m-math_2025_006

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
кафедра математичного моделювання

СЕРВІС ОПТИМІЗАЦІЇ САЙТІВ ДЛЯ ПОШУКОВИХ СИСТЕМ: БЕКЕНД ТА ГЕНЕРАЦІЯ КОНТЕНТУ

Кваліфікаційна робота
Рівень вищої освіти – другий (магістерський)

Виконав:
студент 6 курсу, 601 групи
Лазарєв Володимир Олександрович
Керівник:
доцент Перцов А.С.

До захисту допущено
на засіданні кафедри
протокол № 5 від 26 листопада 2024 р.
Зав. кафедрою _____ проф. Черевко І.М.

Чернівці – 2024

Анотація

Розроблено інструмент для автоматизованої генерації SEO-контенту, який сприяє покращенню видимості вебсайтів у пошукових системах завдяки створенню індексованих статей, взаємопов'язаних для підвищення популярності й релевантності контенту, основним функціоналом є автоматизація процесу створення статей: від аналізу контексту запитів і генерації пропонованих заголовків до формування структурованих нарисів, генерації статей та розміщення матеріалів на системах управління контентом (CMS). Було використано такі технології, як: мова програмування Python, фреймворк Django, OpenAI API, інструмент контейнеризації Docker, база даних RDS, асинхронна черга задач Celery.

***Ключові слова*:** генерація, SEO-контент, автоматизація, CMS, OpenAI API, Python, Django, Docker, RDS.

Abstract

A tool for automated generation of SEO content has been developed that helps to improve the visibility of websites in search engines by creating indexed articles interconnected to increase the popularity and relevance of content, the main functionality is to automate the process of creating articles: from analysing the context of queries and generating proposed titles to forming structured outlines, generating articles and placing materials on content management systems (CMS). The technologies used are: Python programming language, Django framework, OpenAI API, Docker containerisation tool, RDS database, Celery asynchronous task queue.

Keywords: generation, SEO content, automation, CMS, OpenAI API, Python, Django, Docker, RDS.

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

_____Лазарєв В.О.

(підпис)

Зміст

Вступ

У сьогоднішньому світі, де онлайн присутність компаній має велике значення у досягненні бізнес-цілей, значення оптимізації контенту для пошукових систем (SEO) важко переоцінити. З кожним роком зростає конкуренція за увагу користувачів у цифровому просторі, що спонукає компанії інвестувати у створення високоякісного контенту, який би не тільки приваблював цільову аудиторію, а й забезпечував високі позиції серед результатів пошукових запитів.

Ефективною маркетинговою стратегією для таких компаній є автоматизація створення SEO-орієнтованих статей, котрі за рахунок їх структури та вмісту підвищують популярність вебсайтів. Автоматизовані інструменти дають змогу створювати значний обсяг оптимального контенту, що сприяє покращенню видимості сайту в пошукових системах, підвищенню рівня органічного трафіку та залученню нових відвідувачів.

Актуальність дослідження. Оптимізація вебсайту для пошукових систем є одним із найбільш результативних способів підвищення трафіку. Проте багато компаній стикаються з проблемами масштабування контенту, оскільки створення статей вручну є довготривалим процесом, що вимагає значних ресурсів. Автоматизація процесів генерації та публікації контенту здатна забезпечити значну ефективність і продуктивність для компаній різного масштабу.

Мета. Метою кваліфікаційної роботи є створення ефективного інструменту для автоматизованого генерування, оптимізації та публікації SEO-орієнтованих статей, який би забезпечував високу якість контенту та підвищував релевантність сайтів у пошукових системах. Розроблений

інструмент має спростити процес створення SEO-контенту шляхом автоматизації ключових етапів: збору контекстної інформації на основі запитів, складання структурованих нарисів статей, вибору ключових слів і автоматичного розміщення статей на підтримуваних платформах CMS.

Завдання роботи:

- Розробити механізми автоматизованої генерації контенту на основі запитів користувача з урахуванням SEO-оптимізації.
- Реалізувати багатокроковий процес генерації, включаючи парсинг контекстів запитів, формування заголовків, нарисів, а також планування публікацій.
- Забезпечити інтеграцію інструменту з популярними CMS-платформами, що дозволить автоматично завантажувати та публікувати статті.

Об’єкт дослідження – процеси автоматизованого створення та публікації SEO-контенту для вебсайтів.

Предмет дослідження – функціональність інструменту автоматизованого створення SEO-контенту, зокрема алгоритми генерації контенту, інтеграція з CMS, використання багатопотоковості для обробки запитів.

Практична значущість роботи полягає у можливості використання розробленого інструменту компаніями для підвищення ефективності створення SEO-контенту та його подальшої індексації у пошукових системах. Використання автоматизованих рішень дозволяє значно спростити процес підготовки контенту, оптимізувати ресурсні витрати та підвищити якість управління інформаційним наповненням вебсайтів.

1 Переваги, актуальність та область застосування

1.1 Вплив SEO-контенту на популярність вебсайтів

SEO-контент є одним із факторів, що визначає успішність вебсайту у сучасному цифровому середовищі. Основна мета створення такого контенту полягає у залученні цільової аудиторії та збільшенні органічного трафіку через підвищення видимості вебсайту в пошукових системах. Це досягається завдяки використанню релевантного до теми статті контексту та оптимізованій структурі тексту.

Ефективний SEO-контент сприяє покращенню рейтингу вебсайту у результатах пошуку, що безпосередньо впливає на залучення нових відвідувачів. Більшу частину кліків та активних дій від користувачів отримують перші декілька результатів пошуку, тоді як інші позиції втрачають значну частину потенційного трафіку. Таким чином, створення оптимізованого контенту стає очевидним кроком для розвитку онлайн-присутності вебсайту.

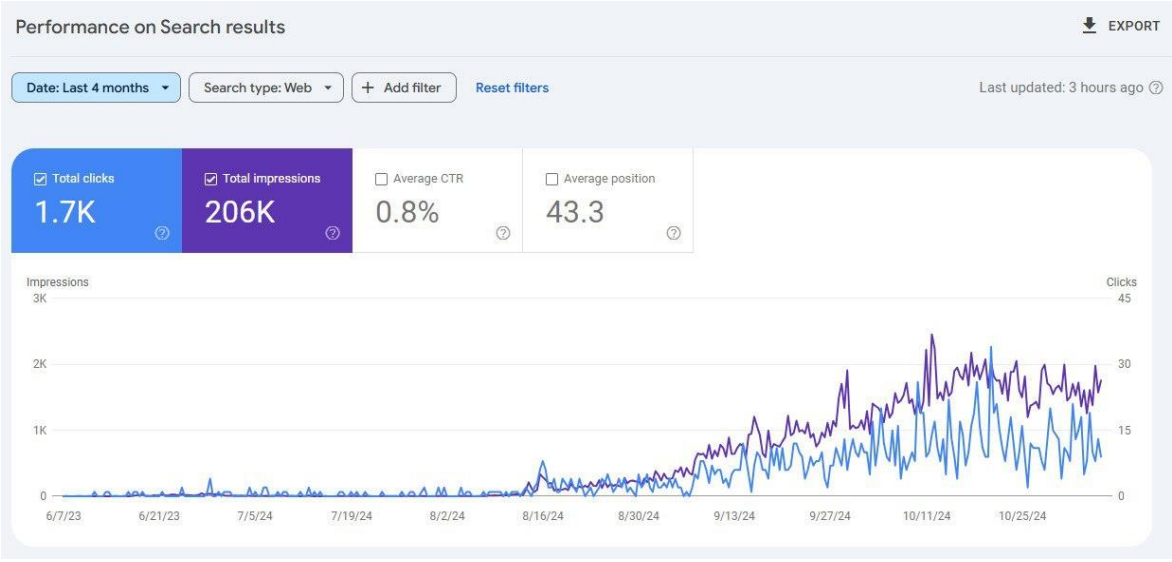


Рис. 1.1 Статистика кліків сайту на якому застосовується система оптимізації в пошукових запитах

Окрім підвищення видимості, якісний SEO-контент забезпечує наступні переваги для вебсайту:

Підвищення конверсій: Користувачі, які знаходять вебсайт через пошукові системи, зазвичай мають чітко визначений інтерес, що збільшує ймовірність виконання цільової дії, наприклад покупки чи підписки.

Покращення взаємодії з користувачем: Оптимізований контент має логічну структуру, містить релевантну інформацію та мультимедійні елементи, які сприяють зручності сприйняття.

Побудова довіри: Висока позиція у результатах пошукової системи додає користувачу довіру, враження надійності та авторитетності ресурсу.

1.2 Переваги автоматизації створення SEO-контенту над ручним створенням

Створення SEO-контенту вручну пов’язане з низкою викликів. Зокрема, процес потребує значних витрат часу та ресурсів адміністраторів вебсайту, особливо якщо мова йде про масштабне оновлення інформації або вебсайти, що підтримують багато мов. Тому автоматизація генерації SEO-контенту стає хорошим рішенням для багатьох компаній, дозволяючи знизити витрати на створення контенту водночас покращивши ефективність свого вебсайту.

Іншою важливою перевагою автоматизованого підходу є масштабованість. Наприклад, компанія, яка публікує контент на багатомовному вебсайті, може автоматично генерувати статті для різних регіонів та аудиторій одночасно. Це допомагає забезпечити регулярне оновлення інформації, необхідне для підтримки високих позицій у пошукових системах.

Серед переваг автоматизації можна назвати такі:

- **Швидкість виконання**: Автоматизована генерація статей скорочує час створення матеріалу з кількох годин до кількох хвилин.
- **Масштабованість**: Можливість обробки великої кількості запитів одночасно, що недоступно при ручному створенні.
- **Послідовність якості**: Всі тексти відповідають заданим стандартам та SEO-вимогам.
- **Гнучкість налаштувань**: Системи дозволяють адаптувати стиль і структуру контенту для різних аудиторій.
- **Інтеграція з CMS**: Автоматична публікація статей виключає потребу у ручному завантаженні.

Ще однією перевагою автоматизації є можливість використовувати передові технології, такі як штучний інтелект. Наприклад, моделі GPT забезпечують не лише генерацію тексту, але й його оптимізацію для підвищення читабельності та відповідності очікуванням користувача. Інтеграція таких інструментів дозволяє досягти рівня якості, який раніше вимагав ручного втручання експертів.

Завдяки цим перевагам автоматизація створення SEO-контенту не лише прискорює процес, але й дозволяє організаціям ефективно масштабувати свої зусилля, забезпечуючи стабільне зростання онлайн-присутності.

1.3 Актуальність проєкту та область використання

Автоматизація створення та публікації SEO-контенту має попит в умовах постійного зростання конкуренції у цифровому середовищі. Вебсайти всіх масштабів — від персональних блогів до великих корпоративних платформ — прагнуть покращити свою видимість у пошукових системах. Для цього вони інвестують значні ресурси у розробку якісного контенту,

оптимізованого під вимоги пошукових алгоритмів. Однак традиційні методи створення контенту, що базуються на ручній роботі, мають суттєві обмеження. Вони є повільними, затратними та складними у масштабуванні, особливо для компаній, які працюють із багатомовними аудиторіями або значними обсягами даних.

Автоматизація вирішує ці проблеми, дозволяючи створювати великий обсяг якісного контенту з мінімальними затратами часу та ресурсів. Сучасні технології забезпечують створення статей, які відповідають вимогам SEO, мають чітку структуру, включають релевантні ключові слова та навіть мультимедійні елементи. Такі рішення стають важливими інструментами для бізнесу, особливо в галузях, де цифровий маркетинг є основним каналом залучення клієнтів.

Проєкт автоматизації створення SEO-контенту знаходить своє застосування у багатьох сферах. Найбільш очевидною областю використання є медіа-ресурси. Новинні сайти, блоги, тематичні сайти онлайн ресурсів можуть використовувати автоматизовані системи для швидкого створення статей, що охоплюють різноманітні теми пов’язані із певними інтересами аудиторії. Не менш актуальною є автоматизація для освітніх платформ, які надають інформаційний контент у вигляді статей, гайдів та інструкцій.

Таким чином, актуальність розробки автоматизованих рішень для створення SEO-контенту зумовлена не лише зростанням попиту на якісний контент, але й необхідністю зниження витрат часу і ресурсів. Завдяки широкій області використання, такі інструменти стають критично важливими для компаній, які прагнуть посилити свою конкурентну позицію в умовах цифрової економіки.

1.4 Системи управління контентом як платформи для оптимізації в пошукових системах

Системи управління контентом (CMS) відіграють ключову роль у створенні та публікації матеріалів, а також у забезпеченні їхньої видимості в пошукових системах. Завдяки вбудованим SEO-інструментам, CMS значно спрощують оптимізацію контенту та автоматизують важливі етапи підготовки матеріалів для індексації.

До основних можливостей CMS для оптимізації відносяться:

- автоматичне створення та оновлення XML-карт сайту, що спрощує індексацію сторінок;
- налаштування метаданих (заголовків, описів, ключових слів) безпосередньо у кожній статті;
- інтеграція з плагінами для аналізу читабельності тексту, ключових слів і швидкості завантаження сторінок;
- підтримка багатомовності, що дозволяє оптимізувати контент для різних регіональних аудиторій.

Такі системи, як WordPress, WebFlow та Strapi, відзначаються своїми унікальними особливостями. WordPress забезпечує простий доступ до численних плагінів для SEO-оптимізації, WebFlow дозволяє гнучко налаштовувати SEO-параметри через кастомізацію HTML-коду, а Strapi підтримує інтеграцію через REST API, що особливо зручно для автоматизації публікації контенту.

Завдяки таким можливостям CMS стають незамінними інструментами для SEO-оптимізації, спрощуючи управління контентом і забезпечуючи відповідність сучасним вимогам пошукових систем.

2 Використані технології та засоби реалізації

2.1 Мова програмування Python

Python є однією з найпопулярніших мов програмування у світі завдяки її простоті, гнучкості та широким можливостям. Вона забезпечує розробникам зручний синтаксис, який легко засвоїти навіть новачкам, і водночас має потужний набір інструментів для вирішення складних завдань. Python використовується у багатьох сферах, включаючи розробку вебдодатків, аналіз даних, машинне навчання, автоматизацію процесів і навіть створення ігор.

Однією з найважливіших переваг Python є його велика і активно підтримувана спільнота. Завдяки цьому для мови доступний широкий спектр бібліотек та фреймворків, які дозволяють розширювати її функціональність і скорочувати час розробки проєктів.

Python також є кросплатформною мовою, тобто код, написаний на Python, може виконуватися на будь-якій операційній системі, яка має встановлений інтерпретатор мови. Це дозволяє створювати універсальні рішення без необхідності адаптації коду для різних платформ.

У проєкті автоматизації створення та публікації SEO-контенту Python використовується як основна мова програмування завдяки своїй зручності та багатій екосистемі бібліотек. Зокрема, Python забезпечує:

- реалізацію бізнес-логіки через потужний синтаксис і широкий набір інструментів;
- інтеграцію з API, такими як OpenAI API, Google Ads API та Google Custom Search API, завдяки підтримці REST-запитів і бібліотек для роботи з мережевими протоколами;
- реалізацію багатопотокових процесів через асинхронні бібліотеки, що сприяє оптимізації продуктивності системи.

Застосування Python у цьому проєкті дозволило забезпечити ефективність розробки, спростити інтеграцію з зовнішніми сервісами та забезпечити високу продуктивність навіть при роботі з великим обсягом даних. Його універсальність та простота зробили його ідеальним вибором для побудови автоматизованої системи.

2.2 Фреймворк Django та DRF

Django є одним із найпопулярніших вебфреймворків для мови програмування Python. Його основною метою є забезпечення швидкої, надійної та безпечної розробки вебдодатків завдяки високому рівню автоматизації та великій кількості вбудованих інструментів. Django надає розробникам змогу працювати з більшістю необхідних функцій без додаткових налаштувань. Це включає управління базами даних, налаштування маршрутизації, а також інструменти для адміністрування моделей додатку.

Фреймворк Django Rest Framework (DRF) доповнює функціональність Django, додаючи можливість створювати API для взаємодії між компонентами системи або зовнішніми сервісами. DRF дозволяє легко створювати RESTful API, що забезпечує стандартизовану передачу даних через HTTP-запити. Інструменти, такі як серіалізатори, автоматична документація та готові класи для обробки запитів, значно спрощують реалізацію взаємодії з іншими системами.

У проєкті автоматизації створення та публікації SEO-контенту Django використовується для реалізації бекенд-логіки, управління базою даних та маршрутизації. Використання DRF дозволило створити ефективні API для роботи з користувацькими запитами, генерацією контенту, а також інтеграції з платформами CMS. Зокрема, фреймворки забезпечують наступні можливості:

- управління базою даних за допомогою ORM Django, що дозволяє працювати з даними на високому рівні абстракції без написання складних SQL-запитів;
- розробку API для інтеграції з зовнішніми сервісами, такими як OpenAI API або Google Custom Search API, для збору й генерації контенту;
- створення інтерфейсів для управління проєктами, запитами та запланованими завданнями;
- автоматичне створення документації для API через інструменти DRF, що спрощує роботу розробників і взаємодію з клієнтами.

Використання Django та DRF також сприяє підвищенню безпеки проєкту, оскільки ці фреймворки включають захист від найбільш поширених вебзагроз.

2.3 Асинхронна черга задач Celery

Celery є одним із найбільш популярних інструментів для управління асинхронними чергами задач у Python-додатках. Це потужна бібліотека, яка дозволяє виконувати завдання у фоновому режимі, зменшуючи навантаження на основний сервер. Для використання Celery потрібен брокер повідомлень, серед них найчастіше використовується Redis або RabbitMQ, що забезпечує швидку передачу даних між компонентами системи.

Основна перевага Celery полягає у його здатності виконувати складні обчислення або тривалі операції, такі як обробка даних, надсилання повідомлень чи виконання HTTP-запитів, без блокування основного потоку програми. Завдяки цьому користувачі отримують швидкий відгук від системи, навіть якщо певна задача ще виконується у фоновому режимі.

Celery оптимізує продуктивність системи за рахунок розподілу обчислювальних ресурсів і використання асинхронного підходу. Наприклад, коли користувач планує кілька статей для генерації одночасно, завдання додаються до черги і виконуються незалежно одне від одного. Це забезпечує масштабованість системи та дозволяє обробляти великі обсяги даних навіть під великими навантаженнями.

Інтеграція Celery із Django здійснюється за допомогою окремого модуля, що спрощує налаштування завдань і обробку результатів. Завдяки цьому можна легко додавати нові завдання або змінювати параметри існуючих без значних витрат часу.

Застосування Celery у цьому проєкті стало критично важливим для забезпечення швидкої та стабільної роботи системи, особливо при виконанні тривалих операцій. Це дозволило створити гнучку й ефективну інфраструктуру для управління складними процесами в автоматизованій системі.

2.4 База даних Amazon RDS

Amazon Relational Database Service (Amazon RDS) є хмарним сервісом, що надає зручний і масштабований спосіб керування реляційними базами даних. Цей сервіс автоматизує завдання налаштування, резервного копіювання, оновлення програмного забезпечення та масштабування, дозволяючи зосередитись на логіці додатка, а не на управлінні інфраструктурою бази даних.

Однією з головних переваг Amazon RDS є підтримка кількох популярних систем управління базами даних, таких як MySQL, PostgreSQL, MariaDB, Oracle та Microsoft SQL Server. Це дає змогу обрати оптимальну платформу залежно від вимог проєкту. У випадку автоматизації створення та

публікації SEO-контенту, основною системою управління базами даних виступає PostgreSQL завдяки її потужності, надійності та підтримці розширених функцій.

Amazon RDS забезпечує:

- автоматичне резервне копіювання даних з можливістю швидкого відновлення;
- масштабування ресурсів бази даних залежно від навантаження;
- високу доступність завдяки реплікації та автоматичному переключенню у разі збоїв;
- інтеграцію із засобами моніторингу, такими як Amazon CloudWatch, для відстеження різних метрик бази даних.

Використання Amazon RDS дозволяє забезпечити високу продуктивність та надійність бази даних навіть при значному навантаженні.

2.5 Інструмент контейнеризації Docker

Docker є найпопулярнішим інструментом для контейнеризації, який дозволяє створювати, розгортати та запускати додатки у стандартизованих ізольованих середовищах, що називаються контейнерами. Використання контейнерів спрощує процес розробки, тестування та розгортання програмного забезпечення, забезпечуючи його стабільність незалежно від середовища, у якому воно виконується.

Контейнери Docker включають усі необхідні залежності, такі як бібліотеки, середовище виконання та налаштування, що робить їх портативними та зручними для масштабування. Крім того, Docker дозволяє

уникнути проблем із сумісністю середовищ, оскільки контейнер працює однаково на будь-якому сервері або хмарній платформі.

У проєкті автоматизації створення та публікації SEO-контенту Docker використовується для:

- ізоляції основних компонентів системи, таких як бекенд і черга задач Celery;
- створення стандартизованого середовища розробки;
- спрощення розгортання системи на локальних серверах.

Проєкт містить конфігураційні файли Docker (`Dockerfile` та `docker-compose.yaml`), які описують, як має бути зібрано і запущено контейнер.

Використання Docker забезпечує низку переваг, серед яких стабільність роботи, спрощення управління залежностями та можливість швидкого відновлення системи у разі збоїв.

2.6 OpenAI API

OpenAI API є одним із інструментів для доступу до текстових моделей, що використовуються для створення та обробки контенту. Цей сервіс надає доступ до потужних мовних моделей, таких як GPT-4, які здатні генерувати текст, відповідати на запити, аналізувати контекст і виконувати складні завдання, пов'язані з обробкою природної мови.

API дозволяє налаштовувати параметри генерації тексту, такі як довжина, тон і стиль, що робить його ефективним для різноманітних сценаріїв. Завдяки цьому сервісу можна створювати високоякісний контент, який відповідає специфічним вимогам користувачів або завдань.

У проєкті автоматизації створення та публікації SEO-контенту OpenAI API використовується для:

- генерації заголовків і нарисів статей на основі контексту;
- створення текстового контенту для статей із урахуванням заданих ключових слів і SEO-вимог;
- рефакторингу тексту, який включає покращення його читабельності, структури та додавання списків чи виділених фраз;

Корисним аспектом OpenAI API є можливість контролю витрат за допомогою системи токенів, що забезпечує гнучкість у плануванні бюджету на генерацію контенту. Крім того, бібліотека OpenAI для Python значно спрощує роботу з API, дозволяючи легко створювати запити та обробляти отримані результати.

Завдяки OpenAI API у проєкті вдалося забезпечити високий рівень автоматизації та якісну генерацію контенту, що відповідає сучасним вимогам SEO.

2.7 Google Ads API та Google Custom Search API

Google Ads API та Google Custom Search API є інструментами, які забезпечують доступ до ресурсів пошукової системи Google і даних, що використовуються для створення ефективного SEO-контенту. У проєкті автоматизації ці сервіси працюють у тісній інтеграції, виконуючи різні, але взаємодоповнюючі функції.

Google Ads API дозволяє отримувати ключові слова та детальну інформацію про них у вигляді кількості пошукових запитів з цим ключовим словом на місяць та конкуренцію цього слова. Цей API є критично важливим для підбору релевантних ключових слів, які використовуються у статтях,

оскільки саме вони визначають, наскільки ефективно контент буде індексуватися пошуковими системами.

У межах проєкту API застосовується для:

- отримання пропозицій ключових слів на основі запитів користувача;
- оцінки їхньої популярності та релевантності для цільової аудиторії;
- формування списку ключових слів для подальшого використання у текстах і заголовках статей.

Google Custom Search API забезпечує доступ до результатів пошуку Google, дозволяючи отримувати структуровані дані про вміст найпопулярніших сторінок за заданими запитами. У проєкті цей API використовується для парсингу контексту, необхідного для генерації статей. Завдяки цьому API система автоматично аналізує найкращий контент за обраною темою, формуючи базу для створення релевантних текстів.

Основні завдання, які вирішуються за допомогою Google Custom Search API:

- отримання JSON-даних про популярні сторінки за запитами користувачів;
- аналіз контенту для виділення ключових тем і підходів до його структурування;

Інтеграція цих API забезпечує синергію, що дозволяє автоматизувати ключові етапи підготовки SEO-контенту.

3 Архітектура та функціональні можливості

3.1 Загальна структура та основні модулі системи

3.1.1 Логіка поділу системи на модулі

Проєкт має чітку структуру, яка дозволяє окремо обробляти основні процеси, такі як генерація контенту, планування задач, інтеграція з системами керування контентом (CMS) і управління користувачами.

Основні модулі системи:

1. **Модуль генерації контенту**

Цей модуль забезпечує створення статей, їх заголовків та нарисів. Основні компоненти:

- Генератор заголовків статті.
- Генератор нарисів статті.
- Абстрактний клас генерації тіла статті, який забезпечує можливість подальшого розширення форматів.
- Класи генерації для різних форматів тіла статті, зокрема Markdown та RichText.

2. **Модуль планування задач**

Відповідає за налаштування та управління розкладом виконання задач. Реалізований з використанням Celery та Celery Beat.

Модуль дозволяє:

- Додавати нові задачі до черги.
- Змінювати налаштування задач.

- Вручну запустити виконання певної задачі.
- Змінити час виконання задач.
- Обробляти задачі в асинхронному режимі.

3. **Модуль управління проєктами**

Цей модуль відповідає за зберігання інформації про проєкти користувачів, а також за інтеграції з різними системами керування контентом (CMS), такими як WordPress, Webflow та Strapi. Основні задачі що виконує цей модуль:

- Створення та налаштування проєктів для інтеграції з CMS.
- Валідація даних доступу до CMS.
- Отримання списків колекцій CMS.
- Налаштування полів для публікації.
- Завантаження контенту статей до CMS.
- Управління токенами для контролю витрат ресурсів.

Модулі системи спроєктовані таким чином, щоб забезпечувати мінімальну залежність один від одного. Для комунікації використовуються:

- REST API між зовнішніми системами (CMS, сервіси OpenAI, Google Ads тощо) і бекендом.
- Внутрішні сигнали Django для передачі даних між модулями.
- Черги Celery для обробки задач у багатопотоковому режимі.

Поділ на модулі забезпечує гнучкість системи, її масштабованість і можливість легкого розширення функціоналу.

3.1.2 Взаємодія модулів через API

Взаємодія між модулями системи побудована на використанні внутрішніх та зовнішніх API, що забезпечує високий рівень узгодженості роботи компонентів і гнучкість у розширенні функціональності. API дозволяють абстрагувати логіку окремих модулів, забезпечуючи їхню взаємодію через стандартизовані запити та відповіді.

Внутрішні API забезпечують комунікацію між різними частинами бекенд-системи. Наприклад, при створенні нових завдань на генерацію контенту ViewSets у модулі REST Framework отримують запити від клієнта, перевіряють їхню валідність через серіалізатори та ініціюють відповідні обчислення через сервіси. Сервіси виконують бізнес-логіку, звертаючись до бази даних, інших модулів або зовнішніх API. Celery використовується для асинхронної обробки задач: модулі взаємодіють через черги повідомлень Redis, що дозволяє обробляти великі обсяги запитів без блокування основного процесу.

Зовнішні API відповідають за інтеграцію із сторонніми сервісами, такими як CMS (WordPress, Webflow, Strapi), а також платформами Google Ads, Google Custom Search і OpenAI. Наприклад, для публікації контенту в CMS використовуються стандартизовані ендпойнти, які приймають підготовлені дані у форматі JSON і здійснюють валідацію запитів. Інтеграція з Google Ads API забезпечує отримання ключових слів, а Google Custom Search API використовується для збору контекстної інформації для генерації статей.

Особливістю побудови API є ретельна валідація вхідних даних і підтримка авторизації. Для роботи із зовнішніми сервісами застосовуються API-ключі або HTTP Basic Auth, що гарантує безпеку доступу до сторонніх систем. Крім того, в разі виникнення помилок інтеграції система надає докладні повідомлення, які спрощують налагодження.

Таким чином, використання API як основного механізму взаємодії забезпечує модульність системи, зручність інтеграції з іншими сервісами та гнучкість у розробці нових функціональних можливостей.

3.1.3 Структура бази даних і ключові моделі

База даних системи побудована на реляційній моделі, що дозволяє ефективно зберігати, обробляти та пов’язувати дані, необхідні для генерації контенту та управління завданнями. Ключові моделі відображають логіку процесів і тісно інтегровані між собою.

Основні моделі:

1. **Project**

   Зберігає інформацію про проєкти користувачів, включаючи назву, тип CMS (WordPress, Webflow, Strapi), API-ключ, базовий URL API, а також налаштування для мапінгу полів, необхідних для інтеграції.

2. **ArticleName**

   Містить назви статей, що генеруються на основі запитів користувача. Основні поля: назва, опис контексту, список запитів та категорія статті. Пов’язана з проєктом через зовнішній ключ.

3. **Outline**

   Зберігає нариси статей, включаючи структуру заголовків, ключові слова та рекомендовані основні зображення. Пов’язана з ArticleName.

4. **ArticleTask**

   Описує завдання на генерацію статей. Поля включають статус, час запланованого виконання та посилання на відповідний Outline.

5. **Article**

   Містить кінцеві статті зі згенерованим текстом і мультимедійним контентом. Пов’язана із завданнями ArticleTask.

Зв’язки між моделями виконані через зовнішні ключі, що забезпечує узгодженість даних. Наприклад, один проєкт може мати кілька статей, а одна стаття може мати кілька ключових слів. Дивитись додаток А. Така структура забезпечує високу продуктивність і гнучкість системи.

3.2 Функціональні можливості: генерація, планування, інтеграція з API систем керування контентом

3.2.1 Генерація статей: підтримувані формати та стилі

Генерація статей є ключовою функцією системи, яка спрямована на створення якісного контенту, оптимізованого для різних платформ. Система підтримує два формати статей.

Підтримувані формати статей:

1. **Розширені оптимізовані статті**

   Найбільш поширений формат, орієнтований на публікацію в блогах чи інформаційних ресурсах. Генерується структурований текст, що включає заголовки, підзаголовки, абзаци та ключові слова. Додається мультимедійний контент, зокрема зображення, що підбираються відповідно до контексту статті. У цьому форматі приділяється особлива увага використанню ключових слів, метаданих статті та заголовків, що сприяє покращенню позицій в пошукових системах.
2. **Статті новин**

   Компактні за розміром тексти, призначені для новинних сайтів або соціальних мереж. Додається декілька відповідних до контексту статті картинок та мінімум метаданих.

Гнучкість налаштувань:

Користувачі мають змогу задавати параметр формату статті індивідуально до кожного проєкту. Це включає вибір довжини тексту, структури заголовків, кількості зображень та ключових слів. Генерація статей відбувається з використанням OpenAI API, що дозволяє забезпечити високу якість тексту.

3.2.2Планування задач для автоматизованої публікації

Планування задач для автоматизованої публікації є ключовою функцією системи, яка дозволяє користувачам налаштовувати публікацію контенту відповідно до заданого розкладу. Ця функція оптимізує управління контентом та забезпечує зручність автоматизації процесу розміщення статей на різних платформах.

Основні етапи планування:

1. **Створення задачі**
   Для кожної статті створюється задача (Article Task), яка містить нарис, інформацію про статтю, запланований час публікації та пов'язані дані, такі як формат і платформа.
2. **Вибір часу публікації**
   Користувач задає конкретний час або інтервал для публікації. Система автоматично перевіряє можливість виконання задачі в заданий час, враховуючи наявність інших запланованих задач і доступні ресурси.
3. **Інтеграція з Celery Beat**
   Планування задач здійснюється за допомогою Celery Beat, який відповідає за управління чергами задач і запуск їх у визначений час. Celery Beat додає задачі до черги Celery Worker, де вони виконуються відповідно до встановлених параметрів.

4. **Гнучке налаштування розкладу**

   Система дозволяє редагувати або переналаштовувати час публікації навіть після створення задачі. Це особливо корисно для динамічного середовища, де терміни публікацій можуть змінюватися.

3.2.3 Інтеграція з CMS: Webflow, WordPress, Strapi

Інтеграція з системами управління контентом (CMS) є важливим компонентом проєкту, який дозволяє автоматизувати публікацію статей безпосередньо на платформах Webflow, WordPress і Strapi. Кожна з цих CMS має власні особливості, і система забезпечує гнучку підтримку для всіх трьох платформ, адаптуючись до їхніх специфікацій через API.

Webflow

Інтеграція з Webflow реалізована через API, який дозволяє додавати статті до колекцій Webflow. Користувачеві необхідно вказати ключ API, ID сайту та ID колекції, у яку будуть додаватися записи. Система дозволяє завантажити статтю як чернетку або як опубліковану. Крім текстового контенту, Webflow дозволяє додавати мультимедійні файли, які автоматично завантажуються через інтегрований механізм завантаження.

WordPress

Для роботи з WordPress використовується REST API, який вимагає базової авторизації через пару "ім'я користувача – пароль". Користувач може створювати записи, вказуючи такі параметри, як заголовок, текст статті, зображення, метадані та статус публікації (опубліковано чи чернетка).

Strapi

Strapi відрізняється своєю гнучкістю та дозволяє працювати з колекціями записів через REST API. Для інтеграції потрібен URL за яким знаходиться API сервісу користувача та ключ доступу до цього API. Процес включає створення

нових записів у колекціях, передачу текстового та мультимедійного контенту, а також управління статусом запису. Крім того, Strapi дозволяє виконувати перевірку прав доступу до колекцій, забезпечуючи додатковий рівень безпеки.

3.2.4 Огляд реалізованих ендпойнтів для роботи з функціоналом

Ендпойнти, реалізовані в системі, забезпечують доступ до всього функціоналу генерації, планування та публікації контенту. Вони структуровані відповідно до основних модулів системи, таких як управління проєктами, генерація статей, нарисів, задач, а також інтеграція з CMS.

Управління проєктами

Користувачі можуть створювати, оновлювати та видаляти проєкти через набір REST API ендпойнтів. Наприклад, ендпойнт */api/projects/* дозволяє створювати нові проєкти, де обов'язкові параметри залежать від обраної CMS. Для перевірки валідності конфігурацій доступний окремий ендпойнт */api/projects/validate/*.

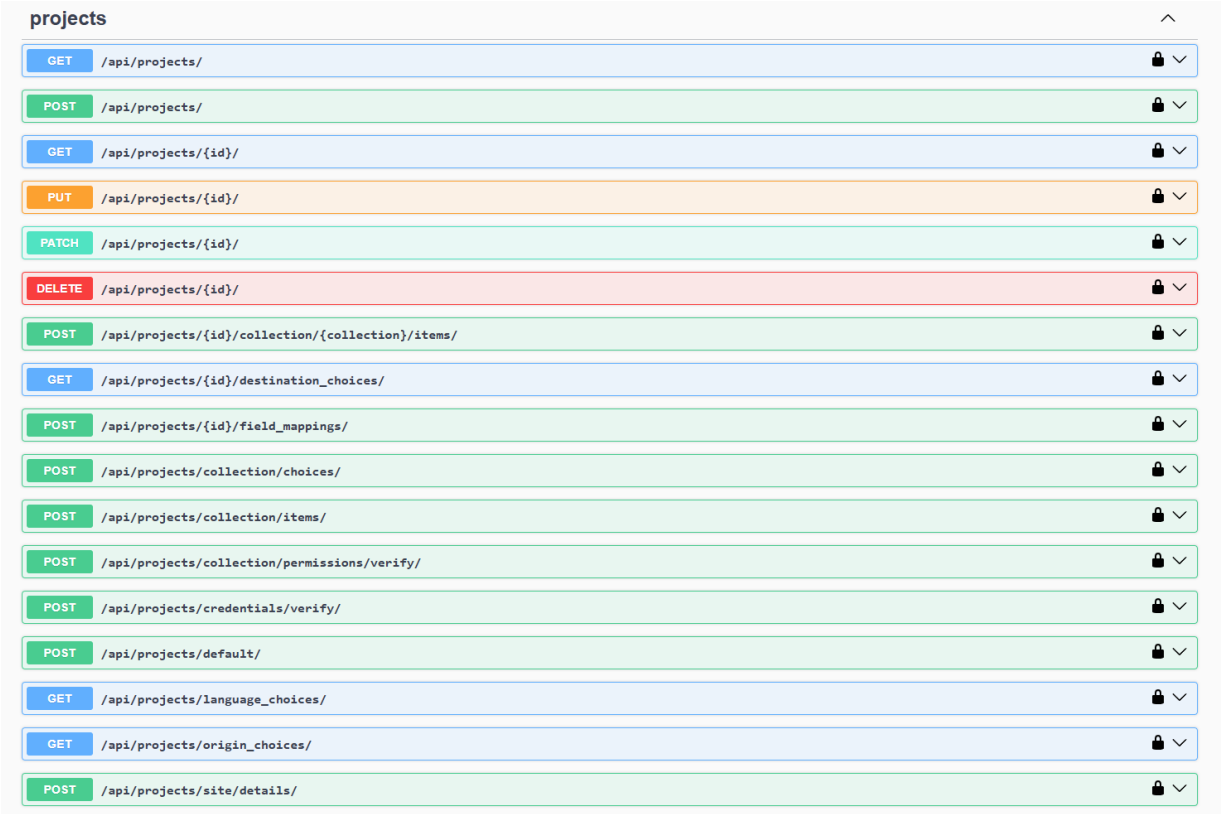


Рис.3.1 Список ендпойнтів для налаштування проєктів

Генерація назв

Користувачі можуть робити CRUD операції над назвами статей через ендпойнти назв */api/names/*. Користувачі можуть завантажувати списки ключових слів через ці ендпойнти, щоб отримати згенеровані назви майбутніх статей. Реалізована підтримка пакетної обробки для роботи з великими обсягами даних.

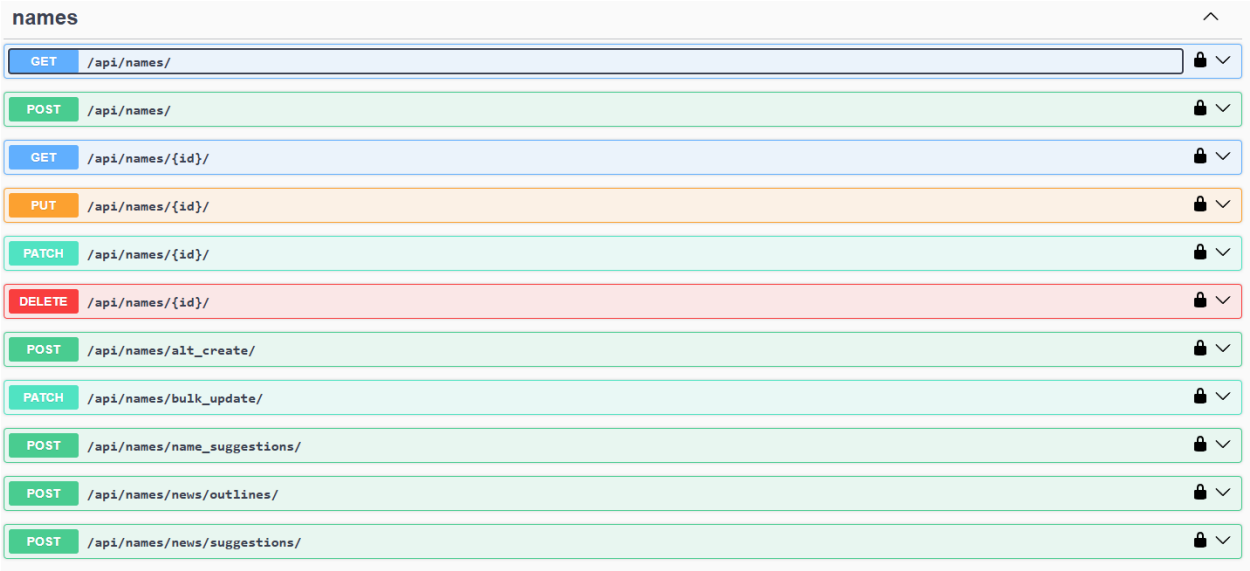


Рис. 3.2 Список ендпойнтів для опрацювання назв статей

Генерація нарисів

Користувачі можуть робити CRUD операції над нарисами статей через ендпойнти */api/outlines/*. Користувачі можуть завантажувати списки контекстів через ці ендпойнти, щоб отримати згенеровані структури майбутніх статей.

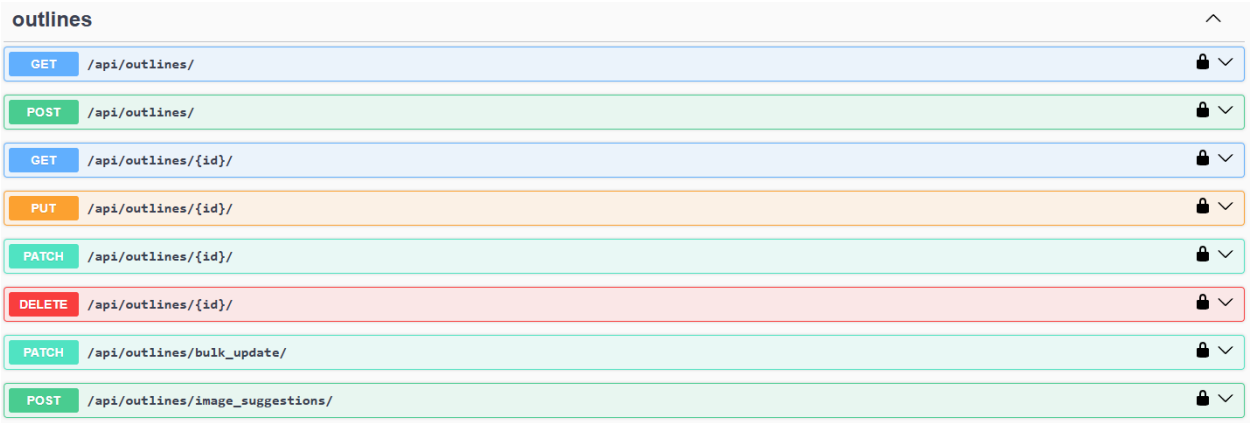


Рис. 3.3 Список ендпойнтів для опрацювання нарисів статей та ключових слів

Планування задач

Для створення і планування задач публікації використовується ендпойнт */api/tasks/*, який приймає нариси статей та параметри розкладу. Додаткові ендпойнти, такі як */api/tasks/approve/* та */api/tasks/reschedule/*, дозволяють затверджувати чи змінювати розклад задач. Дивитись додаток Б.

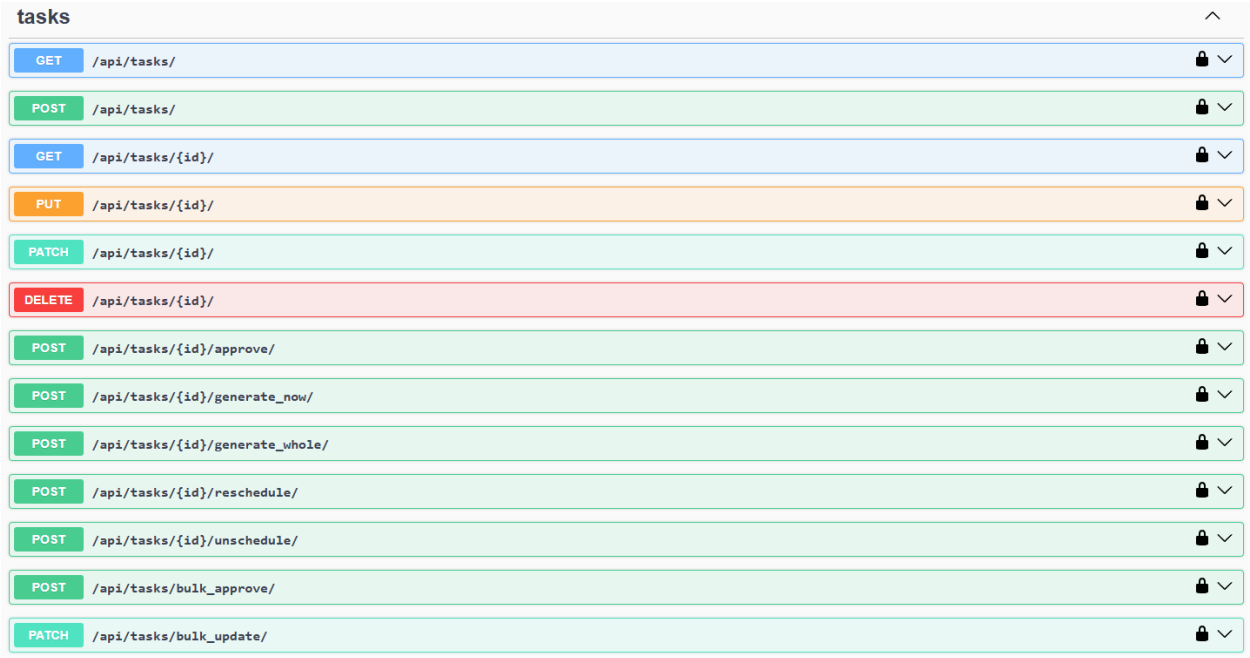


Рис. 3.4 Список ендпойнтів для роботи з запланованими задачами

3.3 Багатопотокова обробка запитів

3.3.1 Використання Celery для асинхронної обробки

Асинхронна обробка задач у системі реалізована за допомогою бібліотеки **Celery**, яка забезпечує ефективне виконання тривалих або ресурсомістких операцій у фоновому режимі. Celery інтегрується з Redis, брокером повідомлень, що дозволяє розподіляти задачі між воркерами в реальному часі.

Основні задачі Celery

Celery використовується для автоматизації процесів генерації та публікації контенту. Серед ключових задач:

- Генерація текстів статей на основі нарисів.
- Збагачення контенту за допомогою ключових слів і мультимедійного матеріалу.
- Планування та виконання публікацій у CMS.
- Моніторинг статусу задач для забезпечення стабільності процесу.

Взаємодія компонентів

Процес асинхронної обробки починається зі створення задачі через REST API. Наприклад, користувач надсилає запит на публікацію статті, після чого задача автоматично передається у Celery. Воркери Celery виконують цю задачу, а її результати зберігаються в базі даних.

3.3.2 Використання Thread Pool Executor для багатопотоковості

Для забезпечення ефективної роботи з зовнішніми API та прискорення виконання задач система використовує багатопотоковість через **ThreadPoolExecutor**. Цей інструмент із модуля *concurrent.futures* дозволяє створювати пул потоків для одночасного виконання кількох задач, оптимізуючи витрати часу та ресурсів.

Принцип роботи

ThreadPoolExecutor створює певну кількість потоків, які працюють паралельно, розподіляючи між ними задачі. Такий підхід дозволяє одночасно

виконувати кілька запитів до зовнішніх сервісів або обробляти великі обсяги даних без простоїв. Після завершення задачі потік стає доступним для виконання наступної, що запобігає зайвим витратам ресурсів на створення нових потоків.

Застосування в системі

У системі ThreadPoolExecutor використовується для кількох основних сценаріїв:

- **Запити до зовнішніх API**: Наприклад, при роботі з CMS, як-от Webflow чи Strapi, виконуються паралельні запити для отримання даних про сайти чи колекції.
- **Швидка обробка задач**: Коли кілька запитів надходять одночасно, багатопотоковість дозволяє зменшити затримки в обробці.
- **Масова генерація чи публікація контенту**: У разі одночасної роботи з великими обсягами даних забезпечується стабільна та швидка взаємодія з різними компонентами системи.

3.4 Параметри публікації: підтримувані формати та налаштування

3.4.1 Налаштування параметрів публікації на CMS

Система надає можливість налаштовувати параметри публікації статей на інтегровані CMS, такі як Webflow, WordPress, та Strapi. Ця функціональність дозволяє користувачам адаптувати процес публікації відповідно до специфічних вимог кожної платформи, забезпечуючи гнучкість та ефективність автоматизованого розгортання контенту.

Основні параметри публікації

Під час створення проєкту користувач може вказати ключові параметри, які впливають на процес публікації:

- **Статус публікації**: Можливість обрати між публікацією у статусі "чернетка" або "опубліковано".
- **Структура та поля контенту**: Налаштування відповідності згенерованих полів статті до полів колекції в CMS. Наприклад, у WordPress це можуть бути поля для заголовку, контенту, категорій та тегів, решта згенерованих полів не буде використовуватись.
- **Час публікації**: Використання запланованого часу для автоматичного розміщення контенту.

Особливості для кожної CMS

- **Webflow**: Система дозволяє інтегрувати додаткові параметри, як-от мовні локалі або спеціальні мета-поля (*meta-title, meta-description*), які використовуються для SEO.
- **WordPress**: Публікація підтримує використання рубрик, тегів і спеціальних полів для медіа або налаштувань форматування.
- **Strapi**: Завдяки гнучкій архітектурі API, налаштування можуть включати специфічні структури колекцій та атрибути.

3.4.2 Формати публікацій: Markdown, RichText

Система підтримує два основні формати публікації контенту: **Markdown** та **RichText**. Це дозволяє забезпечити сумісність із різними платформами CMS та адаптувати форматування статей до специфічних потреб кожного проєкту.

Markdown

Markdown є одним із найпопулярніших форматів для створення текстового контенту. Його основною перевагою є простота використання та можливість зберігати текст у читабельному вигляді без додаткових інструментів для рендерингу. Markdown особливо зручний для технічних блогів, документації та статей, які публікуються на платформах із вбудованою підтримкою цього формату.

Під час генерації статті у форматі Markdown система автоматично додає:

- Заголовки (#, ##, ###),
- Абзаци та списки,
- Вставки зображень (![alt text](url)),
- Гіперпосилання ([text](url)).

RichText

RichText — це формат, що дозволяє створювати контент із використанням багатих текстових елементів, таких як:

- Жирний і курсивний текст,
- Таблиці та списки,
- Вбудовані мультимедійні елементи,
- Складні стилі для заголовків і параграфів.

RichText підходить для платформ, які потребують готового HTML-коду або мають вбудовані редактори з розширеним форматуванням, як-от WordPress або Webflow.

Гнучкість вибору формату

Вибір між Markdown та RichText визначається потребами користувача та специфікою платформи. Наприклад, Markdown підходить для публікацій, орієнтованих на швидкість обробки тексту, тоді як RichText забезпечує максимальну візуальну адаптацію контенту.

Підтримка обох форматів дозволяє системі бути універсальною та задовольняти потреби широкого кола користувачів, від технічних блогерів до редакторів складних мультимедійних платформ.

3.4.3 Обробка мультимедійного контенту

Мультимедійний контент є важливим елементом публікацій, який підвищує їхню візуальну привабливість та залучення аудиторії. У рамках системи реалізовано механізми для роботи із зображеннями та іншими мультимедійними елементами, що забезпечує інтеграцію з платформами CMS та спрощує процес наповнення статей.

Завантаження та зберігання зображень

Система автоматично знаходить відповідні зображення для статей за допомогою API Unsplash або Pexels. Пошук базується на ключових словах та тематиці статті, що забезпечує релевантність вибраних зображень. Завантажені зображення обробляються для оптимізації розміру та відповідності вимогам платформ CMS.

Інтеграція з платформами CMS

Під час публікації мультимедійний контент завантажується безпосередньо в CMS. Наприклад, у Webflow зображення додаються як медіа-файли колекції, а у WordPress вони завантажуються до бібліотеки медіа з автоматично згенерованими підписами та описами.

Адаптивність мультимедіа

Системи CMS підтримують автоматичне масштабування та оптимізацію зображень для різних пристроїв. Це дозволяє забезпечити швидке завантаження контенту та високу якість зображень на будь-якій платформі.

Механізм обробки мультимедійного контенту робить систему універсальною та зручною для створення насичених публікацій із мінімальними витратами часу та ресурсів.

4 Програмна реалізація

4.1 Збір і обробка контексту для статей

Збір та обробка контексту для статей є першим і надзвичайно важливим етапом у створенні якісного контенту. Саме на цьому етапі визначаються ключові теми, структура матеріалів і релевантність статей до запитів користувачів. Ефективність цієї фази значною мірою залежить від інтеграції з сучасними API-сервісами, які забезпечують доступ до релевантної інформації та автоматизують її обробку.

У цьому підрозділі розглянуто два основні аспекти процесу: використання Google Custom Search API для пошуку інформації, яка відповідає запитам користувачів, а також генерацію заголовків і нарисів статей на основі зібраного контексту.

4.1.1 Використання Google Custom Search API

Google Custom Search API використовується для збору контексту статті. Цей інструмент забезпечує доступ до великої кількості результатів пошукової системи Google, релевантних до заданих користувачем запитів на створення статей. Це дозволяє скоротити час, необхідний для підготовки контенту, та підвищити його якість завдяки актуальній і перевіреній інформації.

Інтеграція з Google Custom Search API реалізована через автоматизовані запити до пошукової системи Google із зазначеними параметрами. Система надсилає пошукові запити, сформовані на основі ключових слів та запитів користувача. У відповідь API повертає структуровані результати, включаючи посилання на сторінки, їхні описи, заголовки та інші метадані. Ці результати потім обробляються, фільтруються і зберігаються для подальшого використання.

Важливим аспектом інтеграції є обробка помилок і лімітів запитів API. Для забезпечення стабільної роботи системи передбачено ретельний моніторинг статусу запитів, контроль використання ключів API та повторне виконання запитів у разі виникнення збоїв.

4.1.2 Генерація назв і нарисів статей

Система використовує результати, отримані через Google Custom Search API, і аналізує контекстні дані, включаючи ключові слова, основні теми та релевантні фрагменти тексту. На основі цього проводиться автоматична генерація назв статей використовуючи OpenAI API.

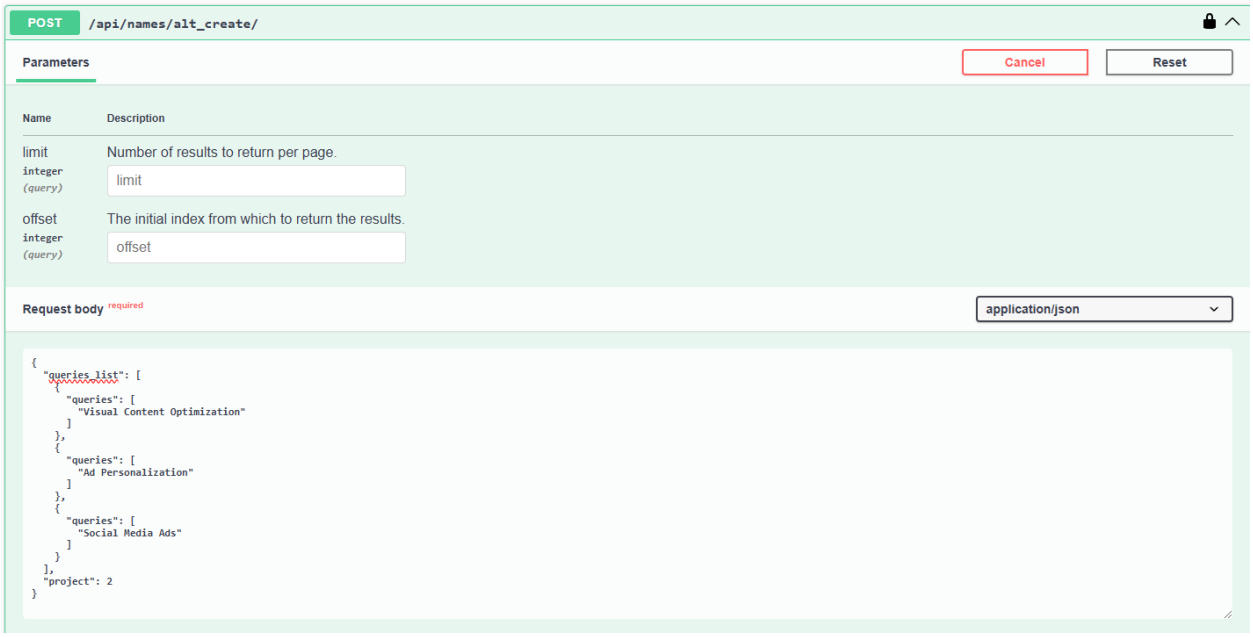


Рис. 4.1 Тіло запиту на створення назв статей

Рис. 4.2 Відповідь на запит створення назв статей

Використовуючи вже згенеровані назви та контексти користувач може згенерувати нариси статей, що відповідають стилю та формату, обраному користувачем. Алгоритми враховують вимоги до довжини, унікальності, читабельності та SEO-оптимізації заголовків.

Нариси статей формуються у вигляді логічно структурованого плану, який може містити заголовки розділів, підрозділів та ключові пункти, що будуть розкриті у статті. Алгоритми використовують рекомендації щодо структури тексту, які ґрунтуються на найкращих практиках для кожного конкретного типу контенту.

Для підвищення гнучкості процесу реалізовано можливість редагування згенерованих заголовків і нарисів. Користувач може коригувати запропоновані системою варіанти через API.

Рис. 4.3 Тіло запиту на створення нарисів статей

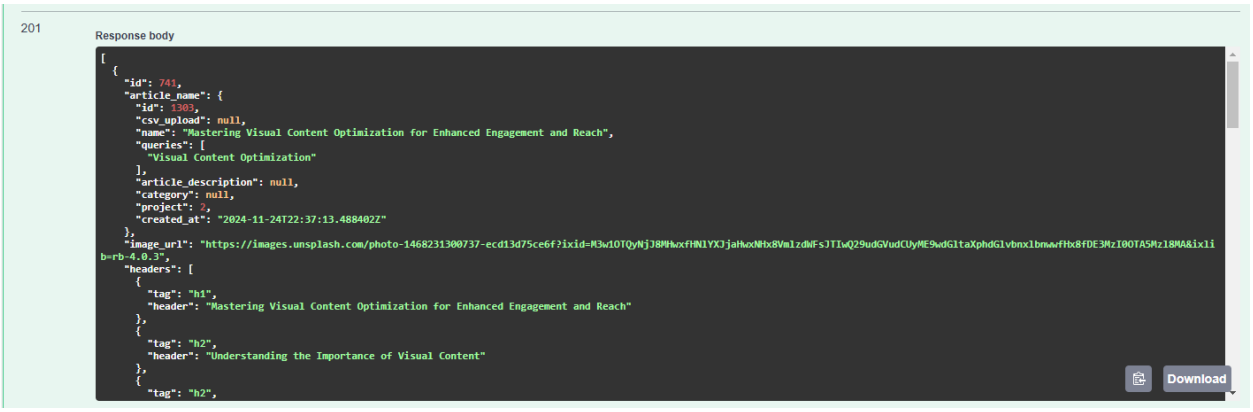


Рис. 4.4 Відповідь на запит створення нарисів статей

4.2 Опрацювання ключових слів через Google Ads API

У системі використовується Google Ads API для отримання ключових слів за пошуковими запитами. Цей інструмент забезпечує доступ до

актуальних даних про популярні запити, рівень конкуренції та пошукові тренди.

Google Ads API надає можливість виконувати пошук ключових слів на основі заданих користувачем тем або запитів. Система інтегрує цей функціонал, дозволяючи отримувати релевантні ключові слова з урахуванням географічного та мовного контексту. Наприклад, користувач може вказати регіон або мову, що дозволить алгоритму фокусуватися на аудиторії, яка відповідає бізнес-цілям.

Процес генерації ключових слів реалізований таким чином:

1. **Збір вхідних даних**: Користувач задає основні теми або базові запити, які система використовує для створення пошукових запитів.
2. **Запит до Google Ads API**: Система відправляє запити до API, витягує результати та аналізує отриману інформацію про ключові слова, такі як частота запитів та рівень конкуренції.
3. **Фільтрація та сортування**: Алгоритми обробляють дані, фільтруючи нерелевантні результати та сортують ключові слова за визначеними критеріями, такими як висока частота пошуку або низький рівень конкуренції.
4. **Результат**: Сформований список ключових слів надається користувачеві для подальшого використання у створенні контенту.

Особливістю системи є можливість налаштування параметрів генерації. Наприклад, користувач може визначити максимальну кількість ключових слів, виставити пріоритет на низьку конкуренцію або високий обсяг пошуку. Крім того, результати можуть бути збережені для подальшого використання в процесі генерації нарисів статей.

Ця функціональність забезпечує створення SEO-ефективного контенту, що відповідає запитам цільової аудиторії. Інтеграція з Google Ads API робить процес генерації ключових слів автоматизованим та інформативним, дозволяючи користувачам зосередитися на стратегічних аспектах контенту.

4.3 Планування та налаштування публікацій у CMS

4.3.1 Автоматизація розкладу публікацій

Автоматизація розкладу публікацій дозволяє знижувати ручну роботу та забезпечувати своєчасне розміщення контенту. Завдяки цьому підходу користувачі можуть створювати задачі на публікацію статей із заздалегідь визначеними параметрами, часом виконання та статусом. Центральним інструментом для реалізації цього функціоналу є бібліотека Celery у поєднанні з розширенням Celery Beat.

Розклад задач формується через спеціальні ендпойнти, які дозволяють встановлювати часові параметри публікації статей. Використовуючи згенеровані нариси статей, система створює задачі, які додаються до черги задач Celery. Celery Beat, у свою чергу, відповідає за тригери та контроль виконання цих задач відповідно до встановленого графіка.

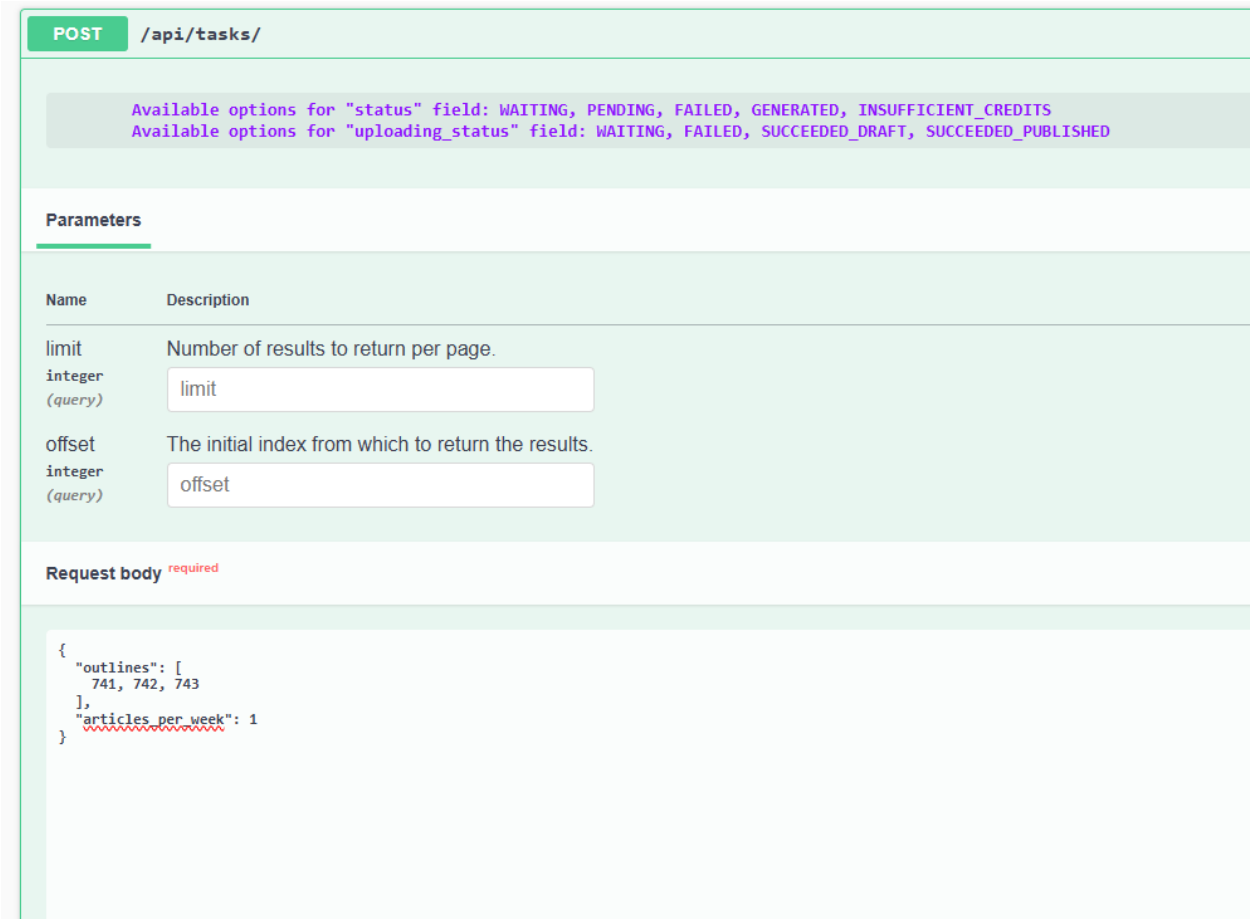


Рис. 4.5 Тіло запиту на створення запланованих задач

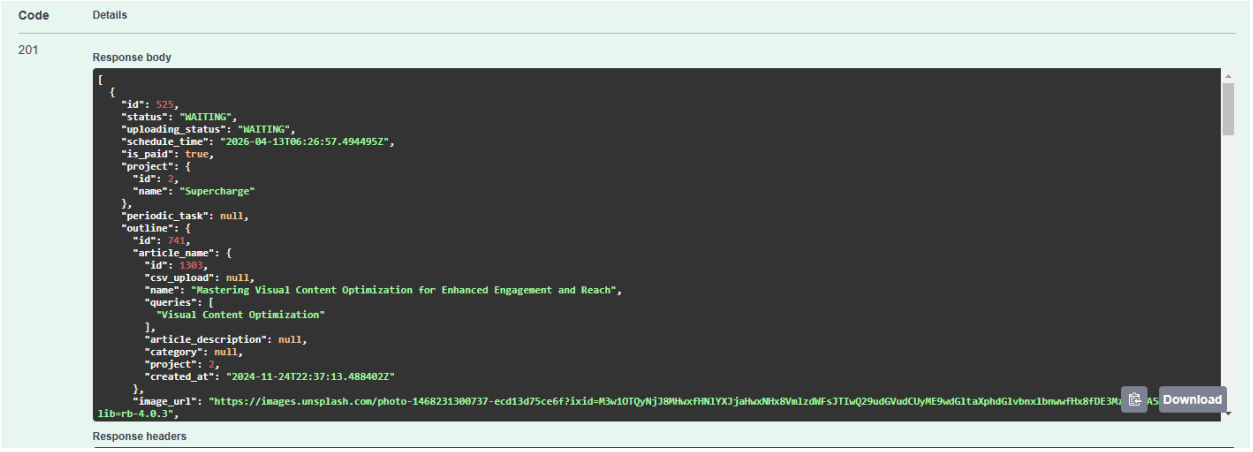


Рис. 4.6 Відповідь на запит створення запланованих задач

Крім того, система підтримує функціонал перепланування та скасування задач, що є важливим для забезпечення гнучкості роботи. У випадках, коли зміни у плануванні необхідні, користувач може переналаштувати час

виконання задачі або видалити її з черги без втрати даних. Це дозволяє уникнути помилок публікації або адаптувати графік відповідно до актуальних потреб.

4.3.1.1 Робота з Celery Beat для планування задач

Основний функціонал Celery Beat полягає у створенні, збереженні та виконанні періодичних задач. У системі публікації статей ці задачі відповідають за запуск процесу генерації та завантаження контенту у відповідну CMS у визначений користувачем час. При створенні задачі через API, система додає відповідний запис до бази даних, який містить інформацію про час виконання, параметри задачі та пов’язаний із нею контент.

Під час роботи Celery Beat постійно моніторить розклад задач, перевіряючи, чи настав час виконання будь-якої з них. Якщо задачі відповідають умовам виконання, вони додаються до черги Celery для подальшої обробки робочими процесами.

Окрім простого виконання задач, Celery Beat дозволяє реалізувати складні сценарії планування. Наприклад, система може підтримувати як одноразові задачі (публікація статті в конкретний час), так і повторювані задачі, наприклад, регулярне оновлення контенту за заданим графіком.

Celery Beat також інтегрується з Django Admin, що дозволяє адміністраторам системи вручну перевіряти та редагувати задачі безпосередньо через інтерфейс. Така можливість корисна для оперативного втручання у разі необхідності коригування розкладу.

4.3.1.2 Перепланування та скасування задач

Перепланування задачі необхідне у випадках, коли користувач бажає змінити час виконання. Це може бути викликано змінами в стратегії публікації або необхідністю синхронізації з іншими активностями у CMS. Процес

перепланування включає оновлення даних про час виконання задачі у базі даних, після чого Celery Beat автоматично враховує ці зміни. Якщо задачі вже було додано до черги на виконання, система видаляє її з черги та додає до нового розкладу з оновленим часом.

Для зручності користувача перепланування реалізовано через спеціальний ендпойнт API, який приймає ID задачі та новий час виконання. Це дозволяє інтегрувати функцію в будь-який зовнішній інтерфейс або систему.

Рис. 4.7 Тіло запиту на перепланування

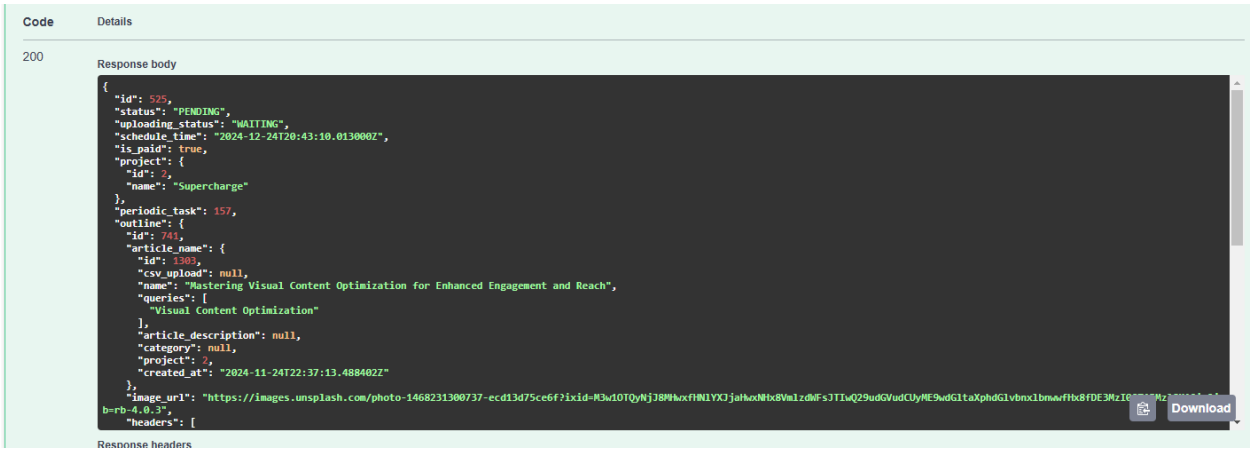


Рис. 4.8 Відповідь на запит перепланування

Скасування задач стає необхідним, якщо завдання втратило актуальність, наприклад, через зміну вимог до контенту або видалення публікації. У системі скасування задач передбачає видалення відповідного запису з бази даних Celery Beat. Якщо задача вже знаходиться в черзі на виконання, система додатково зупиняє її обробку, уникаючи непотрібних операцій.

Ця функція також доступна через ендпойнт API. Користувач або адміністратор може викликати відповідний запит із вказанням задачі, яку потрібно скасувати. Це забезпечує простий та ефективний спосіб управління задачами без прямого доступу до інфраструктури.

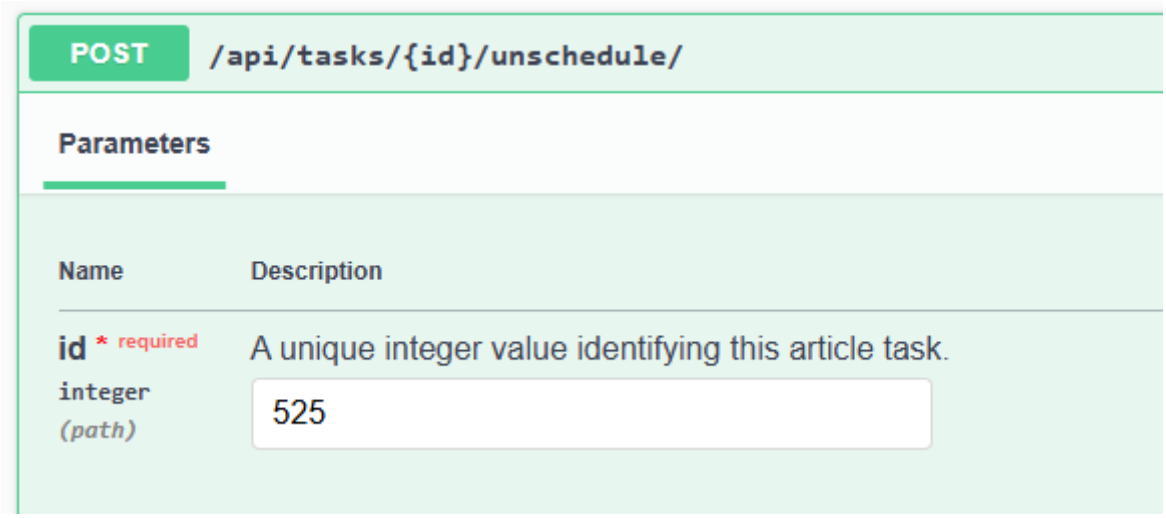


Рис. 4.9 Запит на скидання планового часу виконання задачі

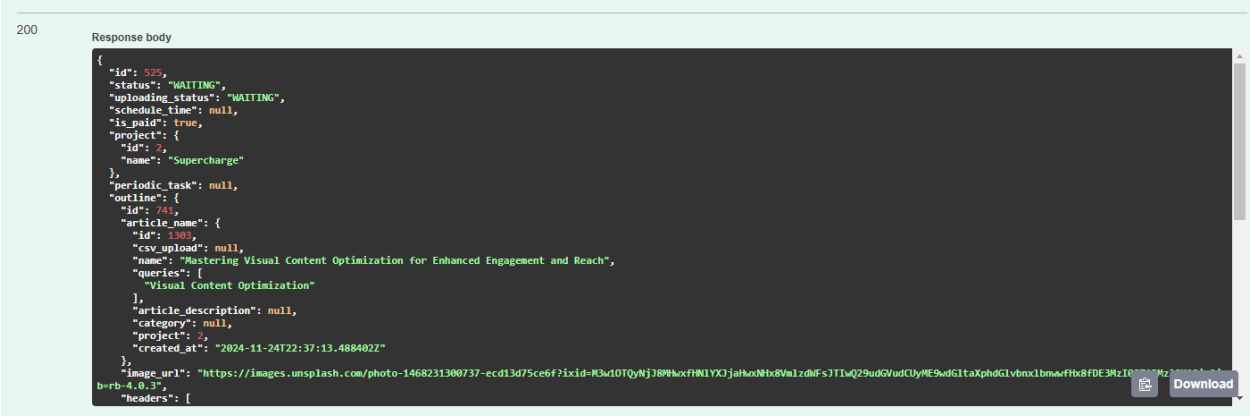


Рис. 4.10 Відповідь на запит скидання планового часу задачі

4.3.2 Контроль витрат токенів на генерацію та логіка обмеження доступу до генерації

У системі генерації контенту токени використовуються як обмежуючий ресурс, що гарантує раціональне використання доступних обчислювальних потужностей та API-лімітів.

Система токенів передбачає, що кожен успішно виконаний запит на генерацію статті витрачає один токен. Ця логіка закладена в систему, але при розширенні функціоналу для кожного типу задачі може бути налаштовано параметри споживання токенів.

Коли користувач ініціює генерацію, система перевіряє доступну кількість токенів у підписці проєкту і у разі наявності хоча б одного токена списує його з рахунку та додає задачу до черги на обробку.

Якщо токени вичерпано, користувач отримує повідомлення про необхідність поповнення ресурсу.

4.4 Генерація контенту статті

Генерація якісного контенту включає в себе створення основного тексту, його структурування та наповнення мультимедійними елементами, які відповідають тематиці та стилю статті. Такий підхід дозволяє забезпечити високу релевантність матеріалів та їх привабливість для кінцевого читача.

Для досягнення цієї мети система використовує сучасні алгоритми обробки тексту та даних. Генерація текстового контенту базується на аналізі ключових слів і контексту, що надається користувачем. Водночас інтеграція мультимедійних елементів, таких як зображення, забезпечує візуальну привабливість та краще сприйняття інформації.

Процес генерації статті поділяється на кілька етапів, кожен з яких виконується автоматизовано:

- **Генерація тексту:** створення основного змісту статті, включаючи заголовки, підзаголовки та текстові блоки. Алгоритми враховують стилістичні особливості та формат статті.
- **Мультимедійне наповнення:** додавання зображень, релевантних до змісту тексту. Цей етап значно підвищує інтерактивність матеріалів та їх привабливість.

4.4.1 Генерація текстової частини статті

На основі ключових слів та контексту, отриманого з попередніх етапів, система формує текст, що відповідає заданим параметрам. Алгоритми, інтегровані через OpenAI API, здатні створювати тексти різних форматів, зокрема аналітичні статті, новини або огляди. Враховуються також і стилістичні уподобання.

Сформований текст поділяється на логічні блоки: заголовки, підзаголовки, абзаци. Кожен блок відповідає конкретному аспекту теми та побудований таким чином, щоб забезпечити плавний перехід між частинами статті. Структура визначається нарисом статті, який формує план тексту. Дивитись додаток В.

Локалізація та адаптація

Система підтримує генерацію тексту різними мовами, враховуючи локалізаційні особливості. Це дозволяє створювати контент, адаптований до аудиторії певного регіону, з урахуванням мовних та культурних нюансів.

Результатом цього етапу є високоякісний текст, який може бути безпосередньо використаний для публікації або ж доопрацьований користувачем. Генерація тексту створює основу для подальшого наповнення статті мультимедійними елементами, що забезпечує її комплексність і завершеність.

Рефакторинг тексту

На цьому етапі система аналізує попередньо згенерований текст, усуваючи можливі недоліки, такі як повтори, некоректні або малозрозумілі фрази, а також адаптує стиль написання до заданого формату або вимог

цільової аудиторії. За допомогою алгоритмів оптимізації тексту виконується покращення читабельності та логічності матеріалу.

Цей етап також включає перевірку відповідності структури тексту попередньо створеному нарису, що дозволяє забезпечити чітку відповідність тематиці статті. Застосування рефакторингу гарантує, що кінцевий текст відповідає очікуванням користувача, має високу якість і придатний до публікації без додаткових ручних виправлень.

4.4.2 Мультимедійне наповнення статті

Мультимедійне наповнення підвищує привабливість, інтерактивність і сприйняття статті. Додавання зображень доповнює текстову частину статті, створюючи повноцінний матеріал для публікації.

Автоматичний підбір зображень

Система інтегрує API таких платформ, як Unsplash і Pexels, для автоматичного пошуку зображень відповідно до тематики статті. Використовуючи ключові слова та заголовки, генератор формує запити до цих платформ, щоб знайти візуальний контент, який найбільше відповідає темі. Механізм фільтрації гарантує, що зображення відповідають вимогам щодо якості, розміру та авторських прав.

Контроль та редагування мультимедійного контенту

Користувачі мають можливість переглядати та редагувати мультимедійні елементи перед публікацією.

Мультимедійне наповнення завершує процес створення статті, забезпечуючи її завершений вигляд для публікації. Поєднання тексту та візуального контенту підвищує якість матеріалу, робить його більш цікавим і зручним для читача.

← Maximizing Success with Customer Retention Analytics Strategies

Status: Published Cancel Save

Understanding Customer Retention Analytics

Customer retention analytics is a robust methodology that aids businesses in understanding user behaviors and metrics, aiming to identify factors that impact customer retention and churn. This analytical approach allows companies to grasp the "survival rate" of existing customers, often termed "survival analytics". By applying statistical methods, businesses can evaluate historical activity patterns and forecast future retention rates for various user segments.

Analyzing Customer Cohorts

Customer retention analytics involves tracking specific cohorts over designated time frames, facilitating a detailed analysis of how distinct customer groups interact with a company's products or services. This thorough examination of user behavior yields actionable insights that enable organizations to refine their strategies for enhanced customer satisfaction and loyalty.

Understanding Churn Causes

By scrutinizing the moments and reasons for customer disengagement, businesses can implement targeted interventions aimed at reducing churn. These interventions are informed by the insights gained from analyzing user behavior patterns, allowing for more effective solutions.

Integrating Analytics into Business Strategy

Incorporating customer retention analytics into business strategies is vital for optimizing customer lifetime value and driving growth. Recognizing which touchpoints most significantly influence retention empowers companies to refine their offerings to align with the changing demands of their customers.

Enhancing Brand Loyalty

This data-driven approach not only aids in retaining customers but also bolsters brand loyalty and enhances the overall user experience. By tailoring services to meet client needs, businesses can foster stronger and more enduring relationships with their clientele.

The Competitive Advantage

In today's competitive market, effectively employing customer retention analytics can be pivotal in maintaining loyal customers rather than losing them to competitors. These analytics are essential for refining efforts that promote long-term engagement and satisfaction among users.

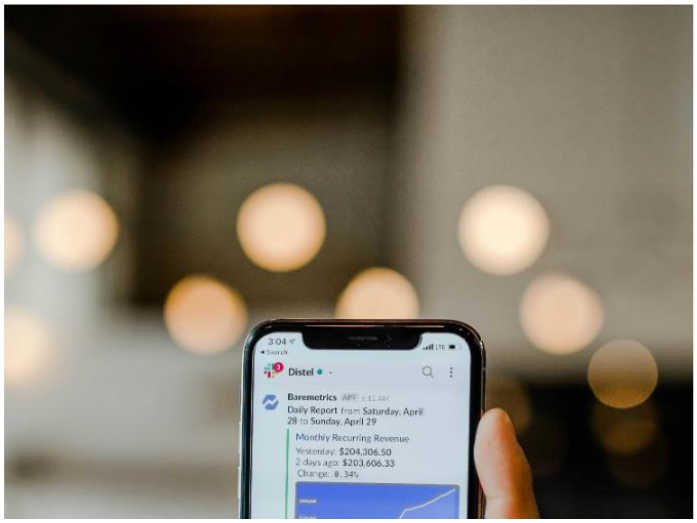


Рис. 4.11 Кінцевий результат генерації статті (частина 1)

Email Retention Strategies

Maximizing customer retention is essential for any SaaS business aiming to thrive. Implementing effective email retention strategies can significantly enhance user engagement, especially for those showing signs of inactivity. Addressing user needs early can prevent potential churn and boost overall growth.

Personalized Solutions

Email campaigns should be tailored to meet individual user needs, highlighting personalized solutions that cater to their specific challenges. This approach helps to re-engage users, forging a stronger connection with the product.

Case Study: Airfocus

Airfocus serves as an excellent example by effectively addressing user concerns through tailored email communications. This not only reaffirms the product's value but also sets a standard for creating impactful retention emails.

In-App Retention Tactics

In-app retention strategies are vital for maintaining user engagement by targeting individuals with low customer health scores. These strategies aim to encourage more comprehensive feature usage, thus increasing the product's perceived value and reducing churn.

Interactive Walkthroughs

Providing users with interactive walkthroughs or tips can guide them toward exploring underutilized features, enhancing their overall experience and satisfaction with the product.

Timely Insights

In-app messages can deliver timely insights that remind or instruct users on features they might not fully leverage. This proactive support not only enhances their experience but also boosts retention rates by ensuring the product continuously meets user expectations.

Рис. 4.12 Кінцевий результат генерації статті (частина 2)

Introduction to Key Retention Metrics

To effectively navigate the landscape of customer retention, businesses must focus on analyzing and monitoring key retention metrics. This begins with selecting the right retention KPIs, which are crucial indicators of customer engagement and churn. Metrics such as Customer Lifetime Value (CLV), Churn Rate, and Repeat Purchase Rate form the foundation for understanding the performance of retention strategies.

Understanding Retention Analysis

Retention analysis, often referred to as survival analysis, helps businesses understand why customers leave and what can be done to keep them. By examining these metrics, companies can gain insights into customer behavior and identify patterns that may indicate potential churn. This understanding is essential for creating strategies that improve customer retention and enhance new user acquisition rates.

Utilizing Retention Tools

To harness the full potential of retention data, tools like Paddle's ProfitWell Metrics offer a comprehensive platform for tracking and analyzing these numbers. By utilizing such tools, businesses can access accurate retention reports and visualizations that provide valuable insights into their customer base.

Features of Retention Tools

These tools enable businesses to generate detailed reports that highlight critical retention trends. They offer visualizations that make it easier to interpret complex data sets, allowing for informed decision-making.

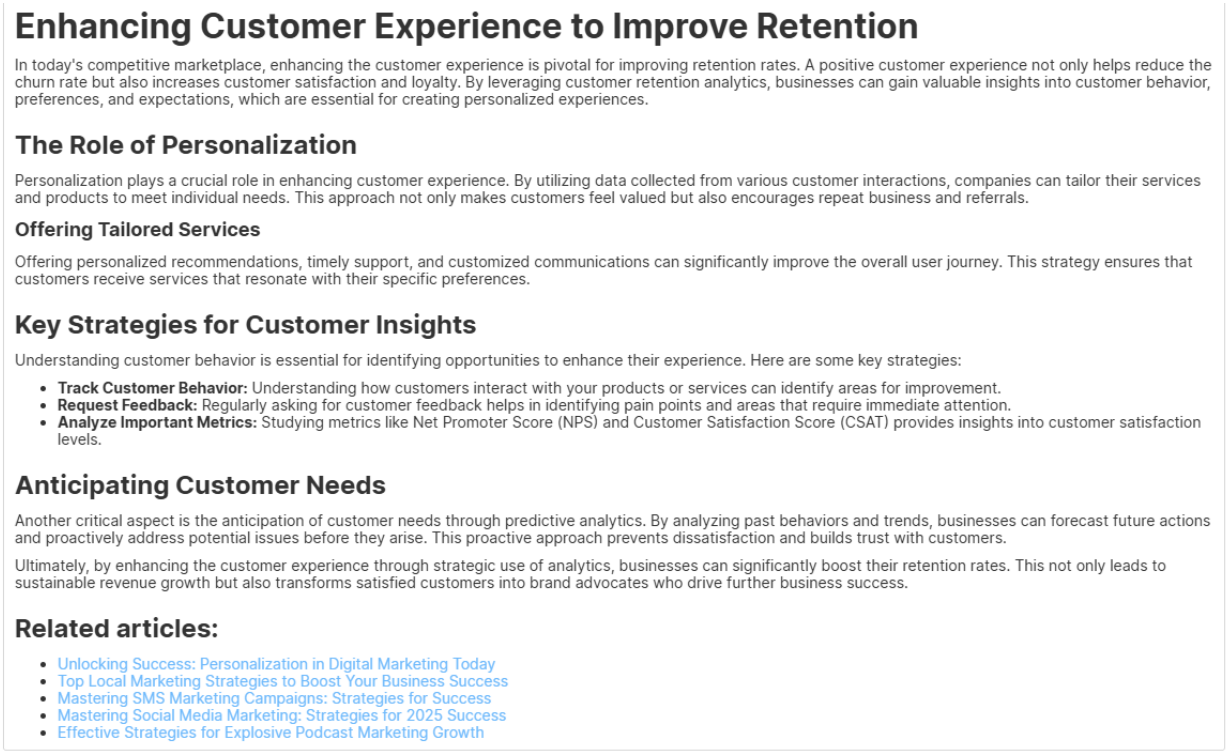
Monitoring Retention Success

Regular monitoring of key retention metrics is crucial for measuring success in maintaining a loyal customer base. By aligning KPIs with specific business goals, organizations can continually refine their retention tactics. This ensures efficient use of marketing budgets.

Benefits of Aligning KPIs

Aligning KPIs with business goals allows companies to transform data into actionable strategies that maximize customer success. This alignment facilitates strategic adjustments that enhance overall marketing effectiveness.

Рис. 4.13 Кінцевий результат генерації статті (частина 3)



Enhancing Customer Experience to Improve Retention

In today's competitive marketplace, enhancing the customer experience is pivotal for improving retention rates. A positive customer experience not only helps reduce the churn rate but also increases customer satisfaction and loyalty. By leveraging customer retention analytics, businesses can gain valuable insights into customer behavior, preferences, and expectations, which are essential for creating personalized experiences.

The Role of Personalization

Personalization plays a crucial role in enhancing customer experience. By utilizing data collected from various customer interactions, companies can tailor their services and products to meet individual needs. This approach not only makes customers feel valued but also encourages repeat business and referrals.

Offering Tailored Services

Offering personalized recommendations, timely support, and customized communications can significantly improve the overall user journey. This strategy ensures that customers receive services that resonate with their specific preferences.

Key Strategies for Customer Insights

Understanding customer behavior is essential for identifying opportunities to enhance their experience. Here are some key strategies:

- **Track Customer Behavior:** Understanding how customers interact with your products or services can identify areas for improvement.
- **Request Feedback:** Regularly asking for customer feedback helps in identifying pain points and areas that require immediate attention.
- **Analyze Important Metrics:** Studying metrics like Net Promoter Score (NPS) and Customer Satisfaction Score (CSAT) provides insights into customer satisfaction levels.

Anticipating Customer Needs

Another critical aspect is the anticipation of customer needs through predictive analytics. By analyzing past behaviors and trends, businesses can forecast future actions and proactively address potential issues before they arise. This proactive approach prevents dissatisfaction and builds trust with customers.

Ultimately, by enhancing the customer experience through strategic use of analytics, businesses can significantly boost their retention rates. This not only leads to sustainable revenue growth but also transforms satisfied customers into brand advocates who drive further business success.

Related articles:

- Unlocking Success: Personalization in Digital Marketing Today
- Top Local Marketing Strategies to Boost Your Business Success
- Mastering SMS Marketing Campaigns: Strategies for Success
- Mastering Social Media Marketing: Strategies for 2025 Success
- Effective Strategies for Explosive Podcast Marketing Growth

Рис. 4.14 Кінцевий результат генерації статті (частина 4)

4.5 Інтеграція результатів генерації з CMS

Після того як стаття створена і мультимедійний контент доданий, результати генерації повинні бути підготовлені для публікації у вибраній CMS.

4.5.1 Публікації у Webflow

Webflow є сучасною платформою для створення і керування вебсайтами, яка надає API для автоматизації багатьох процесів, зокрема публікації контенту. У контексті інтеграції з Webflow система автоматизації використовує можливості API для передачі текстового та мультимедійного контенту безпосередньо в колекції Webflow.

Процес формування публікацій починається з підготовки контенту у вигляді структурованих даних. Система забезпечує трансформацію згенерованих статей та мультимедійного контенту у формат, який відповідає

вимогам Webflow API. Це включає передачу текстів, зображень, ключових слів та іншої інформації, пов'язаної зі статтею.

Ключовим етапом є зв’язок із колекціями Webflow. Система отримує список доступних колекцій через API, що дозволяє ідентифікувати відповідну колекцію для публікації контенту. Далі відбувається формування відповідностей полів між внутрішніми моделями системи та полями колекції Webflow. Наприклад, заголовки статей передаються у поле *title*, основний текст – у *content*. Це забезпечує правильне розміщення текстового контенту, зображень та мета-даних.

Для створення запису в колекції система відправляє запит до Webflow на наступний шлях *api.webflow.com/v2/collections/collection_id/items* із підготовленим тілом запиту. Якщо публікація має бути доступною одразу, стаття позначається як "опублікована". У разі необхідності автоматичного розкладу публікацій, система може створювати чернетки з подальшим плануванням дати і часу їх публікації.

4.5.2 Публікації у WordPress

Процес публікації у WordPress розпочинається зі створення запису про статтю, який включає текстовий контент, заголовки, ключові слова, мультимедіа та мета-дані. Завдяки підтримці WordPress API, система перетворює внутрішні дані у формат, що відповідає вимогам API, включаючи поля для текстів, зображень та тегів.

Для передачі статті система використовує авторизацію через облікові дані користувача WordPress. Це забезпечує безпеку доступу до ресурсу і гарантує, що тільки авторизовані користувачі можуть додавати або змінювати записи. Залежно від налаштувань, стаття може бути одразу опублікована або створена як чернетка для подальшої модифікації.

Для мультимедіа система підтримує попереднє завантаження зображень у медіабібліотеку WordPress, після чого їх ідентифікатори використовуються у записі.

Функціонал інтеграції також включає обробку помилок, таких як відсутність доступу чи неправильна конфігурація API. У таких випадках система надсилає відповідні сповіщення користувачу для усунення проблем.

4.5.3 Публікації у Strapi

Для передачі даних у Strapi використовується API-ключ, який забезпечує безпечний доступ до платформи. Система генерує HTTP-запити із JSON-представленням статті, яке відповідає структурі бази даних Strapi. У разі відсутності потрібних полів або помилок авторизації система надсилає відповідні сповіщення для їх усунення.

Інтеграція з Strapi дозволяє зручно керувати публікаціями, розширювати структуру контенту та легко адаптувати систему до нових вимог. Завдяки цьому Strapi стає ідеальним вибором для масштабованих проєктів з кастомними налаштуваннями полів.

4.6 Логіка обробки помилок

У процесі роботи система стикається з безліччю потенційних викликів: збої під час взаємодії з API, непередбачені помилки в обробці даних, або технічні обмеження на стороні платформ, з якими вона інтегрується. Для забезпечення безперебійної роботи важливою є реалізація чіткої логіки обробки помилок, яка не лише виявляє й реєструє проблеми, але й забезпечує їхню оперативну корекцію.

4.6.1 Обробка помилок при взаємодії з API

У процесі інтеграції системи з зовнішніми API, такими як Google Custom Search API, Google Ads API, або API платформ керування контентом (Webflow, WordPress, Strapi), можуть виникати помилки взаємодії. Важливо забезпечити стабільність роботи системи за допомогою механізмів виявлення, обробки та реагування на такі помилки.

Основні типи помилок

- **Помилки авторизації:** Неправильні облікові дані (API-ключ, логін, пароль) можуть призвести до неможливості доступу до API. У таких випадках система повинна виявляти помилку та надавати зрозумілий опис проблеми для її подальшого вирішення.

- **Перевищення квот:** Більшість API мають обмеження на кількість запитів. У таких випадках система повинна зупиняти подальші запити до API та надавати інформацію про необхідність відновлення ліміту.

- **Технічні збої:** Непередбачені збої на стороні API можуть призвести до недоступності сервісу або некоректної відповіді. Система повинна бути здатною розпізнати ці збої та вживати відповідних заходів.

Реалізація обробки помилок

Виявлення: Кожен запит до API обгортається у виклик, що обробляє помилки за допомогою механізмів виключних ситуацій. Це дозволяє ідентифікувати проблеми, такі як неправильні відповіді або тайм-аути.

Реагування: У разі помилки система реалізує різні стратегії:

- **Автоматичний повтор запиту:** Якщо помилка має тимчасовий характер (наприклад, мережевий збій), запит виконується повторно з збільшенням інтервалу між спробами.

- **Зупинка операції:** Для критичних помилок (наприклад, вичерпання квот) система зупиняє виконання задачі, інформуючи користувача про причини.

- **Повідомлення:** Усі помилки записуються в лог системи, а також можуть бути передані користувачеві через інтерфейс або систему повідомлень.

Завдяки впровадженню обробки помилок при взаємодії з API система стає більш стійкою до збоїв, мінімізується ризик втрати даних або невиконання задач, а також забезпечується зручність для користувачів, які отримують чітке розуміння причин помилок і можливих шляхів їхнього вирішення.

4.6.2 Контроль статусів задач та повторні запити

Ефективне виконання задач, пов'язаних із генерацією контенту та інтеграцією з CMS, залежить від чіткого контролю їхнього статусу.

Кожна задача в системі має статус, що визначає її поточний стан. Основні статуси:

- **Очікування (Pending):** задача створена та очікує передачу до Celery на виконання.

- **Виконується (In Progress):** задача передана в Celery для обробки.

- **Успішно виконана (Completed):** задача виконана без помилок.

- **Неуспішна (Failed):** під час виконання задачі сталася помилка.

Celery забезпечує оновлення статусів задач у базі даних. Це дозволяє відстежувати прогрес виконання задач у реальному часі.

Завдяки контролю статусів система може виявляти задачі, які завершилися зі статусом **Failed**. Це може бути наслідком помилок API, мережевих збоїв або некоректних даних.

Якщо причина збою є тимчасовою (наприклад, мережевий тайм-аут), задача автоматично додається до черги повторного виконання.

Повторні спроби виконання реалізуються за допомогою механізму **збільшення інтервалу між спробами**, що дозволяє уникнути перевантаження API.

Щоб уникнути нескінченних повторів, встановлюється обмеження на кількість спроб виконання задачі. Якщо задача не виконується після декількох спроб, її статус оновлюється на **Failed Permanently**, і система інформує користувача про неможливість її завершення.

Контроль статусів задач та механізм повторних запитів забезпечують стабільність роботи системи. Це дозволяє мінімізувати вплив збоїв на виконання задач, зберігати логіку обробки даних послідовною та зручною для користувачів.

4.6.3 Логування помилок у процесі генерації та публікації

Логування забезпечує стабільність та прозорість роботи системи, зокрема під час виконання задач генерації контенту та публікації. Завдяки ретельному збору й аналізу журналів подій можна не лише швидко виявляти помилки, але й запобігати їх повторенню в майбутньому.

Логи формуються у структурованому вигляді, що включає час події, тип помилки, ідентифікатор задачі, пов’язаний API-запит і текст помилки. Це полегшує їхнє зчитування та аналіз.

Градація рівнів:

- **Інформаційні повідомлення (Info):** відображають успішне виконання дій, наприклад, початок і завершення генерації статті.
- **Попередження (Warning):** повідомляють про потенційні ризики, як-от повільний відгук API.
- **Критичні помилки (Error):** фіксують збої, наприклад, неможливість публікації у CMS.

Кожен запис логів включає контекст виконуваної задачі, як-от проєкт, стаття або CMS, з якою відбувається взаємодія.

Логування реалізовано за допомогою бібліотеки Python *logging*, яка дозволяє налаштовувати формат і збереження журналів.

Для централізованого збору логів у продакшн-середовищі використовується Sentry, який забезпечує зручний інтерфейс для перегляду й аналізу.

Логи зберігаються у двох місцях: локальні файли на сервері для швидкого доступу та на Sentry, для довгострокового зберігання та аналітики.

У разі виникнення помилок система фіксує деталі виклику API або задачі, яка спричинила збій додає у лог текст виключення, отриманий від API або внутрішніх обробників.

Адміністратори можуть використовувати логи для швидкого виявлення причин помилок та їх усунення. Завдяки журналам можна ідентифікувати слабкі місця в системі.

Висновки

У магістерській роботі було розроблено, описано та реалізовано систему для автоматизації створення, обробки та публікації SEO-контенту з інтеграцією в популярні системи керування контентом, такі як Webflow, WordPress і Strapi. Основна увага приділялася побудові гнучкої та масштабованої архітектури, яка забезпечує зручність взаємодії між модулями, а також легко адаптується до нових вимог.

Система включає всі ключові етапи роботи з контентом: від збору контексту й генерації заголовків до створення текстової частини статті, мультимедійного наповнення та інтеграції з CMS. Завдяки використанню сучасних інструментів, таких як OpenAI API, Google Ads API та Google Custom Search API, вдалося досягти високої якості створеного контенту та його відповідності SEO-вимогам. Механізми планування задач, реалізовані за допомогою Celery і Celery Beat, забезпечують автоматизацію виконання складних процесів і знижують навантаження на користувачів системи.

Розробка логіки обробки помилок і ведення логів підвищила надійність системи, дозволяючи фіксувати збої в роботі, виконувати повторні запити до API й моніторити статус виконання задач. Це сприяє створенню прозорої системи, яка ефективно працює навіть у випадках виникнення проблемних ситуацій.

Запропонована система є практичним інструментом для автоматизації створення й публікації контенту, що значно економить час і ресурси, необхідні для організації SEO-кампаній. Її універсальність забезпечує можливість використання як у невеликих проєктах, так і у великих компаніях із масштабними обсягами даних. Розроблена архітектура дає змогу легко інтегрувати нові CMS і вдосконалювати існуючий функціонал.

У майбутньому система може бути розширена за рахунок інтеграції з додатковими CMS, удосконалення процесу генерації контенту та розробки інтуїтивно зрозумілого інтерфейсу для користувачів. Це дозволить задовольнити потреби ширшої аудиторії та забезпечити її актуальність у сфері цифрового маркетингу, яка динамічно розвивається. Результати роботи підтверджують, що створена система має великий потенціал для вдосконалення та подальшого використання.

Список використаної літератури

1. Python – **[Електронний ресурс].** Режим доступу : https://docs.python.org/3.12/
2. Django документація – **[Електронний ресурс].** Режим доступу: https://docs.djangoproject.com/en/5.1/
3. Асинхронна черга задач Celery – **[Електронний ресурс].** Режим доступу: https://docs.celeryq.dev/en/stable/
4. Документація Redis – **[Електронний ресурс].** Режим доступу : https://redis.io/docs/latest/
5. Документація Docker – **[Електронний ресурс].** Режим доступу : https://docs.docker.com/
6. REST API Design Guide – **[Електронний ресурс].** Режим доступу : https://www.freecodecamp.org/news/rest-api-design-best-practices-build-a-rest-api/
7. Google Custom Search API – **[Електронний ресурс].** Режим доступу : https://developers.google.com/custom-search/v1/overview
8. Документація Google Ads API – **[Електронний ресурс].** Режим доступу: https://developers.google.com/google-ads/api/docs/keyword-planning/generate-keyword-ideas
9. Документація Webflow CMS API – **[Електронний ресурс].** Режим доступу : https://developers.webflow.com/data/reference/rest-introduction
10. Документація WordPress REST API – **[Електронний ресурс].** Режим доступу : https://developer.wordpress.org/rest-api/
11. Документація Strapi Developer – **[Електронний ресурс].** Режим доступу: https://docs.strapi.io/dev-docs/api/rest#endpoints
12. Бібліотека concurrent – **[Електронний ресурс].** Режим доступу : https://docs.python.org/3/library/concurrent.futures.html
13. Документація OpenAI API – **[Електронний ресурс].** Режим доступу : https://platform.openai.com/docs/api-reference/introduction

Додатки

Додаток А

```
# src/name/models.py

from django.contrib.postgres.fields import ArrayField
from django.core.exceptions import ValidationError
from django.db import models

from base.choices import ArticleCategoryChoices, ArticleType


class CSVUpload(models.Model):
    file = models.FileField(upload_to="csvs/")
    uploaded_at = models.DateTimeField(auto_now_add=True)
    project = models.ForeignKey("project.Project", on_delete=models.CASCADE, null=True, blank=True)


class ArticleName(models.Model):
    csv_upload = models.ForeignKey(CSVUpload, on_delete=models.CASCADE, null=True, blank=True)
    project = models.ForeignKey("project.Project", on_delete=models.CASCADE, null=True, blank=True)
    name = models.CharField(max_length=512)
    alt_name = models.CharField(max_length=512, null=True, blank=True)
    article_description = models.TextField(null=True, blank=True)
    category = models.CharField(max_length=64, choices=ArticleCategoryChoices.choices, null=True, blank=True)
    context = models.JSONField(null=True, blank=True)  # TODO: change to OneToOneField with ArticleNameContext model
    queries = ArrayField(models.CharField(max_length=255), blank=True, default=list)
    created_at = models.DateTimeField(auto_now_add=True)

    def save(self, *args, **kwargs):
        self.full_clean()
        super().save(*args, **kwargs)

    def clean(self):
        if self.project.article_type == ArticleType.NEWS_ARTICLE and not self.article_description:
            raise ValidationError("Article description cannot be empty for news articles.")


# src/project/models.py

from django.db import models
from rest_framework.exceptions import ValidationError

from article.models import ArticleField
from base.choices import API, ArticleFormat, ArticleType, Language
from project.services.uploads import ProjectService
from user.models import Subscription, User


class Project(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    subscription = models.OneToOneField(Subscription, on_delete=models.CASCADE, null=True, blank=True)
    name = models.CharField(max_length=256)
    api = models.CharField(max_length=50, choices=API.choices)
    api_base_url = models.CharField(max_length=256, blank=True, null=True)
    article_format = models.CharField(max_length=50, choices=ArticleFormat.choices)
    article_type = models.CharField(max_length=64, choices=ArticleType.choices, default=ArticleType.DEFAULT)
    target_article_word_count = models.IntegerField(default=1000)
    publish_on_upload = models.BooleanField(default=True)
    api_key = models.CharField(max_length=256, blank=True, null=True)
    username = models.CharField(max_length=256, blank=True, null=True)
    password = models.CharField(max_length=256, blank=True, null=True)
```

```
    api_collection_path = models.CharField(max_length=512)
    display_article_url = models.CharField(max_length=256, blank=True, null=True)
    language = models.CharField(max_length=2, choices=Language.choices, default=Language.ENGLISH)
    metadata = models.JSONField(default=dict, blank=True)
    tasks = models.ManyToManyField("django_celery_beat.PeriodicTask", related_name="projects", blank=True)

    def __str__(self):
        return self.name

    def save(self, *args, **kwargs):
        project_service_class = self.get_project_service_class()

        project_service = project_service_class()
        credentials = {"collection": self.api_collection_path}
        if self.api == API.WORDPRESS:
            credentials["username"] = self.username
            credentials["password"] = self.password
            credentials["base_url"] = self.api_base_url
        elif self.api == API.WEBFLOW:
            credentials["api_key"] = self.api_key
        elif self.api == API.STRAPI:
            credentials["api_key"] = self.api_key
            credentials["base_url"] = self.api_base_url
        else:
            raise ValueError(f"Unknown API: {self.api}")

        if not any("default" in value for value in credentials.values()):
            rights = project_service.has_permissions_for_collection(**credentials)
            if rights["exception"]:
                raise ValidationError(str(rights["exception"].detail))

        super().save(*args, **kwargs)

    def get_project_service_class(self) -> type[ProjectService]:
        service_map = API.get_project_service_map()
        return service_map[self.api]

    def get_destination_choices(self) -> list[dict[str, str]]:
        project_service_class = self.get_project_service_class()
        project_service = project_service_class()
        return project_service.get_destination_choices(self)


class FieldMapping(models.Model):
    project = models.ForeignKey(Project, on_delete=models.CASCADE, related_name="field_mappings")
    origin = models.CharField(max_length=50, choices=ArticleField.get_choices(), null=True, blank=True)
    destination = models.CharField(max_length=256)
    default_value = models.CharField(max_length=256, null=True, blank=True)

    class Meta:
        unique_together = ("project", "destination")

    def __str__(self):
        return f"{self.origin} -> {self.destination} for {self.project.name}"


# src/outline/models.py

from django.contrib.postgres.fields import ArrayField
from django.db import models

from name.models import ArticleName


class Outline(models.Model):
    article_name = models.ForeignKey(ArticleName, on_delete=models.CASCADE)
```

```
    image_url = models.URLField(null=True, blank=True, max_length=1024)
    created_at = models.DateTimeField(auto_now_add=True)


class OutlineEmbedding(models.Model):
    outline = models.OneToOneField(Outline, related_name="embedding", on_delete=models.CASCADE)
    values = ArrayField(models.FloatField())


class Header(models.Model):
    outline = models.ForeignKey(Outline, related_name="headers", on_delete=models.CASCADE)
    tag = models.CharField(max_length=255)
    header = models.CharField(max_length=255)


class Keyword(models.Model):
    outline = models.ForeignKey(Outline, related_name="keywords", on_delete=models.CASCADE)
    text = models.CharField(max_length=255)
    competition = models.IntegerField(default=0)
    avg_monthly_searches = models.IntegerField(default=0)


# src/article/models.py

from django.db import models
from django_celery_beat.models import PeriodicTask

from base.choices import Status, UploadingStatus
from outline.models import Outline


class ArticleField:
    @classmethod
    def get_choices(cls):
        fields = Article._meta.fields
        article_fields = [
            (field.name, field.verbose_name.title()) for field in fields if field.name not in ["id", "article_task"]
        ]
        return article_fields

    @classmethod
    def get_choice_names(cls):
        return [choice[0] for choice in cls.get_choices()]


class ArticleTask(models.Model):
    outline = models.ForeignKey(Outline, on_delete=models.CASCADE)
    schedule_time = models.DateTimeField(null=True, blank=True)
    status = models.CharField(max_length=64, choices=Status.choices, default=Status.WAITING)
    uploading_status = models.CharField(max_length=64, choices=UploadingStatus.choices, default=UploadingStatus.WAITING)
    created_at = models.DateTimeField(auto_now_add=True)
    periodic_task = models.OneToOneField(PeriodicTask, on_delete=models.SET_NULL, null=True, blank=True)

    def __str__(self):
        return f"task {self.pk}: {self.outline.article_name.name}"


class Article(models.Model):
    article_task = models.OneToOneField(ArticleTask, on_delete=models.CASCADE, related_name="article")
    title = models.CharField(max_length=256)
    body = models.TextField()
    related_urls = models.TextField(default="", null=True)
    shortName = models.CharField(max_length=256)
    oneLiner = models.CharField(max_length=512)
    metaTitle = models.CharField(max_length=256, null=True)
    metaDescription = models.CharField(max_length=512, null=True)
```

```
    category = models.CharField(max_length=64, null=True)
    category_id = models.CharField(max_length=64, null=True, help_text="Category ID in the target system.")
    image_url = models.URLField(null=True, blank=True, max_length=1024)
    targetKeywords = models.CharField(max_length=512, null=True)
    publishedAt = models.DateTimeField(auto_now_add=True)

    def get_body(self) -> str:
        return f"{self.body}\n\n{self.related_urls}"


# src/prompt/models.py

from django.core.validators import MaxValueValidator, MinValueValidator
from django.db import models


class Prompt(models.Model):
    name = models.CharField(max_length=255, null=False)
    user_prompt = models.TextField(null=False)
    system_prompt = models.TextField(null=False)
    temperature = models.FloatField(validators=[MinValueValidator(0.0), MaxValueValidator(1.0)], default=0.8)
    is_active = models.BooleanField(default=False)

    def __str__(self):
        return f"{self.name} ({'Active' if self.is_active else 'Inactive'})"

    def save(self, *args, **kwargs):
        if self.is_active:
            Prompt.objects.filter(name=self.name, is_active=True).update(is_active=False)

        super().save(*args, **kwargs)

    @classmethod
    def get_active_prompt(cls, name: str):
        return cls.objects.filter(name=name, is_active=True).first()


# src/user/models.py - these models are not managed on django backend

# This is an auto-generated Django model module.
# You'll have to do the following manually to clean this up:
#   * Rearrange models' order
#   * Make sure each model has one field with primary_key=True
#   * Make sure each ForeignKey and OneToOneField has `on_delete` set to the desired behavior
#   * Remove `managed = False` lines if you wish to allow Django to create, modify, and delete the table
# Feel free to rename the models, but don't rename db_table values or field names.
from django.db import models


class Account(models.Model):
    id = models.TextField(primary_key=True)
    userid = models.ForeignKey("User", models.DO_NOTHING, db_column="userId")  # Field name made lowercase.
    type = models.TextField()
    provider = models.TextField()
    provideraccountid = models.TextField(db_column="providerAccountId")  # Field name made lowercase.
    refresh_token = models.TextField(blank=True, null=True)
    access_token = models.TextField(blank=True, null=True)
    expires_at = models.IntegerField(blank=True, null=True)
    token_type = models.TextField(blank=True, null=True)
    scope = models.TextField(blank=True, null=True)
    id_token = models.TextField(blank=True, null=True)
    session_state = models.TextField(blank=True, null=True)
    refresh_token_expires_in = models.IntegerField(blank=True, null=True)

    class Meta:
        managed = False
```

```
        db_table = "Account"
        unique_together = (("provider", "provideraccountid"),)


class Session(models.Model):
    id = models.TextField(primary_key=True)
    sessiontoken = models.TextField(db_column="sessionToken", unique=True)  # Field name made lowercase.
    userid = models.ForeignKey("User", models.DO_NOTHING, db_column="userId")  # Field name made lowercase.
    expires = models.DateTimeField()

    class Meta:
        managed = False
        db_table = "Session"


class Subscription(models.Model):
    id = models.TextField(primary_key=True)
    stripesubscriptionid = models.TextField(db_column="stripeSubscriptionId", unique=True)  # Field name made lowercase.
    plan = models.TextField()
    tokens = models.IntegerField()
    startdate = models.DateTimeField(db_column="startDate")  # Field name made lowercase.
    enddate = models.DateTimeField(db_column="endDate")  # Field name made lowercase.

    class Meta:
        managed = False
        db_table = "Subscription"


class User(models.Model):
    id = models.TextField(primary_key=True)
    name = models.TextField(blank=True, null=True)
    firstname = models.TextField(db_column="firstName", blank=True, null=True)  # Field name made lowercase.
    lastname = models.TextField(db_column="lastName", blank=True, null=True)  # Field name made lowercase.
    email = models.TextField(unique=True, blank=True, null=True)
    emailverified = models.DateTimeField(db_column="emailVerified", blank=True, null=True)  # Field name made lowercase.
    password = models.TextField(blank=True, null=True)
    image = models.TextField(blank=True, null=True)
    is_active = models.BooleanField()
    is_staff = models.BooleanField()
    is_superuser = models.BooleanField()
    date_joined = models.DateTimeField()

    class Meta:
        managed = False
        db_table = "User"


class Verificationtoken(models.Model):
    identifier = models.TextField()
    token = models.TextField(unique=True)
    expires = models.DateTimeField()

    class Meta:
        managed = False
        db_table = "VerificationToken"
        unique_together = (("identifier", "token"),)
```

Додаток Б

```
import json
from collections import defaultdict
from datetime import timedelta

from django.db.models import BooleanField, Case, F, QuerySet, Value, When
from django.shortcuts import get_object_or_404
```

```
from django.utils import timezone as tz
from django_celery_beat.models import ClockedSchedule, PeriodicTask
from drf_spectacular.utils import extend_schema
from rest_framework import exceptions, status, viewsets
from rest_framework.decorators import action
from rest_framework.request import Request
from rest_framework.response import Response

from article.celery_tasks.default_article import generate_and_upload_article
from article.filters import ArticleTaskFilterSet
from article.models import Article, ArticleTask
from article.serializers import (
    ArticleTaskInputSerializer,
    ArticleTaskSerializer,
    BulkApproveArticleTasksSerializer,
    BulkArticleTaskUpdateSerializer,
    ScheduleArticleTaskSerializer,
)
from article.serializers.article import ArticleSerializer
from base.choices import Status, UploadingStatus
from user.models import Subscription


class ArticleTaskViewSet(viewsets.ModelViewSet):
    serializer_class = ArticleTaskSerializer
    filterset_class = ArticleTaskFilterSet
    ordering_fields = "__all__"

    def get_queryset(self):
        queryset = ArticleTask.objects.all()
        return self.annotate_queryset(queryset)

    def annotate_queryset(self, queryset: QuerySet[ArticleTask]):
        queryset = (
            queryset.select_related("outline__article_name__project__subscription")
            .annotate(
                project_id=F("outline__article_name__project__id"),
                project_name=F("outline__article_name__project__name"),
                subscription_tokens=F("outline__article_name__project__subscription__tokens"),
            )
            .order_by("project_id", "schedule_time")
        )

        tasks = list(queryset)
        project_tokens = Subscription.objects.values_list("project__id", "tokens")
        project_tokens_dict = {project_id: tokens for project_id, tokens in project_tokens}

        project_cumulative_tokens = defaultdict(int)

        annotations = []
        for task in tasks:
            project_id = task.project_id
            if task.status in (Status.GENERATED, Status.FAILED) or task.uploading_status in (
                UploadingStatus.SUCCEEDED_DRAFT,
                UploadingStatus.SUCCEEDED_PUBLISHED,
            ):
                is_paid = None
            else:
                if project_cumulative_tokens[project_id] < project_tokens_dict.get(project_id, 0):
                    is_paid = True
                    project_cumulative_tokens[project_id] += 1
                else:
                    is_paid = False

            annotations.append({"id": task.id, "is_paid": is_paid})
```

```
    case_whens = [When(pk=annotation["id"], then=Value(annotation["is_paid"])) for annotation in annotations]

    queryset = queryset.annotate(
        is_paid=Case(
            *case_whens,
            output_field=BooleanField(),
        )
    )

    return queryset.order_by("id")

@extend_schema(
    request=ArticleTaskInputSerializer,
    responses={status.HTTP_201_CREATED: ArticleTaskSerializer(many=True)},
    description=(
        f"""
        Available options for \"status\" field: {', '.join(Status.names)}
        Available options for \"uploading_status\" field: {', '.join(UploadingStatus.names)}
    """
    ),
)
def create(self, request: Request, *args, **kwargs) -> Response:
    input_serializer = ArticleTaskInputSerializer(data=request.data)
    input_serializer.is_valid(raise_exception=True)
    validated_data = input_serializer.validated_data

    outlines = validated_data["outlines"]

    articles_per_week = validated_data["articles_per_week"]

    interval = timedelta(days=7) / articles_per_week
    created_task_ids = []

    for outline in outlines:
        project = outline.article_name.project

        latest_task_time = tz.now()
        tasks_of_service = ArticleTask.objects.filter(
            outline__article_name__project=project, schedule_time__gt=tz.now()
        )
        if tasks_of_service.exists():
            latest_task = tasks_of_service.latest("schedule_time")
            latest_task_time = latest_task.schedule_time

        eta = latest_task_time + interval

        article_task = ArticleTask.objects.create(outline=outline, schedule_time=eta)

        created_task_ids.append(article_task.id)

    serializer = ArticleTaskSerializer(self.get_queryset().filter(id__in=created_task_ids), many=True)
    return Response(serializer.data, status=status.HTTP_201_CREATED)

def destroy(self, request, *args, **kwargs):
    instance: ArticleTask = self.get_object()
    periodic_task = instance.periodic_task
    if periodic_task:
        periodic_task.delete()
    instance.periodic_task = None
    instance.save()
    self.perform_destroy(instance)
    return Response(status=status.HTTP_204_NO_CONTENT)

@extend_schema(
    request=BulkArticleTaskUpdateSerializer(many=True),
    responses={status.HTTP_200_OK: ArticleTaskSerializer(many=True)},
```

```
    description=(
        f"""
        Available options for \"status\" field: {', '.join(Status.names)}
        Available options for \"uploading_status\" field: {', '.join(UploadingStatus.names)}
    """
    ),
)
@action(detail=False, methods=["patch"], url_path="bulk_update")
def bulk_update(self, request: Request, *args, **kwargs):
    serializer = BulkArticleTaskUpdateSerializer(data=request.data, many=True)
    serializer.is_valid(raise_exception=True)

    updated_task_ids = []
    for data in serializer.validated_data:
        task = get_object_or_404(ArticleTask, id=data["id"])

        if "schedule_time" in data:
            schedule_time = data["schedule_time"]
            task.schedule_time = schedule_time

            clocked_schedule, _ = ClockedSchedule.objects.get_or_create(clocked_time=schedule_time)
            if not task.periodic_task:
                task.periodic_task = PeriodicTask.objects.create(
                    clocked=clocked_schedule,
                    name=str(task),
                    task="article.celery_tasks.default_article.generate_and_upload_article",
                    args=json.dumps([task.id]),
                    one_off=True,
                )
            else:
                task.periodic_task.clocked = clocked_schedule
            task.periodic_task.save()
            task.status = Status.PENDING
            task.save()

        if "status" in data:
            task.status = data["status"]

        if "uploading_status" in data:
            task.uploading_status = data["uploading_status"]

        task.save()
        updated_task_ids.append(task.id)

    task_serializer = ArticleTaskSerializer(self.get_queryset().filter(id__in=updated_task_ids), many=True)
    return Response(task_serializer.data, status=status.HTTP_200_OK)

@extend_schema(
    request=None,
    responses={status.HTTP_200_OK: ArticleTaskSerializer(many=True)},
)
@action(detail=True, methods=["post"], url_path="approve")
def approve(self, request: Request, pk=None):
    task: ArticleTask = self.get_object()

    if not task.schedule_time:
        raise exceptions.ValidationError(
            f"Tasks schedule time must be not null to approve\nTask: {task}", status.HTTP_400_BAD_REQUEST
        )

    if not task.periodic_task:
        clocked_schedule, _ = ClockedSchedule.objects.get_or_create(clocked_time=task.schedule_time)
        task.periodic_task = PeriodicTask.objects.create(
            clocked=clocked_schedule,
            name=str(task),
            task="article.celery_tasks.default_article.generate_and_upload_article",
```

```
            args=json.dumps([task.id]),
            one_off=True,
        )

    task.status = Status.PENDING
    task.save()

    task_serializer = ArticleTaskSerializer(task)
    return Response(task_serializer.data, status=status.HTTP_200_OK)

@extend_schema(
    request=BulkApproveArticleTasksSerializer,
    responses={status.HTTP_200_OK: ArticleTaskSerializer(many=True)},
)
@action(detail=False, methods=["post"], url_path="bulk_approve")
def bulk_approve(self, request: Request):
    serializer = BulkApproveArticleTasksSerializer(data=request.data)
    serializer.is_valid(raise_exception=True)

    tasks = serializer.validated_data.get("article_task_ids")
    for task in tasks:
        if not any(task.schedule_time for task in tasks):
            raise exceptions.ValidationError(
                f"Tasks schedule time must be not null to approve\nTask: {task}", status.HTTP_400_BAD_REQUEST
            )

    approved_task_ids = []
    for task in tasks:
        if not task.periodic_task:
            clocked_schedule, _ = ClockedSchedule.objects.get_or_create(clocked_time=task.schedule_time)
            task.periodic_task = PeriodicTask.objects.create(
                clocked=clocked_schedule,
                name=str(task),
                task="article.celery_tasks.default_article.generate_and_upload_article",
                args=json.dumps([task.id]),
                one_off=True,
            )
        task.status = Status.PENDING
        task.save()

        approved_task_ids.append(task.id)

    task_serializer = ArticleTaskSerializer(self.get_queryset().filter(id__in=approved_task_ids), many=True)
    return Response(task_serializer.data, status=status.HTTP_200_OK)

@extend_schema(request=None, responses={status.HTTP_200_OK: ArticleTaskSerializer})
@action(detail=True, methods=["post"], url_path="unschedule")
def unschedule(self, request: Request, pk=None):
    task: ArticleTask = self.get_object()
    if task.periodic_task:
        task.periodic_task.delete()
        task.periodic_task = None

    task.schedule_time = None
    task.status = Status.WAITING
    task.save()

    serializer = self.get_serializer(task)
    return Response(serializer.data, status=status.HTTP_200_OK)

@extend_schema(
    request=ScheduleArticleTaskSerializer,
    responses={status.HTTP_200_OK: ArticleTaskSerializer},
    description="Schedule a periodic task to a given time. The 'new_schedule_time' field is required.",
)
@action(detail=True, methods=["post"], url_path="reschedule")
```

```
def reschedule(self, request: Request, pk=None):
    task: ArticleTask = self.get_object()
    serializer = ScheduleArticleTaskSerializer(data=request.data)
    serializer.is_valid(raise_exception=True)
    new_schedule_time = serializer.validated_data.get("new_schedule_time")
    new_clocked_schedule, _ = ClockedSchedule.objects.get_or_create(clocked_time=new_schedule_time)
    if not task.periodic_task:
        task.periodic_task = PeriodicTask.objects.create(
            clocked=new_clocked_schedule,
            name=str(task),
            task="article.celery_tasks.default_article.generate_and_upload_article",
            args=json.dumps([task.id]),
            one_off=True,
        )
    else:
        task.periodic_task.clocked = new_clocked_schedule
    task.periodic_task.save()
    task.status = Status.PENDING
    task.schedule_time = new_schedule_time
    task.save()

    serializer = self.get_serializer(task)
    return Response(serializer.data, status=status.HTTP_200_OK)

@extend_schema(
    request=None,
    responses={status.HTTP_200_OK: ArticleSerializer},
    description="Generate an article immediately for the specified task.",
)
@action(detail=True, methods=["post"], url_path="generate_now")
def generate_now(self, request: Request, pk=None):
    task: ArticleTask = self.get_object()

    if task.status not in [Status.PENDING, Status.WAITING]:
        return Response(
            {"error": "Only tasks with PENDING or WAITING status can be generated."},
            status=status.HTTP_400_BAD_REQUEST,
        )

    article_id = generate_and_upload_article(task.id)
    if not article_id:
        return Response(
            {"error": "Article generation failed"},
            status=status.HTTP_400_BAD_REQUEST,
        )
    article_serializer = ArticleSerializer(Article.objects.get(id=article_id))
    return Response(article_serializer.data, status=status.HTTP_200_OK)

@extend_schema(
    request=None,
    responses={status.HTTP_200_OK: ArticleSerializer},
    description="Generate a whole article immediately for the specified task.",
)
@action(detail=True, methods=["post"], url_path="generate_whole")
def generate_whole_article_now(self, request: Request, pk=None):
    task: ArticleTask = self.get_object()

    if task.status not in [Status.PENDING, Status.WAITING]:
        return Response(
            {"error": "Only tasks with PENDING or WAITING status can be generated."},
            status=status.HTTP_400_BAD_REQUEST,
        )

    article_id = generate_and_upload_article(task.id, True)

    article_serializer = ArticleSerializer(Article.objects.get(id=article_id))
```

```
        return Response(article_serializer.data, status=status.HTTP_200_OK)
```

Додаток В

```
import itertools
from concurrent.futures import ThreadPoolExecutor

from django.utils.text import slugify

from base.choices import ImageService
from base.mappings import IMAGE_SERVICE_CLASS_MAP
from base.services.image.image_scraper import ImageScraper
from base.services.openai_.gpt_completion import get_gpt_completion
from generator.services.article import ArticleGeneratorService
from generator.structs import ArticleNameStruct, ArticleStruct, HeaderStruct, KeywordStruct, OutlineStruct
from prompt.models import Prompt


class RichTextArticleGeneratorService(ArticleGeneratorService):
    def generate_article(self, article_name: ArticleNameStruct, outline: OutlineStruct) -> ArticleStruct:
        body = self._generate_rich_text_article(article_name, outline)
        joined_keywords = ", ".join([kw.text for kw in outline.keywords[: len(outline.headers)]])
        shortName = slugify(article_name.article_name)
        article_details = self.generate_article_details(body)
        article = ArticleStruct(
            title=article_name.article_name,
            body=body,
            shortName=shortName,
            oneLiner=article_details.oneLiner,
            metaTitle=article_details.metaTitle,
            metaDescription=article_details.metaDescription,
            targetKeywords=joined_keywords,
        )
        return article

    def _generate_rich_text_article(self, article_name: ArticleNameStruct, outline: OutlineStruct):
        article_chunks = []
        relevant_chunks = self._get_relevant_chunks(article_name.context)[: self._enrich_chunk_limit]
        self._enrich_chunks(relevant_chunks)
        most_similar_chunks_list = [
            self._get_most_similar_chunks(target_text=header.header, enriched_chunks=relevant_chunks)
            for header in outline.headers
        ]
        with ThreadPoolExecutor(max_workers=20) as executor:
            article_chunks = executor.map(
                self._generate_chunk,
                outline.headers,
                itertools.repeat(article_name),
                most_similar_chunks_list,
                itertools.chain(outline.keywords, itertools.repeat(None)),
            )
        article_chunks = [self._clean_html(chunk) for chunk in article_chunks]

        refactored_article_chunks = self.refactor_article_chunks(article_chunks, "RICH_TEXT", self.target_word_count)
        image_tag_template = '<img src="{image_url}" alt="{alt_text}" />'
        image_service: ImageScraper = IMAGE_SERVICE_CLASS_MAP[ImageService.UNSPLASH]()
        image_suggestions = image_service.fetch_images_below_size(
            article_name.article_name, amount=len(refactored_article_chunks) - 1, page=1, max_size_mb=3
        )
        chunks_with_images = []
        for chunk, image in zip(refactored_article_chunks, itertools.chain(image_suggestions, itertools.repeat(None))):
            if image:
                image_url = list(image.values())[0]
                chunk = f"{chunk}\n{image_tag_template.format(image_url=image_url, alt_text=image_url)}"
            chunks_with_images.append(chunk)
```

```
    article_body = "\n".join(chunks_with_images)
    article_body = self._clean_html(article_body)
    return article_body

def _generate_chunk(
    self, header: HeaderStruct, article_name: ArticleNameStruct, most_similar_chunks: list, keyword: KeywordStruct
):
    context_prompt = "Context: " + "\n".join(
        [
            f"{the_most_similar_chunk['header']} - {the_most_similar_chunk['content']}"
            for the_most_similar_chunk in most_similar_chunks
        ]
    )

    raw_keyword_prompt = Prompt.get_active_prompt("ADD_KEYWORD_PROMPT").user_prompt
    keyword_prompt = raw_keyword_prompt.format(keyword=keyword.text) if keyword and keyword.text else ""
    rich_text_article_prompt = Prompt.get_active_prompt("RICH_TEXT_ARTICLE_PROMPT")
    prompt = rich_text_article_prompt.user_prompt.format(
        relevant_header=header.header,
        name=article_name.article_name,
        keyword_prompt=keyword_prompt,
        words_per_paragraph=self._words_per_paragraph,
        context_prompt=context_prompt,
    )
    response = get_gpt_completion(
        prompt=prompt,
        system_prompt=rich_text_article_prompt.system_prompt.format(language=self.language),
    )
    return response

def generate_whole_article(self, article_name: ArticleNameStruct, outline: OutlineStruct) -> ArticleStruct:
    body = self._generate_body_at_once(article_name, outline)
    joined_keywords = ", ".join([kw.text for kw in outline.keywords[: len(outline.headers)]])
    shortName = slugify(article_name.article_name)
    article_details = self.generate_article_details(body)
    article = ArticleStruct(
        title=article_name.article_name,
        body=body,
        shortName=shortName,
        oneLiner=article_details.oneLiner,
        metaTitle=article_details.metaTitle,
        metaDescription=article_details.metaDescription,
        targetKeywords=joined_keywords,
    )
    return article

def _generate_body_at_once(self, article_name: ArticleNameStruct, outline: OutlineStruct):
    relevant_chunks = self._get_relevant_chunks(article_name.context)[: self._enrich_chunk_limit]
    self._enrich_chunks(relevant_chunks)
    most_similar_chunks_list = [
        self._get_most_similar_chunks(target_text=header.header, enriched_chunks=relevant_chunks)
        for header in outline.headers
    ]
    context_prompts = []
    raw_keyword_prompt = Prompt.get_active_prompt("ADD_KEYWORD_PROMPT").user_prompt
    for most_similar_chunks, keyword, header, index in zip(
        most_similar_chunks_list,
        itertools.chain(outline.keywords, itertools.repeat(None)),
        outline.headers,
        range(len(most_similar_chunks_list)),
    ):
        # mark each of the context prompts
        keyword_prompt = raw_keyword_prompt.format(keyword=keyword.text) if keyword and keyword.text else ""
        header_prompt = f"Relevant header: {header.tag} - {header.header}\n" if header else ""
        unpacked_chunks = "\n".join(
```

```
            [
                f"{the_most_similar_chunk['header']} - {the_most_similar_chunk['content']}"
                for the_most_similar_chunk in most_similar_chunks
            ]
        )
        base_context_prompt = (
            f"Context[{index}]: \n{unpacked_chunks}\n"
            f"Keyword instruction: {keyword_prompt}\n"
            f"Header instruction: {header_prompt}\n"
        )
        context_prompts.append(base_context_prompt)

    contexts_param = "\n\n".join(context_prompts)
    whole_rich_text_article_prompt = Prompt.get_active_prompt("WHOLE_RICH_TEXT_ARTICLE_PROMPT")
    prompt = whole_rich_text_article_prompt.user_prompt.format(
        name=article_name.article_name,
        words_per_paragraph=self._words_per_paragraph,
        contexts_prompt=contexts_param,
    )
    response = get_gpt_completion(
        prompt=prompt,
        system_prompt=whole_rich_text_article_prompt.system_prompt.format(language=self.language),
    )
    article_body = self._clean_html(response)
    return article_body

def generate_news_article(self, title: str, description: str, category: str) -> ArticleStruct:
    news_article_prompt = Prompt.get_active_prompt("NEWS_ARTICLE_PROMPT")
    prompt = news_article_prompt.user_prompt.format(
        title=title,
        description=description,
        category=category,
        language=self.language,
    )
    response = get_gpt_completion(
        prompt=prompt,
        system_prompt=news_article_prompt.system_prompt.format(language=self.language),
    )
    body = self._clean_html(response)

    joined_keywords = None
    shortName = slugify(title)
    article_details = self.generate_article_details(body)
    article = ArticleStruct(
        title=title,
        body=body,
        shortName=shortName,
        oneLiner=article_details.oneLiner,
        metaTitle=article_details.metaTitle,
        metaDescription=description,
        targetKeywords=joined_keywords,
        category=category,
    )
    return article
```