатів за тиждень. Звіт зберігається в директорії reports у форматі PDF з іменем, яке включає поточну дату і час створення.

- utils.py

Файл utils.py має в собі допоміжні функції та класи для управління медіафайлами, оптимізації зображень, створення унікальних URL-ідентифікаторів і отримання IP-адрес користувачів — тобто для допоміжних інструментів, які можуть знадобитися в різних додатках. Клас CkeditorCustomStorage налаштовує шлях для збереження медіафайлів, які завантажуються через редактор Ckeditor[8]. Він визначає методи для отримання назви папки на основі поточної дати, дозволяючи структурувати зберігання за датою. А вже в методі _save відбувається об’єднання назви файлу з датованою папкою перед передачею його в базовий метод збереження. Файли зберігаються в директорії uploads, яка знаходиться всередині основного каталогу MEDIA_ROOT(Дод. А, рис. А.4.3).

Функція unique_slugify генерує унікальні URL-ідентифікатори (slugs) для моделей, додаючи до базового slug випадковий рядок символів, якщо такий slug уже існує в базі даних, що дозволяє уникнути конфліктів і забезпечує унікальність URL і робить навігацію сайтом зручнішим для користувача — адже йому не треба буде згадувати(чи вгадувати) айді поста.

Функція get_client_ip отримує IP-адресу користувача - спочатку перевіряє наявність заголовка HTTP_X_FORWARDED_FOR для обробки випадків, коли

користувач заходить через проксі, і, за відсутності цього заголовка, використовує IP-адресу з заголовка REMOTE_ADDR.

Функція image_compress оптимізує зображення, зменшуючи його до заданих висоти і ширини, якщо воно перевищує ці розміри. Вона також обробляє орієнтацію зображення за допомогою ImageOps.exif_transpose, щоб уникнути проблем із відображенням після повороту, і зберігає його у форматі JPEG з високим рівнем стиснення та оптимізацією для економії місця на сервері(Дод. А, рис. А.4.4).

- tasks.py

Файл tasks.py у модулі services містить визначення асинхронних завдань, що виконуються за допомогою бібліотеки Celery.

Перше завдання, trigger_contact_email_message_task, виконується для відправки електронного листа, воно отримує параметри, такі як тема листа, зміст, IP-адреса користувача, його email та псевдонім. Але, так як це лише “трігер”, дане завдання лише відправляє повідомлення в чергу завдань для виконання іншої задачі email_sender.send_contact_email_message_task, передаючи ці дані через аргументи.

Друге завдання, db_backup_task, здійснює резервне копіювання бази даних PostgreSQL - воно викликає команду db_backup за допомогою функції call_command, яка дозволяє виконувати консольні команди в середовищі Django.

Третє завдання, generate_weekly_report_task, викликає функцію для генерації щотижневого звіту про активність користувачів. Це завдання запускає процес створення звіту в форматі PDF за допомогою функції generate_weekly_report, яка збирає інформацію про нових користувачів, пости та їхні перегляди за останній тиждень і яка була детально описана в одному із попередніх пунктів.

Усі ці завдання використовують Celery для асинхронного виконання, що дозволяє виконувати їх у фоновому режимі, при цьому не приносячи жодного дискомфорту користувачам(Дод. А, рис. А.4.5).

- mixins.py

Файл mixins.py має міксин AuthorRequiredMixin, який контролює доступ користувачів до модифікації чи видалення певних об’єктів на основі їхніх прав. Він успадковує функціонал від AccessMixin та перевизначає метод dispatch. Спочатку даний міксин перевіряє, чи є користувач автентифікованим. Якщо ні, то викликається метод handle_no_permission(), що обмежує доступ неавторизованим користувачам.

Далі міксин перевіряє, чи є поточний користувач автором об’єкта або належить до адміністративного персоналу (користувач із правами is_staff). Якщо ця перевірка проходить успішно, запит обробляється за допомогою методу dispatch батьківського класу, що дозволяє доступ до операцій модифікації або видалення. Якщо ж користувач не є автором або адміністратором, з'являється сповіщення про те, що зміна або видалення посту доступні лише автору або адміністратору. Потім користувача перенаправляють на сторінку профілю (profile) із зазначеним ідентифікатором користувача(Дод. А, рис. А.4.6).

- db_backup.py

Файл db_backup.py у модулі services містить Django-команду для створення резервної копії бази даних PostgreSQL, яка запускається щоденно за допомогою Celery.

Клас Command наслідує BaseCommand[9] і реалізує метод handle, який визначає основну логіку збереження резервної копії. В якості першого кроку відображається повідомлення про початок процесу резервного копіювання і створюється директорія backups в кореневому каталозі проєкту (якщо вона ще не існує).

Шлях до файлу резервної копії створюється із зазначенням поточної дати й часу, щоб кожен файл мав унікальну назву, яка містить часову мітку. Потім викликається команда dumpdata, яка зберігає дані бази у JSON-форматі з опціями --natural-foreign і --natural-primary для поліпшення зв'язків між об'єктами. Параметри --exclude запобігають збереженню деяких системних таблиць(таких як contenttypes та admin.logentry, наприклад). Після успішного завершення процесу виводиться повідомлення про успішне збереження резервної копії(Дод.

А, рис. А.4.7).

2.3.5. Модуль(кореневий) blog_project

- settings.py

Файл settings.py є одним із найбільш важливих файлів в будь-якому проєкті Django і містить різні налаштування, для баз даних, кешування, завдань Celery(Дод. А, рис. А.5.1). а також інші важливі параметри, такі я назви папок з медіа-файлами, список активних додатків, тощо. Для обробки черг завдань використовується Celery, з брокером RabbitMQ і RPC як бекенд для збереження результатів. Параметри CELERY_BEAT_SCHEDULE визначають щоденне резервне копіювання бази даних та щотижневе створення звіту про активність користувачів[10].

Конкретно цей додаток-блог використовує PostgreSQL як основну базу даних та Redis для кешування. Підключені різні пакети та додатки, такі як django_celery_results, django_celery_beat, crispy_forms, taggit, django_cleanup та django_ckeditor_5, який, кілька разів уже згадувалось, додає функціональність редактора тексту з розширеними налаштуваннями та кастомною палітрою кольорів.

Налаштування Django також включають параметри для статичних і медіафайлів, перевірку паролів, часового поясу, а також базову конфігурацію шаблонів із процесорами контексту. Модуль django.middleware обробляє безпеку, сесії та аутентифікацію, а також специфічний для даного проєкту ActiveUserMiddleware, що належить додатку accounts.

- celery.py

Файл celery.py налаштовує інтеграцію Celery з Django. Спочатку визначається середовище налаштувань Django за замовчуванням для коректної роботи Celery, вказуючи на файл settings.py проекту. Створюється об'єкт Celery з ім'ям проєкту, після чого він налаштовується на основі параметрів, зазначених у налаштуваннях Django, за допомогою простору імен CELERY. Метод autodiscover_tasks() дозволяє автоматично виявляти завдання з інших додатків

проєкту, що значно спрощує структуру і забезпечує готовність Celery виконувати асинхронні завдання без додаткової конфігурації(Дод. А, рис. А.5.2)[11]

- urls.py

Даний файл urls.py задає маршрутизацію для всього проєкту. Спеціальні обробники handler403, handler404 і handler500 налаштовані для переадресації на відповідні представлення у разі виникнення помилок доступу, відсутньої сторінки або внутрішньої помилки сервера. Основні маршрути включають адміністративний інтерфейс, URL-адреси для додатків accounts, blog та system, а також підтримку редактора ckeditor5[12]. Якщо режим налагодження (DEBUG) увімкнено, до urlpatterns додається можливість обробки медіафайлів для забезпечення доступу до них під час розробки, використовуючи налаштування MEDIA_URL і MEDIA_ROOT. Також, поки що обробники помилок будуть спрацьовувати лише якщо режим налагодження вимкнено, інакше будуть показуватися повні повідомлення про помилки — для кращої інформативності(Дод. А, рис. А.5.3).

2.3.6. Модуль static

- backend.js

Даний файл реалізує функцію getCookie - вона перебирає всі cookies, розділені крапкою з комою, і перевіряє, чи починається рядок cookie з потрібною назвою. Якщо cookie знайдено, його значення декодується і зберігається в змінній, після чого повертається як результат. Файл також містить змінну csrftoken, яка зберігає значення токена CSRF, отримане за допомогою функції getCookie для подальшого використання в запитах AJAX(Дод. А, рис. А.6.1)[13].

- comments.js

Файл comments.js керує функціоналом коментарів на вебсторінці. Він визначає форму коментарів, необхідні змінні та додає обробник подій для надсилання коментарів. Після натискання кнопки відправки форми спрацьовує функція createComment, яка відправляє дані форми на сервер для асинхронної обробки запиту. У разі успішної відповіді сервер надсилає інформацію про новий

коментар, яку скрипт використовує для динамічного створення шаблону коментаря, що додається до списку на сторінці і залежно від того, чи є коментар відповіддю на інший, він додається або в загальний список коментарів, або до відповідного коментаря як вкладений - з невеликим відступом вправо для кращої ілюстративності(Дод. А, рис. А.6.2).

Крім того, функція replyUser додає функціонал відповіді на конкретний коментар, прикріплюючи до кожної кнопки відповіді обробник подій replyComment. Для зручності цей обробник вставляє ім'я користувача та ідентифікатор коментаря в форму, щоб позначити відповідь. Після створення коментаря форма очищається, а кнопка надсилання знову стає активною.

- ratings.js

Файл ratings.js додає функціонал для оцінювання публікацій - спочатку він вибирає всі елементи з класом rating-buttons і додає до кожного обробник події, що реагує на натискання кнопки оцінювання. Після натискання кнопки отримуються значення оцінки та ідентифікатор публікації через атрибути data-value і data-post. Ці значення додаються до об'єкта FormData, який потім передається на сервер за допомогою fetch для обробки запиту на зміну рейтингу. Запит надсилається на ендпоінт /blog/post/<postId>/rating/ методом POST, зокрема з використанням заголовків для захисту від CSRF-атак.

У відповідь сервер надсилає оновлене значення загального рейтингу публікації, яке замінює на сторінці попереднє значення, відображене у відповідному елементі. Якщо виникає помилка під час процесу, вона виводиться в консоль для відстеження(Дод. А, рис. А.6.3).

2.3.7. Кореневий модуль email_service(мікросервіс email_service)

- settings.py

Даний файл settings.py конфігурує мікросервіс, який виконує функції, пов'язані з відправкою електронної пошти. У списку INSTALLED_APPS у нас наявні основні додатки разом із додатком email_sender, що відповідає за обробку електронних повідомлень. Завданням набору MIDDLEWARE є управління

сесіями, автентифікація користувачів і обробка повідомлень — забезпечення безпеки загалом. ROOT_URLCONF вказує на файл конфігурації URL-маршрутів мікросервісу, а налаштування шаблонів у TEMPLATES дозволяють використовувати вбудовані контекстні процесори та файли шаблонів із директорії templates. Для бази даних використовується sqlite3, яка зберігає дані локально у файлі db.sqlite3(тому що на даному етапі нічого серйознішого для цього додатку не треба — він має лише одну модель-лог)

Статичні файли визначає змінна STATIC_URL. Параметр DEFAULT_AUTO_FIELD вказує на тип поля первинного ключа за замовчуванням. Налаштування для відправки електронної пошти включають EMAIL_BACKEND, а також SMTP-сервер, порт, TLS-захист і облікові дані для доступу до сервісу Gmail(емейл і пароль на скріншоті приховані в цілях безпеки).

У конфігурації Celery вказано URL-адресу брокера повідомлень RabbitMQ для асинхронних задач, а також формат даних для передачі — JSON, що дозволяє серіалізувати завдання і приймати їх у JSON-форматі(Дод. А, рис. А.7.1).

- celery.py

В даному файлі спочатку встановлюється значення змінної середовища для правильного доступу до налаштувань Django, що дозволяє Celery отримувати конфігурацію з цього мікросервісу. Потім створюється об'єкт app з назвою email_service, що є екземпляром Celery, і він налаштовується з використанням параметрів з Django-конфігурації, де усі налаштування Celery отримуються з префіксом CELERY. Метод autodiscover_tasks() автоматично знаходить і реєструє всі завдання, визначені у додатках Django, що дозволяє легко інтегрувати обробку завдань у мікросервіс. Тобто відбувається все те саме, що було й в цьому мікросервісі-блозі(Дод. А, рис. А.7.2)

2.3.8. Модуль email_sender(мікросервіс email_service)

- models.py

Файл models.py має модель EmailLog, яка є, по суті, просто логом відправлених електронних листів. Модель має чотири основні поля: subject для

зберігання теми листа, recipient для електронної пошти отримувача, message для тексту листа та status для зберігання статусу відправки (наприклад, успіх або помилка). Поле timestamp зберігає дату та час надсилання листа і автоматично заповнюється поточним часом за допомогою timezone.now.

У методі __str__ реалізовано зручне представлення об'єкта, яке дозволяє при перегляді в адміністративній панелі Django або в консолі побачити інформацію про отримувача листа та його статус(Дод. А, рис. А.8.1).

- tasks.py

Файл tasks.py містить три фонових завдання для надсилання електронних листів через Celery.

Перший метод з назвою send_contact_email_message_task, відповідає за надсилання електронного листа через форму зворотного зв'язку - після отримання необхідних параметрів (тема, вміст, IP користувача, його електронна пошта та нікнейм), завдання формує повідомлення за допомогою шаблону HTML. А потім намагається надіслати листа на адресу, вказану в налаштуваннях, і записує результат в базу даних у модель EmailLog(Дод. А, рис. А.8.2).

Друге завдання, send_approval_email_message_task, надсилає листа автору поста після того, як його публікацію було схвалено. Параметри завдання включають адресу отримувача та назву поста. Лист також генерується за допомогою шаблону HTML і відправляється на вказану адресу. Статус відправлення також записується у EmailLog(Дод. А, рис. А.8.3).

Третє завдання, send_new_post_email_message_task, надсилає повідомлення підписникам про новий пост на вебсайті. Завдання приймає список отримувачів, ім'я автора та назву поста, генерує відповідний HTML лист і намагається надіслати його кожному отримувачу. Статус кожного відправленого листа зберігається в EmailLog після виконання завдання(Дод. А, рис. А.8.4).

В усіх трьох завданнях обробка помилок здійснюється через блоки try-except, і результат відправки листа (успіх або невдача) фіксується в моделі журналу EmailLog.

- send_approval_email.html

Файл send_approval_email.html є просто шаблоном електронного листа, який повідомляє автора поста про успішне схвалення його публікації. Він використовує механізм шаблонів Django для генерації контенту і містить просте повідомлення без використання ескейпингу(застосовується тег {% autoescape off %}), що дозволяє вставляти текст без автоматичного перетворення спеціальних символів. Це може нести певну небезпеку, якби це була вхідна форма, але в даному кейсі нам немає про що переживати.

У листі відображається привітання, в якому вказується отримувач (змінна recipient). Потім повідомляється, що його пост з назвою post_name, успішно схвалено(Дод. А, рис. А.8.5).

- send_feedback_email.html

Файл send_feedback_email.html є шаблоном листа адміністратору, який містить інформацію про повідомлення та сам текст того повідомлення, яке було надіслане через форму зворотного зв'язку. Шаблон також використовує механізм шаблонів Django для генерації змісту листа, при цьому автоматичне екранізування HTML символів, як в попереднього шаблону, вимкнено за допомогою {% autoescape off %}.

Лист починається з привітання "Hi, Administrator", далі вказується, що повідомлення було надіслане через "Feedback Form". Після цього основний зміст повідомлення відображається через змінну content, яка передає текст, який залишив користувач у формі фідбеку.

Після цього надаються додаткові дані про відправника: IP-адреса (ip), email (user_email) та псевдонім користувача (user_nickname). Це дозволяє адміністратору отримати більш детальну інформацію про того, хто надіслав зворотний зв'язок, і за потреби звернутися за додатковою інформацію(Дод. А, рис. А.8.6).

- send_new_post_email.html

Файл send_new_post_email.html є шаблоном для надсилання електронного листа підписникам, коли автор публікує новий пост в блозі. Шаблон починається з привітання, далі повідомляється, що автор з іменем, яке передається через

змінну author_name, опублікував новий пост з назвою, що міститься в змінній post_name.

Це повідомлення повинне заохотити отримувача відвідати вебсайт, де можна прочитати новий матеріал улюбленого автора. Лист також пояснює, що отримувач є підписником цього автора, що пояснює отримання цього повідомлення(Дод. А, рис. А.8.7).

2.4. Висновок до розділу 2

У даному розділі було розглянуто створення веб-блогу, що складається з 2 мікросервісів, які забезпечують його функціональність.

Першим кроком було обрання мов програмування та технологій для реалізації проекту. Для цього було вибрано мови Python, JavaScript та фреймворк Django, оскільки використовуючи їх, можна доволі ефективно , легко і швидко створювати веб-додатки та мікросервіси. Для асинхронних задач було використано Celery з брокером RabbitMQ.

Проект складається з кількох ключових частин: основного додатку веб-блогу, і сервісу для відправки електронних листів (email_sender).

Було детально розглянуто структуру проекту - зокрема основні файли, такі як settings.py, celery.py, models.py, tasks.py, forms.py а також шаблони та javascript файли, які додають свій функціонал на стороні фронтенду. Кожен з цих файлів виконує важливу роль в роботі системи, а також робить систему ефективною та зручною для використання і, що ще більш важливо — зручною для подальшої розробки, адже все структуровано і логічно розбито по різним додаткам, модулям і папкам.

РОЗДІЛ 3
ПРЕЗЕНТАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-БЛОГУ

3.1. Підготовка до запуску проєкту локально

Враховуючи, що в даному проєкті буде працювати одночасно два мікросервіси(тобто будуть запущені два Django-проєкти), то й запускати потрібно і сервіс-блог, і сервіс-надсилач листів. Але окрім цього в обов'язковому порядку повинні бути також запущені Redis, Celery та RabbitMQ. При цьому для основного сервісу(блогу) запускається процес celery-виконавець(worker) та celery-надсилач завдань(beat), а для сервісу, який надсилає емейли - лише celery-виконавець(worker).

3.1.1 Мікросервіс-блог

Першим потрібно запустити сервер Redis, який є інстанцією бази даних в оперативній пам'яті, використовуючи команду redis-server(рис. 3.1):

```
(.venv) tigerk00@MacBook-Air-Kosta blog_project % redis-server
64031:C 13 Nov 2024 22:13:06.045 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
64031:C 13 Nov 2024 22:13:06.045 # Redis version=7.0.7, bits=64, commit=00000000, modified=0, pid=64031, just started
64031:C 13 Nov 2024 22:13:06.045 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
64031:M 13 Nov 2024 22:13:06.045 * monotonic clock: POSIX clock_gettime
```

Рис. 3.1. Запуск серверу Redis

Потім повинен запуститися сервер RabbitMQ, який є надійним брокером повідомлень для організації черг повідомлень у розподілених системах. Для цього можна використати команду rabbitmq-server(рис 3.2).

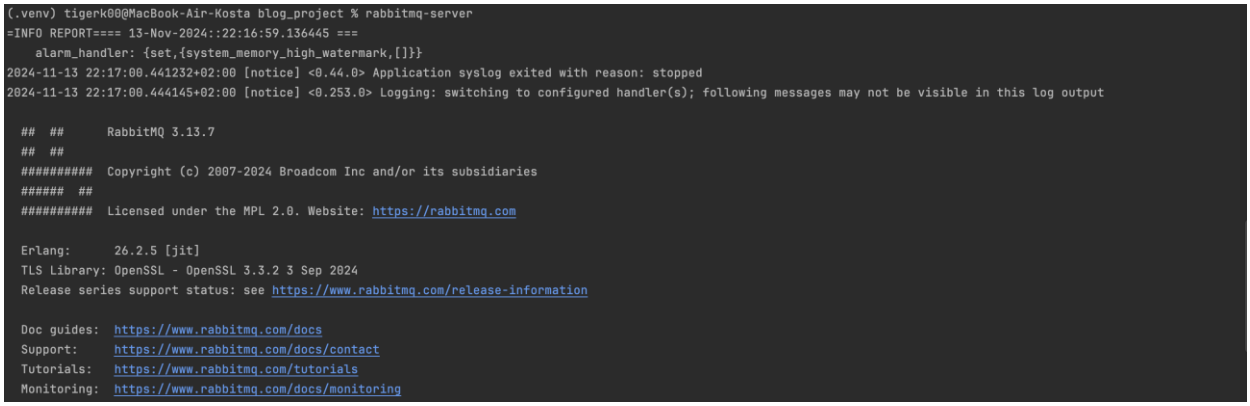

Рис. 3.2. Запуск серверу RabbitMQ

Далі треба запустити воркер (робітник) для обробки завдань Celery. Робиться це наступною командою - celery --app=blog_project worker --loglevel=info --pool=solo(рис. 3.3).

Опція worker в даній команді запускає воркер, який слухає чергу завдань і обробляє їх у фоновому режимі, без потреби чекати їх завершення зі сторони користувача. Параметр info виводить основну інформацію про процес виконання завдань. Параметр --pool=solo змушує воркер працювати в однопоточному режимі, що означає, що завдання обробляються послідовно, без паралельного виконання, що зазвичай використовується для тестування або в середовищах, де не підтримується багатопоточність, наприклад на Windows, що для локальної презентації цілком підходить.

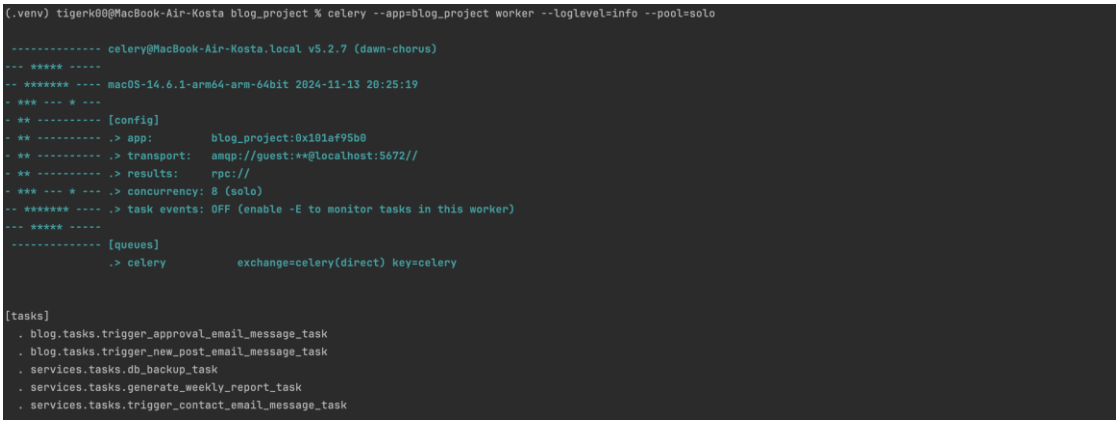

Рис. 3.3. Запуск воркера Celery для додатку блогу

До речі, при запускові воркера, він виведе в консолі список наявних celery-завдань, які визначені в проєкті - їх також можна побачити на рис. 3.3, під текстом “[tasks]”.

Далі необхідно запустити планувальник задач Celery Beat для додатку blog_project. Робиться це через команду celery -A blog_project beat -l info(рис 3.4).

Celery Beat[14] періодично додає заплановані завдання до черги, керуючи їхнім запуском за розкладом, який чітко визначений у конфігурації blog_project (файл settings.py). Параметр -l info встановлює рівень логування на info, що забезпечує виведення основної інформації про запуск і планування задач.

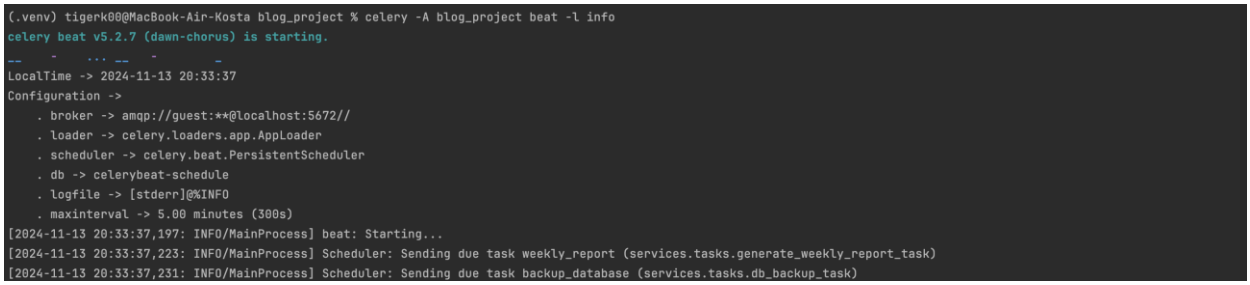

Рис. 3.4. Запуск планувальника задач Celery Beat

Ну і останнім можна запустити сам Django проєкт, використавши команду python manage.py runserver(рис. 3.5).

Рис. 3.5. Запуск проєкту Django(сервіс-блог)

3.1.2 Мікросервіс-надсилач емейлів

Тут потрібно діяти за схожою схемою, лишень потрібно запустити меншу кількість сервісів. Спершу запускається воркер Celery для цього мікросервісу, за

допомогою команди celery -A email_service worker —loglevel=info(рис. 3.6):

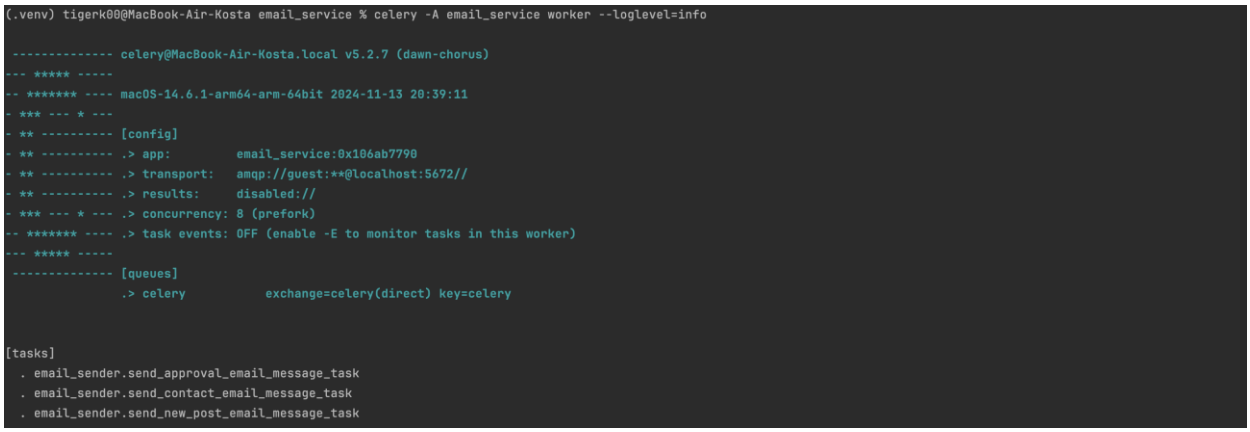

Рис. 3.6. Запуск воркера Celery для додатку, який надсилає емейли

Потім потрібно запустити сам Django проєкт даного сервісу, використовуючи команду python manage.py runserver 8001, де 8001 — порт, який він буде «слухати» (бо порт 8000 уже зайнятий веб-блогом)(рис. 3.7).

Рис. 3.7. Запуск проєкту Django(сервіс, який надсилає емейли)

3.2. Презентація веб-блогу

При першому відвідуванні додатку користувача перенаправляє на форму реєстрації, на якій є ключові поля, а також ReCAPTCHA, яка має допомогти уникнути спаму і заповнення форми ботами різних видів(рис. 3.8). Якщо будуть наявні якісь помилки при підтвердженні форми, їх буде відображено коло відповідного поля, для зручності відвідувача.

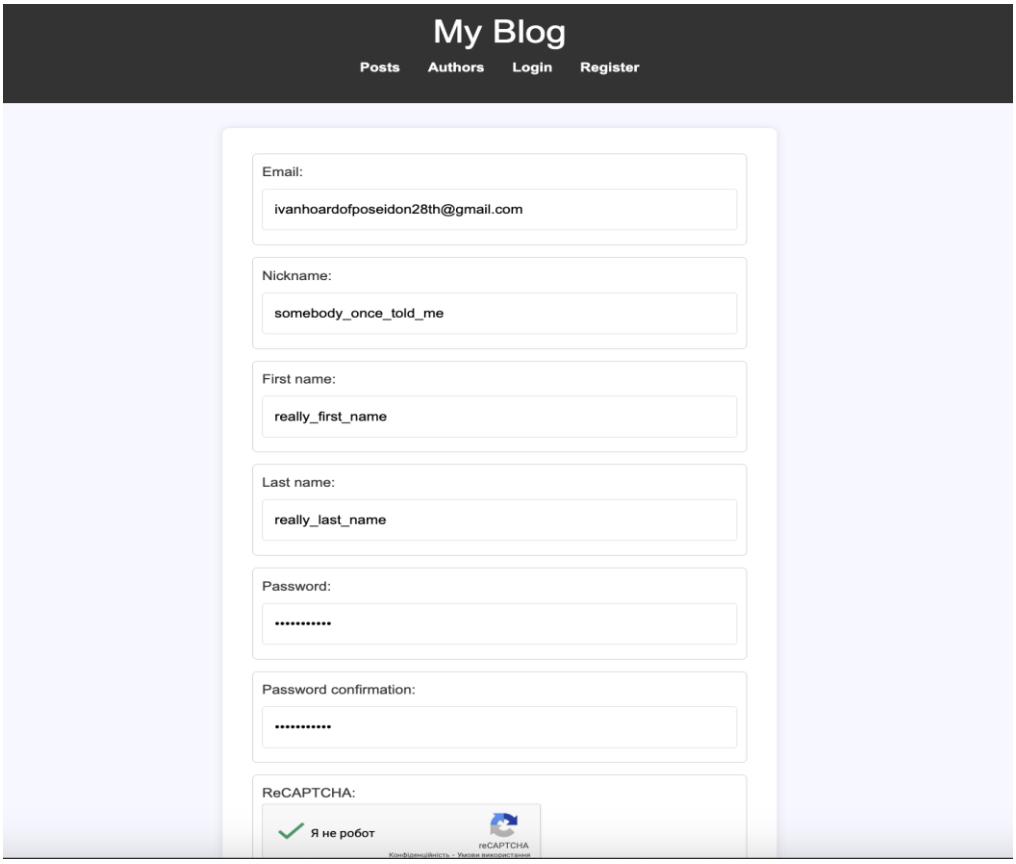

Рис. 3.8. Сторінка реєстрації з заповненими даними

Після успішної реєстрації його буде перенаправлено на частково заповнену сторінку, яка поки що в більшій мірі пустує, але на якій вже помітно поля, які можна модифікувати(рис 3.9).

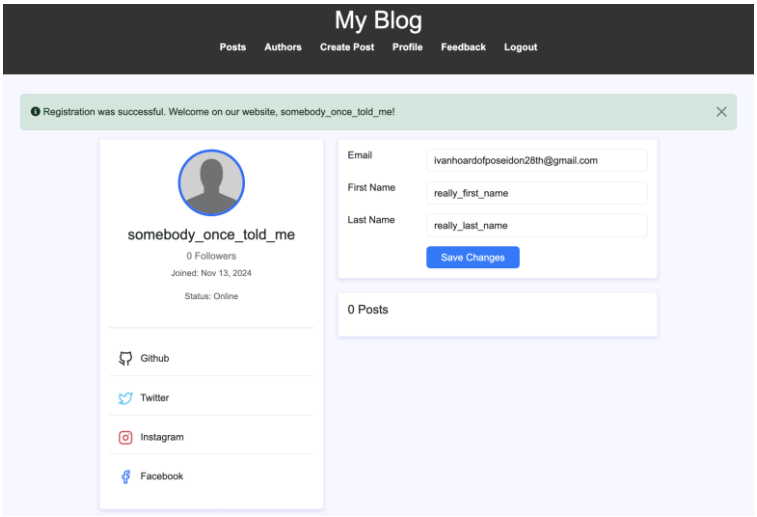

Рис. 3.9. Початковий вигляд сторінки користувача

Тепер він може заповнити профіль та додати свою фотографію(рис. 3.10).

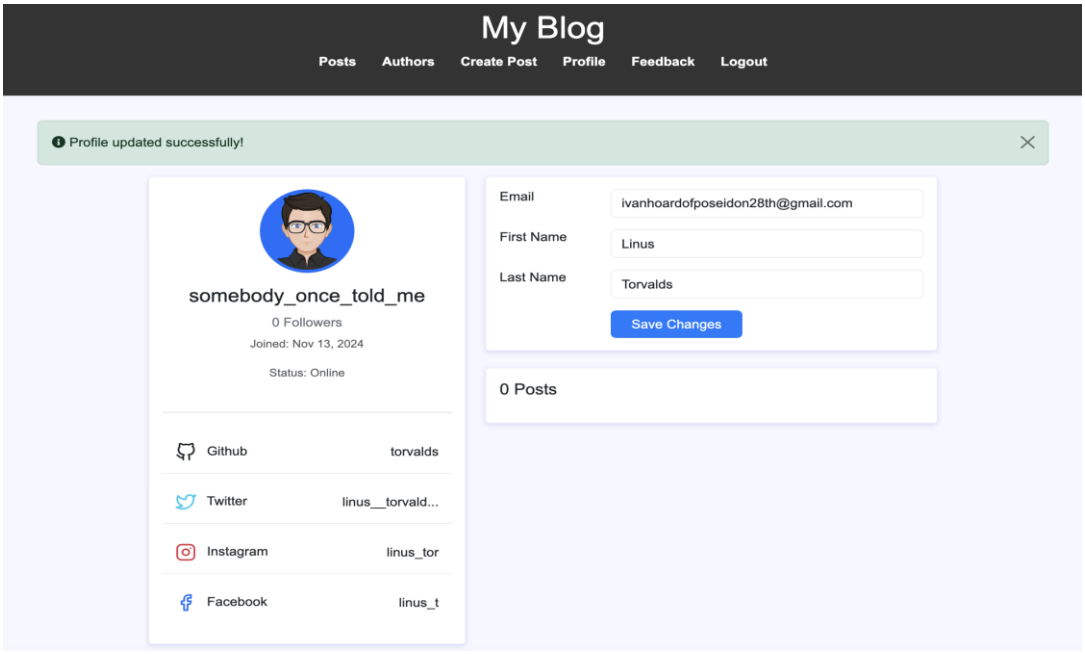

Рис. 3.10. Вигляд сторінки користувача із заповненою інформацією

Тепер, натиснувши на посилання “Posts” в хедері, є можливість переглянути список підтверджених постів інших користувачів(рис 3.11).

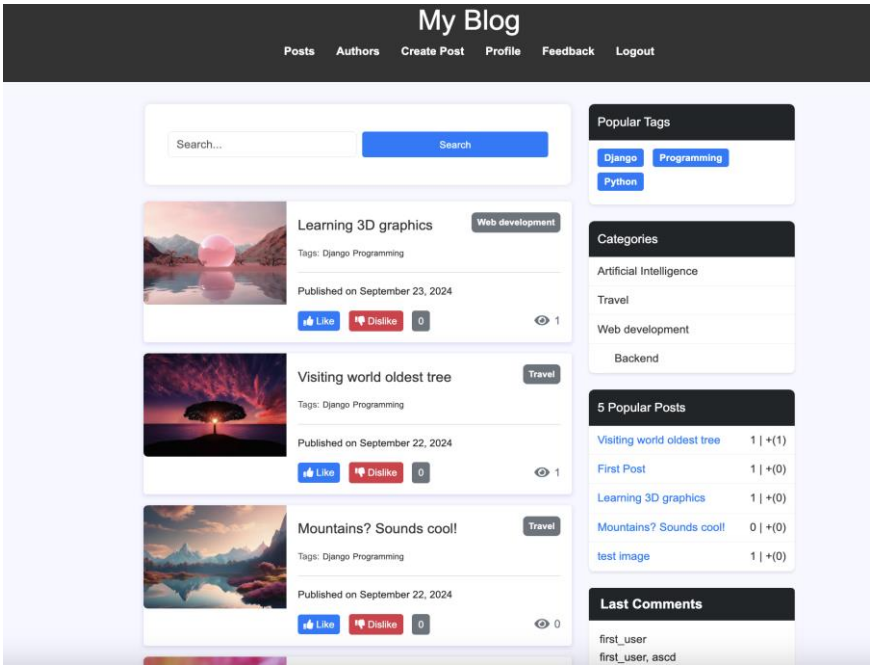

Рис. 3.11. Вигляд сторінки списку постів

У верхній частині екрану є чорна панель навігації з такими розділами: "Posts," "Authors," "Create Post," "Profile," "Feedback" та "Logout", а під ним вже якраз розташоване поле пошуку з білим текстовим полем і синьою кнопкою "Search."

Основний вміст складається з декількох постів(п’яти на сторінку, бо дана сторінка має пагінацію), кожен із яких містить зображення, заголовок, дату публікації, теги та категорію. При цьому, кожен пост має кнопки для "Like" і "Dislike," а також індикатор кількості переглядів у вигляді іконки з оком та числом поруч із нею.

На правій панелі знаходяться різні секції. Перша секція називається "Popular Tags," де відображаються теги "Django," "Programming" та "Python" — це ті теги, які поки що наявні в цій системі. Нижче знаходиться секція "Categories" з переліком категорій: "Artificial Intelligence," "Travel," "Web development" та "Backend"(остання категорія є підкатегорією "Web development", тому відображена з відступом).

Далі йде секція "5 Popular Posts," яка містить заголовки популярних постів із відповідною кількістю переглядів за весь час і за останній тиждень у дужках.

В останній секції, "Last Comments", відображаються останні 3 коментарі та користувачі, що їх написали. При натисканні на будь-яку категорію чи тег будуть відображені пости саме тієї категорії/тегу.

Для прикладу, можна обрати категорію подорожей(Travel), це видозмінить url, і будуть показані лише потрібні пости(рис 3.12).

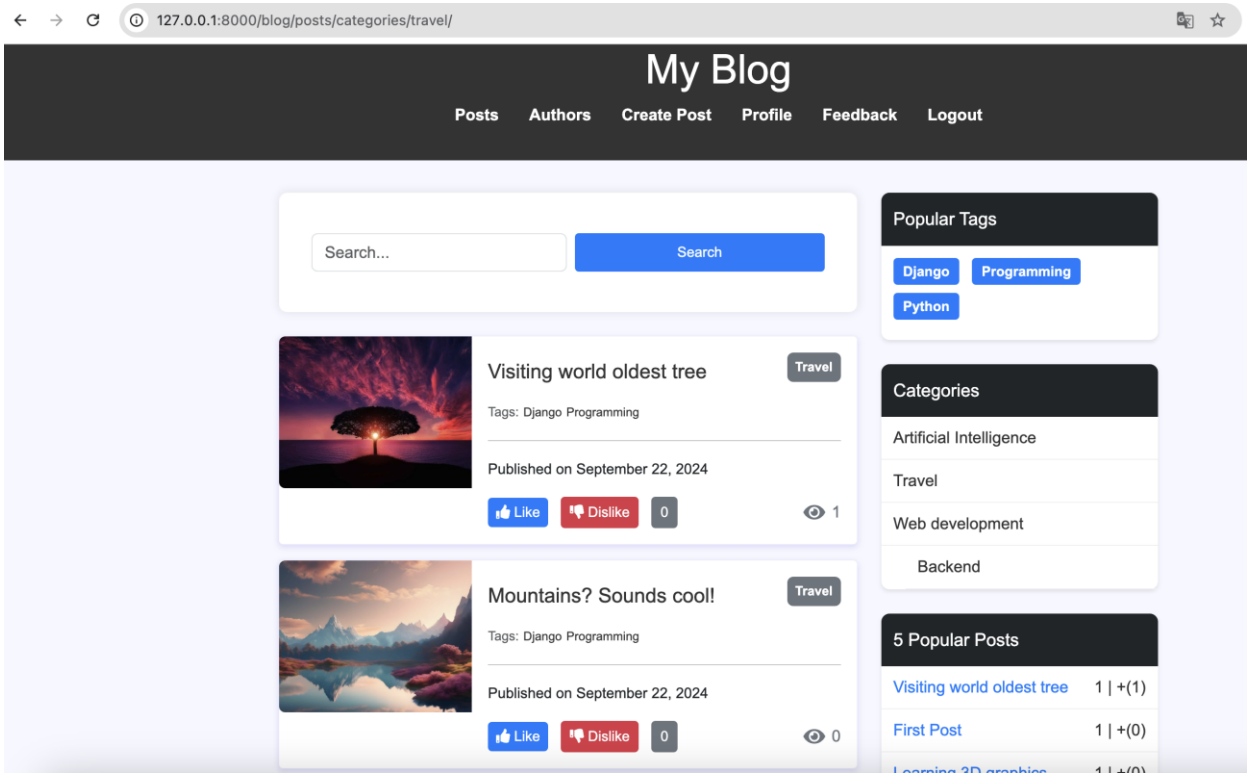

Рис. 3.12. Вибір постів по Категорії

Подібний результат буде отримано і якщо буде пошук постів по тегам, наприклад, по тегу “Django”(рис 3.13).

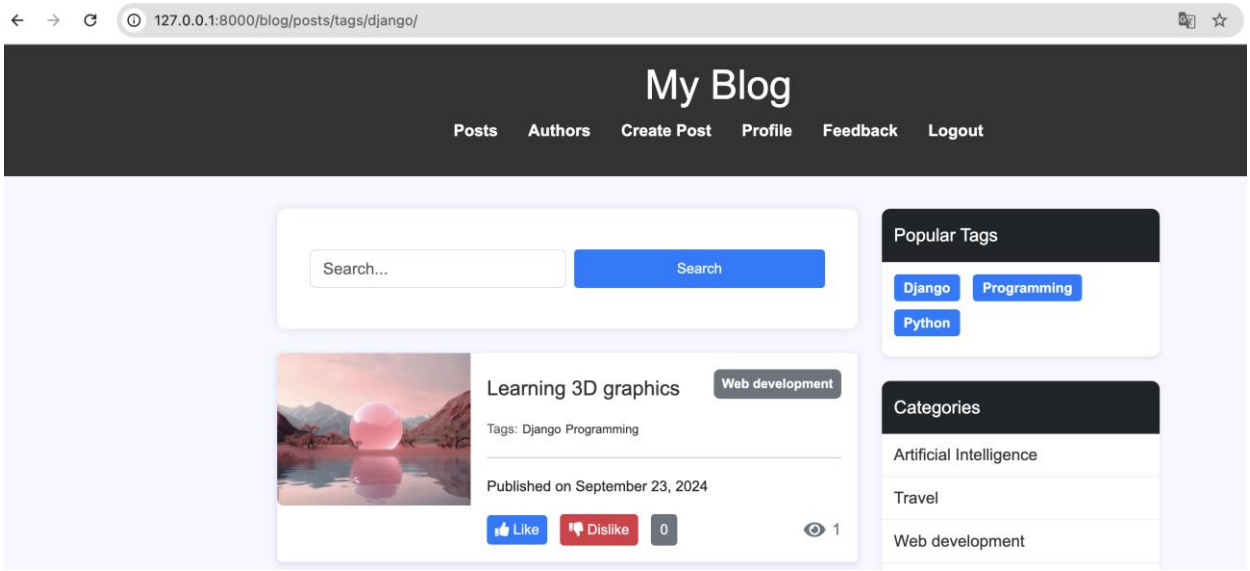

Рис. 3.13. Вибір постів по Тегу

Також можна використати поле пошуку, щоб знайти необхідний пост, наприклад, при введенні в пошук “Tree”, користувач отримає наступний результат(рис 3.14).

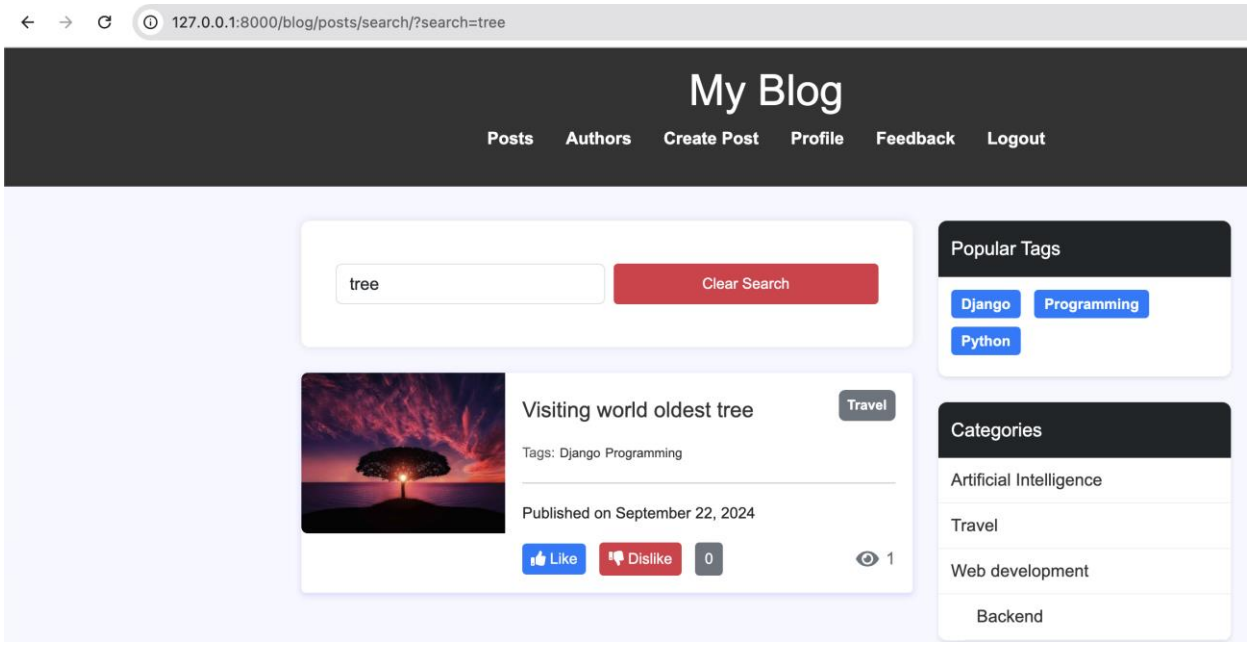

Рис. 3.14. Робота пошуку

При натисканні на кнопку вподобання можна підняти загальний рейтинг посту, наприклад на скриншоті знизу в нього зараз значення «1», бо наявне одне вподобання і немає жодного дизлайку(рис 3.15).

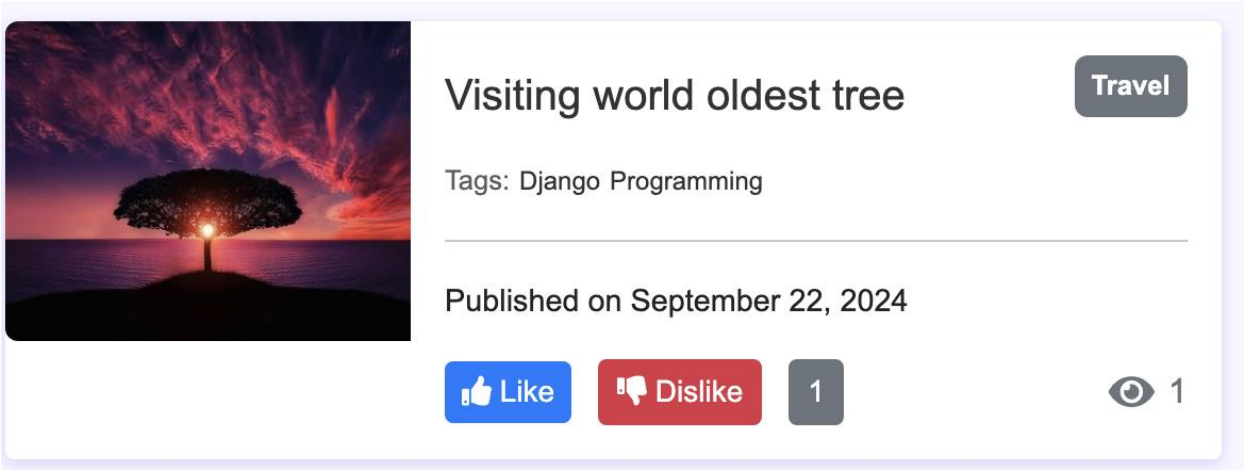

Рис. 3.15. Система лайк-дизлайк

Тепер можна переглянути, який контент має той пост, натиснувши на його назву(рис 3.16).

Рис. 3.16. Сторінка-контент посту

При цьому, горнувши нижче, можна побачити форму додавання коментаря і існуючі коментарі, але зараз(поки що) там їх немає, тож користувач може бути першим, хто його додасть(рис 3.17).

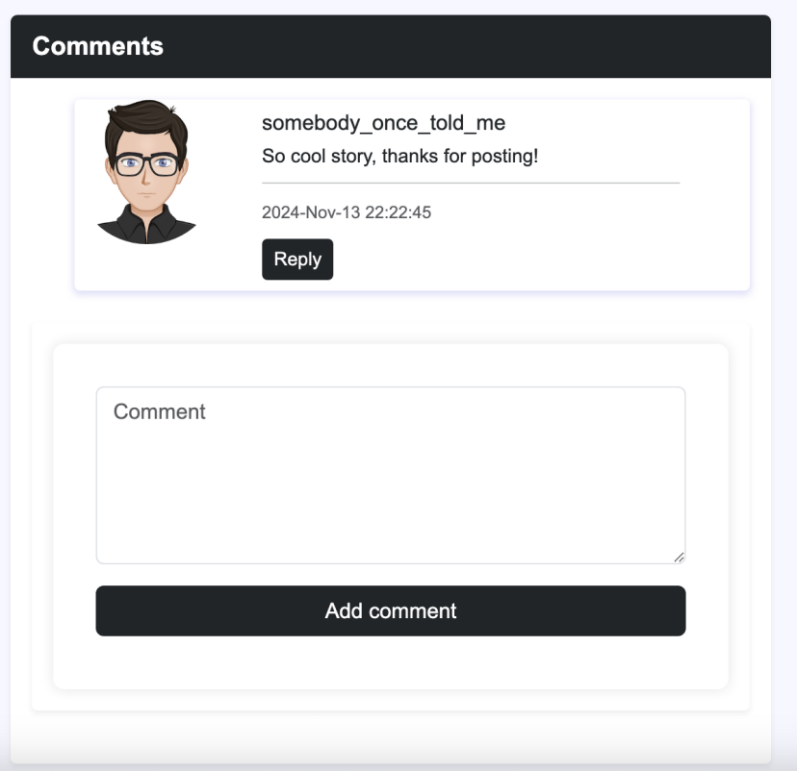

Рис. 3.17. Форма додавання нового коментаря і відображення існуючого

Тепер, в частині із останніми коментарями його коментар буде йти першим, і будь-хто, хто відвідає сайт, зможе його побачити, поки він буде в тому списку(рис 3.18).

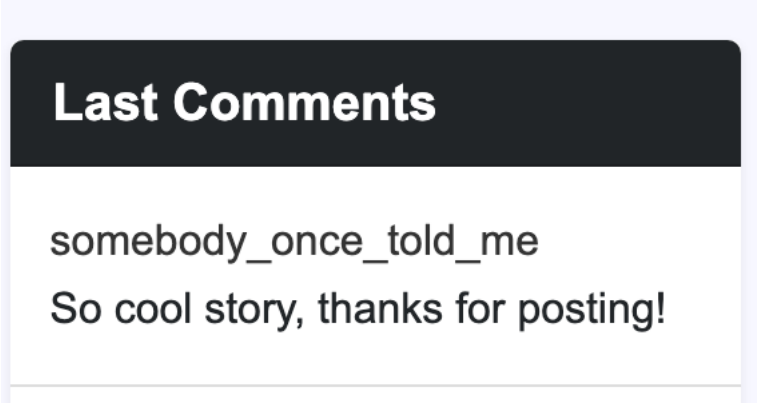

Рис. 3.18. Коментар в списку останніх коментарів

Якщо користувачу дуже сподобався пост іншої людини і він захоче побачити більше його контенту, чи підписатися на його нові публікації, то йому потрібно знайти сторінку тієї людини, а для цього спершу потрібно відкрити сторінку списку всіх користувачів блогу(рис 3.19).

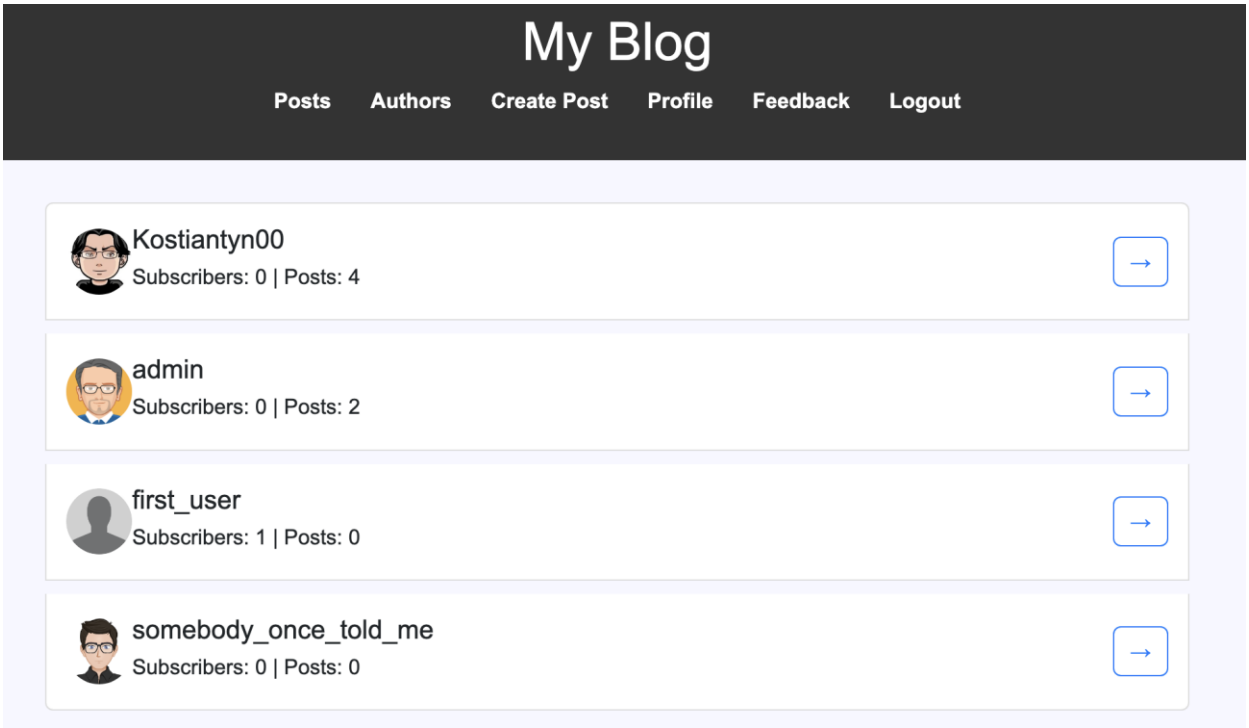

Рис. 3.19. Список користувачів

Натиснувши на стрілку напроти потрібного користувача, можна перейти на його сторінку, де є можливість побачити його інформацію, соціальні мережі, а також пости та кнопку підписки(рис 3.20).

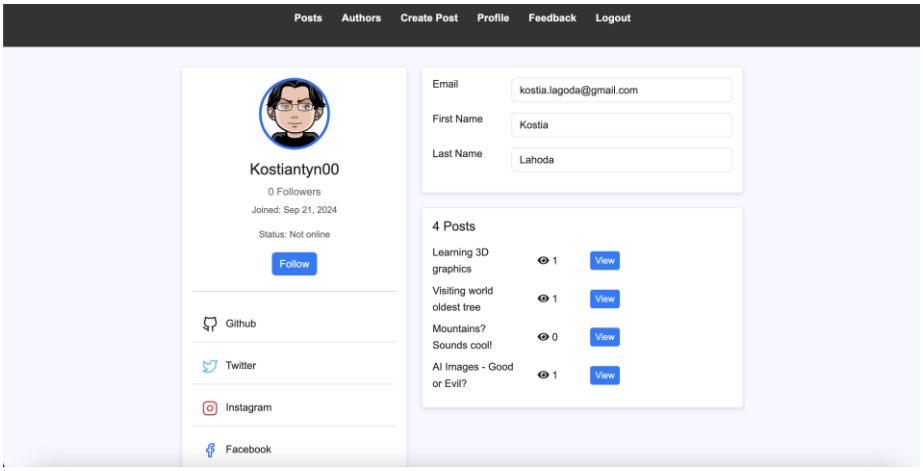

Рис. 3.20. Сторінка-профіль іншого користувача

Після підписки на того користувача, колір кнопки зміниться, а його кількість підписників збільшиться на одного(рис 3.21).

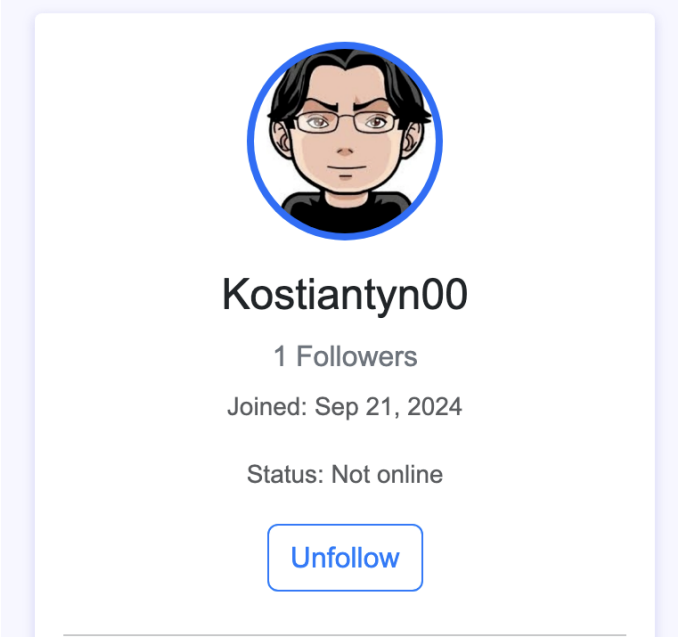

Рис. 3.21. Візуальні зміни після підписки

Тепер користувач зможе спробувати створити пост й самостійно. Для цього потрібно натиснути на посилання “Create Post” в хедері.

На цій сторінці можна побачити форму створення нового посту, яка включає в себе заголовок, категорію, теги, основний контент та зображення, яке буде відображатися на сторінці списку постів(рис 3.22).

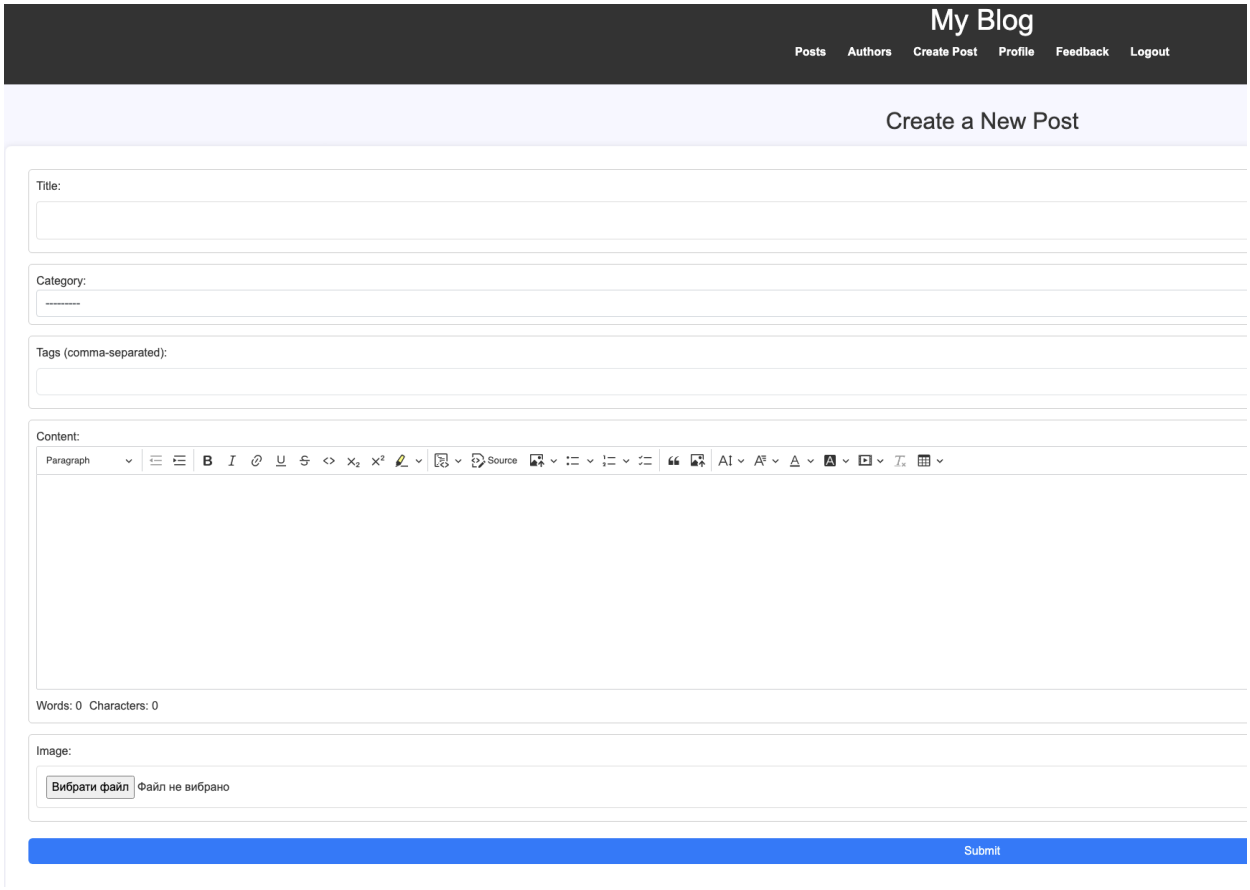

Рис. 3.22. Форма створення посту

Нехай користувач вирішив написати дуже коротко про бекенд-розробку блогів з використанням Django, Celery, Redis та RabbitMQ. При цьому, завдяки наявності CKEditor 5, у нього є хороший інструмент для форматування(рис 3.29).

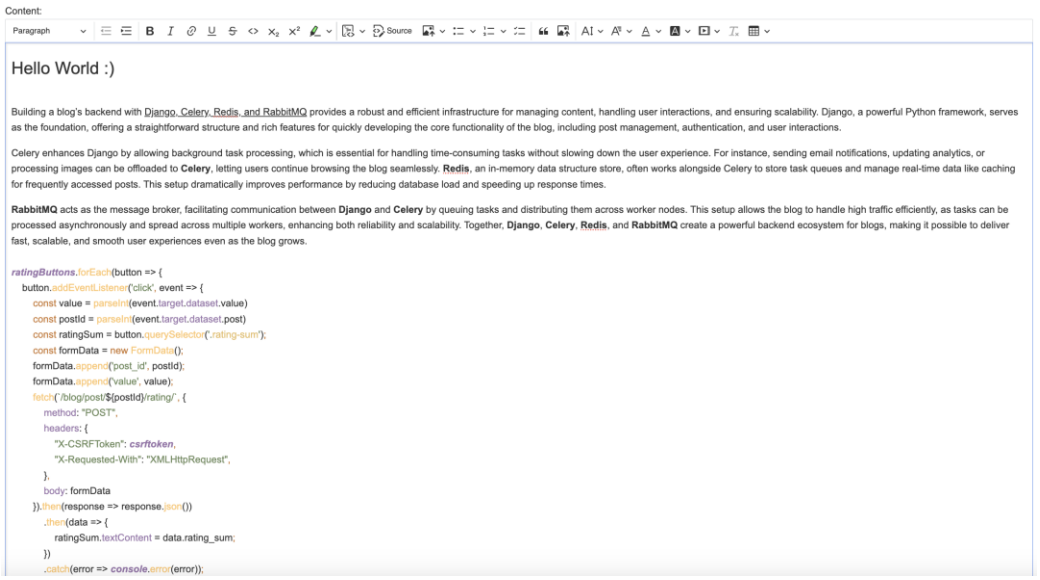

Рис. 3.23. Приклад заповненої форми CKEditor 5

При натисканні на кнопку підтвердження форми його перенаправить на особисту сторінку, де він зможе побачити повідомлення і новий пост в списку його постів(рис 3.24).

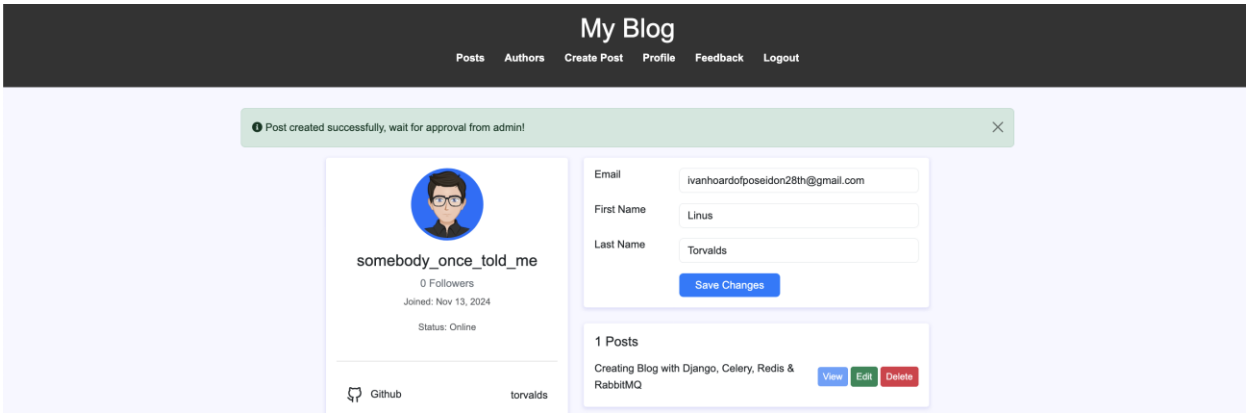

Рис. 3.24. Результат після створення посту

Проте, якщо він відвідає відразу після цього список постів, чи спробує шукати новостворений пост, його не буде знайдено — тому що потрібно, аби його схвалив адміністратор(-и) (рис 3.25).

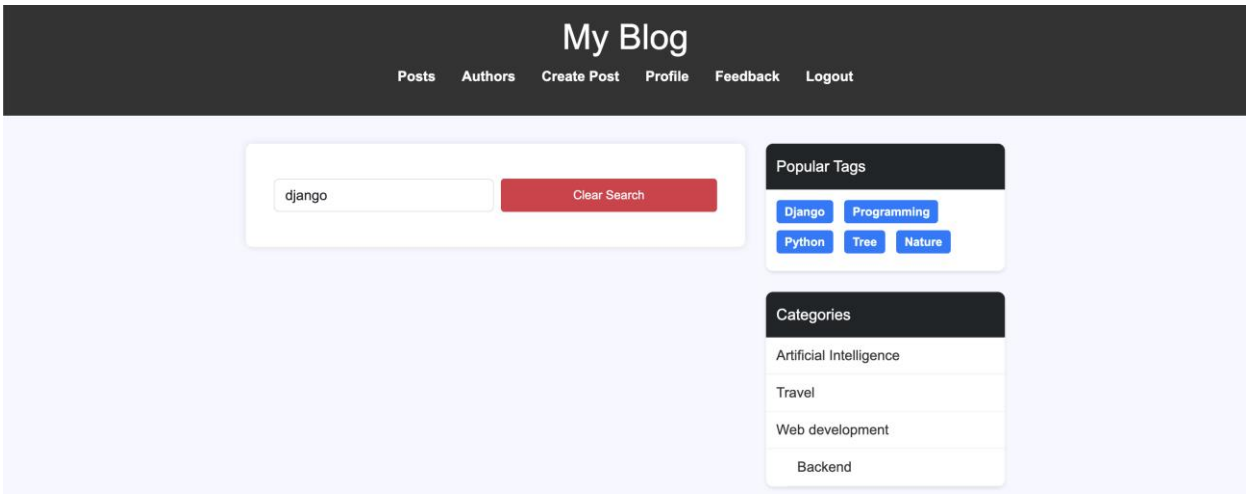

Рис. 3.25. Відсутність непідтверджених постів в пошуку

Для того, щоб його схвалити, адміністратору потрібно буде зайти в адмін-панель, відкрити сторінку даного посту і виставити галку в полі “Approved by Admin”(рис 3.26).

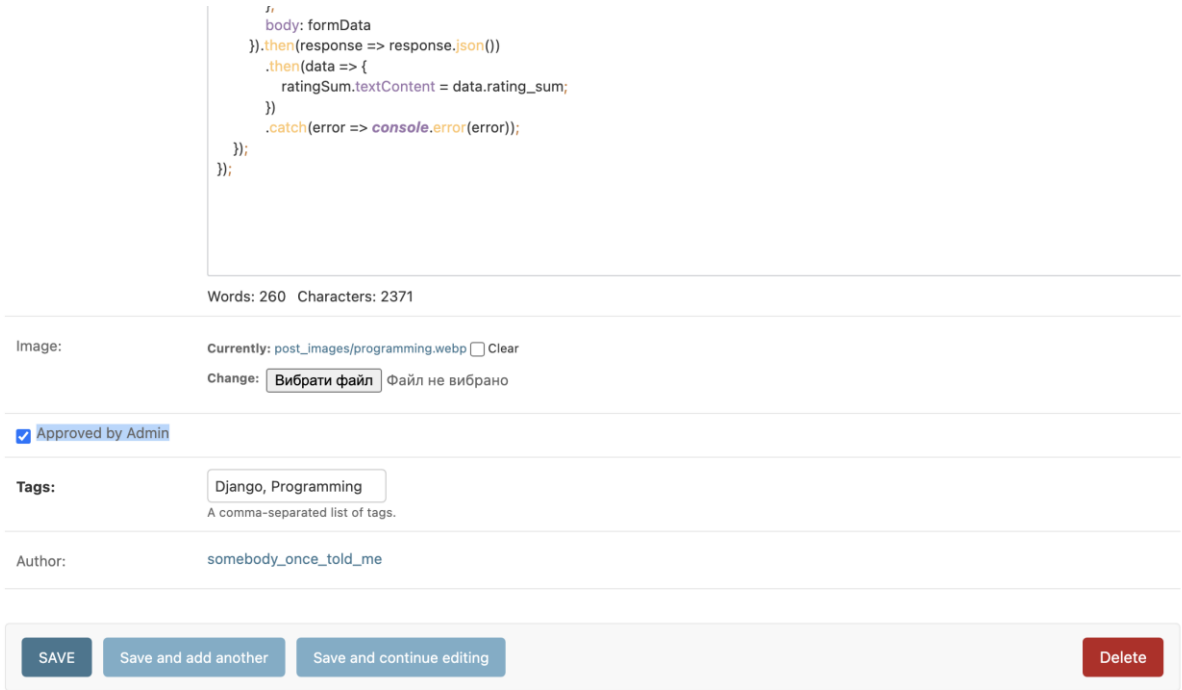

Рис. 3.26. Процес схвалення поста в адмін-панелі

Як тільки процес підтвердження буде завершений, діло буде за RabbitMQ, Celery та мікросервісом, який надсилає пошту, і це важлива частина даного проєкту — адже викликає завдання саме сервіс-блог(рис 3.27).

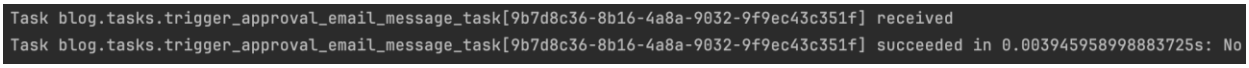

Рис. 3.27. Запуск Celery-завдання сервісом-блогом

А отримує і опрацьовує(шле емейл і створює відповідний запис в свої базі даних) — уже сервіс відправки пошти(рис 3.28).

Рис. 3.28. Отримання і опрацювання Celery-завдання сервісом-надсилачем

При цьому, автор отримає на свою пошту спеціальне повідомлення(рис 3.29).

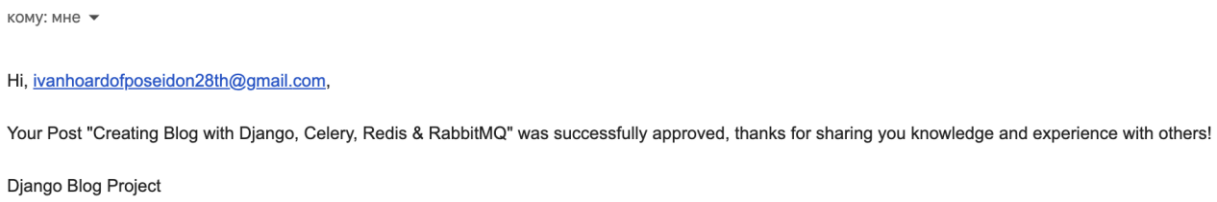

Рис. 3.29. Приклад повідомлення про схвалення посту адміністрацією

За подібним принципом працюють і інші типи емейлів, які надсилаються в даному проєкті, але про це згодом. Зараз потрібно переглянути пост, який уже підтверджений адміністрацією, і який помітно при пошукові(рис 3.30).

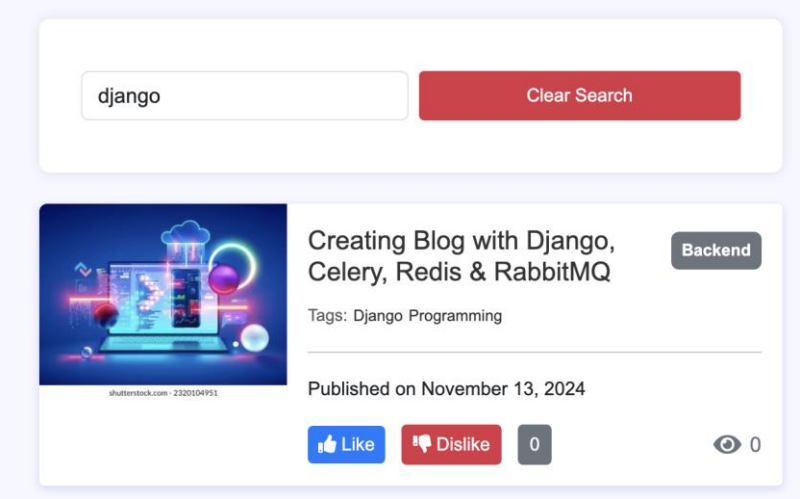

Рис. 3.30. Успішний пошук тепер уже підтвердженого посту

Пост наявний, і відображається коректно, але окрім цього, з’явився новий розділ під самим постом, який показує подібні пости по тегам, які можуть бути цікаві для користувача(“Similar Posts”). Його відображено на рис 3.31.

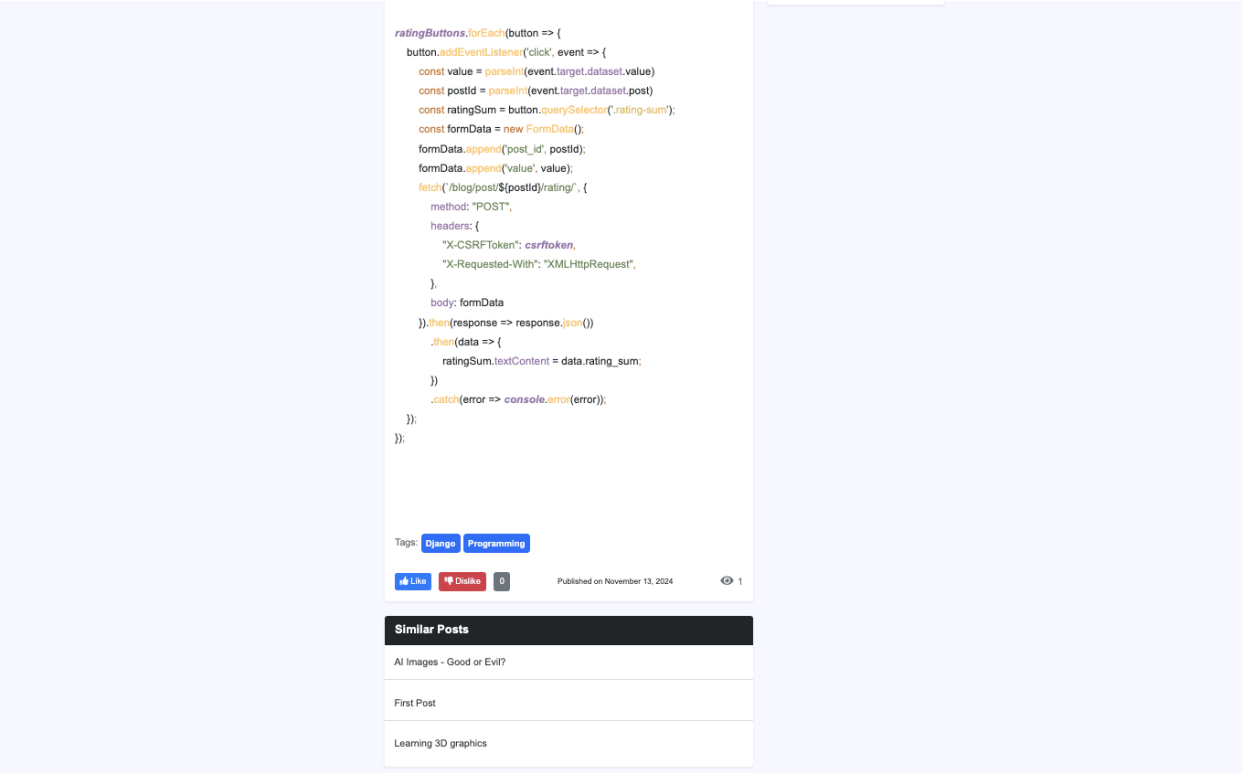

Рис. 3.31. Відображення подібних постів

Якщо ж було помічено якусь неточність, або є бажання просто підправити

текст/заголовок/теги/категорію посту, можна натиснути на кнопку “Edit” напроти назви даного посту на особистій сторінці і отримати таку можливість(рис 3.32).

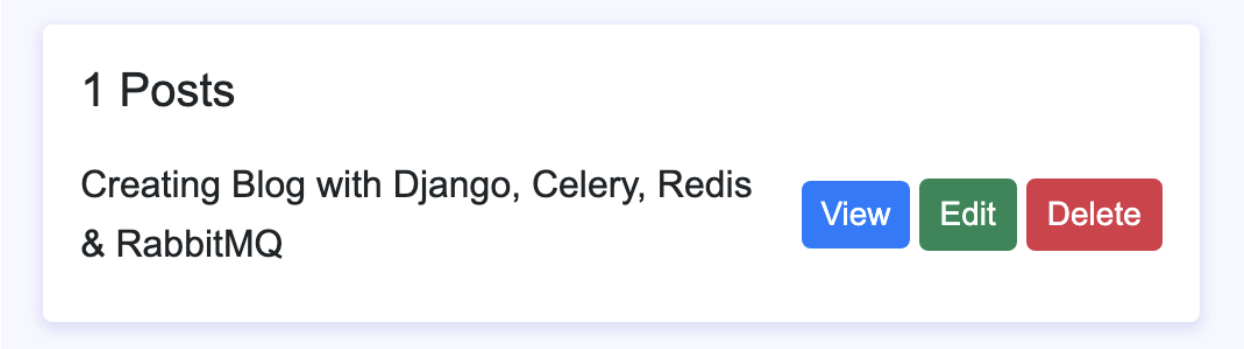

Рис. 3.32. Розміщення кнопок управління постом

Після оновлення його знову перенаправить на особисту сторінку, де на нього вже буде чекати повідомлення(рис 3.33).

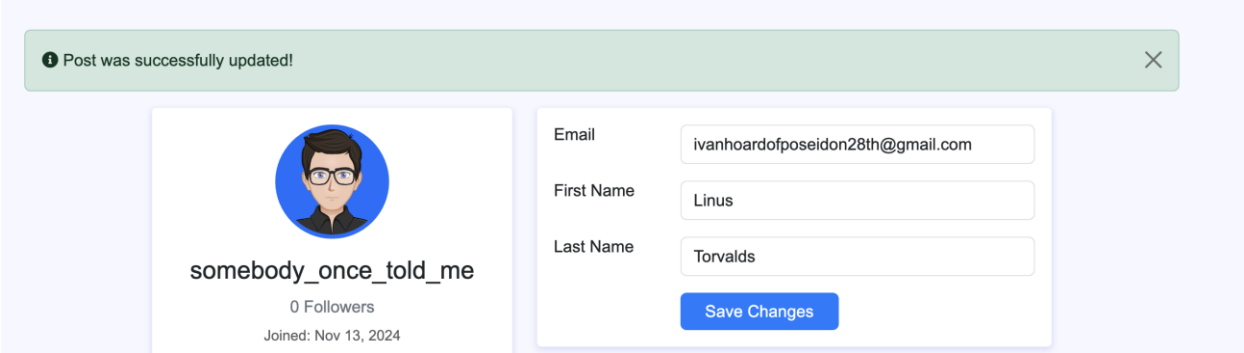

Рис. 3.33. Повідомлення про успішне оновлення посту

Якщо інший користувач, на якого він підписався, створить новий пост і його схвалить адміністрація, не тільки той користувач отримає повідомлення про схвалення, а також всі його підписники отримають спеціальні повідомлення(рис 3.34).

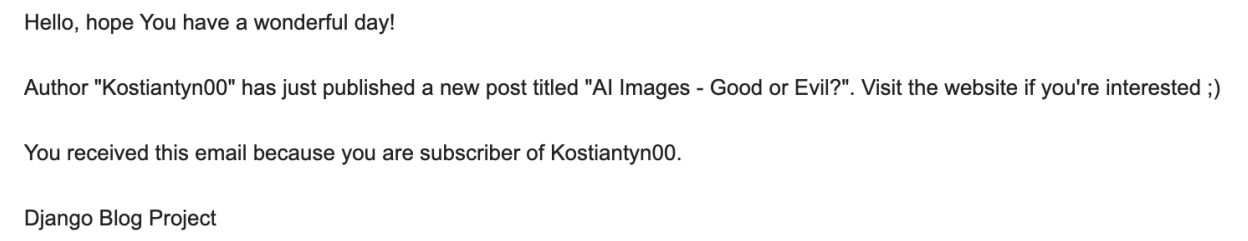

Рис. 3.34. Лист підписникам

Ну і насамкінець, в будь-якого авторизованого користувача є можливість написати листа адміністрації сайту - наприклад, про знайдену проблему чи просто якісь побажання, для цього треба перейти по посиланню “Feedback” в хедері сайту. Сторінку зворотнього зв'язку можна побачити на рис. 3.35.

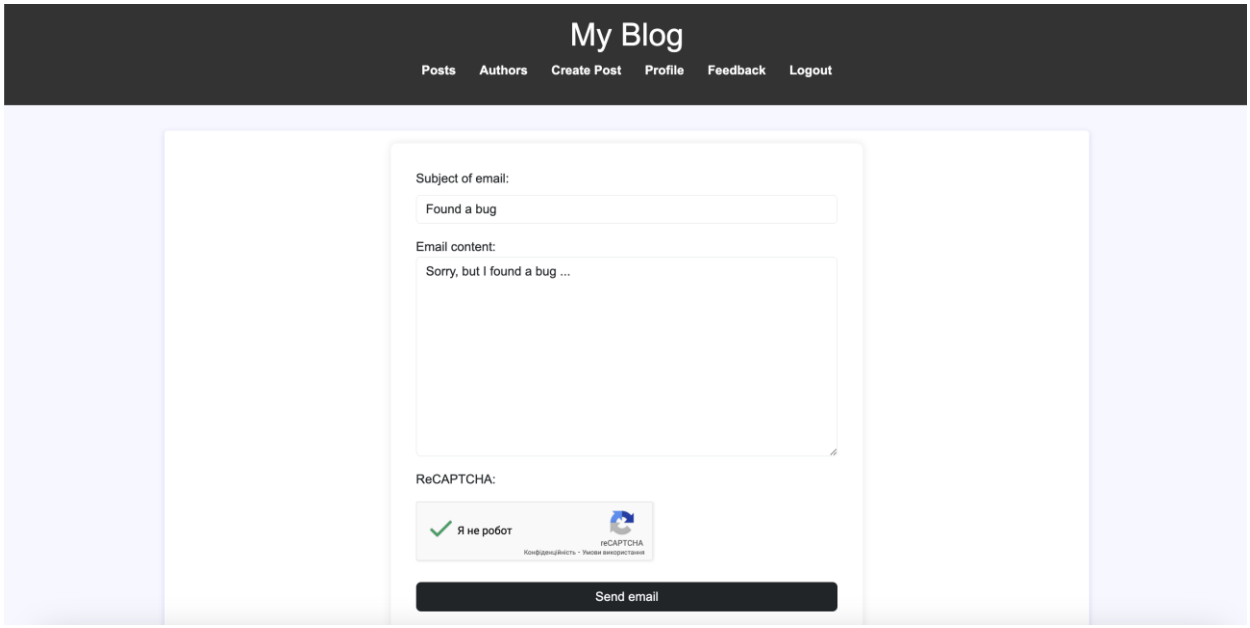

Рис. 3.35. Форма зворотнього зв’язку

При цьому, адміністрація сайту отримає таке повідомлення на свою пошту(рис 3.36).

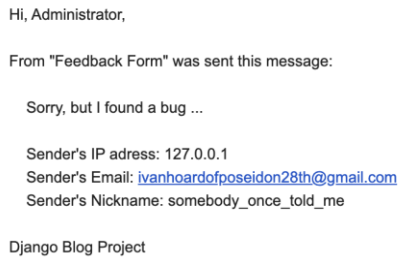

Рис. 3.36. Приклад листа-фідбеку

Що ж до самої адміністрації, то вона може однаково користуватися і звичайним сайтом, і адмін-панеллю - якщо їй буде потрібний ширший набір інструментів.

3.3. Можливі покращення

Незважаючи на те, що даний додаток повністю справляється із поставленим завданням, є ряд покращень, які можливі у майбутньому. Такі покращення сприятимуть удосконаленню програмного застосунку та додаванню нових функцій.

Перше, і найбільш корисне покращення — це використання Docker, який зможе зробити процес розробки і розгортання проєкту значно зручнішим, і 7 команд, які запускалися для розгортання даного проєкту, можна буде замінити на одну, або й автоматизувати їх.

Також можна покращити репорт, який генерується в даний момент та додати нові підвиди репортів, щоб мати точніше уявлення про роботу сервісу і дії користувачів — їх вподобання, активність та популярність тих чи інших авторів. При цьому генерацію репортів можна вивести в окремий мікросервіс, щоб він виконувався на задньому фоні, так само, як і надсилання емейлів. Додатково, в майбутньому можна залучити до обробки репортів і побудови передбачень штучний інтелект.

Далі можна запустити проєкт на віддаленому сервісі і відштовхуватися від побажань користувачів — додавання нічної теми, якихось спеціальних можливостей та функціоналу - це все доволі тяжко розгледіти розробнику, не

питаючи думки користувачів. Тому це необхідний крок для розвитку будь-якого проєкту.

Враховуючи, що це не проєкт, який розрахований на отримання прибутку, можна викласти код в відкритий доступ — наприклад, в GitHub. Це допоможе знайти однодумців, які захочуть допомогти з покращенням проєкту, зможуть знайти проблемні місця в безпеці і просто зможуть глянути на нього зі сторони, що буває дуже корисно. Загалом, відкритий вихідний код додатку дає можливість інтегрувати його з іншими проектами або інструментами, що теж надзвичайно корисно[16].

3.4. Висновок до розділу 3

У даному розділі було показано приклад запуску проєкту локально та презентація роботи розробленого веб-блогу. Для функціонування додатку необхідно запустити Django, Celery, Redis і RabbitMQ.

Після запуску додаток готовий до роботи. У користувача є можливість зареєструватися, редагувати свій профіль, переглядати пости інших користувачів, оцінювати їх а також створювати/редагувати/видаляти свої пости. Також наявна система тегів, категорій, коментарів та відображення подібних постів.

При підтвердженні посту, або при створенні посту іншою людиною, на якого він підписаний, користувач отримає відповідні електронні листи. Надсилатимуться вони допоміжним мікросервісом, який працює окремо, і має свої моделі, завдання Celery і свою окрему базу даних.

Адміністрація сайту має доступ до адмін-панелі, де може створювати, редагувати та видаляти різні сутності, а також підтверджувати пости користувачів.

В майбутньому є чимало можливих шляхів розвитку проєкту, серед яких додавання Docker, модифікація і додавання нових видів репортів, але на даному етапі даний проєкт цілком добре виконує поставлене завдання.

ВИСНОВКИ

У сучасних веб-додатках популярність мікросервісної архітектури значно зросла через її здатність забезпечити гнучкість, масштабованість і стійкість систем. Це особливо важливо для проєктів, які передбачають інтенсивне використання фонових задач або зіштовхуються з великими навантаженнями. У таких випадках традиційні монолітні додатки часто не здатні ефективно справлятися з усіма вимогами, що виникають під час роботи з великою кількістю одночасних запитів або ресурсомісткими операціями. Мікросервісна архітектура дає змогу розподіляти навантаження між численними автономними компонентами, що дозволяє досягти кращої стабільності та ефективності, а також забезпечити легкість масштабування системи.

У даній кваліфікаційній роботі розглянуто процес розробки мікросервісного веб-додатку «Блог», який реалізує асинхронну обробку запитів за допомогою фреймворка Django, системи задач Celery та брокера повідомлень RabbitMQ. Такий підхід дозволяє значно підвищити ефективність обробки задач, зокрема, тих, що потребують тривалого часу на виконання або взаємодії з іншими системами, наприклад, відправка електронних листів, обробка великих обсягів даних або створення резервних копій.

Одним із ключових рішень при розробці цього додатку стало використання фреймворка Django, який обрано за його простоту, надійність і можливість швидкого створення веб-додатків. Django надає чітку структуровану архітектуру, що дозволяє розробникам легко організувати код, забезпечити зручність роботи з базами даних і створювати RESTful API. Важливою особливістю Django є його екосистема, яка включає численні сторонні бібліотеки та інструменти для швидкої інтеграції додаткових функціональних можливостей, що значно полегшує процес розробки. Бібліотеки для автентифікації, безпеки та роботи з базами даних дозволяють зосередитись на реалізації основної логіки додатку, замість того, щоб вирішувати стандартні проблеми безпеки та роботи з даними.

Для реалізації асинхронних задач у проекті було обрано систему Celery.

Celery дозволяє виконувати задачі поза основним потоком виконання, що є особливо корисним для операцій, які займають багато часу і не потребують миттєвої реакції. Прикладом таких задач є обробка великих обсягів даних, відправка повідомлень електронною поштою, а також інші фонові операції, що можуть бути відкладені або виконуватися без блокування основного користувацького інтерфейсу. Використання Celery знижує навантаження на основну систему та дозволяє значно покращити продуктивність додатку в цілому.

Брокер повідомлень RabbitMQ був обраний для забезпечення комунікації між компонентами Django та Celery. RabbitMQ служить посередником, через який відправляються і приймаються повідомлення між мікросервісами. Така архітектура дозволяє ефективно масштабувати додаток, оскільки RabbitMQ може розподіляти навантаження між різними робочими процесами і забезпечувати надійну доставку повідомлень навіть у випадку збою однієї зі складових системи. RabbitMQ здатний обробляти велику кількість повідомлень одночасно, що дозволяє зберегти стабільність системи при високих навантаженнях.

У процесі розробки було детально описано кожен етап створення мікросервісного веб-додатку. Зокрема, було розглянуто налаштування RabbitMQ для забезпечення ефективної взаємодії між мікросервісами, а також інтеграція Celery з Django для виконання асинхронних задач. Ці етапи потребують уважного налаштування і тестування, оскільки будь-які помилки в конфігурації можуть призвести до збоїв у доставці задач або втрати повідомлень. З цією метою були розроблені тести, що дозволили перевірити стабільність і коректність роботи кожного компонента додатку, а також забезпечити безперебійну роботу при великих навантаженнях.

Крім того, було проведено тестування продуктивності, яке показало, що система може ефективно справлятися з великою кількістю одночасних запитів і задач, забезпечуючи належну швидкість обробки та мінімальний час затримки для кінцевих користувачів. Для забезпечення високої доступності та надійності

системи використовувалися додаткові механізми для моніторингу і логування, що дозволяють оперативно виявляти й усувати проблеми в роботі додатку.

Завершальним етапом було проведення презентації, яка продемонструвала, як саме додаток справляється з поставленими завданнями. Презентація включала показ роботи веб-додатку в реальному часі.

Таким чином, реалізований мікросервісний веб-додаток «Блог» довів свою ефективність у контексті асинхронної обробки задач, використовуючи такі потужні інструменти, як Django, Celery і RabbitMQ. Цей підхід не лише дозволяє значно покращити продуктивність та масштабованість додатку, але й робить його більш стійким до збоїв і зручним для подальшого розвитку. Робота також показала важливість використання мікросервісної архітектури для сучасних веб-додатків, де асинхронна обробка та масштабованість є критичними для ефективного функціонування системи в умовах високих навантажень.

В майбутньому можна розглянути розширення функціоналу додатку шляхом інтеграції нових мікросервісів, вдосконалення механізмів обробки задач і підвищення надійності системи за допомогою технологій, таких як контейнеризація (Docker. Це дозволить ще більше оптимізувати процеси обробки та управління додатком, забезпечивши його подальшу еволюцію відповідно до вимог ринку та технологічних тенденцій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A Sixteen Year Old's Look at the Net — Links (links.net) [Електронний ресурс] – Режим доступу до ресурсу: https://www.links.net/vita/web/story.html (дата звернення: 25.11.2024)
2. Wait But Why — Wait But Why (waitbutwhy.com) [Електронний ресурс] – Режим доступу до ресурсу: https://waitbutwhy.com/ (дата звернення: 25.11.2024)
3. Luciano Ramalho. Fluent Python. Clear, Concise, and Effective Programming. 2nd Edition / Luciano Ramalho. – США: Видавництво O'Reilly Media, 2022. – 234 с.
4. David Flanagan. JavaScript. The Definitive Guide. Master the World's Most-Used Programming Language 7th Edition / David Flanagan. – США: Видавництво O'Reilly Media, 2020. – 706 с.
5. Django – Офіційна документація (djangoproject.com)[Електронний ресурс] – Режим доступу до ресурсу: https://docs.djangoproject.com/ (дата звернення: 25.11.2024)
6. Асинхронність і Celery в Python — FoxMinded (foxminded.ua) [Електронний ресурс] – Режим доступу до ресурсу: https://foxminded.ua/celery-shcho-tse/ (дата звернення: 25.11.2024)
7. RabbitMQ — Wikipedia (uk.wikipedia.org) [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/RabbitMQ (дата звернення: 25.11.2024)
8. Створення сайтів і додатків з базою даних на REDIS — Brander (brander.ua) [Електронний ресурс] – Режим доступу до ресурсу: https://brander.ua/technologies/redis (дата звернення: 25.11.2024)
9. How to create custom django-admin commands — Django (docs.djangoproject.com) [Електронний ресурс] – Режим доступу до ресурсу: https://docs.djangoproject.com/en/5.1/howto/custom-management-commands/ (дата звернення: 25.11.2024)

10. Building Scalable Applications with Django, Celery, and RabbitMQ: A Step-by-Step Guide – Medium (medium.com) [Електронний ресурс] – Режим доступу до ресурсу: http://surl.li/cqpypx (дата звернення: 25.11.2024)

11. How to use Celery with Django – Medium (medium.com) [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/django-unleashed/how-to-use-celery-with-django-c4c341997704 (дата звернення: 25.11.2024)

12. django-ckeditor-5 — PyPI (pypi.org) [Електронний ресурс] – Режим доступу до ресурсу: https://pypi.org/project/django-ckeditor-5/EF EPI 2022 – EF English Proficiency Index [Електронний ресурс] – Режим доступу до ресурсу: https://www.ef.com/wwen/epi/ (дата звернення: 25.11.2024)

13. What is AJAX? — W3Schools (w3schools.com) [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3schools.com/whatis/whatis_ajax.asp (дата звернення: 25.11.2024)

14. Periodic Tasks — Celery (docs.celeryq.dev) [Електронний ресурс] – Режим доступу до ресурсу: https://docs.celeryq.dev/en/latest/userguide/periodic-tasks.html (дата звернення: 25.11.2024)

15. Microservices vs. monolithic architecture — Atlassian (atlassian.com) [Електронний ресурс] – Режим доступу до ресурсу: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (дата звернення: 25.11.2024)

16. Importance of Open-Source: How GitHub helps you in Contributing More? — Medium (medium.com) [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/analytics-vidhya/importance-of-open-source-how-github-helps-you-in-contributing-more-c7687715be3a (дата звернення: 25.11.2024)

17. The What, Why, and How of a Microservices Architecture — Medium (medium.com) [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9 (дата звернення: 25.11.2024)

18. Положення про кваліфікаційні роботи (проєкти) — Національний авіаційний університет (nau.edu.ua) [Електронний ресурс] – Режим доступу доресурсу: http://surl.li/dwhqah (дата звернення: 25.11.2024)

ДОДАТОК

Додаток А

Скриншоти файлів в модулях(додатках Django)

А.1 Модуль accounts

```python
class BlogUserManager(BaseUserManager):
    1 usage
    def create_user(self, email, nickname, password=None, **extra_fields):
        if not email:
            raise ValueError('The Email field must be set')
        if not nickname:
            raise ValueError('The Nickname field must be set')
        email = self.normalize_email(email)
        user = self.model(email=email, nickname=nickname, **extra_fields)
        user.set_password(password)
        user.save(using=self._db)
        return user

    def create_superuser(self, email, nickname, password=None, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)

        return self.create_user(email, nickname, password, **extra_fields)
```

Рис. А.1.1. файл models.py - модель BlogUserManager

```python
class BlogUser(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    nickname = models.CharField(max_length=30, unique=True)
    slug = models.SlugField(verbose_name='URL', max_length=255, blank=True, unique=True)
    profile_image = models.ImageField(upload_to='profile_images/', blank=True, null=True)
    first_name = models.CharField(max_length=30, blank=True)
    last_name = models.CharField(max_length=30, blank=True)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    date_joined = models.DateTimeField(default=timezone.now)

    # other social media fields
    github_username = models.CharField(max_length=50, blank=True, null=True)
    instagram_username = models.CharField(max_length=50, blank=True, null=True)
    facebook_username = models.CharField(max_length=50, blank=True, null=True)
    twitter_username = models.CharField(max_length=50, blank=True, null=True)

    # Following relationships
    followers = models.ManyToManyField( to: 'self', symmetrical=False, related_name='following', blank=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['nickname']

    objects = BlogUserManager()

    class Meta:
        ordering = ('nickname',)
        verbose_name = 'User'
        verbose_name_plural = 'Users'
```

Рис. А.1.2. файл models.py - модель BlogUser

```python
def register(request):
    if request.method == 'POST':
        form = BlogUserCreationForm( *args: request.POST, request.FILES)
        if form.is_valid():
            user = form.save()
            login(request, user)
            messages.success(request, message: f'Registration was successful. Welcome on our website, {user.nickname}!')
            return redirect( to: 'profile', user_id=user.id)
    else:
        form = BlogUserCreationForm()
    return render(request, template_name: 'accounts/register.html', context: {'form': form, 'title': 'Register'})
```

Рис. А.1.3. файл views.py - функція реєстрації

```python
def login_view(request):
    if request.method == 'GET':
        return render(request, template_name: 'accounts/login.html', context: {'form': LoginForm(), 'title': 'Log In'})

    form = LoginForm(request.POST)
    if form.is_valid():
        email = form.cleaned_data['email']
        password = form.cleaned_data['password']
        user = authenticate(request, username=email, password=password)
        if user:
            login(request, user)
            messages.success(request, message: f'Hi {email}, welcome back!')
            return redirect( to: 'profile', user_id=user.id)

    return render(request, template_name: 'accounts/login.html', context: {'form': form, 'title': 'Log In'})


/accounts/logout/  2 usages
def logout_view(request):
    logout(request)
    return redirect(reverse('login'))
```

Рис. А.1.4. файл views.py - функції логіну та логауту

```python
@login_required
def profile(request, user_id):
    profile_user = get_object_or_404(BlogUser, id=user_id)
    is_following = request.user.is_following(profile_user)

    if profile_user == request.user:
        if request.method == 'POST':
            form = BlogUserChangeForm(request.POST, request.FILES, instance=request.user)
            if form.is_valid():
                form.save()
                messages.success(request, message: 'Profile updated successfully!')
                return redirect( to: 'profile', user_id=user_id)
        else:
            form = BlogUserChangeForm(instance=request.user)

        return render(request, template_name: 'accounts/profile_edit.html', context: {
            'form': form,
            'title': 'My profile',
        })

    return render(
        request,
        template_name: 'accounts/profile_view.html',
        context: {
            'profile_user': profile_user,
            'is_following': is_following,
            'title': f'Profile of {profile_user.nickname}'
        }
    )
```

Рис. А.1.5. файл views.py - функція відображення-зміни профіля

```python
@login_required
@require_POST
def follow(request, user_id):
    user_to_follow = get_object_or_404(BlogUser, id=user_id)
    request.user.follow(user_to_follow)
    return JsonResponse({'success': True})

/accounts/unfollow/{user_id}/  2 usages
@login_required
@require_POST
def unfollow(request, user_id):
    user_to_unfollow = get_object_or_404(BlogUser, id=user_id)
    request.user.unfollow(user_to_unfollow)
    return JsonResponse({'success': True})
```

Рис. А.1.6. файл views.py - функції підписки та відписки від користувача

```
def user_list(request):
    users = BlogUser.objects.all()
    return render(request, template_name: 'accounts/user_list.html', context: {'users': users})
```

Рис. А.1.7. файл views.py - функція отримання списку користувачів

```
class LoginForm(forms.Form):
    email = forms.EmailField(max_length=65)
    password = forms.CharField(max_length=65, widget=forms.PasswordInput)
    recaptcha = ReCaptchaField(widget=ReCaptchaV2Checkbox, label='ReCAPTCHA')
```

Рис. А.1.8. файл forms.py – форма логіну

```
class BlogUserCreationForm(UserCreationForm):
    class Meta(UserCreationForm.Meta):
        model = BlogUser
        fields = ('email', 'nickname', 'first_name', 'last_name')

    recaptcha = ReCaptchaField(widget=ReCaptchaV2Checkbox, label='ReCAPTCHA')
```

Рис. А.1.9. файл forms.py – форма створення нового користувача

```
class BlogUserChangeForm(forms.ModelForm):
    class Meta:
        model = BlogUser
        fields = (
            'email', 'nickname', 'first_name', 'last_name', 'profile_image',
            'github_username', 'instagram_username', 'facebook_username', 'twitter_username'
        )
```

Рис. А.1.10. файл forms.py – форма зміни профіля користувача

```python
from django.urls import path
from .views import register, profile, login_view, logout_view, follow, unfollow, user_list

/accounts/
urlpatterns = [
    /accounts/register/
    path('register/', register, name='register'),
    /accounts/login/
    path('login/', login_view, name='login'),
    /accounts/logout/
    path('logout/', logout_view, name='logout'),
    /accounts/profile/{user_id}/
    path('profile/<int:user_id>/', profile, name='profile'),
    /accounts/follow/{user_id}/
    path('follow/<int:user_id>/', follow, name='follow'),
    /accounts/unfollow/{user_id}/
    path('unfollow/<int:user_id>/', unfollow, name='unfollow'),
    /accounts/users/
    path('users/', user_list, name='user_list'),
]
```

Рис. А.1.11. файл routes.py – список доступних маршрутів

```python
class BlogUserAdmin(UserAdmin):
    model = BlogUser
    add_form = BlogUserCreationForm
    form = BlogUserChangeForm
    list_display = ('email', 'nickname', 'first_name', 'last_name', 'is_staff', 'is_active')
    list_filter = ('is_staff', 'is_active')
    fieldsets = (
        (None, {'fields': ('email', 'password')}),
        ('Personal info', {'fields': ('nickname', 'first_name', 'last_name', 'profile_image')}),
        ('Permissions', {'fields': ('is_active', 'is_staff', 'is_superuser', 'groups', 'user_permissions')}),
        ('Important dates', {'fields': ('last_login', 'date_joined')}),
    )
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('email', 'nickname', 'password1', 'password2', 'first_name', 'last_name', 'profile_image')}
        ),
    )
    search_fields = ('email', 'nickname')
    ordering = ('email',)


admin.site.register(BlogUser, BlogUserAdmin)
```

Рис. А.1.12. файл admin.py – налаштування моделі в адмін-панелі

```python
class ActiveUserMiddleware(MiddlewareMixin):
    def process_request(self, request):
        if request.user.is_authenticated and request.session.session_key:
            cache_key = f'last-seen-{request.user.id}'
            last_login = cache.get(cache_key)

            if not last_login:
                BlogUser.objects.filter(id=request.user.id).update(last_login=timezone.now())
                # set caching to 300 seconds from the current date by key last-seen-{id of user}
                cache.set(cache_key, timezone.now(), 300)
```

Рис. А.1.13. файл middleware.py – статус «активний-неактивний»

```html
{% extends 'base.html' %}

{% block content %}
<div class="container">
  <h2>Login</h2>
  <form method="post">
    {% csrf_token %}
    {{ form.as_p }}
    <button type="submit" class="btn btn-primary">Login</button>
  </form>
</div>
{% endblock %}
```

Рис. А.1.14. файл login.html – шаблон логіну

```
{% extends 'base.html' %}
{% load static %}

{% block content %}
<div id="message-container"></div>
<div class="container">
  <div class="main-body">
    <div class="row">
      <div class="col-lg-5">
        <div class="card">
          <div class="card-body">
            <div class="d-flex flex-column align-items-center text-center">
              <div class="profile-pic-wrapper">
                {% if form.instance.profile_image %}
                  <img src="{{ form.instance.profile_image.url }}" alt="User" class="rounded-circle p-1 bg-primary" width="110" id="profileImage">
                {% else %}
                  <img src="{% static 'images/default_user_image.png' %}" alt="User" class="rounded-circle p-1 bg-primary" width="110" id="profileImage">
                {% endif %}
                <input id="profileImageInput" type="file" name="profile_image" accept="image/*" style="...">
              </div>
              <div class="mt-3">
                <h4 id="nickname">{{ form.instance.nickname }}</h4>
                <p class="text-secondary mb-1">{{ form.instance.followers.count }} Followers</p>
                <p class="text-muted font-size-sm">Joined: {{ form.instance.date_joined|date:"M j, Y" }}</p>
                <p class="text-muted font-size-sm">Status: {% if user.is_online %}Online{% else %}Not online{% endif %}</p>
              </div>
            </div>
            <hr class="my-4">
            <ul id="socialMediaLinks" class="list-group list-group-flush">
```

Рис. А.1.15. файл profile_edit.html – частина шаблону зміни профілю

```
{% block scripts %}
  <script>
    document.querySelectorAll('#socialMediaLinks li').forEach(function(li) {
      li.addEventListener('click', function() {
        const usernameSpan = this.querySelector('.username');
        const input = this.querySelector('.username-input');

        // Get the full username from the data attribute
        const fullUsername = usernameSpan.getAttribute('data-full-username').trim();

        // Hide the username span and show the input field with the full username
        usernameSpan.style.display = 'none';
        input.style.display = 'inline-block';
        input.value = fullUsername; // Set full username as input value
        input.focus(); // Focus the input field

        // Save changes on blur or Enter key press
        input.addEventListener('blur', function() {
          saveUsername();
        });

        input.addEventListener('keypress', function(event) {
          if (event.key === 'Enter') {
            saveUsername();
          }
        });
```

Рис. А.1.16. файл profile_edit.html – частина JS коду зміни профілю

```
{% block content %}
<form method="post" enctype="multipart/form-data">
  {% csrf_token %}

  {% for field in form %}
  <div class="form-group{% if field.errors %} form-group-error{% endif %}">
    {{ field.label_tag }}
    {{ field }}
    {% if field.errors %}
    <ul style="...">
      {% for error in field.errors %}
      <li>{{ error }}</li>
      {% endfor %}
    </ul>
    {% endif %}
  </div>
  {% endfor %}

  <button type="submit">Create Profile</button>
</form>
{% endblock %}
```

Рис. А.1.17. файл register.html – шаблон реєстрації нового користувача

```
{% extends 'base.html' %}
{% load static %}

{% block content %}
<div id="message-container"></div>
<div class="container">
  <div class="main-body">
    <div class="row">
      <div class="col-lg-5">
        <div class="card">
          <div class="card-body">
            <div class="d-flex flex-column align-items-center text-center">
              <div class="profile-pic-wrapper">
                {% if form.instance.profile_image %}
                  <img src="{{ form.instance.profile_image.url }}" alt="User" class="rounded-circle p-1 bg-primary" width="110" id="profileImage">
                {% else %}
                  <img src="{% static 'images/default_user_image.png' %}" alt="User" class="rounded-circle p-1 bg-primary" width="110" id="profileImage">
                {% endif %}
                <input id="profileImageInput" type="file" name="profile_image" accept="image/*" style="...">
              </div>
              <div class="mt-3">
                <h4 id="nickname">{{ form.instance.nickname }}</h4>
                <p class="text-secondary mb-1">{{ form.instance.followers.count }} Followers</p>
                <p class="text-muted font-size-sm">Joined: {{ form.instance.date_joined|date:"M j, Y" }}</p>
                <p class="text-muted font-size-sm">Status: {% if user.is_online %}Online{% else %}Not online{% endif %}</p>
              </div>
            </div>
            <hr class="my-4">
            <ul id="socialMediaLinks" class="list-group list-group-flush">
```

Рис. А.1.18. файл profile_view.html – частина шаблону профілю користувача

```
{% block scripts %}
  <script>
    document.querySelectorAll('#socialMediaLinks li').forEach(function(li) {
      li.addEventListener('click', function() {
        const usernameSpan = this.querySelector('.username');
        const input = this.querySelector('.username-input');

        // Get the full username from the data attribute
        const fullUsername = usernameSpan.getAttribute('data-full-username').trim();

        // Hide the username span and show the input field with the full username
        usernameSpan.style.display = 'none';
        input.style.display = 'inline-block';
        input.value = fullUsername; // Set full username as input value
        input.focus(); // Focus the input field

        // Save changes on blur or Enter key press
        input.addEventListener('blur', function() {
          saveUsername();
        });

        input.addEventListener('keypress', function(event) {
          if (event.key === 'Enter') {
            saveUsername();
          }
        });
```

Рис. А.1.19. файл profile_view.html – JS код шаблону профілю користувача

```
{% extends 'base.html' %}
{% load static %}


{% block content %}
<div class="container">
    <div class="list-group">
        {% for user in users %}
        <div class="list-group-item d-flex justify-content-between align-items-center">
            <div class="d-flex align-items-center">
                <img src="{% if user.profile_image %}{{ user.profile_image.url }}{% else %}{% static 'images/default_user_image.png' %}{% endif %}"
                     alt="{{ user.nickname }} profile picture"
                     class="rounded-circle"
                     width="50"
                     height="50">
                <div class="ml-3">
                    <h5 class="mb-1">{{ user.nickname }}</h5>
                    <p class="mb-1">Subscribers: {{ user.followers.count }} | Posts: {{ user.posts.count }}</p>
                </div>
            </div>
            <a href="{% url 'profile' user_id=user.id %}" class="btn btn-outline-primary">
                →
            </a>
        </div>
        {% empty %}
        <p>No users found.</p>
        {% endfor %}
    </div>
</div>
{% endblock %}
```

Рис. А.1.20. файл user_list.html – шаблон списку користувачів

А.2 Модуль blog

```python
class PostManager(models.Manager):
    """Custom manager for the Post objects"""

    def all(self):
        """return only approved posts by default"""
        return self.get_queryset().select_related('author', 'category').prefetch_related('ratings', 'views').filter(is_approved=True)

    def all_with_non_approved(self):
        """return all posts"""
        return self.get_queryset().select_related('author', 'category').prefetch_related('ratings', 'views')

    def detail(self):
        """post details with fetching all needed connected instances to avoid N+1 problem"""
        return self.get_queryset() \
            .select_related('author', 'category') \
            .prefetch_related('comments', 'comments__author', 'tags', 'ratings') \
            .filter(is_approved=True)

    def detail_with_non_approved(self):
        return self.get_queryset() \
            .select_related('author', 'category') \
            .prefetch_related('comments', 'comments__author', 'tags', 'ratings')
```

Рис. А.2.1. файл models.py – PostManager, який використовується моделлю посту

```python
    title = models.CharField(max_length=255, verbose_name="Title")
    slug = models.SlugField(verbose_name='URL', max_length=255, blank=True, unique=True)
    category = TreeForeignKey( to: 'Category', on_delete=models.PROTECT, related_name='posts', verbose_name='Category')
    content = CKEditor5Field( *args: 'Content', config_name='extends')
    author = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, related_name="posts",
                               verbose_name="Author")
    created_at = models.DateTimeField(auto_now_add=True, verbose_name="Created at")
    updated_at = models.DateTimeField(auto_now=True, verbose_name="Updated at")
    image = models.ImageField(
        upload_to='post_images/',
        blank=True,
        null=True,
        verbose_name="Image",
        validators=[FileExtensionValidator(allowed_extensions=('png', 'jpg', 'webp', 'jpeg', 'gif'))]
    )
    is_approved = models.BooleanField(default=False, verbose_name="Approved by Admin")
```

Рис. А.2.2. файл models.py – поля, які є в моделі поста

```python
def __init__(self, *args, **kwargs):
    super().__init__( *args: *args, **kwargs)
    self.__image = self.image if self.pk else None

def __str__(self):
    return self.title

def get_absolute_url(self):
    return reverse( viewname: 'post_details', kwargs={'slug': self.slug})

def save(self, *args, **kwargs):
    """Add slug if not filled and compress image"""
    if not self.slug:
        self.slug = unique_slugify(self, self.title)
    super().save(*args, **kwargs)

    if self.image and self.__image != self.image:
        image_compress(self.image.path, width=500, height=500)

4 usages (3 dynamic)
def get_sum_rating(self):
    return sum([rating.value for rating in self.ratings.all()])

2 usages (1 dynamic)
def get_view_count(self):
    return self.views.count()

def get_today_view_count(self):
    return self.views.filter(viewed_at__date=date.today()).count()
```

Рис. А.2.3. файл models.py – методи, які є в моделі посту

```python
title = models.CharField(max_length=255, verbose_name='Category Title')
slug = models.SlugField(max_length=255, verbose_name='Category Slug', blank=True)
description = models.TextField(verbose_name='Category Description', max_length=300)
parent = TreeForeignKey(
    to: 'self',
    on_delete=models.CASCADE,
    null=True,
    blank=True,
    db_index=True,
    related_name='children',
    verbose_name='Parent Category'
)
```

Рис. А.2.4. файл models.py – поля, які є в моделі категорії

```
    class MPTTMeta:
        order_insertion_by = ('title',)

    class Meta:
        verbose_name = 'Category'
        verbose_name_plural = 'Categories'

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse( viewname: 'posts_by_category', kwargs={'slug': self.slug})
```

Рис. А.2.5. файл models.py – класи та методи, які є в моделі категорії

```
class Comment(MPTTModel):
    """Comments with tree structure"""
    post = models.ForeignKey(Post, on_delete=models.CASCADE, verbose_name='Post', related_name='comments')
    author = models.ForeignKey(settings.AUTH_USER_MODEL, verbose_name='Author of comment', on_delete=models.CASCADE, related_name='comments_author')
    content = models.TextField(verbose_name='Text of comment', max_length=3000)
    created_at = models.DateTimeField(verbose_name='Datetime of creation', auto_now_add=True)
    updated_at = models.DateTimeField(verbose_name='Datetime of update', auto_now=True)
    is_updated = models.BooleanField(default=False, verbose_name='Is edited')
    parent = TreeForeignKey( to: 'self', verbose_name='Parent comment', null=True, blank=True, related_name='children', on_delete=models.CASCADE)

    class MTTMeta:
        order_insertion_by = ('-created_at',)

    class Meta:
        indexes = [models.Index(fields=['-created_at', 'updated_at', 'parent'])]
        ordering = ['-created_at']
        verbose_name = 'Comment'
        verbose_name_plural = 'Comments'

    def __str__(self):
        return f'{self.author}:{self.content}'
```

Рис. А.2.6. файл models.py – модель коментарів

```
class Rating(models.Model):
    """Rating model: Like - Dislike"""
    post = models.ForeignKey(to=Post, verbose_name='Post', on_delete=models.CASCADE, related_name='ratings')
    user = models.ForeignKey(to=settings.AUTH_USER_MODEL, verbose_name='User', on_delete=models.CASCADE, blank=True, null=True)
    value = models.IntegerField(verbose_name='Value', choices=[(1, 'Like'), (-1, 'Dislike')])
    created_at = models.DateTimeField(verbose_name='Created at', auto_now_add=True)
    ip_address = models.GenericIPAddressField(verbose_name='IP Address')

    class Meta:
        unique_together = ('post', 'ip_address')
        ordering = ('-created_at',)
        indexes = [models.Index(fields=['-created_at', 'value'])]
        verbose_name = 'Rating'
        verbose_name_plural = 'Ratings'

    def __str__(self):
        return self.post.title
```

Рис. А.2.7. файл models.py – модель рейтингу постів

```python
class ViewCount(models.Model):
    """Counter of Post's views"""
    post = models.ForeignKey( to: 'Post', on_delete=models.CASCADE, related_name='views')
    ip_address = models.GenericIPAddressField(verbose_name='IP Address')
    viewed_at = models.DateTimeField(auto_now_add=True, verbose_name='Datetime of view')

    class Meta:
        ordering = ('-viewed_at',)
        indexes = [models.Index(fields=['-viewed_at'])]
        verbose_name = 'View'
        verbose_name_plural = 'Views'

    def __str__(self):
        return self.post.title
```

Рис. А.2.8. файл models.py – модель підрахунку переглядів постів

```python
class PostListView(ListView):
    model = Post
    template_name = 'blog/posts_list.html'
    context_object_name = 'posts'
    paginate_by = 5
    queryset = Post.objects.all()

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = 'Posts'
        return context
```

Рис. А.2.9. файл views.py – клас для отримання списку постів

```python
class PostSearchResultView(ListView):
    """Post search view"""
    model = Post
    context_object_name = 'posts'
    paginate_by = 10
    allow_empty = True
    template_name = 'blog/posts_list.html'

    def get_queryset(self):
        query = self.request.GET.get('search')
        search_vector = SearchVector( *expressions: 'content', weight='B') + SearchVector( *expressions: 'title', weight='A')
        search_query = SearchQuery(query)
        return self.model.objects.annotate(rank=SearchRank(search_vector, search_query)).filter(rank__gte=0.2).order_by('-rank')

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = f'Search results: {self.request.GET.get("search")}'
        return context
```

Рис. А.2.10. файл views.py – клас для пошуку постів по ключовому слову

```python
class PostByTagListView(ListView):
    model = Post
    template_name = 'blog/posts_list.html'
    context_object_name = 'posts'
    paginate_by = 10
    tag = None

    def get_queryset(self):
        self.tag = Tag.objects.get(slug=self.kwargs['tag'])
        queryset = Post.objects.all().filter(tags__slug=self.tag.slug)
        return queryset

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = f'Posts by tag: {self.tag.name}'
        return context
```

Рис. А.2.11. файл views.py – клас для пошуку постів по обраному тегу

```python
class PostByCategoryListView(ListView):
    model = Post
    template_name = 'blog/posts_list.html'
    context_object_name = 'posts'
    category = None

    def get_queryset(self):
        self.category = Category.objects.get(slug=self.kwargs['slug'])
        return Post.objects.all().filter(category__slug=self.category.slug)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = f'Posts by category: {self.category.title}'
        return context
```

Рис. А.2.12. файл views.py – клас для пошуку постів по обраній категорії

```python
class PostDetailView(ViewCountMixin, DetailView):
    model = Post
    template_name = 'blog/post_details.html'
    context_object_name = 'post'
    queryset = model.objects.detail()

    1 usage
    def get_similar_posts(self, obj):
        post_tags_ids = obj.tags.values_list('id', flat=True)
        similar_posts = Post.objects.filter(tags__in=post_tags_ids).exclude(id=obj.id)
        similar_posts = similar_posts.annotate(related_tags=Count('tags')).order_by('-related_tags')
        similar_posts_list = list(similar_posts.all())
        random.shuffle(similar_posts_list)
        return similar_posts_list[:3]

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = self.object.title
        context['form'] = CommentCreateForm
        context['similar_posts'] = self.get_similar_posts(self.object)
        return context
```

Рис. А.2.13. файл views.py – клас для відображення всієї інформації посту

```python
class PostCreateView(LoginRequiredMixin, SuccessMessageMixin, CreateView):
    model = Post
    template_name = 'blog/create_post.html'
    form_class = PostCreateForm
    login_url = 'login'
    success_message = 'Post created successfully, wait for approval from admin!'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = 'Create New Post'
        return context

    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)

    def get_success_url(self):
        return reverse( viewname: 'profile', kwargs={'user_id': self.request.user.id})
```

Рис. А.2.14. файл views.py – клас для створення нового посту

```python
class PostUpdateView(AuthorRequiredMixin, SuccessMessageMixin, UpdateView):
    model = Post
    template_name = 'blog/update_post.html'
    context_object_name = 'post'
    form_class = PostUpdateForm
    login_url = 'login'
    success_message = 'Post was successfully updated!'
    queryset = model.objects.detail_with_non_approved()

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = f'Update Post: {self.object.title}'
        return context

    def form_valid(self, form):
        form.save()
        return super().form_valid(form)

    def get_success_url(self):
        return reverse( viewname: 'profile', kwargs={'user_id': self.request.user.id})
```

Рис. А.2.15. файл views.py – клас для оновлення існуючого посту

```python
class PostDeleteView(AuthorRequiredMixin, SuccessMessageMixin, DeleteView):
    model = Post
    context_object_name = 'post'
    template_name = 'blog/delete_post.html'
    success_message = 'Post was successfully deleted!'
    queryset = model.objects.detail_with_non_approved()

    def get_success_url(self):
        return reverse( viewname: 'profile', kwargs={'user_id': self.request.user.id})

    def get_context_data(self, *, object_list=None, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = f'Deletion of post: {self.object.title}'
        return context
```

Рис. А.2.16. файл views.py – клас для видалення посту

```python
class CommentCreateView(LoginRequiredMixin, CreateView):
    form_class = CommentCreateForm
    2 usages
    def is_ajax(self):
        return self.request.headers.get('X-Requested-With') == 'XMLHttpRequest'
    def form_invalid(self, form):
        if self.is_ajax():
            return JsonResponse( data: {'error': form.errors}, status=400)
        return super().form_invalid(form)
    def form_valid(self, form):
        comment = form.save(commit=False)
        comment.post_id = self.kwargs.get('id')
        comment.author = self.request.user
        comment.parent_id = form.cleaned_data.get('parent')
        comment.save()

        if self.is_ajax():
            return JsonResponse( data: {
                'is_child': comment.is_child_node(),
                'id': comment.id,
                'author': comment.author.nickname,
                'parent_id': comment.parent_id,
                'created_at': comment.created_at.strftime('%Y-%b-%d %H:%M:%S'),
                'profile_image': comment.author.profile_image.url,
                'content': comment.content,
                'get_absolute_url': comment.author.get_absolute_url()
            }, status=200)

        return redirect(comment.post.get_absolute_url())
    def handle_no_permission(self):
        return JsonResponse( data: {'error': 'Log-in or register to have possibility to write comments!'}, status=400)
```

Рис. А.2.17. файл views.py – клас для видалення коментарів

```python
class RatingCreateView(View):
    model = Rating

    def post(self, request, *args, **kwargs):
        post_id = request.POST.get('post_id')
        value = int(request.POST.get('value'))
        ip_address = get_client_ip(request)
        user = request.user if request.user.is_authenticated else None

        rating, created = self.model.objects.get_or_create(
            post_id=post_id,
            ip_address=ip_address,
            defaults={'value': value, 'user': user},
        )

        if not created:
            if rating.value == value:
                rating.delete()
                return JsonResponse({'status': 'deleted', 'rating_sum': rating.post.get_sum_rating()})
            else:
                rating.value = value
                rating.user = user
                rating.save()
                return JsonResponse({'status': 'updated', 'rating_sum': rating.post.get_sum_rating()})
        return JsonResponse({'status': 'created', 'rating_sum': rating.post.get_sum_rating()})
```

Рис. А.2.18. файл views.py – клас для лайків/дізлайків постів

```python
urlpatterns = [
    /blog/posts/
    path('posts/', PostListView.as_view(), name='posts'),
    /blog/posts/tags/{tag}/
    path('posts/tags/<str:tag>/', PostByTagListView.as_view(), name='posts_by_tags'),
    /blog/posts/categories/{slug}/
    path('posts/categories/<str:slug>/', PostByCategoryListView.as_view(), name="posts_by_category"),
    /blog/posts/search/
    path('posts/search/', PostSearchResultView.as_view(), name='search_posts'),
    /blog/post/{slug}/
    path('post/<str:slug>/', PostDetailView.as_view(), name='post_details'),
    /blog/create_post/
    path('create_post/', PostCreateView.as_view(), name='create_post'),
    /blog/post/{slug}/update/
    path('post/<str:slug>/update/', PostUpdateView.as_view(), name='update_post'),
    /blog/post/{slug}/delete/
    path('post/<str:slug>/delete/', PostDeleteView.as_view(), name='delete_post'),
    /blog/post/{id}/add_comment/
    path('post/<int:id>/add_comment/', CommentCreateView.as_view(), name='add_comment'),
    /blog/post/{id}/rating/
    path('post/<int:id>/rating/', RatingCreateView.as_view(), name='add_rating'),
]
```

Рис. А.2.19. файл routes.py – доступні маршрути постів

```python
class PostCreateForm(forms.ModelForm):
    class Meta:
        model = Post
        fields = ['title', 'category', 'tags', 'content', 'image']

    def __init__(self, *args, **kwargs):
        super(PostCreateForm, self).__init__(*args, **kwargs)
        self.fields['title'].required = False
        self.fields['content'].widget.attrs.update({'class': 'form-control django_ckeditor_5'})
        self.fields['content'].required = False
        self.fields['tags'].widget.attrs.update({'class': 'form-control'})
        self.fields['category'].widget.attrs.update({'class': 'custom-select'})

    def clean_content(self):
        content = self.cleaned_data.get('content')
        if content:
            soup = BeautifulSoup(content, features: 'html.parser')
            text = soup.get_text(strip=True)
            if not text:
                raise forms.ValidationError('This field cannot be empty.')
        else:
            raise forms.ValidationError('This field cannot be empty.')
        return content

    def clean_title(self):
        title = self.cleaned_data.get('title')
        if not title:
            raise forms.ValidationError('This field cannot be empty.')
        return title
```

Рис. А.2.20. файл forms.py – форма створення постів

```python
class PostUpdateForm(PostCreateForm):
    class Meta:
        model = Post
        fields = PostCreateForm.Meta.fields

    def __init__(self, *args, **kwargs):
        super().__init__( *args: *args, **kwargs)
```

Рис. А.2.21. файл forms.py – форма оновлення постів

```python
class CommentCreateForm(forms.ModelForm):
    """Form for creation a post's comment"""
    parent = forms.IntegerField(widget=forms.HiddenInput, required=False)
    content = forms.CharField(label='', widget=forms.Textarea(attrs={'cols': 30, 'rows': 5, 'placeholder': 'Comment', 'class': 'form-control'}))

    class Meta:
        model = Comment
        fields = ('content',)
```

Рис. А.2.22. файл forms.py – форма створення коментарів

```python
@admin.register(Category)
class CategoryAdmin(DraggableMPTTAdmin):
    """Category model in admin-panel"""
    list_display = ('tree_actions', 'indented_title', 'id', 'title', 'slug')
    list_display_links = ('title', 'slug')
    prepopulated_fields = {'slug': ('title',)}

    fieldsets = (
        ('Main Info', {'fields': ('title', 'slug', 'parent')}),
        ('Description', {'fields': ('description',)})
    )
```

Рис. А.2.23. файл admin.py – налаштування категорій в адмін-панелі

```python
@admin.register(Comment)
class CommentAdminPage(DraggableMPTTAdmin):
    """Comment model in admin-panel"""
    list_display = ('tree_actions', 'indented_title', 'post', 'author', 'created_at', 'is_updated')
    mptt_level_indent = 2
    list_display_links = ('post',)
    list_filter = ('created_at', 'updated_at', 'author', 'post')
```

Рис. А.2.24. файл admin.py – налаштування коментарів в адмін-панелі

```python
@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    prepopulated_fields = {'slug': ('title',)}

    def get_readonly_fields(self, request, obj=None):
        return ['author'] if obj else []
```

Рис. А.2.25. файл admin.py – налаштування постів в адмін-панелі

```python
@admin.register(ViewCount)
class ViewCountAdmin(admin.ModelAdmin):
    pass
```

Рис. А.2.26. файл admin.py – налаштування підрахунку переглядів в адмін-панелі

```python
class ViewCountMixin:
    """Mixin for increasing view's count"""
    def get_object(self):
        post_object = super().get_object()
        ip_address = get_client_ip(self.request)
        ViewCount.objects.get_or_create(post=post_object, ip_address=ip_address)
        return post_object
```

Рис. А.2.27. файл mixins.py – міксин для підрахунку переглядів постів

```python
# A dictionary to store the old values of `is_approved` before the instance is saved
old_values = {}


@receiver(pre_save, sender=Post)
def store_old_is_approved(sender, instance, **kwargs):
    if instance.pk:
        post_before = Post.objects.get(pk=instance.pk)
        old_values[instance.pk] = post_before.is_approved


@receiver(post_save, sender=Post)
def post_approved_handler(sender, instance, created, **kwargs):
    if created:
        return

    old_is_approved = old_values.get(instance.pk)
    if not old_is_approved and instance.is_approved:
        trigger_approval_email_message_task.delay(instance.author.email, instance.title)
        print(f'Approval email sent to {instance.author.email} !')
        if instance.author.followers.exists():
            trigger_new_post_email_message_task.delay(
                recipients=[f.email for f in instance.author.followers.all()],
                author_name=instance.author.nickname,
                post_name=instance.title
            )
    old_values.pop(instance.pk, None)
```

Рис. А.2.28. файл signals.py – сигнали, які спрацьовують після модифікації постів

```python
@shared_task
def trigger_approval_email_message_task(recipient: str, post_name: str):
    current_app.send_task(
        'email_sender.send_approval_email_message_task',
        kwargs={'recipient': recipient, 'post_name': post_name}
    )


2 usages
@shared_task
def trigger_new_post_email_message_task(recipients: list, author_name: str, post_name: str):
    current_app.send_task(
        'email_sender.send_new_post_email_message_task',
        kwargs={'recipients': recipients, 'author_name': author_name, 'post_name': post_name}
    )
```

Рис. А.2.29. файл tasks.py – асинхронні завдання, які є для постів

```
{% block content %}
<div class="container">
    <h2>Create a New Post</h2>
    <form method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.media }}
        {% if form.errors %}
        <div class="alert alert-danger">
            <ol>
                {% for field in form %}
                    {% if field.errors %}
                        <li><strong>"{{ field.label }}" Field:</strong> {{ field.errors }}</li>
                    {% endif %}
                {% endfor %}
                {% if form.non_field_errors %}
                    <li>{{ form.non_field_errors }}</li>
                {% endif %}
            </ol>
        </div>
        {% endif %}

        <div class="form-group">
            {{ form.title.label_tag }}
            {{ form.title }}
        </div>

        <div class="form-group">
            {{ form.category.label_tag }}
            {{ form.category }}
        </div>
```

Рис. А.2.30. файл create_post.html – частина шаблону створення постів

```
{% extends 'base.html' %}

{% block content %}
<div class="card mb-3 border-0 nth-shadow">
  <div class="card-body">
    <div class="card-title nth-card-title">
      <h4>Deletion of post</h4>
    </div>
    <form method="post">
      {% csrf_token %}
      <div class="alert alert-warning" role="alert">
        <i class="fas fa-info-circle"></i>You are going to delete: <strong>{{ post.title }}</strong>, are you sure about it?
      </div>
      <div class="row align-items-center mt-2">
        <div class="col">
          <button type="submit" class="btn btn-danger w-100">Delete it</button>
        </div>
        <div class="col">
          <a href="{{ post.author.get_absolute_url }}" class="btn btn-primary w-100">No, leave it</a>
        </div>
      </div>
    </form>
  </div>
</div>
{% endblock %}
```

Рис. А.2.31. файл delete_post.html – шаблон видалення постів

```
{% extends 'base.html' %}
{% load mptt_tags static %}

{% block extra_styles %}
<link rel="stylesheet" href="{% static 'custom_css/sidebar.css' %}">
{% endblock %}

{% block content %}
<div class="container mt-4">
    <div class="row">
        <div class="col-lg-8">
            <div class="card mb-4 border-0 shadow-sm">
                <div class="card-body">
                    <div class="d-flex justify-content-center">
                        <h1 class="card-title">{{ post.title }}
                            <a href="{{ post.author.get_absolute_url }}" class="text-muted" style="...">by {{ post.author }}</a>
                        </h1>
                    </div>

                    <div class="post-content mt-4">
                        {{ post.content|safe }}
                    </div>

                    {% if post.tags.all %}
                    <div class="mt-3">
                        <span class="text-muted">Tags:</span>
                        {% for tag in post.tags.all %}
                            <a href="{% url 'posts_by_tags' tag.slug %}" class="badge bg-primary text-light">{{ tag }}</a>
                        {% endfor %}
                    </div>
```

Рис. А.2.32. файл post_details.html – частина шаблону деталей посту

```
{% extends 'base.html' %}
{% load static %}

{% block extra_styles %}
<link rel="stylesheet" href="{% static 'custom_css/sidebar.css' %}">
{% endblock %}

{% block content %}
<div class="container">
    <div class="row">
        <div class="col-lg-8 mb-4">
            <form class="col-12 col-lg-auto mb-4 d-flex" role="search" method="get" action="{% url 'search_posts' %}">
                <input type="search" class="form-control me-2" placeholder="Search..." aria-label="Search" name="search" autocomplete="off"

                {% if request.GET.search %}
                    <button class="btn btn-sm btn-danger">
                        <a href="{% url 'posts' %}" style="...">Clear Search</a>
                    </button>
                {% else %}
                    <button type="submit" class="btn btn-sm btn-primary">Search</button>
                {% endif %}
            </form>

            {% for post in posts %}
            <div class="card mb-3">
                <div class="row g-0">
                    <div class="col-md-4">
                        <img src="{% if post.image %}{{ post.image.url }}{% else %}{% static 'images/default_post_image.webp' %}{% endif %}'
                    </div>
```

Рис. А.2.33. файл posts_list.html – частина шаблону списку постів

```
{% block extra_styles %}
    <link rel="stylesheet" href="{% static 'blog/css/create_post.css' %}">
{% endblock %}

{% block content %}
<div class="card mb-3 border-0 nth-shadow">
    <div class="card-body">
        <form method="post" enctype="multipart/form-data">
            {% csrf_token %}
            {{ form.media }}
            {% if form.errors %}
            <div class="alert alert-danger">
                <ol>
                    {% for field in form %}
                    {% if field.errors %}
                    <li><strong>"{{ field.label }}" Field:</strong> {{ field.errors }}</li>
                    {% endif %}
                    {% endfor %}
                    {% if form.non_field_errors %}
                    <li>{{ form.non_field_errors }}</li>
                    {% endif %}
                </ol>
            </div>
            {% endif %}

            <div class="form-group">
                {{ form.title.label_tag }}
                {{ form.title }}
            </div>
```

Рис. А.2.34. файл update_post.html – частина шаблону оновлення постів

```python
@register.simple_tag
def popular_tags():
    tags = Tag.objects.annotate(num_times=Count('post')).order_by('-num_times')
    return list(tags.values('name', 'num_times', 'slug'))


1 usage
@register.simple_tag
def popular_posts():
    start_date = timezone.now() - timedelta(days=7)
    today_start = timezone.make_aware(datetime.combine(date.today(), time.min))
    posts = Post.objects.annotate(
        total_view_count=Count( expression: 'views', filter=Q(views__viewed_at__gte=start_date)),
        today_view_count=Count( expression: 'views', filter=Q(views__viewed_at__gte=today_start))
    ).prefetch_related('views')
    return posts.order_by('-total_view_count', '-today_view_count')[:5]


1 usage
@register.inclusion_tag('latest_comments.html')
def show_latest_comments(count=5):
    comments = Comment.objects.select_related('author').order_by('-created_at')[:count]
    return {'comments': comments}
```

Рис. А.2.35. файл blog_tags.py – три спеціальні теги для шаблонів постів

```python
from django import template

register = template.Library()


@register.filter
def add_class(field, css_class):
    return field.as_widget(attrs={'class': css_class})
```

Рис. А.2.36. файл custom_filters.py – спеціальний фільтр для постів

А.3 Модуль system

```python
class Feedback(models.Model):
    """Feedback model"""
    subject = models.CharField(max_length=255, verbose_name='Subject of email')
    content = models.TextField(verbose_name='Email content')
    created_at = models.DateTimeField(auto_now_add=True, verbose_name='Created at')
    ip_address = models.GenericIPAddressField(verbose_name="Sender's IP address",  blank=True, null=True)
    user = models.ForeignKey(settings.AUTH_USER_MODEL, verbose_name='User', on_delete=models.CASCADE)

    class Meta:
        verbose_name = 'Feedback'
        verbose_name_plural = 'Feedback'
        ordering = ['-created_at']

    def __str__(self):
        return f'Message from {self.user.email}'
```

Рис. А.3.1. файл models.py – модель зворотнього зв’язку

```python
class FeedbackCreateForm(forms.ModelForm):
    """Feedback creation form"""
    class Meta:
        model = Feedback
        fields = ('subject', 'content')

    recaptcha = ReCaptchaField(widget=ReCaptchaV2Checkbox, label='ReCAPTCHA')

    def __init__(self, *args, **kwargs):
        """Update fields styles"""
        super().__init__(*args, **kwargs)
        for field in [f for f in self.fields if f != 'recaptcha']:
            self.fields[field].widget.attrs.update({'class': 'form-control', 'autocomplete': 'off'})
```

Рис. А.3.2. файл forms.py – форма створення фідбеку

```python
urlpatterns = [
    /feedback/
    path('feedback/', FeedbackCreateView.as_view(), name='feedback'),
]
```

Рис. А.3.3. файл urls.py – маршрути модуля

```python
class FeedbackCreateView(LoginRequiredMixin, SuccessMessageMixin, CreateView):
    model = Feedback
    form_class = FeedbackCreateForm
    success_message = 'Your email has been successfully sent to site administration!'
    template_name = 'system/feedback.html'
    extra_context = {'title': 'Feedback Form'}

    def get_success_url(self):
        return reverse( viewname: 'profile', kwargs={'user_id': self.request.user.id})

    def form_valid(self, form):
        if form.is_valid():
            feedback = form.save(commit=False)
            feedback.ip_address = get_client_ip(self.request)
            feedback.user = self.request.user
            trigger_contact_email_message_task.delay(
                feedback.subject,
                feedback.content,
                feedback.ip_address,
                self.request.user.email,
                self.request.user.nickname
            )
        return super().form_valid(form)
```

Рис. А.3.4. файл views.py – клас для створення фідбеку

```python
def tr_handler404(request, exception):
    """Handler for 404 errors."""
    return render(request=request, template_name='system/errors/error_page.html', status=404, context={
        'title': 'Page Not Found',
        'error_message': "Page doesn't exist or was moved to another URL",
    })


def tr_handler500(request):
    """Handler for 500 errors."""
    return render(request=request, template_name='system/errors/error_page.html', status=500, context={
        'title': 'Server Error',
        'error_message': 'Internal Server Error, we are working on its solution',
    })


def tr_handler403(request, exception):
    """Handler for 403 errors."""
    return render(request=request, template_name='system/errors/error_page.html', status=403, context={
        'title': 'Forbidden',
        'error_message': "You don't have permission to access this resource",
    })
```

Рис. А.3.5. файл views.py – методи для модифікації сторінок-помилок

```python
@admin.register(Feedback)
class FeedbackAdmin(admin.ModelAdmin):
    """Feedback model in admin panel"""
    list_display = ('ip_address', 'user')
    list_display_links = ('user', 'ip_address')
```

Рис. А.3.6. файл admin.py – методи для модифікації сторінок-помилок

```html
{% block content %}
    <div class="card mb-3 border-0 nth-shadow">
        <div class="card-body">
            <form method="post" action="{% url 'feedback' %}">
                {% csrf_token %}
                {{ form.as_p }}

                <div class="d-grid gap-2 d-md-block mt-2">
                <button type="submit" class="btn btn-dark">Send email</button>
             </div>
            </form>
        </div>
    </div>
{% endblock %}
```

Рис. А.3.7. файл feedback.html – шаблон зворотнього зв’язку

```html
{% block content %}
    <div class="alert alert-danger alert-dismissible fade show" role="alert">
        {{ error_message }} | <a href="{% url 'posts' %}"><strong>Return to posts list page</strong></a>
        <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>
    </div>
{% endblock content %}
```

Рис. А.3.8. файл error_page.html – шаблон сторінки-помилки

А.4 Модуль services

```python
def generate_weekly_report():
    """Generation of users activity report(.pdf) for the last week"""
    def process_long_str_value(value: str) -> str:
        return value if len(value) < 15 else f'{value}...'

    one_week_ago = timezone.now() - timedelta(days=7)
    recent_posts = Post.objects.filter(created_at__gte=one_week_ago)
    recent_users = BlogUser.objects.filter(date_joined__gte=one_week_ago)
    total_views = sum(post.views.count() for post in recent_posts)

    # Create PDF
    pdf = PDF(orientation='L')
    pdf.add_page()

    # New Posts table
    pdf.chapter_title("New Posts")
    posts_data = [
        [
            process_long_str_value(post.author.nickname),
            process_long_str_value(post.title),
            process_long_str_value(post.category.title),
            post.created_at.strftime('%Y-%m-%d'),
            "+" if post.is_approved else '-',
            post.get_view_count()
        ]
        for post in recent_posts
    ]
    pdf.create_table( header: ['Author', 'Title', 'Category', 'Created At', 'Is Approved', 'Views'], posts_data)

    pdf.ln(10)
```

Рис. А.4.1. файл weekly_report.py – метод створення репорту(частина 1)

```python
    # New Users Table
    pdf.chapter_title("New Users")
    users_data = [
        [
            process_long_str_value(user.email),
            process_long_str_value(user.nickname),
            process_long_str_value(user.first_name) if user.first_name else '',
            process_long_str_value(user.last_name) if user.last_name else '',
            user.date_joined.strftime('%Y-%m-%d'),
            user.followers.count()
        ]
        for user in recent_users
    ]
    pdf.create_table( header: ['Email', 'Nickname', 'First Name', 'Last Name', 'Date Joined', 'Followers'], users_data)

    # Total Info - on a separate page and in bold
    pdf.add_page()
    pdf.set_font( family: "Arial", style: 'B', size: 12)
    pdf.cell( w: 0, h: 10, txt: f"Total New Posts: {recent_posts.count()}", border: 0, ln: 1)
    pdf.cell( w: 0, h: 10, txt: f"Total New Users Registered: {recent_users.count()}", border: 0, ln: 1)
    pdf.cell( w: 0, h: 10, txt: f"Total Posts Views: {total_views}", border: 0, ln: 1)

    # Save the PDF
    reports_dir = 'reports'
    os.makedirs(reports_dir, exist_ok=True)
    report_path = os.path.join(reports_dir, f"weekly_report_{timezone.now().strftime('%Y-%m-%d-%H-%M-%S')}.pdf")
    pdf.output(report_path)
```

Рис. А.4.2. файл weekly_report.py – метод створення репорту(частина 2)

```python
class CkeditorCustomStorage(FileSystemStorage):
    """Custom storage path of media files from Ckeditor"""
    1 usage
    def get_folder_name(self):
        return datetime.now().strftime('%Y/%m/%d')

    1 usage
    def get_valid_name(self, name):
        return name

    def _save(self, name, content):
        folder_name = self.get_folder_name()
        name = os.path.join(folder_name, self.get_valid_name(name))
        return super()._save(name, content)

    location = os.path.join(settings.MEDIA_ROOT, 'uploads/')
    base_url = urljoin(settings.MEDIA_URL, url: 'uploads/')
```

Рис. А.4.3. файл utils.py – збереження медіа-файлів із редактору Ckeditor

```python
def unique_slugify(instance, slug):
    """Generate unique slug in case if slug already exists."""
    model = instance.__class__
    unique_slug = slugify(slug)
    while model.objects.filter(slug=unique_slug).exists():
        unique_slug = f'{unique_slug}-{uuid4().hex[:8]}'
    return unique_slug


6 usages
def get_client_ip(request):
    """Get user's IP"""
    x_forwarded_for = request.META.get('HTTP_X_FORWARDED_FOR')
    return x_forwarded_for.split(',')[0] if x_forwarded_for else request.META.get('REMOTE_ADDR')


2 usages
def image_compress(image_path, height, width):
    """Image optimization"""
    img = Image.open(image_path)
    if img.mode != 'RGB':
        img = img.convert('RGB')
    if img.height > height or img.width > width:
        output_size = (height, width)
        img.thumbnail(output_size)
    img = ImageOps.exif_transpose(img)
    img.save(image_path, format='JPEG', quality=90, optimize=True)
```

Рис. А.4.4. файл utils.py – методи генерації слагу, отримання IP та стиснення зображень

```python
@shared_task
def trigger_contact_email_message_task(subject: str, content: str, ip: str, user_email: str, user_nickname: str):
    current_app.send_task(
        'email_sender.send_contact_email_message_task',
        kwargs={
            'subject': subject,
            'content': content,
            'ip': ip,
            'user_email': user_email,
            'user_nickname': user_nickname
        },
    )

@shared_task
def db_backup_task():
    """Background task for making pgSQL DB backup"""
    call_command('db_backup')

@shared_task
def generate_weekly_report_task():
    """Background task for generation of users activity report(.pdf) for the last week"""
    return generate_weekly_report()
```

Рис. А.4.5. файл tasks.py – асинхронні завдання модуля services

```python
class AuthorRequiredMixin(AccessMixin):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return self.handle_no_permission()

        if request.user == self.get_object().author or request.user.is_staff:
            return super().dispatch(request, *args, **kwargs)

        messages.info(request, message: 'Modification/Deletion of post available only for its author and admin!')
        return redirect( to: 'profile', user_id=request.user.id)
```

Рис. А.4.6. файл mixins.py – міксин валідації прав доступа користувача

```python
class Command(BaseCommand):
    """Command for backing up the pgSQL database."""
    def handle(self, *args, **options):
        self.stdout.write('Waiting for database dump...')
        backups_dir = os.path.join(os.getcwd(), 'backups')
        os.makedirs(backups_dir, exist_ok=True)

        # Construct the file path for the backup
        backup_file_path = os.path.join(
            backups_dir,
            f'database-{datetime.now().strftime("%Y-%m-%d-%H-%M-%S")}.json'
        )

        call_command(
            command_name: 'dumpdata',
            *args: '--natural-foreign',
            '--natural-primary',
            '--exclude=contenttypes',
            '--exclude=admin.logentry',
            '--indent=4',
            f'--output={backup_file_path}'
        )
        self.stdout.write(self.style.SUCCESS('Database successfully backed up'))
```

Рис. А.4.7. файл db_backup.py – команда створення бекапів для БД

А.5 Модуль(кореневий) blog_project

```python
CELERY_BROKER_URL = 'amqp://guest:guest@localhost//'
CELERY_RESULT_BACKEND = 'rpc://'
CELERY_TASK_TRACK_STARTED = True
CELERY_TASK_TIME_LIMIT = 30 * 60
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TASK_SERIALIZER = 'json'

CELERY_BEAT_SCHEDULE = {
    'backup_database': {
        'task': 'services.tasks.db_backup_task',
        'schedule': crontab(hour=0, minute=0),  # Every Day at midnight
    },
    'weekly_report': {
        'task': 'services.tasks.generate_weekly_report_task',
        'schedule': crontab( minute: 0,  hour: 0, day_of_week='mon'),  # Every Monday at midnight
    },
}
```

Рис. А.5.1. файл settings.py – налаштування Celery

```python
# Set the default Django settings module for the 'celery' program.
os.environ.setdefault( key: 'DJANGO_SETTINGS_MODULE', value: 'blog_project.settings')

app = Celery('blog_project')

app.config_from_object( obj: 'django.conf:settings', namespace='CELERY')

app.autodiscover_tasks()
```

Рис. А.5.2. файл celery.py – інтегрування Celery в Django

```python
handler403 = 'system.views.tr_handler403'
handler404 = 'system.views.tr_handler404'
handler500 = 'system.views.tr_handler500'
/
urlpatterns = [
    /admin/
    path('admin/', admin.site.urls),
    /accounts/
    path('accounts/', include('accounts.urls')),
    /blog/
    path('blog/', include('blog.urls')),
    /
    path('', include('system.urls')),

    /ckeditor5/
    path("ckeditor5/", include('django_ckeditor_5.urls')),
]

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Рис. А.5.3. файл urls.py – маршрутизація всього проєкту

А.6 Модуль static

```javascript
const getCookie = (name) : string  => {
    let cookieValue : null  = null;
    if (document.cookie && document.cookie !== "") {
        const cookies : string[]  = document.cookie.split( separator: ";");
        for (let i : number  = 0; i < cookies.length; i++) {
            const cookie : string  = cookies[i].trim();
            // Does this cookie string begin with the name we want?
            if (cookie.substring(0, name.length + 1) === name + "=") {
                cookieValue = decodeURIComponent(cookie.substring(name.length + 1));
                break;
            }
        }
    }
    return cookieValue;
};

const csrftoken : string  = getCookie( name: "csrftoken");
```

Рис. А.6.1. файл backend.js – метод отримання cookie

```
async function createComment(event) :Promise<void>  {
    event.preventDefault();
    commentFormSubmit.disabled = true;
    commentFormSubmit.innerText = "Waiting for response from server...";
    try {
        const response :Response  = await fetch( input: `/blog/post/${commentPostId}/add_comment/`, init: {
            method: 'POST',
            headers: {
                'X-CSRFToken': csrftoken,
                'X-Requested-With': 'XMLHttpRequest',
            },
            body: new FormData(commentForm),
        });
        const comment = await response.json();

        let commentTemplate :string  = `<ul id="comment-thread-${comment.id}"...>`;
        if (comment.is_child) {
            document.querySelector( selectors: `#comment-thread-${comment.parent_id}`).insertAdjacentHTML( position: "beforeend", commentTemplate);
        }
        else {document.querySelector( selectors: '.nested-comments').insertAdjacentHTML( position: "beforeend", commentTemplate)}
        commentForm.reset()
        commentFormSubmit.disabled = false;
        commentFormSubmit.innerText = "Add comment";
        commentFormParentInput.value = null;
        replyUser();
    }
    catch (error) {console.log(error)}
}
```

Рис. А.6.2. файл comments.js – метод створення коментаря

```
const ratingButtons :NodeListOf<Element>  = document.querySelectorAll( selectors: '.rating-buttons');

ratingButtons.forEach( callbackfn: button :Element  => {
    button.addEventListener( type: 'click', listener: event :Event  => {
        const value :number  = parseInt(event.target.dataset.value)
        const postId :number  = parseInt(event.target.dataset.post)
        const ratingSum :Element  = button.querySelector( selectors: '.rating-sum');
        const formData :FormData  = new FormData();
        formData.append( name: 'post_id', postId);
        formData.append( name: 'value', value);
        fetch( input: `/blog/post/${postId}/rating/`, init: {
            method: "POST",
            headers: {
                "X-CSRFToken": csrftoken,
                "X-Requested-With": "XMLHttpRequest",
            },
            body: formData
        }).then(response :Response  => response.json()) Promise<any>
            .then(data => {
                ratingSum.textContent = data.rating_sum;
            }) Promise<void>
            .catch(error => console.error(error));
    });
});
```

Рис. А.6.3. файл ratings.js – метод оцінювання постів

А.7 Кореневий модуль email_service(мікросервіс email_service)

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_USE_TLS = True
EMAIL_PORT = 587
EMAIL_HOST_USER = 'someemail.com@gmail.com'
EMAIL_HOST_PASSWORD = 'xxxx xxxx xxxx xxxx'


CELERY_BROKER_URL = 'amqp://localhost'  # RabbitMQ URL
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
```

Рис. А.7.1. файл settings.py – налаштування пошти та Celery

```
os.environ.setdefault( key: 'DJANGO_SETTINGS_MODULE', value: 'email_service.settings')

app = Celery('email_service')

app.config_from_object( obj: 'django.conf:settings', namespace='CELERY')

app.autodiscover_tasks()
```

Рис. А.7.2. файл celery.py – інтеграція Celery в мікросервіс

А.8. Модуль email_sender(мікросервіс email_service)

```
class EmailLog(models.Model):
    subject = models.CharField(max_length=255)
    recipient = models.EmailField()
    message = models.TextField()
    status = models.CharField(max_length=10)
    timestamp = models.DateTimeField(default=timezone.now)

    def __str__(self):
        return f"Email to {self.recipient} - {self.status}"
```

Рис. А.8.1. файл models.py – модель логування надісланих листів

```python
@shared_task(name='email_sender.send_contact_email_message_task')
def send_contact_email_message_task(
        subject: str,
        content: str,
        ip: str,
        user_email: str,
        user_nickname: str
):
    """BG Task to send contact form email"""
    message = render_to_string( template_name: 'email_sender/send_feedback_email.html', context: {
        'content': content,
        'ip': ip,
        'user_email': user_email,
        'user_nickname': user_nickname,
    })
    email_log_status = 'SUCCESS'
    try:
        send_mail(
            subject,
            message,
            from_email: "noreply@blogapp.com",
            recipient_list: [settings.EMAIL_HOST_USER],
            fail_silently=False,
        )
    except Exception as e:
        email_log_status = 'FAIL'
    finally:
        EmailLog.objects.create(subject=subject, recipient=settings.EMAIL_HOST_USER, message=message, status=email_log_status)
```

Рис. А.8.2. файл tasks.py - асинхронне завдання для листа-фідбека

```python
@shared_task(name='email_sender.send_approval_email_message_task')
def send_approval_email_message_task(recipient: str, post_name: str):
    """BG Task to send message about successful approval to Post's author"""
    message = render_to_string( template_name: 'email_sender/send_approval_email.html', context: {
        'recipient': recipient,
        'post_name': post_name,
    })
    email_log_status = 'SUCCESS'
    try:
        send_mail(
            subject: 'Post Approval',
            message,
            from_email: "noreply@blogapp.com",
            recipient_list: [recipient],
            fail_silently=False,
        )
    except Exception as e:
        email_log_status = 'FAIL'
    finally:
        EmailLog.objects.create(
            subject='Post Approval',
            recipient=settings.EMAIL_HOST_USER,
            message=message,
            status=email_log_status
        )
```

Рис. А.8.3. файл tasks.py – асинхронне завдання для листа-схвалення

```python
@shared_task(name='email_sender.send_new_post_email_message_task')
def send_new_post_email_message_task(recipients: list, author_name: str, post_name: str):
    """BG Task to send message about new post on website"""
    message = render_to_string( template_name: 'email_sender/send_new_post_email.html', context: {
        'author_name': author_name,
        'post_name': post_name,
    })
    for recipient in recipients:
        email_log_status = 'SUCCESS'
        try:
            send_mail(
                subject: 'New post on website',
                message,
                from_email: "noreply@blogapp.com",
                recipient_list: [recipient],
                fail_silently=False,
            )
        except Exception as e:
            email_log_status = 'FAIL'
        finally:
            EmailLog.objects.create(
                subject='New post on website',
                recipient=recipient,
                message=message,
                status=email_log_status
            )
```

Рис. А.8.4. файл tasks.py – асинхронне завдання для листа-розсилки

```
{% autoescape off %}
Hi, {{ recipient }},

Your Post "{{ post_name }}" was successfully approved, thanks for sharing you knowledge and experience with others!

Django Blog Project
{% endautoescape %}
```

Рис. А.8.5. файл send_approval_email.html– шаблон листа-схвалення

```
{% autoescape off %}
Hi, Administrator,

From "Feedback Form" was sent this message:

    {{ content }}

    Sender's IP adress: {{ ip }}
    Sender's Email: {{ user_email }}
    Sender's Nickname: {{ user_nickname }}

Django Blog Project
{% endautoescape %}
```

Рис. А.8.6. файл send_feedback_email.html– шаблон листа-фідбека