m-math_2025_001

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
кафедра математичного моделювання

СТВОРЕННЯ СЕРВІСУ EXAMMAKER.AI ДЛЯ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ТЕСТОВИХ ЗАВДАНЬ НА ОСНОВІ НАВЧАЛЬНИХ МАТЕРІАЛІВ ІЗ ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

Кваліфікаційна робота

Рівень вищої освіти – другий (магістерський)

Виконав:
студент 2 курсу, 601 групи
Ганек Микола Юрійович
Керівник:
доктор фізико-математичних наук,
професор Черевко І.М.

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

Чернівці - 2024

АНОТАЦІЯ

У кваліфікаційній роботі розроблено веб-додаток **"ExamMaker.ai"**, призначений для автоматичної генерації тестів на основі навчальних матеріалів. Система дозволяє користувачам завантажувати файли у форматах PDF, EPUB, DOC та DOCX, які аналізуються та обробляються з використанням API OpenAI GPT-4. Реалізовано функціонал налаштування тестів: вибір розділу навчального матеріалу для тестування, встановлення кількості та типів питань, а також формату зворотного зв'язку та стилю питань.

Розроблений додаток інтегровано з сервісами AWS S3 для зберігання файлів, Sentry для моніторингу, Slack для повідомлень та Google Analytics для відстеження активності користувачів. Технологічний стек включає Python, Django, Django REST Framework, Celery, Redis, PostgreSQL з розширенням pgvector, Docker та Nginx. Система готова до впровадження та може бути легко інтегрована в існуючі освітні процеси, сприяючи підвищенню ефективності навчання та оцінювання знань.

Ключові слова: автоматична генерація тестів, веб-додаток, штучний інтелект, OpenAI, Django, AWS S3, Celery, освіта, тестування, онлайн-навчання.

ABSTRACT

This qualification work presents the development of the web application **"ExamMaker.ai",** designed for the automated generation of tests based on educational materials. The system enables users to upload files in PDF, EPUB, DOC, and DOCX formats, which are analyzed and processed using the OpenAI GPT-4 API. The application includes features for customizing tests: selecting sections to test, setting the number and types of questions, as well as configuring feedback formats and question styles.

The developed application is integrated with AWS S3 for file storage, Sentry for monitoring, Slack for notifications, and Google Analytics for tracking user activity. The technological stack includes Python, Django, Django REST Framework, Celery, Redis, PostgreSQL with pgvector extension, Docker, and Nginx. The system is ready for deployment and can be seamlessly integrated into existing educational processes, enhancing the efficiency of learning and knowledge assessment.

Keywords: automated test generation, web application, artificial intelligence, OpenAI, Django, AWS S3, Celery, education, testing, online learning.

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів наукових досліджень інших авторів має посилання на відповідні джерела.

_____ М.Ю. Ганек

Зміст

Вступ

Актуальність теми. У сучасному освітньому середовищі ефективні методи навчання та оцінювання знань є ключовими для розвитку інтелектуального потенціалу суспільства. Дослідження в галузі когнітивної психології свідчать про значний вплив процесу тестування на закріплення та перенесення знань. Зокрема, Капріке та Рьодігер (2006**)** [1] встановили, що студенти, які практикували повторне відтворення інформації шляхом тестування, запам'ятали до 80% навчального матеріалу через тиждень, тоді як ті, хто лише повторював читання, запам'ятали лише близько 36%. Автори довели, що процес активного відтворення інформації під час тестування значно зміцнює пам'ять і покращує здатність до перенесення знань у нові контексти. Це явище, відоме як "ефект тестування", підкреслює важливість використання тестів не лише як засобу оцінювання, але й як ефективного інструменту навчання.

Аналогічно, Лайл та Кроуфорд (2011) [2] показали, що використання практики відтворення шляхом регулярних тестових сесій після лекцій призвело до підвищення результатів студентів на фінальних іспитах на 10–15% порівняно з тими, хто не брав участі в таких сесіях. У своїй роботі автори виявили, що регулярне тестування не лише покращує запам'ятовування, але й сприяє глибшому розумінню матеріалу. Це підкреслює роль активного відтворення в процесі навчання та необхідність інтеграції тестування як невід'ємної складової освітнього процесу.

Однак створення якісних тестових завдань є складним та ресурсозатратним процесом. Викладачі та тренери часто стикаються з браком часу та засобів для розробки індивідуалізованих тестів, що відповідають конкретним навчальним матеріалом. Це обмежує можливості адаптивного навчання та ефективного закріплення знань, що особливо актуально в умовах швидкого оновлення інформації та зростання обсягів навчальних матеріалів.

Стан проблеми. Існуючі сервіси для створення тестів здебільшого базуються на стандартизованих базах питань або вимагають ручного введення інформації, що не дозволяє швидко та ефективно створювати тести на основі конкретних книг чи статей. Відсутність автоматизованих рішень, які б використовували сучасні технології штучного інтелекту для генерації релевантних питань з навчальних матеріалів, залишається суттєвим недоліком у сфері освіти.

Мета роботи. Розробка сервісу **ExamMaker.ai** для автоматизованої генерації тестових завдань на основі книг та навчальних матеріалів із використанням штучного інтелекту. Сервіс призначений для широкого кола користувачів і спрямований на спрощення процесу створення якісних тестів, що сприяють ефективному навчанню та закріпленню знань.

Для реалізації мети кваліфікаційної роботи поставлено такі завдання:

1. Провести аналіз існуючих рішень та виявити їх недоліки.
2. Обґрунтувати вибір технологій та інструментів для реалізації сервісу.
3. Розробити архітектуру веб-сервісу з інтеграцією API OpenAI GPT-4 для генерації питань.
4. Реалізувати функціонал завантаження та обробки книг у форматах PDF, EPUB, DOC та DOCX.
5. Забезпечити можливість налаштування параметрів генерації тестів, включаючи вибір розділів та типів питань.
6. Інтегрувати хмарні сервіси (AWS S3) для зберігання файлів та забезпечення масштабованості.
7. Запровадити механізми безпеки та захисту даних користувачів, використовуючи сучасні стандарти шифрування та аутентифікації.
8. Провести тестування та оцінку ефективності розробленого сервісу в реальних умовах.

Наукове та практичне значення роботи. Робота поєднує сучасні технології штучного інтелекту з освітнім процесом, дозволяючи автоматизувати створення високоякісних тестових завдань. Враховуючи висновки Капріке та Рьодігера [1] про те, що повторне тестування покращує довготривалу пам'ять на 44%, а також результати Лайла та Кроуфорда [2] щодо підвищення успішності студентів на 10–15% завдяки регулярному тестуванню, розроблений сервіс сприятиме підвищенню якості навчання та закріпленню знань. Використання **ExamMaker.ai** може значно зменшити часові витрати на підготовку тестів, підвищити мотивацію до навчання та покращити процес оцінювання.

Підстави і вихідні дані для розробки. У роботі використано сучасні технології та інструменти: мову програмування Python, фреймворк Django та Django REST Framework для бекенд-розробки, API OpenAI GPT-4 для генерації тексту, базу даних PostgreSQL з розширенням pgvector для роботи з векторними даними, хмарне сховище AWS S3, а також інструменти для обробки PDF, EPUB та DOCX файлів. Проект реалізовано з використанням

контейнеризації (Docker) та оркестрації сервісів (Docker Compose), що забезпечує масштабованість та зручність розгортання.

Значущість роботи. Розроблений сервіс може бути впроваджений у освітніх закладах, на онлайн-платформах та в корпоративному навчанні, сприяючи автоматизації процесу оцінювання знань та самонавчання. Це особливо актуально в контексті сучасних освітніх тенденцій в Україні, де впровадження інноваційних технологій є пріоритетом розвитку галузі. **ExamMaker.ai** сприятиме підвищенню якості освіти, забезпечуючи доступ до персоналізованих та ефективних навчальних інструментів.

РОЗДІЛ 1. ПРИЗНАЧЕННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

1.1 Мета та область застосування системи ExamMaker.ai

Система **ExamMaker.ai** призначена для автоматизованої генерації тестових завдань на основі книг та інших навчальних матеріалів із використанням технологій штучного інтелекту. Головна мета розробки полягає у створенні інструменту, який спрощує процес підготовки якісних та релевантних тестів, надаючи користувачам можливість швидко генерувати індивідуалізовані тестові матеріали без значних часових та ресурсних витрат. **ExamMaker.ai** орієнтована на широкий спектр користувачів і може бути застосована в різних напрямах:

- **Освітні заклади**: школи, коледжі, університети та інші навчальні установи можуть використовувати систему для автоматичного створення тестів, що відповідають їхнім навчальним програмам та підручникам. Це сприятиме підвищенню ефективності навчального процесу та об'єктивності оцінювання знань студентів.
- **Онлайн-платформи навчання**: системи дистанційного навчання та масові відкриті онлайн-курси (MOOC) можуть інтегрувати **ExamMaker.ai** для автоматизації створення тестових завдань, що покращить якість освітнього контенту та залученість користувачів.
- **Корпоративне навчання та тренінги**: компанії та організації можуть застосовувати систему для розробки тестів на основі внутрішніх навчальних матеріалів, підвищуючи кваліфікацію співробітників та ефективність навчальних програм.
- **Індивідуальне використання**: викладачі, тренери, студенти та інші особи, зацікавлені в перевірці та закріпленні знань, можуть використовувати **ExamMaker.ai** для самостійного генерування тестів на основі власних або доступних навчальних матеріалів.
- **Видавництва та автори навчальної літератури**: система може бути використана для створення супровідних тестових матеріалів до навчальних посібників, підручників та інших освітніх видань, підвищуючи їхню цінність та конкурентоспроможність.
- **Науково-дослідні установи**: для проведення опитувань, анкетувань та оцінювання знань у рамках наукових досліджень, що вимагають швидкого створення специфічних тестових завдань.

Система **ExamMaker.ai** відповідає сучасним тенденціям в освіті, де особлива увага приділяється індивідуалізації навчального процесу, інтеграції інноваційних технологій та підвищенню мотивації до навчання. Використання штучного інтелекту дозволяє генерувати високоякісні тестові завдання, що сприяють глибшому засвоєнню матеріалу та розвитку критичного мислення.

Впровадження **ExamMaker.ai** має потенціал значно спростити процес створення тестів, зробити його більш доступним та ефективним для широкого кола користувачів, що позитивно вплине на якість освіти та професійний розвиток у різних галузях.

1.2 Огляд існуючих аналогів та обґрунтування вибору технологій

1.2.1 Аналіз аналогічних систем автоматичної генерації тестів

У сучасному освітньому просторі існує декілька сервісів, що пропонують автоматичну генерацію тестових завдань. Однак більшість з них мають певні обмеження щодо функціональності, налаштувань та інтеграції. Розглянемо деякі з них детальніше.

QuizGecko [3]

QuizGecko використовує алгоритми штучного інтелекту для створення тестів з різних джерел, включаючи текстові файли та PDF-документи. Підтримує типи питань, такі як вибір однієї правильної відповіді та правда/неправда. Однак сервіс не надає детального контролю над складністю питань чи їхньою кількістю. Обмежений API-доступ доступний лише у бізнес-плані, що може бути недосяжним для індивідуальних користувачів чи малих організацій.

Testportal [4]

Testportal дозволяє генерувати питання з тексту та документів, орієнтуючись на швидкість створення тестів. Сервіс підтримує інтеграцію через API, що зручно для корпоративних клієнтів. Проте вартість

використання може бути високою для окремих викладачів чи невеликих установ, а налаштування складності питань обмежене.

Revisely [5]

AI Quiz Generator від Revisely спрямований на простоту використання та дозволяє створювати питання з PDF-файлів та презентацій. Сервіс підходить для студентів та викладачів, але має обмеження щодо обсягу завантажуваного контенту та не надає можливості глибокого налаштування генерації питань. Інформація про API-доступ відсутня, що ускладнює інтеграцію з іншими платформами.

PrepAI [6]

PrepAI виділяється використанням таксономії Блума для створення питань різних когнітивних рівнів. Сервіс може обробляти складний технічний контент і надає API-доступ. Однак деталі щодо ціноутворення та доступності певних функцій можуть бути не прозорими, що створює перешкоди для потенційних користувачів.

Підсумовуючи огляд існуючих сервісів, що пропонують автоматичну генерацію тестових завдань, можна стверджувати, що вони мають такі недоліки:

- Обмежена функціональність: більшість сервісів не дозволяють детально налаштовувати складність, типи та кількість питань.
- Відсутність гнучкої інтеграції: обмежений або відсутній API-доступ ускладнює інтеграцію з іншими системами та платформами.
- Обмеження щодо обробки контенту: не всі сервіси здатні працювати зі складними або вузькопрофільними матеріалами.
- Цінові бар'єри: висока вартість використання може бути неприйнятною для індивідуальних користувачів або малих організацій.

Ці недоліки створюють потребу в новому інструменті, який би поєднував передові технології штучного інтелекту з гнучкістю налаштувань та доступністю для широкого кола користувачів.

1.2.2 Порівняльний аналіз технологій та інструментів для реалізації проекту

При розробці **ExamMaker.ai** був проведений ретельний аналіз доступних технологій з метою вибору оптимальних рішень, які забезпечать високу якість сервісу та його конкурентні переваги.

Технології штучного інтелекту та обробки природної мови

- **OpenAI GPT-4 API [19]**: сучасна модель для генерації природномовного тексту, здатна працювати з великим контекстом та генерувати високоякісні питання, відповідні складності вихідного матеріалу.
- **Альтернативні моделі**: такі як BERT чи T5, менш підходять для генерації тексту і більше орієнтовані на класифікацію або вилучення інформації.

Хмарні сервіси та зберігання даних

- **AWS S3 [15]**: забезпечує надійне та масштабоване зберігання файлів з високим рівнем безпеки. Інтеграція з іншими сервісами AWS спрощує управління інфраструктурою.
- **Альтернативи**: Google Cloud Storage та Azure Blob Storage мають схожий функціонал, але вибір AWS S3 обумовлений його популярністю та гнучкістю.

Фреймворки та мови програмування

- **Python з Django [7,8]**: поєднує простоту та швидкість розробки з потужністю та масштабованістю. Велика спільнота та багата екосистема бібліотек роблять цей вибір оптимальним.
- **Node.js з Express.js [9]**: хоча Node.js ефективний для деяких задач, він менш підходить для проектів, що вимагають тісної інтеграції з AI-моделями на Python.

Бази даних

- **PostgreSQL [13]**: потужна реляційна база даних з підтримкою складних запитів та розширень, ідеальна для зберігання структурованих даних.

- **Альтернативи**: MySQL та MongoDB, але PostgreSQL надає більшу функціональність для потреб проекту, особливо з розширенням **pgvector** для роботи з векторними даними.

Інструменти для обробки асинхронних завдань

- **Celery з Redis [10,14]**: стандартне рішення для обробки фонових завдань у Python-додатках, забезпечує масштабованість та надійність.

Контейнеризація та розгортання

- **Docker та Docker Compose [20]**: дозволяють створювати ізольовані середовища для кожного компонента, спрощуючи розгортання та масштабування.

Інтеграції та сторонні сервіси

- **Sentry [16]**: для моніторингу та відстеження помилок у реальному часі.
- **Slack SDK**: для налаштування оповіщень та швидкої комунікації з командою розробників.

1.2.3 Обґрунтування вибору мови програмування та фреймворку

Вибір мови програмування — Python

- **Підтримка штучного інтелекту**: Python є стандартом де-факто в області машинного навчання та штучного інтелекту, що робить його ідеальним для інтеграції з OpenAI API [19] та іншими AI-сервісами.
- **Багата екосистема**: велика кількість бібліотек для різних завдань, від обробки даних до веб-розробки.
- **Простота та ефективність**: легкий у вивченні та використанні, що прискорює розробку та зменшує кількість помилок.

Вибір фреймворку — Django [7]

- **Швидка розробка**: Django надає вбудовані інструменти та бібліотеки, що дозволяє зосередитися на бізнес-логіці проекту.
- **Безпека**: вбудовані засоби захисту від поширених веб-атак, таких як SQL-ін'єкції, XSS та CSRF.

- **Масштабованість та гнучкість**: підтримує легку інтеграцію з іншими сервісами, що важливо для майбутнього розширення функціональності.
- **Активна спільнота**: велика кількість ресурсів та підтримки, що спрощує вирішення проблем та впровадження нових можливостей.

Інтеграція з обраними технологіями

- **Celery та Redis [10,14]**: забезпечують ефективну обробку фонових завдань та масштабованість сервісу.
- **PostgreSQL [13]**: сумісний з Django ORM, що спрощує роботу з базою даних та забезпечує високу продуктивність.
- **Docker та Docker Compose [20]**: полегшують процес розробки, тестування та розгортання додатка в різних середовищах.

Вибір Python та Django обумовлений необхідністю інтеграції з передовими AI-технологіями, потребою в швидкій розробці та забезпеченні масштабованості системи. Ці технології дозволяють створити потужний та гнучкий сервіс, здатний обробляти складний контент та задовольняти потреби різних категорій користувачів.

1.3 Постановка завдання

У цьому розділі визначаються основні вимоги до системи **ExamMaker.ai**, встановлюються її функціональні можливості, основні параметри та характеристики, а також обмеження та припущення, враховані при розробці.

1.3.1 Вимоги до функціональності системи

Завантаження навчальних матеріалів

- Надання користувачам можливості завантажувати документи у форматах PDF, EPUB, DOC та DOCX, які містять навчальний контент.
- Автоматичний аналіз структури завантаженого документа з визначенням глав, розділів та підрозділів.

Генерація тестових завдань

- Автоматичне створення тестових питань на основі завантажених матеріалів із використанням технологій штучного інтелекту.
- Підтримка різних типів питань: з однією правильною відповіддю, з кількома правильними відповідями, правда/неправда, відкриті питання.

- Можливість вибору стилю питань (фактичні, аналітичні, на основі цитат тощо).

Налаштування параметрів генерації

- Користувачі можуть налаштовувати кількість питань у тесті.
- Вибір конкретних розділів або тем для генерації питань.
- Встановлення рівня складності питань (базовий, середній, високий).

Проходження тестів

- Надання інтерфейсу для онлайн-проходження згенерованих тестів.
- Миттєвий зворотний зв'язок після кожного питання або підсумковий результат після завершення тесту.
- Відображення правильних відповідей та надання пояснень.

Управління користувачами

- Реєстрація та аутентифікація користувачів з використанням безпечних протоколів.
- Розмежування ролей користувачів (звичайний користувач, адміністратор).
- Збереження історії пройдених тестів та результатів.

Інтеграція зі сторонніми сервісами

- Відстеження помилок та винятків за допомогою інструментів моніторингу (Sentry [16]).
- Надсилання оповіщень та сповіщень через комунікаційні платформи (Slack [18]).
- Збирання статистики та аналітики користувацької активності (Google Analytics).

Зберігання та безпека даних

- Збереження завантажених файлів у хмарному сховищі (AWS S3 [15]).
- Використання бази даних для зберігання структурованих даних (PostgreSQL [13]).
- Забезпечення конфіденційності та захисту персональних даних користувачів відповідно до стандартів (наприклад, GDPR).

Масштабованість та продуктивність

- Система повинна обробляти одночасно велику кількість запитів та користувачів.
- Використання інструментів для обробки фонових завдань (Celery, Redis [10,14]) та оптимізації продуктивності.

1.3.2 Основні параметри та характеристики системи

Технічні характеристики

- Мова програмування: Python 3.11+
- Фреймворк бекенду: Django та Django REST Framework [7,8]
- База даних: PostgreSQL [13] з розширенням pgvector
- Хмарне сховище: AWS S3 [15]
- Обробка фонових завдань: Celery [10] з використанням Redis [14]
- Контейнеризація: Docker та Docker Compose [20]
- Веб-сервер: Nginx [12]
- Сервер додатків: Gunicorn [11]

Функціональні характеристики

- Максимальний розмір завантажуваних файлів: 50 МБ
- Підтримувані формати файлів: PDF, EPUB, DOC, DOCX
- Середній час генерації тесту з 10 питань: до 2 хвилин (залежно від обсягу матеріалу та навантаження)
- Кількість одночасних користувачів: система оптимізована для підтримки до 1000 одночасних користувачів з можливістю масштабування

Безпека та захист даних

- Шифрування даних: використання SSL/TLS для захищеного з'єднання
- Автентифікація: використання JWT-токенів та захист від CSRF-атак
- Зберігання паролів: використання алгоритмів хешування з сіллю (наприклад, bcrypt)

Інтеграція зі сторонніми сервісами

- OpenAI API [19]: для генерації питань
- Sentry SDK [16]: для моніторингу помилок

- Slack SDK [18]: для оповіщень адміністраторам

Користувацький інтерфейс

- Веб-інтерфейс: адаптивний дизайн для різних пристроїв (десктоп, планшет, смартфон)

1.3.3 Обмеження та припущення при розробці

Обмеження

- Ліміти API OpenAI [19]: використання OpenAI API має обмеження щодо кількості запитів та обсягу даних, що може впливати на швидкість генерації питань.
- Підтримувані формати файлів: наразі підтримуються PDF, EPUB, DOC та DOCX; інші формати можуть бути додані в майбутньому.
- Необхідність структурованого контенту: для коректної генерації питань матеріали повинні мати чітку структурну організацію (зміст, заголовки тощо).

Припущення

- Якість завантажених матеріалів: передбачається, що файли містять текстовий контент (не відскановані зображення), що дозволяє коректно витягувати текст.
- Користувачі мають базові навички роботи з веб-додатками: інтерфейс розроблений з урахуванням стандартних практик UX/UI.
- Доступ до інтернету: стабільне інтернет-з'єднання необхідне для роботи системи, оскільки більшість функціональності залежить від зовнішніх сервісів.

Технічні припущення

- Масштабованість інфраструктури: при зростанні навантаження можлива горизонтальна масштабованість шляхом додавання серверів та ресурсів.
- Стабільність сторонніх сервісів: передбачається безперебійна робота сервісів, таких як OpenAI API [19] та AWS S3 [15]; збої можуть тимчасово впливати на функціональність системи.

Юридичні та етичні аспекти

- Дотримання ліцензій та політик: використання API та інших сервісів здійснюється відповідно до їхніх ліцензій та умов використання.
- Конфіденційність та захист даних: система відповідає вимогам щодо захисту персональних даних користувачів (наприклад, GDPR).

Обмеження щодо обробки контенту

- Тематика матеріалів: система може не гарантувати коректну генерацію питань з матеріалів, що містять вузькопрофільну термінологію або спеціалізований контент.
- Обробка зображень та графіків: на даний момент система не обробляє графічний контент та не генерує питання на його основі.

РОЗДІЛ 2. ОПИС ТА ОБҐРУНТУВАННЯ ПРОЕКТНИХ РІШЕНЬ

2.1 Опис функціонування системи

2.1.1 Загальна архітектура

ExamMaker.ai побудована на основі багаторівневої архітектури, що забезпечує розподіл відповідальностей, гнучкість та масштабованість системи. Архітектура складається з таких основних рівнів:

- Клієнтський рівень (Frontend): відповідає за взаємодію з користувачем через веб-інтерфейс. Розроблений з використанням сучасних веб-технологій, забезпечує адаптивний дизайн для різних пристроїв і зручну навігацію.
- Серверний рівень (Backend): реалізований на основі фреймворку Django та Django REST Framework [7,8]. Відповідає за обробку запитів від клієнта, управління бізнес-логікою та взаємодію з базою даних і зовнішніми сервісами.
- База даних: використовується PostgreSQL [13] з розширенням pgvector для зберігання структурованих даних, таких як інформація про користувачів, книги, розділи, питання та результати тестів.
- Зовнішні сервіси включають::
    - OpenAI GPT-4 API [19]: використовується для генерації тестових питань на основі навчальних матеріалів.
    - AWS S3 [15]: хмарне сховище для зберігання завантажених користувачами файлів.
    - Sentry: інструмент для моніторингу помилок та винятків у реальному часі.
    - Slack [18]: платформа для оповіщення команди розробників про критичні події.
- Асинхронні завдання: Celery з Redis [10,14] використовується для обробки ресурсомістких завдань у фоновому режимі, таких як аналіз завантажених книг та генерація питань.

Розподіл на фронтенд- і бекенд-частини дозволяє незалежно розробляти, тестувати та масштабувати кожен з компонентів системи. Це

забезпечує гнучкість при впровадженні нових функцій та оптимізацію продуктивності.

2.1.2 Сценарії використання

Типові дії користувача:

1. Реєстрація та автентифікація:
    - Користувач створює обліковий запис або входить у систему за допомогою наявних облікових даних або через Google OAuth 2.0.
2. Завантаження навчального матеріалу:
    - Користувач завантажує файл з навчальним матеріалом у підтримуваному форматі (PDF, EPUB, DOC, DOCX).
    - Система зберігає файл у AWS S3 [15] та реєструє метадані в базі даних.
    - Запускається асинхронне завдання для аналізу структури документа та розбиття його на розділи.
3. Налаштування параметрів генерації:
    - Користувач вибирає книгу та розділи для генерації питань.
    - Встановлює кількість питань, рівень складності та типи питань.
4. Генерація питань:
    - Система відправляє запити до OpenAI GPT-4 API [19] з відповідними параметрами.
    - Отримані питання зберігаються у базі даних та стають доступними для користувача.
5. Проходження тесту:
    - Користувач розпочинає тестування, відповідаючи на згенеровані питання.
    - Система надає зворотний зв'язок і підсумковий результат після завершення тесту.
6. Перегляд результатів та аналітика:
    - Користувач може переглядати історію пройдених тестів та аналізувати свої результати.
    - Адміністратори мають доступ до агрегованих даних для оцінки ефективності системи.

2.1.3 Потоки даних

Переміщення даних між компонентами системи:

- Клієнтський інтерфейс відправляє запити до серверної частини через REST API для виконання дій користувача.
- Серверна частина обробляє запити, взаємодіє з базою даних та зовнішніми сервісами, а також управляє бізнес-логікою.
- База даних PostgreSQL [13] зберігає структуровані дані, забезпечуючи швидкий та надійний доступ до інформації.
- AWS S3 [15] використовується для зберігання великих файлів, таких як завантажені книги, забезпечуючи масштабованість та доступність.
- Celery з Redis управляє асинхронними завданнями, дозволяючи виконувати ресурсомісткі процеси у фоновому режимі без впливу на продуктивність основного додатку.
- OpenAI GPT-4 API [19] використовується для генерації питань на основі наданого тексту з навчальних матеріалів.
- Sentry та Slack [18] забезпечують моніторинг та оповіщення про помилки та критичні події, сприяючи швидкому реагуванню на проблеми.

Взаємодія з базою даних та зовнішніми сервісами:

- При завантаженні книги сервер зберігає метадані у базі даних та сам файл у AWS S3 [15].
- Під час аналізу книги система використовує Unstructured.io [21] та Fitz (PyMuPDF) для витягування тексту та структури документа.
- Для генерації питань сервер відправляє запити до OpenAI GPT-4, отримує згенеровані питання та зберігає їх у базі даних.
- Результати тестування користувача зберігаються для подальшого аналізу та відображення у звітах.

2.2 Структурна схема системи

2.2.1 Компоненти системи

Система **ExamMaker.ai** складається з таких основних компонентів:

1. Компонент **"Book"**:
    - Призначення: управління завантаженням та зберіганням книг, а також обробкою їхнього вмісту.
    - Функціональність:
        - Прийом та валідація файлів від користувачів.
        - Збереження файлів у AWS S3 [15].
        - Збереження метаданих книг у базі даних.
        - Ініціація процесу аналізу книги.
2. Компонент **"Chapter"**:
    - Призначення: управління розділами книг та підготовка матеріалів для генерації питань.
    - Функціональність:
        - Розбиття книги на розділи з використанням структури документа.
        - Збереження структури розділів у базі даних.
        - Генерація резюме для кожного розділу за допомогою OpenAI GPT-4 [19].
3. Компонент **"Question"**:
    - Призначення: генерація та управління тестовими питаннями.
    - Функціональність:
        - Генерація питань на основі тексту розділів.
        - Збереження питань та відповідей у базі даних.
        - Категоризація питань за типом та складністю.
4. Компонент **"Authentication"**:
    - Призначення: забезпечення автентифікації та авторизації користувачів.
    - Функціональність:
        - Реєстрація нових користувачів.
        - Вхід у систему з використанням JWT-токенів.
        - Інтеграція з Google OAuth 2.0 для спрощеного входу.
        - Управління ролями та дозволами користувачів.

5. Компонент "**Base**":
    - Призначення: надає спільні сервіси та утиліти для інших компонентів.
    - Функціональність:
        - Базові класи моделей та ViewSet'ів.
        - Утиліти для роботи з файлами, логування, обробка помилок.
        - Інтеграція зі сторонніми сервісами, такими як Sentry [16] та Slack [18].

2.2.2 Взаємодія між компонентами

Взаємодія між компонентами системи:

- **"Book" та "Chapter":** після завантаження книги компонент "Book" передає інформацію компоненту "Chapter" для розбиття на розділи та підготовки тексту для генерації питань.
- **"Chapter" та "Question":** компонент "Chapter" надає текст розділів компоненту "Question" для генерації питань з використанням OpenAI GPT-4 [19].
- **"Authentication":** контролює доступ користувачів до функціональності інших компонентів, перевіряючи права та дозволи.
- **"Base":** надає спільні ресурси та сервіси, які використовуються всіма компонентами, забезпечуючи узгодженість та повторне використання коду.

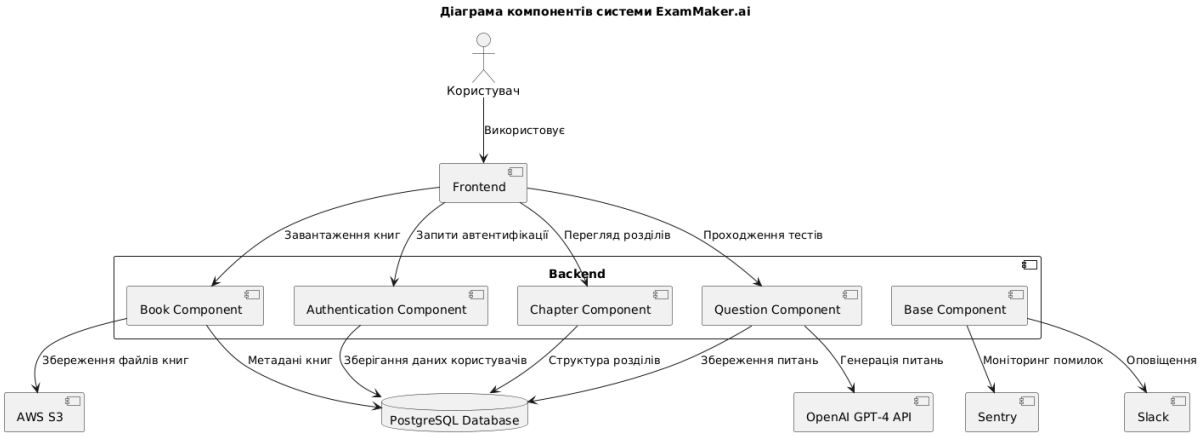
Використання API та сервісів для комунікації:

- Компоненти взаємодіють через внутрішні API, реалізовані за допомогою Django REST Framework [8].
- Асинхронні завдання керуються Celery та Redis [10,14], дозволяючи компонентам обмінюватися повідомленнями та виконувати завдання у фоновому режимі.
- Зовнішні сервіси інтегруються через відповідні SDK або API-клієнти, забезпечуючи надійну та ефективну взаємодію.

2.2.3 Схема взаємодії

Опис діаграми компонентів системи:

- Користувач: взаємодіє з системою через веб-браузер або мобільний пристрій.
- Frontend: відображає інтерфейс користувача, відправляє запити до Backend, отримує та відображає результати.
- Backend: складається з компонентів "Book", "Chapter", "Question", "Authentication", "Base".
- База даних: зберігає дані про користувачів, книги, розділи, питання, результати тестів та іншу необхідну інформацію.
- Зовнішні сервіси**:** OpenAI GPT-4 API [19], AWS S3 [15], Sentry [16], Slack [18].



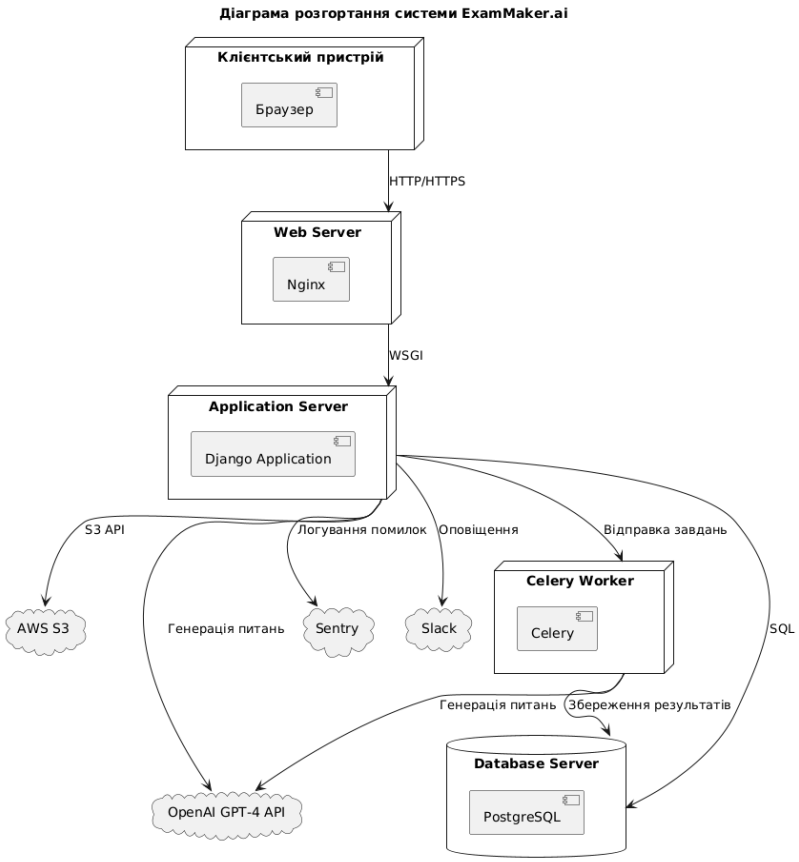
2.2.4 Діаграма розгортання (Deployment Diagram)

Опис діаграми

Діаграма розгортання відображає фізичне розташування програмних компонентів системи **ExamMaker.ai** на серверах та їх взаємозв'язок. Вона ілюструє, як компоненти системи розміщені на апаратних ресурсах та як вони взаємодіють між собою та з зовнішніми сервісами.

Основні вузли:

- Клієнтський пристрій: пристрій користувача з веб-браузером або мобільним додатком.
- Веб-сервер (Web Server): сервер, що обслуговує фронтенд частину додатку (наприклад, Nginx [12]).
- Сервер додатків (Application Server): сервер, де працює бекенд додатку на базі Django та Gunicorn [11].
- Сервер асинхронних завдань (Celery Worker [10]): сервер(и) для виконання асинхронних завдань.
- Сервер бази даних (Database Server): сервер з базою даних PostgreSQL [13].
- Зовнішні сервіси: OpenAI GPT-4 API [19], AWS S3 [15], Sentry [16], Slack [18].



2.2.5 Обґрунтування вибору структурних рішень

Переваги обраної структури:

- Модульність: розподіл системи на окремі компоненти полегшує розробку, тестування та підтримку кожного з них незалежно.
- Масштабованість: можливість окремо масштабувати компоненти залежно від навантаження та вимог.
- Повторне використання коду: компонент "Base" забезпечує спільні функції, що зменшує дублювання та сприяє узгодженості коду.
- Гнучкість: використання API та сервісів спрощує інтеграцію з новими функціями або зовнішніми сервісами в майбутньому.
- Безпека: централізоване управління автентифікацією та авторизацією підвищує безпеку системи.

2.3 Діаграми процесів та їх принципові схеми

2.3.1 Діаграма діяльності (Activity Diagram)

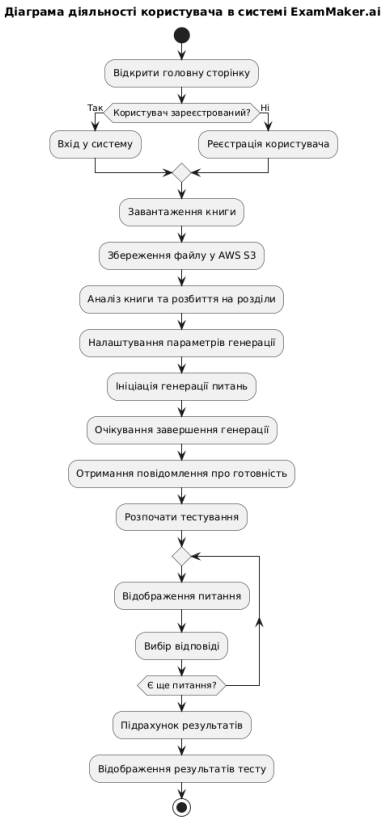
Опис діаграми:

Діаграма діяльності ілюструє основні процеси у системі, відображаючи послідовність дій користувача та системи під час виконання завдань.

Основні процеси:

1. Реєстрація/Вхід:
    - Користувач відкриває сторінку входу.
    - Вводить облікові дані або використовує Google OAuth 2.0.
    - Система перевіряє дані та надає доступ.
2. Завантаження книги:
    - Користувач переходить до розділу завантаження.
    - Обирає файл та завантажує його.
    - Система зберігає файл та ініціює обробку.
3. Налаштування генерації:
    - Користувач обирає книгу та розділи.
    - Встановлює параметри генерації питань.
4. Генерація питань:
    - Система відправляє запити до OpenAI GPT-4 API [19].
    - Отримує та зберігає питання у базі даних.

5. Проходження тесту:
    - Користувач розпочинає тест.
    - Відповідає на питання.
    - Отримує результати та зворотний зв'язок.



2.3.2 Діаграма послідовності (Sequence Diagram)

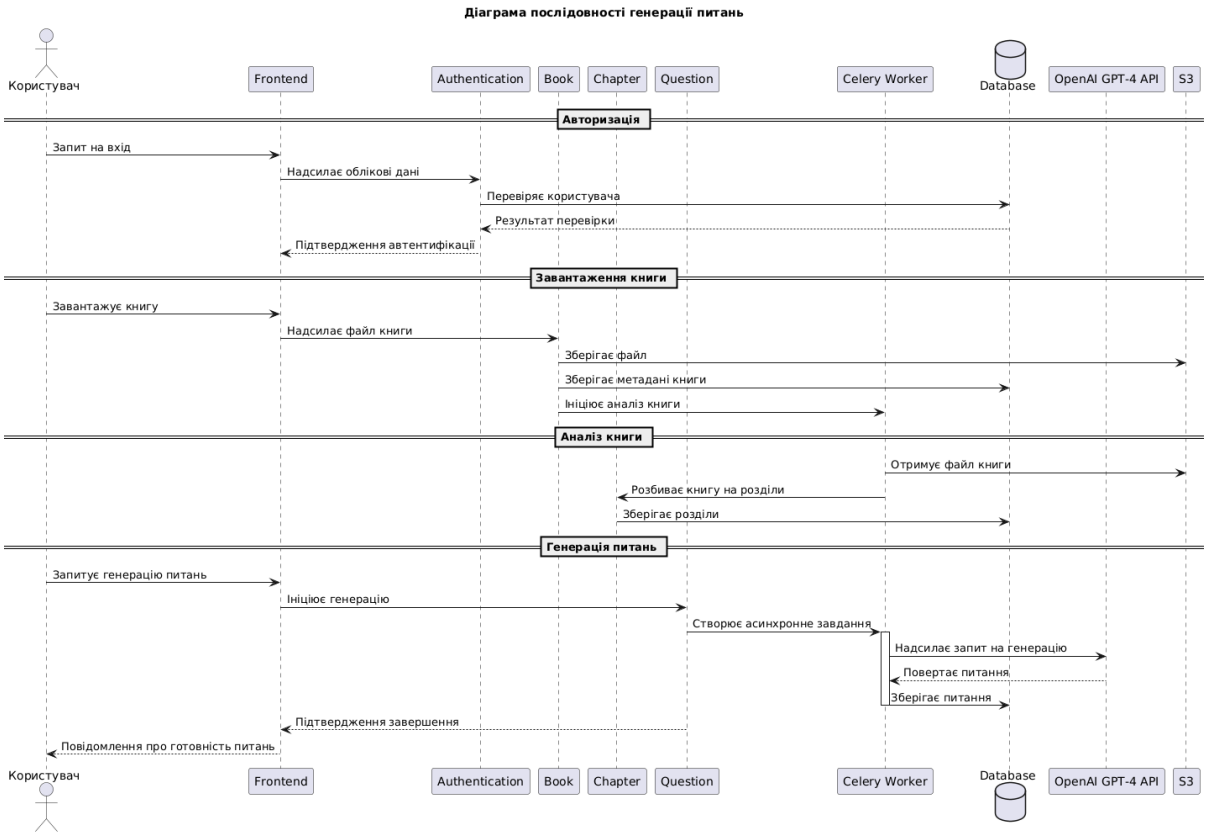
Приклад сценарію: Генерація питань на основі завантаженої книги

Учасники:

- Користувач
- Frontend
- Backend (Authentication, Book, Chapter, Question)
- OpenAI GPT-4 API [19]
- База даних

Послідовність дій:

1. Користувач ініціює генерацію питань.
2. Frontend відправляє запит до Backend з необхідними параметрами.
3. Backend автентифікує користувача.
4. Backend створює асинхронне завдання для генерації питань.
5. Celery виконує завдання, звертаючись до OpenAI GPT-4 API [19].
6. Отримані питання зберігаються у базі даних.
7. Система повідомляє користувача про завершення генерації.

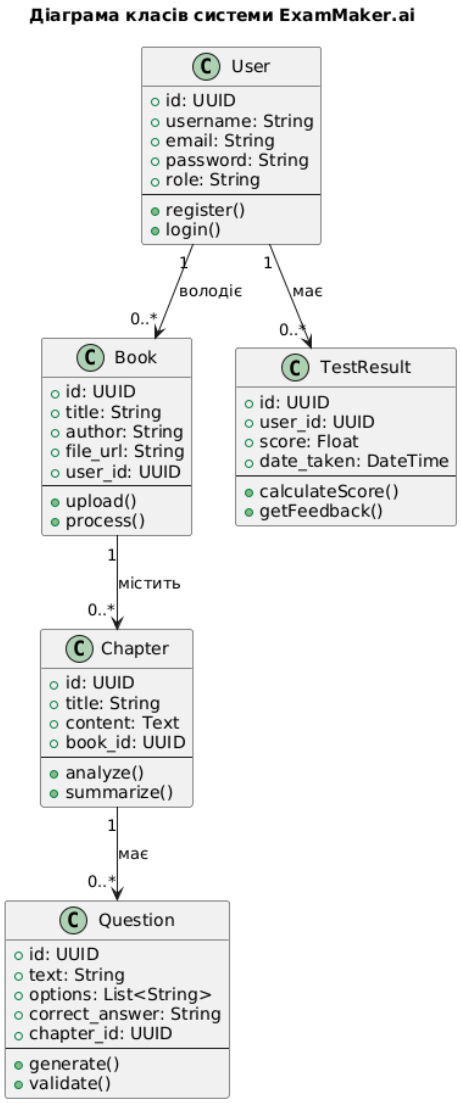


2.3.3 Діаграма класів (Class Diagram)

Основні класи та їх зв'язки:

- User:
  - Атрибути: id, username, email, password, role
  - Зв'язки: має багато Book, має багато TestResult
- Book:
  - Атрибути: id, title, author, file_url, user_id

    - Зв'язки: належить User, складається з багатьох Chapter
- Chapter:
    - Атрибути: id, title, content, book_id
    - Зв'язки: належить Book, має багато Question
- Question:
    - Атрибути: id, text, options, correct_answer, chapter_id
    - Зв'язки: належить Chapter
- TestResult:
    - Атрибути: id, user_id, score, date_taken
    - Зв'язки: належить User



2.3.4 Обґрунтування вибору алгоритмів та методів

Вибір алгоритмів та підходів базується на наступних факторах:

- Інтеграція з OpenAI GPT-4 [19]:
  - Забезпечує високоякісну генерацію питань завдяки потужним можливостям моделі.
  - Дозволяє адаптувати параметри генерації для отримання бажаних результатів.
- Асинхронна обробка з Celery та Redis [10,14]:
  - Дозволяє виконувати ресурсомісткі завдання у фоновому режимі.
  - Покращує продуктивність та швидкодію системи.
- Використання PostgreSQL [13] з pgvector:
  - Дозволяє зберігати та обробляти векторні представлення даних.
  - Підвищує ефективність пошуку та аналізу даних.
- Django та Django REST Framework [7,8]:
  - Забезпечують швидку та безпечну розробку веб-додатків.
  - Надають зручні інструменти для створення RESTful API.

2.4 Вибір та інтеграція зовнішніх сервісів

2.4.1 OpenAI GPT-4 [19]

Обґрунтування використання:

- Забезпечує високий рівень розуміння контексту та генерації якісних питань.
- Підтримує різні типи питань та предметні області.

Особливості інтеграції:

- Використання офіційного SDK для взаємодії з API.
- Обробка можливих помилок та затримок через асинхронні завдання.

- Оптимізація запитів для зменшення витрат та часу очікування.

2.4.2 Unstructured.io [21]

Використання для парсингу документів:

- Підтримка різних форматів файлів.
- Збереження структури документа для коректного розбиття на розділи.

Інтеграція:

- Використання офіційних бібліотек для безпосередньої обробки файлів у коді.

2.4.3 Fitz (PyMuPDF)

Робота з PDF та EPUB файлами:

- Витягування тексту та метаданих з документів.
- Підтримка різних форматів та робота з графічним контентом за потреби.

Приклад коду:

```
import fitz


def get_document_text(file_path: str) -> str:
    doc = fitz.open(file_path)
    text = ""
    for page in doc:
        text += page.get_text()
    return text
```

2.4.4 Сервіси аутентифікації

Інтеграція з Google OAuth 2.0:

- Спрощує процес реєстрації та входу для користувачів.
- Підвищує безпеку через використання надійного стороннього сервісу.

Інтеграція здійснюється за допомогою бібліотек, таких як django-allauth.

2.4.5 Інші інтеграції

Slack [18]:

- Використовується для оперативного оповіщення команди про помилки та критичні події.

Google Analytics:

- Відстежує активність користувачів та збирає статистику для аналізу та прийняття рішень.

2.5 Безпека та контроль доступу

2.5.1 Система ролей та дозволів

Реалізація різних рівнів доступу:

- Зареєстрований користувач: доступ до базових функцій системи.
- Адміністратор: розширені права для управління системою та користувачами.

Реалізація в коді:

- Використання вбудованої системи дозволів Django [7].
- Додаткові перевірки на рівні представлень та моделей.

2.5.2 Захист даних

Методи шифрування та зберігання конфіденційної інформації:

- Паролі: зберігаються в зашифрованому вигляді з використанням алгоритмів хешування з сіллю.
- З'єднання: використання SSL/TLS для захищеного з'єднання між клієнтом та сервером.
- Конфіденційні дані: зберігаються у змінних оточення та не потрапляють у репозиторій коду.

Захист від атак:

- CSRF-захист: вбудований у Django.
- SQL-ін'єкції: запобігається через використання ORM.
- XSS-захист: екранування введених даних.

2.5.3 Обмеження для користувачів

Реалізація обмежень для безкоштовних та платних користувачів:

- Безкоштовні користувачі:
    - Обмеження на кількість завантажених книг.
    - Ліміт на кількість згенерованих питань.
- Платні користувачі:
    - Розширені можливості без обмежень.

Реалізація обмежень:

- Перевірки на рівні бізнес-логіки перед виконанням операцій.

2.5.4 Обґрунтування обраних рішень з безпеки

- Стандартні механізми безпеки Django [7] забезпечують надійність та перевіреність.
- Шифрування даних гарантує конфіденційність інформації користувачів.
- Розмежування доступу та ролей дозволяє контролювати дії користувачів та запобігати несанкціонованому доступу.
- Обмеження для користувачів сприяють стійкості бізнес-моделі та стимулюють переходу на платні плани.
- Моніторинг та оповіщення через Sentry [16] та Slack [18] підвищують стабільність та безпеку системи.

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ РОБОТИ

3.1 Розробка алгоритмів та їх опис

3.1.1 Алгоритм обробки завантажених документів

Покроковий опис процесу парсингу та структурування

У проекті ExamMaker.ai процес обробки завантажених документів є критичним етапом, оскільки від якості розпізнавання та структурування тексту залежить точність згенерованих питань. При реалізації цього алгоритму виникли наступні задачі:

- Різноманітність форматів документів: необхідно було забезпечити підтримку різних форматів файлів (PDF, EPUB, DOC, DOCX), кожен з яких має свої особливості.
- Збереження структури документа: важливо було не лише витягнути текст, але й зберегти логічну структуру (глави, розділи, підрозділи) для коректної генерації питань.
- Обробка великих файлів: деякі документи можуть мати значний обсяг, що вимагало оптимізації алгоритмів для забезпечення швидкої та надійної обробки.

Реалізація алгоритму обробки документів включає наступні кроки:

1. Завантаження файлу користувачем:
    - Користувач через веб-інтерфейс завантажує файл з навчальним матеріалом.
    - Встановлено обмеження на розмір файлу та перевірку підтримуваних форматів для забезпечення стабільності системи.
2. Збереження файлу в AWS S3 [15]:
    - Використання AWS S3 забезпечує надійне та масштабоване зберігання файлів.
    - Розроблено утиліту для завантаження файлів у S3, яка генерує унікальні ключі для кожного файлу.

Приклад коду завантаження файлу в S3:

```
import boto3
import uuid


def upload_file_to_s3(file):
    s3 = boto3.client('s3')
    file_key = f"uploads/{uuid.uuid4()}.pdf"
    s3.upload_fileobj(file, 'exam-maker-bucket', file_key)
    return file_key
```

3. Ініціація асинхронного завдання для обробки:
    - Щоб не затримувати користувача, використовується Celery [10] для асинхронної обробки документів.
    - Після завантаження файлу створюється завдання, яке виконується у фоновому режимі.
4. Витягування текстового контенту:
    - Для кожного формату файлу застосовуються відповідні бібліотеки:
        - PDF та EPUB: бібліотека Fitz (PyMuPDF).
        - DOC та DOCX: бібліотека python-docx.
    - Створено абстрактний клас DocumentParser, від якого успадковуються конкретні парсери для кожного формату.

Приклад класу парсера для PDF:

```
import fitz


class PDFParser(DocumentParser):
    def parse(self, file_path):
        doc = fitz.open(file_path)
        text = ""
        for page in doc:
```

```
            text += page.get_text()

        return text
```

5. Розбиття тексту на структурні елементи:
    - Реалізовано алгоритм для виявлення заголовків та підзаголовків на основі стилів тексту (розмір шрифту, жирність) та структури документа.
    - Для DOCX-файлів використовуються стилі заголовків, а для PDF — аналіз розміру та форматування тексту.
6. Задачі та їх вирішення:
    - Непослідовність форматування: в деяких документах відсутня чітка структура заголовків. Було реалізовано механізм налаштування параметрів парсингу, щоб користувач міг вказати критерії виявлення заголовків.
    - Різноманітність стилів: через різні стилі форматування в документах, алгоритм був адаптований для автоматичного виявлення патернів заголовків.
7. Збереження структури в базі даних:
    - Структура документа зберігається у вигляді ієрархії моделей Book, Chapter, SubChapter.
    - Використовуються зв'язки ForeignKey для встановлення відносин між моделями.
8. Обробка великих файлів:
    - Для оптимізації розбивається текст на менші частини (chunks) та обробляється послідовно.
    - Це дозволяє зменшити навантаження на пам'ять та підвищити швидкість обробки.
9. Повідомлення користувача про завершення:
    - Після завершення обробки користувач отримує сповіщення через веб-інтерфейс або електронну пошту.
    - Реалізовано механізм оповіщень за допомогою WebSocket та бібліотеки Django Channels для реального часу.

Приклад коду оповіщення:

```
from channels.layers import get_channel_layer

from asgiref.sync import async_to_sync
```

```
def notify_user(user_id, message):
    channel_layer = get_channel_layer()
    async_to_sync(channel_layer.group_send)(
        f"user_{user_id}",
        {"type": "user.message", "message": message}
    )
```

3.1.2 Алгоритм генерації резюме

Використання TextSummaryService та взаємодія з OpenAI

Після парсингу та структурування документа виникла потреба в отриманні короткого резюме для кожного розділу. Це дозволяє стисло передавати основний зміст розділу для подальшої генерації питань.

Реалізація алгоритму генерації резюме:

1. Отримання тексту розділу:
    - З бази даних витягується текст конкретного розділу.
    - Обмежується довжина тексту, щоб уникнути перевищення лімітів OpenAI API [19].
2. Підготовка запиту до OpenAI:
    - Створено спеціальний промпт, який інструктує модель створити коротке та інформативне резюме.
    - Враховується, що модель повинна ігнорувати зайві деталі та фокусуватися на ключових аспектах.

Приклад промпту:

"Please summarize the following text, highlighting the main ideas and concepts. The summary should not exceed 150 words.\n\n{text of the section}"

3. Відправка запиту до OpenAI AP [19]I:
    - Використовується бібліотека openai для взаємодії з API.
    - Реалізовано TextSummaryService, який абстрагує цей процес.

Код сервісу:

```
import openai


class TextSummaryService:
    def summarize(self, text):
        prompt = f"Please summarize the following text...\n\n{text}"
        response = openai.Completion.create(
            engine='text-davinci-003',
            prompt=prompt,
            max_tokens=150,
            temperature=0.5
        )
        summary = response.choices[0].text.strip()
        return summary
```

4. Обробка відповіді та збереження:
    - Отримане резюме перевіряється на наявність помилок чи невідповідностей.
    - Зберігається резюме в базі даних у відповідному полі моделі Chapter.

Вирішені задачі:

- Обмеження OpenAI API [19]: щоб уникнути лімітів на кількість символів у запиті, великі розділи розбиваються на менші частини або скорочується текст.
- Якість резюме: шляхом експериментів з промптами та параметрами вдалося покращити якість згенерованих резюме.

3.1.3 Алгоритм генерації екзаменаційних питань

Як на основі структурованого контенту генеруються питання

Генерація якісних тестових питань на основі навчального матеріалу є основною функцією системи.

Етапи реалізації:

1. Вибір матеріалу для генерації:
    - Користувач обирає книгу та розділи для генерації питань.
    - Надається можливість вибрати рівень складності та типи питань (з вибором відповіді, відкриті питання тощо).
2. Підготовка тексту:
    - Об'єднується текст вибраних розділів або використовуються резюме, якщо текст занадто великий.
    - Текст розбивається на chunks розміром до 2000 символів, щоб відповідати лімітам OpenAI.
3. Формування промпту для генерації:
    - Створюється динамічний промпт, який враховує параметри, обрані користувачем.
    - Вказується кількість питань, тип та бажаний рівень складності.

Приклад промпту:

"Based on the following text, create 5 multiple-choice questions at a 'medium' difficulty level. Each question should have 4 answer options, one of which is correct.\n\n{text}"

4. Відправка запиту до OpenAI:
    - Використовуються асинхронні завдання Celery для відправки запитів, щоб не затримувати інші процеси.
    - Реалізовано QuestionGenerationService для обробки цього процесу.

Код сервісу:

```
class QuestionGenerationService:

    def generate(self, text, num_questions, question_type, difficulty):

        prompt = self.create_prompt(text, num_questions, question_type, difficulty)
```

```
        response = openai.Completion.create(
            engine='text-davinci-003',
            prompt=prompt,
            max_tokens=1500,
            temperature=0.7
        )
        questions = self.parse_response(response.choices[0].text)
        return questions

    def create_prompt(self, text, num_questions, question_type, difficulty):
        return f"Based on the following text, create {num_questions} {difficulty} {question_type} questions..."

    def parse_response(self, response_text):
        # Logic for parsing the response text into a question structure
        pass
```

5. Обробка та збереження питань:
    - Парситься відповідь від моделі, виділяючи питання, варіанти відповідей та правильні відповіді.
    - Використовуються регулярні вирази та шаблони для точного виділення інформації.
    - Питання зберігаються в базі даних з прив'язкою до розділів.
6. Перевірка якості питань:
    - Реалізовано автоматичні перевірки на унікальність питань та відсутність дублювання.
    - Планується додати модерацію питань або можливість редагування користувачем.

Вирішені задачі:

- Неточності в генерації: шляхом коригування промптів та додавання інструкцій для моделі вдалося покращити відповідність питань тексту.
- Форматування відповідей: створено універсальні шаблони для розпізнавання структури питань, що вирішило проблему різних варіацій форматування у відповідях моделі.

3.1.4 Інші ключові алгоритми

Аутентифікація та авторизація

Для забезпечення безпеки та контролю доступу реалізовано систему аутентифікації та авторизації з використанням JWT-токенів.

1. Реєстрація користувача:
    - Користувач надає необхідні дані (email, пароль).
    - Пароль хешується за допомогою алгоритму PBKDF2 з сіллю.
    - Валідація даних здійснюється на рівні серіалізаторів.
2. Вхід у систему:
    - Після перевірки облікових даних користувач отримує JWT-токен.
    - Токен містить інформацію про користувача та строк дії.

Приклад використання бібліотеки Simple JWT:

```
from rest_framework_simplejwt.tokens import RefreshToken


def get_tokens_for_user(user):
    refresh = RefreshToken.for_user(user)
    return {
        'refresh': str(refresh),
        'access': str(refresh.access_token),
    }
```

3. Авторизація запитів:
    - Для захищених маршрутів використовуються пермишени IsAuthenticated.
    - Токен передається в заголовку Authorization кожного запиту.

Обмеження для безкоштовних користувачів

Для реалізації моделі монетизації введено обмеження для безкоштовних користувачів.

1. Визначення планів підписки:

- Створено модель SubscriptionPlan, яка містить параметри лімітів.
- Користувач має поле subscription_plan, яке вказує на його поточний план.

2. Перевірка лімітів:
   - Перед виконанням критичних операцій (генерація питань, завантаження книг) перевіряється, чи не перевищено ліміти.
   - Реалізовано декоратори та middleware для автоматизації цього процесу.

Приклад декоратора:

```
def check_limit(limit_name):
    def decorator(func):
        def wrapper(request, *args, **kwargs):
            user = request.user
            if user.get_limit_remaining(limit_name) <= 0:
                return Response({'detail': 'Limit exceeded'}, status=403)
            return func(request, *args, **kwargs)
        return wrapper
    return decorator
```

3. Оновлення лімітів:
   - Після успішної операції оновлюються лічильники використання.
   - Лічильники зберігаються в моделі користувача або окремій моделі.

Обробка запитів до API

1. Стандартизація відповідей:
   - Встановлено єдиний формат для успішних та помилкових відповідей.
   - Використовується кастомний мідлвер для обробки винятків та формування помилкових відповідей.
2. Логування та моніторинг:
   - Логуються всі критичні події та помилки.
   - Використовується Sentry для відстеження винятків.

3.2 Розробка компонентів системи

3.2.1 Моделі даних

Моделювання даних є фундаментальним для будь-якої системи. Особлива увага приділена нормалізації даних та встановленню правильних зв'язків між моделями.

Приклад моделі Book:

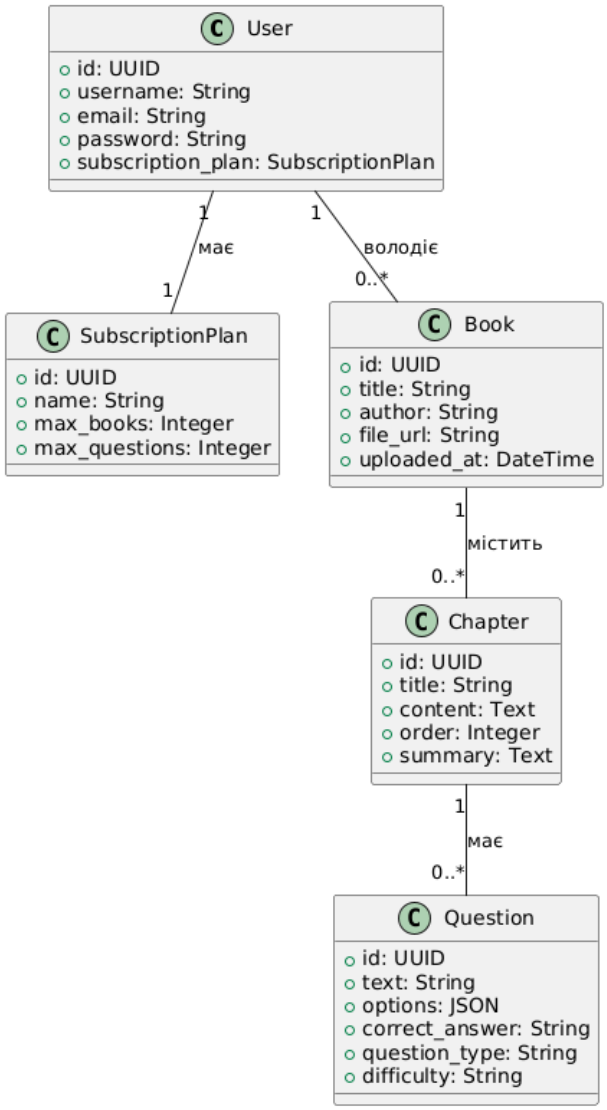
```
from django.db import models

from django.contrib.auth import get_user_model


User = get_user_model()


class Book(models.Model):

    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)

    user = models.ForeignKey(User, on_delete=models.CASCADE,
related_name='books')

    title = models.CharField(max_length=255)

    author = models.CharField(max_length=255, blank=True)

    file_url = models.URLField()

    uploaded_at = models.DateTimeField(auto_now_add=True)



    def __str__(self):

        return self.title
```

Особливості реалізації:

- Використання UUIDField для унікальних ідентифікаторів.
- Зв'язки ForeignKey з каскадним видаленням для підтримки цілісності даних.

- Додаткові методи моделі для бізнес-логіки (наприклад, обчислення кількості розділів).

Діаграма класів моделей даних:



3.2.2 Серіалізатори та фільтри

Серіалізатори

- Забезпечують валідацію вхідних даних та перетворення моделей у JSON.

- Використовується ModelSerializer для автоматичного генерування полів.

Приклад серіалізатора для Question:

```
from rest_framework import serializers

from .models import Question


class QuestionSerializer(serializers.ModelSerializer):

    class Meta:

        model = Question

        fields = ['id', 'text', 'options', 'correct_answer', 'question_type',
'difficulty']
```

Фільтри

- Дозволяють користувачу отримувати лише потрібні дані.
- Реалізовано фільтрацію за датою створення, автором тощо.

Приклад використання фільтрів:

```
from django_filters.rest_framework import DjangoFilterBackend


class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer_class = BookSerializer

    filter_backends = [DjangoFilterBackend]

    filterset_fields = ['uploaded_at', 'author']
```

3.2.3 ViewSets та маршрутизація

ViewSets

- Об'єднують логіку для набору пов'язаних представлень.
- Використовується ModelViewSet, який надає реалізацію CRUD-операцій.

Приклад ViewSet для Chapter:

```
class ChapterViewSet(viewsets.ModelViewSet):
    queryset = Chapter.objects.all()
    serializer_class = ChapterSerializer
    permission_classes = [IsAuthenticated]

    def get_queryset(self):
        return self.queryset.filter(book__user=self.request.user)
```

Маршрутизація

- Встановлено версіонування API для майбутнього розширення.
- Використовується DefaultRouter для автоматичної генерації маршрутів.

Приклад маршрутизації:

```
from rest_framework.routers import DefaultRouter
from .views import BookViewSet, ChapterViewSet

router = DefaultRouter()
router.register(r'books', BookViewSet)
router.register(r'chapters', ChapterViewSet)
urlpatterns = [
    path('api/v1/', include(router.urls)),
]
```

3.2.4 Сервіси та утиліти

Сервіси

- Виносять бізнес-логіку з представлень та моделей для кращої організації коду.
- Наприклад, FileService для роботи з файлами, NotificationService для оповіщень.

Утиліти

- Містять допоміжні функції, які використовуються в різних частинах проекту.
- Наприклад, функція для генерації випадкових рядків, перетворення форматів дат тощо.

Приклад утиліти для генерації випадкового рядка:

```
import random
import string


def generate_random_string(length=12):
    letters = string.ascii_letters + string.digits
    return ''.join(random.choice(letters) for i in range(length))
```

3.3 Реалізація взаємодії з зовнішніми сервісами

3.3.1 Інтеграція з **OpenAI** [19]

Технічні аспекти:

- Використовуються асинхронні запити з використанням бібліотеки httpx для підвищення продуктивності.

- Реалізовано обмеження швидкості запитів, щоб уникнути перевищення квот OpenAI.

Обробка помилок:

- Враховуються можливі помилки, такі як RateLimitError, TimeoutError.
- Реалізовано механізм повторних спроб із затримкою.

Приклад асинхронного запиту:

```
import httpx


async def generate_text(prompt):
    async with httpx.AsyncClient() as client:
        response = await client.post(
            'https://api.openai.com/v1/completions',
            headers={'Authorization': f'Bearer {OPENAI_API_KEY}'},
            json={
                'model': 'text-davinci-003',
                'prompt': prompt,
                'max_tokens': 1000
            }
        )
    return response.json()
```

3.3.2 Асинхронні завдання з Celery [10]

Налаштування:

- Використовується Redis як брокер та бекенд для зберігання стану завдань.
- Налаштовано пули воркерів для обробки великої кількості завдань.

Моніторинг:

- Використовується Flower для моніторингу стану завдань Celery.
- Налаштовано алерти на випадок накопичення невиконаних завдань.

Приклад завдання Celery:

```
from celery import shared_task


@shared_task
def process_uploaded_file(file_key, user_id):
    # Logic for processing the uploaded file
    pass
```

3.3.3 Збереження та обробка файлів

AWS S3 [15]

- Використовується бібліотека django-storages для інтеграції з S3.
- Налаштовано політику безпеки S3, щоб обмежити доступ до файлів лише автентифікованим користувачам.

Локальне зберігання (для розробки):

- Використовується локальна файлова система під час розробки для зручності.
- Налаштовано умови в settings.py, щоб змінювати бекенд зберігання в залежності від середовища.

Приклад налаштувань:

```
import os


if os.environ.get('ENV') == 'production':
```

```
    DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
else:
    DEFAULT_FILE_STORAGE = 'django.core.files.storage.FileSystemStorage'
```

3.3.4 Моніторинг та логування

Sentry [16]

- Інтеграція з Sentry дозволяє отримувати детальну інформацію про помилки, включаючи стек трейси та дані запиту.
- Встановлено рівні логування, щоб відокремити критичні помилки від менш важливих.

Slack [18]

- Використовуються веб-хуки для надсилання повідомлень про критичні події в спеціальний канал.
- Налаштовано формат повідомлень для зручного читання та швидкого реагування.

Логування

- Використовується стандартна бібліотека logging.
- Налаштовано різні хендлери для запису логів у файл та виводу в консоль.

Приклад конфігурації логування:

```
LOGGING = {
    'version': 1,
    'handlers': {
        'file': {
            'level': 'INFO',
            'class': 'logging.FileHandler',
            'filename': 'exam_maker.log',
```

```
        },
    },
    'loggers': {
        'exam_maker': {
            'handlers': ['file'],
            'level': 'INFO',
            'propagate': True,
        },
    },
}
```

3.4 Забезпечення безпеки та захисту даних

3.4.1 Реалізація механізмів аутентифікації та авторизації

Використання JWT та OAuth 2.0

У системі ExamMaker.ai безпека користувачів та їхніх даних є пріоритетом. Для забезпечення надійної аутентифікації та авторизації реалізовано наступні механізми:

1. JSON Web Tokens (JWT):
    - Призначення: Забезпечення безпечної передачі інформації про автентифікацію між клієнтом та сервером.
    - Реалізація:
        - При успішному вході користувача система генерує JWT-токен, який містить зашифровану інформацію про користувача та строк дії токена.
        - Токен передається клієнту та зберігається у локальному сховищі або cookies.
        - Для захищених API-запитів клієнт надсилає токен у заголовку Authorization: Bearer <token>.

- Переваги:
  - Зменшує навантаження на сервер, оскільки не потребує зберігання стану сесії.
  - Легко масштабується для мікросервісної архітектури.

Приклад генерації JWT-токена:

```
from rest_framework_simplejwt.tokens import RefreshToken


def get_tokens_for_user(user):
    refresh = RefreshToken.for_user(user)
    return {
        'refresh': str(refresh),
        'access': str(refresh.access_token),
    }
```

2. OAuth 2.0 з інтеграцією Google Sign-In:
   - Призначення: Спрощення процесу реєстрації та входу для користувачів шляхом використання їхніх облікових записів Google.
   - Реалізація:
     - Використано бібліотеку django-allauth для підтримки OAuth 2.0.
     - Користувач може увійти або зареєструватися в системі, використовуючи свій Google-акаунт.
     - Після успішної автентифікації з Google система отримує основні дані користувача (ім'я, електронну пошту) та створює локальний обліковий запис.

Налаштування OAuth 2.0 в settings.py:

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
```

```
)


INSTALLED_APPS = [
    # ...
    'django.contrib.sites',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'allauth.socialaccount.providers.google',
    # ...
]


SOCIALACCOUNT_PROVIDERS = {
    'google': {
        'SCOPE': [
            'profile',
            'email',
        ],
        'AUTH_PARAMS': {
            'access_type': 'online',
        },
    }
}
```

Захист від атак на аутентифікацію та авторизацію:

- Захист від підробки токенів: Використовуються секретні ключі та алгоритми шифрування для підпису токенів.
- Контроль строку дії токенів: Встановлено час життя токенів доступу та оновлення для зменшення ризику компрометації.
- Ревокація токенів: Можливість анулювання токенів у разі підозрілої активності.

3.4.2 Захист API

Обмеження доступу та перевірка прав користувачів

Для забезпечення безпеки API та контролю доступу до ресурсів реалізовано наступні механізми:

1. Пермишени (Permissions):
    - Використовуються вбудовані та кастомні пермишени Django REST Framework для контролю доступу.
    - Приклади пермишенів:
        - IsAuthenticated: Доступ тільки для автентифікованих користувачів.
        - IsAdminUser: Доступ тільки для адміністраторів.
        - IsOwnerOrReadOnly: Користувач може редагувати тільки свої ресурси.

Приклад використання пермишенів у ViewSet:

```
from rest_framework.permissions import IsAuthenticated

from .permissions import IsOwnerOrReadOnly


class BookViewSet(viewsets.ModelViewSet):

    queryset = Book.objects.all()

    serializer_class = BookSerializer

    permission_classes = [IsAuthenticated, IsOwnerOrReadOnly]
```

2. Обмеження за ролями:
    - Реалізовано систему ролей (user, admin) для розмежування прав доступу.
    - Встановлено спеціальні декоратори та middleware для перевірки ролі користувача при доступі до певних маршрутів.
3. Обмеження швидкості запитів (Rate Limiting):
    - Використовується для запобігання DDoS-атакам та зловживанню API.

- Реалізовано за допомогою Throttle класів у Django REST Framework.

Приклад налаштування обмеження швидкості:

```
from rest_framework.throttling import UserRateThrottle


class BurstRateThrottle(UserRateThrottle):
    rate = '60/min'


class SustainedRateThrottle(UserRateThrottle):
    rate = '1000/day'


class MyAPIView(APIView):
    throttle_classes = [BurstRateThrottle, SustainedRateThrottle]
```

3.4.3 Шифрування та зберігання даних

Методи захисту конфіденційної інформації

1. Захищене з'єднання (HTTPS):
   - Всі дані передаються через захищений протокол HTTPS з використанням SSL/TLS сертифікатів.
   - Це забезпечує шифрування даних між клієнтом та сервером, запобігаючи перехопленню інформації.
2. Зберігання паролів:
   - Паролі користувачів хешуються з використанням алгоритму PBKDF2 з сіллю, що є стандартом у Django.
   - Це ускладнює можливість розшифрування паролів у разі компрометації бази даних.
3. Захист конфіденційних даних:
   - Критичні змінні оточення (секретні ключі, API-ключі) зберігаються у безпечному сховищі, наприклад, у файлі .env, який не додається до репозиторію.

- Використовується бібліотека python-decouple для завантаження змінних оточення.

Приклад використання python-decouple:

```
from decouple import config


SECRET_KEY = config('SECRET_KEY')

DEBUG = config('DEBUG', default=False, cast=bool)
```

4. Шифрування даних у базі даних:
    - Для особливо чутливих даних може використовуватися додаткове шифрування на рівні полів моделей.
    - Використовується бібліотека django-encrypted-model-fields для прозорого шифрування даних.

Приклад шифрованого поля:

```
from encrypted_model_fields.fields import EncryptedCharField


class UserProfile(models.Model):

    user = models.OneToOneField(User, on_delete=models.CASCADE)

    ssn = EncryptedCharField(max_length=11)
```

3.4.4 Обробка помилок та виключень

Як система реагує на нештатні ситуації

1. Уніфікована обробка виключень:
    - Реалізовано глобальний обробник помилок, який перехоплює винятки та формує уніфіковану відповідь.

- Використовується middleware ExceptionMiddleware для обробки непередбачених помилок.

Приклад middleware :

```
class ExceptionMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        try:
            response = self.get_response(request)
        except Exception as e:
            # Логування помилки
            logger.error(f"Unhandled exception: {str(e)}")
            # Формування стандартної відповіді
            return JsonResponse({'detail': 'Internal server error'},
status=500)
        return response
```

2. Логування помилок:
   - Всі помилки та винятки логуються з деталями, включаючи стек викликів.
   - Використовується Sentry для збору та аналізу помилок у реальному часі.
3. Оповіщення про критичні події:
   - У разі виникнення критичних помилок система надсилає оповіщення команді розробників через Slack.
   - Це дозволяє швидко реагувати на проблеми та мінімізувати час простою системи.
4. Користувацькі повідомлення про помилки:
   - Користувач отримує зрозумілі повідомлення про помилки без розкриття внутрішньої інформації системи.
   - Наприклад, при недоступності сервісу користувач побачить повідомлення: "Наразі сервіс недоступний, спробуйте пізніше".

Переваги реалізованого підходу:

- Підвищення стабільності системи за рахунок проактивного виявлення та усунення помилок.
- Покращення користувацького досвіду через надання зрозумілих повідомлень.
- Забезпечення безпеки шляхом приховування внутрішніх деталей реалізації від користувачів.

3.5 Тестування та відлагодження системи

3.5.1 Модульне тестування

Написання тестів для окремих компонентів

1. Вибір фреймворку для тестування:
    - Використовується вбудований у Django фреймворк тестування разом із бібліотекою pytest для зручності написання тестів.
    - pytest-django дозволяє інтегруватися з Django та використовувати його можливості.
2. Структура тестів:
    - Тести розміщуються в директоріях tests всередині кожного додатку.
    - Назви файлів та функцій тестів починаються з test_ для автоматичного виявлення.
3. Приклади модульних тестів:

Тест моделей: Перевірка коректності збереження та отримання даних з бази.

```
def test_book_creation(db):
    user = User.objects.create_user(username='testuser', password='testpass')
    book = Book.objects.create(title='Test Book', user=user)
    assert book.title == 'Test Book'
    assert book.user == user
```

```

Тест серіалізаторів: Валідація вхідних даних та коректність серіалізації.

```
def test_book_serializer():
    data = {'title': 'Test Book', 'author': 'Author Name'}
    serializer = BookSerializer(data=data)
    assert serializer.is_valid()
```

Тест представлень (views): Перевірка HTTP-відповідей та статусів.

```
def test_book_list_view(api_client, db):
    response = api_client.get('/api/v1/books/')
    assert response.status_code == 200
```

3.5.2 Інтеграційне тестування

Перевірка взаємодії між компонентами

1. Мета інтеграційного тестування:
    - Переконатися, що окремі модулі системи коректно взаємодіють між собою.
    - Виявити помилки, які можуть виникнути на стиках компонентів.
2. Реалізація тестів:
    - Тести, які перевіряють повний цикл роботи функціоналу.
    - Використовуються фікстури для налаштування тестового середовища (наприклад, створення тестових користувачів, налаштування бази даних).

Приклад інтеграційного тесту для генерації питань:

```
def test_question_generation_flow(api_client, db):
```
```

```
    # Create a user and log in

    user = User.objects.create_user(username='testuser', password='testpass')

    api_client.force_authenticate(user=user)

    # Upload a book

    response = api_client.post('/api/v1/books/', {'title': 'Test Book',
'file': SimpleUploadedFile('test.pdf', b'file_content')})

    assert response.status_code == 201

    book_id = response.data['id']

    # Initiate question generation

    response =
api_client.post(f'/api/v1/books/{book_id}/generate_questions/',
{'num_questions': 5})

    assert response.status_code == 200

    # Verify that the questions are generated

    # …
```

3.5.3 Навантажувальне тестування

Оцінка продуктивності системи під час пікових навантажень

1. Мета навантажувального тестування:
    - Визначити, як система поводиться під великим навантаженням.
    - Виявити "вузькі місця" та оптимізувати продуктивність.
2. Інструменти для тестування:
    - Locust: Інструмент для навантажувального тестування веб-додатків.
    - JMeter: Потужний інструмент для моделювання навантаження та аналізу продуктивності.
3. Сценарії тестування:
    - Моделювання одночасного завантаження великої кількості користувачів.
    - Тестування швидкості генерації питань при великій кількості запитів.
4. Результати та оптимізація:

- Виявлено, що при великій кількості одночасних запитів до OpenAI API виникають затримки.
- Реалізовано чергу запитів та обмеження на кількість одночасних завдань Celery.
- Кешування результатів для популярних запитів з метою зменшення навантаження.

3.5.4 Виправлення помилок та оптимізація

Процес ідентифікації та усунення недоліків

1. Використання систем моніторингу:
   - Sentry: Для відстеження помилок та винятків у реальному часі.
   - New Relic: Для моніторингу продуктивності додатку та серверів.
2. Аналіз логів:
   - Регулярний перегляд логів для виявлення повторюваних помилок.
   - Налаштування ротації логів та їх архівування.
3. Процес виправлення помилок:
   - Створення тасків у системі управління проєктами (наприклад, Jira) при виявленні помилок.
   - Пріоритизація задач за критичністю.
   - Проведення код-рев'ю перед злиттям виправлень у основну гілку.
4. Оптимізація коду та запитів:
   - Профілювання коду для виявлення "важких" місць.
   - Оптимізація запитів до бази даних (наприклад, використання select_related, prefetch_related).
   - Рефакторинг коду для підвищення його читабельності та ефективності.

Результати оптимізації:

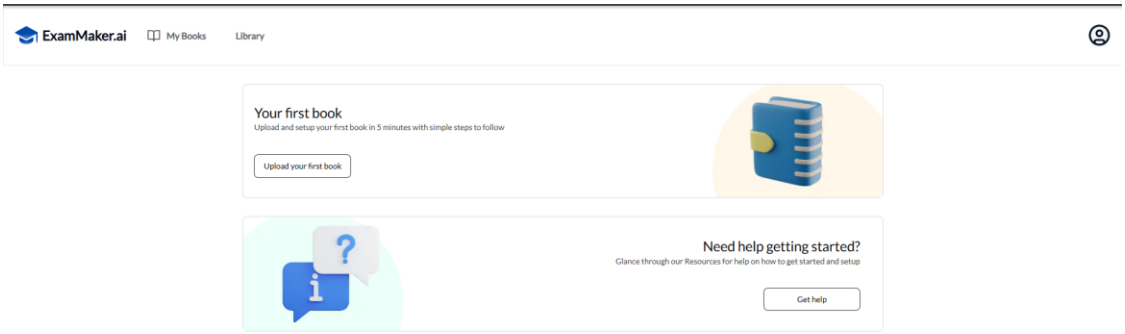
- Зменшено час відповіді API на 30% за рахунок оптимізації запитів до бази даних.
- Знижено використання пам'яті на сервері шляхом звільнення не використовуваних ресурсів.
- Підвищено стійкість системи до пікових навантажень.

3.6 Інструкція користувача та демонстрація роботи

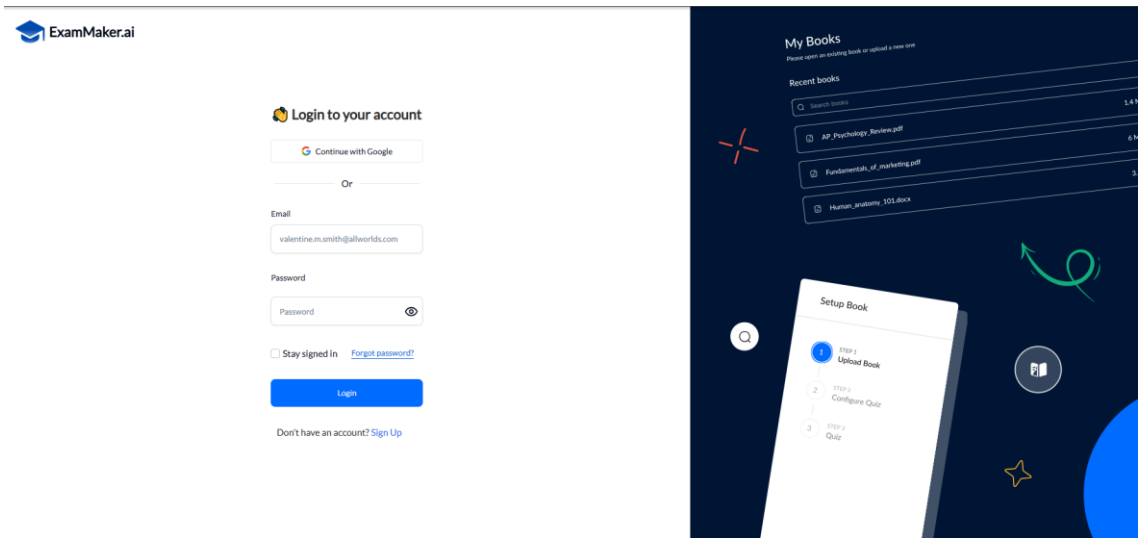
3.6.1 Скріншоти інтерфейсу

Візуальне представлення основних функцій системи

1. Головна сторінка:

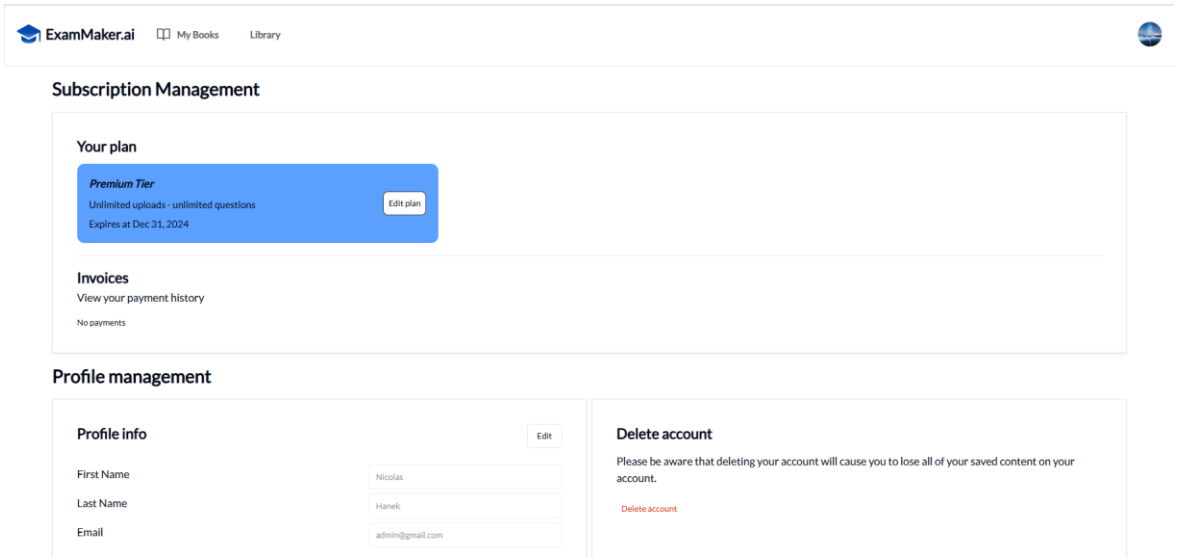


   - Вітальний екран із коротким описом можливостей платформи.
   - Кнопки для реєстрації та входу.
2. Сторінка реєстрації та входу:

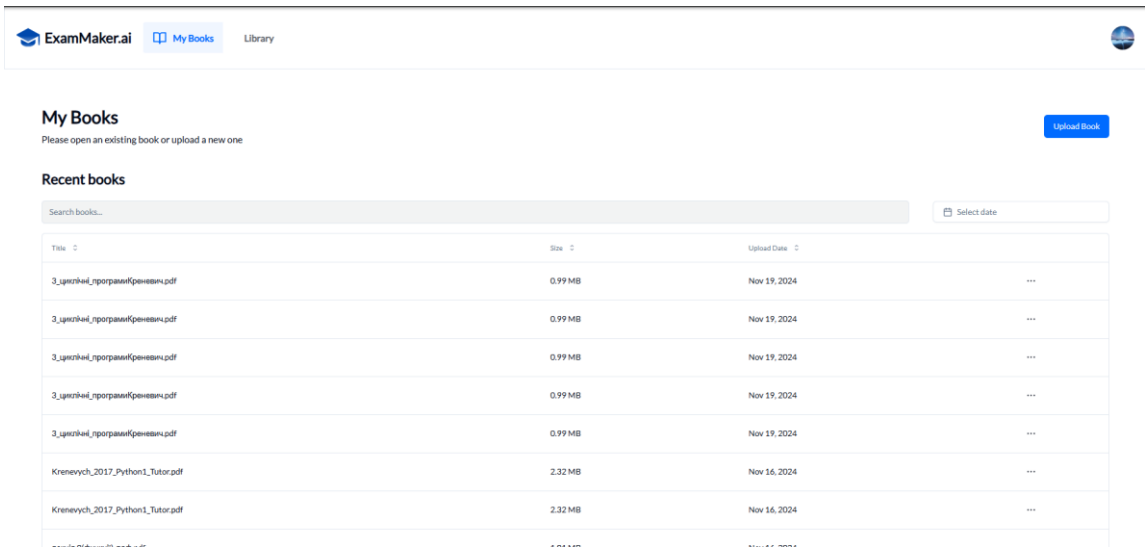


- Форма для введення електронної пошти та пароля.
- Кнопка для входу через Google.

3. Сторінка профілю користувача:

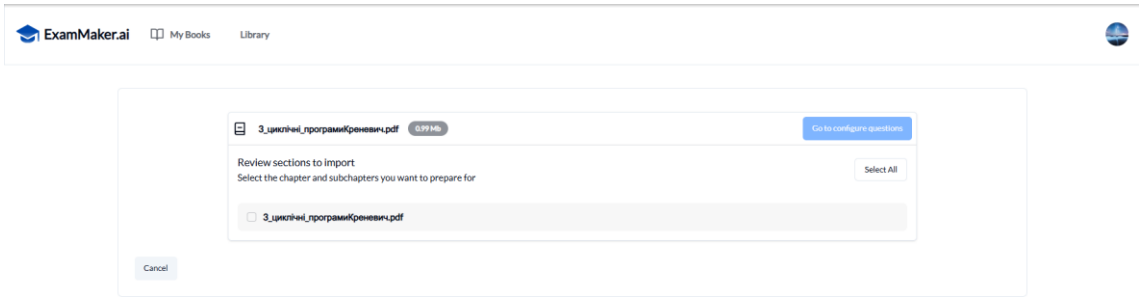


4. Сторінка завантаження книги:



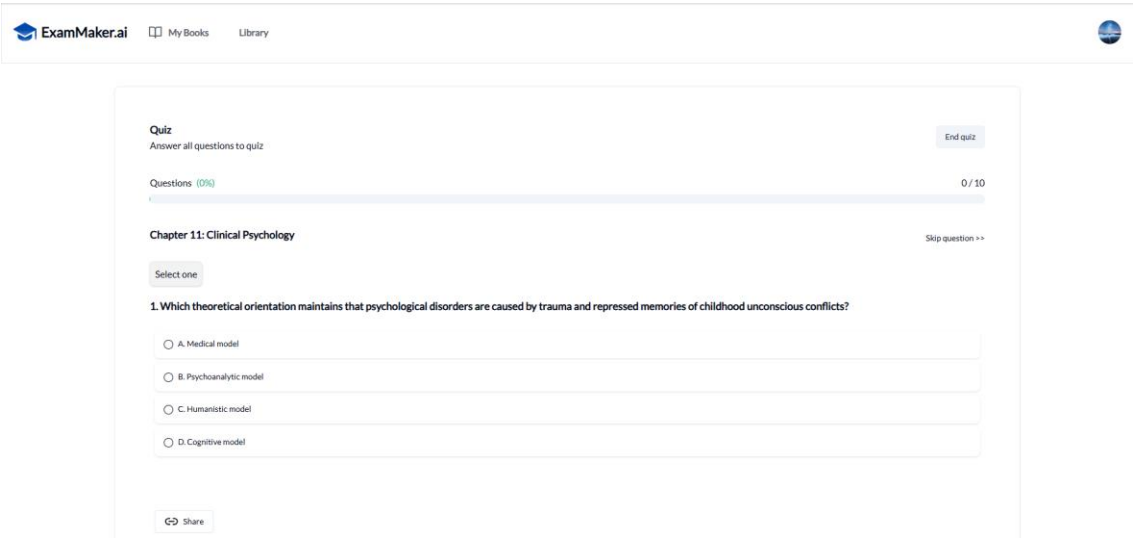
- Форма для завантаження файлу та введення назви книги.
- Інформація про підтримувані формати та обмеження розміру файлу.

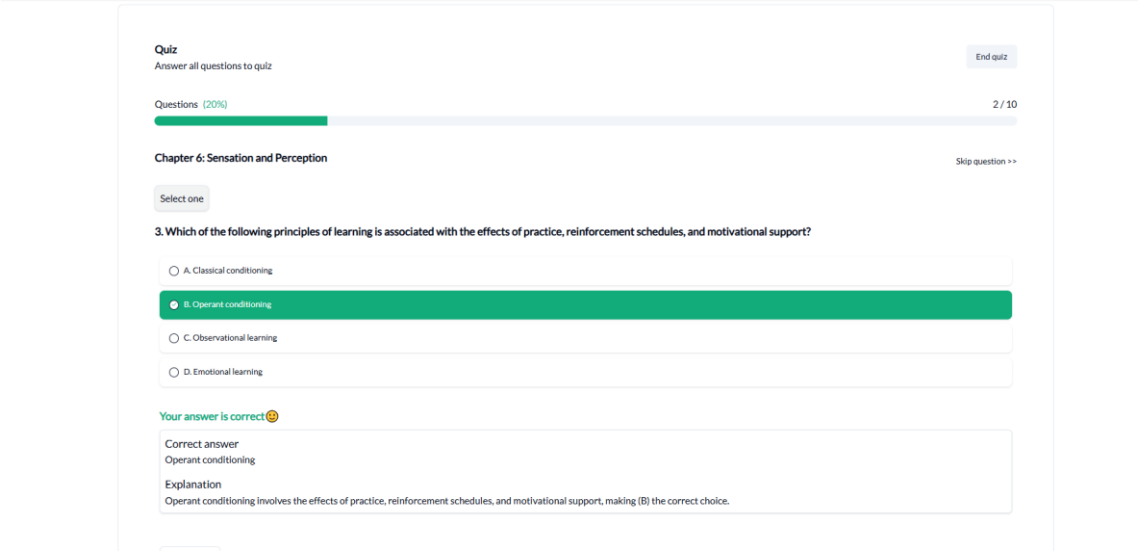
5. Сторінка генерації питань:



- Вибір книги та розділів для генерації питань.
- Налаштування параметрів (кількість питань, типи питань, рівень складності).

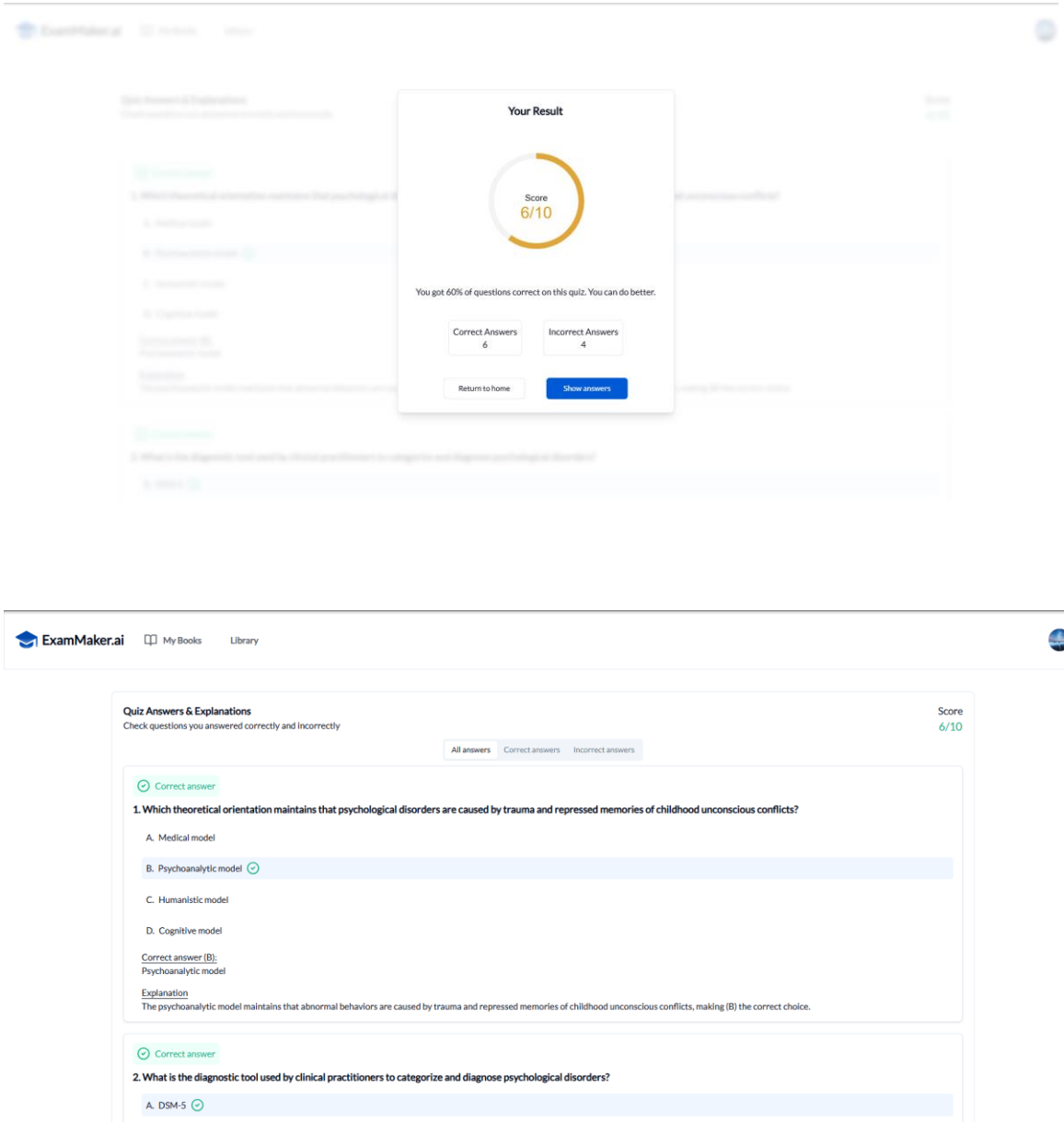
6. Сторінка тестування:





- Інтерфейс для проходження тесту з відображенням питань та варіантів відповідей.
- Кнопки для переходу між питаннями та завершення тесту.

7. Сторінка результатів тестування:



- Підсумковий бал та відсоток правильних відповідей.
- Детальний аналіз помилок з можливістю перегляду правильних відповідей.

3.6.2 Керівництво користувача

Покрокова інструкція з використання платформи

1. Реєстрація в системі:
   - Перейдіть на головну сторінку та натисніть "Реєстрація".

- Заповніть форму, вказавши електронну пошту та пароль.
- Або скористайтеся кнопкою "Увійти через Google" для швидкої реєстрації.

2. Вхід у систему:
   - Введіть свої облікові дані на сторінці входу.
   - Якщо ви забули пароль, скористайтеся функцією відновлення паролю.
3. Завантаження навчального матеріалу:
   - Після входу перейдіть до панелі управління та натисніть "Додати книгу".
   - Виберіть файл на своєму комп'ютері та введіть назву книги.
   - Натисніть "Завантажити" та дочекайтеся повідомлення про успішне завантаження.
4. Генерація екзаменаційних питань:
   - У списку книг натисніть на книгу, з якої бажаєте згенерувати питання.
   - Виберіть розділи або глави, що вас цікавлять.
   - Налаштуйте параметри генерації (кількість питань, типи, складність).
   - Натисніть "Згенерувати питання" та зачекайте завершення процесу.
5. Проходження тесту:
   - Після генерації питань натисніть "Розпочати тест".
   - Відповідайте на питання, обираючи варіанти відповідей.
   - Після завершення тесту натисніть "Завершити" для перегляду результатів.
6. Перегляд результатів та аналіз:
   - На сторінці результатів ознайомтеся зі своїм балом.
   - Перегляньте правильні відповіді та пояснення до них.
   - Ви можете повторити тест або згенерувати новий.
7. Управління профілем:
   - У налаштуваннях профілю ви можете змінити свої дані, пароль або переглянути підписку.
   - Також доступна історія ваших тестів та згенерованих питань.

3.6.3 Приклади використання

Реальні сценарії роботи з системою

1. Студент готується до іспиту:
    - Завантажує конспект лекцій у форматі PDF.
    - Генерує тестові питання з вибором відповіді для самоперевірки.
    - Проходить тест та аналізує результати, приділяючи увагу слабким місцям.
2. Викладач створює матеріали для занять:
    - Завантажує навчальний посібник у форматі DOCX.
    - Генерує питання відкритого типу для обговорення на семінарі.
    - Експортує питання у зручний формат для роздачі студентам.
3. Підготовка до сертифікаційного іспиту:
    - Користувач завантажує офіційну документацію або специфікації.
    - Обирає високий рівень складності питань.
    - Регулярно проходить тести, відслідковуючи свій прогрес.

Переваги використання ExamMaker.ai:

- Персоналізоване навчання: Користувач самостійно обирає матеріали та параметри генерації питань.
- Економія часу: Автоматична генерація питань дозволяє швидко підготуватися до тестування.
- Підвищення ефективності: Аналіз результатів допомагає виявити прогалини в знаннях та зосередитися на їх усуненні.

Висновки

У кваліфікаційної роботи розроблено веб-сервіс **ExamMaker.ai** для автоматичної генерації тестових питань на основі навчальних матеріалів з використанням технологій штучного інтелекту. Проведений аналіз існуючих аналогічних систем дозволив виявити їхні переваги та недоліки, що сприяло формуванню вимог до власного проекту та обґрунтуванню вибору технологій і методів реалізації.

Основні результати роботи:

- Створено багаторівневу архітектуру, що включає клієнтський інтерфейс, серверну частину, базу даних та інтеграцію з зовнішніми сервісами. Це забезпечило модульність, масштабованість та гнучкість системи.
- Реалізація ключових компонентів:
  - **Компонент "Book":** Забезпечує завантаження, збереження та обробку навчальних матеріалів у різних форматах.
  - **Компонент "Chapter":** Відповідає за структурування контенту та генерацію резюме розділів.
  - **Компонент "Question":** Реалізує генерацію екзаменаційних питань з використанням OpenAI GPT-4 та налаштування параметрів генерації.
  - **Компонент "Authentication":** Забезпечує безпечну автентифікацію та авторизацію користувачів з використанням JWT та OAuth 2.0.
- Реалізовано взаємодію з AWS S3 для зберігання файлів, OpenAI GPT-4 для генерації питань, Sentry та Slack для моніторингу та оповіщення.
- Впроваджено сучасні методи шифрування, захист від поширених веб-атак, система ролей та дозволів для контролю доступу.
- Проведено модульне, інтеграційне та навантажувальне тестування, виявлено та виправлено помилки, оптимізовано продуктивність та стабільність роботи сервісу.

Теоретичне та практичне значення роботи:

- Інноваційність підходу: Поєднання технологій штучного інтелекту та веб-розробки для автоматизації створення тестових завдань є актуальним напрямком у галузі освітніх технологій.

- Практична цінність: Розроблений сервіс може суттєво спростити процес підготовки тестів для викладачів, тренерів та інших фахівців, а також підвищити ефективність самостійної підготовки студентів.
- Внесок у розвиток освітніх технологій: Система сприяє впровадженню нових методів навчання та оцінювання знань, що відповідає сучасним тенденціям цифровізації освіти.

Перспективи подальшого розвитку:

- Розширення функціональності:
    - Додавання підтримки нових форматів файлів та обробки графічного контенту.
    - Розробка інструментів для спільної роботи над тестами та обміну матеріалами між користувачами.
- Покращення алгоритмів генерації:
    - Впровадження найновітніших моделей штучного інтелекту для підвищення якості та релевантності згенерованих питань.
    - Розробка адаптивних алгоритмів, що враховують індивідуальний рівень знань користувача.
- Інтеграція з освітніми платформами:
    - Створення модулів для інтеграції з системами управління навчанням (LMS) та іншими освітніми сервісами.
- Мобільні рішення:
    - Розробка мобільного додатку для забезпечення доступу до сервісу з різних пристроїв та підвищення зручності використання.

Загальний висновок

Поставлені в роботі цілі були досягнуті. Розроблено ефективний інструмент для автоматичної генерації тестових питань, що має широкий спектр застосувань у сфері освіти та професійного навчання. Система ExamMaker.ai відзначається високою функціональністю, безпекою та зручністю використання, що робить її перспективним рішенням на ринку освітніх технологій. Подальший розвиток проекту та впровадження нових можливостей сприятимуть його популяризації та корисності для широкого кола користувачів.

Список літератури

1. Karpicke JD, Roediger HL III (2006). Testing Effect: A Review of Research on the Benefits of Retrieval Practice for Learning. [Електронний ресурс] Режим доступу: https://doi.org/10.1111/j.1467-9280.2006.01693.x
2. Lyle KB, Crawford NA (2011). Enhanced Learning Through Testing: Evidence from Quizzing in a Classroom Setting [Електронний ресурс] Режим доступу: https://doi.org/10.1080/00220973.2010.542911
3. QuizGecko – Інструмент для автоматизованого створення тестів на основі тексту та документів з використанням штучного інтелекту. [Електронний ресурс] Режим доступу: https://quizgecko.com/
4. TestPortal – Платформа для створення онлайн-тестів, квізів і оцінювання знань у навчальних і корпоративних середовищах. [Електронний ресурс] Режим доступу: https://www.testportal.net/
5. Revisely – Сервіс для автоматизованої оцінки письмових робіт і створення навчальних завдань для освітніх установ. [Електронний ресурс] Режим доступу: https://www.revisely.com/
6. PrepAI – Платформа для генерації тестових завдань та іспитів за допомогою штучного інтелекту, орієнтована на освітні потреби. [Електронний ресурс] Режим доступу: https://www.prepai.io/eu/
7. Django Documentation – Офіційна документація для Django, що охоплює розробку, налаштування та оптимізацію веб-застосунків. [Електронний ресурс] Режим доступу до ресурсу: https://docs.djangoproject.com/en/stable/
8. Django Rest Framework Documentation – Офіційна документація для Django Rest Framework з прикладами створення API. [Електронний ресурс] Режим доступу до ресурсу: https://www.django-rest-framework.org/
9. Next.js Documentation – Офіційна документація для Next.js, фреймворку для серверного рендерингу з використанням React. [Електронний ресурс] Режим доступу до ресурсу: https://nextjs.org/docs
10. Celery Documentation – Офіційна документація для Celery, системи для розподіленої черги завдань. [Електронний ресурс] Режим доступу до ресурсу: https://docs.celeryproject.org/en/stable/
11. Gunicorn Documentation – Офіційна документація для Gunicorn, WSGI HTTP сервера для Python у UNIX-системах. [Електронний ресурс] Режим доступу до ресурсу: https://gunicorn.org/
12. Nginx Documentation – Офіційна документація для Nginx, високопродуктивного HTTP сервера та зворотного проксі. [Електронний ресурс] Режим доступу до ресурсу: https://nginx.org/

13. PostgreSQL Documentation – Офіційна документація для PostgreSQL, реляційної бази даних з відкритим кодом. [Електронний ресурс] Режим доступу до ресурсу: https://www.postgresql.org/docs/
14. Redis Documentation – Офіційна документація для Redis, високошвидкісної бази даних типу key-value, яка використовується для кешування та обробки даних у реальному часі. [Електронний ресурс] Режим доступу до ресурсу: https://redis.io/documentation
15. AWS S3 Documentation – Офіційна документація для AWS S3, хмарного сховища для об'єктів. [Електронний ресурс] Режим доступу до ресурсу: https://aws.amazon.com/s3/
16. Sentry SDK Documentation – Офіційна документація для Sentry SDK, що допомагає відстежувати помилки в реальному часі. [Електронний ресурс] Режим доступу до ресурсу: https://docs.sentry.io/platforms/python/guides/django/
17. Hotjar Documentation – Офіційна документація для Hotjar, інструменту для аналізу користувацької взаємодії через теплові карти. [Електронний ресурс] Режим доступу до ресурсу: https://help.hotjar.com/hc/en-us
18. Slack SDK Documentation – Офіційна документація для Slack SDK, що використовується для створення інтеграцій та ботів у Slack. [Електронний ресурс] Режим доступу до ресурсу: https://api.slack.com/sdk
19. OpenAI API Documentation – Офіційна документація для API OpenAI, що надає доступ до моделей машинного навчання. [Електронний ресурс] Режим доступу до ресурсу: https://platform.openai.com/docs/
20. Docker Documentation – Офіційна документація для Docker, що охоплює використання контейнерів, їх налаштування та управління. [Електронний ресурс] Режим доступу: https://docs.docker.com/
21. unstructured Documentation – Офіційна документація для unstructured, бібліотеки для обробки неструктурованих даних. [Електронний ресурс] Режим доступу до ресурсу: https://github.com/Unstructured-IO/unstructured

Додаток

1. Компонент "chapter"
1.1 Моделі Chapter, SubChapter та BookChunk
Файл: chapter/models.py
```
from django.db import models

from base.models import CreatedUpdatedAt

class Chapter(CreatedUpdatedAt):
    book = models.ForeignKey("book.Book", on_delete=models.CASCADE)
    name = models.CharField(max_length=256)
    description = models.TextField(blank=True)
    page_start = models.PositiveIntegerField()
    page_end = models.PositiveIntegerField(null=True, blank=True)
    topic = models.ForeignKey("topic.Topic", on_delete=models.SET_NULL, null=True, blank=True)
    summary = models.TextField(blank=True, default="")

    class Meta:
        db_table = "chapters"
        ordering = ("book", "id")
        unique_together = (("book", "name"),)

    def __str__(self) -> str:
        return self.name

class SubChapter(CreatedUpdatedAt):
    chapter = models.ForeignKey("chapter.Chapter", on_delete=models.CASCADE)
    topic = models.ForeignKey("topic.Topic", on_delete=models.SET_NULL, null=True, blank=True)
    name = models.CharField(max_length=256)
    description = models.TextField(blank=True)
    page_start = models.PositiveIntegerField(null=True, blank=True)
    page_end = models.PositiveIntegerField(null=True, blank=True)
    summary = models.TextField(blank=True, default="")

    class Meta:
        db_table = "subchapters"
        ordering = ("chapter__book", "chapter", "id")
        unique_together = (("chapter", "name"),)

    def __str__(self) -> str:
        return self.name

class BookChunk(CreatedUpdatedAt):
    topic = models.ForeignKey("topic.Topic", on_delete=models.CASCADE, null=True, blank=True)
    type = models.CharField(max_length=255)
    content = models.TextField()
    subchapter = models.ForeignKey("chapter.SubChapter", on_delete=models.CASCADE)
    page_number = models.PositiveIntegerField()

    class Meta:
        db_table = "book_chunks"
        ordering = ("subchapter__chapter__book", "subchapter__chapter", "subchapter", "id")
```

1.2 ViewSets для Chapter, SubChapter та BookChunk
Файл: chapter/viewsets/chapter.py
```
from rest_framework.decorators import action
from rest_framework.request import Request
from rest_framework.response import Response
from rest_framework.viewsets import ModelViewSet

from base.services import TextSummaryService
from chapter.filters import ChaperFilterSet
from chapter.models import BookChunk, Chapter
from chapter.serializers import ChapterSerializer

class ChapterViewSet(ModelViewSet):
    queryset = Chapter.objects.all()
    serializer_class = ChapterSerializer
    filterset_class = ChaperFilterSet
    ordering_fields = "__all__"

    @action(methods=["GET"], detail=True)
    def summary(self, request: Request, pk: int, *args, **kwargs) -> Response:
        instance = self.get_object()
        filter_kwargs = {
            "subchapter__chapter__book_id": instance.book_id,
            "page_number__gte": instance.page_start,
            "page_number__lte": instance.page_end,
        }
        queryset = BookChunk.objects.all().filter(**filter_kwargs)
        text = "\n".join(content for content in queryset.values_list("content", flat=True))
        service = TextSummaryService()
        summary = service.summarize(text)
        instance.summary = summary
        instance.save()
        return Response(summary)
```

1.3 Сервіс TextSummaryService
Файл: base/services/text_summary.py
```
import json

from base.openai_.utils import get_gpt_completion_with_backoff

class TextSummaryService:
    PROMPT_TOKEN_LIMITATION = 4096
    RESPONSE_FORMAT = "Return summary as JSON object in the following format: {'summary': 'summary
text'}"
    TEMPLATE = (
        "Write a comprehensive and informative chapter summary of the provided text for "
        "book content. The summary should immediately capture the key points without starting with direct "
        "references to the text, such as 'the text describes...' or similar phrases. Ensure that the summary is clear "
        "and engaging while maintaining a neutral tone. Write at least three paragraphs."
        "Return the summary in the following format: {response_format}."
        "Text: {text}"
```

```
    )
    SYSTEM_PROMPT_TEMPLATE = (
        "You are a professional book chapter summarizer. "
        "You will be given a text and you will summarize it in a concise and informative manner."
    )

    def summarize(self, text: str) -> str:
        # Prepare prompt
        prompt = self._refine_prompt(text)
        system_prompt = self.SYSTEM_PROMPT_TEMPLATE
        # Make a request to Chat GPT to summarize text
        summary = self._send_request(system_prompt, prompt)
        summary_json = json.loads(summary)
        return summary_json["summary"]

    def _refine_prompt(self, text: str) -> str:
        prompt = self.TEMPLATE.format(response_format=self.RESPONSE_FORMAT, text=text[:
self.PROMPT_TOKEN_LIMITATION])
        return prompt

    def _send_request(self, system_prompt: str, prompt: str):
        return get_gpt_completion_with_backoff(system_prompt, prompt)
```

1.4 Завдання Celery для створення резюме
Файл: chapter/tasks.py
```
from concurrent.futures import ThreadPoolExecutor

from django.db.models import Prefetch

from base.services.text_summary import TextSummaryService
from book.models import Book
from chapter.models import BookChunk, Chapter, SubChapter
from config.celery import app

def summarize_subchapter(subchapter):
    queryset = subchapter.bookchunk_set.all()
    text = "\n".join(content for content in queryset.values_list("content", flat=True))
    service = TextSummaryService()
    summary = service.summarize(text)
    subchapter.summary = summary
    subchapter.save()

def summarize_chapter(chapter):
    filter_kwargs = {
        "subchapter__chapter__book_id": chapter.book_id,
        "page_number__gte": chapter.page_start,
        "page_number__lte": chapter.page_end,
    }
    queryset = BookChunk.objects.all().filter(**filter_kwargs)
    text = "\n".join(content for content in queryset.values_list("content", flat=True))
    service = TextSummaryService()
    summary = service.summarize(text)
    chapter.summary = summary
```

```
    chapter.save()

@app.task()
def create_chapters_and_subchapters_summary_task(book_id: int) -> None:
    book = Book.objects.prefetch_related(
        Prefetch(
            "chapter_set",
            queryset=Chapter.objects.prefetch_related(
                Prefetch("subchapter_set", queryset=SubChapter.objects.all(), to_attr="prefetched_subchapters")
            ),
            to_attr="prefetched_chapters",
        )
    ).get(id=book_id)
    all_subchapters = []
    for chapter in book.prefetched_chapters:
        subchapters = [
            subchapter for subchapter in chapter.prefetched_subchapters if len(chapter.prefetched_subchapters) > 1
        ]
        all_subchapters.extend(chapter.prefetched_subchapters)

    with ThreadPoolExecutor(max_workers=10) as executor:
        executor.map(summarize_subchapter, subchapters)

    with ThreadPoolExecutor(max_workers=10) as executor:
        executor.map(summarize_chapter, book.prefetched_chapters)
```

2. Компонент "book"
2.1 Сервіс BookService
Файл: book/services.py
```
import logging
from dataclasses import dataclass, field
from functools import partial
from io import BytesIO
from operator import attrgetter
from typing import Any, TypeVar

from django.core.files import File
from unstructured.partition.doc import partition_doc
from unstructured.partition.docx import partition_docx
from unstructured.partition.pdf import partition_pdf
from unstructured.partition.strategies import determine_pdf_or_image_strategy

from base.decorators import execute_method_on_condition, parse_elements_to_json
from base.utils.file import get_file_name_extension
from base.utils.partition import get_epub_partition
from book.enums import SupportedBookExtensions
from book.files import DocDocument, Document, DocxDocument, EpubDocument, PDFDocument
from book.models import Book
from book.utils import check_partition_skip
from chapter.models import BookChunk, Chapter, SubChapter

logger = logging.getLogger(__file__)
Topic = TypeVar("Topic")
```

```
@dataclass
class BookServiceSettings:
    chapter_level: int = field(default=1)
    subchapter_level: int = field(default=2)
    include_last_chapter: bool = field(default=False, repr=False)
    generate_chapters: bool = field(default=True)
    generate_subchapters: bool = field(default=True)
    generate_bookchunks: bool = field(default=True)

    def __post_init__(self) -> None:
        assert self.chapter_level > 0 and self.subchapter_level > 0, "Wrong chapter-subchapter setting!"
        assert self.chapter_level < self.subchapter_level, "Wrong chapter-subchapter setting!"

@dataclass
class BookService:
    book: Book
    document: Document | None = field(default=None)
    settings: BookServiceSettings = field(default_factory=BookServiceSettings)

    def create_full_book(self) -> None:
        self.create_table_of_content()
        self.create_bookchunks()

    def create_table_of_content(self) -> None:
        file = self.book.file.file
        _, extension = get_file_name_extension(file, include_dot=False)
        # Parse the document to get table of content
        self.document = self.get_document(file, extension)
        toc = self.document.get_toc()
        # Prepare raw TOC data to create chapters & sub-chapters
        rowitems = self._transform_toc(toc, document=self.document)
        self.create_chapters(rowitems)

    @execute_method_on_condition(attrgetter("settings.generate_chapters"))
    def create_chapters(self, rowitems: list[dict[str, Any]]) -> tuple[list[Chapter], list[SubChapter]]:
        chapter_list, subchapter_list = [], []
        for item in rowitems:
            subchapters = item.pop("subchapters", None)
            chapter, _ = Chapter.objects.get_or_create(
                book=item["book"], name=item["name"], defaults=item | {"summary": ""}
            )
            chapter_list.append(chapter)

            # Create related sub-chapters
            if subchapters:
                subchapter_data = [{**subchapter, "chapter": chapter} for subchapter in subchapters]
                subchapter_set = self.create_subchapters(subchapter_data)
            else:
                # Create one default subchapter per chapter if no more related
                # subchapter are present in the book
                subchapter_data = model_to_dict(chapter, fields=("name", "description", "page_start", "page_end"))
                subchapter_data.update({"chapter": chapter})
                subchapter_set = self.create_subchapters([subchapter_data])
```

```
            subchapter_list.extend(subchapter_set)

        self.chapter_list: list[Chapter] = chapter_list
        self.subchapter_list: list[SubChapter] = subchapter_list

        return self.chapter_list, self.subchapter_list

    @execute_method_on_condition(attrgetter("settings.generate_subchapters"))
    def create_subchapters(self, rowitems: list[dict[str, Any]]) -> list[SubChapter]:
        objs = []
        for item in rowitems:
            subchapter, _ = SubChapter.objects.get_or_create(
                chapter=item["chapter"], name=item["name"], defaults=item | {"summary": ""}
            )
            objs.append(subchapter)
        return objs

    @execute_method_on_condition(attrgetter("settings.generate_bookchunks"))
    def create_bookchunks(self) -> list[BookChunk]:
        partitioned_text = self.get_partitions(self.book.file)
        rowitems = self._transform_partitioned_text(partitioned_text)
        items = self._create_bookchunks(rowitems)
        return items

    # Additional methods omitted for brevity
```

2.2 Модель Book
Файл: book/models.py
```
import logging
from typing import Any

from django.core.validators import FileExtensionValidator
from django.db import models
from django.db.models import Max, Min

from base.models import CreatedLastModifiedBy, CreatedUpdatedAt
from book.enums import SupportedBookExtensions

logger = logging.getLogger(__file__)

class Book(CreatedUpdatedAt, CreatedLastModifiedBy):
    name = models.CharField(max_length=1028)
    description = models.TextField(blank=True)
    file = models.FileField(
        verbose_name="PDF File",
        upload_to="books/",
        validators=[FileExtensionValidator(SupportedBookExtensions.values)],
        max_length=256,
    )

    class Meta:
        ordering = ("-id",)
        db_table = "books"
```

```
    def __str__(self) -> str:
        return f"<{self.__class__.__name__}: {self.name}>"

    def get_page_range(self) -> tuple[int, int]:
        # Get range from subchapters
        data: dict[str, Any] = self.chapter_set.aggregate(Min("subchapter__page_start"),
Max("subchapter__page_end"))
        page_range = tuple(data.values())
        if any(page_number > 0 for page_number in page_range if page_number is not None):
            return page_range

        # Get range from chapters
        data: dict[str, Any] = self.chapter_set.aggregate(Min("page_start"), Max("page_end"))
        page_range = tuple(data.values())
        if any(page_number > 0 for page_number in page_range if page_number is not None):
            return page_range

        logger.warning(f"{self} range page is not estimated.")
        return (0, 0)
```

2.3 Валідатори та дозволи
Файл: base/validators.py
```
from dataclasses import dataclass, field
from typing import Any

from django.core.exceptions import ValidationError
from django.utils import timezone
from django.utils.deconstruct import deconstructible

@deconstructible
@dataclass
class IsFutureValidator:
    field_name: str = field()
    error_message: str = field(default="Make sure {field_name} is a future date.")

    def __call__(self, value: timezone.datetime, *args: Any, **kwds: Any) -> Any:
        if value and value < timezone.now():
            raise ValidationError(self.error_message.format(field_name=self.field_name))

    def __eq__(self, __value: object) -> bool:
        return self.field_name == __value.field_name and self.error_message == __value.error_message
```

2.4 Адміністративні дії
Файл: book/admin.py
```
import csv

from django.contrib import admin
from django.http.response import HttpResponse

from book.models import Book
```

```
from chapter.tasks import create_chapters_and_subchapters_summary_task

@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ("id", "name", "created_by", "created_at", "updated_at")
    date_hierarchy = "created_at"
    readonly_fields = ("created_at", "updated_at")
    list_select_related = ("created_by", "last_modified_by")
    raw_id_fields = ("created_by", "last_modified_by")
    search_fields = ("=id", "name__icontains")
    search_help_text = "Search by: id, name."
    actions = ["generate_chapter_summary", "export_chapter_summary"]

    @admin.action(description="Generate Chapter Summary")
    def generate_chapter_summary(modeladmin, request, queryset):
        for item in queryset:
            create_chapters_and_subchapters_summary_task.delay(book_id=item.id)

    @admin.action(description="Export Chapter Summary")
    def export_chapter_summary(modeladmin, request, queryset) -> None:
        item: Book = queryset.first()
        data = []
        for chapter in item.chapter_set.all():
            data.append(["1", chapter.name, chapter.summary])

            for subchapter in chapter.subchapter_set.all():
                if subchapter.summary:
                    data.append(["2", subchapter.name, subchapter.summary])

        response = HttpResponse(
            content_type="text/csv",
            headers={"Content-Disposition": f'attachment; filename="{item.name}-summary.csv"'},
        )
        writer = csv.writer(response)
        writer.writerow(["Level", "Chapter", "Summary"])
        writer.writerows(data)
        return response
```

3. Компонент "authentication"
3.1 OAuth з Google
Файл: authentication/views.py
```
import imghdr
from typing import Any

import requests
from django.contrib.auth import get_user_model
from django.contrib.auth.base_user import AbstractBaseUser, BaseUserManager
from django.contrib.auth.hashers import make_password
from django.core.exceptions import ObjectDoesNotExist
from django.core.files.base import ContentFile
from rest_framework import serializers, status
from rest_framework.generics import GenericAPIView
from rest_framework.permissions import AllowAny
```

```
from rest_framework.request import Request
from rest_framework.response import Response
from rest_framework.utils import json
from rest_framework_simplejwt.tokens import RefreshToken

from user.serializers import UserSerializer

class GoogleAuthSerializer(serializers.Serializer):
    access_token = serializers.CharField()

class GoogleAuthAPIView(GenericAPIView):
    serializer_class = GoogleAuthSerializer
    permission_classes = [AllowAny]

    def post(self, request: Request, *args, **kwargs) -> Response:
        serializer: GoogleAuthSerializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        payload = serializer.validated_data

        r = requests.get("https://www.googleapis.com/oauth2/v2/userinfo", params=payload)
        data = json.loads(r.text)

        if "error" in data:
            return Response(
                {"message": "Google token is either wrong or expired.", **data},
status=status.HTTP_400_BAD_REQUEST
            )

        try:
            user = get_user_model().objects.get(email=data["email"])
        except ObjectDoesNotExist:
            user = self.create_user(data)

        token = RefreshToken.for_user(user)
        return Response(
            {
                "user": UserSerializer(instance=user).data,
                "access": str(token.access_token),
                "refresh": str(token),
            }
        )

    def create_user(self, data: dict[str, Any]) -> AbstractBaseUser:
        User: AbstractBaseUser = get_user_model()

        user = User(
            email=data["email"],
            password=make_password(BaseUserManager().make_random_password()),
            first_name=data.get("given_name", ""),
            last_name=data.get("family_name", ""),
        )
        if picture_url := data.get("picture"):
            response = requests.get(picture_url)
            file = ContentFile(response.content)
            filename = f"{user.first_name}_{user.last_name}s_picture"
```

```
            extension = imghdr.what(file)
            if extension:
                filename = f"{filename}.{extension}"

            user.picture.save(filename, file, save=False)

        user.is_verified = True
        user.save()
        return user
```

3.2 Скидання паролю через JWT
Файл: authentication/views.py
```
import jwt
from django.conf import settings
from django.shortcuts import get_object_or_404
from rest_framework import status
from rest_framework.generics import GenericAPIView
from rest_framework.permissions import AllowAny
from rest_framework.request import Request
from rest_framework.response import Response

from authentication.serializers import ResetPasswordSerializer
from user.models import User

class ResetPasswordAPIView(GenericAPIView):
    permission_classes = (AllowAny,)
    queryset = User.objects.all()
    serializer_class = ResetPasswordSerializer

    def post(self, request: Request, *args, **kwargs) -> Response:
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        token = serializer.validated_data["token"]
        data = jwt.decode(token, settings.JWT_SECRET, algorithms=["HS256"])
        user = self.get_user(email=data["email"])
        user.set_password(serializer.validated_data["new_password1"])
        user.is_verified = True
        user.save()
        return Response(status=status.HTTP_204_NO_CONTENT)

    def get_user(self, email: str) -> User:
        queryset = self.get_queryset()
        return get_object_or_404(queryset, email=email)
```

4. Компонент "base"
4.1 Базові моделі та дозволи
Файл: base/models.py
```
from typing import Any
from uuid import uuid4
```

```
from django.contrib.auth import get_user_model
from django.db import models
from django.utils import timezone

class Model(models.Model):
    class Meta:
        abstract = True

    def __str__(self) -> str:
        return f"<{self.__class__.__name__}: #{self.id}>"

    def __repr__(self) -> str:
        return self.__str__()

class CreatedUpdatedAt(Model):
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True
        ordering = ("-id",)

class CreatedLastModifiedBy(Model):
    created_by = models.ForeignKey(
        get_user_model(), on_delete=models.CASCADE, related_name="created_%(class)s_set"
    )
    last_modified_by = models.ForeignKey(
        get_user_model(), on_delete=models.CASCADE, related_name="modified_%(class)ss_set"
    )

    class Meta:
        abstract = True
        ordering = ("-id",)

class Task(CreatedUpdatedAt, CreatedLastModifiedBy):
    class State(models.IntegerChoices):
        PENDING = 1
        IN_PROGRESS = 2
        DONE = 3
        FAILED = 4

    uuid = models.UUIDField(default=uuid4)
    name = models.CharField(max_length=256)
    state = models.IntegerField(choices=State.choices, default=State.PENDING)
    completed_at = models.DateTimeField(null=True, blank=True)
    args = models.JSONField(help_text="Positional arguments", default=dict, blank=True)
    kwargs = models.JSONField(help_text="Key arguments", default=dict, blank=True)
    result = models.JSONField(help_text="Task result", default=dict, blank=True)

    class Meta:
        ordering = ("-id",)
        db_table = "tasks"

    def __init__(self, *args: Any, **kwargs: Any) -> None:
        super().__init__(*args, **kwargs)
```

```
        self._initial_state = self.state

    def __str__(self) -> str:
        return f"<{self.__class__.__name__}: {self.name}>"

    def save(self, *args, **kwargs) -> None:
        if self.state_changed() and self.state == self.State.DONE:
            self.completed_at = timezone.now()
        else:
            self.completed_at = None
        return super().save(*args, **kwargs)

    def state_changed(self) -> bool:
        return self._initial_state != self.state
```

4.2 Кастомні класи дозволів
Файл: base/permissions.py
```
from typing import Any

from django.conf import settings
from django.contrib.auth.models import AbstractUser
from django.core.files import File
from django.db.models import Count
from rest_framework import status
from rest_framework.generics import GenericAPIView
from rest_framework.permissions import SAFE_METHODS, BasePermission, IsAuthenticated
from rest_framework.request import Request

from base.utils.file import get_file_size, get_page_number

class IsReadOnly(BasePermission):
    def has_permission(self, request: Request, view: GenericAPIView) -> bool:
        return request.method in SAFE_METHODS

    def has_object_permission(self, request: Request, view: GenericAPIView, obj: Any) -> bool:
        return self.has_permission(request, view)

class IsPremiumActiveOrReadOnly(IsAuthenticated, IsReadOnly):
    message = "You need to have premium to perform this action."

    def has_permission(self, request: Request, view: GenericAPIView) -> bool:
        user = request.user

        is_read_only = IsReadOnly.has_permission(self, request, view)
        if is_read_only:
            return True

        is_authenticated = IsAuthenticated.has_permission(self, request, view)
        return user and is_authenticated and user.premium_active

    def has_object_permission(self, request: Request, view: GenericAPIView, obj: Any) -> bool:
        return self.has_permission(request, view)
```

4.3 Декоратори та утиліти
Файл: base/decorators.py
```
import json
import logging
from typing import Any, Callable

from unstructured.staging.base import elements_to_json

logger = logging.getLogger(__file__)

def parse_elements_to_json(partition_function: Callable) -> Callable:
    def wrapper(*args, **kwargs) -> Any:
        elements = partition_function(*args, **kwargs)
        string_streams = elements_to_json(elements)
        elements = json.loads(string_streams)
        return elements

    return wrapper

def execute_method_on_condition(*flags: Any) -> Callable:
    def inner(bound_method: Callable) -> Callable:
        def wrapper(self, *args, **kwargs) -> Any:
            if all(predicate(self) for predicate in flags):
                return bound_method(self, *args, **kwargs)
            logger.log(logging.INFO, f"{bound_method} call is skipped.")

        return wrapper

    return inner

def log_on_exception(f: Callable) -> Callable:
    def wrapper(*args, **kwargs) -> Any:
        try:
            return f(*args, **kwargs)
        except Exception as exc:
            logger.error(exc)

    return wrapper
```

4.4 Сервіси інтеграції
Файл: base/services/google_analytics.py
```
import json
import logging
from typing import Any

import requests
from django.conf import settings
from rest_framework import status

logger = logging.getLogger("GoogleAnalyticsService")

class GoogleAnalyticsService:
```

```
    GOOGLE_ANALYTICS_ENABLED = settings.GOOGLE_ANALYTICS_ENABLED
    GOOGLE_ANALYTICS_URL = settings.GOOGLE_ANALYTICS_URL
    GOOGLE_ANALYTICS_MEASUREMENT_ID = settings.GOOGLE_ANALYTICS_MEASUREMENT_ID
    GOOGLE_ANALYTICS_API_SECRET = settings.GOOGLE_ANALYTICS_API_SECRET

    @classmethod
    def log_event(cls, payload: dict[str, Any]) -> None:
        if not cls.is_enabled():
            logger.warning("Google Analytics is not enabled.")
            return

        headers = {"Content-Type": "application/json"}
        params = cls._prepare_query_params()
        r = requests.post(cls.GOOGLE_ANALYTICS_URL, headers=headers, params=params,
data=json.dumps(payload))

        if r.status_code != status.HTTP_204_NO_CONTENT:
            logger.error(f"Request to Google-Analytics failed:\n{payload}.")

    @classmethod
    def _prepare_query_params(cls) -> dict[str, Any]:
        return {
            "measurement_id": cls.GOOGLE_ANALYTICS_MEASUREMENT_ID,
            "api_secret": cls.GOOGLE_ANALYTICS_API_SECRET
        }

    @classmethod
    def is_enabled(cls) -> bool:
        return bool(
            cls.GOOGLE_ANALYTICS_ENABLED
            and cls.GOOGLE_ANALYTICS_URL
            and cls.GOOGLE_ANALYTICS_MEASUREMENT_ID
            and cls.GOOGLE_ANALYTICS_API_SECRET
        )
```

5. Міграції та конфігураційні файли
5.1 Міграції Django
Файл: base/migrations/0003_industry_keyword.py
```
# Generated by Django 5.0.1 on 2024-02-08 10:51

from django.db import migrations, models

class Migration(migrations.Migration):
    dependencies = [
        ("base", "0002_task_created_by_task_last_modified_by"),
    ]

    operations = [
        migrations.CreateModel(
            name="Industry",
            fields=[
                ("id", models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name="ID")),
```

```
            ("created_at", models.DateTimeField(auto_now_add=True)),
            ("updated_at", models.DateTimeField(auto_now=True)),
            ("name", models.CharField(max_length=256)),
            ("description", models.TextField(blank=True)),
        ],
        options={
            "verbose_name_plural": "Industries",
            "db_table": "industries",
            "ordering": ("name",),
            "indexes": [models.Index(fields=["name"], name="industry__name_idx")],
        },
    ),
    migrations.CreateModel(
        name="Keyword",
        fields=[
            ("id", models.BigAutoField(auto_created=True, primary_key=True, serialize=False,
verbose_name="ID")),
            ("created_at", models.DateTimeField(auto_now_add=True)),
            ("updated_at", models.DateTimeField(auto_now=True)),
            ("name", models.CharField(max_length=256)),
        ],
        options={
            "db_table": "keywords",
            "ordering": ("name",),
            "indexes": [models.Index(fields=["name"], name="keyword__name_idx")],
        },
    ),
  ]
```

5.2 Конфігурації додатків
Файл: base/apps.py
```
from django.apps import AppConfig

class BaseConfig(AppConfig):
  default_auto_field = "django.db.models.BigAutoField"
  name = "base"
```

6. Тестування та покриття коду
6.1 Тестові класи
Файл: base/tests.py
```
from contextlib import contextmanager
from functools import partialmethod
from typing import Any, Generator
from unittest.mock import patch

from django.conf import settings
from django.contrib.auth import get_user_model
from django.core.files import File
from django.test import SimpleTestCase, TestCase
from mixer.backend.django import mixer
from rest_framework import status
```

```
from rest_framework.response import Response
from rest_framework.test import APIClient

class BasicTestCase(TestCase):
    @classmethod
    def setUpTestData(cls) -> None:
        cls.sys_user = mixer.blend(get_user_model(), email="system.administrator@exammaker.ai")
        cls.chat_postMessage = patch("base.services.slack.WebClient.chat_postMessage").start()

class APITestCase(SimpleTestCase):
    def assertContentType(self, response: Response, content_type: str) -> None:
        self.assertEqual(
            response.get("content-type"), content_type, f"Unexpected content type: {response.content_type}."
        )

    def assertStatus(self, response: Response, status_code: int) -> None:
        self.assertEqual(response.status_code, status_code, f"Unexpected status code: {response.status_code}.")

    assertJsonContentType = partialmethod(assertContentType, content_type="application/json")
    assertOkStatusCode = partialmethod(assertStatus, status=status.HTTP_200_OK)
    assertCreatedStatusCode = partialmethod(assertStatus, status=status.HTTP_201_CREATED)
    assertBadRequestStatusCode = partialmethod(assertStatus, status=status.HTTP_400_BAD_REQUEST)

    @classmethod
    def setUpClass(cls) -> None:
        cls.anon_client = APIClient()
        cls.chat_postMessage = patch("base.services.slack.WebClient.chat_postMessage").start()
        cls.google_analytics_post = patch("base.services.google_analytics.requests.post").start()

    def setUpClient(self, **client_kwargs) -> Any:
        password = client_kwargs.pop("password", None)
        self.user = mixer.blend(settings.AUTH_USER_MODEL, **client_kwargs)
        if password is not None:
            self.user.set_password(password)
            self.user.save()
        self.client = APIClient()
        self.client.force_authenticate(user=self.user)
        return self.user

@contextmanager
def load_asset_file(app: str, filename: str, mode: str = "rb", **kwargs) -> Generator[File, None, None]:
    filepath = settings.BASE_DIR / app / "tests/assets" / filename
    file = open(filepath, mode=mode, **kwargs)
    try:
        yield file
    finally:
        file.close()

@contextmanager
def override_properties(obj: object, **new_properties) -> Generator[object, None, None]:
    old_properties = {key: getattr(obj, key) for key in new_properties}
    try:
        for key, value in new_properties.items():
            setattr(obj, key, value)
        yield obj
```

```
    finally:
        for key, value in old_properties.items():
            setattr(obj, key, value)
```

7. Утиліти та допоміжні сервіси
7.1 Сервіс ToCSVConvertService
Файл: base/services/to_csv_convert.py
```
import csv
import datetime
import os
from pathlib import Path
from typing import Any

from django.conf import settings

class ToCSVConvertService:
    @classmethod
    def convert(cls, output_filename: str, rowdicts: list[dict[str, Any]]) -> None:
        absolute_directory_path: Path = settings.ROOT_DIR
        cls._make_output_dir(absolute_directory_path)
        filename = cls._get_filename(output_filename)

        with open(absolute_directory_path / filename, "x") as f:
            headers = rowdicts[0].keys()
            writer = csv.DictWriter(f, headers)
            writer.writeheader()
            writer.writerows(rowdicts)

    @classmethod
    def _get_filename(cls, filename: str) -> str:
        name, extension = os.path.splitext(filename)
        now = datetime.datetime.now().strftime("%Y-%m-%d_%H:%M:%S")

        if extension:
            return f"{name}_{now}.csv"
        return f"{filename}_{now}.csv"

    @staticmethod
    def _make_output_dir(absolut_path: str | Path) -> None:
        if not os.path.exists(absolut_path):
            os.mkdir(absolut_path)
```

7.2 Утиліти для роботи з OpenAI
Файл: base/openai_/utils.py
```
import openai
from django.conf import settings

def connect_to_openai() -> None:
    openai.api_key = settings.OPENAI_API_KEY
```

Файл: base/openai_/similarity.py
```
from base.utils import cosine_similarity
from .embedding import generate_embeddings

def get_similarity_score(target: str, text_list: list[str]) -> list[dict]:
    """
    Vectorize a list of texts and return the similarity score between the target
    text and each text in the list.
    """
    vectorized_target = generate_embeddings(target)
    vectorized_text: list = [generate_embeddings(text) for text in text_list]
    # Calculate similarity scores
    similarity_scores: list[dict] = []
    for index, vectorized_text_element in enumerate(vectorized_text):
        similarity_scores.append(
            {
                "text": text_list[index],
                "similarity": cosine_similarity(vectorized_target, vectorized_text_element),
            }
        )
    # Sort similarity scores in descending order (highest similarity first)
    similarity_scores.sort(key=lambda x: x["similarity"], reverse=True)
    return similarity_scores
```

Файл: base/openai_/completion.py
```
from typing import Any

import backoff
import openai
import requests
from django.conf import settings
from openai.error import ServiceUnavailableError

def get_gpt_completion(
    prompt: str,
    system_prompt: str = "You are an exam paper writer.",
    model: str = settings.GPT_VERSION,
    temperature: float = 0.6,
    top_p: float = 1.0,
    frequency_penalty: float = 0.25,
    presence_penalty: float = 0.0,
    stop: list[str] = None,
    response_format: dict | None = None,
) -> str:
    """Get GPT completion for a given prompt."""
    stop = stop or ["<<END>>"]
    response_format = response_format or {"type": "json_object"}
    chat_completion = openai.ChatCompletion.create(
        model=model,
        response_format=response_format,
        messages=[
            {"role": "system", "content": system_prompt},
```

```
            {"role": "user", "content": prompt},
        ],
        temperature=temperature,
        top_p=top_p,
        frequency_penalty=frequency_penalty,
        presence_penalty=presence_penalty,
        stop=stop,
    )
    return chat_completion["choices"][0]["message"]["content"]

@backoff.on_exception(
    backoff.constant,
    (requests.exceptions.RequestException, ServiceUnavailableError),
    max_tries=5,
)
def get_gpt_completion_with_backoff(system_prompt: str, prompt: str) -> Any:
    return get_gpt_completion(system_prompt=system_prompt, prompt=prompt)
```

7.3 Міксини для ViewSet'ів
Файл: base/mixins/bulk_create_mixin.py
```
from rest_framework import status
from rest_framework.decorators import action
from rest_framework.request import Request
from rest_framework.response import Response

class BulkCreateMixin:
    @action(methods=["post"], detail=False)
    def bulk_create(self, request: Request, *args, **kwargs) -> Response:
        serializer = self.get_serializer(data=request.data, many=True)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
```

Файл: base/mixins/bulk_update_mixin.py
```
from django.db import transaction as t
from django.db.models import Model, QuerySet
from rest_framework import status
from rest_framework.decorators import action
from rest_framework.exceptions import ValidationError
from rest_framework.request import Request
from rest_framework.response import Response

from base.serializers import IdSerializer

class BulkUpdateMixin:
    @t.atomic()
    @action(methods=["patch"], detail=False)
    def bulk_update(self, request: Request, *args, **kwargs) -> Response:
        serializer = IdSerializer(data=request.data, many=True)
        serializer.is_valid(raise_exception=True)
        model = self._get_model()
```

```
        queryset = model.objects.all().filter(id__in=(item["id"] for item in self.request.data))
        if queryset:  # To evaluate queryset
            pass
        queryset = self._validate_queryset(queryset)

        serializers = []
        errors = []
        for item_data in request.data:
            instance = queryset.get(pk=item_data["id"])
            serializer = self.get_serializer(instance, data=item_data, partial=True)
            try:
                serializer.is_valid(raise_exception=True)
            except ValidationError as exc:
                errors.append(exc.detail)
                continue
            serializers.append(serializer)

        if errors:
            raise ValidationError(errors)
        for serializer in serializers:
            serializer.save()
        return Response([serializer.data for serializer in serializers], status=status.HTTP_200_OK)

    def _validate_queryset(self, queryset: QuerySet) -> QuerySet:
        errors = []
        for item_data in self.request.data:
            try:
                queryset.get(pk=item_data["id"])
            except queryset.model.DoesNotExist:
                errors.append({item_data["id"]: "Object with such id does not exist."})
        if errors:
            raise ValidationError(errors)
        return queryset

    def _get_model(self) -> Model:
        return self.get_queryset().model
```

Файл: base/mixins/bulk_delete_mixin.py
```
from rest_framework import status
from rest_framework.decorators import action
from rest_framework.request import Request
from rest_framework.response import Response

from base.serializers import IDsSerializer

class BulkDeleteMixin:
    @action(methods=["delete"], detail=False)
    def bulk_delete(self, request: Request, *args, **kwargs) -> Response:
        serializer = IDsSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        ids = serializer.validated_data["ids"]
        queryset = self.get_queryset().filter(id__in=ids)
        queryset.delete()
```

```
        return Response(status=status.HTTP_204_NO_CONTENT)
```

8. API та маршрутизація
8.1 URL-конфігурація для автентифікації
Файл: authentication/urls.py
```
from django.urls import path
from rest_framework_simplejwt.views import TokenObtainPairView, TokenRefreshView, TokenVerifyView

from authentication.views import GoogleAuthAPIView, RegisterAPIView, ResetPasswordAPIView

# Root: /api/
urlpatterns = [
    path("token/", TokenObtainPairView.as_view(), name="token-obtain-pair"),
    path("token/refresh/", TokenRefreshView.as_view(), name="token-refresh"),
    path("token/verify/", TokenVerifyView.as_view(), name="token-verify"),
    path("oauth/google/", GoogleAuthAPIView.as_view(), name="oauth-google"),
    path("auth/register/", RegisterAPIView.as_view(), name="register"),
    path("auth/reset-password/", ResetPasswordAPIView.as_view(), name="reset-password"),
]
```

8.2 Ендпоінт для перевірки стану системи
Файл: base/views.py
```
from rest_framework.decorators import api_view, permission_classes, renderer_classes
from rest_framework.permissions import AllowAny
from rest_framework.renderers import JSONRenderer
from rest_framework.request import Request
from rest_framework.response import Response

@api_view(("GET",))
@permission_classes((AllowAny,))
@renderer_classes((JSONRenderer,))
def health(request: Request, *args, **kwargs) -> Response:
    return Response({"status": "healthy"})
```

8.3 Маршрутизація для базових моделей
Файл: base/urls.py
```
from rest_framework.routers import DefaultRouter

from base.viewsets import TaskViewSet

router = DefaultRouter()
router.register("tasks", TaskViewSet, basename="tasks")

urlpatterns = router.urls
```