Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мережних технологій

Курсова робота

освітній ступінь – бакалавр

на тему: «**Побудова багаторівневого веб-застосування на платформі Google Cloud Platform (GCP)**»

Виконав: студент 4-го року навчання,
Спеціальності
122 «Комп’ютерні науки»

Кармелюк Костянтин Олександрович

Керівник Черкасов Д.І.
К.т.н, старший викладач
«20» травня 2022 р.

Київ – 2022

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра Мережних технологій

Освітній ступінь бакалавр

Спеціальність 122 «Комп'ютерні науки»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Малашонок Г.І.

"10" жовтня 2021 року

ЗАВДАННЯ

ДЛЯ КУРСОВОЇ РОБОТИ СТУДЕНТУ

Кармелюка Костянтина Олександровича

1. Тема роботи **Побудова багаторівневого веб-застосування на платформі Google Cloud Platform (GCP)"**, керівник роботи Черкасов Дмитро Іванович, старший викладач, кандидат фізико-математичних наук.

2. Строк подання студентом роботи 20 травня 2022

3. План роботи

Анотація
Вступ
Розділ 1. Аналіз існуючих рішень
- 1.1 Огляд типів архітектур
  - 1.1.1 n-рівнева архітектура
    - 1.1.1.1 Однорівнева
    - 1.1.1.2 Дворівнева
    - 1.1.1.3 Трирівнева
    - 1.1.1.4 Багаторівнева
  - 1.1.2 Мікросервісна архітектура
  - 1.1.3 Порівняння архітектур
  - 1.1.4 Висновок
- 1.2 Порівняльний аналіз існуючих систем

Розділ 2. Структурна розробка власного рішення
- 2.1 Порівняльний аналіз сервісів
- 2.2 Визначення використаних систем
  - 2.2.1 Рівень представлення
  - 2.2.2 Рівень бізнес-логіки (застосунку)
  - 2.2.3 Рівень баз даних
  - 2.2.4 Додаткові сервіси
    - 2.2.4.1 Cloud Build
    - 2.2.4.2 Logging
    - 2.2.4.3 IAM

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	11 жовтеня 2021			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	15 жовтня 2021 – 27 листопада 2021			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	30 листопада 2021			
4.	Написання розділів роботи *або* Постановка експерименту, аналіз отриманих результатів наукового дослідження	1 грудня 2021 – 15 лютого 2022			
5.	Проміжний контроль виконання роботи	16 лютого 2022			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	16 лютого 2022 – 30 квітня 2022			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	20 грудня 2021			
	Розділ 2 (аналітично-дослідницька частина) (експериментальна частина для природничих і біологічних наук)	31 березня 2022			
	Розділ 3 (проектно-рекомендаційна частина) (аналіз результатів експерименту для природничих і біологічних наук)	18 квітня 2022			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	10 травня 2022			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	20 травня 2022			

Графік узгоджено «11» жовтня 2021 р.
Науковий керівник Черкасов Дмитро Іванович
Виконавець курсової роботи Кармелюк Костянтин Олександрович

Зміст

Анотація

У даній роботі розглядається використання Google Cloud Platform (GCP) для побудови багаторівневого веб-застосування, який відповідає вимогам якості сучасних застосунків. Аналізуються та порівнюються різні архітектурні рішення та різні підходи розгортання веб-застосувань, а також сервіси, які при цьому використовуються. У якості прикладу застосування розглядається створення веб-застосунку для перевірки текстів на плагіат.

Вступ

За останні 10 років кількість користувачів в інтернеті стрімко зросло (див рисунок 1), і тому з'являється необхідність підвищення якості веб-застосунків, їх відмовостійкості. Також зараз активно змінюються потреби бізнесу, тому також необхідно забезпечити можливість швидкого додавання і оновлення можливостей веб-застосунків.

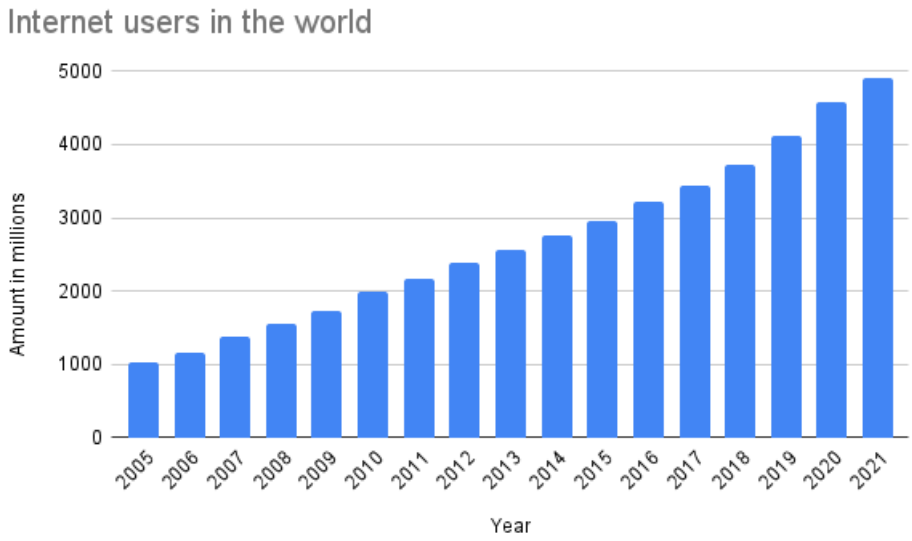


Рисунок 1. Кількість користувачів мережі інтернет. З даних statista

Для задоволення гнучкості, масштабованості, відмовостійкості а також ефективного розподілу задач між розробниками використовуються багаторівневі архітектури. Найпоширеніший варіант має три базові рівні:

- Користувацький інтерфейс
- Бізнес логіка
- База даних

Також можлива наявність проміжних рівнів, які забезпечують безпеку і покращують ефективність, але створюють інші проблеми.

Використання трирівневої архітектури полегшує розробку та підтримку наявних можливостей, проте не вирішує ряд проблем, до яких належить:

- Розгортання та налаштування застосунку
- Оновлення версії застосунку
- Моніторинг та логування даних
- Адміністрування бази даних
- Тощо

Усі перелічені проблеми ефективно вирішують хмарні сервіси, які об’єднують можливі рішення, які можливо під'єднати до проекту для використання.

Однією з таких платформ, яка об’єднує хмарні сервіси є GCP (Google Cloud Platform), яку використовує компанія Google для своїх застосунків.

Метою цієї роботи є аналіз архітектур, порівняння переваг і недоліків використання хмарних сервісів, та розробка веб-застосунку на основі багаторівневої архітектури з використанням хмарних обчислень.

В результаті війни, розпочатої росією, велика кількість українських користувачів відмовились використовувати російські сервіси та платформи. Усі лідери ринку перевірки на плагіат, а саме advego, content-watch, etxt тощо - мають російське походження, а наявність якісних альтернатив на українському ринку відсутня. Тому розробка даного веб-застосунку є актуальною під час написання цієї роботи.

Основними вимогами до застосунку є:

- Багаторівнева архітектура
- Динамічна масштабованість в залежності від рівня використання ресурсів

- Інтеграція в процес CI/CD та автоматизоване розгортання застосунку
- Налаштування моніторингу стану веб-застосунку

Розділ 1. Аналіз існуючих рішень

1.1 Огляд типів архітектур

Використання хмарної інфраструктури потребує від розробників чіткого бачення системи на етапі проектування. Кілька фундаментальних шаблонів з'являються знову і знову протягом історії архітектури програмного забезпечення, оскільки вони забезпечують корисний погляд на організацію коду, розгортання або інші аспекти архітектури [1].

На початку життєвого циклу розробки будь-якого застосування проектується архітектура, яка згодом може виявитись непридатною до використання із певним архітектурним рішенням. Таким чином, при розробці архітектури особливу увагу потрібно приділити які переваги та недоліки вона має з інфраструктурного боку. Розглянемо н-рівневі та мікросервісні архітектури та порівняємо їх.

1.1.1 n-рівнева архітектура

Багаторівнева архітектура, також відома як n-рівневий стиль архітектури, розділяє застосунок на логічні шари та фізичні рівні. Шари – це спосіб розділити обов’язки та керування залежностями.

Рівні фізично розділені, та працюють на окремих машинах. Рівень може звертатися безпосередньо до іншого рівня або використовувати асинхронний обмін повідомленнями (черга повідомлень) для взаємодії. Хоча кожен шар може бути розміщений на окремому рівні, це не обов’язково. Фізичне розділення рівнів покращує масштабованість і стійкість, та також додає можливість розподілення обов’язків, проте також

додає затримку від додаткового мережевого зв’язку і складнощі керування розгортанням. [2]

Цей стиль архітектури є де-факто стандартом для більшості програм, перш за все завдяки своїй простоті, звичності та невисокій вартості. Це також дуже природний спосіб розробки додатків відносно закону Конвея, який стверджує, що організації, які розробляють системи, вимушені робити системи, які архітектурно є копіями комунікаційних структур цих організації [1, ст. 263]. Організації зазвичай мають розробників інтерфейсу користувача, розробників серверної частини, і іноді експертів баз даних. Ці організаційні рівні зручно співпадають з рівнями багаторівневої архітектури, що робить її природним вибором для багатьох додатків.

Більш того, на розповсюдженість цієї архітектури впливає той фактор, що якщо розробник або архітектор не впевнені, який стиль архітектури вони використовують, або якщо команда розробників Agile «починає щось кодувати», то існують великі шанси, що вони реалізують багаторівневу архітектуру. [1, ст. 263]

1.1.1.1 Однорівнева

Однорівнева архітектура передбачає розташування усіх компонентів системи на одному сервері (див рисунок 1.1.1.1) і є найпростішою з усіх n-рівневих. Це гарний спосіб тестування застосунків під час розробки і варіант для розташування застосунків з невеликим трафіком, які потребують ефективного використання ресурсів. Таку систему легко налаштовувати і підтримувати, і також вона є ефективною з фінансової сторони, оскільки потребує невеликих грошових затрат[3].

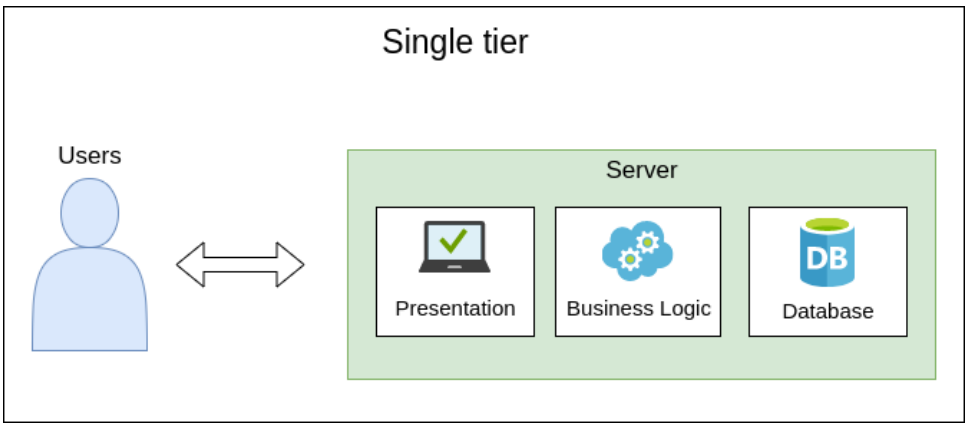


Рисунок 1.1.1.1 Однорівнева архітектура

Проте існування усіх ресурсів системи на одній машині може створювати проблеми з безпекою та доступністю. Якщо сервер вийде з ладу, то уся система перестане працювати. Також, якщо сервер буде атаковано, то шанси втратити дані дуже великі.

Варто зауважити, для масштабування однорівневої архітектури необхідно збільшувати сервер вертикально, тобто збільшувати обчислювальні здатності пристрою. Це створює проблему “стелі”, до якої сервер може масштабуватись.

1.1.1.2 Дворівнева

Дворівнева архітектура побудована на клієнт-серверній моделі, і передбачає від’єднання рівню даних від клієнтської частини на інший сервер (див рисунок 1.1.1.2), тобто клієнтський сервер відповідає за інтерфейс і бізнес логіку. Зв’язок між сервером бази даних і клієнтським застосунком швидкий, оскільки не має проміжний шарів.

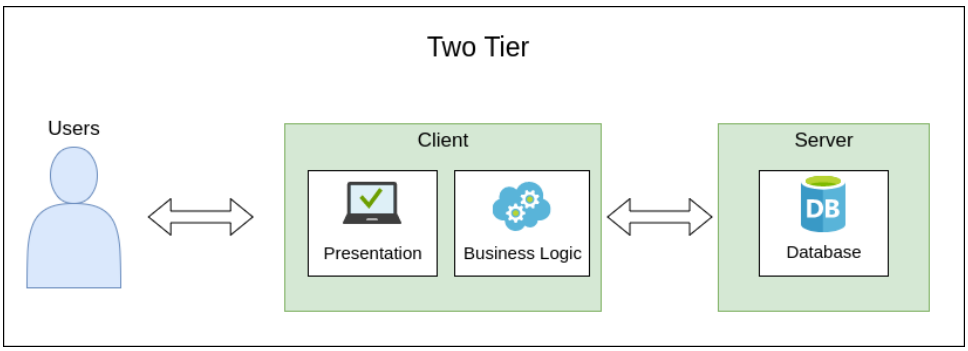


Рисунок 1.1.1.2 Дворівнева архітектура

Клієнтська частина містить код, який вміє взаємодіяти з користувачем, а також який вміє зберігати дані в базу даних. Клієнтський застосунок надсилає запит до серверу баз даних, який в свою чергу оброблює запит і повертає результат.

Така система так само не є легко розширюваною, оскільки клієнтська частина сильно пов’язана між собою, проте можливості масштабованості збільшились, оскільки тепер можливо розширювати, як і клієнтську частину, так і рівень даних. Основним вузьким місцем, окрім проблем з масштабованістю, гнучкістю та високим мережевим трафіком, є проблема з обслуговуванням (оновленням коду) [4].

1.1.1.3 Трирівнева

Трирівнева архітектура є найпопулярнішою. Від дворівневої її відрізняє винесення користувацького інтерфейсу на інший рівень (див рисунок 1.1.1.3).

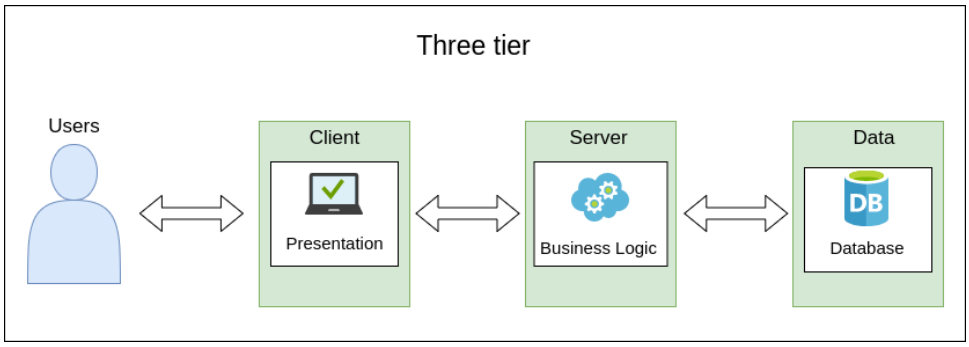


Рисунок 1.1.1.3 Трирівнева архітектура

В результаті маємо три рівні:

1) Рівень відображення, який відповідає за користувацький інтерфейс, за допомогою якого кінцевий користувач взаємодіє з застосунком . Головна мета цього рівня - відображення даних та збір даних від користувача [5].
2) Рівень бізнес-логіки (або застосунку). На цьому рівні інформація, зібрана на рівні презентації, обробляється з використанням бізнес-логіки, певного набору бізнес-правил. Рівень програми також може додавати, видаляти або змінювати дані на рівні даних.
3) Рівень даних. На цьому рівні зберігаються дані, які обробив рівень бізнес-логіки.

Трирівнева архітектура вирішує проблеми, які були зазначені до цього, а саме:

- Підвищує можливість масштабованості, оскільки кожен рівень може бути масштабований окремо, що в свою чергу підвищує доступність.

- Покращує гнучкість системи та можливість оновлення рівнів без впливу на інші рівні, що в свою чергу надає можливість розробки частин системи окремими командами одночасно.

Такі досягнення зумовлені відділенням рівню бізнес-лолгіки та сервісів безпеки, надійності, транзакцій тощо на проміжний рівень [4]. Разом з з’являється можливість реплікування бази даних на декілька серверів, що запобігає проблемі втрати даних. Ну головною перевагою є можл

Проте така система ускладнює налаштування та керування рівнями і моніторинг за статусом. Також збільшується кількість взаємодій між рівнями, що призводить до зменшення швидкості відповіді а також до необхідності забезпечення безпеки передачі даних між рівнями.

1.1.1.4 Багаторівнева

Багаторівнева архітектура виникає з 3-рівневої додаванням нових шарів, що розширюють рівень застосунку. Кожен з цих рівнів повинен мати чітко визначений інтерфейс, який дозволяє їм контактувати та спілкуватися один з одним [4].

Головною проблемою такого підходу може бути випадкове створення додаткового рівню застосунку у вигляді звичайного CRUD-сервісу, який додає надлишкову затримку, не виконуючі жодної корисної роботи [2]. Як і раніш, додавання додаткових рівнів ускладнює підтримку та моніторинг за статусом системи.

1.1.2 Мікросервісна архітектура

Мікросервісна архітектура складається з колекції малих, автономних сервісів, які і називаються мікросервісами (див рисунок 1.1.2). Кожен сервіс має реалізовувати окрему бізнес-логіку у визначеному контексті і взаємодіяти між собою за допомогою визначених API [6].

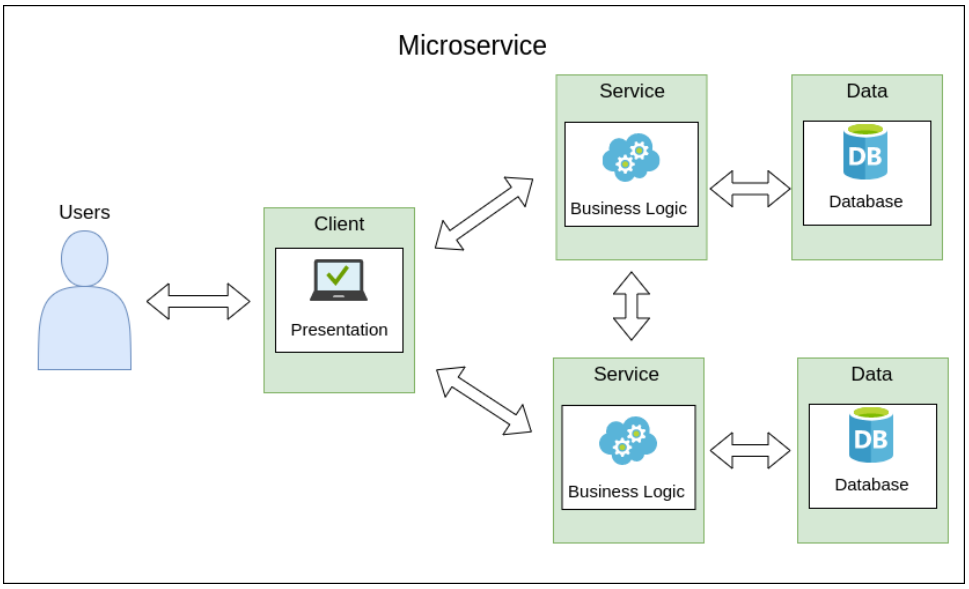


Рисунок 1.1.2 Мікросервісна архітектура

Оскільки мікросервіси автономні, то кожен мікросервіс може розроблювати і підтримувати окрема невелика команда, яка також має змогу використання різних технологій та бібліотек для різних мікросервісів. Більш того, їх можна оновлювати без впливу на інші сервіси та без необхідності оновлювати увесь застосунок.

Кожен сервіс також може бути масштабований окремо. За допомогою оркестратору, наприклад Kubernetes, можна згрупувати сервіси на одному хості, що дозволить ефективніше використовувати ресурси.

Хоча кожен сервіс автономний і поодинці легко налаштовуваний, то вже система в цілому набагато важча в управлінні, ніж рівневе

застосування. Також, тестувати такий застосунок важко, оскільки компоненти пов’язані між собою. Більш того, швидкість відповіді може сильно сповільнитися в результаті ланцюгового звертання одного сервісу до іншого. Для того, щоб правильно керувати і налаштовувати мікросервісну архітектуру, необхідно мати досвідченого DevOps фахівця, на якого буде покладена велика частина роботи.

1.1.3 Порівняння архітектур

Архітектура	Переваги	Недоліки
Однорівнева	- Дуже легко реалізувати - Дуже легко підтримувати і тестувати	- Доступна лише вертикальна масштабованість - Вихід з ладу серверу спричинить повне вимикання застосунку
Дворівнева	- Легкість реалізації - Легкість підтримки і тестування	- Складність масштабованості - Складність оновлення застосунку
Трирівнева	- Підвищення доступності - Швидкість розробки, оскільки можлива одночасна розробка сервісів різними командами - Збільшення надійності	- Складність управління - Збільшення часу відповіді - Складність управління та моніторингу

Багаторівнева	- Усі переваги трирівневої - Збільшення масштабованості (в порівнянні)	- Усі недоліки трирівневої - Збільшення часу відповіді, в залежності від взаємодіями між рівнями
Мікросервісна	- Найкраща масштабованість - Швидкість розробки, оскільки можлива одночасна розробка сервісів різними командами - Ізоляція помилок	- Важке налаштування взаємодії сервісів - Складність управління та тестування - Збільшення часу відповіді - Дорожче за інші архітектури

Таблиця 1.3 Порівняння арзітектур

1.1.4 Висновок

Кожен з наступних рівнів підвищує масштабованість і швидкість розробки, завдяки розділенню рівнів, проте ускладнює управління цими системами та моніторинг їх статусу.

Трирівнева є найбільш збалансованою архітектурою, оскільки її легше підтримувати і розробляти, за н-рівневу, та вона є більш відмовостійкою, ніж архітектури з меншою кількістю рівнів.

1.2 Порівняльний аналіз існуючих систем

За останні десять років сфері SEO (**Search Engine Optimization**) почали приділяти велику увагу, оскільки інтернет швидко популяризується і користувачам необхідно видавати під час пошуку якісний контент, який буде відповідати їх запитам.

Для того, щоб сторінка видавалась однією з перших, необхідно її якісно оптимізувати. Пошукові системи при видачі результатів веб-сторінок звертають увагу на технічну сторону веб-сторінки і на контент, який вона містить. З технічної сторони враховується швидкість завантаження сторінки та її оптимізація під різні пристрої (мобільні або персональні комп'ютери). З сторони контенту, враховуються такі критерії [7]:

- Унікальність тексту - сторінка, в якої "вкрадений" текст може бути непроіндексована, в результаті чого її не буде видавати при пошуку.
- Відповідність тексту ключовим словам - кожна сторінка рекомендується відповідно до певних ключових слів, які користувач вводить в пошукову систему. Велика кількість входження ключових слів може спричинити зменшення рейтингу сторінки.
- Степінь "нудоти" - перенасичення тексту одними і тими самими словами. Існують два види - класична й академічна. Відрізняються вони алгоритмом розрахунку. Чітко визначеного ліміту нудоти не існує, але зазвичай такою межею вважають 7 для класичної і 4% для академічної.
- Степінь води тексту.
- тощо

Перед початком розробки системи необхідно проаналізувати існуючі рішення на ринку. Таким чином можна визначити, наскільки доцільно розробляти нову систему. З основних факторів було виділено такі критерії (в порядку спадання важливості):

1. Визначення степені унікальності та оригінальності та визначення з яких джерел було запозичено текст.

Виділення частин тексту, які були виявлені, як плагіат, надає змогу користувачеві можливість переробити, або перефразувати текст, для підвищення оригінальності та унікальності тексту. Визначення джерел, звідки запозичено текст надає змогу користувачеві вказати джерела, звідки були взяті дані.

2. Визначення степені нудоти, як академічної, так і класичної

Степінь нудоти, а разом з ним визначення слів, які дуже часто використовуються в тексті, надають змогу переробити текст на більш оптимізований, що дозволить отримати вищий рейтинг у пошукових системах.

3. Визначення степені води

Степінь води вказують користувачеві, що необхідно більше зосередитись на головних моментах тексту, що в свою чергу оптимізує текст для видачі у пошукових системах.

4. Кількість символів, які можна перевірити безкоштовно.

Бажано, щоб витрати на перевірку були невеликими, або взагалі були відсутні. Чим більша кількість символів, які можна перевірити, тим краще. Особливо це стосується користувачів, які створюють тексти для використання на невеликих веб-сторінках.

5. Можливість коригування помилок

Граним доповненням системи може бути автоматичне пропонування коригування помилок. Таким чином система буде корисною не лише у визначенні плагіату, а й у покращенні тексту користувача.

6. Зрозумілий інтерфейс

Інтерфейс має бути зрозумілим для користувача та легким у використанні. Оновлення параметрів аналізу не має втрачати інформацію надану від користувача до цього.

Варто зауважити, що будуть порівнюватись лише ті системи, які:

- Не знаходяться у реєстрі російських сайтів (не мають російські домени)
- Дозволяють перевірку україномовних текстів
- Не зареєстровані у росії

Для перевірки роботи сервісів було створено текст, який містить плагіат (див додаток А). Було порівняно такі сервіси Н: plag.com.ua, unicheck.com, strikeplagiarism.com, та duplichecker.com. Розглянемо кожен з них більш детально відповідно до визначених критеріїв.

1. Plag.com.ua [8]

Сервіс надає можливості перевірки на плагіат і визначає три їх види:

- Плагіат "копіювати-вставити"
- Плагіат неправильного цитування
- Перефразований плагіат

Додатково сервіс визначає оригінальність тексту і з яких джерел було взято текст. Для вчителів, професорів, лекторів надає безкоштовну повну перевірку. Для університетів є можливість безкоштовної співпраці. Інтерфейс зрозумілий та легкий у використанні. Безкоштовний варіант сервісу не має обмежень на кількість символів, проте обмежений у функціоналі.

З недоліків сервісу можна виділити - неможливість перевірки чистого тексту, тобто сервіс приймає на вхід лише файли. Використання сервісу звичайними користувачами потребує або довгого очікування в черзі, або використання так званих “токенів”, які надають можливість отримувати розширений звіт і швидку перевірку. Токени купуються за гроші, або є можливість отримання малої кількісті один раз в день.

В результаті роботи було визначено плагіат (див малюнок 1.2.1), показано актуальні джерела, визначено рівень плагіату та інші характеристики. На жаль, ресурс не демонструє нудоту тексту та кількість води, також хотілося б бачити рівень плагіату у відсотках. Сервіс додатково пропонує виправлення помилок за токени за допомогою стороннього сервісу **Mistakeless.**

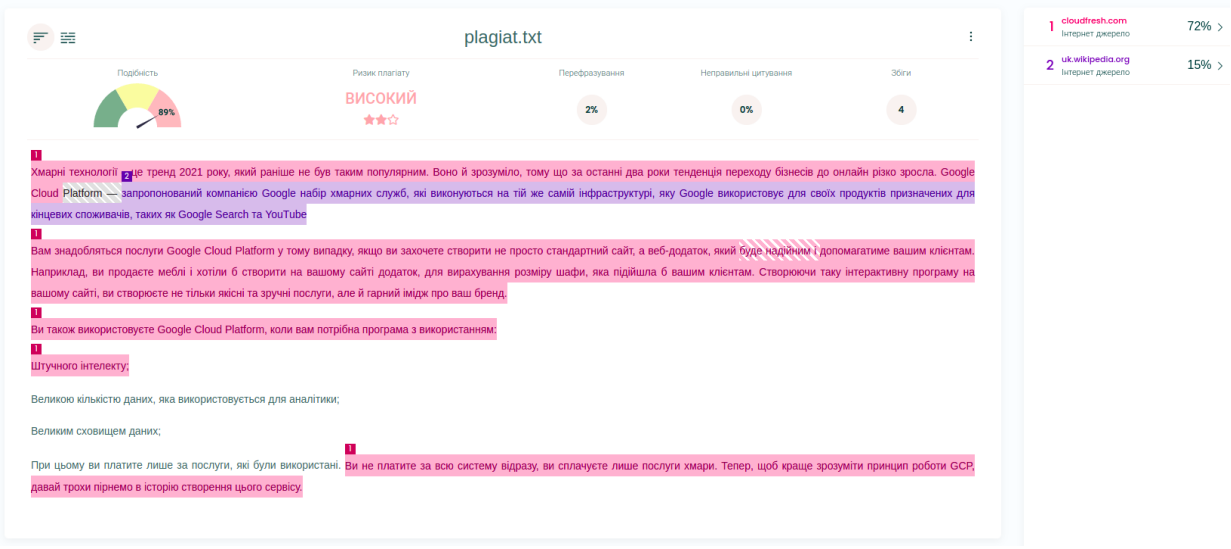


Рисунок 1.2.1 інтерфейс plag.com.ua

Загалом система виконує свою роботу, але не є зручною для користувачів, які потребують частої перевірки своїх текстів.

2. Unicheck.com [9]

Цей сервіс більше розрахований на академічний аналіз великих текстів. Сервіс перевіряє на плагіат, визначає джерела. Безкоштовно можлива лише перевірка до 200 символів. Під час тестування безкоштовний варіант мав збій, тому було взято приклад з вебсторінки(див малюнок 1.2.2).

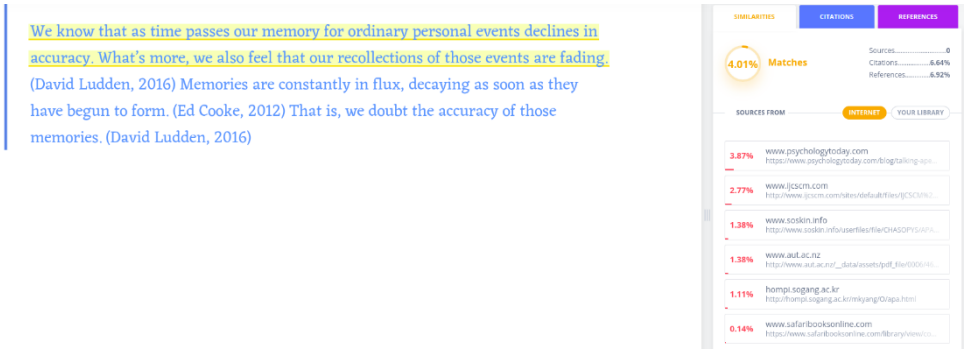


Рисунок 1.2.2 інтерфейс unicheck

Сервіс аналізує рівень плагіату, позначає місця, де його було знайдено. Також сервіс автоматично виключає з визначення плагіату цитати, які можуть впливати на рівень плагіату.

Загалом сервіс має зручний інтерфейс та виконує аналіз на плагіат, проте не відповідає іншим критеріям.

3. Strikeplagiarism.com [10]

Сервіс надає змогу аналізувати текст на плагіат з перевіркою на схожість, перефразування тощо. Сервіс не має безкоштовних варіантів перевірки, що робить його менш привабливим. Можливо аналізувати як документи, так і звичайний текст. Користувач за токени може пришвидшити перевірку,

також проаналізувати документи відповідно до великої бази даних дисертацій та публікацій і перевірити документ на перефраз.

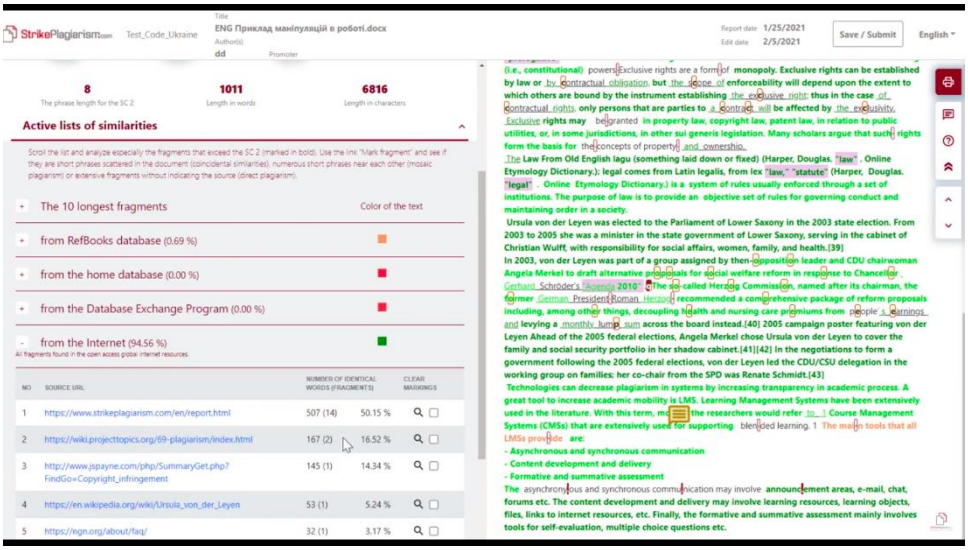


Рисунок 1.2.3 інтерфейс strikeplagiarism

Було взято приклад з запису вебсторінки(див малюнок 1.2.3), оскільки немає безкоштовного варіанту перевірки.

В результаті роботи було визначено плагіат та його рівень, показано актуальні джерела. Також для кожного джерела було визначено, з якої бази даних було його взято, а також відсоток співпадінь з цим джерелом. Також є можливість підтвердження джерел, що зменшує рівень плагіату в реальному часі, що зручно для перевірки академічних робіт. Сервіс не визначає нудоту тексту, рівень води, оригінальність та не надає можливості коригування тексту. Також немає відповідності тексту і джерелу, звідки взято текст - усі джерела відображаються списком.

Система виконує перевірку на плагіат та визначає джерела і надає розширений звіт, але має не має можливості безкоштовної перевірки і не відповідає іншим критеріям.

4. Duplichecker.com [11]

Сервіс надає можливість перевірки на плагіат, визначення джерел та виділення частин тексту, визначених, як плагіат. Можливо перевіряти, як документи, так і звичайний текст. Можлива лише перевірка до 1000 (тисячі) символів у безкоштовному варіанті.

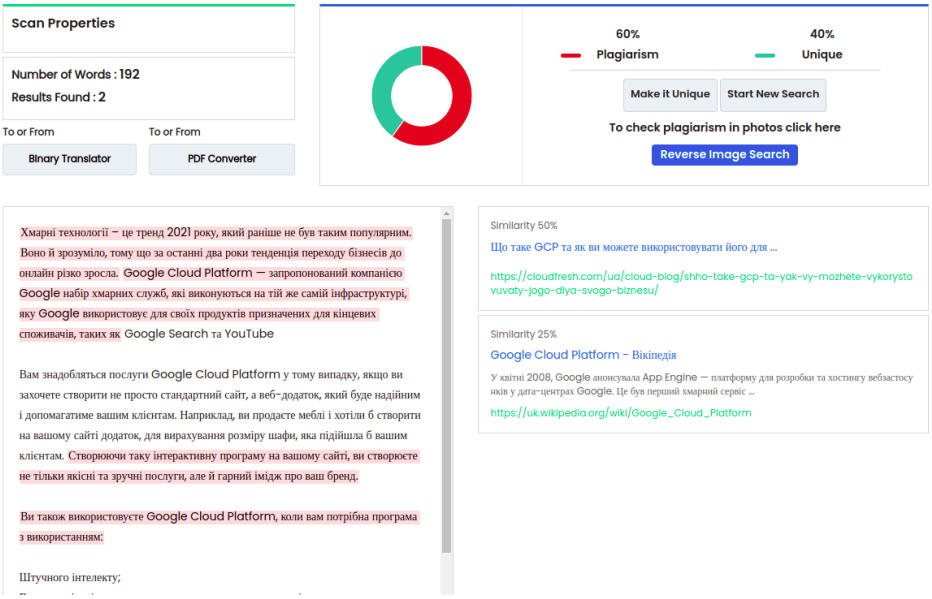


Рисунок 1.2.4 інтерфейс duplichecker

В результаті роботи було визначено плагіат та його рівень, показано актуальні джерела. Сервіс не визначає нудоту тексту, рівень води, оригінальність та не надає можливості коригування тексту. Також немає відповідності тексту джерелу, звідки взято текст - усі джерела відображаються списком.

Система виконує перевірку на плагіат та визначає джерела, але має невелику кількість безкоштовних символів і не відповідає іншим критеріям.

Як видно з результату огляду існуючих систем, не було виявлено систему, яка буде відповідати зазначеним критеріям, що підтверджує доцільність розробки нової системи.

Розділ 2. Структурна розробка власного рішення

Визначивши основні критерії застосунку та розібравши типи архітектур, необхідно створити власну архітектуру та визначити технології та сервіси, які будуть використані. Було вирішено використовувати трирівневу архітектуру, але необхідно визначити, за допомогою яких сервісів буде запущений кожен з рівнів.

2.1 Порівняльний аналіз сервісів

Для роботи рівня бізнес-логіки(застосунку) та рівня представлення можна використати один або поєднання декількох сервісів з таких [12]:

- Compute Engine – IAAS у вигляді віртуальних машин, при створенні яких визначаються їх параметри, а саме - кількість процессорів, оперативної пам’яті, яка операційна система має бути на ньому тощо.
- Kubernetes Engine - Використання кластерів Kubernetes. Kubernetes — це портативна платформа для розгортання, керування та масштабування контейнерних додатків за допомогою інфраструктури Google. Середовище GKE (Google Kubernetes Engine) складається з декількох машин Compute Engine згрупованих у кластер [13].
- Cloud Run - безсерверна платформа, яка запускає окремі контейнери. Платформа отримує код або контейнер, який містить застосунок, і автоматично запускає його і масштабує за потреби. Така платформа дозволяє зосередити більшість уваги на написанні коду і невелику кількість на конфігурацію та масштабування застосунку [14].
- App Engine - повністю керована, безсерверна платформа для розробки та розміщення веб-додатків. Вона автоматично керує мережею, сервером і базою даних і масштабує їх за потреби [12].

- Cloud functions - Безсерверні функції, які реагують на події. Необхідно писати окремі функції, які будуть викликатись після появи зазначених подій. Наприклад поява нового файлу у сховищі, або новий запит до АПІ тощо [12].

Вибір сервісу залежить від рівня абстракції, якого потребує рівень. Якщо необхідно визначати інфраструктуру застосунку, тоді гарним варіантом буде Compute Engine. Його зазвичай використовують для міграції застарілих застосунків у хмару.

Якщо застосунок контейнерізований, тоді можливі два варіанти: GKE або Cloud Run. GKE надає більшу гнучкість у налаштуванні, надає змогу обрати ОС та різні мережеві протоколи, найкраще підходить для мікросервісної архітектури. З іншого боку, Cloud Run автоматично масштабує інфраструктуру і гарно підходить для застосунків, які працюють з HTTP запитами і вебсокетами.

Якщо застосунок просто виконує дію у відповідь на запит або подію, тоді гарним варіантом буде Cloud Functions.

Також на вибір сервісів впливає розмір команди розробки, кількість грошей, яка витрачається на підтримку роботи застосування:

- Якщо команда невелика, тоді гарним варіантом буде Cloud Run та App Engine, тому що для налаштування цих сервісів немає необхідності мати людей, які будуть відповідти за інфраструктуру і масштабування застосунку. Для більших команд краще підходить Computer Engine і GKE, які дають більшу свободу налаштувань сервісу безперервної інтеграції, безпеки тощо [12].
- Якщо звертати увагу на використання коштів на підтримку, то Computer Engine та GKE стягують плату за підтримку існуючих

екземплярів систем, незалежно від їх використання, тобто сплачуються попередньо обрані обчислювальні ресурси. З іншого боку Cloud Run, App Engine та Cloud Functions стягують плату за кожен запит, тобто сплата стягується лише за те, що дійсно використовується [12].

Для рівня баз даних, який буде реляційним, був вибір між Cloud SQL та Cloud Spanner. Cloud SQL автоматизує підтримку бази даних, дозволяє автоматично робити бекап системи та відновлювати системи, які постраждали у результаті катастроф чи інших проблем [15]. Cloud Spanner розрахований на надвелике навантаження, забезпечує 99.999 відсоткову доступність і велику масштабованість, але є доволі коштовним сервісом, порівнюючи з Cloud SQL

2.2 Визначення використаних систем

З огляду на зазначені переваги і недоліки кожного з сервісів було визначено використовувати такі:

2.2.1 Рівень представлення

Для рівня представлення було вирішено використовувати сервіс Cloud Run. Він надає змогу ефективно використовувати придбані ресурси, не витрачати ресурси на контроль за масштабуванням, оскільки це робиться автоматично, та такий сервіс легше тестувати і до нього дуже зручно налаштовується безперервна інтеграція.

В якості фреймворку було вирішено використовувати **nuxt**, який використовує реактивний фреймворк **vue** та його доповнення **vue-router** і **vuex**. Також фреймворком підтримується використання пакувальнику **webpack**, транспілятор **esbuild** тощо [16]. Головною перевагою **nuxt** над vue

є можливість налаштування вебсторінки під **SEO** за допомогою серверного рендерінгу.

Інформація про користувача, його токен, та загальний стан зберігається у vuex. Запити на отримання інформації та відправки даних виконуються за допомогою бібліотеки axios.

2.2.2 Рівень бізнес-логіки (застосунку)

Для рівня бізнес-логіки було вирішено використовувати App Engine. Таким чином ми дозволяємо сервісу безперервно працювати і виконувати фонові задачі, як аналіз тексту. Для розгортання необхідний описовий файл app.yml. Було вирішено використовувати мову **python** та фреймворк **Django.** Було вирішено використовувати архітектуру **REST** і її було реалізовано за допомогою розширення **drf** (django-rest-framework). Для авторизації та аутентифікації використовуються **jwt** токени. Структура бази даних визначається за допомогою моделей і можлива взаємодія за допомогою вбудованої ORM системи.

Застосунок архітектурно поділений на моделі, представлення і серіалізатори. Моделі представляють дані, які зберігаються у застосунку, і відповідають за збереження даних і отримання їх з бази даних. Представлення отримують запити користувачів, оброблюють і серіалізують дані, отримані від моделей, у визначеному форматі і повертають їх користувачеві.

Усі необхідні параметри налаштувань записуються у файлі config.settings, який потім можливо використовувати у будь-якій частині застосунку. Там задаються дані для підключення до бази даних, налаштовується логування та задаються параметтри налаштувань

під’єднаних розширень. Для документації ендпойнтів використовується **SwaggerUI** та **OpenAPI**

2.2.3 Рівень баз даних

Для рівня баз даних було обрано Cloud SQL з сервером PostgreSQL, оскільки він відповідає вимогам застосунку. Сервіс автоматично створює резервні копії, що дозволяє відновити стан бази даних в результаті катастрофи.

База даних містить 12 таблиць, з яких 3 відповідають створеним сутностям, а інші 9 створені автоматично фреймворком django. Також база даних буде закрита ззовні і буде мати лише приватну IP адресу, що означає, лише дозволені сервіси будуть мати змогу підключитись до неї.

2.2.4 Додаткові сервіси

2.2.4.1 Cloud Build

Cloud Build - сервіс, який забезпечує безперервну інтеграцію та розгортання застосунків. Побудова застосунку відбувається на продуктивних машинах, також вихідний код та образи кешується, що робить побудову дуже швидкою. Можливо визначити яким чином буде відбуватися побудова з додаванням проміжних етапів, таких як тестування, статичний аналіз, використання gulp тощо.

Цей сервіс було використано для безперервної інтеграції та побудови застосунку за допомогою підключення його до репозиторію github. Побудова та розгортання відбувається за рахунок docker контейнерів

2.2.4.2 Logging

Cloud logging - сервіс, який відповідає за збір та подальший аналіз логувань застосунку. За допомогою **Logs Explorer** можливо виконувати пошук, сортування та візуалізації логувань. Також можливо встановлення сповіщень, якщо заздалегідь визначені логування з'явились. **Error Reporting** аналізує логування автоматично і збирає їх у логічні групи, що дозволяє виявляти нові проблеми. Також можливо налаштування фільтрів логувань, які автоматично будуть відкидати неважливі логування [22]. Цей сервіс було використано для збіру та аналізу логів застосунку.

2.2.4.3 IAM

IAM - Сервіс управління ідентифікацією та доступом (**Identity and Access Management**), який надає змогу визначити права доступу окремим сервісам чи користувачам до визначених ресурсів. Цей сервіс дозволяє застосувати принцип найбільш необхідних прав, який полягає у надані ресурсам (користувачам та сервісам) лише тих прав, які їм дійсно необхідні [19]. Сервіс використовується при будь-якій взаємодії з GCP.

2.2.4.4 Cloud Load Balancing

Cloud Load Balancing - сервіс, який надає можливість будувати високодоступні сервіси, за допомогою балансування навантаження, який надає змогу переносити трафік користувачів з сервісів, які вийшли з ладу, на працюючі. Балансувальний реагує на кількість користувачів, трафіку, навантаження мережі, "здоров'я" сервісів та інші умови, що надає можливість швидко відповідати на несподівані зростання навантаження за допомогою перенаправлення трафіку на інші сервери Надає можливість балансування трафіку HTTP(S), TCP/SSL та UDP [21]. Також є можливість підключення балансувальника до сервісу логування, що надає змогу проаналізувати трафік.

Цей сервіс було використано для масштабування застосунку у відповідь на зміну трафіку.

2.2.4.5 Cloud Storage

Cloud Storage - відмовостійкий сервіс для зберігання об’єктів будь-якого типу з високою безпекою об’єктів. Об’єкт - набір незмінних даних будь-якого формату. Для кожного об’єкту чи набору об’єктів можливо визначити права доступу і зробити їх публічно доступними. Існують різні типи сховищ, які відрізняються вартістю в залежності від необхідності регулярного доступу до даних та наявності автоматичного копіювання даних [17].

Цей сервіс було використано для збереження необхідних файлів, які використовуються CDN а також для збереження документів користувачів.

2.2.4.6 Cloud CDN

Cloud CDN - сервіс, який дозволяє швидко доставляти веб контент за допомогою мережевої інфраструктури, яка надсилає клієнту файли з найближчого вузла гугл, що зменшує час відповіді. Також за допомогою кеша пришвидшуються відповіді на подальші запити. Це робиться шляхом кешування статичних веб файлів(javascript, css) або картинок. Cloud CDN автоматично кешує файли при використанні, але можливо визначити окремо бажані налаштування [18]. Цей сервіс було використано для пришвидшення відповіді клієнтам.

2.2.4.7 Container Registry

Container Registry - приватне сховище **docker** контейнерів, в якому можливо їми керувати, аналізувати на вразливості, та налаштовувати права доступу [23]. Цей сервіс було використано для збереження контейнерів застосунку і запуску їх у Cloud Run.

2.2.4.8 Cloud Armor

Cloud Armor - сервіс, який допомагає захистити сервіси, які працюють за балансувальником навантажень, від різних загроз, таких як **DDOS, XSS, SQL Injection**. Він надає автоматичний захист, який миттєво визначає та знешкоджує мережеві атаки і дозволяє лише правильно створені запити, а також дозволяє додатково налаштувати власний захист, такий який брендмауер. Такий сервіс з коробки вміє знешкоджувати 10 найбільших небезпек відповідно до **OWASP** top 10 [20]. Цей сервіс було використано для підвищення безпеки застосунку.

2.2.4.9 Cloud Monitoring

Cloud Monitoring - сервіс, який дозволяє слідкувати за доступністю, продуктивністю та “здоров’ям” застосунку. За допомогою цього сервісу можливо налаштувати **SLO** (**service-level objectives**), тобто бажану продуктивність, за допомогою **SLI** (**service-level indicators**), тобто індикатору продуктивності. Цей сервіс було використано для моніторингу статусу застосунку.

2.2.4.10 Serverless VPC

Serverless VPC - сервіс, який дозволяє безсервісним застосункам, таким як Cloud Run, App Engine та Cloud Functions підключатись до приватної хмарної мережі напряму і відправляти запити до мережі VPC за допомогою внутрішньої IP адреси. Цей сервіс було використано для взаємодії рівнем бізнес-логіки та базою даних.

2.2.4.11 Cloud DNS

Cloud DNS - Надійний DNS сервер з малою затримкою, який працює з мережі Google по всьому світу. Сервіс надає можливість автоматичної

масштабованості по різним зонам, що пришвидшує відповідь користувачам.

2.2.5 Визначення архітектури

Визначивши усі сервіси, які будуть використовуватись маємо таку архітектуру:

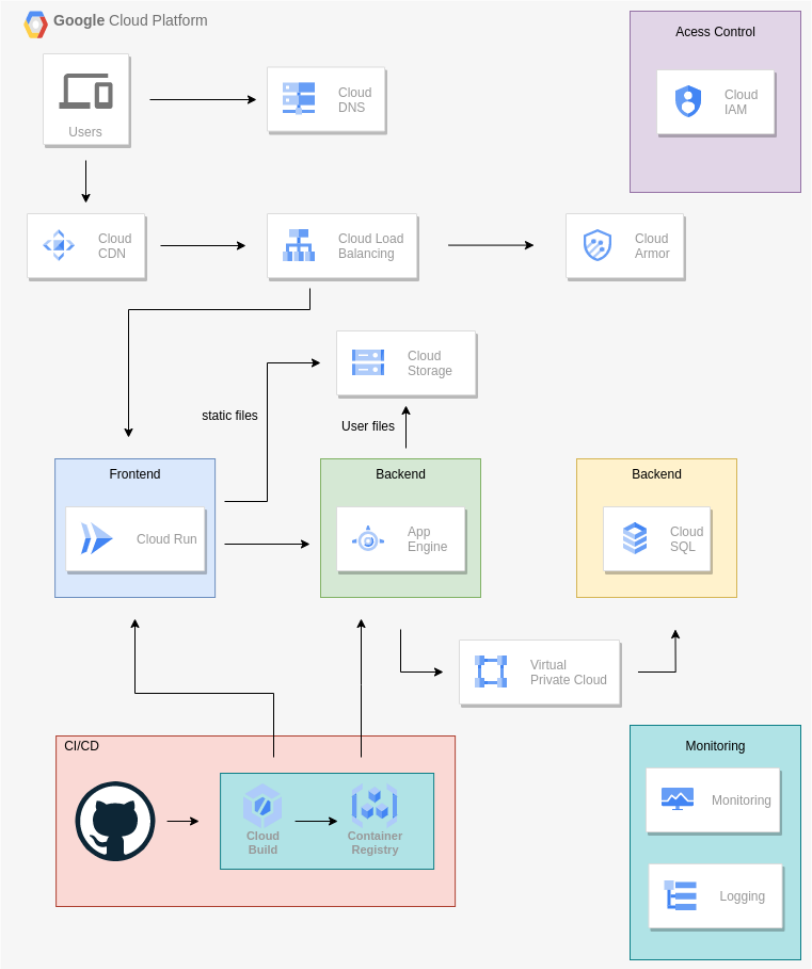


Рисунок 2.2.5 архітектура застосунку

Умовно застосунок поділений на три частини(див малюнок 2.2.5). Перша - сам застосунок, який відповідає на запити користувача, другий - процес CI/CD і третій відповідає за моніторинг і логування.

За допомогою github, Cloud Build та Cloud Container Repository виконується процес CI/CD. Логи з усіх рівнів збираються за допомогою

Cloud Logging і потім можуть бути візуалізовані та проаналізовані за допомогою сервісу Cloud Monitoring. IAM визначає права доступу сервісів.

2.2.6 Опис функціонування сервісу

Спочатку клієнт доступається до сторінки через зареєстровану адресу. За допомогою сервісу Cloud DNS користувач отримує адресу застосунку. Якщо CDN не має закешовних даних, які запитує користувач, то він потрапляє на балансувальник навантаження, який перевіряє що запитує користувач, якщо користувач запитує статичні файли, то вони отримуються з Cloud Storage, якщо вони публічні. Усі інші запити перенаправляються на рівень представлення, який їх опрацьовує і повертає згенеровану сторінку. Дані отримуються з рівня бізнес-логіки, який також має публічну адресу, як і рівень представлення даних.

При відкритті сторінки застосунку користувач бачить головну сторінку, на якій може або ввести чистий текст, який буде відправлено на перевірку на плагіат, або ж залити документ. Тут встановлено обмеження до 200 символів для користувачів, які не зареєстровані. Користувач може зареєструватися і тоді обмеження знімається. Аутентифікація відбувається за допомогою емейлу та паролю, у відповідь на які отримується JWT токен. Запити вимагають від користувача наявність цього токена у хедері Authorization. При переході на сторінку власного аккаунту користувач може побачити минулі перевірки, завантажені файли та результати цих перевірок.

Для реалізації jwt було використано розширення **djangorestframework-simplejwt.** На малюнку (див малюнок 2.2.6) можна побачити діаграму послідовності отримання токену та аутентифікації користувача за допомогою нього.

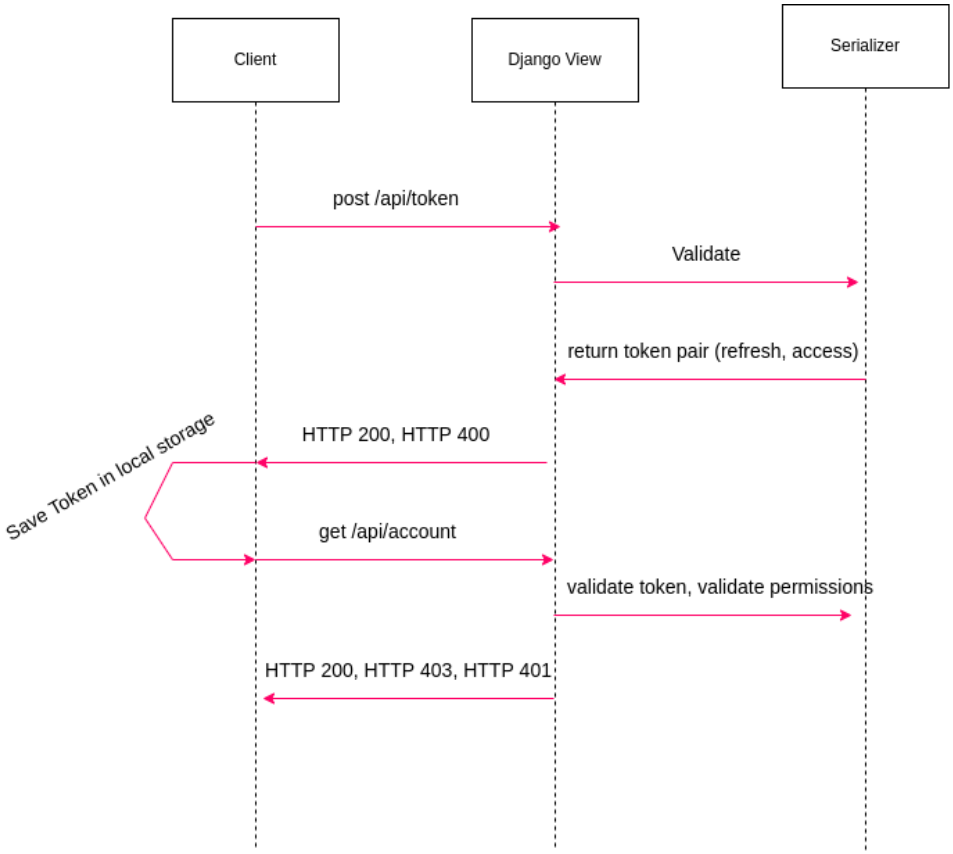


Рисунок 2.2.6 діаграма потоку використання jwt токену

Розділ 3. Процес розробки застосування

Для розробки застосунку спочатку було створено окремий проект у GCP з назвою plagitarism.

3.1 Рівень Бізнес-логіки

Було створено необхідні моделі для роботи з даними. Було перевизначено модель користувача, який надається фреймворком **Django** за замовченням, оскільки він має дані, які нам не потрібні, а саме first_name, last_name, username. Замість цього було встановлено email як первинний ключ користувача. Відповідно до змін стандартного коду було змінено менеджера моделі (додаток Б), який відповідає за основні дії ОРМ.

Для використання змінних середовища було використано розширення django-environ, яке автоматично парсить існуючі змінні середовища, які також можуть бути прочитані з файлу, що полегшує локальне тестування.

Для використання Cloud Storage разом з **Django** було використано cloud-storages, який полегшує взаємодію між сервісом та застосунком. Було написано функцію, яка розмежовує місця зберігання відповідно до встановлених налаштувань, що полегшує тестування локально, оскільки немає потреби для підключення до сервісів google (додаток Г)

Для інтеграції логування Google Logging до застосунку було використано бібліотеку **google-cloud-logging,** яка використовує вбудований модуль logging, який підтримується django (додаток Ґ).

Для створення схеми OpenAPI було використано розширення django-rest-swagger (додаток Д).

Для підключення до бази даних будуть використовуватись змінні середовища. Завдяки попередньо зазначеному django-environ, який автоматично парсить стрічку для підключення до бази даних у форматі, який сприймає django, можливий такий лаконічний запис (див рисунок 3.1.1).

```
DATABASES = {"default": env.db()}
```

Рисунок 3.1.1 функція, яка автоматично перетворює юрл для підключення до бази даних у правильний формат

Для парсингу документів, які користувач завантажує було створено окрему функцію, яка відповідно до типу MIME читає вміст. Для парсингу PDF файлів було використано **pdftotext** (додаток Е)**.**

Для роботи з текстом було використано бібліотеку nltk, яка полегшує роботу з текстами і надає можливості розпізнавання різних частин мови.

Для підрахунку класичної тошноти тексту було знайдено кількість використань найбільш вживаного слова і взято його корінь. Для підрахунку академічної тошноти береться найбільш вживане слово у всіх його морфемах, ділиться на загальну кількість слів і множиться на 100%. Поганим вважається показник більше 7 для класичної тошноти і більше 6% для академічної [24]. Для підрахунку води було використано словник стоп-слів - слів, які не несуть цінності у реченні, і рахується воно як відношення кількісті стоп-слів до інших слів. Базовим показником вважається не більше 60%.

Для визначення рівня оригінальності і плагіату було використано API unicheck, який згадувався до цього [9]. Для виправлення помилок було використано API Webspellchecker, який надає змогу працювати з текстами, написаними українською мовою [25].

Для деплою було створено окремий файл app.yaml, який містить необхідні для запуску параметри, а саме версію python, змінні середовища. (додаток Є) . Також для взаємодії бази даних та рівня бізнес-логіки необхідно налаштувати serverless vpc, оскільки база даних має лише приватну IP адресу, що і було зазначено у файлі.

Також було налаштовано дошку з метриками за допомогою сервісу Monitoring.

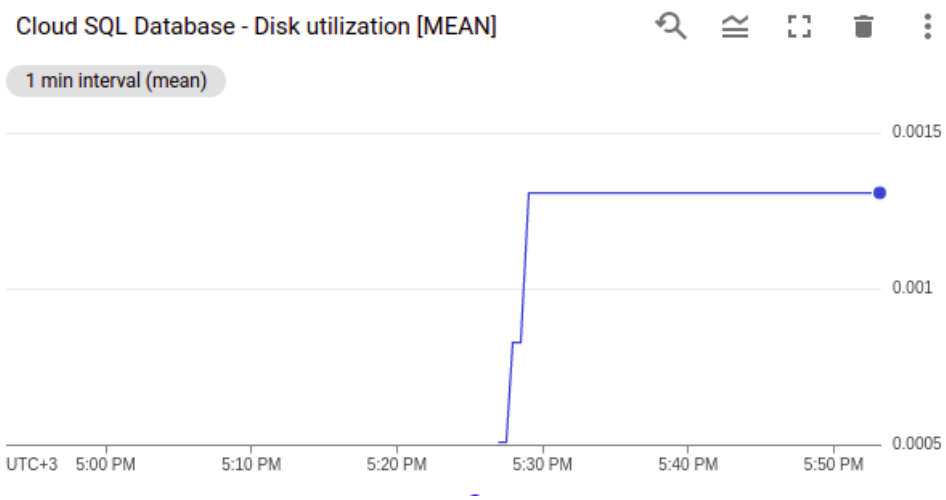


Рисунок 3.1.2 налаштоване логування системи

Розгортання застосунку, відбувається за допомогою тригерів. Кроки описуються у файлі cloudbuild.yaml (див малюнок 3.1.3).

```
steps:
  - name: "gcr.io/cloud-builders/gcloud"
    args:
      [
        "app",
        "deploy"
      ]
timeout: "1600s"
```

Рисунок 3.1.3 файл, за допомогою якого відбувається деплой рівня застосунку

3.2 Рівень відображення

Для відображення було використано Nuxt. Запити на рівень представлення відправляються за допомогою axios. Для зберігання токенів jwt та даних користувача використовується vuex.

Необхідно ввімкнути Cloud Container Repository і Cloud Build для того, щоб працював автоматичний деплой. Також необхідно дозволити сервісному аккаунту, який виконує побудову, виконувати дії деплою Cloud Run.

Для деплою необхідно було створити Dockerfile, в якому був би описаний процес запуску рівня відображення (див малюнок 3.2.1).

```
FROM node:11.13.0-alpine

WORKDIR /app
# Install dependencies
COPY package*.json ./
RUN npm install

# Copy everything
COPY . ./
RUN npm run build
# Define variables
EXPOSE 3000

ENV HOST 0.0.0.0
ENV PORT 3000
# Start app
CMD [ "npm", "start" ]
```

Рисунок 3.2.1 докерфайл рівня представлення

Застосунок автоматично будується і розгортається у сервісі Cloud Run. Було налаштовано CI/CD у онлайн сервісі контролю версій github. Розгортання відбувається завдяки попередньо-зазначеному Dockerfile.

На малюнку (див рисунок 3.2.2) можна побачити результат побудови застосунку. Було побудовано контейнер, який був визначений на минулому малюнку.

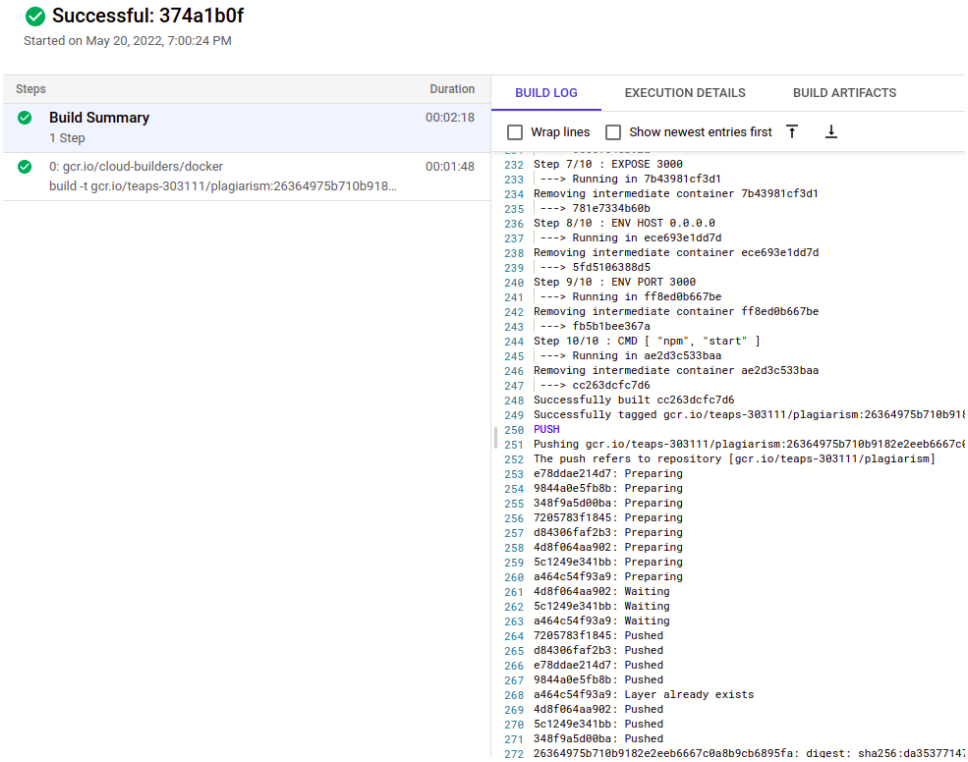


Рисунок 3.2.2 результат побудови застосунку

Розгортання відбувається у три кроки (див малюнок 3.2.3). На першому, за допомогою Cloud Build будується контейнер, потім він додається до регістру. Третій крок запускає деплой у Cloud Run.

```yaml
steps:
  - name: gcr.io/cloud-builders/docker
    args:
      ["build", "-t", "gcr.io/$PROJECT_ID/plagiarism:${SHORT_SHA}", "."]

  - name: "gcr.io/cloud-builders/docker"
    args: ["push", "gcr.io/$PROJECT_ID/plagiarism"]

  - name: "gcr.io/cloud-builders/gcloud"
    args:
      [
        "run",
        "deploy",
        "plagiarism",
        "--image",
        "gcr.io/$PROJECT_ID/${_SERVICE_NAME}:${SHORT_SHA}",
        "--region",
        "eu-west3",
        "--platform",
        "managed",
        "--allow-unauthenticated",
      ]
timeout: "1600s"
```

Рисунок 3.2.3 файл, який містить кроки для безперервної інтеграції

3.3 База даних

У сервісі Cloud SQL було створено базу даних PostgreSQL 14-ї версії у регіоні EU-West3 (Frankfurt) з лише внутрішньою IP адресою. Також було налаштовано автоматичний бекап даних.

Було створено окремого користувача баз даних, який має лише найбільш необхідні права для керування нею.

Особливим є поле result моделі PlagiarismCheck, оскільки воно має тип JSON і зберігає результати аналізу тексту (див малюнок 3.3).

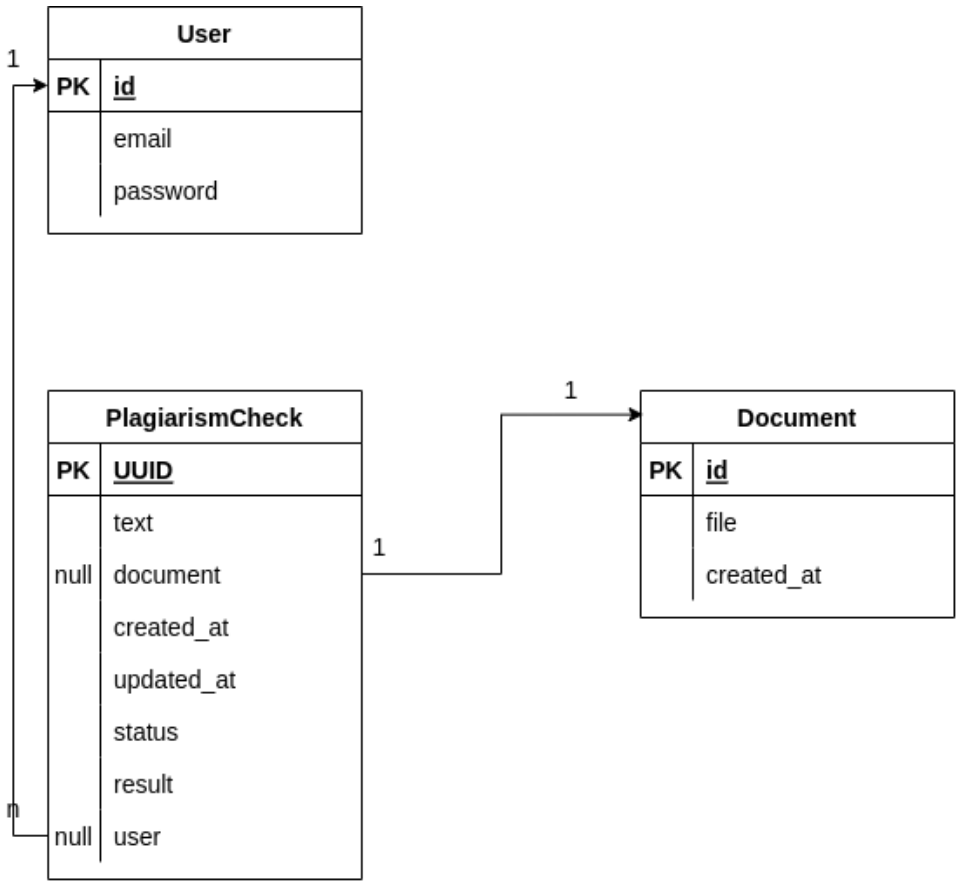


Рисунок 3.3 Схема бази даних

Додатково в базі даних створились таблиці, які відповідають за внутрішні процеси django, такі як: ролі, групи, дозволи користувачів, міграції, сесії, логування та інші.

Висновки

У результаті роботи було спроектоване і побудоване багаторівневе веб-застосування на платформі Google Cloud Platform, головна ціль якого перевірка текстів та документів на плагіат. Спочатку було проаналізовано переваги та недоліки різних рівневих архітектур, в результаті чого було обрано трирівневу. Також було досліджено актуальність розробки застосунку і порівняння існуючих систем, виявилось, що не існує системи, яка б задовільнила усі вимоги, які були поставлені.

Було проведено аналіз сервісів GCP, а саме різних варіантів розгортань застосунків, порівняно їх переваги та недоліки, також було проведено цінову оцінку. Було проаналізовано різні варіанти баз даних, сервісів для підтримання безпеки застосунку, можливостей інтегрування і розгортання застосунку у результаті безперервної інтеграції тощо. Результатом стало використання таких сервісів: Cloud Run, App Engine, Cloud Load Balancing, Cloud Build, Cloud SQL та інших. Для застосунку були використані актуальні технології, а саме - реактивне представлення інтерфейсу користувача за допомогою фреймворку nuxt, використання фрейморку django, що прискорює розробку застосунку, використання оптимізованих бібліотек python, що полегшують аналіз текстів, а саме nltk, використання контейнеризації Docker та інше.

У подальшому розроблену систему можливо розширити, додавши нові алгоритми перевірки файлів, також можливо додати підтримки різних регіонів планети, щоб зменшити час відповіді для користувачів, які знаходяться у різних місцях світу. Також можливо подальше перенесення застосунку на технологію GKE, при збільшенні команди розробки, для підвищення масштабованості та відмовостійкості.

Джерела

[1] Neal Ford & Mark Richards. Fundamentals of Software Architecture – 2020.

[2] N tier architecture [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier

[3] Singletier vs Multitier [Електронний ресурс] - Режим доступу до ресурсу: https://docs.bitnami.com/aws-templates/singletier-vs-multitier/

[4] Клієнт-северна архітектура [Електронний ресурс] - Режим доступу до ресурсу: - https://www.mcours.net/cours/pdf/info/ClientServer_Architectures.pdf

[5] Three Tier Architecure [Електронний ресурс] - Режим доступу до ресурсу: https://www.ibm.com/cloud/learn/three-tier-architecture

[6] Microservices [Електронний ресурс] - Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[7] Seo Starter Guide [Електронний ресурс] - Режим доступу до ресурсу: https://developers.google.com/search/docs/beginner/seo-starter-guide

[8] Plag [Електронний ресурс] - Режим доступу до ресурсу: https://www.plag.com.ua/

[9] Unichek [Електронний ресурс] - Режим доступу до ресурсу: https://unicheck.com/

[10] Strikeplagiarism [Електронний ресурс] - Режим доступу до ресурсу: https://strikeplagiarism.com/ua/

[11] Duplichecker [Електронний ресурс] - Режим доступу до ресурсу: https://www.duplichecker.com/

[12] Where should I run my staff [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/blog/topics/developers-practitioners/where-should-i-run-my-stuff-choosing-google-cloud-compute-option

[13] Kubernetes engine overview [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview

[14] What is cloud Run [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/run/docs/overview/what-is-cloud-run

[15] Google cloud database options explained [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/blog/topics/developers-practitioners/your-google-cloud-database-options-explained

[16] Nuxt introduction [Електронний ресурс] - Режим доступу до ресурсу: https://v3.nuxtjs.org/guide/concepts/introduction/

[17] Google Storage [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/storage

[18] What is Cloud cdn and how does it work [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/blog/topics/developers-practitioners/what-cloud-cdn-and-how-does-it-work

[19] Cloud IAM [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/iam/docs/overview

[20] Cloud Armor [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/armor/docs/cloud-armor-overview

[21] Load Balancing [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/load-balancing

[22] Logging [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/logging

[23] Container Registry [Електронний ресурс] - Режим доступу до ресурсу: https://cloud.google.com/container-registry

[24] Що таке тошнота тексту [Електронний ресурс] - Режим доступу до ресурсу: https://serpstat.com/ru/blog/chto-takoe-toshnota-teksta/

[25] WebspellChecker [Електронний ресурс] - Режим доступу до ресурсу: https://docs.webspellchecker.net/display/WebSpellCheckerCloud

Додатки

Додаток А

Текст, який був використаний для перевірки існуючих систем на визначення плагіату

```
Хмарні технології – це тренд 2021 року, який раніше не був таким популярним. Воно й зрозуміло, тому що за останні два роки
тенденція переходу бізнесів до онлайн різко зросла. Google Cloud Platform — запропонований компанією Google набір хмарних
служб, які виконуються на тій же самій інфраструктурі, яку Google використовує для своїх продуктів призначених для кінцевих
споживачів, таких як Google Search та YouTube

Вам знадобляться послуги Google Cloud Platform у тому випадку, якщо ви захочете створити не просто стандартний сайт, а веб-
додаток, який буде надійним і допомагатиме вашим клієнтам. Наприклад, ви продаєте меблі і хотіли б створити на вашому сайті
додаток, для вирахування розміру шафи, яка підійшла б вашим клієнтам. Створюючи таку інтерактивну програму на вашому сайті, ви
створюєте не тільки якісні та зручні послуги, але й гарний імідж про ваш бренд.

Ви також використовуєте Google Cloud Platform, коли вам потрібна програма з використанням:

Штучного інтелекту;
Великою кількістю даних, яка використовується для аналітики;
Великим сховищем даних;

При цьому ви платите лише за послуги, які були використані. Ви не платите за всю систему відразу, ви сплачуєте лише послуги
хмари. Тепер, щоб краще зрозуміти принцип роботи GCP, давай трохи пірнемо в історію створення цього сервісу.
```

Додаток Б

Оновлений менеджер користувача.

```
class UserManager(BaseUserManager):
def create_user(self, email, password=None):
if not email:
raise ValueError('Users must have an email address')

user = self.model(
email=self.normalize_email(email),
)

user.set_password(password)
user.save(using=self._db)
return user

def create_staffuser(self, email, password):
user = self.create_user(
email,
password=password,
)
user.staff = True
user.save(using=self._db)
return user

def create_superuser(self, email, password):
user = self.create_user(
email,
password=password,
)
user.staff = True
user.admin = True
user.save(using=self._db)
return user
```

Додаток В

Код визначених моделей. Модель користувача:

```
class User(AbstractUser):
    first_name = None
    second_name = None
    email = models.EmailField(
        verbose_name='Email address',
        max_length=255,
        unique=True,
        primary_key=True
    )
    USERNAME_FIELD = 'email'
    required_fields = []
    objects = UserManager()

    def __str__(self):
        return self.email
```

Модель завантаженого документу

```
class Document(models.Model):
    file = models.FileField(
        upload_to='documents',
        storage=get_private_storage(),
        verbose_name='File'
    )
    created_at = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Created at"
    )
```

Перша частина моделі перевірки

```python
class PlagiarismCheck(models.Model):
    STATUS_CHOICES = (
        ("submitted", "Submitted"),
        ("scheduled", "Scheduled"),
        ("done", "Done"),
        ("error", "Error"),
    )
    id = models.UUIDField(
        default=uuid.uuid4,
        primary_key=True,
        unique=True
    )
    text = models.TextField(
        verbose_name="Text"
    )
    created_at = models.DateTimeField(
        auto_now_add=True,
        verbose_name="Created at"
    )
```

Продовження моделі перевірки

```
updated_at = models.DateTimeField(
    auto_now=True,
    verbose_name="Updated at"
)
document = models.OneToOneField(
    Document,
    on_delete=models.SET_NULL,
    null=True, blank=True,
    verbose_name="File document",
    related_name="plagiatarism_check"
)
user = models.ForeignKey(
    User,
    null=True, blank=True,
    on_delete=models.SET_NULL
)
status = models.CharField(
    verbose_name="Status",
    choices=STATUS_CHOICES,
)
result = models.JSONField(
    verbose_name="Result",
)
```

Додаток Г

Реалізація сховища при використанні застосунку у реальних умовах

```
class PrivateMediaStorage(GoogleCloudStorage):
    location = 'private-media'
    default_acl = 'private'
    file_overwrite = False

    def get_signed_put_url(self, name):
        name = self._normalize_name(clean_name(name))
        blob = self.bucket.blob(name)

        return blob.generate_signed_url(
            expiration=datetime.datetime.now() + self.expiration,
            version='v4',
            method='PUT',
            scheme="https",
        )
```

І функція, яка визначає яке сховище обирати, в залежності від наявних налаштувань

```
def get_private_storage(*args, **kwargs):
    store = settings.PRIVATE_STORAGE
    if isinstance(store, Storage):
        return store
    module_name, klass = store.rsplit('.', maxsplit=1)
    module = import_module(module_name)
    return getattr(module, klass)(*args, **kwargs)
```

Додаток Ґ

Налаштування логування застосунку

```
import google.cloud.logging

client = google.cloud.logging.Client()
client.setup_logging()

LOGGING = {
    'handlers': {
        'cloud_logging': {
            'class': 'google.cloud.logging.handlers.CloudLoggingHandler',
            'client': client
        },
        "console": {
            "level": "DEBUG",
            "class": "logging.StreamHandler",
            "formatter": "verbose",
        },

    },
    'loggers': {
        '': {
            'handlers': ['console', 'cloud_logging'],
            'level': 'INFO'
        }
    },
}
```

Додаток Д

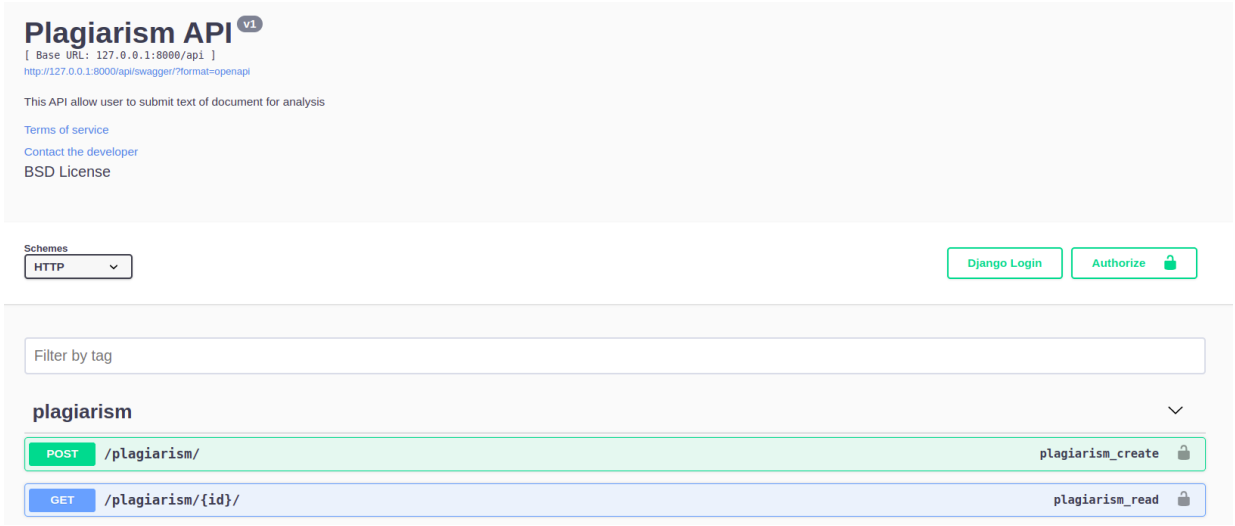
Додавання схеми OpenAPI

```python
from django.contrib import admin
from django.urls import path, include, re_path
from rest_framework_swagger.views import get_swagger_view

schema_view = get_swagger_view(title='Pastebin API')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('plagiarism.urls')),
]


urlpatterns += [
    re_path(r'^api/swagger/$', schema_view, name='schema-swagger-ui'),
]
```



Додаток Е

Функція трансформування документів у текст

```python
def parse_text_from_document(document):
    mime_type, _ = mimetypes.guess_type(document.name)
    if mime_type == "text/plain":
        return document.read().decode('utf-8')
    elif mime_type == "application/pdf":
        pdf = pdftotext.PDF(document)
        result = ""
        for page in pdf:
            result += page
        return result
    raise ValidationError("Unsupported file type")
```

Додаток Є

Файл app.yaml, який визначає яким чином запускається рівень бізнес-логіки

```yaml
runtime: python39

env_variables:
  DJANGO_SETTINGS_MODULE: "config.settings.prod"
  GS_BUCKET_NAME: "plagiarism"
  GCLOUD_PROJECT: "plagiarism-303111"
  FRONTEND_URL: "plagiarism.ua"
  DATABASE_URL: "postgres://plagiarism:plagiarism@10.6.64.5:5432/plagiarism"

handlers:
- url: /.*
  secure: always
  script: auto

vpc_access_connector:
  "projects/plagiarism-303111/locations/europe-west3/connectors/plagiarism-vpc"
```