ДИПЛОМНА РОБОТА
БАКАЛАВРА

ДРБ. ІП - 20.00.00.000 ПЗ

Група ІП-20-2

Радченко Ярослав

2024

Міністерство освіти і науки України
Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

УДК 004.4

ДИПЛОМНА РОБОТА БАКАЛАВРА

Тема: “ Розробка програмного забезпечення системи обробки та агрегації різнорідних даних ”

(назва згідно з наказом ректора)

спеціальність – 121 “Інженерія програмного забезпечення”

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДРБ.ІП - 20.00.00.000 ПЗ

(позначення)

Консультанти:

з питань нормоконтролю

асист. /М.М. Піх/
(посада) (підпис) (дата)(розшифровка підпису)

Студент

ІП-20-2 /Я.Р.Радченко/
(шифр групи) (підпис) (дата) (розшифровка підпису)

Керівник роботи

к.т.н., доц. /В.Я.Піх/
(посада) (підпис) (дата) (розшифровка підпису)

Допускається до захисту
Завідувач кафедри

к.т.н., доц. /В.В.Бандура/
(посада) (підпис) (дата) (розшифровка підпису)

2024
(рік)

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітньо-кваліфікаційний рівень бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ____ ” _____2024 р.

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ БАКАЛАВРА СТУДЕНТОВІ

Радченку Ярославу Руслановичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “ Розробка програмного забезпечення системи обробки та агрегації різнорідних даних”

керівник проекту (роботи) Піх Володимир Ярославович, доцент

затверджені наказом закладу вищої освіти від **“** ____ ” травня 2024 р. № _____

2. Строк подання студентом проекту (роботи) 17 червня 2024 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд концепцій функціонування агрегаторів різнорідних даних
2. Розробка програмної моделі системи для агрегації даних
3. Проектування структури та алгоритмів функціонування системи
4. Програмна реалізація функціональності системи обробки та агрегації різнорідних даних
5. Виконання процесу тестування розробленого програмного забезпечення

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Загальна схема роботи сервісу з прикладним програмним інтерфейсом (рис. 1.1)
2. Приклад розмітки XML-формату (рис. 1.2)
3. Приклад агрегування по часу (рис. 1.7)
4. Приклад просторової агрегації (рис. 1.8)
5. Метапошуковий агрегатор metaGer (рис. 1.11)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **22 травня 2024 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Огляд концепцій функціонування агрегаторів різнорідних даних	25.05.2024	виконано
2	Розробка програмної моделі системи для агрегації даних	29.05.2024	виконано
3	Проектування структури та алгоритмів функціонування системи	30.05.2024	виконано
4	Програмна реалізація функціональності системи обробки та агрегації різнорідних даних	04.06.2024	виконано
5	Тестування ефективності розробленого програмного забезпечення	10.06.2024	виконано
6	Оформлення пояснювальної записки дипломної роботи завідувачем кафедри	17.06.2024	виконано

Студент – дипломник _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Робота присвячена розробці програмного забезпечення для системи обробки та агрегації різнорідних даних. У роботі розглядаються основні проблеми, пов'язані з обробкою та агрегацією різнорідних даних, такі як розбіжність форматів, різноманітність джерел та потреба в стандартизації даних. Представлено архітектурні рішення для побудови системи, що дозволяє ефективно збирати, обробляти та агрегувати дані з різних джерел. Результатом роботи є програмне забезпечення, яке дозволяє користувачам ефективно обробляти та аналізувати різнорідns даних з метою отримання цінної інформації.

КЛЮЧОВІ СЛОВА: АГРЕГАЦІЯ ДАНИХ, ОБРОБКА ДАНИХ, ФОРМАТ JSON, СТРУКТУРА ЗАПИТУ, РОЗПІЗНАВАННЯ ТЕКСТУ, ПАРАМЕТРИЗОВАНИЙ ЗАПИТ

ANNOTATION

The work is devoted to the development of software for the system of processing and aggregation of heterogeneous data. The work considers the main problems related to the processing and aggregation of disparate data, such as the disparity of formats, the variety of sources and the need for data standardization. Architectural solutions for building a system that allows efficient collection, processing and aggregation of data from various sources are presented. The result is software that allows users to efficiently process and analyze a variety of data in order to obtain valuable information.

KEYWORDS: DATA AGGREGATION, DATA PROCESSING, JSON FORMAT, QUERY STRUCTURE, TEXT RECOGNITION, PARAMETERIZED QUERY

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 8
ВСТУП 10
1. ОГЛЯД КОНЦЕПЦІЙ ФУНКЦІОНУВАННЯ АГРЕГАТОРІВ РІЗНОРІДНИХ ДАНИХ 12
1.1. Проблема уніфікованого доступу до інформації 12
1.2. Аналіз основних способів та форматів обміну даними 14
1.3. Аналіз типів агрегації даних 20
1.4. Вимоги до програмної реалізації сервісу для агрегації різнорідних даних 24
1.5. Аналіз існуючих сервісів агрегації даних 26
1.6. Висновки до розділу 33
2. РОЗРОБКА ПРОГРАМНОЇ МОДЕЛІ СИСТЕМИ ДЛЯ АГРЕГАЦІЇ ДАНИХ 34
2.1. Аналіз принципу роботи системи 34
2.2. Проектування структури та алгоритмів функціонування системи 43
2.3. Висновки до розділу 47
3. ПРОГРАМНА РЕАЛІЗАЦІЯ ФУНКЦІОНАЛЬНОСТІ СИСТЕМИ ОБРОБКИ ТА АГРЕГАЦІЇ РІЗНОРІДНИХ ДАНИХ 48
3.1. Вибір засобів та інструментів згідно з вимогами до розробки 48
3.2. Програмна розробка основних модулів системи агрегації даних 52
3.3. Тестування функціоналу системи обробки та агрегації даних 68
3.4 Висновки до розділу 73

ВИСНОВКИ 74
ПЕРЕЛІК ПОСИЛАНЬ НА ДЖЕРЕЛА 75
ДОДАТКИ
БІБЛІОГРАФІЧНА ДОВІДКА

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Серіалізація – процес перетворення об’єктів у послідовність бітів. Цей процес дозволяє зберігати об’єкт у файл чи надсилати його мережею.

Десеріалізація – процес, зворотній до серіалізації; дозволяє відновити початковий об’єкт з послідовності бітів. Застосовується при отриманні об’єкта із зовнішнього джерела.

Unicode – Юнікод – стандарт кодування символів, що вміщає знаки всіх мов світу та інші символи.

JavaScript – мова програмування. Найчастіше застосовується як мова сценаріїв, що надає інтерактивності веб-сторінкам.

AJAX – Asynchronous JavaScript and XML – підхід до проектування інтерфейсів веб-додатків, за якого відбувається інтерактивне оновлення даних у «фоновому» режимі, коли сторінка не перезавантажується.

DOM – Document Object Model – програмний інтерфейс, який дозволяє отримувати доступ до XML чи HTML документів.

CSS – Cascading Style Sheets – каскадна таблиця стилів; мова розмітки стилів, що використовується для опису зовнішнього вигляду сторінки, написаної на мові розмітки.

ШІ (Штучний Інтелект) – AI (Artificial Intelligence) – властивість інтелектуальних систем виконувати творчі функції, які вважаються подібними до людської діяльності.

Великі дані – Big Data – набір даних такого великого розміру, що потребують особливого способу роботи з ними.

Нейронні мережі – Neural Networks – інформаційні обчислювані системи, побудовані за принципом роботи біологічних нейронних мереж.

URL – Uniform Resource Locator – стандартизована адреса ресурсу в інтернеті.

URI – Uniform Resource Identifier – універсальний ідентифікатор абстрактного або фізичного ресурсу.

HTTP – протокол передачі даних прикладного рівня.

JDBC – Java DataBase Connectivity – стандарт взаємодії Java-застосунків з базами даних.

ORM – Object-Relational Mapping – технологія, яка пов’язує бузи даних з принципами ООП.

Parsing - парсинг – синтаксичний аналіз.

Артефакт – небажані особливості графічного зображення на екрані, що з’являються як результат роботи технічних чи програмних засобів.

Mapping – процес створення схеми взаємодії двох моделей даних.

Чат-бот – комп’ютерна програма, яка інтерактивно взаємодіє з користувачем за допомогою інтерфейсів.

ВСТУП

Агрегація даних – це процес, яким люди займаються протягом всього свого життя. Будь-який вид діяльності людини пов’язаний зі збором та обробкою інформації, адже будь-який звіт, будь-яке ведення журналу – це спроба об’єднати та класифікувати інформацію в одному місці. Наприклад, квартальні звіти компанії передбачають статистику всього виробництва, підсумок всіх попередніх звітів кожного цеху. Кожен шкільний журнал класу несе в собі різну інформацію, об’єднану одним змістом. Сфера управління передбачає систему збору даних з підрядних відомств та передачу цих даних у надрядні органи. Задачу збору даних добре розуміють оператори пультів управління, де потрібно спостерігати за станом інформаційної системи за допомогою багатьох дисплеїв та датчиків. Це дуже важлива й клопітка робота, яка вимагає високого рівня концентрації уваги. Було б дуже незручно, якби потрібна інформація розміщувалась за межами досяжності відповідного працівника - спостереження потребувало би більше часу й уваги, або й більшої кількості працівників.

Задача проектування інформаційних агрегаторів лежить на поверхні розв’язку проблем ефективності при роботі з розгалуженими системами. Ба більше – сучасний спосіб ведення бізнесу не може ефективно існувати у такій формі, у якій він існує зараз, без застосування агрегуючих систем, адже вони вирішують спектр проблем пов’язаних із відображенням, обробкою та передачею даних задач.

Предметом дослідження дипломного проєкту є агрегація різнорідних даних. Актуальність роботи залишається на високому рівні, адже агрегатор може бути як і простим мостом даних, так і застосовувати методи штучного інтелекту для аналізу та опрацювання даних, проведення аналітики та статистики та, на основі них, приймати рішення. Основним завданням

дипломного проєкту є створення агрегатора, який працює з гетерогенними даними та джерелами з різним інтерфейсом.

Об’єктом агрегування даних є провайдери перекладу. Даний напрямок обумовлений професійними потребами перекладачів та студентів, які часто потребують взаємодії з різними джерелами – перекладачі, словники, аналізатори документів. Сьогодні словники містяться у відкритому онлайн доступі, але зачасту не організовані в один ресурс і, як наслідок, потребують багато часу на доступ до них. Сервіс-агрегатор здатний забезпечити уніфіковану роботу з даними сервісами та надати оптимізацію пошукових запитів завдяки опрацьованій вибірці з декількох джерел.

1. ОГЛЯД КОНЦЕПЦІЙ ФУНКЦІОНУВАННЯ АГРЕГАТОРІВ РІЗНОРІДНИХ ДАНИХ

1.1. Проблема уніфікованого доступу до інформації

Сьогодні існує дуже багато веб-сервісів, у яких одна і та ж ціль, але підхід до виконання – різний. Як наслідок, й інтерфейси взаємодії з користувачами теж різні, що накладає обмеження у швидкості роботи користувача з веб-сервісом. Зважаючи на те, що інформація, яка надається різними веб-сервісами (веб-сайтами), знаходиться в різних місцях мережі, користувач витрачає багато часу у пошуках потрібних сервісів аби побачити результат їх роботи. Якщо користувач є працівником та виконує виробничі завдання, то такий режим роботи розсіює увагу та призводить до втрат ефективності на робочому місці.

Розглядаючи проблеми оптимізації та аналізу однотипних по змісту даних, потрібно чітко визначити інформаційні об’єкти, які аналізуються, та процеси, що йдуть після такого аналізу. Правильний вибір об’єкта аналізу та процесу його обробки дозволяє ефективно агрегувати дані та надавати результативний підсумковий звіт обробки цих даних. Існує клас сервісів, які спеціалізуються на зборі даних, їх аналізі та формуванні результатів. Такі сервіси-агрегатори також можуть надавати додаткові аналітичні інструменти для роботи з даними.

Агрегатори можуть не тільки відображати актуальний стан інформаційних об’єктів, а й агрегувати запити до сервісів, що надають суміжний спектр послуг. Типовим прикладом є сервіси, які надають послуги роботи із мовними словниками. Такі сервіси, по-суті, агрегують словникові ресурси та надають єдиний інтерфейс для роботи з різними за структурою та наповненням джерелами інформації.

Сьогодні існує велика кількість провайдерів перекладу, які надають змогу працювати з текстом. Основними завданнями сервісів перекладу та роботи з текстом є:

- Переклад слів, словосполучень різними мовами;
- Змістовний переклад речень, текстів різними мовами;
- Розпізнавання звукової інформації;
- Аналіз текстових документів та їх змістовний переклад;
- Розпізнавання тексту на фото;
- Озвучення тексту машинним голосом;
- Робота з енциклопедичними, багатомовними, тлумачними та іншими словниками.
- Переклад веб-сторінок, веб-сайтів;
- Проводити перевірку на плагіат, порівнювати тексти, документи.

Існує широкий вибір сервісів для роботи із текстом. Професійні перекладачі використовують онлайн словники з повним зібранням можливих перекладів, в той час коли новачки у вивченні мов та люди, які не володіють іноземними мовами, використовують переклад текстів, розпізнавання голосу чи переклад документів. Крім того, виконання деяких завдань в різних ситуаціях відбувається різним чином. До прикладу, переклад речень та великих текстів потребує застосування не тільки словників, а й алгоритмів нейронних мереж для змістовного аналізу тексту. Змістовне розпізнавання тексту з голосу чи світлини передбачає використання сучасних бібліотек та технологій ШІ. Також у різних провайдерів варіюється спектр доступних для роботи мов, що накладає відчутне обмеження на роботу з цими сервісами. Відрізняється також і спосіб перекладу, кількість доступних словників та відкрита спільнота, що займається підтримкою сервісу.

Таким чином, створення сервіс-агрегатора провайдерів перекладу гетерогенних даних має за мету створити програмний продукт, який надає єдиний зручний інтерфейс для роботи з різними провайдерами, передбачаючи

гнучке налаштування засобів перекладу. Зважаючи на те, що розпізнавання та змістовний переклад є завданнями аналізу великих даних, навчання нейронних мереж та інших обчислень, перш за все, варто реалізувати роботу з сервісами, що надають можливість використання власних потужностей задля вирішення даного спектру завдань.

Хмарні обчислення дозволяють зручно отримувати доступ до обчислювальних ресурсів через мережу. Існує багато провайдерів хмарних обчислень, але тільки деякі надають API для роботи з перекладом:

1. IBM Cloud
2. Google Cloud Platform (GCP)
3. Microsoft Azure
4. Amazon Web Services (AWS)

Вибір хмарного сервісу полягає у доступному функціоналі, тобто такий сервіс має виконувати якнайширший спектр завдань, які покладені на нього, а також забезпечувати достатню швидкодію, адже агрегатори, зважаючи на принцип їх роботи, не володіють високою швидкодією.

1.2. Аналіз основних способів та форматів обміну даними

Для взаємодії між користувачами та сервісами існує велика кількість різноманітних стандартів, протоколів, архітектурних рішень, мов розмітки та форматів. Дане різноманіття засобів взаємодії дає змогу створювати гнучке програмне забезпечення, яке здатне взаємодіяти з сервісами одразу ж з різними форматами передачі даних. Також, не всі користувачі мають змогу реалізувати доступ за єдиним шаблоном.

1.2.1 Прикладні програмні інтерфейси

Комп’ютерні програми часто використовують функції інших програм. Для упорядкування такого виду взаємодії існує прикладний програмний

інтерфейс (API), який описує протоколи взаємодії різних компонентів, як це показано на рисунку 1.1. Таки підхід дозволяє пришвидшити розробку програмного забезпечення завдяки використанню бібліотек, програм, утиліт, низькорівневих функцій. API використовується як для операційних систем, програмних бібліотек, апаратного забезпечення, баз даних, так і для веб-базованих програм.

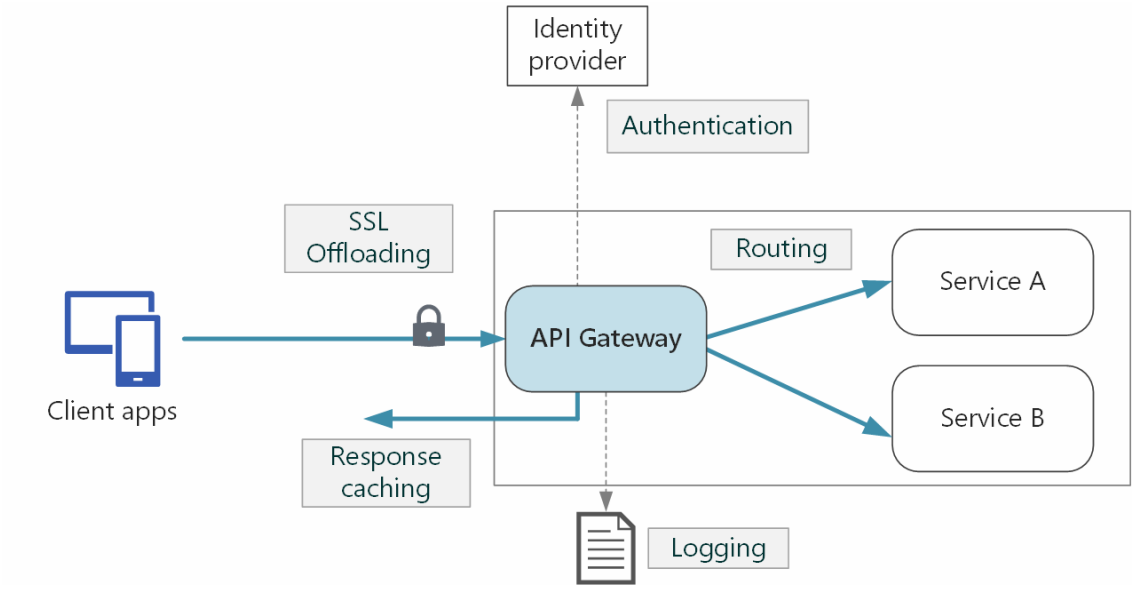


Рисунок 1.1 – Загальна схема роботи сервісу з прикладним програмним інтерфейсом

Перевага використання API полягає в його логічній зрозумілості, а також у документуванні. Більшість API є детально документованими та описаними, що спрощує розробку. Так, добре документованим є API програмних бібліотек, апаратного забезпечення та системних утиліт.

Прикладний програмний інтерфейс часто використовується і у веб-розробці. Такий інтерфейс описує способи взаємодії з сервісом, визначає набір можливих запитів, а також визначає формат та структуру відповіді, описує способи створення запитів та їх передачі.

Зважаючи на те, що робота агрегатора базується на обміні даними з різноманітними сервісами, прикладний програмний інтерфейс є основним

способом взаємодії, адже такий спосіб передбачає швидку передачу потрібних даних.

1.2.2 Основні формати обміну даними

Для ефективної взаємодії між програмами потрібен формат даних, який буде оптимальним для передачі різнотипних даних. Існує декілька основних форматів даних, що виконують схожі задачі, але по-різному реалізуються:

1. XML (*eXtensible Markup Language)* – розширювана мова розмітки, яка, в тому числі, застосовується й для передачі даних за допомогою мережевих протоколів. Цей стандарт є дуже гнучким та дозволяє описувати складні структури даних за допомогою простих тегів, що зображено на рисунку 1.2. Такий формат передачі даних добре відомий ще з початку 2000-х років, існує багато API, програмних бібліотек та документації для створення та читання XML-документів, проте ці процеси є досить ресурсозатратними та, як наслідок, повільними.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recipe PUBLIC "-//Happy-Monkey//DTD RecipeBook//EN"
"http://www.happy-monkey.net/recipebook/recipebook.dtd">

<recipe>

    <title>Peanut-butter On A Spoon</title>

    <ingredientlist>
        <ingredient>Peanut-butter</ingredient>
    </ingredientlist>

    <preparation>
        Stick a spoon in a jar of peanut-butter,
        scoop and pull out a big glob of peanut-butter.
    </preparation>

</recipe>
```

Рисунок 1.2 – Приклад розмітки XML-формату

2. RSS (Rich Site Summary) – це формат, який є діалектом XML і застосовується для відображення новин, статей, блогів. Всі RSS документи повинні відповідати стандарту XML (рисунок 1.3) [3]. RSS-формат створений для зручної трансляції змін з джерела до клієнта. Таким клієнтом часто виступає сервіс-агрегатор. RSS створено задля оптимізації новин з веб-сайтів, а зараз зазвичай використовується для передачі опису актуальної інформації та посилання на її отримання.

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
  <title>W3Schools Home Page</title>
  <link>https://www.w3schools.com</link>
  <description>Free web building tutorials</description>
  <item>
    <title>RSS Tutorial</title>
    <link>https://www.w3schools.com/xml/xml_rss.asp</link>
    <description>New RSS tutorial on W3Schools</description>
  </item>
  <item>
    <title>XML Tutorial</title>
    <link>https://www.w3schools.com/xml</link>
    <description>New XML tutorial on W3Schools</description>
  </item>
</channel>

</rss>
```

Рисунок 1.3 – Приклад розмітки RSS-формату

3. YAML (Yet Another Markup Language) – формат серіалізації даних, який концентрує увагу на даних, а не на розмітці документу. YAML базується на Unicode та покликаний бути легким, зрозумілим людині та універсальним засобом передачі даних. Головною перевагою є можливість відображати типові моделі даних, описувати вільні послідовності та відсутність DOM-

моделей. На рисунку 1.4 зображено форма YAML та тим самим продемонстровано легкість його сприйняття людиною.

```yaml
---
# An employee record
name: Martin D'vloper
job: Developer
skill: Elite
employed: True
foods:
    - Apple
    - Orange
    - Strawberry
    - Mango
languages:
    perl: Elite
    python: Elite
    pascal: Lame
education: |
    4 GCSEs
    3 A-Levels
    BSc in the Internet of Things
```

Рисунок 1.4 – Приклад синтаксису YAML-формату

4. HTML – мова розмітки (рисунок 1.5), яка є основною мовою у мережі Інтернет [6]. Базується на SGML – спільному предку з форматом XML. HTML-формат сьогодення представлений станлартом HTML-5, який є провідним на ринку. Цей формат знайшов своє використання у графічному зображенні інформації для клієнта-браузера, у поєднанні з CSS.

```html
<!DOCTYPE>
<html>

<head>
<titile> Demo </title>
</head>

<body>
<p> this is first demo of html structure </p>
</body>

</html>
```

Рисунок 1.5 – Приклад розмітки HTML-формату

5. JSON (JavaScript Object Notation) – текстовий формат обміну даними, який пішов від JavaScript. Легко читається людьми, лаконічний формат, що підходить для серіалізації складних об'єктів та структур (рисунок 1.6). Завдяки походженню від JavaScript, може бути легко десеріалізований вбудованим функціоналом мови [5]. Формат JSON часто використовується для обміну даними між браузером і сервером при реалізації технології AJAX, а також для обміну даними між додатками.



Рисунок 1.6 – Приклад синтаксису JSON-формату

Дані формати можуть застосовуватись як і окремо, так і у поєднанні. Особливо поширено таке поєднання при використанні програмних інтерфейсів.

1.2.3 SOAP і REST

Використовуючи різні формати обміну повідомленнями та прикладний програмний інтерфейс, можна забезпечити швидку взаємодію з іншими сервісами. При проектуванні даної системи виникає проблема вибору й організації API та форматів обміну даними. Існує два підходи до створення оптимального архітектурного рішення: REST та SOAP.

SOAP (Simple Object Access Protocol) – це сімейство стандартів і протоколів, які призначені для обміну структурованими повідомленнями. SOAP використовує XML, який має наступну структуру:

- Envelope (конверт) – кореневий елемент, який визначає повідомлення та простір імен документу;
- Header (заголовок) – містить атрибути повідомлення (безпека, мережева маршрутизація);
- Body (тіло) – містить повідомлення;
- Fault (помилка) – містить повідомлення про помилки, якщо вони існують.

Такі запити можуть надсилатись за допомогою будь-якого протоколу (HTTP, HTTPS, FTP, SMTP та ін.). SOAP також використовує WSDL (Web Services Description Language) – базована на XML мова опису веб-сервісів та доступу до них. При використанні SOAP потрібно визначити сервіс за допомогою WSDL.

REST (Representational State Transfer) – це архітектурний стиль, який передбачає легку взаємодію між сервісами, використання якого може вести до збільшення продуктивності. Особливістю REST-сервісів є те, що вони дозволяють найкращим чином використовувати протокол HTTP.

1.3. Аналіз типів агрегації даних

Агрегування даних - це процес, в ході якого вихідні дані збираються і представляються в узагальненому вигляді для статистичного аналізу.

Агреговані дані в основному використовуються дослідниками та аналітиками, політиками, банками і адміністраторами з багатьох причин. Вони використовуються для оцінки політики, розпізнавання тенденцій і закономірностей процесів, отримання актуальної інформації і оцінки поточних заходів для стратегічного планування. Агреговані дані, зібрані з

різних джерел, використовуються в різних областях досліджень, таких як порівняльний політичний аналіз і науковий аналіз APD, для подальшого аналізу. Агреговані дані також використовуються в медичних і освітніх цілях. Агреговані дані широко використовуються, але вони також супроводжуються деякими обмеженнями, включаючи неточні висновки і помилкові висновки, які також називають "екологічним помилкою". "Екологічне оману" означає, що користувачі не можуть робити висновки про екологічні відносинах між двома кількісними змінними на індивідуальному рівні.

Існує два типи агрегування даних:

- Time aggregation

Всі точки даних для одного ресурсу за певний період часу.

- Spatial aggregation

Всі точки даних для групи ресурсів за певний період часу.

1.3.1 Time aggregation

Time aggregation (тимчасова агрегація) - це агрегація всіх точок даних для одного ресурсу за певний період (гранулярність). Агрегація даних в звітах Resource Time Series відноситься до типу тимчасової агрегації.

Результатом агрегування є одна точка даних, яка відображає статистичне уявлення зібраних і агрегованих точок даних. Наприклад, середнє значення, мінімум, максимум, сума або кількість. Як правило, в звіті за певний звітний період представлено кілька агрегованих точок даних.

На наступному рисунку 1.7 показано агрегування за часом.

На рисунку 1.7:

− Точки даних збираються з одного ресурсу (Ресурс А) кожні 5 хвилин (період опитування).

− Гранулярність для цієї агрегації становить 20 хвилин.

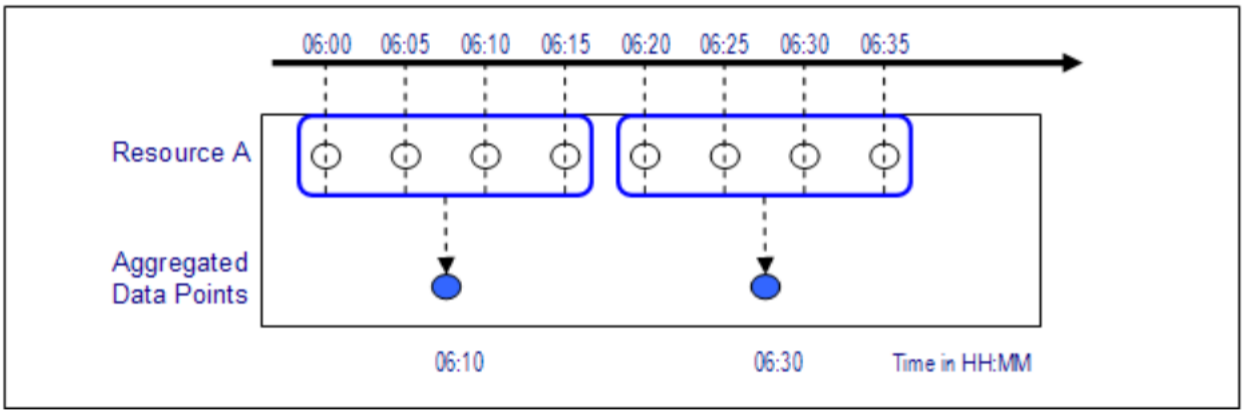


Рисунок 1.7 - Приклад агрегування по часу

– Точки даних, зібрані протягом кожного періоду гранулярності, об'єднуються в одне статистичне значення. Наприклад, середнє значення всіх зібраних точок даних.

– Тимчасова мітка для кожного агрегованого значення береться з середини періоду гранулярності. На цьому рисунку - 6:10 (на півдорозі між 6:00:00 і 6:19:59) і 6:30 (на півдорозі між 6:20:00 і 6:39:59).

1.3.2 Spatial aggregation

Spatial aggregation (просторова агрегація) - це агрегація всіх точок даних для групи ресурсів за певний період (гранулярність). Агрегація даних в звітах Group Time Series відноситься до типу просторової агрегації.

Результатом агрегування є одна точка даних, яка відображає деякий статистичне уявлення зібраних і агрегованих точок даних - наприклад, середнє значення, мінімум, максимум, сума або кількість. Зазвичай в звіті за певний звітний період представлено кілька агрегованих точок даних.

На рисунку 1.8 показана просторова агрегація, де Гранулярність = Період опитування.

На рисунку 1.8:

– Точки даних збираються з групи з трьох ресурсів (інтерфейс A, інтерфейс B і інтерфейс C) кожні 5 хвилин (період опитування).

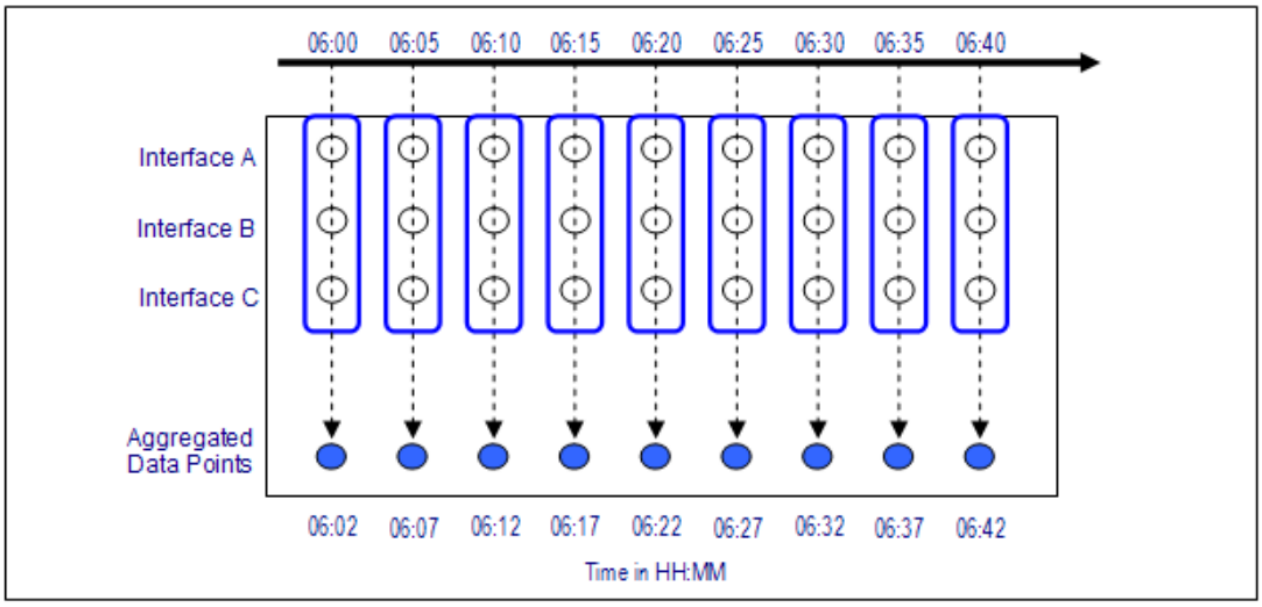


Рисунок 1.8 – Приклад просторової агрегації

– Гранулярність для цієї агрегації також становить 5 хвилин.

– Точки даних, зібрані протягом кожного періоду гранулярності, об'єднуються в одне статистичне значення. Наприклад, середнє значення всіх зібраних точок даних.

– Тимчасова мітка для кожного агрегованого значення береться з середини 5-хвилинного періоду гранулярності. Наприклад, 6.02 - це середина між 6:00:00 і 6:04:59 з урахуванням точності.

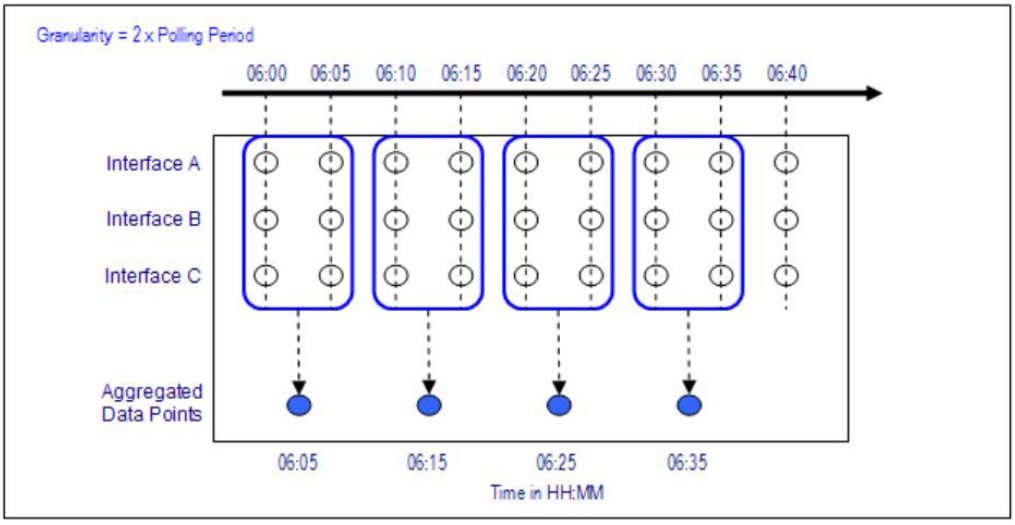


Рисунок 1.9 – Приклад просторової агрегації (змінена дискретність)

На наступному рисунку 1.9 показаний попередній випадок, але з дискретністю 10 хвилин. Зверніть увагу, що точки даних, зібрані за 5-хвилинний період опитування 6: 40: 00-6: 44: 59, ще не були включені в наступну 10-хвилинну агрегацію.

Таким чином, агрегатор - це сайт або додаток, що об'єднує однорідну інформацію або контент з Інтернету. Це спрощує кінцевому користувачу навігацію і пошук інформації в Інтернеті.

Коли потенційному клієнту або користувачу Інтернету доводиться переглядати безліч сторінок та інформації, щоб досягти своєї мети, на допомогу приходять агрегатори.

Отже, сервіс агрегатор надає однорідні послуги на своєму сайті. Він об'єднує постачальників послуг на своєму сайті і дозволяє клієнту/ користувачу скористатися такими послугами через свій сайт або додаток.

1.4. Вимоги до програмної реалізації сервісу для агрегації різнорідних даних

Мета розроблюваного сервісу полягає у швидкій агрегації даних, їх аналізі та поширенні. Перш за все важливою є швидкодія та її забезпечення. Складність полягає у особливостях реалізації даного типу сервісів: після запиту на переклад сервіс агрегатор звертається до потрібного джерела чи групи джерел, очікує відповіді, аналізує її, формує звіт на основі отриманих даних. Сам по собі, цей процес є ресурсозатратним та довготривалим, отже перша вимога – висока швидкодія обраних технологій.

Можливість роботи з різними джерелами є першочерговим завданням у проектуванні агрегатора. Ці джерела можуть мати різні інтерфейси та режими взаємодії, отже програмний продукт повинен забезпечувати всі можливі формати взаємодії з джерелами з можливістю аналізу отриманих даних.

Як відомо, сервіси-агрегатори користуються своєю популярністю саме через зручність доступу до них та змістовністю об’єднаної інформації. Використання прикладного програмного інтерфейсу дає можливість забезпечити швидкий та зручний доступ до агрегатора. Таким чином забезпечується швидке розширення програмного продукту наступними можливими інтерфейсами:

- веб-сайт;
- чат-бот;
- додаток на Android, iOS;
- застосунок;
- псевдографічний інтерфейс;
- прикладний програмний інтерфейс;

Також за даної реалізації інші сервіси та програмні продукти зможуть використовувати доступні функції агрегатора.

Для забезпечення прикладного програмного інтерфейсу сервіс-агрегатора доцільним є використання REST, адже ця концепція базується на ефективному використанні HTTP та його властивостей. Такий підхід передбачає легкість користування та швидкість передачі. Сервіс-агрегатор не вимагає підвищеної безпеки чи роботи з транзакціями, як у SOAP, а тому є доцільним використання простішої та швидкої концепції REST.

За концепції REST зручно використовувати формат JSON. На відміну від інших форматів обміну даних, JSON не потребує складного аналізу та орієнтований саме та передачу об’єктів. Крім цього, даний формат підтримується «з коробки» мовою JavaScript, зручний у використанні при створенні AJAX-сторінок та є підтримуваною багатьма програмними бібліотеками.

Як один з провайдерів перекладу застосовується Google Cloud Platform, адже він доступний через API. Висока швидкодія цього сервісу дозволяє користувачу комфортно працювати. Також GCP володіє найширшим спектром

мов, доступних для перекладу, та надає доступ до розпізнавання тексту з фото- та голосових повідомлень.

1.5. Аналіз існуючих сервісів агрегації даних

У зв'язку з полегшенням процесу організації доступу контенту для користувачів, надавачі послуг створюють сервіси-агрегатори. Це популярна практика, що дозволяє привабити більше клієнтів та поширити свій вплив на цільову категорію ринку.

RSS-агрегатори.

RSS-агрегатор – це клієнтська програма, яка виконує функцію автоматичного збору інформації з кола джерел. Такими джерелами можуть бути заголовки блогів, новин, підкастів чи відеохостингів. Користувач додає в програму-агрегатор бажані джерела інформації, після чого агрегатор, за допомогою форматів обміну даними RSS чи Atom, перевіряє джерела на предмет оновлення. Перевірка може виконуватись згідно зазначеного проміжку часу або за потребою. Інформація, отримана з джерел, може бути згрупованою та представленою у будь-якому вигляді.

RSS-стрічка – це документ у форматі XML, який регулярно оновлюється і вміщує в собі інформацію зазвичай новинного характеру. RSS-стрічки використовуються багатьма сайтами для видачі вмісту новин безпосередньо на робочий стіл користувача. Для читання RSS можуть використовуватися спеціальні програми.

RSS-агрегатори бувають двох типів:

- Веб-агрегатор. Такий агрегатор є веб-сервісом з вільним доступом.
- Програмний агрегатор. Цей вид агрегаторів є програмою, яка може бути вбудована в операційну систему або як розширення для браузера. Сучасні системи та браузери містять інструменти роботи з RSS. Також

програмні RSS-агрегатори можуть встановлюватись як окремі програми на мобільні операційні системи (Android, iOS).

Новинні агрегатори.

Цей вид агрегаторів, які автоматично аналізують новинні джерела та надають користувачу готові структуровані результати. Новинні агрегатори не просто агрегують дані, а проводять ретельний аналіз новин та розподіляють їх за:

- Кількістю цитат та посилань на дану новину;
- Датою публікації, порівнюючи її з іншими джерелами новин;
- Змістовим наповненням.

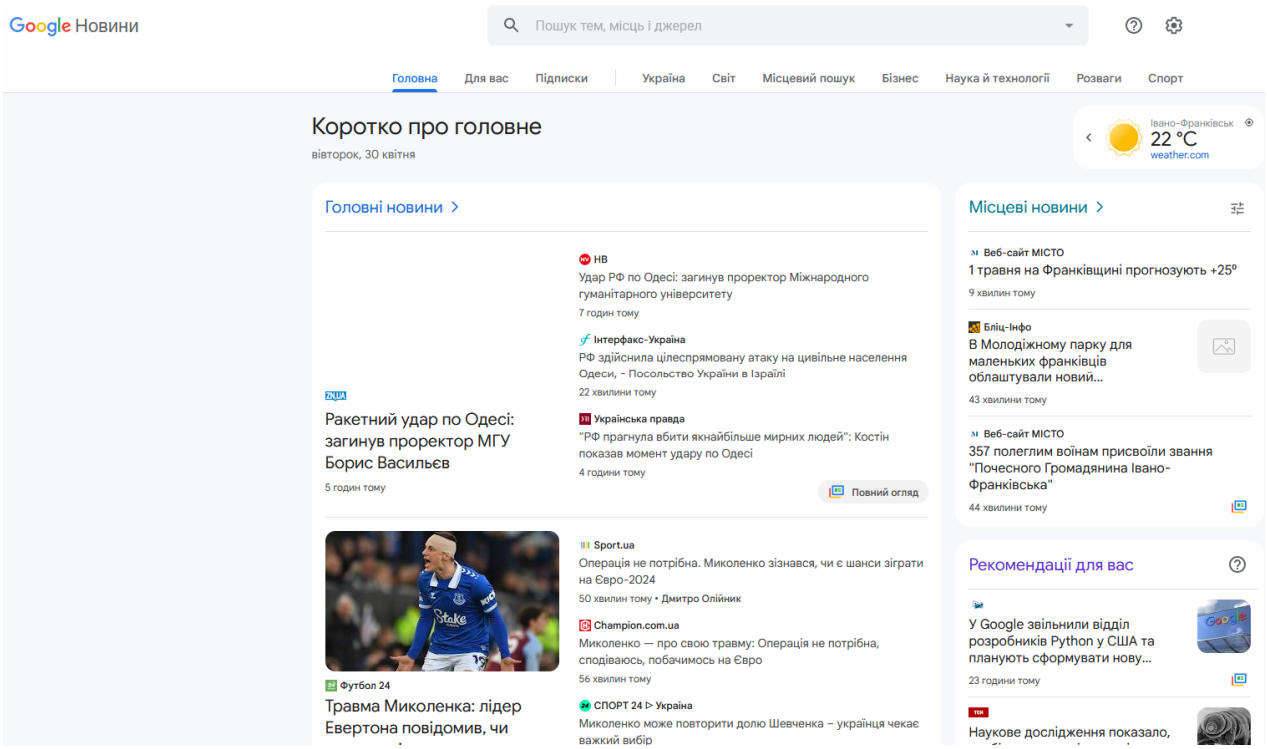


Рисунок 1.10 – Приклад новинного агрегатора (Google Новини)

За цими та супутніми їм критеріями проводиться ретельний алгоритмічний аналіз, який дозволяє надавати читачам адекватну підбірку новин, розбиту на категорії.

Агрегатори соціальних мереж.

Сервіс, який збирає інформацію з соціальних мереж, блогів та інших джерел в один ресурс, називається агрегатором соціальних мереж. Особливо корисним є для бізнесу (SMM та соціальні медіа), адже напрямок соціальних мереж - цільовий для багатьох маркетингових стратегій. Новинні агрегатори дають можливість оперативно розміщувати інформацію у декілька джерел і отримувати відгуки сукупно для всіх джерел, аналізувати їх, проводити обробку та готувати звіт згідно з результатом. Даний функціонал зачасту реалізований за допомогою REST API, які дають доступ до функцій та ресурсів користувачів, авторизовувати їх.

Агрегатори відео.

Ці сервіси дають змогу агрегувати відео з різних ресурсів. Агрегатори відео можуть використовуватись для колекціонування відео:

- професійного чи іншого спрямування;
- високої якості;
- обраних згідно з голосуванням користувачів;
- обраних модераторами;
- найпопулярніших на кожному ресурсі;
- для персонального використання та поширення між користувачами, обговорення та створення власних плейлистів;

Пошукові агрегатори.

Пошукові агрегатори працюють за принципом метапошукових систем, тобто теж формулюють результат на основі багатьох пошукових систем. Ці агрегатори поєднують визначені користувачем канали пошуку, що забезпечує високий рівень контролю пошуку користувачем.

Відколи був введений RSS-формат, сайти почали публікувати власні пошукові результати за допомогою RSS. Пошукові агрегатори є дуже

популярними саме через можливість використовувати переваги декількох пошукових систем одночасно. Такий спосіб є чудовою альтернативою типовим шляхам пошуку інформації. Для кінцевого користувача такий агрегатор може виглядати як ще один пошуковий сервіс з можливістю гнучких налаштувань, тобто такий агрегатор може бути неочевидним для користувача.

Типовий пошуковий агрегатор дозволяє користувачу обирати різні пошукові сервіси для опрацювання свого запиту. Агрегатор маніпулює параметризованими URL, куди в якості параметру вставляється користувацький пошуковий запит. Такий параметризований URL може мати наступний вигляд: *https://www.google.com/search?q={SEARCH_TERMS}&ie=UTF-8&output=rss*, де параметр {SEARCH_TERMS} буде замінено запитом користувача.

Пошукові агрегатори мають декілька переваг над традиційними пошуковими системами. Головна перевага – можливість користувачеві обирати які пошукові сервіси повинні опрацьовувати його запит. Також можна без проблем додавати нові пошукові системи до особистих колекцій.

Наприклад, metaGer – це метапошуковий агрегатор, заснований у 1996 році німецькою некомерційною організацією SUMA-EV (Асоціація вільного доступу до знань та інформації).

В MetaGer відсутня реклама і цензура. Він об’єднує в собі результати різних пошукових систем. Як запевняють розробники: “Наші алгоритми прозорі та доступні для читання всім. Наш вихідний код безкоштовний. Для нас важливі стабільність та ефективне використання ресурсів. Усі наші послуги працюють з використанням відновлюваної енергії. Від сервера до кавоварки”.

Пошук в MetaGer може здійснюватися по Вебу, Зображенням, Товарам, Новинам/Політиці. Присутній також Пошук за асоціацією. Є непогане

керівництво по факт-чекінгу (Fact Checking). Власники сайтів можуть вбудувати MetaGer собі на сторінку у вигляді віджета.

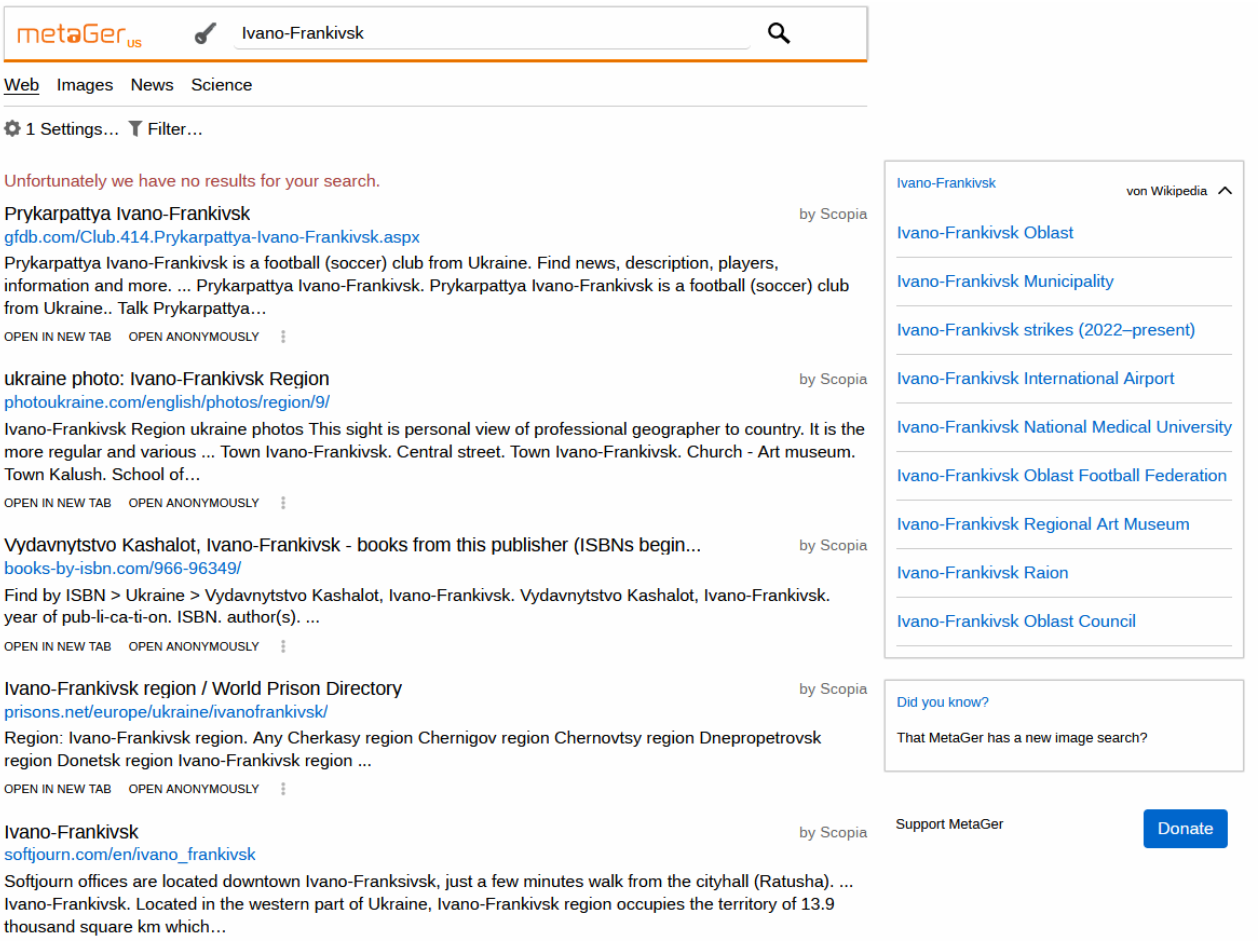


Рисунок 1.11 - Метапошуковий агрегатор metaGer

Агрегатори робочих оголошень.

Це сервіси, які використовують дошки, веб-сайти оголошень, для збору та класифікації пропозицій роботи. Зручні для людей, які знаходяться у пошуку роботи, та агенціями зайнятості, які пропонують роботу.

Агрегатор опитувань.

Агрегатор опитувань є системою, яка відстежує та агрегує дані опитувань шляхом аналізу джерел окремих опитувань з метою ведення загальної статистики. Такі агрегатори здатні прогнозувати ймовірні результати виборів,

збираючи та аналізуючи передвиборчі опитування, проведені різними людьми чи організаціями. Також сервіс може опрацьовувати зібрану інформацію згідно з доступними судженнями про політику, які можуть впливати на результат виборів. Наприклад, можна прогнозувати результати виборів на основі поточних опитувань, а також згідно з веб-трафіком за певний період до виборів.

Алгоритмізація агрегування та аналізу опитувань може бути різною. Деякі сервіси визначають середнє значення опитувань в короткостроковому чи довгостроковому проміжку, інші ж можуть надавати перевагу новішим опитуванням, коли давніші опитування прийматимуться до уваги, але з меншою вагою. Існує також багато параметрів, які можуть впливати на результат, як от: вік опитування, соціальний стан, стать й інші особливості чи алгоритми, які рахуватимуть підсумкові значення. Використання тільки усереднених значень не є ефективним, адже не враховуються інші параметри, як от розмір вибірки. Сьогодні використовують додаткові моделі, пов’язані з історичним розвитком виборів чи економічними даними.

Method	Biden	Trump
- PollyVote	52	48
+ Polls	53.9	46.1
+ Models	52.4	47.6
- Expectations	51.5	48.5
- Betting markets	52.3	47.7
IEM	52.9	47.1
Predictit	51.6	48.4

Рисунок 1.12 – Джерела агрегатора агрегаторів PollyVote у опитуванні «2020. Trump vs. Biden»

Агрегатори не здатні визначати помилки самих опитувань, адже коли опитувальники неправильно оцінюють параметри виборчого процесу або свідомо корегують результати, сервіси-агрегатори відображатимуть це так само.

Агрегаторів опитувань існує велика кількість, адже вибори проходять майже в усіх країнах світу і потреба у зважених попередніх результатах існує завжди. Прикладом є європейський *pollofpolls.eu* та австралійський агрегатор *BludgerTrack*.

Також існують агрегатори агрегаторів, які мають на меті об'єднати об'єднані дані різних джерел. Такий підхід, можливо, дає стабільніші результати, хоча математично це не доведено та не спростовано. Таким агрегатором агрегаторів є PollyVote, який агрегує дані з інших агрегаторів, які у свою чергу керуються власними джерелами, що демонструється рисунком 1.12.

Агрегатори рецензій.

Агрегатори рецензій – це сервіси, які збирають рецензії (огляди) на книги, фільми, товари, відеоігри і т. д. Ці дані можуть використовуватись для аналізу тенденцій, продажу, дослідження потенційних споживачів. Така система агрегації дозволяє швидко та зручно проводити огляд продуктів та послуг. Також системи агрегації рецензій дозволяють вираховувати оцінки на основі відгуків користувачів.

Дані агрегатори суттєво впливають на ринок товарів і послуг, який описують, адже відгуки та оцінки є важливою передумовою вибору товару, тому багато компаній зацікавлені в хорошій оцінці продукції.

Прикладом агрегаторів рецензій є *Metacritic* – сервіс, який збирає відгуки про кіно, ТБ, музику, відеоігри та інше. Відомим сервісом музичних рецензій також є *Last.fm*. Агрегатор книжковий рецензій – *Book Marks*.

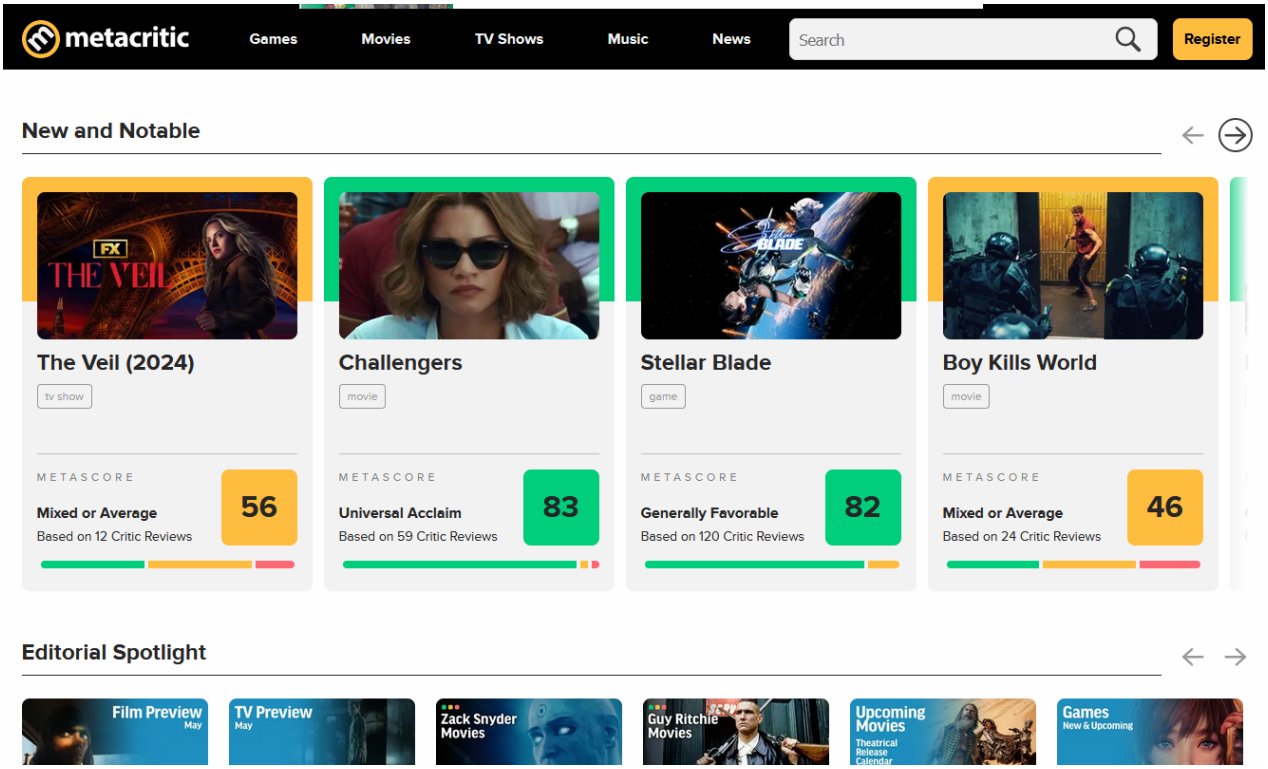


Рисунок 1.13 - Агрегатор рецензій Metacritic

1.6. Висновки до розділу

В даному розділі було розглянути проблему агрегації даних. Виконано аналіз засобів та методів для реалізації сервіс-агрегатора, а також здійснено огляд існуючих сервісів агрегації даних.

2. РОЗРОБКА ПРОГРАМНОЇ МОДЕЛІ СИСТЕМИ ДЛЯ АГРЕГАЦІЇ ДАНИХ

2.1. Аналіз принципу роботи системи

Робота агрегатора полягає у зборі, опрацюванні та видачі потрібних даних. Схематично цей процес зображено на рисунку 2.1.

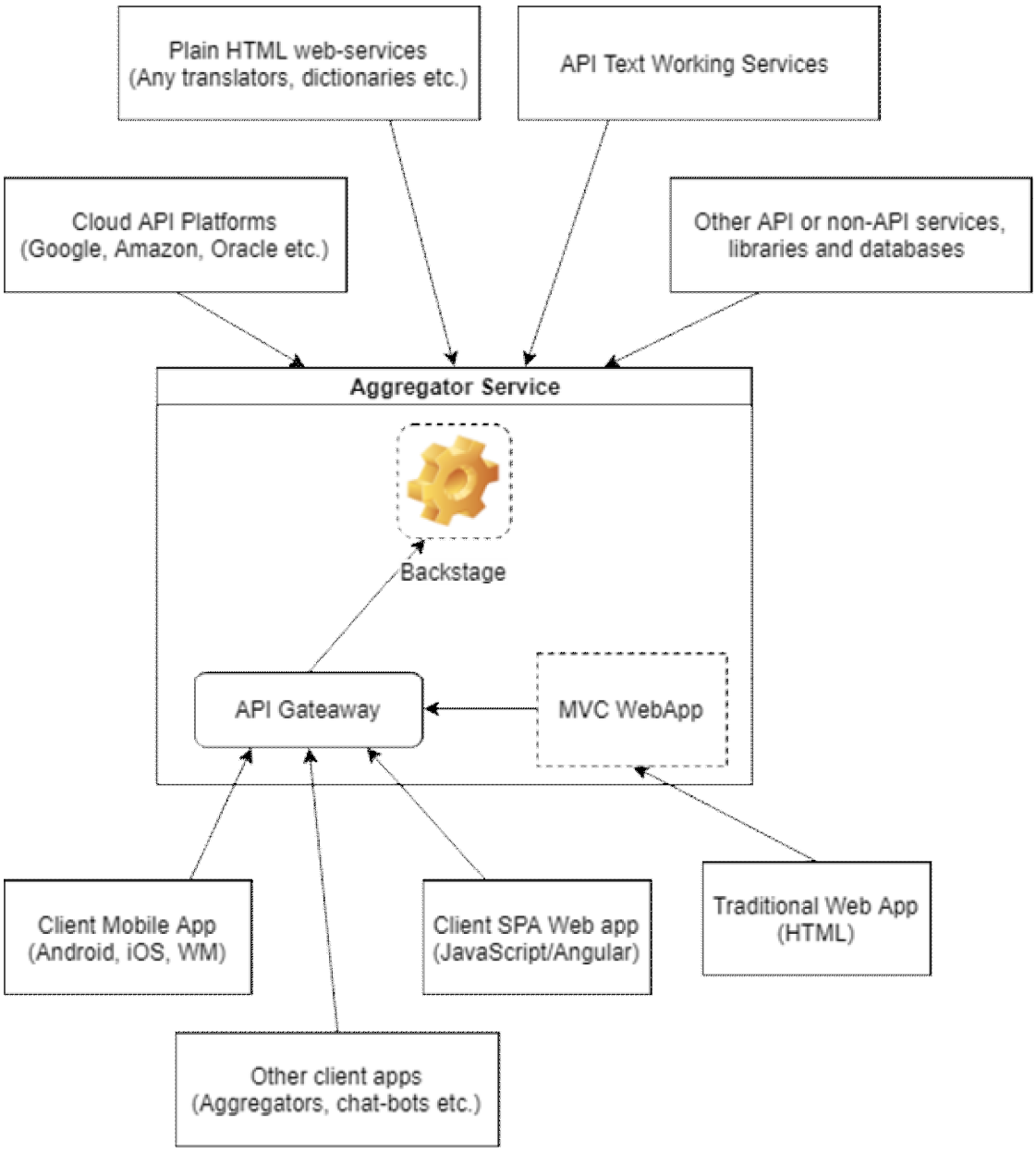


Рисунок 2.1 – Робота API сервіс-агрегатора

Реалізуючи дану модель розглядається три основних задачі:

1. Агрегація даних – проблема роботи з джерелами.
2. Аналіз та опрацювання даних.
3. Поширення результатів роботи сервіс-агрегатора.

2.1.1. Поширення результатів роботи сервіс агрегатора.

Взаємодія клієнтських застосунків з сервісом-агрегатором передбачає роботу з API, що не накладає суттєвих обмежень на передачу даних. Обраний архітектурний стиль REST визначає способи передачі даних, у свою чергу API визначає режим роботи з сервісом. В сукупності дані засоби формують поняття «RESTful API» - це прикладний програмний інтерфейс, побудований за принципами REST.

Ключовою складовою даних в REST є ресурс – відображення того, що потрібно передати у обраному форматі. У агрегаторі таким ресурсом є *Результат*, який відображає сукупній результат виконання агрегації.

Для ідентифікації певного ресурсу використовується URI. У поєднанні з протоколом HTTP це дає наступну схему запитів:

- POST /users – створює користувача
- DELETE /users/1 – видаляє користувача з ID 1
- GET /users – повертає список користувачів
- GET /users/1 – повертає ресурс, який містить користувача з ID 1

Схема доступу до ресурсів агрегатора:

- /services – повертає список користувачів;
- /services/{service_id} – повертає сервіс перекладання;
- /services/{service_id}/languages – повертає список доступних для перекладу мов даного сервісу;
- /services/{service_id}/?sl={source_language}&tl={target_language}&text={text} – повертає переклад обраного сервісу перекладу.
- /vision

Відповідь містить дані у форматі JSON.

2.1.2. Агрегація даних.

Для роботи з різноманітними джерелами перекладу застосовується комплексний підхід.

Робота з Google Cloud Platform.

Для роботи з хмарними технологіями Google існують зручні бібліотеки, які можна під'єднати до проекту як залежності, що зображено на рисунку 2.2.

Google Cloud Translate дає змогу динамічно перекладати текст між більш ніж ста мовами, використовуючи навчені моделі машинного навчання. Також можна використовувати власні моделі та під'єднувати їх до Google Translate. Доступна функція автовизначення мови для випадків, коли користувач не впевнений чи не знає мови, яку він використовує. В такому випадку, застосовуючи машинне навчання, автоматично з високою точністю визначається введена мова. Для цього, а також для точно перекладу, застосовуються глосарії, які містять вживані слова та словосполучення.

```
<dependencies>
  <dependency>
    <groupId>com.google.cloud</groupId>
    <artifactId>google-cloud-translate</artifactId>
  </dependency>
```

Рисунок 2.2 – Під'єднання залежності Google Cloud Translate.

Для виконання запитів використовується URI *https://translation.googleapis.com/language/translate/v2*. Важливою є система версій API, яка вказується вкінці шляху «v2». Система версій дозволяє контролювати режим розширення функціоналу.

```
curl -s -X POST -H "Content-Type: application/json" \
    -H "Authorization: Bearer "$(gcloud auth application-default print-access-token)
    --data "{
  'q': 'The Great Pyramid of Giza (also known as the Pyramid of Khufu or the
       Pyramid of Cheops) is the oldest and largest of the three pyramids in
       the Giza pyramid complex.',
  'source': 'en',
  'target': 'es',
  'format': 'text'
}" "https://translation.googleapis.com/language/translate/v2"
```

Рисунок 2.3 – Структура запиту до Google Translation API

Структура запиту показана на рисунку 2.3. Перш за все, вказується тип запиту POST, адже це відповідає контракту. В head вказуються атрибути *Content-Type* та *Authorization*. Перший атрибут вказує на тип контенту – формат JSON. Поле *Authorization* містить у собі так званий «токен» - унікальний ключ, що дозволяє авторизувати користувача в Google Cloud Platform. Даний токен є обов'язковим, без нього запит не пройде.

Тіло запиту містить обов'язкові елементи:

- source – параметр, який вміщує код мови оригінального тексту;
- target – вміщує код параметру;
- format – вказує на формат даних до перекладу;
- q – текст, який повинен бути перекладеним.

Коди мов відповідають ISO-639-1 – стандартизований список ідентифікаторів мов, який існує для їх класифікації [10]. Складається здебільшого з двосимвольних кодів. Приклад класифікації наведено в таблиці 2.1.

Таблиця 2.1 – Ідентифікатори мов згідно з класифікацією ISO-639-1

Назва мови	**Оригінальна назва мови**	**Двосимвольний ідентифікатор**
Arabic	العربيـــــة	ar
English	English	en
French	Français	fr
Latin	Latine	la

Ukrainian	Українська	uk
Zhuang	Saɯ cueŋb	za

Формат відповіді є досить простим і містить тільки перекладений текст (рисунок 2.4).


```
{
  "data": {
    "translations": [
      {
        "translatedText": "La Gran Pirámide de Giza (también conocida como la
         Pirámide de Khufu o la Pirámide de Keops) es la más antigua y más
         grande de las tres pirámides en el complejo de la pirámide de Giza."
      }
    ]
  }
}
```


Рисунок 2.4 – Структура відповіді у форматі JSON

Speech-to-Text сервіс застосовує алгоритми нейронних мереж для перетворення людського голосу в текст. Працює схожим чином, як і попередній сервіс, що зображено на рисунку 2.5.


```
{
  "config": {
      "encoding":"FLAC",
      "sampleRateHertz": 16000,
      "languageCode": "en-US",
      "enableWordTimeOffsets": false
  },
  "audio": {
      "uri":"gs://cloud-samples-tests/speech/brooklyn.flac"
  }
}
```


Рисунок 2.5 – Структура запиту до Google Speech-To-Text API

Дана структура містить в собі два параметри – *config* та *audio*. Параметр audio зберігає аудіо-файл, який потрібно опрацювати. Config містить параметри, які відповідають характеристикам аудіо доріжки:

- кодування – тип кодування аудіо-файлу, наприклад *FLAC*, *mp3*, *mp4*;
- мовний код – код мови, якою ведеться мовлення.
- частота та інші;

Дані параметри варіюються залежно від типу аудіодоріжки. Використовуючи веб-серівс, користувач має змогу завантажити аудіодоріжку у різних форматах та кодуваннях, працюючи ж через інтерфейс чат-ботів, клієнтські програми можуть заздалегідь вирішувати проблему кодування файлів. Можуть застосовуватись також сторонні утиліти чи програмні бібліотеки для роботи з аудіо та визначення його параметрів.


```
{
  "results": [
    {
      "alternatives": [
        {
          "transcript": "how old is the Brooklyn Bridge",
          "confidence": 0.98267895
        }
      ]
    }
  ]
}
```


Рисунок 2.6 – Структура відповіді в JSON-форматі

На рисунку 2.6 розглянуто структуру відповіді від Google Speech-to-Text сервісу. Відповідь передбачає поле *result*, яке містить альтернативні варіанти розпізнаного тексту. Вони можуть з’являтись у великих файлах і містяться в полі *alternatives*. Агрегатор використовуватиме першу з доступних

альтернатив, адже саме вона, за рекомендацією Google, є найдостовірнішою. Альтернативи мають всього два поля: текстове *transcript*, яке містить розпізнаний текст, та числове *confidence*, яке відображає точність перекладу.

```
{
  "input":{
    "text":"Android is a mobile operating system developed by Google
  },
  "voice":{
    "languageCode":"en-gb",
    "name":"en-GB-Standard-A",
    "ssmlGender":"FEMALE"
  },
  "audioConfig":{
    "audioEncoding":"MP3"
  }
}
```

Рисунок 2.7 – Структура запиту до Google Text-to-Speech

Сервіс Text-to-Speech дозволяє перетворювати текст у аудіо-файл, що є додатковим функціоналом для агрегатора гетерогенних даних. Щоб конвертувати текст в аудіо потрібно надіслати HTTP POST запит до *https://texttospeech.googleapis.com/v1/text:synthesize* згідно з представленою структурою на рисунку 2.7.

Як показано, структура містить поле *input* з текстом, параметри голосу *voice*, які містять код мови, назву шаблону мовлення обраною мовою, та поле *ssmlGender*, яке містить вказівку мовою якого ґендерну (чоловічий, жіночий) потрібно озвучити текст. Поле *audioConfig* містить налаштування запрошеного аудіо, до прикладу, *audioEncoding* вказує на спосіб кодування аудіозапису.

У відповіді міститься запрошене аудіо, яке закодовано форматом *base64-encoded*. Його легко розкодувати за допомогою звичайних інструментів.

Google Cloud Vision дозволяє розробникам інтегрувати функції машинного бачення у власний сервіс, включаючи визначення людей, символів, місць, товарів та інших характеристик зображеного на фото. Для роботи з текстом сервіс-агрегатора використовується функція OCR (Optical Character Recognition), який дає доступ до:

- розпізнавання тексту у PDF/TIFF/GIF;
- розпізнавання тексту на зображеннях;
- розпізнавання рукописного тексту;

Існує дві анотації, які підтримуються OCR:

- TEXT_DETECTION визначає і видобуває текст з будь-якої фотографії, на якій може бути зображено, наприклад, плакати чи вуличні вивіски. Зворотня відповідь JSON-формату міститиме всі розпізнані слова.
- DOCUMENT_TEXT_DETECTION теж розпізнає текст, але для документів.

Запит до сервісу міститиме структуру, зображену на рисунку 2.8. Згідно цією структурою потрібно вказати тип запиту (TEXT_DETECTION або DOCUMENT_TEXT_DETECTION); Поле *image* може містити закодоване фото або URI фото розміщеного в інтернеті;

```
{
  "requests": [
    {
      "image": {
        "content": "base64-encoded-image"
      },
      "features": [
        {
          "type": "TEXT_DETECTION"
        }
      ]
    }
  ]
}
```

Рисунок 2.8 – Структура запиту до Cloud Vision

Синтаксичний аналіз.

Застосовувати API сервіси для роботи з текстом дуже зручно, але не всі сервіси мають змогу працювати за допомогою прикладних програмних інтерфейсів. Такі сервіси працюють з користувачем за допомогою HTML-документів, які відтворює браузер. Робота з такими документами є повільнішою, аніж з API, адже надісланий сервісом HTML-документ містить, окрім потрібних даних, елементи оформлення та інші додаткові елементи, які збільшують вагу повідомлення та ускладнюють отриманння потрібних даних. Для оптимізації процесу отримання даних супутньо до HTML-файлів не завантажуємо CSS та JavaScript-файли. Проблема добування потрібних даних є ключовою у задачі роботи з HTML-документами. Задля вирішення проблеми застосовується синтаксичний аналіз. Синтаксичним аналізом називається процес опрацювання вхідної послідовності символів з почерговим створення граматичної структури. Кінцева структура даних добре підходить для подальшої обробки. Для синтаксичного аналізу застосовуються спеціальні програми - синтаксичні аналізатори. Вони отримують вхідний текст і, обробляючи його, створює ієрархічну структуру, а також може перевіряти правильність синтаксису та проводити різного роду оптимізацію і форматування. Вихідні дані – структура даних або окремі запрошені елементи. Принцип роботи синтаксичного аналізатора подано на рисунку 2.9.

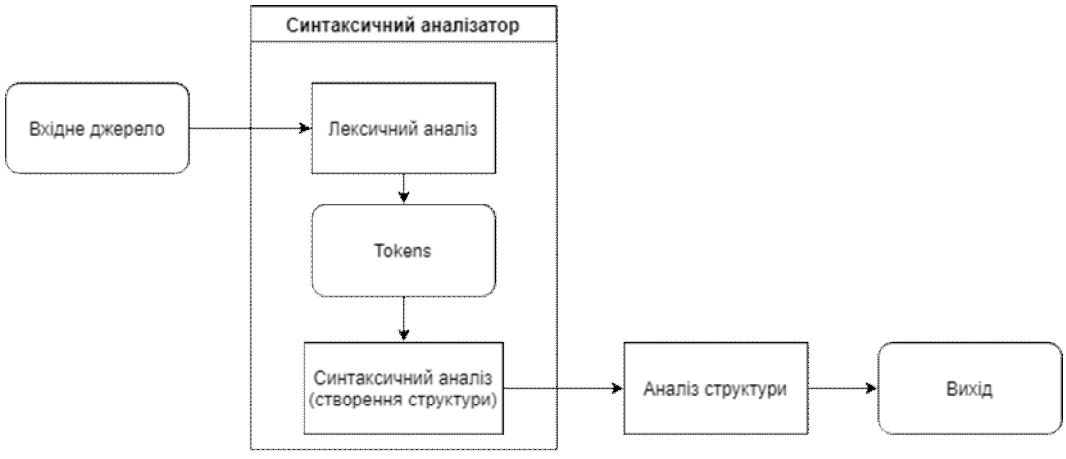


Рисунок 2.9 – Принцип роботи синтаксичного аналізатора

2.2. Проектування структури та алгоритмів функціонування системи

Підтримка та масштабування будь-якого програмного продукту напряму залежать від його початкового проектування. Ця проблема дуже вагома у розробці сервіс-агрегатора, адже сервіс передбачає розширення і роботу з абсолютно різними джерелами. Основні проблеми проектування сервіс-агрегатора:

1. Різні принципи роботи із джерелами. Деякі надають прикладний програмний інтерфейс для роботи з ними, а з іншими потрібно працювати, імітуючи клієнт (браузер).
2. Неоднорідність відповідей від серверів (JSON, XML, HTML).
3. Неоднорідність структур даних.

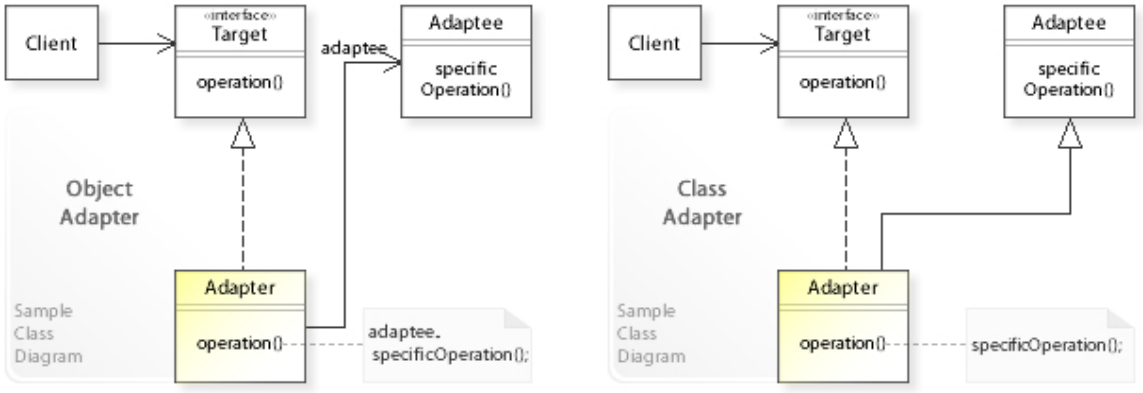


Рисунок 2.10 – UML-діаграма шаблону проектування «Адаптер»

Зважаючи на те, що реалізація роботи з кожним із сервісів може бути різною, потрібно застосувати шаблон проектування «Адаптер». Даний шаблон призначений для управління функціями об’єкта не зважаючи та окремо від його реалізації. Викладена проблема вирішується за допомогою створення оболонки з потрібним інтерфейсом. Такий підхід дозволяє заміщувати та додавати нові об’єкти незалежно від їх реалізації.

Щоб реалізувати даний шаблон проектування створюється інтерфейс на рисунку 2.11.



Рисунок 2.11 – Інтерфейс TranslationService

Інтерфейс TranslationService об’єднує потрібний функціонал та дозволяє працювати з кожним сервісом уніфіковано. Даний підхід дозволяє використовувати єдиний інтерфейс взаємодії з класом, ігноруючи його внутрішню реалізацію. Існує два способи визначення взаємодії з об’єктами сервісу за допомогою спільних функцій:

1. Визначення програмної структури «інтерфейс». Інтерфейси визначають спільну поведінку для групи об’єктів, які між собою пов’яхані нічим.
2. Абстрактні класи – це класи, які реалізують певну логіку, але делегують реалізацію певних функцій своїм нащадкам.

Проблема вибору використання даних двох способів полягає у різних властивостях інтерфейсів та абстрактних класів:

- Інтерфейс вказує тільки на спосіб взаємодії, в той час як абстрактний клас реалізувати логіку виконання визначених функцій.
- У деяких мов програмування дозволяється наслідування тільки від одного класу, в то час як допускається довільна кількість імплементованих інтерфейсів.
- Наслідування декількох абстрактних класів веде до складнощів проектування та розробки програмного забезпечення.

Зважаючи на приведені особливості, перевага надається інтерфейсу, адже не існує конкретної реалізації TranslationService. Таким чином створюється інтерфейс TranslationService, який імплементують створені сервіси, як це показано на рисунку 2.12.

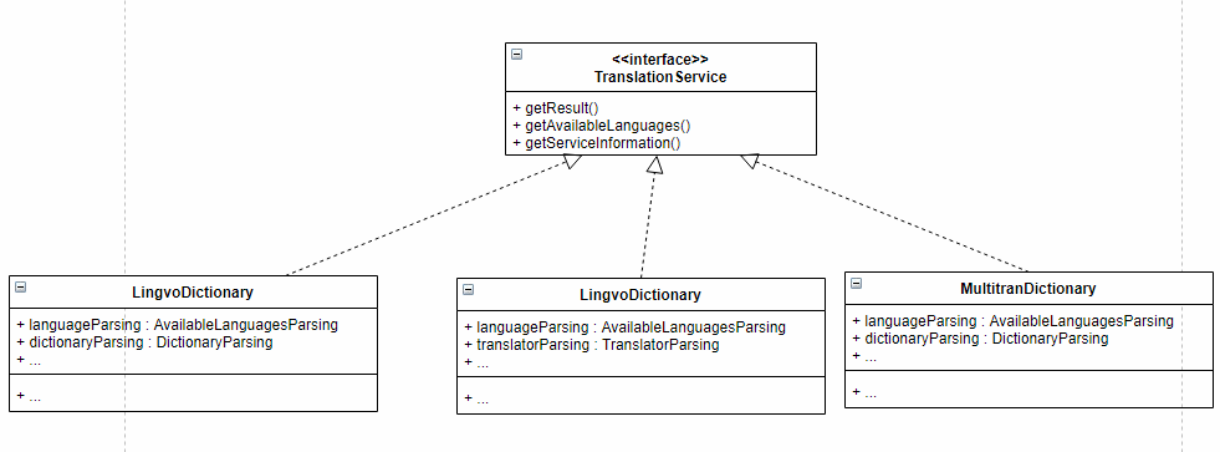


Рисунок 2.12 – Структура сервісів

Кожен програмний сервіс повинен реалізувати взаємодію з джерелом, до якого він надає доступ. При застосуванні технології синтаксичного аналізу може існувати декілька способів провести даний аналіз, аби він був стійким до змін початкової веб-сторінки. Для відділення логіки аналізу документів від сервісу проведено делегування даної функції спеціальному класу-парсеру.

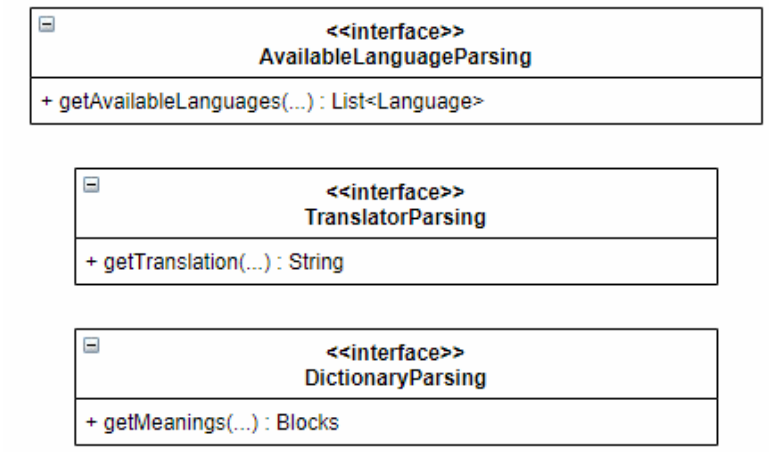


Рисунок 2.13 – Інтерфейси синтаксичного аналізу

На рисунку 2.13 зображено інтерфейс реалізації синтаксичного аналізу. Даний спосіб дозволяє створювати багаторівненвий аналіз, а також, в разі потреби, легко змінювати реалізацію на іншу.

Продемонстрований спосіб проектування реалізує принципи SOLID:

1. Принцип відкритості/закритості – розширення може здійснюватись через наслідування, сутності повинні бути закритими для змін.
2. Принцип єдиної відповідальності – кожен об’єкт має реалізовувати лише одну задачу.
3. Принцип підстановки Лісков – нащадки можуть замінювати батьківські елементи без додаткових змін у коді.
4. Принцип розділення інтерфесу – для забезпечення гнучкості при проектуванні, інтерфейси повинні бути поділеними на спеціалізовані
5. Принцип інверсії залежностей – система будується на абстракціях, а не на їх реалізаціях. Абстракції не повинні залежати від свої реалізацій.

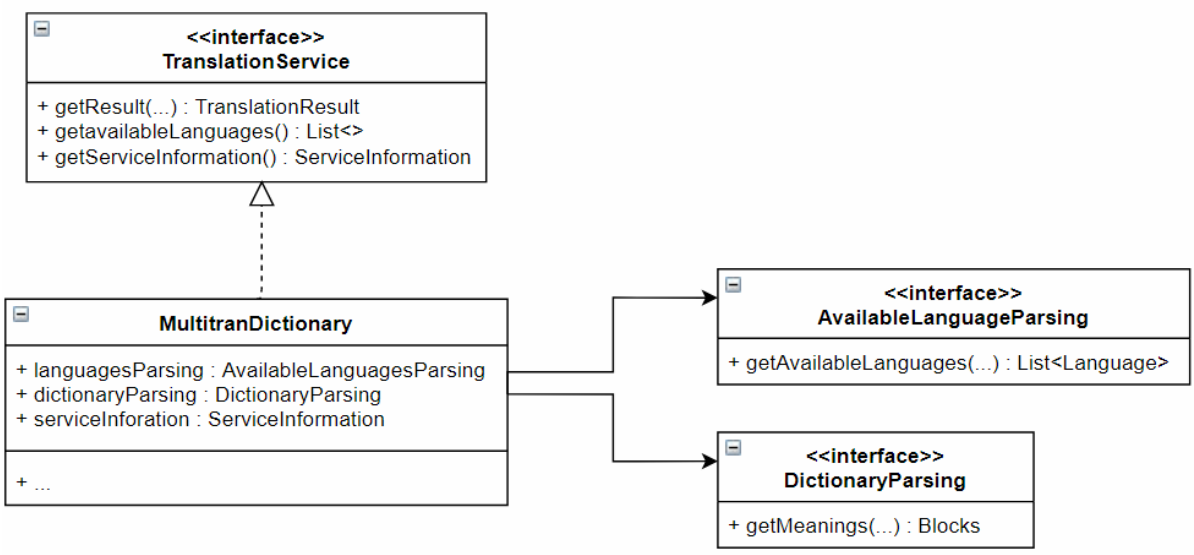


Рисунок 2.14 – Кінцева структура взаємодії компонентів

Дотримання даних принципів дозволяє розробляти масшбтабоване програмне забезпечення, сприятливе до розширення та підтримки. На рисунку 2.14 зображено один зі способів реалізації роботи з джерелом.

2.3. Висновки до розділу

У даному розділі розглянуто модель роботи системи та її структурні елементи. Також детально спроектовано реалізацію системи для агрегації різнорідних даних, за якої вона буде масштабованою та розширюваною.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ ФУНКЦІОНАЛЬНОСТІ СИСТЕМИ ОБРОБКИ ТА АГРЕГАЦІЇ РІЗНОРІДНИХ ДАНИХ

3.1. Вибір засобів та інструментів згідно з вимогами до розробки

Java - одна з найпопулярніших мов розробки. Вона спрямована на розробку мобільних додатків, захищених стільникових додатків, корпоративних рішень і веб-додатків, а також цю мову програмування можна застосувати для вирішення ряду інших проблем. Такий широкий спектр використання Java потребує масштабного підходу до розробки, який можна з легкістю забезпечити завдяки сучасним середовищам розробки.

Розглядаючи середовища розробки, повинні враховуватись основні потреби розробника, а також зручності, що здатні пришвидшити розробку ПЗ. Існують такі вільні середовища розробки для мови Java:

1. NetBeans
2. Eclipse
3. IntelliJ IDEA Community Edition
4. Android Studio
5. Enide Studio
6. jEdit
7. jGRASP
8. Jsource
9. JDeveloper
10. DrJava
11. BlueJ

IntelliJ IDEA Community Edition та Eclipse – сучасні середовища розробки, які є актуальними рішеннями та здатні конкурувати один з одним та стояти вище за інших. Їх перевагами над іншими є стійкість до збоїв, якість розробки, велика кількість прихильників, і, як наслідок, велика служба

підтримки, можливість розробки на декількох мовах програмування, а також кросплатформенність.

IntelliJ IDEA – розробляється компанією JetBrains та використовується для розробки додатків для Android, а також для Scala, Groovy, Java SE та plain-old-Java. Це легке середовище, що надається разом із корисним функціоналом, як JUnit для тестування, TestNG, засоби зневадження, інструменти, що перевіряють та доповнюють код. А також інструменти збирання, такі як Maven. Володіє такими додатковими перевагами, як нативна підтримка популярних фреймворків, JavaScript, JavaEE, Swagger, OpenAPI, інструменти для зручної роботи з SQL та визначення дублікатів коду.

Eclipse – розробляється вільною спільнотою розробників, написане переважно на Java. Екліпс дозволяє створювати кросплатформенні програми для використання у сегментах мобільної розробки, веб-розробки, розробки настільних програм та для комерції. Основні переваги включать Windows Builder, Maven, Mylyn, XML editor, Git client, CVS client, PyDev. Середовище також містить базовий простір розробки з підтримкою плагінів для модифікування середовища як вам заманеться. Завдяки плагінам існує можливість розробки на інших мовах програмування – C, C++, JavaScript, Perl, PHP, Prolog, Python, R, Ruby.

Звісно, багато в чому ці два середовища розробки є абсолютно схожими, адже реалізують схожий підхід до розробки ПЗ, але кожне з них має свої особливості, за яких одне середовище обирають на противагу іншому і навпаки. Вдаючись у деталі, є декілька ключових елементів, за якими ці середовища схожі чи не схожі.

1. Підтримка. Eclipse розробляється інженерами-ентузіастами, в той час, коли IntelliJ компанією-розробником. Це означає, що Eclipse є більш демократичною, хоча і з більшою кількістю багів і з повільнішою підтримкою, а IntelliJ у свою чергу має менше багів та швидшу зворотну підтримку.

2. Системні потреби. Eclipse є менш вибагливою до апаратного забезпечення користувача.

3. Підтримка фреймворків – лідером є IntelliJ, адже така підтримка реалізується досить швидко і якісно.

4. Ціна. Eclipse та IntelliJ IDEA Community Edition повністю безплатні, але у IntelliJ є платна версія з ширшими, аніж у безплатної версії, можливостями.

Зараз вибір між цими фреймворками зводиться до вибору маленьких зручностей – швидкість роботи, автоматизація процесів, зовнішній вигляд і можливість його модифікації, підтримка великої кількості інструментів і т. д. Звісно, повноцінно розробляти ПЗ можна за допомогою обох середовищ розробки.

Для реалізації сервіс-агрегатора використовується Intellij IDEA.

Також для підтримання темпу розробки та застосування актуальних бібліотек застосовуватиметься *Maven* – інструмент автоматизованого збору проєктів. Для опису проєкту застосовується pom-файл формату XML.

Щоб ефективно реалізувати сервіс-агрегатор використовується фреймворк Spring, який дає широкі можливості проектування програмного продукту, а також має добре документовані засоби вирішення проблем. Забезпечує рішення багатьох задач, які виникають у процесі створення інформаційної системи на Java. Spring надає шляхи розробки, які є кращою практикою в середовищі розробки. Одною з головних переваг Spring є Inversion of Control – інверсія управління, що забезпечує налаштування компонентів на управління життєвими циклами об’єктів. Контейнер відповідає за створення об’єтків, їх налаштування. Такі об’єкти називаються *beans* і конфігуруються за допомогою анотацій чи XML-файлів конфігурації. Створюються такі об’єкти за допомогою:

- пошуку залежностей, в якому об’єкт викликає в контейнера екземпляр об’єкту певного типу.

- впровадження залежностей шляхом передачі екземплярів об’єктів по імені іншим об’єктам за допомогою конструктора чи фабричного методу.

Також вагомим інструментом є Spring MVC – каркас який базується на моделі передачі даних через мережу та надає багато можливостей для розширення та налаштування:

- Розділення між шарами в MVC та запитах.
- Стратегія інтерфейсів – кожен виконує своє завдання.
- Можливість змінити інтерфейс іншою реалізацією.
- Пов’язаність інтерфейсів з Servlet API.
- Високий рівень абстракції для веб-застосунків.

Гнучке тестування, модульне та інтеграційне, забезпечується завдяки вбудованим засобам Spring. Даний фреймворк підтримує роботу з ресурсами за допомогою шару роботи з базами даних завдяки JDBC, таким чином підтримуються й популярні ORM.

Сервіс-агрегатор має бути реалізований з використанням парсингу. Для цієї мети існує програмна бібліотека Jsoup. Дана бібліотека реалізує системний аналіз HTML-файлів та надає гнучке управління даними, що знаходяться в документах. Дана бібліотека підключається до проекту за допомогою Maven, що демонструється рисунком 3.1.

```
<!-- Jsoup -->
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.13.1</version>
</dependency>
```

Рисунок 3.1 – Додавання програмної бібліотеки Jsoup

За допомогою URL веб-ресурсу бібліотека завантажує HTML сторінку, оптимізуючи її залежності. Доступний широкий функціонал модифікації запиту відносно «клієнта», який надсилає цей запис. Можна встановити користувацького клієнта як будь-який браузер, наприклад, *Chrome/ Safari*. Залежно від джерела запиту, можна отримати різний результат, тому варто вказати в полі *referrer* коректне джерело.

Дана бібліотека надає інструменти роботи з отриманим документом. Можна проводити вибірку елементів за тегами, атрибутами, класами та іншими параметрами. Дізнатись код цих елементів можна за допомогою браузерних інструментів швидкого пошуку коду вибраного елемента. Після цього налаштовуємо вибірку таким чином, аби Jsoup виділив потрібні дані.

3.2. Програмна розробка основних модулів системи агрегації даних

Для реалізації поставлених задач висунуто:

- Підтримка роботи з Google Cloud Translation API.
- Розпізнавання тексту засобами Google Cloud Vision API.
- Забезпечення роботи з онлайн-словником Multitran за допомогою синтаксичного аналізу.
- Розпізнавання голосу засобами Speech-to-Text.
- Перетворення тексту в голос функцією Text-to-Speech.

3.2.1 Контролер

Контролер сервісу є зовнішнім інтерфейсом системи. Після остаточного проектування прикладний програмний інтерфейс має наступну схему:

- /services – GET-запит повертає список сервісів;
- /services/{service_code} – GET-запит повертає інформацію про сервіс;
- /services/{service_code}?sl={source_lang}&tl={target_lang}&text={text} – GET-запит повертає результат перекладу;

- /services/{service_code}/languages – GET-запит повертає список мов даного сервісу;
- /vision – POST-запит повертає розпізнаний текст.

Контролер опрацьовує запит користувача та визначає режим доступу.

Особливості роботи з фрейморком Spring полягають у легкому конфігуруванні за допомогою анотацій – форми синтаксичних метаданих. В даному випадку для опису методу, що отримує запити від клієнта, використовується анотація *@GetMapping*, яка визначає схему (mapping) даних. Загалом, сервісні функція виглядатимуть так:

```
@RestController
@RequestMapping(produces="application/json")
public class MainController {
    TranslatorsManager manager = new TranslatorsManagerImpl();

    @RequestMapping("/services")
    public List<ServiceInformation> index() {
          ...
    }

    @RequestMapping(value = "/services/{serviceCode}", produces =
MediaType.APPLICATION_JSON_VALUE)
    public TranslationResult index(@PathVariable String
serviceCode,
                                   @RequestParam String sl,
                                   @RequestParam String tl,
                                   @RequestParam String
text) {

          ...
}

    @GetMapping("/services/{serviceCode}/languages")
```

```
    public List<Language> getServiceLanguages(@PathVariable
String serviceCode) {
        ...
    }
    @GetMapping("/services/{serviceCode}")
    public ServiceInformation getServiceInfo(@PathVariable String
serviceCode) {
        ...
    }
    @PostMapping("/vision")
    public String getTextFromDocument(@NotNull @RequestBody
VisionParametersDTO visionParametersDTO) {
          ...
}
```

З даного лістингу зрозуміло, що анотації класу вказують про використання стилю REST. Анотація *@RequestMapping(produces="application/json")* обмежує відповідь форматом даних JSON.

3.2.2 Реалізація внутрішньої структури

Обрана модифікація шаблону проектування «Адаптер» потребує загального інтерфейсу, завдяки якому можна уніфіковано взаємодіяти з сервісами джерел.

```
    public interface TranslationService {
        TranslationResult getResult(Language sourceLanguage,
Language targetLanguage, String text);
        List<Language> getAvailableLanguages();
        ServiceInformation getServiceInformation();
    }
```

Таким чином визначається режим взаємодії з сервісом:

- отримати результат виконання перекладу;
- отримати список доступних мов сервісу;
- отримати інформацію про сервіс;

Забезпечення даного підходу дозволяє наповнювати класи сервісів будь-якою внутрішньою реалізацією, що дає гнучкості у виконанні.

Так, наприклад, клас онлайн-словника Multitran виглядатиме наступним чином:

```
    public class MultitranDictionary implements
TranslationService {
        private AvailableLanguagesParsing languagesParsing;
        private DictionaryParsing dictionaryParsing;
        private ServiceInformation serviceInformation;
        private List<Language> availableLanguages;
        public MultitranDictionary(AvailableLanguagesParsing
languagesParsing, DictionaryParsing dictionaryParsing) {
            this.languagesParsing = languagesParsing;
            this.dictionaryParsing = dictionaryParsing;
            this.serviceInformation = new
ServiceInformationImpl("Multitran", "mtn");
        }

        @Override
        public TranslationResult getResult(Language
sourceLanguage, Language targetLanguage, String text) {
            TranslationResult result = new TranslationResult();

esult.setMeanings(dictionaryParsing.parseMeanings(sourceLanguage,
targetLanguage, text));
            return result;
        }

        @Override
```

```
        public List<Language> getAvailableLanguages() {
            if (availableLanguages == null ||
availableLanguages.isEmpty()) {
                availableLanguages =
languagesParsing.parseAvailableLanguages();
            }
            return availableLanguages;
        }
        public ServiceInformation getServiceInformation() {
            return serviceInformation;
        }
    }
```

Таким чином, внутрішня реалізація не обмежується інтерфейсом, що дає змогу реалізувати будь-яку потрібну логіку для кожного провайдера даних.

3.2.3 Синтаксичний аналіз HTML-документів.

Функціональне розширення коду та легкість підтримки забезпечуються додатковим рівнем абстракції для відділення реалізації від інтерфейсів. Для цього в реалізації сервісів виконання синтаксичного аналізу делегується спеціальним класам. Дані класи – парсери – виконують одну функцію: аналіз документів та виділення потрібних даних.

Реалізовано наступні інтерфейси парсерів:

- DictionaryParsing;
- TranslatorParsing;
- AvailableLanguagesParsing;

DictionaryParsing реалізує інтерфейс аналізу словників для виділення значення запитаного слова чи словосполучення з вхідного документу:

```
    public interface DictionaryParsing {
        Blocks parseMeanings(Language sourceLanguage, Language
targetLanguage, String text);}
```

До прикладу, за допомогою цього інтерфейсу реалізується аналіз веб-сторінки сервісу Multitran:

```
    public class MultitranDictionaryParser implements
DictionaryParsing {
        private String formattedRequestURL;
        public MultitranDictionaryParser() {
            this.formattedRequestURL =
"https://www.multitran.com/m.exe";
        }
        @Override
        public Blocks parseMeanings(Language sourceLanguage,
Language targetLanguage, String text) {
            final String endingURL =
this.formattedRequestURL + "?l1=" + sourceLanguage.getId() +
"&l2=" + targetLanguage.getId() + "&s=" + text;
            Blocks mainBlocks = new Blocks();
            try {
                Document doc =
Jsoup.connect(endingURL).get();
                Element table = doc.select("table").get(1);
                Elements rows = table.select("tr");
                Blocks subjectBlocks = new Blocks();
                for (int i = 1; i < rows.size(); i++) {
                    Element row = rows.get(i);
                    Elements columns = row.select("td");
                    Element firstColumn = columns.get(0);
                    if (firstColumn.hasClass("gray")) {
                        if (!subjectBlocks.isEmpty()) {
                            mainBlocks.add(subjectBlocks);
                            subjectBlocks = new Blocks();
                        }
  ubjectBlocks.setText(firstColumn.text());
```

```
                    } else if (firstColumn.hasClass("subj"))
    {
                        Element secondColumn =
    columns.get(1);
                        Blocks meaningsBlocks =
    this.parseTranslField(secondColumn);meaningsBlocks.setText(f
    irstColumn.text());
                        meaningsBlocks.add(new Blocks(
    secondColumn.text() )); //added Blocks
                        subjectBlocks.add(meaningsBlocks);
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            return mainBlocks;
        }
    private Blocks parseTranslField(Element transl) {
            ...
        }
}
```

Як бачимо, сервіс не залежить від реалізації парсингу документів. Даний аналіз відбувається за допомогою бібліотеки Jsoup. Структурні елементи сторінки зберігаються в масиві Elements, а кожен окремий у об’єкті Element – така реалізація дозволяє зручно працювати зі структурними одиницями.

TranslatorParsing – інтерфейс, реалізований засобами Java, аналізує сервіси-перекладачі великих змістовних текстів:

```
public interface TranslatorParsing {
    String parseTranslation(Language sourceLanguage,Language
targetLanguage, String text);
}
```

Інтерфейс AvailableLanguagesParsing визначає парсинг мов та їх кодів, з якими працює даний сервіс:

```
public interface AvailableLanguagesParsing {
    List<Language> parseAvailableLanguages();
}
```

Реалізація даного сервісу на прикладі Multitran:

```
    public class MultitranAvailableLanguagesParser
implements AvailableLanguagesParsing {
        private String formattedAvailableLanguagesPageURL;
        public MultitranAvailableLanguagesParser() {
            formattedAvailableLanguagesPageURL =
"https://www.multitran.com/";
        }
        @Override
        public List<Language> parseAvailableLanguages() {
            List<Language> languageList = new
LinkedList<>();
            try {
                Document doc =
Jsoup.connect(formattedAvailableLanguagesPageURL).get();
                Element table = doc.select("table").get(4);
                Elements rows = table.select("tr");
                for (Element row : rows) {
                    Elements columns = row.select("td");
                    for (Element column : columns) {
                        String langName = column.text();
                        if (column.hasText()) {
                            String source =
column.select("a").attr("href");
                            String languageCode =
findNums(source).get(1);
```

```
                                languageList.add(new
    Language(langName, languageCode));
                            }
                        }
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }

                return languageList;
            }
            private List<String> findNums(String source) {
                LinkedList<String> nums = new LinkedList<>();
                Pattern pattern =
     Pattern.compile("\\b[\\d]+\\b");

                Matcher matcher = pattern.matcher(source);
                while (matcher.find()) {
                    nums.add(matcher.group());
                }
                return nums;
            }
    }
```

Синтаксичний аналіз можна виконувати не тільки HTML-документів, тому даний клас є корисним та застосовуваним за різних випадків, коли потрібно делегувати опрацювання формату даних іншому об’єкту.

3.2.4 Робота з Google Cloud Platform.

Робота з хмарними сервісами працює за допомогою API цих сервісів. Даний метод є складним в реалізації, але часто провайдери хмарних технологій створюють програмні бібліотеки для зручної роботи з API сервісу. GCP також має свою реалізацію програмних бібліотек, яка й

використовуються в даній роботі. Для під'єднання до хмарних сервісів додається залежність за допомогою Maven, рисунок 3.2.

```
<!-- Cloud Translation -->
<dependency>
    <groupId>com.google.cloud</groupId>
    <artifactId>google-cloud-translate</artifactId>
</dependency>
```

Рисунок 3.2 – Додавання програмної бібліотеки Google Cloud Platform

За допомогою даної бібліотеки реалізується клас GCP-сервісу для перекладу:

```
    public class GoogleCloudTranslator implements
TranslationService {
        @Override
        public TranslationResult
getResult(com.narentie.translation.utils.Language sourceLanguage,
com.narentie.translation.utils.Language targetLanguage, String
text) {
            Translate translate =
TranslateOptions.getDefaultInstance().getService();
            Translation translation;
            if (sourceLanguage.getId().equals("auto")) {
                translation = translate.translate(
                        text
Translate.TranslateOption.targetLanguage(targetLanguage.getId()),
                        // Use "base" for standard edition, "nmt"
for the premium model.
                        Translate.TranslateOption.model("nmt"));
            } else {
                translation = translate.translate(
```

```
                        text,
Translate.TranslateOption.sourceLanguage(sourceLanguage.getId())
Translate.TranslateOption.targetLanguage(targetLanguage.getId()),
                        // Use "base" for standard edition, "nmt"
for the premium model.
                        Translate.TranslateOption.model("nmt"));
            }
            TranslationResult result = new
TranslationResult(translation.getTranslatedText());
            return result;
        }
        @Override
        public List<com.narentie.translation.utils.Language>
getAvailableLanguages() {
            Translate translate =
TranslateOptions.getDefaultInstance().getService();
            List<Language> languages =
translate.listSupportedLanguages();
            List<com.narentie.translation.utils.Language>
resultList = new ArrayList<>();
            resultList.add(new
com.narentie.translation.utils.Language("Auto", "auto"));
            for (Language language : languages) {
                resultList.add(new
com.narentie.translation.utils.Language(language.getName(),
language.getCode()));
            }
            return resultList;
        }
        @Override
        public ServiceInformation getServiceInformation() {
            return new ServiceInformationImpl("Google
Translation", "gctr");
        }
    }
```

Дана реалізація є дуже гнучкою, тобто вона може налаштовуватись та розширюватись, завдяки чому працювати, наприклад, окремим потоком із застосуванням паттерну «Одинак».

3.2.5 Сервіси розпізнавання.

Google Cloud Vision тексту дозволяє легко реалізувати розпізнавання тексту. Особливості роботи з Google Cloud API полягають у передачі однієї з двох типів анотацій TEXT_DETECTION та DOCUMENT_TEXT_ DETECTION. За допомогою існуючих бібліотек взаємодії з інтерфейсом хмари реалізовано функцію визначення тексту малюнку:

```
    public static String detectText(String documentURL) throws
Exception, IOException {
            List<AnnotateImageRequest> requests = new
ArrayList<>();
            ImageSource imgSource =
ImageSource.newBuilder().setImageUri(documentURL).build();
            Image img =
Image.newBuilder().setSource(imgSource).build();
            Feature feat =
Feature.newBuilder().setType(Feature.Type.TEXT_DETECTION).build()
;
            AnnotateImageRequest request =
AnnotateImageRequest.newBuilder().addFeatures(feat).setImage(img)
.build();
            requests.add(request);
            BatchAnnotateImagesResponse response =
client.batchAnnotateImages(requests);
            List<AnnotateImageResponse> responses =
response.getResponsesList();
                return
responses.get(0).getTextAnnotations(0).getDescription();
        }
```

Робота з розпізнаванням тексту завжди продукує більш ніж один варіант розпізнаного тексту. Кожен з таких варіантів позначений числовим коефіцієнтом (0;1), що означає прогнозовану точність перекладу.

3.2.6 Допоміжні класи.

Передача даних ведеться у форматі JSON та має містити уніфікований об’єкт передачі даних. Клас *TranslationResult* реалізує поставлену задачу та забезпечує повноту передання тексту.

```
    @XmlRootElement
    public class TranslationResult {
        private String translation;
        private Blocks meanings;
        public TranslationResult() {
            this("", new Blocks());
        }
        public TranslationResult(String translation) {
            this(translation, new Blocks());
        }
        public TranslationResult(String translation, Blocks
meanings) {
            this.translation = translation;
            this.meanings = meanings;
        }
        public String getTranslation() {
            return translation;
        }
        public void setTranslation(String translation) {
            this.translation = translation;
        }
        public Blocks getMeanings() {
            return meanings;
        }
```

```
        public void setMeanings(Blocks meanings) {
            this.meanings = meanings;
        }
        public boolean isEmpty() {
            return  translation == null || "".equals(translation)
|| meanings == null || meanings.isEmpty();
        }
        @Override
        public String toString() {
            return "TranslationResult{" +
                    "translation='" + translation + '\'' +
                    ", meanings=" + meanings +
                    '}';
        }
    }
```

Структура даного рішення містить два ключові елементи. Перший з них – *String*, який містить переклад тексту. Іншим є «блоків». Даний масив може містити безкінечно вкладений масив даних.

```
    public class Blocks {
        private String text;
        private List<Blocks> blocks;
        public Blocks() {
            this("", new LinkedList<>());
        }
        public Blocks(String text) {
            this(text, new LinkedList<>());
        }
        public Blocks(List<Blocks> blocks) {
            this("", blocks);
        }
        public Blocks(String text, List<Blocks> blocks) {
            this.text = text;
```

```
        this.blocks = blocks;
    }
    public Blocks add(Blocks element) {
        blocks.add(element);
        return this;
    }
    public Blocks remove(Blocks element) {
        blocks.remove(element);
        return this;
    }
    public boolean isEmpty() {
        return blocks.isEmpty();
    }
    public void setText(String text) {
        this.text = text;
    }
    public String getText() {
        return text;
    @Override
    public String toString() {
        return "Blocks{" +
                "text='" + text + '\'' +
                ", blocks=" + blocks +
                '}';
    }
```

Представлений клас містить текст блоку та підмножину наступних блоків. Приведена реалізація, а також наявність поля для перекладеного тексту утворюють умови для гнучкої роботи з API сервіс-агрегатора.

Робота з провайдерами гетерогенних даних передбачає перш за все роботу з кожним з провайдерів окремо із забезпеченням уніфікованих стандартів всередині. Так, для реалізації представлення мов перекладу різних сервісів застосовується клас *Language*. Кожен сервіс містить власний перелік

мов та застосовує різне їх кодування, тому реалізація рішення містить простий спосіб класифікувати мови у спільну структуру.

```
    public class Language {
        private String name;
        private String id;
        public Language(String name, String id) {
            this.name = name;
            this.id = id;
        }
        public String getName() {
            return name;
        }
        public String getId() {
            return id;
        }
        public static Language findLanguageByCode(String code,
List<Language> list) {
            for (Language l : list) {
                if (l.id.equals(code)) {
                    return l;
                }
            }
            return null;
        }
        @Override
        public String toString() {
            return "Language{" +
                    "name='" + name + '\'' +
                    ", shortCharacterCode='" + id + '\'' +
                    '}';
        }
    }
```

У класі міститься два поля *“name”* і *“id”*, завдяки чому можна охопити будь-яку класифікацію мов. Для набору мов застосовується вбудований засіб мови Java – списки *List<E>*.

Представлення сервісу відбувається за допомогою інтерфейсу *ServiceInformation*.

```
public interface ServiceInformation {
    String getName();
    String getCode();
}
```

Даний інтерфейс дозволяє дізнатись назву та ідентифікатор сервісу. Може бути розширеним та надавати додаткову інформацію про сервіс, як от логотип, опис сервісу та його розміщення в мережі.

3.3. Тестування функціоналу системи обробки та агрегації даних

Для того аби перевірити роботу програми, потрібно окреслити функціональність, яку надає система агрегації, тобто що саме очікується від програмного інтерфейсу системи, а саме:

- Отримати список всіх джерел, до яких надає доступ агрегатор, що містить назву та ідентифікатор джерела.;

- Отримати інформацію про конкретне джерело;

- Отримати список мов, які підтримує обране джерело;

- Отримати переклад тексту та переклад значень – почергова робота з перекладачами та словниками;

- Розпізнати текст на фото;

Проведення тестування забезпечує якісний контроль на кінцевим результатом та дозволяє відслідковувати зміни, адже єдиний прикладний програмний інтерфейс забезпечує роботу будь-яких клієнтських сервісів.

Роблячи за допомогою HTTP GET-запит до */services*, отримуємо відповідь у форматі JSON, яка вміщає основні заголовки та список доступних сервісів (рисунок 3.3).

```
GET http://localhost:8080/services

HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Sat, 13 Jun 2020 18:49:20 GMT

[
  {
    "name": "Google Translator",
    "code": "gtr"
  },
  {
    "name": "Globo",
    "code": "lbo"
  },
  {
    "name": "Multitran",
    "code": "mtrn"
  },
  {
    "name": "ABLingvo",
    "code": "abl"
  },
  {
    "name": "ReversoContext",
    "code": "rvc"
```

Рисунок 3.3 – Результат виконання запиту до */services*.

Надсилаємо запит для отримання списку мов потрібного джерела. Такий запит здійснюємо методом GET за адресою */services/mtn/languages*, що показано на рисунку 3.4. Відповідь міститиме перелік мов, представлених списком. У форматі JSON такий перелік буде оформлено як список та виділено символами “[“ і “]”. Кожен елемент списку містить об’єкт *Language*,

виділений символами “{“ і “}”, тобто кінцевий JSON-файл виглядатиме як список об’єктів із двома полями.

```
GET http://localhost:8080/services/mtn/languages

HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked

[
  {
    "name": "Англійська"
    "id": "1"
  },
  {
    "name": "Іспанська"
    "id": "5"
  },
  {
    "name":  "Латвійська"
    "id": "27"
  },
  {
    "name": "Румунська"
    "id": "13"
  },
```

Рисунок 3.4 – Результат виконання запиту */services/mtn/languages*

Для отримання інформації про сервіс виконується GET-за шляхом /service/mtn. Результати демонструються рисунком 3.5.

```
{
  "name": "Multitran",
  "code": "mtn"
}
```

Рисунок 3.5 – Результат виконання запиту до /service/mtn

Дані мови можуть бути сформованими у список та поданими клієнтським застосунком на вибір вибору. На рисунку 3.6 показаний такий список на прикладі реалізації клієнтського чат-бота Telegram.



Рисунок 3.6 – Організація списку мов клієнтським застосунком

Переклад тексту відбувається за допомогою параметризованого GET-запиту (рисунок 3.7). Це означає, що у запиті містяться параметри, які вказують додаткові умови.

Даний запит містить коди мов того сервісу, з яким працює. Результат містить логічні блоки перекладу, що відповідають розділам та підрозділам. Використання даного формату дозволяє злегкістю будувати складі вкладені відповіді. Мається наувазі повна відсутність будь-якого структурного обмеження. Справа в тому, що сервіси перекладу та словники по-різному класифікують результати:

- за частинами мови;
- за галуззю діяльності людини;
- за частотою вживаноті;

- за часом додавання;
- за новизною;



Рисунок 3.7 – Результат виконання параметризованого запиту перекладу.

Не завжди результати різних перекладів можна привести до єдиного типу структури, що веде до неможливості аналізу та обробки групи запитів. Використання класу Block означає, що структуру визначає онлайн-сервіс, а користувцький застосунок сам обирає спосіб аналізу та зображення користувачу. Такий підхід навіть дозволяє класифікувати результати різних джерел єдиним чином, не вказуючи при цьому назви структурних одиниць.

Також за допомогою класу Block можна утворювати складніші структури перекладу великих текстів шляхом використання блокових елементів – ідея полягає у розмежуванні сторінок, параграфів, речень та слів задля можливості гнучкої модифікації.

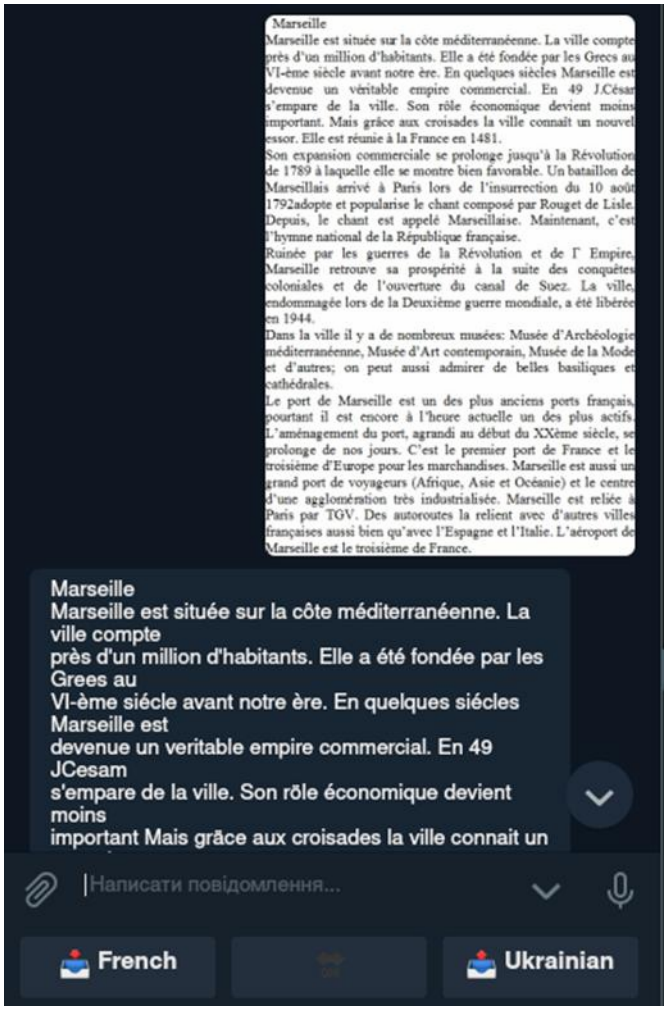


Рисунок 3.8 – Робота клієнтського застосунку з розпізнаванням тексту.

Робота з розпізнаванням тексту теж здійснюється за шляхом */vision*. Робота клієнтського сервісу з даною функцією демонструється на рисунку 3.8.

3.4 Висновки до розділу

В даному розділі розглянуто реалізацію сервіс-агрегатора провайдерів гетерогенних даних. Здійснено аналіз особливостей будови масштабованої системи та наведено приклади організації роботи з програмним інтерфейсом клієнтськими застосунками.

ВИСНОВКИ

В ході виконання роботи було досліджено предметну область розробки систем обробки та агрегації різнорідних даних на прикладі перекладання тексту та розроблено програмний продукт для забезпечення уніфікованого доступу до ресурсів, що є важливим елементом ефективної роботи з інформацією. Також розглянуто актуальність та проблематику обраної теми та зручність використання її рішень. Проаналізовані основні задачі агрегаторів та їх основні типи на сучасному ринку. Ознайомився з технологіями реалізацій веб-сервісів та шляхи їх побудови, а також основні переваги та недоліки даних програмних рішень.

Так, було описано та опрацьовано декілька варіантів комплексного підходу до розробки програмного забезпечення та обрано остаточну структуру системи для агрегації даних різної структури. Також забезпечено обов'язкову вимогу програмних агрегаторів – можливість розширення, адже вимоги та потреби користувачів ростуть і змінюються. З часом може знадобитись додавання нових сервісів для роботи з перекладом та текстом.

Проаналізовано важливість користувацьких інтерфейсів, адже ідея агрегаторів полягає у наданні швидкого та комфортного доступу до інформації. У сучасному світі люди взаємодіють зазвичай з десятками інтерфейсів: починаючи від інтерфейсів програмних компонентів операційних систем та закінчуючи інтерфейсами різних веб-сайтів. Зважаючи на такий широкий спектр доступних користувацьких інтерфейсів, обрано найбільш оптимальний варіант – використання прикладних програмних інтерфейсів.

ПЕРЕЛІК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. The NIST Definition of Cloud Computing – Режим доступу: https://csrc.nist.gov/publications/detail/sp/800-145/final.
2. RSS 2.0 SPECIFICATION – Режим доступу: https://validator.w3.org/feed/docs/rss2.html
3. YAML™ Specification – Режим доступу: http://yaml.org/spec/
4. Codes for the Representation of Names of Languages – Режим доступу: https://www.loc.gov/standards/iso639-2/php/English_list.php
5. Аналіз систем розпізнавання образів структури композитів : монографія / Добротвор І.Г., Стухляк П.Д., Микитишин А.Г., Митник М.М. – Тернопіль : Тернопільський національний технічний університет імені Івана Пулюя, 2018. – 224 с.
6. Бахрушин В.Є. Методи аналізу даних : навчальний посібник для студентів / В.Є. Бахрушин. – Запоріжжя : КПУ, 2011. – 268 с.
7. Офіційний сайт Google Cloud Documentation – Режим доступу: https://cloud.google.com/docs
8. Офіційний сайт IBM Cloud – Режим доступу: https://cloud.ibm.com/docs
9. Офіційний програмної бібліотеки Jsoup – Режим доступу: https://jsoup.org/
10. Мета пошуковий агрегатор: https://kr-labs.com.ua/blog/top-private-and-unique-search-engines
11. ТОП пошукових систем для OSINT | KR. Laboratories. – URL: https://kr-labs.com.ua/blog/top-private-and-unique-search-engines/
12. M.S. Aldenderfer, R.K. Blashfield. Cluster Analysis (Quantitative Applications in the Social Sciences). - SAGE Publications, 1984. - 88 p.
13. A.K. Jain R. Algorithms for Clustering Data / R.C. Dubes A.K. Jain. — New Jersey: Prentice Hall, 1988. — 334 p.

14. L. Kaufman. Finding Groups in Data: An Introduction to Cluster Analysis. Wiley Series in Probability and Statistics / L. Kaufman, P. J. Rousseeuw. — New York: John Wiley and Sons, 1990.

15. Albright S. C., Winston W., Zappe C. Data Analysis and Decision Making. Boston : Cengage Learning, 2016. 948 p.

16. Cao L., Philip S. Yu, Zhang C., Zhang H. Data Mining for Business Applications. Springer Science; Business Media, 2008. 402 p

17. Movie Reviews, TV Reviews, Game Reviews, and Music Reviews – Metacritic. - https://www.metacritic.com/

18. Tryon R. C. Cluster analysis / R. C. Tryon. – London : Ann Arbor Edwards Bros, 1939. – 139 p.

19. Hartigan J. A. Clustering Algorithms / J. A. Hartigan. – New York : Wiley, 1975. – 351 p.

20. Everitt B.S., Landau S., Leese M., Stahl D. Cluster Analysis. 5th Edition. John Wiley & Sons, 2011. 346 p.

21. Kassambara A. Practical Guide to Cluster Analysis in R. New York: STHDA, 2017. 187 p.

22. Ibm.com URL: https://www.ibm.com/docs/en/tnpm/1.4.2?topic=data-aggregation

ДОДАТКИ

Додаток А

Лістинг програми

```
import com.narentie.services.TranslatorsManager;
import com.narentie.services.TranslatorsManagerImpl;
import com.narentie.translation.utils.ServiceInformation;
import com.narentie.translation.TranslationService;
import com.narentie.translation.utils.Language;
import com.narentie.translation.utils.TranslationResult;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping(produces="application/json")
public class MainController {
    TranslatorsManager manager = new TranslatorsManagerImpl();

    @RequestMapping("/services")
    public List<ServiceInformation> index() {
        TranslatorsManager manager = new TranslatorsManagerImpl();

        List<ServiceInformation> serviceInformationList =
manager.getServicesInformation();

        return serviceInformationList;
    }

    @RequestMapping(value = "/{serviceCode}/", produces =
MediaType.APPLICATION_JSON_VALUE)
    public TranslationResult index(@PathVariable String serviceCode,
                                   @RequestParam String sl,
                                   @RequestParam String tl,
                                   @RequestParam String text) {

        TranslationService translationService =
manager.getServiceByCode(serviceCode);

        List<Language> availableLanguages =
translationService.getAvailableLanguages();

        Language sourceLanguage = Language.findLanguageByCode(sl,
availableLanguages);
        Language targetLanguage = Language.findLanguageByCode(tl,
availableLanguages);

        if (sourceLanguage != null || targetLanguage != null) {
            TranslationResult result =
translationService.getResult(sourceLanguage, targetLanguage, text);
            return result;
        } else {
            return null;
        }

    }

    @GetMapping("/{serviceCode}/languages")
```

```
    public List<Language> getServiceLanguages(@PathVariable String
serviceCode) {

        TranslationService translationService =
manager.getServiceByCode(serviceCode);

        List<Language> availableLanguages =
translationService.getAvailableLanguages();
        return availableLanguages;
    }

    @GetMapping("/{serviceCode}")
    public ServiceInformation getServiceInfo(@PathVariable String serviceCode)
{
        ServiceInformation serviceInformation =
manager.getServiceByCode(serviceCode).getServiceInformation();
        return serviceInformation;
    }
}
import com.narentie.translation.utils.ServiceInformation;
import com.narentie.translation.TranslationService;

import java.util.List;

public interface TranslatorsManager {

    TranslationService getServiceByCode(String shortCode);
    List<ServiceInformation> getServicesInformation();

}
import com.narentie.translation.utils.ServiceInformation;
import com.narentie.translation.TranslationService;
import com.narentie.translation.parsers.MultitranAvailableLanguagesParser;
import com.narentie.translation.parsers.MultitranDictionaryParser;
import com.narentie.translation.services.MultitranDictionaryService;

import java.util.ArrayList;
import java.util.List;

public class TranslatorsManagerImpl implements TranslatorsManager {

    private List<TranslationService> services;

    public TranslatorsManagerImpl() {
        services = new ArrayList<>();
        services.add(new MultitranDictionaryService(new
MultitranAvailableLanguagesParser(), new MultitranDictionaryParser()));
    }

    @Override
    public TranslationService getServiceByCode(String shortCode) {
        for (TranslationService ts : services) {
            if (ts.getServiceInformation().getCode().equals(shortCode))
                return ts;
        }
        return null;
    }

    @Override
    public List<ServiceInformation> getServicesInformation() {
        List<ServiceInformation> siList = new ArrayList<>();
        for (TranslationService ts : services) {
```

```
            siList.add(ts.getServiceInformation());
        }

        return siList;
    }
import com.narentie.translation.utils.Language;

import java.util.List;

public interface AvailableLanguagesParsing {
    List<Language> parseAvailableLanguages();
}

import com.narentie.translation.utils.Blocks;
import com.narentie.translation.utils.Language;

public interface DictionaryParsing {
    Blocks parseMeanings(Language sourceLanguage, Language targetLanguage,
String text);
}
import com.narentie.translation.utils.Language;
import com.narentie.translation.utils.ServiceInformation;
import com.narentie.translation.utils.TranslationResult;

import java.util.List;

public interface TranslationService {

    TranslationResult getResult(Language sourceLanguage, Language
targetLanguage, String text);
    List<Language> getAvailableLanguages();
    ServiceInformation getServiceInformation();

}

import com.narentie.translation.utils.Language;

public interface TranslatorParsing {
    String parseTranslation(Language sourceLanguage,Language targetLanguage,
String text);
}

import com.narentie.translation.AvailableLanguagesParsing;
import com.narentie.translation.utils.Language;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MultitranAvailableLanguagesParser implements
AvailableLanguagesParsing {

    private String formattedAvailableLanguagesPageURL;

    public MultitranAvailableLanguagesParser() {
        formattedAvailableLanguagesPageURL = "https://www.multitran.com/";
```

```
    }

    @Override
    public List<Language> parseAvailableLanguages() {

        List<Language> languageList = new LinkedList<>();

        try {
            Document doc =
Jsoup.connect(formattedAvailableLanguagesPageURL).get();
            Element table = doc.select("table").get(4);
            Elements rows = table.select("tr");

            for (Element row : rows) {
                Elements columns = row.select("td");
                for (Element column : columns) {
                    String langName = column.text();
                    if (column.hasText()) {
                        String source = column.select("a").attr("href");
                        String languageCode = findNums(source).get(1);

                        languageList.add(new Language(langName,
languageCode));
                    }
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }

        return languageList;
    }

    private List<String> findNums(String source) {
        LinkedList<String> nums = new LinkedList<>();

        Pattern pattern = Pattern.compile("\\b[\\d]+\\b");
        Matcher matcher = pattern.matcher(source);

        while (matcher.find()) {
            nums.add(matcher.group());
        }

        return nums;
    }

}
import com.narentie.translation.AvailableLanguagesParsing;
import com.narentie.translation.utils.Language;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

public class GoogleTranslatorAvailableLanguagesParser implements
AvailableLanguagesParsing {
```

```
    private String formattedAvailableLanguagesPageURL;

    public GoogleTranslatorAvailableLanguagesParser() {
        formattedAvailableLanguagesPageURL =
"https://cloud.google.com/translate/docs/languages";
    }

    @Override
    public List<Language> parseAvailableLanguages() {
        List<Language> languages = new LinkedList<>();

        try {
            Document doc =
Jsoup.connect(formattedAvailableLanguagesPageURL).get();

            Elements divs = doc.select("div");

            //System.out.println(divs);


            for (Element div : divs) {
                if (div.hasClass("devsite-article-body")) {
                    System.out.println(div);
                }
            }

            //Elements divs = doc.select("devsite-article-body clearfix");

            //System.out.println(divs);

        } catch (IOException e) {
            e.printStackTrace();
        }

        return languages;
    }

}

import com.narentie.translation.DictionaryParsing;
import com.narentie.translation.utils.Blocks;
import com.narentie.translation.utils.Language;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import java.io.IOException;

public class MultitranDictionaryParser implements DictionaryParsing {

    private String formattedRequestURL;

    public MultitranDictionaryParser() {
        this.formattedRequestURL = "https://www.multitran.com/m.exe";
    }

    @Override
    public Blocks parseMeanings(Language sourceLanguage, Language
targetLanguage, String text) {
        final String endingURL = this.formattedRequestURL + "?l1=" +
sourceLanguage.getId() + "&l2=" + targetLanguage.getId() + "&s=" + text;
```

```
        Blocks mainBlocks = new Blocks();
        try {
            Document doc = Jsoup.connect(endingURL).get();

            Element table = doc.select("table").get(1);
            Elements rows = table.select("tr");

            Blocks subjectBlocks = new Blocks();

            for (int i = 1; i < rows.size(); i++) {
                Element row = rows.get(i);
                Elements columns = row.select("td");

                Element firstColumn = columns.get(0);
                if (firstColumn.hasClass("gray")) {
                    if (!subjectBlocks.isEmpty()) {
                        mainBlocks.add(subjectBlocks);
                        subjectBlocks = new Blocks();
                    }
                    subjectBlocks.setText(firstColumn.text());
                } else if (firstColumn.hasClass("subj")) {
                    Element secondColumn = columns.get(1);

                    Blocks meaningsBlocks =
this.parseTranslField(secondColumn);
                    meaningsBlocks.setText(firstColumn.text());
                    meaningsBlocks.add(new Blocks( secondColumn.text() ));
//added Blocks
                    subjectBlocks.add(meaningsBlocks);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println(mainBlocks);
        return mainBlocks;
    }

    private Blocks parseTranslField(Element transl) {
        Blocks meanings = new Blocks();
        Blocks mean = new Blocks();

        for (Element e : transl.getAllElements()) {
            if (e.hasAttr("href")) {
                if (!"".equals(mean.getText())) {
                        meanings.add(mean);
                                }
                mean = new Blocks();
                mean.setText(e.text());
            } else {
                String innerText = e.text()
                        .replace("(", "")
                        .replace(")", "");

                if (!"".equals(innerText)) {
                    mean.add(new Blocks(innerText)); //added Bloks
                }
            }
        }
```

```
        meanings.add(mean);

        return meanings;
    }

}

public class MultitranDictionaryService implements TranslationService {

    private AvailableLanguagesParsing languagesParsing;
    private DictionaryParsing dictionaryParsing;
    private ServiceInformation serviceInformation;

    private List<Language> availableLanguages;

    public MultitranDictionaryService(AvailableLanguagesParsing
languagesParsing, DictionaryParsing dictionaryParsing) {
        this.languagesParsing = languagesParsing;
        this.dictionaryParsing = dictionaryParsing;
        this.serviceInformation = new ServiceInformationImpl("Multitran",
"mtn");
    }

    @Override
    public TranslationResult getResult(Language sourceLanguage, Language
targetLanguage, String text) {
        TranslationResult result = new TranslationResult();
        result.setMeanings(dictionaryParsing.parseMeanings(sourceLanguage,
targetLanguage, text));

        return result;
    }

    @Override
    public List<Language> getAvailableLanguages() {
        if (availableLanguages == null || availableLanguages.isEmpty()) {
            availableLanguages = languagesParsing.parseAvailableLanguages();
        }

        return availableLanguages;
    }

    public ServiceInformation getServiceInformation() {
        return serviceInformation;
    }
}

import java.util.LinkedList;
import java.util.List;

public class Blocks {
    private String text;
    private List<Blocks> blocks;

    public Blocks() {
        this("", new LinkedList<>());
    }

    public Blocks(String text) {
        this(text, new LinkedList<>());
    }
```

```
    public Blocks(List<Blocks> blocks) {
        this("", blocks);
    }

    public Blocks(String text, List<Blocks> blocks) {
        this.text = text;
        this.blocks = blocks;
    }

    public Blocks add(Blocks element) {
        blocks.add(element);

        return this;
    }

    public Blocks remove(Blocks element) {
        blocks.remove(element);

        return this;
    }

    public boolean isEmpty() {
        return blocks.isEmpty();
    }

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }

    public List<Blocks> getBlocks() {
        return blocks;
    }

    public void setBlocks(List<Blocks> blocks) {
        this.blocks = blocks;
    }

    @Override
    public String toString() {
        return "Blocks{" +
                "text='" + text + '\'' +
                ", blocks=" + blocks +
                '}';
    }
}

import java.util.List;

public class Language {

    private String name;
    private String id;

    public Language(String name, String id) {
        this.name = name;
        this.id = id;
    }
```

```
    public String getName() {
        return name;
    }

    public String getId() {
        return id;
    }

    public static Language findLanguageByCode(String code, List<Language>
list) {
        for (Language l : list) {
            if (l.id.equals(code)) {
                return l;
            }
        }

        return null;
    }

    @Override
    public String toString() {
        return "Language{" +
                "name='" + name + '\'' +
                ", shortCharacterCode='" + id + '\'' +
                '}';
    }
}
public interface ServiceInformation {
    String getName();
    String getCode();
}
public class ServiceInformationImpl implements ServiceInformation {
    private String name;
    private String code;

    public ServiceInformationImpl(String name, String code) {
        this.name = name;
        this.code = code;
    }

    public String getName() {
        return name;
    }

    public String getCode() {
        return code;
    }
}
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class TranslationResult {
    private String translation;
    private Blocks meanings;

    public TranslationResult() {
        translation = "";
        meanings = new Blocks();
    }

    public TranslationResult(String translation, Blocks meanings) {
        this.translation = translation;
```

```
        this.meanings = meanings;
    }

    public String getTranslation() {
        return translation;
    }

    public void setTranslation(String translation) {
        this.translation = translation;
    }

    public Blocks getMeanings() {
        return meanings;
    }

    public void setMeanings(Blocks meanings) {
        this.meanings = meanings;
    }

    public boolean isEmpty() {
        return  translation == null || "".equals(translation) || meanings ==
null || meanings.isEmpty();
    }

    @Override
    public String toString() {
        return "TranslationResult{" +
                "translation='" + translation + '\'' +
                ", meanings=" + meanings +
                '}';
    }
}

</project>
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: «Розробка програмного забезпечення системи обробки та агрегації різнорідних даних»

Обсяг пояснювальної записки: 76 аркушів

Кількість рисунків 35

Кількість додатків 1

Дата завершення роботи: 17 червня 2024 р.