НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.91

«До захисту допущено»

В.о. завідувача кафедри

________ Едуард ЖАРІКОВ

«___»_____________2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Архітектурне рішення для програмного забезпечення повнотекстового пошуку у відсканованих документах на основі методів машинного навчання»

Виконав:
студент ІІ курсу, групи ІТ-з03мп
Колпак Максим Віталійович ___________

Керівник:
доцент кафедри ІПІ, к.т.н.
Мажара Ольга Олександрівна ___________

Рецензент:
доцент кафедри АПЕПС, к.т.н., доц.,
Шаповалова Світлана Ігорівна ___________

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_______ Едуард ЖАРІКОВ
«___»____________2021р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Колпак Максим Віталійович

1. Тема дисертації «Архітектурне рішення для програмного забезпечення повнотекстового пошуку у відсканованих документах на основі методів машинного навчання», науковий керівник дисертації Мажара Ольга Олександрівна, старший викладач кафедри ІПІ, к.т.н., затверджені наказом по університету від «25» жовтня 2021 р. № 3575-с

2. Термін подання студентом дисертації «6» грудня 2021 р.

3. Об'єкт дослідження – програмне забезпечення електронного документообігу

4. Предмет дослідження – архітектура програмного забезпечення текстового пошуку в системах електронного документообігу

5. Перелік завдань, які потрібно розробити – дослідження наявних підходів до реалізації повнотекстового пошуку в системах електронного документообігу, аналіз існуючих алгоритмів конвертації зображення в текстовий формат, розробка архітектури для реалізації підсистеми конвертації текстового зображення та пошуку в ньому, створення сервісу повнотекстового пошуку для системи електронного документообігу з використанням машинного розпізнавання тексту на основі запропонованої архітектури.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – відсутні.

7. Орієнтовний перелік публікацій – Міжнародна наукова інтернет-конференція «Інформаційне суспільство: технологічні, економічні та технічні спекти становлення (випуск 62)» (Тернопіль, 2021);

Всеукраїнська науково-практична конференція молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології»(SoftTech-2021) (Київ, 2021).

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9.Дата видачі завдання «30» вересня 2020 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Аналіз предметної області	15.09.2021	
2	Аналіз літератури	1.10.2021	
3	Проектування архітектури	20.10.2021	
4	Виконання експериментальних досліджень	23.10.2021	
5	Розробка модуля пошуку	1.11.2021	
6	Розробка модуля розпізнавання	10.11.2021	
7	Інтеграція мікросервісу	13.11.2021	
8	Оформлення пояснювальної записки	21.11.2021	
9	Подання дисертації на попередній захист	22.11.2021	
10	Подання дисертації на захист	6.12.2021	

Студент Максим КОЛПАК

Науковий керівник Ольга МАЖАРА

РЕФЕРАТ

Розмір пояснювальної записки – 90 аркушів, містить 29 ілюстрацій, 26 таблиць, 3 додатки.

Актуальність теми. У роботі розглянуто проблему в області програмного забезпечення електронного документообігу. Під час їх використання виникає проблема взаємодії цифрових систем з вхідною документацією на паперових носіях: листів, договорів тощо. Зазвичай їх оцифровують за допомогою сканування і в подальшому використовують їх зображення, через що виникає проблема роботи системи зі змістом цих документів, в тому числі для повнотекстового пошуку по ним.

Мета дослідження. Основною метою є розробити архітектури програмної системи для здійснення повнотекстового пошуку в системах електронного документообігу та електронних архівах сканованих документів шляхом машинного розпізнавання тексту.

Об’єкт дослідження: програмне забезпечення електронного документообігу.

Предмет дослідження: архітектура програмного забезпечення повнотекстового пошуку в системах електронного документообігу.

Для реалізації поставленої мети **сформульовані наступні завдання**:

- дослідити наявні підходи до реалізації повнотекстового пошуку в системи електронного документообігу;
- проаналізувати існуючі алгоритми конвертації зображення в текстовий формат, обрати ті, що відповідають вимогам системи;
- розробити архітектуру для реалізації підсистеми конвертації текстового зображення та пошуку в ньому;
- на основі запропонованої архітектури створити сервіс повнотекстового пошуку для системи електронного документообігу з використанням машинного розпізнавання тексту.

Наукова новизна результатів магістерської дисертації полягає в удосконаленні програмного забезпечення електронного документообігу шляхом розробки архітектури, яка надає можливість збільшити кількість підтримуваних форматів документів шляхом використання методів оптичного розпізнавання.

Практичне значення результатів полягає в імплементації розробленої архітектури в розробці сервісу повнотекстового пошуку по сканованим документам та інтеграцію його в систему електронного документообігу

Зв'язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Апробація. Наукові положення дисертації пройшли апробацію на:

Міжнародній наукової інтернет-конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення»;

Першій Всеукраїнській науково-практичній конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології»(SoftTech-2021).

Публікації. Наукові положення дисертації опубліковані в:

Колпак М.В. Повнотекстовий пошук у відсканованих документах в системах електронного документообігу / Колпак М.В. // Збірка тез Міжнародної наукової інтернет-конференції «Інформаційне суспільство: технологічні, економічні та технічні спекти становлення». – 2021. – № 62. – С. 30-32.

Колпак М. В. Архітектура програмного забезпечення повнотекстового пошуку у відсканованих документах в системах електронного документообігу / Колпак М. В. // Перша Всеукраїнська науково-практична конференція молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології» (SoftTech-2021). – Київ. – 2021. – С. 111-113.

Ключові слова: ОПТИЧНЕ РОЗПІЗНАВАННЯ ТЕКСТУ, ПОВНОТЕКСТОВИЙ ПОШУК.

ABSTRACT

Explanatory note size – 90 pages, contains 29 illustrations, 26 tables, 3 applications.

Topicality. Examines the problem in the field of electronic document management software. There is a problem of interaction of digital systems with input documentation on paper: letters, contracts, etc. Usually they are digitized by scanning and then their images are used, which causes a problem of system interaction with the content of these documents, including full-text search.

The aim of the study. The main target is to develsin software system architecture for full-text search in electronic document management systems and electronic archives for scanned documents by machine text recognition.

Object of research: electronic document management software.

Subject of research: full-text search software architecture in electronic document management systems.

To achieve this goal, the **following tasks** were formulated:

- invistigate existing approaches to the implementation of full-text search in electronic document management systems;
- analise existing algorithms for converting images to text format, choose those that meet the requirements of the system;
- develop an architecture for the implementation of the text image conversion and search subsystem;
- create a full-text search service for electronic document management system based on the proposed architecture using machine text recognition.

The **scientific novelty** of the results of the master`s dissertation is to improving the software of electronic document management by developing an architecture that provides an opportunity to increase the number of supported document formats through using of optical recognition methods.

The **practical value** of the obtained results is implementation of the designed architecture in the development of a full-text search service for scanned documents and its integration into the electronic document management system.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Informatics and Software Engineering of the National Technical University of Ukraine «Kyiv Polytechnic Institute. Igor Sikorsky».

Approbation. The scientific provisions of the dissertation were tested at the:

International Scientific Internet Conference "Information Society: Technological, Economic and Technical Aspects of Formation"

First All-Ukrainian Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies" (SoftTech-2021).

Publications. The scientific provisions of the dissertation published in:

Kolpak M.V. Full-text search in scanned documents for electronic document management systems //Abstracts of International Scientific Internet Conference "Information Society: Technological, Economic and Technical Aspects of Formation".– 2021. – № 62. – С. 30-32.

Kolpak M.V. Full-text search in scanned documents software architecture for electronic document management systems // First All-Ukrainian Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies" (SoftTech-2021). – 2021.

Keywords: OPTICAL TEXT RECOGNITION, FULL-TEXT SEARCH.

ЗМІСТ

ВСТУП ... 9
1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ... 11
1.1 Системи електронного документообігу ... 11
1.1.1 Загальний огляд ... 11
1.1.2 Docvision ... 11
1.1.3 E-Docs ... 12
1.1.4 FossDoc ... 12
1.2 Огляд алгоритмів машинного розпізнавання тексту на графічних зображеннях ... 14
1.2.1 Розпізнавання на основі метрик ... 14
1.2.2 Розпізнавання за допомогою машинного навчання ... 15
1.2.3 Аналіз розглянутих підходів ... 16
1.3 Повнотекстовий пошук ... 17
Висновок до розділу 1 ... 19
2 ПРОЕКТУВАННЯ СИСТЕМИ ... 20
2.1.1 Цільний моноліт ... 20
2.1.2 Шаровий моноліт ... 21
2.1.3 Мікросервіси ... 23
2.1.4 Безсерверна архітектура ... 25
2.2 Детальний огляд мікросервісного підходу в контексті підсистеми, що проектується ... 27
2.3 Структура мікросервісу повнотекстового пошуку по сканованим зображенням ... 31
Висновок до розділу 2 ... 33
3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ... 34
3.1 Обраний технологічний стек ... 34
3.1.1 Мова програмування ... 34
3.1.2 Серверний фреймворк ... 35
3.1.3 Субд ... 36
3.1.4 Стек клієнтської частини ... 37
3.2 Загальний огляд бібліотек для повнотекстового пошуку ... 39
3.2.1 Sphinx ... 39
3.2.2 Apache Lucene ... 39
3.2.3 ElasticSearch ... 40

3.3 Детальний огляд ElasticSearch 42
3.4 Імплементація модуля пошуку 46
3.5 Загальний огляд бібліотек для оптичного розпізнавання тексту 49
3.5.1 Puma.Net 49
3.5.2 Tessnet2 50
3.5.3 Tesseract 50
3.5.4 Microsoft Azure Cognitive Services 51
3.6 Дослідження якості роботи алгоритмів розпізнавання на документах з різними метаданими 52
3.7 Імплементація модуля розпізнавання 56
3.8 Огляд Masstransit 62
3.9 Огляд Hangfire 65
3.10 Імплементація загальної структури мікросервісу 68
Висновки до розділу 3 72
4 РЕЗУЛЬТАТИ 73
4.1 Транспортні контракти мікросервісу 73
4.2 Опис відладочного HTTP API 75
4.3 Демонстрація роботи сервісу 77
4.4 Інструкція з розвертання 80
Висновки до розділу 4 83
5 МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЄКТУ 84
5.1 Опис ідеї проєкту 84
5.2 Технологічний аудит ідеї проєкту 85
5.3 Аналіз ринкових можливостей запуску стартап-проєкту 86
5.4 Розроблення ринкової стратегії проєкту 91
5.5 Розроблення маркетингової програми стартап-проєкту 93
Висновки по розділу 5 96
ЗАГАЛЬНІ ВИСНОВКИ 97
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 99
ДОДАТОК А 102
ДОДАТОК Б 104
ДОДАТОК В 109
ДОДАТОК Г 110

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – програмний інтерфейс додатку

Ендпоїнт – кінцева точка HTTP API

MSSQL – Microsoft SQL Server

БД – база даних

DI - впровадження залежностей

ВСТУП

У сучасному світі через стрімкий розвиток програмного забезпечення з кожним роком зростає рівень цифровізації життєвих та суспільних процесів, що дозволяє значно спростити та удосконалити традиційні підходи до формування та вирішення завдань, досягання результатів[1].

У корпоративній сфері одним з виявленням цих процесів стало усебічне використання систем електронного документообігу. Це відкриває перед компаніями нові можливості, такі як: економія на інфраструктурі для паперового документообігу, збільшення швидкості прийняття рішень, за рахунок автоматизації деяких процесів тощо[2]. Це надає суттєві конкурентні переваги.

Одним з основних невирішених питань цієї сфери є те, що поки усі суб’єкти суспільних відносин повністю не відмовляться від користування традиційними документами, завжди буде існувати проблема взаємодії цифрових систем з вхідною документацією на паперових носіях: листів, договорів тощо[2]. Зазвичай їх оцифровують за допомогою сканування і в подальшому використовують їх зображення, через що виникає проблема взаємодії системи зі змістом цих документів, в тому числі для повнотекстового пошуку по ним.

Відповідно, метою даної роботи є розробка архітектури програмної системи для здійснення повнотекстового пошуку в системах електронного документообігу та електронних архівах сканованих документів шляхом машинного розпізнавання тексту.

Для досягнення мети було поставлено такі завдання:

- дослідити наявні підходи до реалізації повнотекстового пошуку в системи електронного документообігу;
- проаналізувати існуючі алгоритми конвертації зображення в текстовий формат, обрати ті, що відповідають вимогам системи;
- розробити архітектуру для реалізації підсистеми конвертації текстового зображення та пошуку в ньому;

– на основі запропонованої архітектури створити сервіс повнотекстового пошуку для системи електронного документообігу з використанням машинного розпізнавання тексту.

Об’єктом дослідження є програмне забезпечення електронного документообігу.

Предметом дослідження є архітектура програмного забезпечення текстового пошуку в системах електронного документообігу.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Системи електронного документообігу

1.1.1 Загальний огляд

Системи електронного документообігу (СЕД), або Системи збірки документів (document assembly) – це комп'ютерні автоматизовані система, що описують документи у електронному вигляді та визначають їх потоки. Вони дозволяють зберігати, переглядати, редагувати та керувати рухом документів.

Невід'ємною їх частиною є підсистема пошуку, яких виділяють три основних види:

- простий пошук - по відповідності полів документу пошуковому запиту;
- логічний пошук - по полям з використанням логічних операторів для формування результату;
- повнотекстовий пошук - по тілу документів, використовується у деяких електронних архівах, потребує можливості індексації змісту.

1.1.2 Docvision

Docvision – модульна система електронного документообігу. Доступна велика кількість функціональних блоків, поєднанням яких створюється система, що добре задовольняє потреби конкретного клієнта. Також доступні шаблонні комплекти модулів, для задач певного виду, такі як Docvision Архів або Docvision діловодство[3].

Виробником заявлена стійкість до високих навантажень[3].

Переваги:

- архівування, тобто можливість переносити рідковикористовувані документи на окремі дешевші сервера з меншою швидкодійністю;
- широка функціональність;
- висока швидкодія;

- висока надійність;
- наявність підсистеми повнотекстового пошуку.

Недоліки:

- загальна дороговизна системи;
- пропрієтарність;
- повнотекстовий пошук працює лише по текстовим документам;
- опціональний модуль повнотекстового пошуку невиправдано дорогий, ціна ліцензії близько 260 доларів за одного працівника.

1.1.3 E-Docs

E-Docs – одна з українських систем електронного документообігу. Модульна система, доступно декілька шаблонних комплектів[4].

Переваги:

- функціональність;
- інтеграція з офісними додатками;
- користувацький інтерфейс;
- інтеграція СЕС ООВ.

Недоліки:

- закритість системи;
- відсутність можливості роботи зі сканованими документами.

Згідно інформації отриманої з платформи проведення державних закупівель «Prozorro», ця система використовується в деяких українських органах державної влади, а саме комунальним підприємством «Київтеплоенерго»[5].

1.1.4 FossDoc

FossDoc – українська система керування потоками документів. Основною функціональністю є керування циркуляцією документів[6]. Система має гнучку маршрутизацію. Функції маніпуляції зі змістом документів обмежені.

Переваги

– попередній перегляд файлів;
– інтеграція з поштовими клієнтами;
– можливість використання сторонніх баз даних;
– інтеграція СЕС ООВ;
– широка інформаційна підтримка виробника;
– підтримка версіонування записів.

Недоліки:

– низька швидкодія;
– відсутність повнотекстового пошуку;
– пропрієтарність.

1.2 Огляд алгоритмів машинного розпізнавання тексту на графічних зображеннях

Оптичне розпізнавання тексту (OCR) — це машинне переведення зображень друкованого або рукописного тексту в символьні значення для використання в програмному забезпеченні. Воно є досліджуваною проблемою в галузях розпізнавання образів, машинного навчання і комп'ютерного зору.

Наразі є декілька основних методів оптичного розпізнавання тексту:

- за допомогою шаблонів;
- за допомогою метрик;
- за допомогою нейронних мереж.

В комерційних продуктах використовується лише останні два.

1.2.1 Розпізнавання на основі метрик

Цей спосіб оптичного розпізнавання передбачає порівняння символів на зображенні з базою шрифтів, за допомогою вираховування метрики схожості. В цій якості зазвичай виступає метрика Хеммінга, яка чисельно показує попіксельну різницю у зображеннях.

Метрика Хемінга визначається відстанню Хемінга для двох строк з однаковою довжиною та алфавітом[7].

Відстань Хемінга між векторами - кількість позицій, у яких ці вектори мають різне значення. Наприклад для векторів 10011100 та 10111101 відстань Хемінга дорівнює 2.

Результатом розпізнавання виступає символ, найбільш схожий на вхідне зображення.

Переваги:

- висока точність;
- порівняна простота реалізації.

Недоліки:

- значне зниження точності при роботі з незнайомими шрифтами;
- неможливість розпізнавання рукописного тексту.

Незважаючи на те, що загалом розпізнавання зображення за допомогою метрик є сигнатурним методом, для збільшення точності вихідного результату також використовують евристичні елементи:

- групування - частина символів (В, Т) мають вісь симетрії, деякі мають суперсиметрію (О, Н) – тобто симетричні у всіх напрямках. Їх можна розглядати окремо – значно підвищивши точність кожної з груп;
- використання контексту - деякі алгоритми використовують словники для збільшення точності.

1.2.2 Розпізнавання за допомогою машинного навчання

Нейронна мережа – структура зв'язаних елементів, на яких задані функції модифікації сигналу, а також коефіцієнти для налаштування на певний характер роботи[8].

Сигнал, що проходить через мережу, модифікується згідно функцій на її елементах та їх коефіцієнтів, та формує вихідний результат, при цьому всі вихідні нейрони асоційовані з певним символьними значеннями.

Переваги:

- можливість роботи з невідомими шрифтами;
- можливість роботи з рукописним текстом.

Недоліки:

- витрати пам'яті;
- витрати ресурсів;
- необхідність збору дата-сету;
- порівняно низька точність;
- складність реалізації.

1.2.3 Аналіз розглянутих підходів

У ході огляду існуючих алгоритмів було виявлено, що вони мають один над іншим переваги в різних сценаріях використання.

Алгоритми засновані метриках Хемінга, повинні краще справлятися з зображеннями високої якості, тож добре підійдуть для скриншотів та документів високої якості сканування, проте можуть стикнутися зі значними проблемами при розпізнаванні неякісних фотографій або сканів з низьким dpi.

Алгоритми з застосуванням машинного навчання навпаки - краще для зображень низької якості, тож і сфера їх використання протилежна.

Оптимальним рішенням для розроблюваної архітектури буде забезпечити можливість використання обох підходів, в залежності від типу документа, це дасть можливість збільшити точність розпізнання.

Для реалізації такої архітектури необхідно практично дослідити ефективність роботи цих підходів для різних видів документів.

1.3 Повнотекстовий пошук

Повнотекстовий пошук – вид семантичного пошуку, коли об'єктом пошуку є відповідність самого тіла документу або веб-сторінки пошуковому запиту.

Якщо кількість даних невелика, пошуковий механізм може безпосередньо доступатися до тексту документу чи сторінки в кожному запиті. Таку поведінку називають «послідовним скануванням». Зазвичай такі системи використовуються для локального пошуку по файловій системі машини.

Але для великої кількості пошукових запитів, або для значної кількості даних такий підхід показує себе дуже погано. Фактично, його неможливо використати для багатокористувацьгого мережевого пошуку. Тому сучасні алгоритми використовують індексацію даних. Індексатор ділить запропонований текст на токени, які фактично є окремими словами та їх коренями, потім записує місця входження цих токенів до тексту. Їх сукупність і є пошуковим індексом. Потім, на етапі пошуку, система використовує лише індекс, не доступаючись до змісту оригінальних документів[9].

При індексації можуть ігноруватись слова-зв'язки, такі як сполучники та частки, вони мало впливають на результат пошуку. Індексатори часто не враховують форму слова, для різних родів та чисел створюється лише один токен.

Також можуть використовуватися словники синонімів, для ширшого охоплення пошукового запиту.

Існує 2 види пошукових індексів: прямий та обернений.

Прямий індекс являє собою словник, де ключем буде документ, а значенням - список його слів. Його використовують лише для побудови та оновлення інвертованого індексу.

Інвертований індекс – словник, ключами якого є слова, а значеннями – список документів, до якого вони входять. За допомогою інвертованого індексу пошукові алгоритми можуть швидко сформувати список документів або сторінок, що можуть відповідати запиту.

Наступним етапом є ранжування результатів.

Базовим варіантом є просте ранжування за кількістю включень слів з пошукового запиту в документ.

Часто алгоритми розробляють таким чином, щоб документи з високою локалізацією пошукових слів в декількох місцях ранжувались вище, ніж ті, де слова розміщені по тексу рівномірніше.

Також важливим фактором ранжування є порядок слів, для деяких мов такі алгоритми забезпечують вищу точність результату.

Часто враховується популярність документу, так, наприклад пошуковий алгоритм, що використовується Google – PageRank одним з основних параметрів ранжування результату пошуку використовує кількість посилань на веб-сторінку або документ з інших місць.

Сучасні алгоритми вміють ранжувати результати, враховуючи контекст, телеметрію та особистість користувача.

Висновок до розділу 1

В даному розділі було розглянуто та проаналізовано доступні рішення в сфері електронного документообігу. Виявлено, що можливість повнотекстового пошуку по документах присутня не у всіх системах, а пошук по сканованим документам та зображенням – відсутній взагалі. Отже проблема взаємодії систем з фізичними документами не є вирішеною, і розробка відповідної архітектури є актуальною.

Далі було розглянуто основні способи оптичного розпізнавання тексту. Їх аналіз показав наявність переваг у різних алгоритмів, які сильно залежать від сценаріїв використання. Була сформульована вимога до розроблюваної архітектури – вона повинна використовувати переваги різних алгоритмів в залежності від ситуації. Виявлена необхідність в практичному дослідженні ефективності роботи алгоритмів з різними видами документів.

Також, розглянуто основні підходи до здійснення повнотекстового пошуку.

2 ПРОЕКТУВАННЯ СИСТЕМИ

2.1 Огляд архітектурних підходів в розробці корпоративних систем

2.1.1 Цільний моноліт

Програмне забезпечення, що спроектовано з використанням цього архітектурного підходу являє собою одну, цілісну архітектурну сутність. Логіка не розділяється на окремі абстрактні частини. Можливий структурний поділ на елементи одного абстрактного рівня. Такий підхід в першу чергу характерний для недосвідчених розробників.

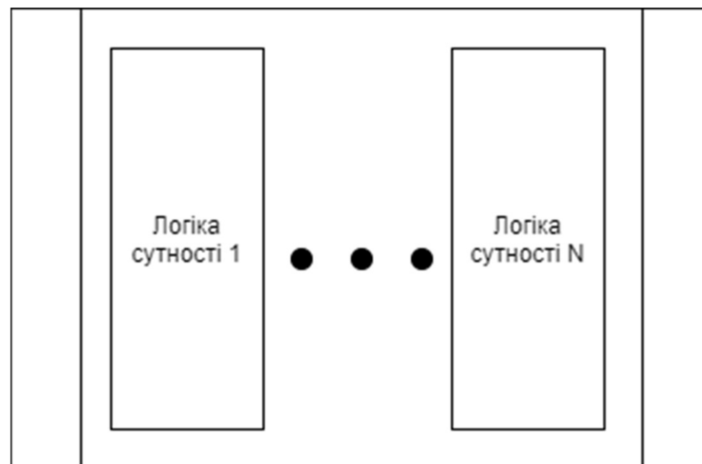

Рисунок 2.1 - Схема монолітної архітектури

Переваги:

- простота, мінімальні вимоги до кваліфікації та навичок програмістів;
- висока швидкість роботи, за рахунок відсутності передавання даних між різними логічними та фізичними архітектурними сутностями;
- неактуальність проблеми значної кількості зайвих рівнів абстракції.

Недоліки:

- проблемність тестування;
- складність пострелізної підтримки;
- заплутаність логіки;
- немасштабованість системи;

– практична неможливість додавання нової функціональності.

Такий архітектурний підхід застарів та вже не використовується в промислових проектах. Використання доцільне лише для малих програм (де використання інших підходів кратно збільшує кодову базу), або з навчальною метою.

2.1.2 Шаровий моноліт

Такий архітектурний підхід описує програму, що складається з різних логічних шарів з різними рівнями абстракції. При цьому, кожен шар зв'язаний лише з двома сусідніми[10]. Класичними варіантами є тришарова (на схемі) та п'ятишарова архітектури.

Перша складається з таких шарів (layers):

- шар доступу до даних - інкапсулює роботу з конкретними джерелами данних надаючи відповідний інтерфейс;
- бізнес-логіка - тут знаходиться сервіси, що містять бізнес-логіку щодо своїх сутностей;
- шар API - містить веб-апі для зв'язку з клієнтською частиною додатку.

П'ятишарова архітектура передбачає додаткові окремі шари маппінгу між різними рівнями абстракції тришарової архітектури[10].

Іноді також виділяють окремий шар для доменної моделі.

Переваги:

- ізольованість шарів, тобто зміна внутрішньої реалізації в межах абстрактного рівня не впливає на роботу інших;
- відносна простота;
- можливість інтеграційного тестування різних шарів;
- простота покриття юніт-тестами, за рахунок можливості створення мок-об'єктів нижчого рівня абстракції;
- надійність;
- розширяємість;

– порівняно висока швидкодійність, через те, що пересилки даних між архітектурними елементами відбуваються лише на логічному рівні;
– прозорість логічної структури.

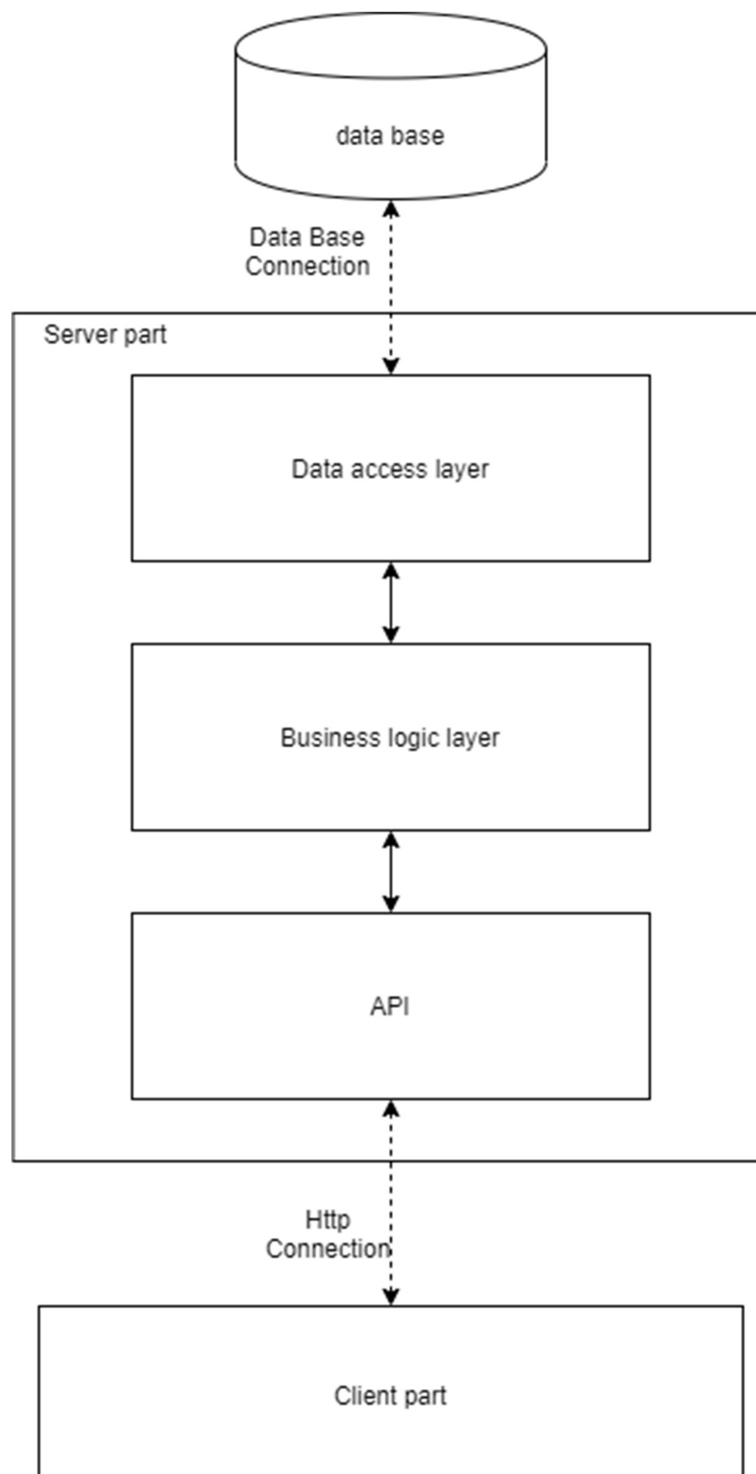

Рисунок 2.2 - Схема веб-додатку з тришаровою серверною архітектурою

Недоліки:

– іноді виникає проблема надмірних абстрактних шарів - кодова база сильно розширяється, знижується швидкість та простота програми;
– необхідність розширення усіх шарів при додаванні нової функціональності, навіть якщо на відповідному рівні не відбувається обробки даних;
– низька масштабованість;
– надмірна кількість інфраструктурного коду;
– порівняно висока зв’язність всередині одного рівня між різними доменними контекстами.

Даний підхід донедавна був надзвичайно поширеним, фактично стандартом для ентерпрайз-розробки. Абсолютна більшість старих програмних продуктів маюсь саме таку архітектуру. Досить активно використовується для нових проектів і зараз.

2.1.3 Мікросервіси

Мікросервісна підхід в архітектурі передбачає систему, що складається з по-суті незалежних програм – мікросервісів. У них є чітке функціональне призначення. Існує багато способів міжсервісної взаємодії. Мікросервіси можуть розгортатися на різних фізичних машинах, часто з використанням контейнерізації[11]. В рамках однієї системи мікросевіси можуть базуватися на повністю різних технологічних стеках.

Рисунок 2.3 - Мікросервісна архітектура

Переваги:

– гнучкість, оскільки окремі частини системи можна повністю переписувати ніяк не взаємодіючи і іншими сервісами, для нової функціональності завжди можна додати новий сервіс;
– висока здатність до горизонтального масштабування;
– простота автоматизованого тестування;
– можливість постійного балансування навантаження між сервісами, оскільки сервіси хостяться незалежно, дуже просто збільшити серверну потужність в навантажених в місцях системи за рахунок вільних ресурсів;
– просте розпаралелювання роботи над системою між різними командами;
– надійність, адже перебої у роботі декількох сервісів не виводять систему з ладу повністю, вона може частково виконувати роботу та відповідати на запити користувачів.

Недоліки:

– висока складність розробки, яка відповідно призводить до високих вимог до кваліфікації команд розробки;
– складність інфраструктури;
– часткова денормалізація даних в реляційних базах;
– проблема необхідності узгодження даних різних сервісів;
– певні втрати в швидкості роботи через міжсервісну пересилку даних;
– складність відтворення та локалізації багів, тому що іноді неможливо точно повторити стан та, можливо, конфігурацію системи у момент виникнення багу;
– складність локального тестування, для якого необхідно розвертати одразу декілька сервісів та частину інфратсруктури на локальній машині, що не завжди можливо;
– велика кількість точок відмови.

Мікросервісна архітектура найкраще себе проявляє у високонавантажених системах, через свою здатність до горизонтального. Останнім часом стала мейнстрімом – дуже часто використовується у нових проектах.

2.1.4 Безсерверна архітектура

Такий архітектурний підхід передбачає повну відмову від власної інфраструктури та передачу виконання свого коду та керування машинними ресурсами на аутсорс, сторонній компанії.

Команда розробників пише лише бізнес логіку в лямбда-функціях, що запускаються в хмархих обчислювальних центрах.

Таку модель називають “функція як послуга” (FaaS - Function as a Service).

Прикладами платформ для розробки додатків на основі безсерверної архітектури є Amazon Web Services Lamda, Google Cloud Functions, Microsoft Azure Functions.

Переваги:

- ціна та швидкість розробки;
- можливість автоматичного масштабування;
- простота системи.

Недоліки:

- відсутність безпеки даних, адже усе розміщується на обладнанні сторонніх компаній;
- низька швидкодійність;
- надзвичайно висока ціна серверної інфраструктури при високих навантаженнях, при цьому витрати можуть зростати значно та несподівано, що призводить до високих фінансових ризиків;
- залежність від сторонніх компаній - якщо власник сервісу FaaS припинить надавання своїх послуг, перевикористати існуючий код буде досить складно, оскільки він дуже платформо-специфічний, також, у хостера з’являються додаткові важелі впливу на редакційну політику щодо контенту в програмному забезпеченні.

Рисунок 2.4 - Безсерверна архітектура

Зараз такий архітектурний підхід підходить лише для програмного забезпечення, де основним фактором для вибору є швидкість розробки та можливість роботи малокваліфікованої команди. Тож часто його використовують стартап-проекти, що намагаються швидко вийти на ринок. Також можливе використання в сценаріях, де високе навантаження неможливе, наприклад в навчальних проектах та окремих, низьконавантажених частинах комплексних систем.

2.2 Детальний огляд мікросервісного підходу в контексті підсистеми, що проектується

Оскільки як розпізнавання тексту на графічних зображеннях, так і повнотекстовий пошук є ресурсозатратними операціями, доцільно винести реалізацію модуля повнотекстового пошуку в сканованих зображеннях в окремий мікросервіс. Це дозволить:

- краще розпоряджатися доступними серверними ресурсами;
- зменшити залежність стабільності роботи цілої системи від підсистеми повнотекстового пошуку;
- розділити вхідні та оброблені (розпізнані та індексовані) документи по різним базам даних;
- спростити інтеграцію модуля у сторонніх системах, за рахунок зменшення зв'язанності.

Основним принципом побудови мікросервісної архітектури є те, що кожен сервіс має мати свої незалежні доменну дані та логіку.

Схожий принцип є в Доменному дизайні (Domain-driven design - DDD), де кожен зв'язаний контекст має мати свою власну доменну модель (сукупність даних і їх поведінки). Кожен зв'язаний контекст з DDD має відповідати одному мікросервісу[11].

На відміну від традиційної монолітної архітектури, де використовується одна база даних, і таблиця кожної сутності містить усі відомі про неї дані, для мікросервісного підходу доцільно використовувати власні бази для кожного сервісу, це дозволить давати доступ лише для даних своєї доменної моделі, що збільшить безпеку даних та збільшить швидкість виконання запитів.

Проте такий підхід робить роботу з данима набагато важчою. Коли доменні моделі різних сервісів містять у собі однакові сутності, бази даних стають денормалізовані і стає необхідне постійне узгодження цих таблиць. Якщо дані модифікуються в одному сервісі – вони повинні модифікуватись в усіх. Надмірна

кількість таких колізій в доменних моделях може свідчити про неправильне розбиття на зв'язані контексти. Узгодження даних відбувається за допомогою обраного способу міжсервісної взаємодії.

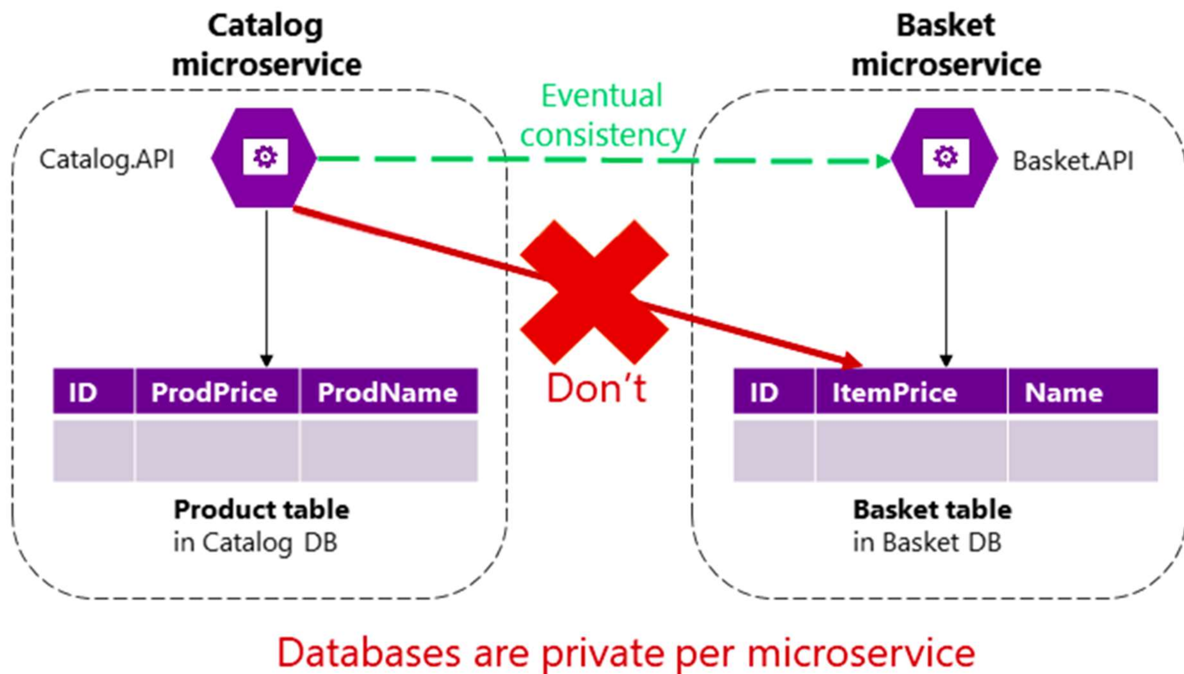

Рисунок 2.5 - Взаємодія зв'язаних контекстів

Іншою проблемою мікросеврісною архітектури є обробка запитів, за зачипають декілька зв'язаних контекстів. Зазвичай для вирішення цього питання використовують API Gateway – це спеціальний сервіс, єдиною метою якого є агрегування запитів від декількох мікросервісів. За можливоті необхідно такого уникати, бо це знижує автономність сервісів да створює єдину точку відмови[11].

Іноді, дані поділяють на «гарячі», що безпосередньо пов'язані з зв'язаним контекстом та зберігаються в мікросервісах, та «холодні» - дані, як потрібні для складних комплексних запитів (наприклад звітів), та зберігаються в централізованій «холодній» базі даних.

Програмне забезпечення створене на основі мікросервісного підходу це розподілені системи, що значить що вони можуть хоститись на різних серверах. Тож дані між ними повинні передаватись через мережеві протоколи, такі як AMQP, HTTP, тощо.

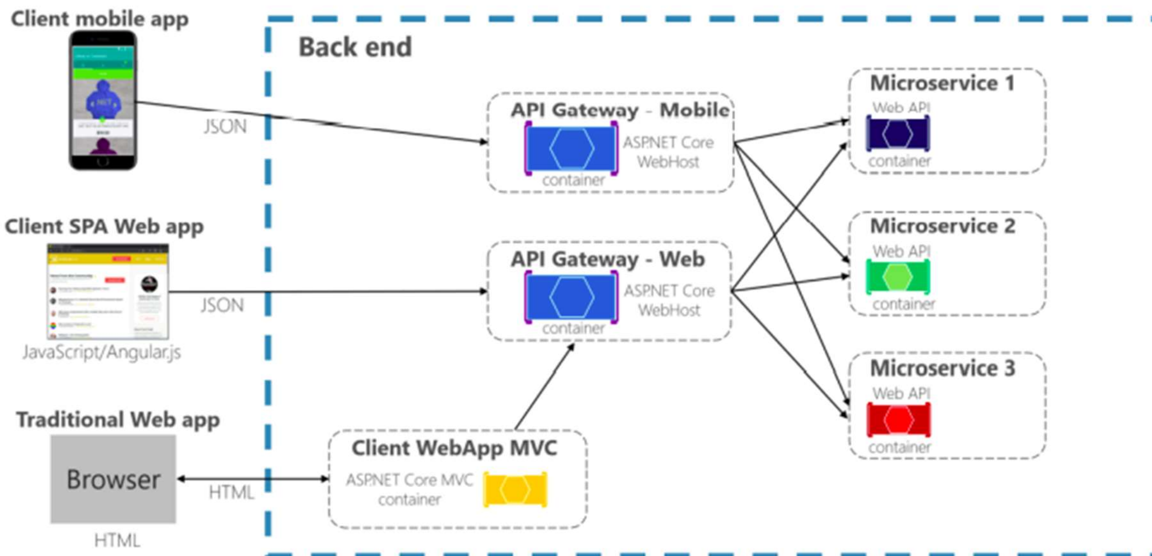

Рисунок 2.6 - Використання API Gateway

Зазвичай використовують один з двох підходів:

– синхронний зв'язок, базований на HTTP реквестах та респонзах;
– асинхронний зв'язок, базований на шинах обміну повідомленнями, такими.

Перший погано підходить для сервісів з ресурсомістними операціями, тому що потік викликаючого сервісу блокуєтья до отримання результату.

Другий – складніший в реалізації та вимогливіший до інфраструктури.

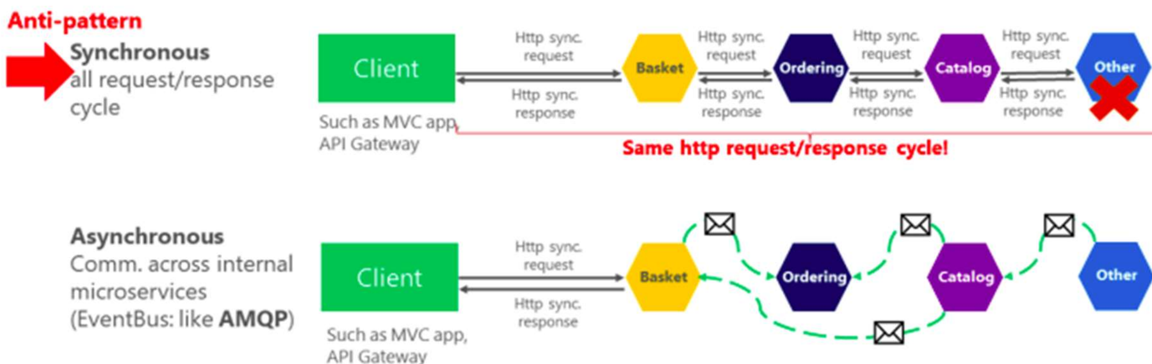

Рисунок 2.7 - Типи зв'язків

Асинхронні повідомлення бувають двох видів:

- команди – мають одного адресата;
- події – мають декілька адресатів.

Необхідність використання того чи іншого виду повідомлення в першу чергу залежить від контексту, події доцільно використовувати для модифікації спільної сутності у декількох сервісах, команди – для конкретного запиту.

2.3 Структура мікросервісу повнотекстового пошуку по сканованим зображенням

Запропонована архітектура передбачає створення окремого мікросервісу повнотекстового пошуку по сканованих зображеннях. Зв’язок з основної частини системи електронного документообігу з мікросервісом здійснюється за допомогою транспортної шини, у якості якої може виступати наприклад RabbitMQ або Azure Service Bus. Сам сервіс складається з ізольованих модулів пошуку та розпізнавання[12].

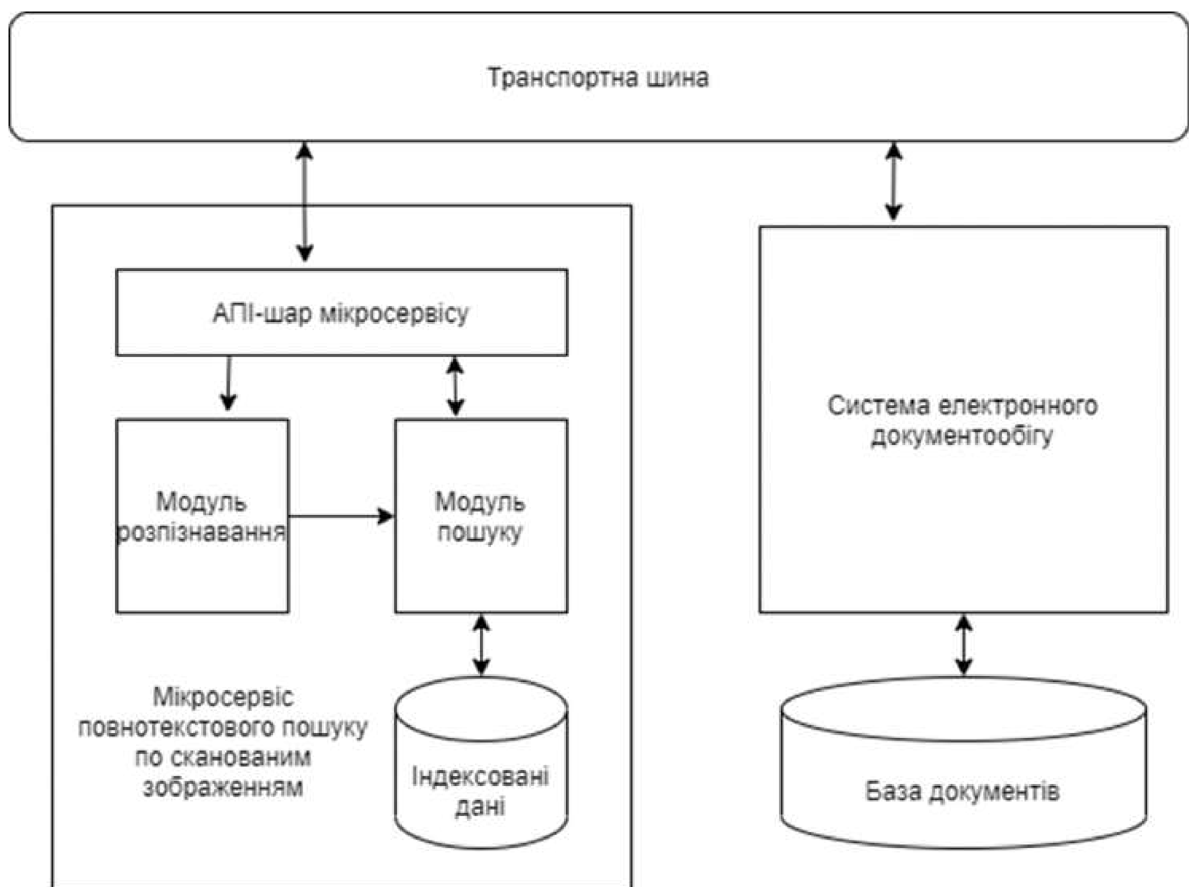

Рисунок 2.8 - Запропонована архітектура

Модуль пошуку містить базу даних з індексованими розпізнаними документами.

Модуль розпізнавання приймає завантажене в систему графічне зображення та видає текстовий зміст документу на виході, передаючи його в модуль пошуку для подальшої індексації[13].

Важливим нововведенням є те, що за такої архітектури може бути імплементовано декілька субмодулів розпізнавання, що реалізують одне АПІ, але використовують різні алгоритми; в такому випадку можна реалізувати динамічний вибір відповідного модуля в аналітичному блоці, в залежності від його ефективності роботи з необхідним форматом документу, що дозволяє збільшити загальну точність розпізнавання.

Для імплементації аналітичного блоку, необхідно дослідити ефективність роботи різних засобів оптичного розпізнавання тексту на графічних зображеннях з різними форматами. В подальшому можна модифікувати цей блок, додаючи інші критерії вибору, такі як: тип документу, об’єм тощо.

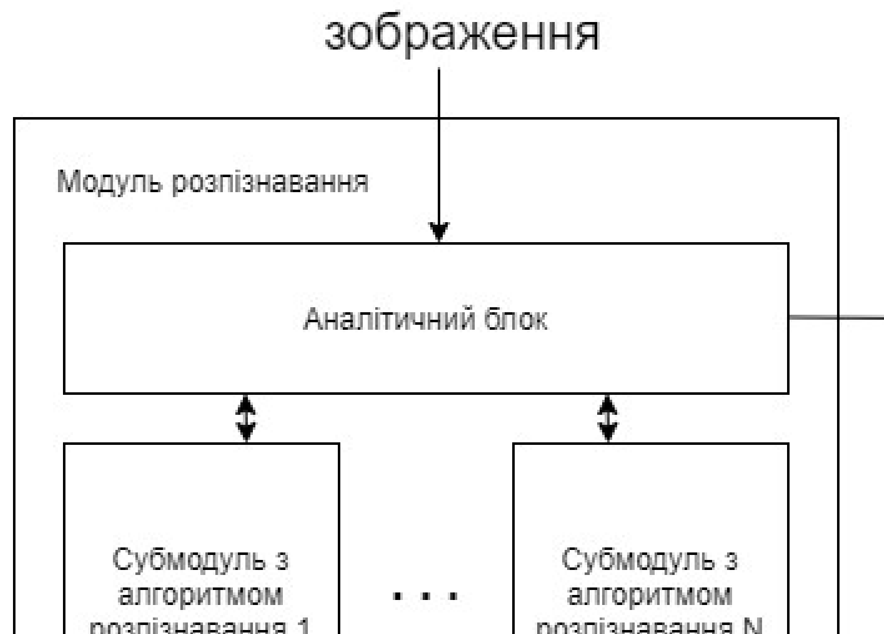

Рис. 2.9 - Структура блоку розпізнавання

Таким чином, збереження текстового представлення в окремій базі даних дозволяє уникнути повторного розпізнавання. Крім того, отримані в результаті розпізнавання документи можуть в подальшому бути використані як базис для створення текстової версії[12].

Висновок до розділу 2

В даному розділі розглянуто та проаналізовано існуючі підходи до побудови архітектури програмної системи. Виокремлено переваги та недоліки для кожної. Обрано мікросервісний підхід, через доцільність мати можливість інтеграції програмного забезпечення, що буде реалізувати архітектуру, що розробляється, з існуючими системами.

Детально оглянуто прийоми та практики проектування програмного забезпечення в рамках мікросервісного підходу.

Розроблено структуру мікросервісу, що передбачає можливість використання різних алгоритмів розпізнавання з динамічним вибором.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Обраний технологічний стек

3.1.1 Мова програмування

Для практичної реалізації модуля повнотекстового пошуку по оптичним зображенням було обрано мову C#.

C# - сучасна об'єктно-орієнтована мова з строгою статичною типізацією. Розроблена в 2000 році компанією Майкрософт. Працює на основі платформи .Net.

Мова має місткий та лаконічний синтаксис, який має багато «синтаксичного цукру» та зручний у використання. Це вигідно виділяє C# з поміж подібних строго типізованих об'єктно-орієнтованих мов. Так, наприклад, механізм властивостей (property) дозволяє працювати з методами встановлення та отримання поля як з самим полем, що дає можливість позбутися зайвих геттерів та сеттерів для усіх приватних полів класу, що призводить до зменшення кодової бази та збільшення охайності коду.

Однією з особливостей є автоматична збірка сміття, яка передбачає 3 покоління об'єктів, на відміну від більшості реалізацій. Перше покоління – новостворені об'єкти, друге – ті, що пережили одну збірку сміття, третє – ті, що пережили більше одної. Наявність третього покоління дозволяє заощадити ресурси системи.

Ще однією особливістю є явне розділення перевантаження та перекриття методів в класах-нащадках та відповідно вимога на явне указання віртуальних методів.

Також, C# підтримує асинхронне програмування, через зручний механізм асинхронних методів та завдань (Task).

.Net – платформа, що включає в себе стандартну бібліотеку типів та спеціального середовища для роботи - common language runtime (CLR). CLR дозволяє використовувати різні мови платформи .Net в рамках однієї програми. Для

цього, вихідниий код спочатку транслюється в «проміжну мову» (intermediate language), яка потім потім компілюється на льоту на машині користувача за допомогою JIT (just-in-time) компілятора.

Розвиток платформи йде у бік кросплатформенності та відкритого програмного забезпечення. Так, наприклад, починаючи з шостої версії, мова C# працює на відкритому новому компіляторі Roslyn, який надає широке публічне апі для аналізу та взаємодії з кодом, що значно покращило інструменти розробки, як самої компанії Microsoft (Visual Studio), так і сторонніх виробників (наприклад JetBrains Rider та ReSharper).

Ще однією тенденцією в еволюції мови C# є поступовий ухил в функціональну парадигму, так, з кожним оновленням C# отримує нові елементи для функціонального програмування, такі як Pattern Matching, або монади Option та Either, зазвичай переймаючи їх у іншої мови платформи .Net - F#.

3.1.2 Серверний фреймворк

У якості серверного фреймворку використаний ASP.NET Core.

Він працює на відкритій незалежній версії платформи .Net - .Net Core. Вона має повністю відкритий код та кросс-платформеність, що дозволяє запускати додатки на різних операційних системах, а також використовувати засоби контейнеризації (Docker) при розвертуванні додатку.

ASP.NET Core являє собою повний редизайн фреймворку ASP.NET – попереднього широковикористаного серверного рішення Майкрософту.

Фреймворк побудований як сукупність легковісних незалежних компонентів, які можна додавати за допомого вбудованого пакетного менеджера NuGet.

Також, ASP.NET Core має вбудований механізм впровадження залежностей, що дозволяє значно знизити зв’язаність проекту без використання сторонніх важких DI-контейнерів, таких як Ninject.

Загалом було значно полегшено конфігурування проекту та керування залежностями в цілому.

ASP.NET Core працює на основі нового конвеєра HTTP-запитів, який має дуже високу швидкодійність да розширюємість (за допомогою власних middleware компонентів).

Ще одним нововведенням було покращення клієнтського фреймворка Blazor та мови розмітки Razor, що дозволило зручно розробляти прості фронт-енди на мові C#.

Тепер стало можливо розміщувати додатки не лише на IIS, але і на сторонніх ваб-серверах, таких як Kestrel.

3.1.3 Субд

У якості системи управління базою даних використано Microsoft SQL Server – реляційна СУБД компанії Майкрософт.

Для запитів використовується мова Transact-SQL (скорочено T-SQL), яка є реалізацією структурованої мови запитів (SQL) із розширеннями. Ця мова додатково підтримує механізми транзакцій та має доповнений синтаксис збережених процедур.

Серверні фреймворки Майкрософта мають вбудовані коннектори до Microsoft SQL Server, що полегшує інтеграцію та дозволяє відмовитись від деяких сторонніх бібліотек.

Переваги:

- швидкодія - за словами компанії Майкрософт, ця СУБД – найшвидша в індустрії, має можливості кластеризації;
- надійність - має широкі можливості резервного копіювання даних;
- безпечність - має вбудовані механізми шифрування;
- простота використання.

Наразі Microsoft SQL Server кроссплатформенна, та має версії для Windows та Linux.

3.1.4 Стек клієнтської частини

Для імплементації клієнтської частини додатку використовається веб-фреймворк Angular та мова TypeScript.

TypeScript – типізована версія мови JavaScript, що розроблена компанією Майкрософт.

Має повноцінну підтримку системи класів, як в інших об'єктно-орієнтованих мовах.

Строга типізація значно полегшує розробку та відладку програми порівняно з звичайним JavaScript, дозволяючи виявляти більшість помилок відразу, ще до виконання, а також використовувати більш розширену підсвітку синтаксису в середовищі розробки.

Серед інших переваг можна відмітити можливість упевнитися в коректності вхідних даних, за допомогою типізації вхідних та вихідних параметрів, а також обмежити область видимості членів класу, за допомогою модифікаторів доступу.

Дуже добре підходить для комплексних веб-додатків.

Код компілюється в чистий JavaScript.

Angular – відкритий фронтенд фреймворк розроблений компанією Google, являє собою покращення та переосмислення попереднього рішення – AngularJs. На відміну від останнього, Angular використовує мову TypeScript, що значно збільшує його функціональні можливості.

Основними концепціями є використання незалежних компонент та двохстороннє зв'язування, що дозволяє при зміні даних в їх джерелі, автоматично змінювати їх на виході (в дочірньому компоненті або розмітці), і навпаки.

Також, важливою особливістю є створення кастомних html-тегів та зміна роботи існуючих за допомогою директив, що дозволяє спростити розмітку.

Важливою перевагою веб-додатків на Angular є ізольваність різних шарів абстракції та сутностей. Так бізнес-логіка ізольвана відкомпонентної, компонентна логіка – від розмітки, а розмітка – від стилів.

Angular надає найбільшу швидкість розробки додатку для досвідчених розробників при збереженні чистоти архітектури.

Серед недоліків фреймворку виділяють дуже великий об'єм вихідного коду, в зв'язку з чим він зовсім не підходить для маленьких додатків, лише для комплексних рішень.

Також, неприємною особливістю є те, що при виходах оновлень часто відсутня зворотна сумісність, що заставляє переписувати частини додатку при переході на нові версії.

Зазвичай Ангулар вважається складним в опануванні, проте підходи, що тут використовуються (об'єктна парадигма, впровадження залежностей, часте використання паттернів проектування, тощо) багато в чому нагадують підходи у розробці серверних додатків на об'єктно-орієнтованих статично-типізованих мовах, тож використання цього фреймворку добре підходить для фулслек розробників з переважним досвідом на бек-енді.

3.2 Загальний огляд бібліотек для повнотекстового пошуку

3.2.1 Sphinx

Sphinx – повнотекстова пошукова система створена у 2001 році. Одне з найперших подібних рішень на нашому ринку.

Її особливістю є збереження індексованих даних в реляційній СУБД.

Переваги:

- висока швидкість індексації;
- низькі вимоги до пам’яті;
- висока швидкість пошуку;
- невибагливість до платформи;
- підтримка різних релятивних баз даних.

Недоліки:

- відсутність розподілених індексів;
- значне зниження ефективності при великих об’ємах даних;
- відсутність підтримки .Net;
- закритість коду в нових версіях.

Загалом це рішення було популярним раніше. 10 років тому середні об’єми даних, які необхідно було індексувати та зберігати були значно нижчими, тому нерозподіленого індексу вистачало, щоб забезпечети необхідні потреби.

Використання даної технології в рамках розроблюваного модуля є недоцільним, основна причина – відсутність нативної підтримки обраного технологічного стеку.

3.2.2 Apache Lucene

Lucene – бібліотека повнотекстового пошуку, створена в 1999 році. З 2005 року розробляється в межах компанії Apache.

Має відкритий програмний код.

Переваги:

- широкі можливості пошукових фільтрів;
- низькі витрати дискової пам’яті на сховище;
- вище точність пошуку;
- можливість використання лексичних аналізаторів для пошуку по неповній відповідності аналізаторів.

Недоліки:

- нижча швидкість індексування;
- неможливість пошуку в реальному часі.

3.2.3 ElasticSearch

ElasticSearch – сучасний сервер повнотекстового пошуку, який з’явився як надбудова над бібліолекою Apache Lucene в 2010 році.

Має відкритий програмний код.

Особливістю цього рішення є збереження індексованих файлів у власній NoSql-подібній базі даних та підтримка безсхемних документів[14].

Також важливою особливістю є широка кластеризація та підтримка розподілених індексів

Переваги:

- гнучкість;
- висока пошукова швидкість;
- розподіленість системи;
- підтримка мультитенантності (використання одних і тих же ресурсів декількома користувачами);
- надає всю функціональність Lucene через зручний інтерфейс;
- має підтримку усіх популярних технологічних стеків;
- широка функціональність пошукових фільтрів.

Недоліки:

- високі вимоги до ресурсів системи;
- складність внутрішньої мови запитів QueryDSL;

– відсутність вбудованої системи авторизації – ці функції перекладені на інфраструктуру.

Крім цього, Elastic природньо наслідує переваги та недоліки Lucene.

Наразі є найпопулярнішим рушієм повнотекстового пошуку.

Серед розглянутих систем має найбільші можливості у пошуку з неповною відповідністю, що робить її дуже підходящим вибором для використання в розроблюваній системі.

3.3 Детальний огляд ElasticSearch

Основною одиницею збереження інформації в ElasticSearch є документ – сутність, що складається з полів з іменами та значеннями, по суті є JSON обєктом.

Документи зберігаються в одному або декількох індексах. Перед збереженням, документи проходять процес маппінгу, до якого входить фільтрація даних (вибірка лише певних полів, або навіть певних даних всередині поля) та токенізація.

Кожен фізичний сервер ElasticSearch являє собою один вузол (ноду). Сукупність декількох вузлів називається кластером. Багатонодові кластери потрібні для збільшення надійності системи, керування великими об’ємами даних, балансування навантаження, тощо.

При збереженні даних, ElasticSearch поділяє їх на декілька індексів Apache Lucene, які називаються фрагментами (шардами). При цьому, ElasticSearch автоматино керує їх комбінуванням, тож для користувача з точки зору АПІ сукупність фрагментів виглядає як одне суцільне сховище[15].

В усіх вузлах кластеру зберігаються копії оригінального фрагменту, які називаються репліками. Це дозволяє зменшити навантаження на кожен вузол, а також збільшити надійність даних, бо початковий фрагмент може буть відновлений з репліки у разі втрати або пошкодження частини інформації[15].

При запуску нового серверу ElasticSearch, він розсилає запити по елементам мережі для того, щоб знайти інші вузли даного кластера та приєднатися до нього, цей процес називається мультикастом.

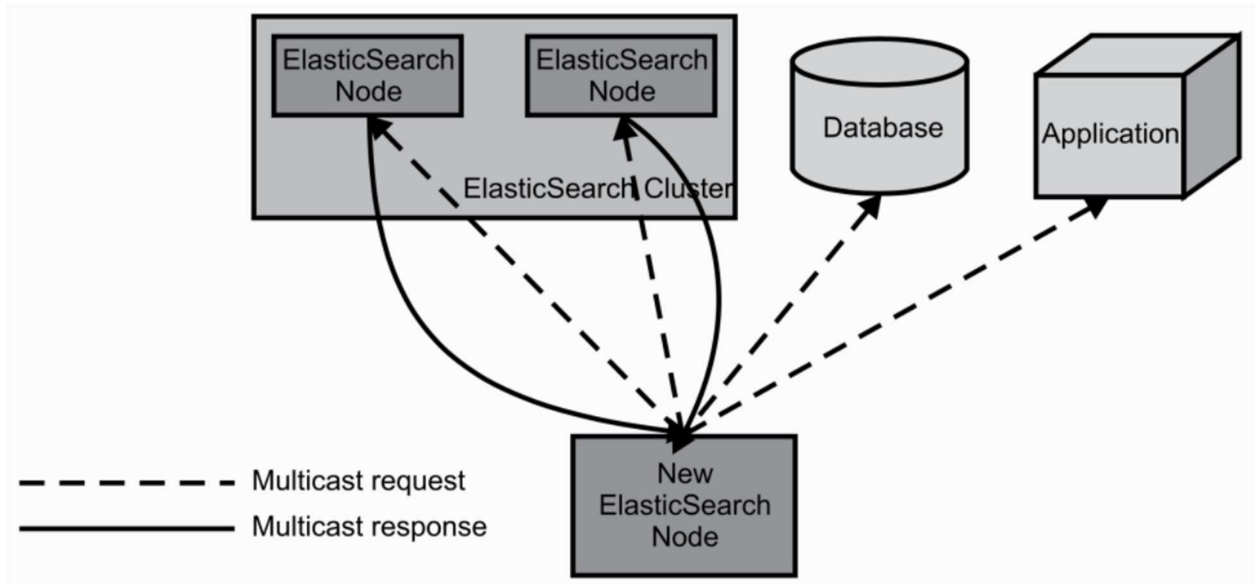

Рисунок 3.1 - Схема мультикасту

У кожному кластері обирається одна головна (мастер) нода. Вона стає відповідальною за керування станом усього кластеру. Так, наприклад, мастер нода постійно сканує усі вузли кластеру, і у разі втрати зв’язку з якимось вузлом переназначає роль первинних фрагментів, які на ньому зберігались, на одну з реплік, забезпечуючи таким чином стійкість тистеми та її роботоспроможність у разі втрати частини мережі.

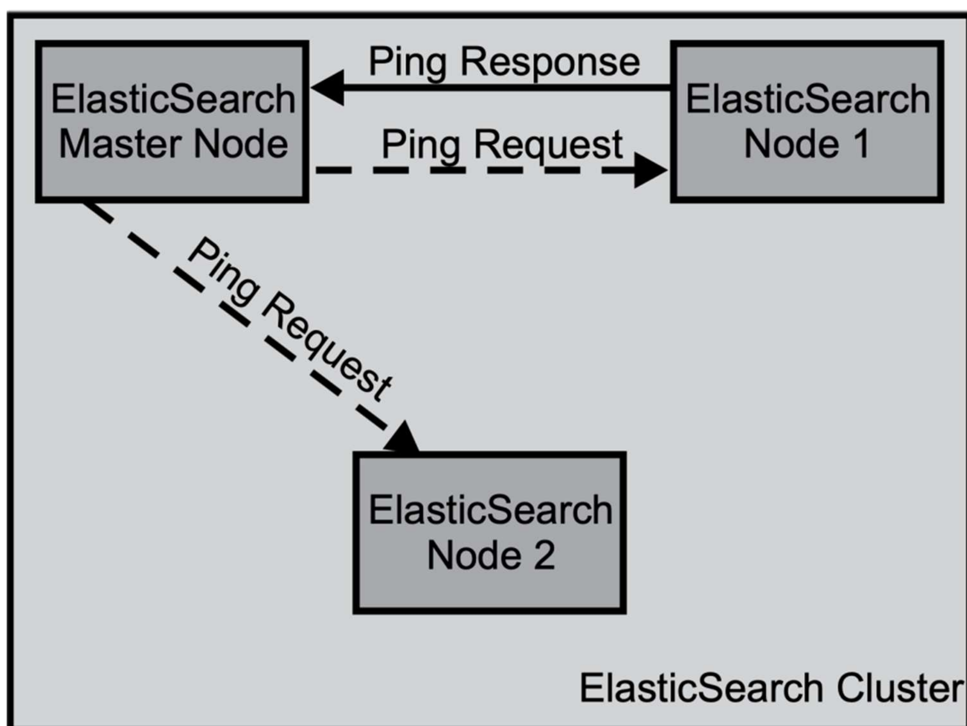

Рисунок 3.2 - Опитування вузлів кластеру

Необхідно зазначити, що будь які запису, такі як індексування або реіндексування відбуваються лише в первинному фрагменті – репліки відтворюються лише через його копіювання і використовуються лише для обробки пошукових запитів[15].

Якщо в вузол, в якому зберігається лише репліка відповідного фрагменту поступає запит на індексацію даних, то цей запит перенаправляється на вузол з первинним фрагментом, як показано на рисунку 3.3.

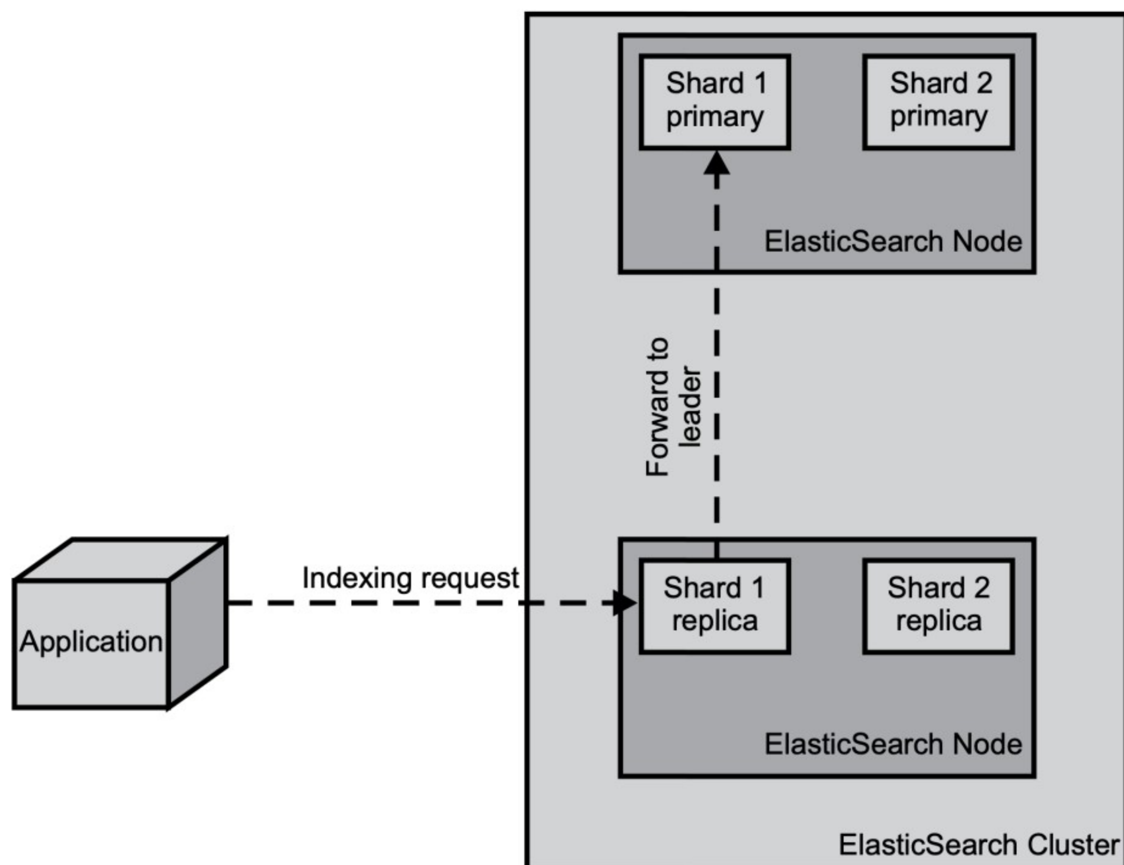

Рисунок 3.3 - Перенаправлення запиту на індексацію

Для пошуку використовується мова Query DSL. Вона базується на Json та дає широкі можливості для створення запитів:

- використання фільтрів з різними типами даних в простих запитах;
- створення складних комплексних запитів комбінуванням декількох простіших;
- виконувати пошук по подібним документам;
- виконувати пошук по неповному співпадінню;

– виконувати «безпечні» запити, ті, виключення з результату якими не впливає на подальше ранжування документа.

Обробка пошукового запиту складається з двох етапів:

– засіювання, під час якого робляться відповідні запити в необхідні фрагменти та репліки;
– збір, під час якого комбінуються результати цих запитів, а також відбувається ранжування документів.

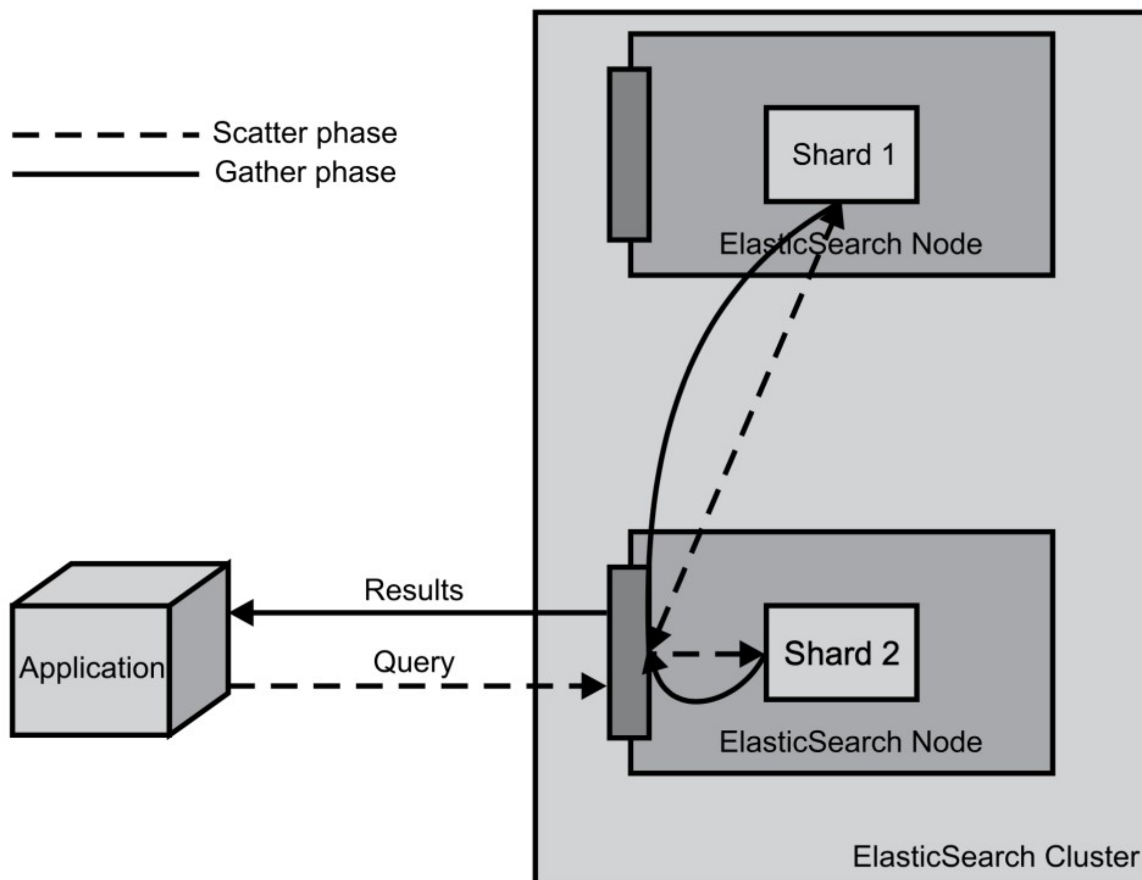

Рисунок 3.4 – Обробка пошукового запиту

3.4 Імплементація модуля пошуку

Модуль пошуку має досить просту структуру. Він складається з таких елементів:

- сервер ElasticSearch, розвернутий та запущений в докер контейнері;
- клієнт, що взаємодіє з цим сервером;
- сервіс, який являється обгорткою для клієнта;
- інтерфейс сервісу, що використовується в інших частинах додатку як залежність.

На рисунку 3.5 представлена схема пошукового модуля.

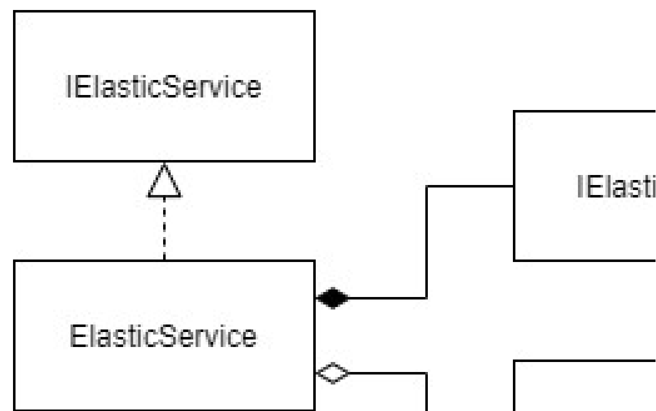

Рисунок 3.5 - Схема модуля пошуку

Перш за все, потрібно створити індекс на сервері ElasticSearch в конфігураційному файлі клієнта:

```
var createIndexResponse = client.Indices.Create(indexName,
    index => index.Map<Document>(x => x
        .AutoMap()
        .Properties(ps => ps
            .Text(s => s
                .Name(n => n.Content)
                .Analyzer("substring_analyzer")
            )
        )
    )
    .Settings(s => s
        .Analysis(a => a
            .Analyzers(analyzer => analyzer
                .Custom("substring_analyzer", analyzerDescriptor =>
analyzerDescriptor.Tokenizer("standard").Filters("lowercase", "substring")))
            .TokenFilters(tf => tf
                .NGram("substring", filterDescriptor =>
filterDescriptor.MinGram(2).MaxGram(15))))));
```

Як бачимо, тут конфігурується прив’язка індексу до доменної моделі, а також додається та налаштовується синтаксичний аналізатор для пошуку по неповному співпадінню.

Лістинг інтерфейсу IElasticService:

```
public interface IElasticService
{
    Task<IEnumerable<SearchResult>> Search(string query, int page = 0, int pageSize = 5);

    Task SaveSingleAsync(Document document);

    Task SaveManyAsync(Document[] documents);

    Task SaveBulkAsync(Document[] documents);

    Task DeleteAsync(Document document);
}
```

Як бачимо, інтерфейс складається з 5 методів:

- метод Search, що використовується для здійснення пошукових запитів, приймає пошукову строку, а також дані про номер та розмір сторінки для пагінації, повертає відповідно пошукові результати;
- методи SaveSingleAsync, SaveManyAsync та SaveBulkAsync, що використовується для індексації документів. Метод SaveSingleAsync індексує один документ, інші одразу декілька. Методи SaveManyAsync та SaveBulkAsync відрізняються способами реалізації – в першому запити на індексацію посилаються почергово для групи документів, в другому запит на індексацію відправляється одразу для усієї групи;
- метод DeleteAsync використовується для видалення документу з індексу.

Інтерфейс використовує моделі Document та SearchResult.

Модель Document описує проекцію сутності документа на контекст пошукового мікросервісу, тож згідно принципів доменної розробки (Domain Driven Development) є доменною моделлю документа.

Містить в собі такі властивості:

- зовнішній ідентифікатор документу;
- заголовок;
- тіло документу.

Модель SearchResult описує результат пошукового запиту.

Містить у собі:

– ідентифікатор знайденого документу

– колекцію «попадань» - відривків тексту документу, що містять слова з пошукової фрази, які додатково виділяються обрамленням з html тегів

Клас ElasticService реалізує вищеописаний інтерфейс, використовуючи залежність у вигляді клієнта ElasticSearch.

Основні задачі сервісу це формування пошукового запиту та перетворення отриманих даних ф формат, що може бути використаним в системі.

Формування запиту відбувається в реалізації методу Search.

Лістинг формування:

```
var response = await _elasticClient.SearchAsync<Document>(s => s
    .From(page * pageSize)
    .Size(pageSize)
    .Query(q => q.QueryString(q => q
        .Fields(fields => fields
            .Field(f => f.Content)
            .Field(f => f.Name))
        .Type(TextQueryType.BestFields)
        .Query(query)))
    .Highlight(h => h
        .Encoder(HighlighterEncoder.Default)
        .Fields(fs => fs
            .Field(p => p.Content)
            .Type("unified")
            .ForceSource()
            .FragmentSize(300)
            .Fragmenter(HighlighterFragmenter.Span)
            .NumberOfFragments(10)
            .NoMatchSize(300)))
    );
```

Як бачимо, тут визначаються поля для пошуку, тип запиту, задаються параметри пагінації, а також конфігурація обрізки знайдених фрагментів та обрамлення слів з пошукової фрази.

Далі, після валідації результату запиту відбувається його приведення в необхідний формат:

```
return response.Hits.Select(h =>
    new SearchResult
    {
        DocumentId = int.Parse(h.Id),
        Hits = h.Highlight.SelectMany(h => h.Value)
});
```

Як результат маємо колекцію моделей SearchResult

Необхідно зазначити, що усі операції – асинхронні.

3.5 Загальний огляд бібліотек для оптичного розпізнавання тексту

3.5.1 Puma.Net

Puma.Net – бібліотека для оптичного розпізнавання тексту, реалізована на основі рушія розпізнавання CuneiForm, розроблена компанією Cognitive Technologies[16]. Puma.Net являє собою обертку з класів, для можливості використання в .Net.

Алгоритм роботи заснований на метриках Хемінга. Для роботи необхідні спеціальні dll з даними про шрифти.

Серед переваг даного рішення можна зазначити можливість збереження структури документа (абзаци, відступи), а також визначення модифікацій шрифтів.

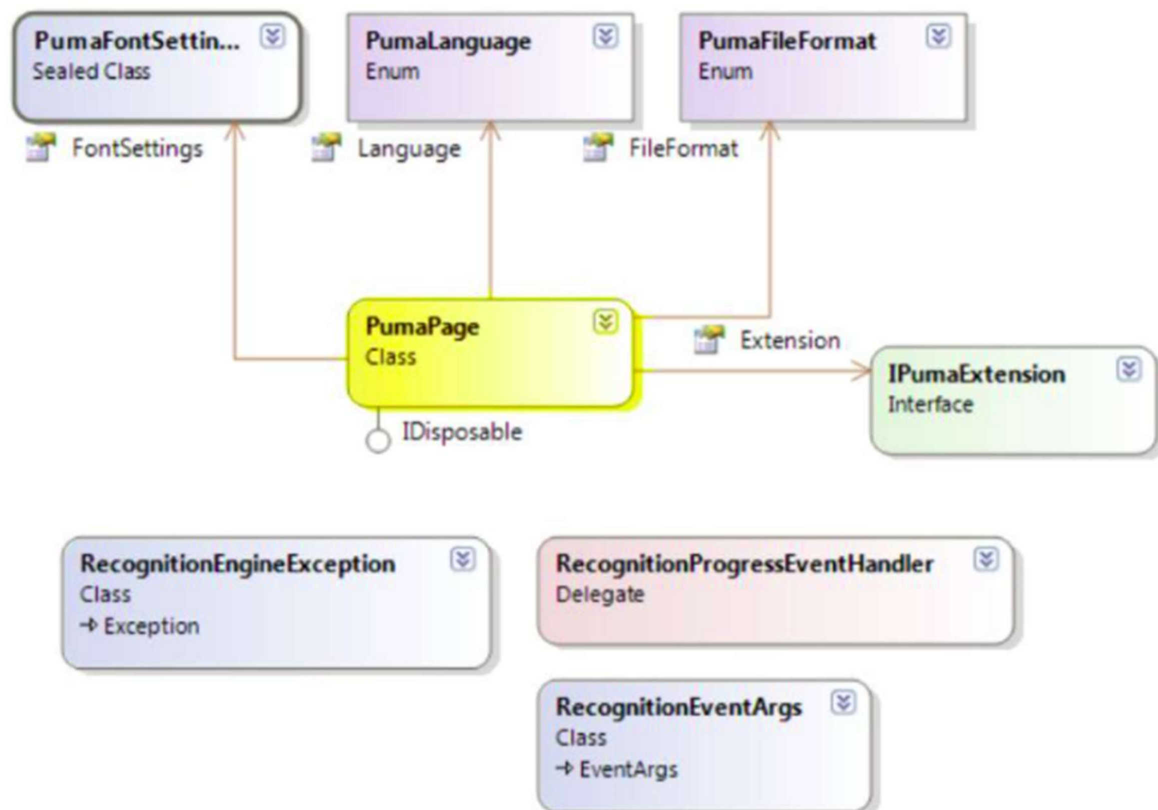

Рисунок 3.6 - Схема бібліотеки Puma.Net

3.5.2 Tessnet2

Бібліотека Tessnet2 – адаптація популярного рушія оптичного розпізнавання Tesseract 2.

Переваги:

- досить висока точність розпізнавання;
- підтримка великої кількості мов;
- підтримка великої кількості шрифтів;
- можливість обмеження алфавіту.

Недоліки:

- проблеми з витоками пам'яті, без перезапуску через певний час роботи програми вона може використати всю оперативну пам'ять, що робить неможливим її використання в веб-сервісах.

3.5.3 Tesseract

Tesseract – нова бібліотека для роботи з одноіменним рушієм розпізнавання. На відміну від вищерозглянутої tessnet2, працює з четвертою (найновішою) версією рушія[17].

Має декілька режимів розпізнавання, кожен з яких має свою власну реалізацію алгоритму:

- режим TesseractOnly - працює на основі власного алгоритму Tesseract заснованого на метриках Хеммінга, фактично – реалізація унаслідувана від попередньої версії рушія (Tesseract 3), теоретично має добре підходити для якісних зображень;
- режим LstmOnly - розпізнавання відбувається за рахунок машинного навчання, для роботи існує декілька різних колекцій файлів тренувань нейронної мережі, які дають різну точність при різних часових затратах, теоретично має краще працювати з нечіткими зображеннями;

- режим TesseractAndLstm - поєднує роботу двох вищезазначених алгоритмів, теоретично має давати найвищу точність при більших часових затратах.

Серед переваг можна зазначити:

- широка підтримка різних мов;
- в новій версії рушія вирішено проблеми з пам’яттю;
- можливість використовувати різні алгоритми в рамках одного рішення;
- новий алгоритм з використанням машинного навчання дає ширшу підтримку шрифтів;
- можна використовувати різні файли з даними тренування нейронних мереж в залежності від потреби.

3.5.4 Microsoft Azure Cognitive Services

Microsoft Azure Cognitive Services – хмарний сервіс компанії Майкрософт, який надає АПІ для використання їх потужностей для аналізу зображень за допомогою машинного навчання[18].

Одним з сервісів є OCR – який надає послуги по оптичному розпізнаванні тексту на зображенні.

Переваги:

- висока точність розпізнавання;
- можливість розпізнавання двомовних документів;
- можливість розпізнавання рукописного тексту;
- оптимізоване розпізнавання тексту на багатосторінкових документах з можливим збереженням структури.

Недоліки:

- висока ціна використання.
- необхідність передавати документи на сервери Майкрософт, що може призвести до проблем з приватністю та безпекою інформації в файлах.

3.6 Дослідження якості роботи алгоритмів розпізнавання на документах з різними метаданими

Для розробки модулю розпізнавання було необхідно провести дослідження ефективності роботи різних алгоритмів розпізнавання на файлах з різними метаданими.

Для цього було створено тестовий стенд у вигляді консольного додатку, який розпізнає задані зображення і надає середній відсоток вірогідності правильності розпізнавання символу по документу.

Метод перевірки:

```
public static float Check(int num, string lang, EngineMode mode)
{
    using (var engine = new TesseractEngine(@"./tessdata", lang, mode))
    {
        using (var img = Pix.LoadFromFile($@"./samples/{num}-{lang}.jpg"))
        {
            using (var page = engine.Process(img))
            {
                return page.GetMeanConfidence();
            }
        }
    }
}
```

Було перевірено декілька наборів файлів різних форматів та мов в трьох режимах роботи бібліотеки Tesseract.

Усього було розпізнано 90 файлів:

- 10 файлів pdf англійською мовою високої якості сканування;
- 10 файлів pdf українською мовою високої якості сканування;
- 10 файлів pdf російською мовою високої якості сканування;
- 10 скриншотів веб-сайтів англійською мовою;
- 10 скриншотів веб-сайтів українською мовою;
- 10 скриншотів веб-сайтів російською мовою;
- 10 файлів DjVu англійською мовою низької якості сканування;
- 10 файлів DjVu українською мовою низької якості сканування;
- 10 файлів DjVu російською мовою низької якості сканування.

Кожен файл розпізнавався 3 рази різними алгоритмами – сумарно 270 розпізнавань.

У результаті було отримано такі дані:

Таблиця 3.1 - Дані середньої вірогідності правильного розпізнавання символів у документі для PDF файлів

Номер файлу	TesseractOnly	LstmOnly	TesseractAndLstm
1	88	93	89
2	89	95	90
3	88	94	89
4	89	95	90
5	89	95	90
6	89	95	91
7	90	94	90
8	87	94	90
9	88	94	91
10	89	94	91

Як бачимо в таблиці 3.1, режим роботи який використовує алгоритм з машинним навчанням показує найкращі результати, алгоритм на метриках Хемінга – нижче на 5-7 відсотків, а комбінований алгоритм, всупереч теоретичним передбачень показує майже такий самий результат, встого на 1-2 відсотки краще. Тож для PDF файлів з високою якістю сканування найкраще підходить режим, що використовує алгоритм з машинним навчанням.

Таблиця 3.2 - Дані середньої вірогідності правильного розпізнавання символів у документі для скриншотів веб-сайтів

Номер файлу	TesseractOnly	LstmOnly	TesseractAndLstm
1	97	94	97
2	95	93	94
3	88	95	90
4	86	95	89
5	88	95	91
6	97	93	97
7	96	94	96
8	96	94	96
9	98	95	97
10	94	93	94

Як бачимо з таблиці 3.2, результати ефективності для скриншотів відрізняються. Ефективність роботи алгоритму на основі машинного навчання приблизно така ж як і для pdf файлів, проте алгоритм на основі метрики Хемінга дає найкращі результати для більшості файлів, і найгірші для декількох інших. Така поведінка, можливо, зумовлена тим, що, незважаючи на те, що зображення високої якості цей алгоритм розпізнає краще, веб-сайти можуть використовувати свої власні шрифти – в такому випадку ефективність значно знижається. Комбнований алгоритм дає середні результати, певним чином згладжуючи провали режиму TesseractOnly.

Таблиця 3.3 - Дані середньої вірогідності правильного розпізнавання символів у документі для DjVu файлів

Номер файлу	TesseractOnly	LstmOnly	TesseractAndLstm
1	80	94	89
2	74	91	83
3	75	89	81
4	74	90	83
5	74	91	83
6	73	92	85
7	79	91	83
8	79	92	84
9	73	91	84
10	73	85	83

Дані з таблиці 3.3 показують, що ефективності розпізнавання різних алгоритмів для DjVu файлів поганої якості співвідносяться між собою співвідносяться так само, як в pdf файлах, але з нижчими абсолютними значеннями.

Необхідно зазначити, що у якості вхідних даних для останнього досліду було взято сторінки книг з технічної літератури, у яких містилося багато схем та формул, що певним чином занизило результати, але це не повинно вплинути на співвідношення ефективності.

Таблиця 3.4 - Середні вірогідності правильного розпізнавання символів

Тип документу	TesseractOnly	LstmOnly	TesseractAndLstm
PDF	88,6	94,3	90,1
Скриншот	93,5	94	94,2
DjVu	75,4	90,6	83,8

В результаті, для сканованих файлів найкраще підходить режим LstmOnly, для скриншотів – TesseractAndLstm.

3.7 Імплементація модуля розпізнавання

Вхідною точкою модуля розпізнавання є інтерфейс IContentProvider.

Лістинг інтерфейсу:

```
public interface IContentProvider
{
    Task<string> GetContentAsync(RecognitionRequest request);
}
```

Як можна побачити, він описує один метод, який використовується для запуску процесу розпізнавання файлу. На виході він повертає строку – розпізнаний текст тіла документу. На вході як аргумент приймає модель RecognitionRequest, у якому міститься дані, необхідні для розпізнавання документу, а саме сам файл та метадані: мова та тип документу.

Клас ContentProvider реалізує інтерфейс IContentProvider, який по суті являє собою аналітичний модуль, у якому визначається конкретний спосіб розпізнавання для даного документу.

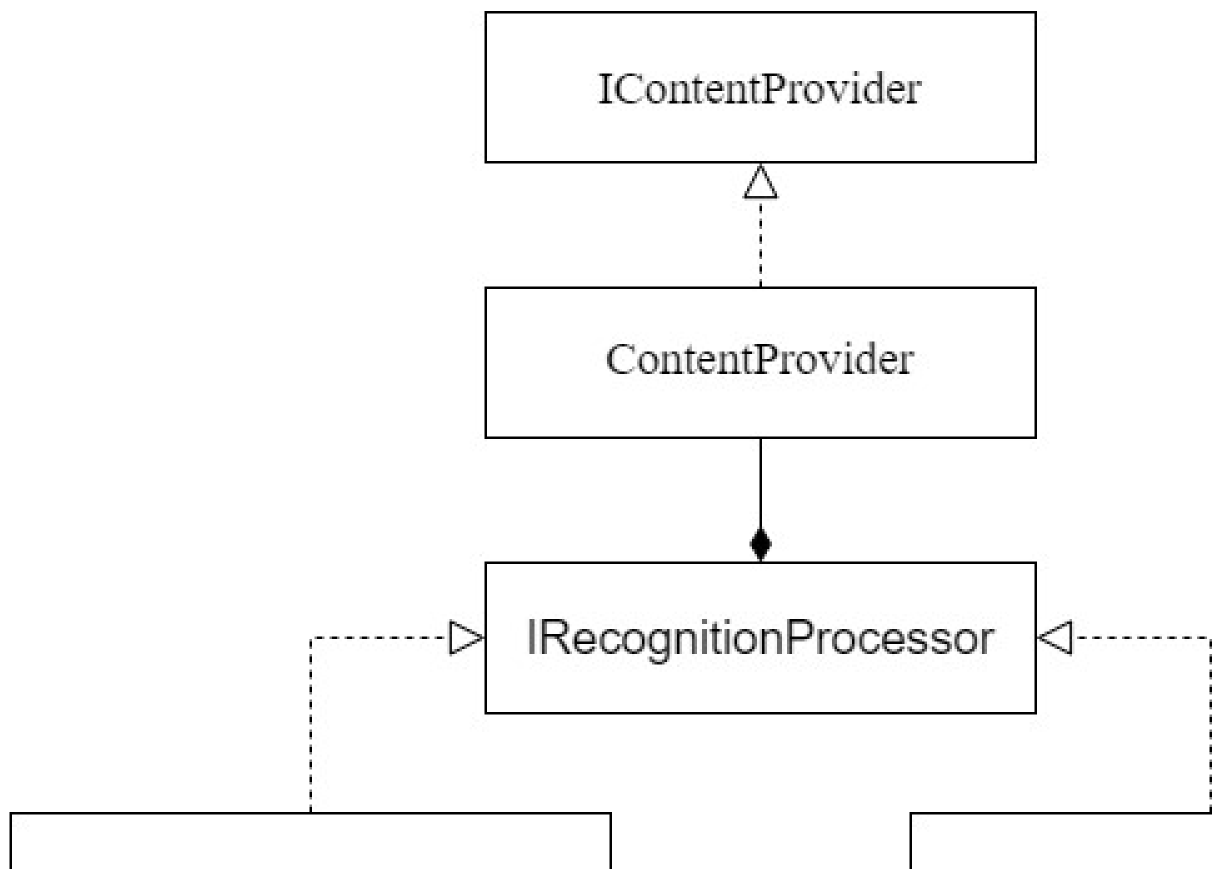

Рисунок 3.7 - Загальна схема модуля розпізнавання

Лістинг реалізації методу GetContentAsync:

```
        public async Task<string> GetContentAsync(RecognitionRequest request) =>
request switch
        {
            { Type: DocumentType.Image } => await
recognitionProcessor.ProcessFile(request.Path, request.Type, request.Language,
Tesseract.EngineMode.TesseractAndLstm),
            { Type: DocumentType.Pdf }  => await
recognitionProcessor.ProcessFile(request.Path, request.Type, request.Language,
Tesseract.EngineMode.LstmOnly),
            { Type: DocumentType.DjVu } => await
recognitionProcessor.ProcessFile(request.Path, request.Type, request.Language,
Tesseract.EngineMode.LstmOnly),
            _ => await recognitionProcessor.ProcessFile(request.Path, request.Type,
request.Language, Tesseract.EngineMode.Default)
        };
```

Як можна зрозуміти зі схеми зображеної на рисунку 3.7 та лістингу, метод GetContentAsync класу ContentProvider визначає конкретну залежність загального типу IRecognitionProcessor та аргументи для запуску.

Логіка роботи базується на дослідженні роботи різних алгоритмів для різних типів документів, описаного вище. Так, для документів з типом Image запускається розпізнавання комбінованим алгоритмом бібліотеки Tesseract, а для типу Pdf та DjVu - алгоритмом базованим на машинному навчанні.

Інтерфейс IRecognitionProcessor – узагальнення різних рушіїв розпізнавання, його реалізація повинна містити інтеграцію з конкретною бібліотекою. В нашій реалізації використано лише бібліотеку Tesseract, бо вона дає можливість одразу розпізнавати різними алгоритмами, проте можлива і бажана інтеграція одразу з декількома бібліотеками, для більш точної спеціалізації на певних метаданих документу.

Лістинг інтерфейсу IRecognitionProcessor:

```
    public interface IRecognitionProcessor
    {
        Task<string> ProcessFile(string path, DocumentType type, Language language);
    }
```

Він описує один метод – ProcessFile, який на вході приймає сам файл, та параметри, які можуть бути необхідні для розпізнавання, результатом його роботи є розпізнаний зміст тіла документу.

Клас TesseractRecognitionProcessor – реалізація інтерфейсу IRecognitionProcessor для бібліотеки Tesseract.

Лістинг реалізації методу ProcessFile:

```
public async Task<string> ProcessFile(
    string path,
    DocumentType type,
    Language language,
    EngineMode mode) =>
        type switch
        {
            DocumentType.Image => await ProcessPage(path, language, mode),
            _ => await ProcessMultiPageDocument(path, type, language, mode)
        };
```

Цей метод – вхідна точка в клас, в ньому передбачене визначення того отриманий файл одно чи багатосторінковий та запуску відповідного метода для його обробки.

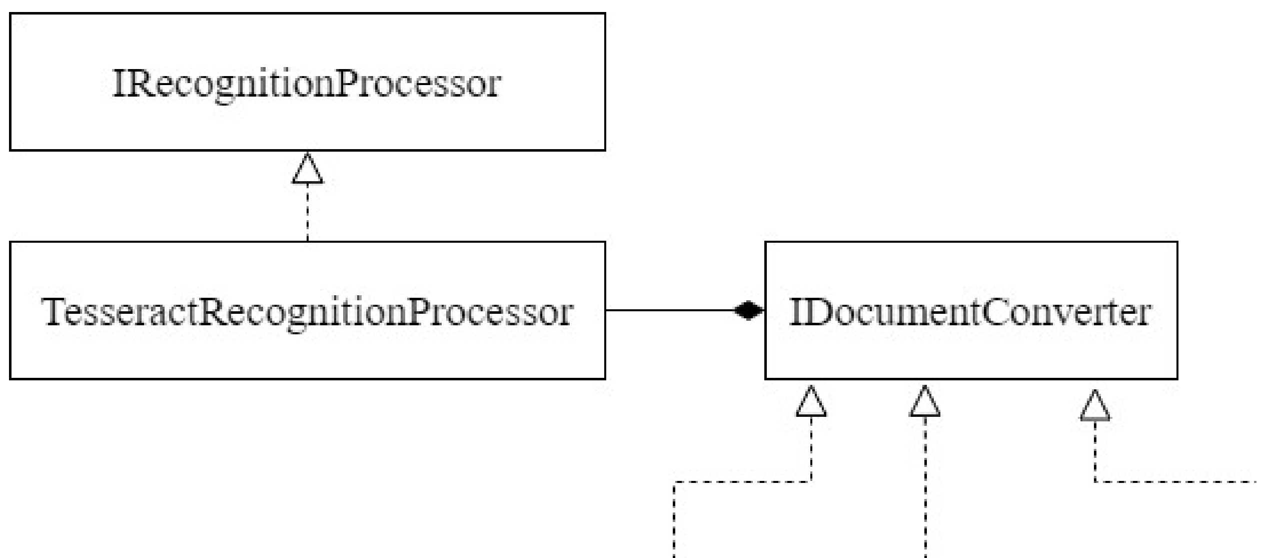

Рисунок 3.8 - Схема реалізації IRecognitionProcessor для бібліотеки Tesseract

Лістинг методу ProcessPage:

```
private Task<string> ProcessPage(string path, Language language, EngineMode mode)
{
    using (var engine = new TesseractEngine(@".", GetLang(language), mode))
    {
        using (var img = Pix.LoadFromFile(path))
        {
            using (var page = engine.Process(img))
            {
                return Task.FromResult(page.GetText());
            }
        }
    }
```

```
                string GetLang(Language language) =>
                    language switch
                    {
                        Language.English => "eng",
                        Language.Ukrainian => "ukr",
                        Language.Russian => "rus"
                    };

            }
```

У цьому методі саме і відбувається розпізнавання односторінкових документів.

У випадку з багатосторінкомими документами – їх спочатку треба розбити на окремі сторінки, а потім уже можна буде скористатися вищеописаним методом розпізнавання однієї сторінки.

Його лістинг:

```
        private async Task<string> ProcessMultiPageDocument(
            string path,
            DocumentType type,
            Language language,
            EngineMode mode)
        {
            var collector = "";
            var converter = GetConverterAsync();
            await converter.SetDocumentAsync(path);
            var pages = await converter.GetPagesAmountAsync();

            for (int i = 0; i < pages; i++)
            {
                collector += await ProcessPage(await converter.GetPageAsync(i),
language, mode);
            }

            await converter.ClearDataAsync();

            return collector;

            IDocumentConverter GetConverterAsync() => type switch
            {
                DocumentType.Pdf => new PdfConverter(),
                _ => throw new ArgumentOutOfRangeException()
            };
        }
```

Як бачимо, метод використовує залежність IDocumentConverter для обробки документу та розбиття його на сторінки, потім ітеративно розпізнає їх складаючи результат в змінну-акумулятор. IDocumentConverter – повинен бути специфічним для кожного формату багатосторінкового документу.

Лістинг інтерфейсу:

```
    public interface IDocumentConverter
    {
        Task<int> GetPagesAmountAsync();

        Task<string> GetPageAsync(int number);

        Task ClearDataAsync();

        Task SetDocumentAsync(string path);
}
```

В ньому описано такі методи:

– SetDocumentAsync - початок роботи з документом;
– GetPagesAmountAsync - отримання загальної кількості сторінок документу;
– GetPageAsync - отримання конкретної сторінки документа по номеру;
– ClearDataAsync - завершення роботи з документом та очистка даних про нього.

Так, наприклад, клас PdfConverter – це реалізація цього інтерфейсу для документів типу Pdf.

В реалізації методу SetDocumentAsync він використовує залежність у вигляді бібліотеки PDFWrapper для обробки pdf-файлу та його розбиття на сторінки, які зберігаються в словнику, де значенням є сторінка, ключем – її номер.

Методи GetPageAsync та GetPagesAmountAsync використовують цей словник для повертання своїх результатів відповідно.

Метод ClearDataAsync – очищає словник зі сторінками.

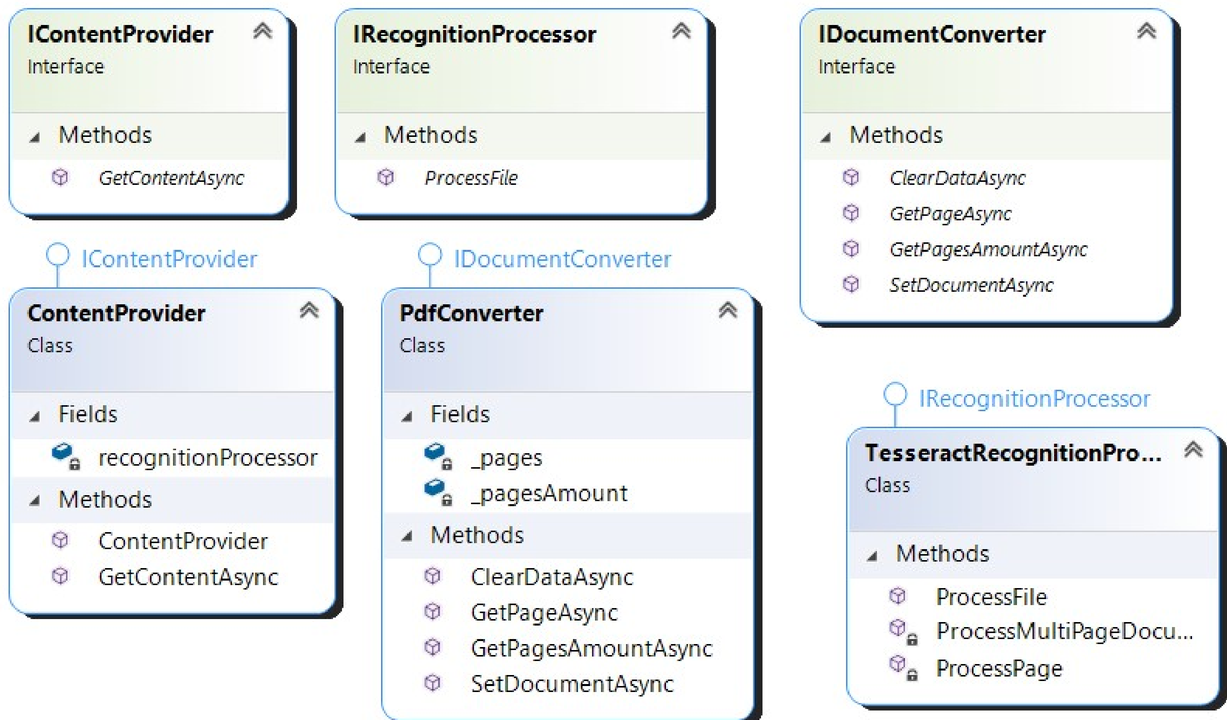

Рисунок 3.9 - Діаграма класів модулю розпізнавання

3.8 Огляд Masstransit

У якості транспорту для мікросервісу пошуку використано бібліотеку Masstransit. Це – бібліотека з відкритим програмним кодом для платформи .Net, яка реалізує паттерн шини данних (Data Bus).

Бібліотека використовується для побудови командно або подіє-орієнтованих мікросервісних архітектур. По своїй суті – вона є інтерфейсом, шаром абстракції над іншими видами міжсервісного транспорту, таких як: RabbitMq, Azure Service Bus, AmazonMq[19]. В даній імплементації у якості транспорту для Masstransit використано саме месседж-брокер RabbitMq.

На відміну від більшості інших видів транспорту, де сервіси зазвичай між собою працюють за схемою видавець-підписник, в Masstransit кожен сервіс може відправляти і отримувати повідомлення[19].

В системі є 2 види повідомлень:

- команди;
- події.

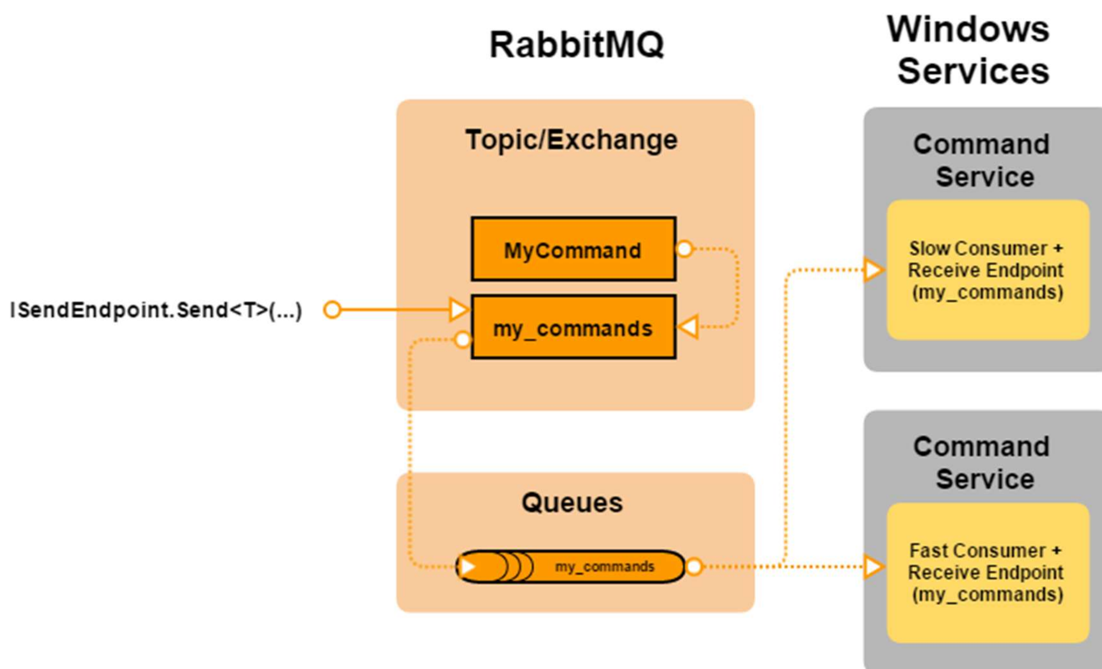

Рисунок 3.10 - Схема роботи механізму команд

Команди мають конкретного адресата, сигналізують про необхідність виконати якусь дію, наприклад команда сервісу емейл-нотифікацій про

необхідність відправки емейлу користувачу, або команда про необхідність індексації нового документу пошуковому сервісу. Команди використовують спільну чергу повідомлень для усіх сервісів, як показано на схемі на рисунку 3.10.

Події ж не мають конкретного адресата, а відправляються одразу усім сервісам. Вони сигналізують про якусь подію, зазвичай зміну даних в одному сервісі, які тепер треба узгодити з цим в усіх інших. Наприклад подія зміни імені користувача в мікросервісі аутентифікації, і подальша його зміна в усіх сервісах, до воно зберігалося, після отримання відповідного повідомлення. Для обробки подій у кожного сервісу створюється своя персональна черга подій, і події додаються в кожну з них, як показано на рисунку 3.11.

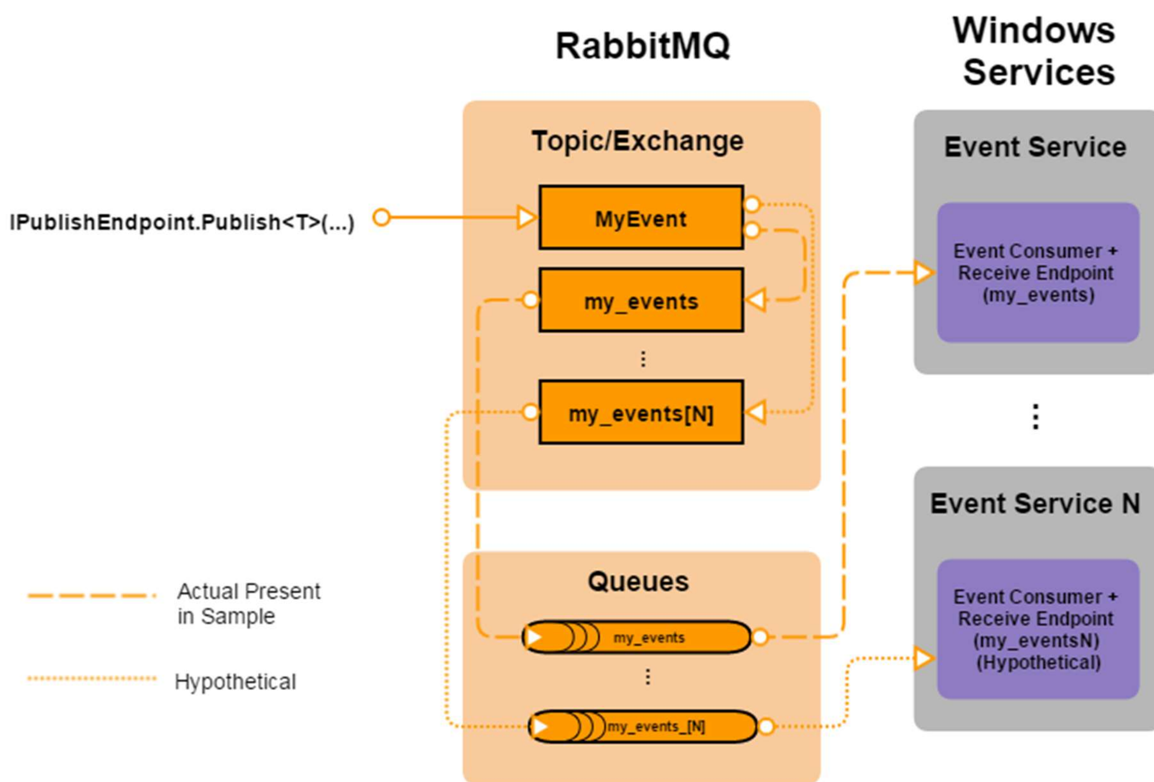

Рисунок 3.11 - Схема роботи механізму подій

Для обробки отриманого повідомлення створюються спеціалізовані класи – консьюмери. Кожен консьюмер приймає один вид команд або подій, які описані у вигляді контрактних інтерфейсів, так наприклад, інтерфейс команди з відправки

емейлу буде містити електронну адресу користувача, тему, заголовок та тіло листа, тощо.

Також, за необхідності, Masstransit може надавати двосторонні моделі зв'язку, такі як запит-відповідь. Для цього в консьюмері потрібно сконфігурувати відправку відповіді назад, а в місці початкової відправки на неї очікувати викликом спеціального метода. Але потреба у використанні таких можливостей може свідчити про проблеми визначенні зв'язаних контекстів мікросервісів та побудови архітектури в цілому.

3.9 Огляд Hangfire

Hangfire – багатопоточний планувальник задач, створений для платформи .Net. Дозволяє винести виконання методів з загального конвеєру обробки запиту в фоновий потік, такі методи називаються задачами (jobs)[19].

Схему роботи з офіційної документації Hangfire представлено на Рисунку 3.12.

Як можна зрозуміти з цієї схеми, інфраструктура Hangfire складаєтся з таких частин:

- клієнт, що додає задачу в чергу;
- база даних, у якій зберігаються задачі;
- сервер, який виконує задачу.

Зазвичай клієнт та сервер у цій схемі являють собою один і той же додаток.

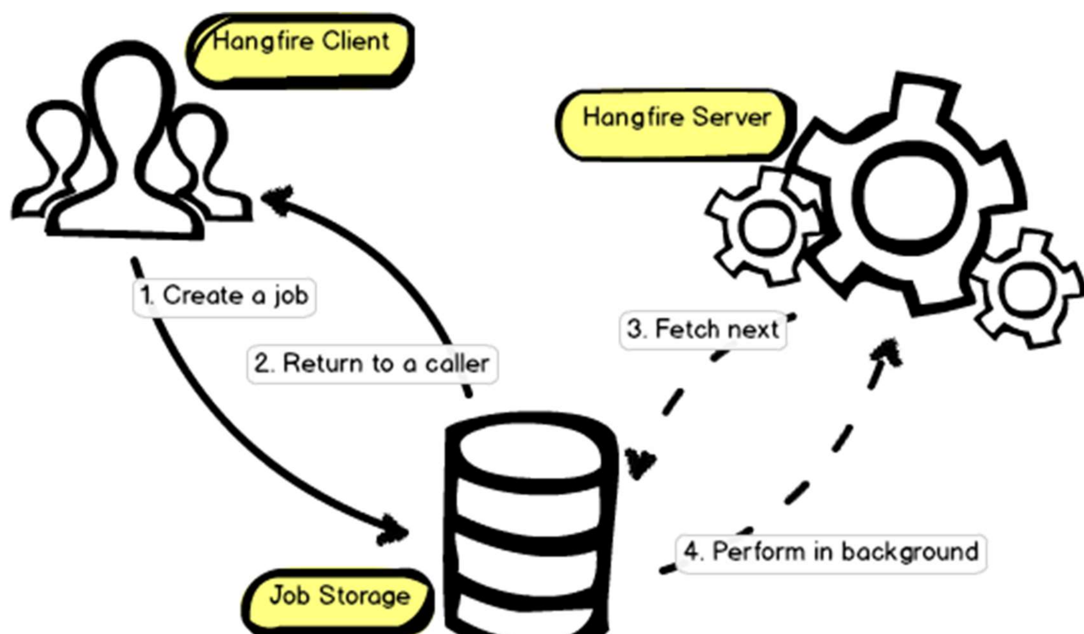

Рисунок 3.12 - Схема роботи Hangfire з офіційної документації

Усі задачі зберігаються в базі даних, робота бібліотеки Hangfire без БД – неможлива. Є підтримка популярних реляційних баз даних, таких як Microsoft SQL Sever або Azure Sql, деяких нереляційних, таких як MongoDb, та таких сховищ як Redis[20].

З точки зору клієнта взаємодія з Hangfire відбувається лише через додавання задач в базу даних.

Сервер же підключається до бази даних, перевіряє чергу задач, що там зберігаються та виконує ті з них, що готові до запуску.

Глобально є 3 підходи до запуску задач в Hangfire:

– негайне виконання (fire and forget) - використовується для випадків, коли треба просто виконати якусь роботу в фоновому потоці, наприклад для затратних з точки зору часу виконання операцій.
– нідкладене виконання - дає можливість запустити задачу через певний проміжок часу, або в конкретний час один раз. Використовується для відкладеного виконання певних дій, наприклад для зміни стану системи за таймером.
– неріодичне виконання (recurring job)- дає можливість запускати задачі за графіком Cron (формат запуску періодичних процесів в Unix системах). Використовується для періодичного виконання певних задач, наприклад щоденні розсилки листів.

Також Hangfire надає АПІ для керування доданими задачами, наприклад, можна змінювати запланований час наступного виконання, видаляти заплановані задачі, тощо.

Ври виникненні помилок при виконанні задачі Hangfire автоматично перезапускає її через певний час, збільшуючи проміжок з кожною невдалою спробою, при цьому таких запусків може бути до десяти для однієї задачі.

Додатково, Hangfire надає зручний веб-інтерфейс для ручного керування задачами, за допомогою нього можна запускати, видаляти, міняти чергу та переглядати статистику по задачам [20]. Скриншот UI інтерфейсу Hangfire представлений на Рисунку 3.13.

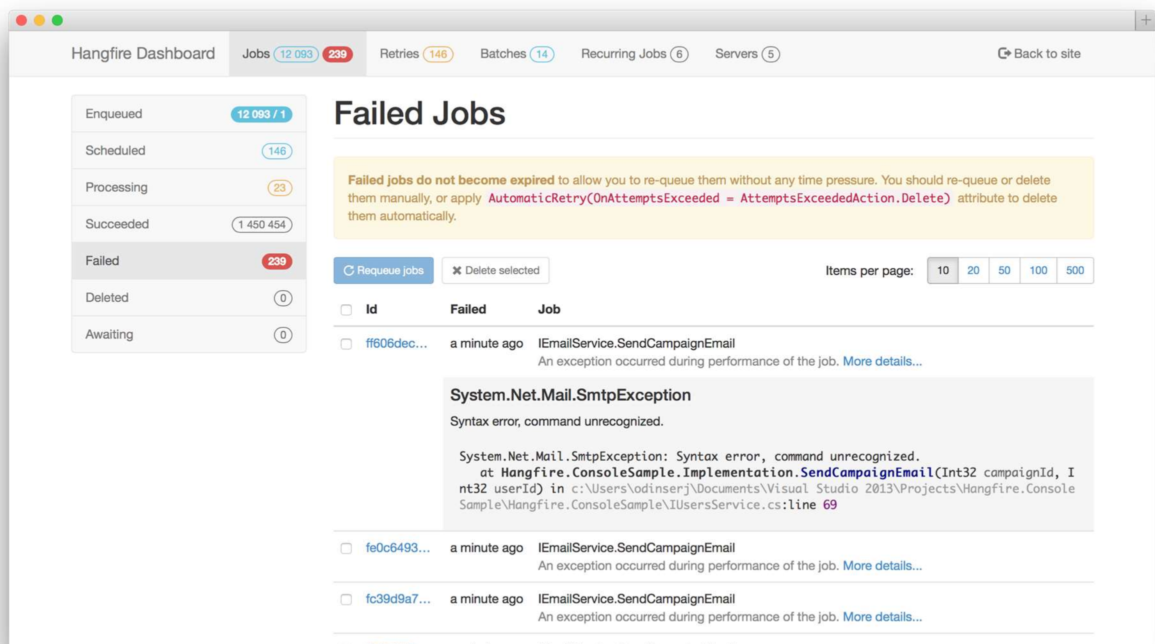

Рисунок 3.13 - Скриншот веб-інтерфейсу Hangfire

3.10 Імплементація загальної структури мікросервісу

У зв’язку з тим, що операції оптичного розпізнавання тексту на зображенні, та індексації його пошуковим рушієм дуже ресурсоїмкі, їх необхідно виконувати як фонові процеси, для цього використана бібліотека Hangfire.

Створена задача IndexJob, яка містить у якості залежностей модулі розпізнавання та пошуку, стаючи таким чином зв’язуючим елементом усієї системи.

Процес додавання документу описаний в методі Run цієї задачі:

```
public async Task Run(IndexJobArgs data, CancellationToken cancellationToken)
{
    var recognitionRequest = _mapper.Map<RecognitionRequest>(data);
    var content = await _contentProvider.GetContentAsync(recognitionRequest);

    var document = _mapper.Map<Document>(data);
    document.Content = content;

    await _elasticService.SaveSingleAsync(document);
}
```

У якості аргументу він приймає об’єкт IndexJobArgs, який містить у собі інформацію про документ що додається, таку як ідентифікатор, мова, заголовок, то сам файл документу.

Потім за допомогою бібліотеки AutoMapper, яка дозволяє за допомогою заданих в конфігурації правил перетворювати об'єкти з одним типом в інший, ми створюємо вхідні дані для методу GetContentAsync інтерфейсу IContentProvider, який є вхідною точною в модуль розпізнавання, таким чином запускаючи розпізнавання.

По завершенні процесу, ми таким же чином формуємо модель документу із вхідних даних, та задаємо її властивість, що описує зміст тіла документу, результатом розпізнавання файлу. Після цього ми відправляємо створений документ на індексацію.

Запуск задачі IndexJob відбувається в консьюмері IndexImageDocumentConsumer бібліотеки Masstransit, що обробляє команди по індексації нового документу з інших сервісів.

Лістинг методу Consume, у якому відбувається обробка команди IndexImageDocument:

```
public Task Consume(ConsumeContext<IndexImageDocument> context)
{
    var jobArgs = _mapper.Map<IndexJobArgs>(context.Message);
    _hangfireService.Enqueue<IndexJob, IndexJobArgs>(jobArgs);

    return Task.CompletedTask;
}
```

Як бачимо, спочатку за допомогою Automapper створюється модель IndexJobArgs з аргументами для запуску задачі IndexJob, потім – за допомогою сервісу IHangfireService запускається виконання задачі в фоновому режимі.

Процес пошукового запиту же відбувається по іншому сценарію. Спочатку консьюмер SearchImageDocumentConsumer приймає команду з пошуковим запитом, потім через прямий зв’язок з модулем пошуку отримує результат та відправляє його назад у відповідь.

Лістинг SearchImageDocumentConsumer:

```
        public async Task Consume(ConsumeContext<SearchImageDocument> context)
        {
            var message = context.Message;
            var result = await _elasticService.Search(message.Term, message.Page,
message.PageSize);

            await context.RespondAsync<SearchImageDocumentResponses>(new { Responses =
_mapper.Map<IEnumerable<SearchImageDocumentResponse>>(result) } );
        }
```

Отже, зв’язуюча частина між модулями пошуку та розпізнавання складається з:

- задачі IndexJob, для зв’язки модулів пошуку і розпізнавання у процесі додавання нового документу;
- консьюмера IndexImageDocumentConsumer, для запуска додавання нового докумета;
- консьюмера SearchImageDocumentConsumer, для запуска пошукового запиту;
- інтерфейса IHangfireService, який описує сервіс по роботі з задачами Hangfire;
- класу HangfireService, який реалізує вищезазначений інтерфейс.

Схему сервісу можна представити у вигляді рисунку 3.11

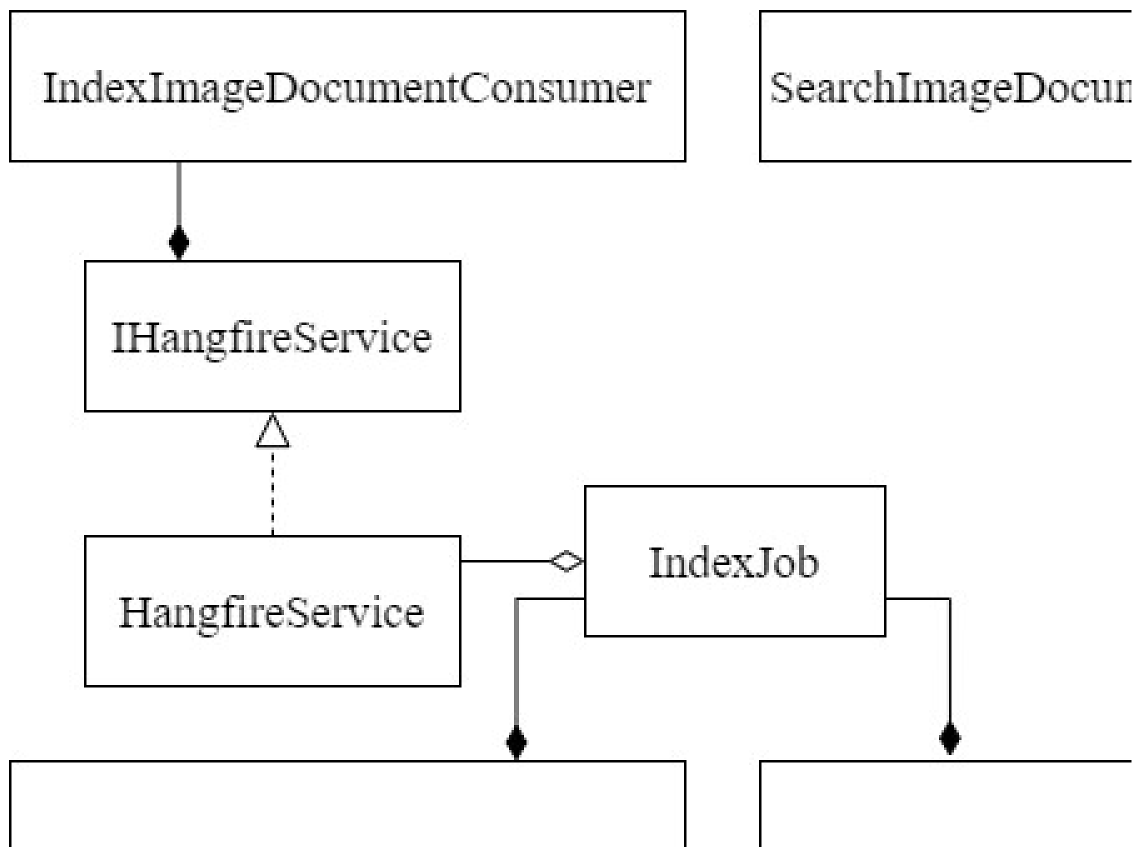

Рисунок 3.13 - Загальна схема мікросервісу

Лістинг інтерфейсу IHangfireService:

```
    public interface IHangfireService
    {
        bool Delete(string jobId);

        string Enqueue<T, TA>(TA args, CancellationToken cancellationToken = default)
            where T : IJob<TA>
            where TA : IJobArgs;

        string Schedule<T, TA>(TA args, TimeSpan delay, CancellationToken
cancellationToken = default)
            where T : IJob<TA>
            where TA : IJobArgs;

        string Schedule<T, TA>(TA args, DateTimeOffset enqueueAt, CancellationToken
cancellationToken = default)
            where T : IJob<TA>
            where TA : IJobArgs;
    }
```

Як можна побачити в лістингу, інтерфейс описує можливі дії над задачами Hangfire, а саме:

– запуск задачі в фоновому режимі негайно;

– планування задачі на запуск через певний час;

- планування запуску задачі на конкретний час.

Висновки до розділу 3

В цьому розділі було імплементовано мікросервіс пошуку по сканованим документам, для системи електронного документообігу.

Спочатку було оглянуто використаний технологічний стек та обґрунтовано його вибір.

Далі – розглянуто та проаналізовано можливі інструменти реалізації модуля пошуку. Обрано ElasticSearch, детально описано його механізм роботи, та імплементовано модуль.

Після цього розглянуто та проаналізовано засоби оптичного розпізнавання тексту, виявлено, що бібліотека Tesseract надає можливість розпізнавання декількома алгоритмами.

Досліджено ефективність роботи різних алгоритмів на різних видах документів, виявлено кращі алгоритми для кожного виду.

На основі отриманих даних, імплементовано модуль розпізнавання, який використовує одразу декілька алгоритмів, динамічно обираючи найбільш підходящий для даного виду документів, що передбачено розробленою архітектурою.

Розроблені модулі інтегровані в загальну структуру мікросервісу.

4 РЕЗУЛЬТАТИ

4.1 Транспортні контракти мікросервісу

Для пошуку по документам використовуються такі повідомлення Masstransit:

– команда SearchImageDocument;

– модель SearchImageDocumentResponse;

– модель SearchImageDocumentResponses.

Лістинг SearchImageDocument:

```
public interface SearchImageDocument
{
    public string Term { get; set; }

    public int Page { get; set; }

    public int PageSize { get; set; }
}
```

В цій команді передаються пошуковий запит, та дані для пагінації (номер сторінки та розмір сторінки). Використовується для запуску пошуку.

У відповідь на цю команду, сервіс відправляє відповідь з моделлю SearchImageDocumentResponses:

```
public interface SearchImageDocumentResponses
{
    public IEnumerable<SearchImageDocumentResponse> Responses { get; set; }
}
```

Ця модель містить колекцію моделей SearchImageDocumentResponse, які описують один результат пошуку.

Лістинг моделі SearchImageDocumentResponse:

```
public interface SearchImageDocumentResponse
{
    public int DocumentId { get; set; }

    public IEnumerable<string> Hits { get; set; }
}
```

Як бачимо, вона містить ідентифікатор документу та колекцію включень пошукового запиту в тіло документу.

Для індексації документу використовується команда IndexImageDocument.

Лістинг:

```
public interface IndexImageDocument
```

```
{
    public int Id { get; set; }

    public string Name { get; set; }

    public Stream File { get; set; }

    public DocumentType Type { get; set; }

    public Language Language { get; set; }
}
```

В ній повинні міститись дані документу, такі як його ідентифікатор, нозва, тип та мова, а також сам файл тіла.

4.2 Опис відладочного HTTP API

Під час імплементації мікросервісу, в ньому було розроблен HTTP API, необідне для відладки роботи модулів.

АПІ складається з двох контроллерів:

- RecognitionTestController;
- ElaticTestController.

Перший – містить в собі ендпоїнт для перевірки роботи модуля розпізнавання, а саме метод POST за шляхом '/api/RecognitionTest', який в тілі запиту приймає модель RecognitionRequest, що містить файл та метадані документу. Ендпоїнт повертає результат розпізнавання наданого документу у вигляді текстової строки.

Контроллер ElaticTestController містить такі ендпоїнти:

- GET за шляхом 'api/ElasticTest/{term}', де term, це пошуковий запит – ендпоїнт служить для перевірки функції пошуку, повертає колекцію об'єктів типу SearchResult, що описують результати пошуку;
- POST за шляхом 'api/ElasticTest', що приймає об'єкт типу Document в тілі запиту – ендпоїнт необхідний для перевірки індексації документа;
- POST за шляхом 'api/ElasticTest/bulk', що приймає колекцію об'єктів типу Document в тілі запиту – ендпоїнт необхідний для перевірки індексації групи документів;
- DELETE за шляхом 'api/ElasticTest/', що приймає об'єкт типу Document в тілі запиту – ендпоїнт необхідний для перевірки видалення з індексу документа.

Для зручного перегляду та тестового використання HTTP API, до проекту підключено бібліотеку Swagger.

Swagger – бібліотека для спрощення проектування, тестового використання та автоматичного документування веб-АПІ. Вона надає користувацький веб-

інтерфейс для перегляду та ручного запуску ендпоїнтів. Скриншот інтерфейсу Swagger даного проекту надано на Рисунку 4.1.

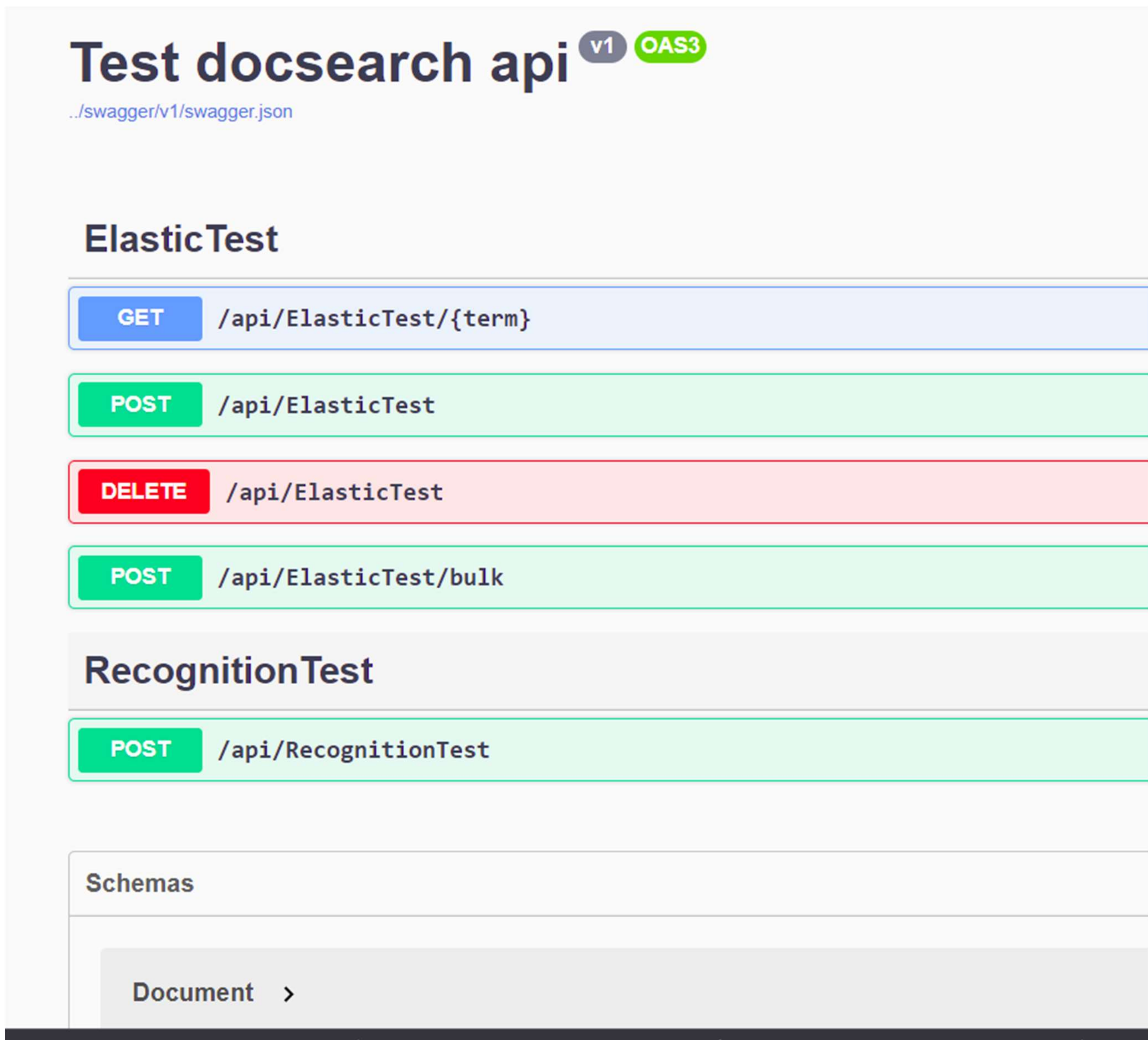

Рисунок 4.1 - Скриншот інтерфейсу Swagger

4.3 Демонстрація роботи сервісу

Спочатку необхідно вибрати пункт меню «My Documents» Рис. 4.2.

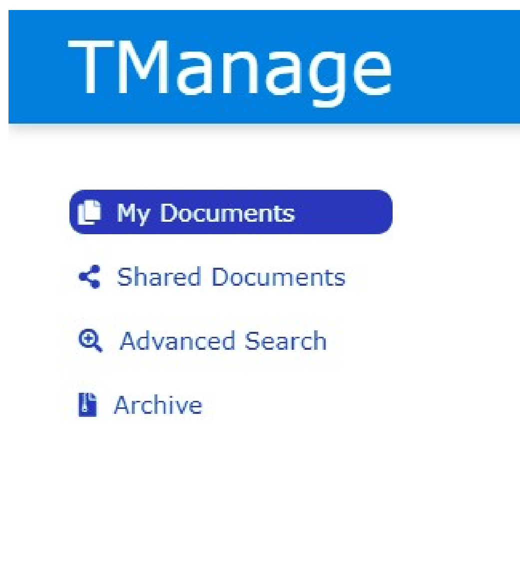

Рисунок 4.2 - Скриншот пунктів меню

Далі Натискаємо кнопку «Add Document» Рис. 4.3.

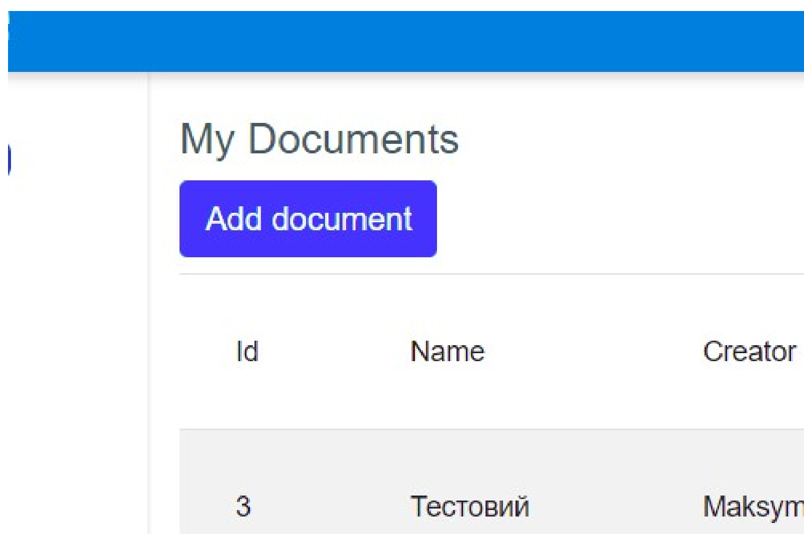

Рисунок 4.3 - Скриншот кнопки додавання нового документу

Після чого відкривається вікно додавання нового документу. Заповнюємо назву та опис, то прикріплюємо файл, це зображено на Рисунку 4.4.

Рисунок 4.4 - Скриншот вікна додавання нового документу

Далі відкриваємо, щоб упевнитися в запуску розпізнавання та індексації відкриваємо панель Hangfire для моніторингу запущених на виконання фонових задач, для цього переходимо за адресою в браузері {домен_додатку}/dashboard, та пересвідчумося що задача додавання нового документа в мікросервіс пошуку запущена, як показано на Рисунку 4.5

Рисунок 4.5 - Скриншот панелі моніторингу Hangfire

По завершенню виконання задачі, переходимо в пункт меню «Advanced Search» (Рис. 4.2) та намагаємося шукати наш документ по змісту, успішність пошуку зображено на Рисунку 4.6.

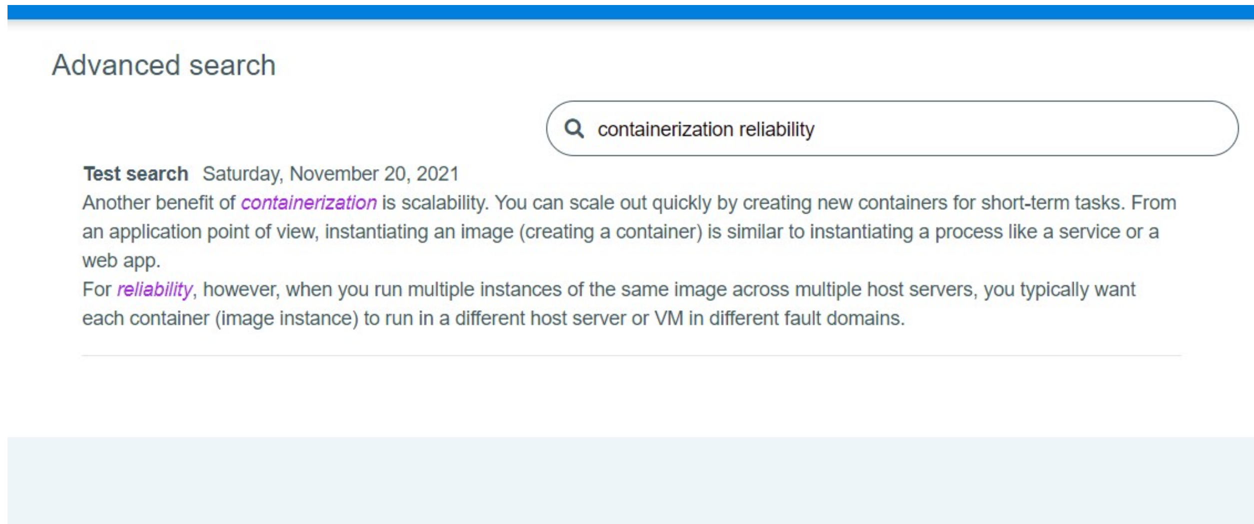

Рисунок 4.6 - Вікно пошуку з результатом

Як можна побачити на скриншоті, якість розпізнавання тексту достатньо висока для можливості використання пошуку.

Знайдені фрази обрамлюються заданими html-тегами, для того щоб їх можна було підсвітити в пошуковій видачі.

4.4 Інструкція з розвертання

Для збірки та запуску системи перш за все необхідно встановити оне з нижчеперерахованих програмних забезпечень:

- IDE Microsoft Visual Studio 19;
- IDE JetBrains Rider;
- .Net Core 3.1 sdk, який можна скачати з офіційного сайту Майкрософт.

Я скористався першим варіантом.

В Visual Studio повинен бути доданим .Net Core 3.1 sdk.

Далі необхідно відновити пакети бібліотек, для цього у вікні терміналу диспетчеру пакетів NuGet виконуємо команду ‘nuget restore TManage.sln’.

Для міжсервісного транспорту нам необхідно запустити RabbitMq, це можна зробити двома способами:

- встановити на свій локальний комп’ютер, скачавши з офіційного сайту виробника тапустити на ньому напряму;
- запустити в докер-контейнері.

Я скористався другим способом.

Для цього спочатку необхідно встановити Docker Desktop, завантаживши його з офіційного сайту.

Далі необхідно встановити WSL2 (Windows Subsystem for Linux), для цього спочатку треба виконати команду ‘wsl --install’ d Powershell запустивши його від обличча адміністратора, після чого перезавантажити систему, потім виконати ще одну команду ‘wsl --set-default-version 2’, щоб активувати другу версію, і ще раз перезавантажити.

Після цього в терміналі виконуємо наступну команду:

docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.9-management

Вона повинна завантажити образ RabbitMq для Docker, та його запустити зі стандартними параметрами.

Далі необхідно встановити сервер ElasticSearch, зробити це також можна через Docker, для цього запускаємо наступні команди:

‘docker pull docker.elastic.co/elasticsearch/elasticsearch:7.15.2’

‘docker run -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.co/elasticsearch/elasticsearch:7.15.2’

В подальшому, встановлення та запуск усього необхідного для Docker можна об’єднати в один docker-compose.

Далі необхідно поправити файли конфігурації під свої потреби. ASP.NET Core передбачає загальний json файл – ‘appsettings.json’, та специфічні – ‘appsettings.{назва оточення}.json’ для кожного оточення, які переписують значення у загальному.

В розділі elasticsearch необхідно задати назву індексу, для збереження документів у полі ‘index’, а також адресу сервера ElasticSearch в розділі ‘url’ – для локально встановленого значення за замовчуванням - "http://localhost:9200/".

В розділі MassTransitOptions необхідно задати адресу підключення RabbitMq. Значення за замовчуванням – ‘localhost’. Те ж саме потрібно зробити в усіх сервісах.

В розділі ConnectionStrings потрібно задати адресу реляційної бази даних для використання Hangfire. Можна встановити Microsoft SQL Server локально, створити там базу та указати її, або скористатися localdb, для цього задаємо значення:

‘Server=(localdb)\\MSSQLLocalDB;Database=TManage;Trusted_Connection=True;MultipleActiveResultSets=true’.

Далі необхідно зібрати клієнтську частину, для цього спочатку необхідно встановити npm пакети. Щоб це зробити спочатку необхідно встановити node.js.

Після цього запускаємо термінал, заходимо в директорію з клієнтом та запускаємо команду ‘npm i’, це повинно встановити усі пакети, в тому числі Angular SLI.

Далі запускаємо команду ‘ng serve -o’, яка повинна зібрати клієнт, запустити його та відкрити вікно браузера з відкритим додатком.

Висновки до розділу 4

В даному розділі представлений мікросервіс, який було спроектовано та розроблено в рамках виконання цієї роботи, та його інтеграція з системою електронного документообігу.

Детально описані контракти міжсервісної шини даних та тестового HTTP API.

Далі, продемонстровано роботу готової системи та надано інструкцію з розвертання мікросервісу.

5 МАРКЕТИНГОВИЙ АНАЛІЗ СТАРТАП-ПРОЄКТУ

5.1 Опис ідеї проєкту

Таблиця 5.1 — Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Створення системи електронного документообі з використанням імплементованого в роботі мікросервісу	Можливість пошуку сканованої вхідної кореспонденції	Спрощення взаємодії з зовнішніми суб'єктами
	Можливість пошуку сканованих архівних даних	Економія на ручній оцифровці

Таблиця 5.2 — Опис ідеї стартап-проєкту

№	Техніко-економічні характери-тики ідеї	Продукція конкурентів			W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проєкт	FossDoc	Docsvision			
1	Наявність повнотекстового пошуку	Так	Ні	Так			+
2	Можливість пошуку по сканам	Так	Ні	Ні			+
3	Можливість архівування документів	Так	Так	Так		+	
4	Дозвільна система доступу до документів	Так	Так	Так		+	
5	Аудит дій	Так	Так	Так		+	

5.2 Технологічний аудит ідеї проєкту

Таблиця 5.3 — Технологічна здійсненність ідеї проєкту

№	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Розпізнавання друкованого тексту	Tesseract, Puma.Net,	Наявні	Безкоштовні, з відкритим сирцевим кодом
2	Розпызнавання рукописного тексту	Microsoft Azure Cognitive Services OCR	Наявні	Платні
3	Повнотекстовий пошук	ElasticSearch	Наявні	Безкоштовні
4				
Обрана технологія реалізації ідеї проєкту:1				

Висновок: технології необхідні в реалізації наявні на даний момент, більшість з них безкоштовні для комерційного використання, платна – не обов’язкова для викоритання в базовому наборі продукту. Реалізація можлива.

5.3 Аналіз ринкових можливостей запуску стартап-проєкту

Таблиця 5.4 — Попередня характеристика потенційного ринку

№	Показники стану ринку	Характеристика
1	Кількість головних гравців, од	3
2	Загальний обсяг продаж, грн./ум.од	Дані відутні
3	Динаміка ринку	Зростає
4	Наявність обмежень для входу	Немає
5	Специфічні вимоги до стандартизації та сертифікації	Необхідна підтримка популярних форматів документів
6	Середня норма рентабельності в галузі або по ринку, %	Дані відсутні

Враховуючи кількість головних гравців по ринку, зростаючу динаміку ринку, можна зробити висновок, що ринок для входження продукту є привабливим.

Таблиця 5.5 — Характеристика потенційних клієнтів стартап-проєкту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці цільових груп клієнтів	Вимоги споживачів до товару
1	Повнотектовий пошук по сканованим документвм	Підприємства, що використовують автоматизацію документообігу та мають багато вхідної кореспонденції або неоцифровані паперові архіви	Навантаженне на систему, вимоги до безпеки даних	Зручність інтерфейсу, стабільність роботи, швидкодійність, точність розпізнаввання

Таблиця 5.6 — Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	Конкуренти	Наявність конкурентів на ринку	Правильність цінової політики; Унікальніхарактеристики товару;
2	Актуальність технологій	Втрата популярності у форматів документів, що підтримує система	Підтримка нових форматів
3	Повна цифровізація	Відсутність потреби в роботі зі сканами	Пошук та вирішення інших проблем

Таблиця 5.7 — Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	Зростання потреби в автоматизації процесів	Більше підприємств потребують системи електронного документообігу	Вихід нової продукції на ринок; Надання різноманітних типів ліцензій в залежності від потреб користувача \ замовника.
2	Зворотній зв’язок від користувачів	Можливість отримання інформації про недоліки та можливі покращення продукту	Врахування отриманої інформації при плануванні оновлень

Таблиця 5.8 — Ступеневий аналіз конкуренції на ринку

№	Особливості конкурентного середовища	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1	Тип конкуренції: олігополія	Цінова політика; Створення функціональних переваг

Продовження Таблиці 5.8

2	Рівень конкурентної боротьби: світовий	Локалізація інтерфейсу
3	Галузева ознака: Міжгалузева	Підтримка більшої кількості форматів
4	Конкуренція за видами товарів: товарно-видова	Надання продукту вищої якослі з використанням сучаснішого технологічного стеку. Забезпечення безпеки даних Надання підтримки.
5	Характер конкурентних переваг: цінова та не цінова	Надання опіональної платної підтримки, різні рівні ліцензування, функціональні переваги
6	За інтенсивністю: марочна	Створення власного бренду

Таблиця 5.9 — Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	FossDoc Docsvision	Є	Відсутні	Підприємства, що потребують впровадження автоматизації документообігу	Товари замінники відсутні

Продовження Таблиці 5.9

Висновки	Існує два конкуренти, один має нішевий продукт, інший досить сильний та розвивається	Крупні технологічні компанії можуть зацікавитись ринком	Відсутня залежність від постачальників	Для користувачів важлива зручність, безпека та надійність	Немає обмежень на ринку

За результатами можна зробити висновок: продукт повинен мати функціональні переваги та нижчу собівартість ніж у конкурента для виходу на ринок.

Таблиця 5.10 — Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування
1	Наявність повнотекстового пошуку по сканам	Функціональна можливість, що відсутня у конкурентів
2	Підтримка різних форматів	Обмеження по форматам зменшує кількість потенційних клієнтів
3	Можливість архівування	Дозволяє заощаджувати на серверних ресурсах
4	Безпека даних	Важливий для корпоративних клієнтів

Таблиця 5.11 — Порівняльний аналіз сильних та слабких сторін системи кешування мало змінних даних

№	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з запропонованим						
			-3	-2	-1	0	+1	+2	+3
1	Наявність пошуку по сканам	10	+						
2	Підтримка різних форматів	20					+		

Продовження Таблиці 5.11

3	Можливість архівування	5				+			
4	Безпека даних	20				+			

Таблиця 5.12 — SWOT аналіз стартап-проєкту

Сильні сторони (S):	Слабкі сторони (W):
– Наявність унікальної функціональності – Якість продукту – Зручний інтерфейс	– Наявність сильних конкурентів – Невідомість бренду
Можливості (O):	**Загрози (T):**
– Збільшення використання систем електронного документообігу	– Поява нових конкурентів

Таблиця 5.13 — Альтернативи ринкового впровадження стартап-проєкту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Постпродажна підтримка	90%	1 місяць
2	Упровадження різних видів ліцензування	80%	1-2 місяці
3	Реклама	50%	2-3 місяці

5.4 Розроблення ринкової стратегії проєкту

Таблиця 5.14 — Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Малі підприємства	Висока	Низька	Низька	Середня
2	Середні підприємства	Середня	Висока	Середня	Висока
3	Крупні підприємства	Низька	Низька	Висока	Низька
Які цільові групи обрано: обрано середні та малі підприємства. Не обрано крупні через високий рівень конкуренції					

Таблиця 5.15 — Визначення базової стратегії розвитку

Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Надання функціональності що відсутня у конкурентів, підтримка клієнтів	Освітлення унікальної функціональності через рекламу формування клієнтів	Лояльність клієнтів Наявність унікальної конкурентної переваги	Стратегія диференціації

Таблиця 5.16 — Визначення базової стратегії конкурентної поведінки

Чи є проєкт «першопрохідцем» на ринку	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, які?	Стратегія конкурентної поведінки
Ні	Залучити нових споживачів та, частково забрати існуючих	Компанія копіює характеристики товару конкурента, основна ціль - розробка нового функціоналу, з підтримкою функціоналу конкурентів	Стратегія заняття конкурентної ніші

Таблиця 5.17 — Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проєкту	Вибір асоціацій, які мають сформувати комплексну позицію власного проєкту
1	Підтримка повнотекстового пошуку	Стратегія диференціації	Пошук по сканах	Простота цифровізації, точність

Відповідно до проведеного аналізу можна зробити висновок, що продукт вибирає як базову стратегію розвитку – стратегію диференціації, як базову стратегію конкурентної поведінки – стратегію заняття конкурентної ніші.

5.5 Розроблення маркетингової програми стартап-проєкту

Таблиця 5.18 — Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Зручність	Сучасний інтерфейс	Більш інтуітивний інтерфейс
2	Пошук документів	Повнотекстовий пошук по зображенням	Підтримка пошуку по сканам
3	Безпека даних	Дані зберігаються локально	Відсутність мережевих пересилок даних

Таблиця 5.19 — Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td colspan="3">Сутність та складові</td></tr>
<tr><td>1. Товар за задумом</td><td colspan="3">Система електронного документообігу</td></tr>
<tr><td rowspan="4">2. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх/Тл/Е/Ор</td></tr>
<tr><td>Пошук по зображенням</td><td>Нм</td><td>Тл/Е</td></tr>
<tr><td colspan="3">Якість: визначається кількістю багів</td></tr>
<tr><td colspan="3">Пакування: ліцензія на сервер, ліцензії на користувачів</td></tr>
<tr><td rowspan="2">3. Товар із підкріпленням</td><td colspan="3">До продажу: різні рівні ліцензування, послуги з розвертання</td></tr>
<tr><td colspan="3">Після продажу: додаткова підтримка</td></tr>
</table>

За рахунок чого потенційний товар буде захищено від копіювання: захист інтелектуальної власності

Таблиця 5.20 — Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
28000 грн + 2800 грн/рік ліцензія + 280 грн/рік кожне робоче місце	-	56000000 грн/рік	15000 грн + 1000 грн/рік ліцензія – 30000 грн + 3000 грн/рік ліцензія + 300 грн/рік кожне робоче місце

Таблиця 5.21 — Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
клієнти купують продукт безпосередньо у компанії-розробника	Технічна підтримка Допомога в впровадженні Обслуговування	0-Канал нульового рівня (виробник безпосередньо продає товар клієнту)	Через сайт виробника

Таблиця 5.22 — Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуютьс я цільові клієнти	Ключові позиції, обрані для позиціонуван ня	Завдання рекламного повідомлення	Концепція рекламного звернення
Клієнти дізнаються на спеціалізова них конференція х, в інтернеті	Інтернет, Спеціальні заходи, конференції	Нові можливості, нижча ціна	Проінформуват и про існування продукту та його переваги	Нові можливості, спеціалізація на роботі з дому

Висновки по розділу 5

В даному розділі проаналізовано особливості ринку систем електронного докуметообігу та його прийнятність для запуску стартап-проекту в цій сфері.

Після розгляду факторів зовнішнього впливу, потенційних можливостей та ризиків було встановлено що проєкт є перспективним. Проведено технологічну експертизу проекту. Аналіз конкурентної боротьби показав, що з наявністю унікальних функціональних можливостей стартап-проект може бути успішним, та його імплементація є доцільною.

ЗАГАЛЬНІ ВИСНОВКИ

В ході виконання роботи було розглянуто існуючі підходи до оптичного розпізнавання текстів. Виявлено переваги кожного алгоритму в своїй сфері застосування, сформульовано вимогу використання переваг декількох алгоритмів в архітектурі, що проектується.

Було розроблено архітектуру мікросервіса, яка відповідає поставленим до неї вимогам, а саме: надає можливість повнотекстового пошуку по зображенням та сканованим документам, при цьому надаючи можливість додавання декількох інструментів розпізнавання, таким чином забезпечуючи використання різних алгоритмів в залежності від контексту.

Для реалізації аналітичного блоку, який повинен динамічно обирати оптимальний для наданого документу алгоритм розпізнавання, проведено дослідження залежності ефективності різних підходів машинного розпізнавання від метаданих документу.

На основі спроектованої архітектури було розроблено мікросервіс повнотекстового пошуку по сканованим документам на базі фреймворку ASP.NET Core. Для забезпечення міжсервісного транспорту використано RabbitMq з обгорткою у вигляді бібліотеки Masstransit. Для імплементації модуля пошуку використано пошуковий рушій ElasticSearch. В якості інструменту оптичного розпізнавання виступає бібліотека Tesseract, яка дає можливість використання декількох різних алгоритмів в межах однієї інтеграції.

Розроблений мікросервіс було інтегровано в систему електронного документообігу.

Результати роботи було опубліковано в збірці тез Міжнародної наукової інтернет-конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення» (випуск 62).

Наукова новизна полягає в удосконаленні програмного забезпечення електронного документообігу шляхом розробки архітектури, яка надає можливість

збільшити кількість підтримуваних форматів документів шляхом використання методів оптичного розпізнавання тексту.

Практичне значення результатів роботи полягає в імплементації розробленої архітектури в розробці сервісу повнотекстового пошуку по сканованим документам та інтеграцію його в систему електронного документообігу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Савицька О. М., Салабай В. О. Особливості діджиталізації бізнесу компанії в умовах розвитку індустрії 4.0 / Савицька О. М., Салабай В. О // Ефективна економіка. - 2020. - № 10. – Режим доступу до журн.: http://www.economy.nayka.com.ua/?op=1&z=8266.

2) Колпак, М. В. Система електронного документообігу підприємства : дипломний проект бакалавра : 123 Комп'ютерна інженерія / Колпак М. В. – Київ, - 2020. – 75 с.

3) Docsvision [Електронний ресурс] / ДоксВижн. - 2020. – Режим доступу: *https://e-docs.ua/*.

4) E-Docs [Електронний ресурс] / e-Docs. - 2020. – Режим доступу: *https://www.docsvision.com/*.

5) Prozorro публічні закупівлі [Електронний ресурс] / ДП "ПРОЗОРРО". - 2019. – Режим доступу: *https://prozorro.gov.ua/tender/UA-2019-07-29-000305-c*.

6) FossDoc [Електронний ресурс] / ФОСС-Он-Лайн. - 1999-2018. – Режим доступу: https://fossdoc.com/.

7) Леонтьев В. К. О мерах сходства и расстояниях между объектами / Леонтьев В. К. // Ж. вычисл. матем. и матем. физ., - 2009. - 49:11. –С. 2041–2058

8) Senka Drobac, Krister Lindén Optical character recognition with neural networks and post-correction with finite state methods / Senka Drobac, Krister Lindén // International Journal on Document Analysis and Recognition (IJDAR) - 2020. - №23. - С. 279–295

9) Full Text Search: How it Works [Електронний ресурс] / Svetlana Stasilovich. - 2018. – Режим доступу: https://blog.issart.com/full-text-search-how-it-works/.

10) Дино Эспозито Microsoft .NET: архитектура корпоративных приложений / Дино Эспозито, Андреа Сальтарелло. - Вильямс, 2016. – С. 427.

11) Caesar de la Torre, Bill Wagner, Mike Rousos. .Net Microservices: Architecture for Containerized .Net Applications. - Redmond, Washington: Microsoft Developer Division, 2021. –350 с.

12) Колпак М.В. Повнотекстовий пошук у відсканованих документах в системах електронного документообігу/ Колпак М.В. //Міжнародна наукова інтернет-конференція «Інформаційне суспільство: технологічні, економічні та технічні спекти становлення». – 2021. – № 62. – С. 30-32.

13) Колпак М. В. Архітектура програмного забезпечення повнотекстового пошуку у відсканованих документах в системах електронного документообігу / Колпак М. В. // Перша Всеукраїнська науково-практична конференція молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології» (SoftTech-2021). Секція кафедри інформатики та програмної інженерії. Матеріали конференції. – Київ. – 2021. 22–26 листопада 2021р. – С. 111-113.

14) ElasticSearch Guide [Електронний ресурс] / Elasticsearch B.V. – 2021. – Режим доступу: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html.

15) How to integrate ElasticSearch in ASP.NET Core [Електронний ресурс] / Blexin. – 2020. – Режим доступу: https://blexin.com/en/blog-en/how-to-integrate-elasticsearch-in-asp-net-core/.

16) Puma.NET OCR SDK [Електронний ресурс] / Scribd. – Режим доступу: https://ru.scribd.com/doc/249360160/Puma-net-Getting-Started.

17) Tesseract Documentation [Електронний ресурс] / tesseract-ocr. – Режим доступу: https://tesseract-ocr.github.io/tessdoc/.

18) Alessandro Del Sole Microsoft Computer Vision APIs Distilled / Alessandro Del Sole // - Cremona, Italy. - Appress, 2018. – С. 98.

19) Masstransit [Електронний ресурс] / Chris Patterson. - 2007-2020. – Режим доступу: https://masstransit-project.com/usage/.

20) Hangfire, - Sergey Odinokov. - 2013-2020. – Режим доступу: https://docs.hangfire.io/en/latest/.

ДОДАТКИ

ДОДАТОК А

Лістинг конфігураційного класу мікросервісу

```
using AutoMapper;
using DocsSearch.Components;
using DocsSearch.Elastic;
using DocsSearch.Hangfire;
using DocsSearch.Recognition;
using DocsSearch.Recognition.Abstractions;
using Hangfire;
using Hangfire.Dashboard;
using Hangfire.SqlServer;
using MassTransit;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.OpenApi.Models;
using System;
using System.Collections.Generic;
using TransportUtils;

namespace DocsSearch.Host
{
    public class Startup
    {
        private const string ApiVersion = "v1";
        private const string ApiName = "Test docsearch api";
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            var connectionString = Configuration.GetConnectionString("DbConnection");
            services.AddControllers();
            services.AddElasticsearch(Configuration);
            services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

            services.AddScoped<IElasticService, ElasticService>();
            services.AddScoped<IContentProvider, ContentProvider>();
            services.AddScoped<IRecognitionProcessor, TesseractRecognitionProcessor>();
            services.AddScoped<IHangfireService, HangfireService>();
            services.AddSwaggerGen(c =>
            {
                c.SwaggerDoc(ApiVersion, new OpenApiInfo { Title = ApiName, Version = 
ApiVersion });
            });

            services.AddHangfire(config => config.UseSqlServerStorage(connectionString));
            services.AddHangfireServer();
            JobStorage.Current = new SqlServerStorage(connectionString);


services.Configure<MassTransitOptions>(Configuration.GetSection(nameof(MassTransitOptions)));
            services.AddMassTransit(cfg =>
            {
                cfg.ConfigureRabbitMq();
                cfg.SetKebabCaseEndpointNameFormatter();

                cfg.AddConsumersFromNamespaceContaining<IndexImageDocumentConsumer>();
```

```
            });

            services.AddMassTransitHostedService();
        }


        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {

            app.UseDeveloperExceptionPage();
            app.UseSwagger();
            app.UseSwaggerUI(c =>
            {
                c.SwaggerEndpoint($"../swagger/{ApiVersion}/swagger.json", $"{ApiName}
{ApiVersion}");
            });

            app.UseCors(builder =>
            {
                builder.AllowAnyMethod();
                builder.AllowAnyHeader();
                builder.SetIsOriginAllowed((host) => true);
                builder.AllowCredentials();
            });
            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
                endpoints.MapHangfireDashboard("/dashboard", new DashboardOptions
                {
                    Authorization = new List<IDashboardAuthorizationFilter>()
                });
            });
        }
    }
}
```

ДОДАТОК Б

Лістинг модулів пошуку та розпізнавання

```
using DocsSearch.Entities;
using Microsoft.Extensions.Logging;
using Nest;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DocsSearch.Elastic
{
    public class ElasticService : IElasticService
    {
        private readonly IElasticClient _elasticClient;

        private readonly ILogger<ElasticService> _logger;

        public ElasticService(IElasticClient elasticClient, ILogger<ElasticService> logger)
        {
            _elasticClient = elasticClient;
            _logger = logger;
        }

        public async Task<IEnumerable<SearchResult>> Search(string query, int page = 0, int
pageSize = 5)
        {
            var response = await _elasticClient.SearchAsync<Document>(s => s
                .From(page * pageSize)
                .Size(pageSize)
                .Query(q => q.QueryString(q => q
                    .Fields(fields => fields
                        .Field(f => f.Content)
                        .Field(f => f.Name))
                    .Type(TextQueryType.BestFields)
                    .Query(query)))
                .Highlight(h => h
                    .Encoder(HighlighterEncoder.Default)
                    .Fields(fs => fs
                        .Field(p => p.Content)
                        .Type("unified")
                        .ForceSource()
                        .FragmentSize(300)
                        .Fragmenter(HighlighterFragmenter.Span)
                        .NumberOfFragments(10)
                        .NoMatchSize(300)))
                );

            if (!response.IsValid)
            {
                return new List<SearchResult>();
            }

            return response.Hits.Select(h =>
                new SearchResult
                {
                    DocumentId = int.Parse(h.Id),
                    Hits = h.Highlight.SelectMany(h => h.Value)
                });
        }

        public async Task SaveSingleAsync(Document document)
```

```
        {
            await _elasticClient.IndexDocumentAsync(document);
        }

        public async Task SaveManyAsync(Document[] documents)
        {
            var result = await _elasticClient.IndexManyAsync(documents);
            if (result.Errors)
            {
                // the response can be inspected for errors
                foreach (var itemWithError in result.ItemsWithErrors)
                {
                    _logger.LogError("Failed to index document {0}: {1}",
                        itemWithError.Id, itemWithError.Error);
                }
            }
        }

        public async Task SaveBulkAsync(Document[] documents)
        {
            var result = await _elasticClient.BulkAsync(b =>
b.Index("documents").IndexMany(documents));
            if (result.Errors)
            {
                // the response can be inspected for errors
                foreach (var itemWithError in result.ItemsWithErrors)
                {
                    _logger.LogError("Failed to index document {0}: {1}",
                        itemWithError.Id, itemWithError.Error);
                }
            }
        }

        public async Task DeleteAsync(Document document)
        {
            await _elasticClient.DeleteAsync<Document>(document);
        }
    }
}
```

```
using DocsSearch.Common.Enums;
using DocsSearch.Entities;
using DocsSearch.Recognition.Abstractions;
using System.Threading.Tasks;

namespace DocsSearch.Recognition
{
    public class ContentProvider : IContentProvider
    {
        private readonly IRecognitionProcessor recognitionProcessor;

        public ContentProvider(IRecognitionProcessor recognitionProcessor)
        {
            this.recognitionProcessor = recognitionProcessor;
        }

        public async Task<string> GetContentAsync(RecognitionRequest request) => request
switch
        {
            { Type: DocumentType.Image } => await
recognitionProcessor.ProcessFile(request.Path, request.Type, request.Language,
Tesseract.EngineMode.TesseractAndLstm),
```

```
            { Type: DocumentType.Pdf }  => await
recognitionProcessor.ProcessFile(request.Path, request.Type, request.Language,
Tesseract.EngineMode.LstmOnly),
            { Type: DocumentType.DjVu } => await
recognitionProcessor.ProcessFile(request.Path, request.Type, request.Language,
Tesseract.EngineMode.LstmOnly),
            _ => await recognitionProcessor.ProcessFile(request.Path, request.Type,
request.Language, Tesseract.EngineMode.Default)
        };
    }
}
using DocsSearch.Common.Enums;
using DocsSearch.Recognition.Abstractions;
using System;
using System.IO;
using System.Threading.Tasks;
using Tesseract;

namespace DocsSearch.Recognition
{
    public class TesseractRecognitionProcessor : IRecognitionProcessor
    {
        public async Task<string> ProcessFile(
            string path,
            DocumentType type,
            Language language,
            EngineMode mode) =>
                type switch
                {
                    DocumentType.Image => await ProcessPage(path, language, mode),
                    _ => await ProcessMultiPageDocument(path, type, language, mode)
                };

        private Task<string> ProcessPage(string path, Language language, EngineMode mode)
        {
            var current = Directory.GetCurrentDirectory();
            using (var engine = new TesseractEngine(@".", GetLang(language), mode))
            {
                using (var img = Pix.LoadFromFile(path))
                {
                    using (var page = engine.Process(img))
                    {
                        return Task.FromResult(page.GetText());
                    }
                }
            }

            string GetLang(Language language) =>
                language switch
                {
                    Language.English => "eng",
                    Language.Ukrainian => "ukr",
                    Language.Russian => "rus"
                };

        }

        private async Task<string> ProcessMultiPageDocument(
            string path,
            DocumentType type,
            Language language,
            EngineMode mode)
        {
            var collector = "";
```

```
            var converter = GetConverterAsync();
            await converter.SetDocumentAsync(path);
            var pages = await converter.GetPagesAmountAsync();

            for (int i = 0; i < pages; i++)
            {
                collector += await ProcessPage(await converter.GetPageAsync(i), language,
mode);
            }

            await converter.ClearDataAsync();

            return collector;

            IDocumentConverter GetConverterAsync() => type switch
            {
                DocumentType.Pdf => new PdfConverter(),
                _ => throw new ArgumentOutOfRangeException()
            };
        }

    }
}
using DocsSearch.Recognition.Abstractions;
using PDFLibNet32;
using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Threading.Tasks;

namespace DocsSearch.Recognition
{
    public class PdfConverter : IDocumentConverter
    {
        private int _pagesAmount;
        private Dictionary<int, string> _pages;

        public Task ClearDataAsync()
        {
            foreach (var page in _pages)
            {
                File.Delete(page.Value);
            }

            _pages = new Dictionary<int, string>();
            _pagesAmount = 0;

            return Task.CompletedTask;
        }

        public Task<string> GetPageAsync(int number) =>
Task.FromResult(_pages.GetValueOrDefault(number));

        public Task<int> GetPagesAmountAsync() => Task.FromResult(_pagesAmount);


        public Task SetDocumentAsync(string path)
        {
            PDFWrapper _pdfDoc = new PDFWrapper();
            _pages = new Dictionary<int, string>();
            _pagesAmount = _pdfDoc.PageCount;
            _pdfDoc.LoadPDF(path);
```

```
            for (int i = 0; i < _pagesAmount; i++)
            {

                var img = RenderPage(_pdfDoc, i);

                var title = string.Format("{0}.jpg", i);
                _pages.Add(i, title);
                img.Save(title);

            }
            _pdfDoc.Dispose();
            return Task.CompletedTask;

            Image RenderPage(PDFWrapper doc, int page)
            {
                doc.CurrentPage = page + 1;
                doc.CurrentX = 0;
                doc.CurrentY = 0;

                doc.RenderPage(IntPtr.Zero);

                // create an image to draw the page into
                var buffer = new Bitmap(doc.PageWidth, doc.PageHeight);
                doc.ClientBounds = new Rectangle(0, 0, doc.PageWidth, doc.PageHeight);
                using (var g = Graphics.FromImage(buffer))
                {
                    var hdc = g.GetHdc();
                    try
                    {
                        doc.DrawPageHDC(hdc);
                    }
                    finally
                    {
                        g.ReleaseHdc();
                    }
                }
                return buffer;

            }

        }
    }
}
```

ДОДАТОК В

Ефективність розпізнавання для різних мов

Номер файлу	Англійська	Українська	Російська
1	93	87	91
2	95	88	91
3	94	87	90
4	95	89	91
5	95	90	89
6	95	88	91
7	94	86	90
8	94	87	91
9	94	87	89
10	94	84	91
11	97	86	79
12	95	86	84
13	95	85	82
14	95	86	81
15	94	81	80
16	97	84	80
17	96	73	83
18	96	77	85
19	98	82	86
20	94	89	86
21	94	82	88
22	91	86	88
23	89	85	78
24	90	85	79
25	91	87	93
26	92	83	83
27	91	88	83
28	92	88	82
29	91	87	83
30	85	85	83

ДОДАТОК Г

Результати перевірки роботи на співпадіння

UNICHECK

Имя пользователя:
Лісовиченко Олег Іванович

ID проверки:
1009292646

Дата проверки:
22.11.2021 13:36:42 EET

Тип проверки:
Doc vs Internet + Library

Дата отчета:
22.11.2021 13:37:30 EET

ID пользователя:
76913

Название файла: IT-з03мп_Колпак

Количество страниц: 73 Количество слов: 9808 Количество символов: 87907 Размер файла: 187.58 KB ID файла: 1009319035

1.21% Совпадения

Наибольшее совпадение: 0.19% с источником из Библиотеки (ID файла: 1003739496)

0.51% Источники из Интернета | 13 .... Страница 75

1.06% Источники из Библиотеки | 126 .... Страница 75

0% Цитат

Исключение цитат выключено

Исключение списка библиографических ссылок выключено

0% Исключений

Нет исключенных источников

Модификации

Обнаружены модификации текста. Подробная информация доступна в онлайн-отчете.

Замененные символы | 8