**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**
**ІМЕНІ ТАРАСА ШЕВЧЕНКА**
Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

# Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки
на тему:

## Розробка системи для збору та інтелектуального аналізу публікацій змі

Виконав студент 4-го курсу
Довгополий Андрій _____ (підпис)

Науковий керівник:
Кандидат технічних наук
Кондратюк Сергій _____ (підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань. Студент _____ (підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теоретичної кібернетики

«____»_____2023 р., протокол No ____

Завідувач кафедри
Юрій КРАК

_____ (підпис)

Київ - 2023

# Вступ

В сучасному світі інформація є однією з найцінніших ресурсів. Завдяки швидкому розвитку технологій та інтернету, ми отримуємо величезну кількість інформації кожного дня. Однак, обробка та аналіз цієї інформації може стати складною задачею через її об'єм та розмаїття.

У рамках дипломної роботи було поставлено завдання розробити систему для збору та інтелектуального аналізу публікацій змі. Метою роботи є створення кастомізовуваного, ефективного та автоматизованого інструменту, який дозволить збирати публікації з різних джерел, проводити їх стандартизацію, аналізувати настрій тексту та генерувати короткий зміст. Така система забезпечить користувачам швидкий та зручний доступ до необхідної інформації, допомагаючи їм ефективно відслідковувати змі та отримувати корисні висновки.

**Розділ 1**

## Аналіз проблеми збору та аналізу публікацій змі

Проблема збору та аналізу публікацій змі виникає з-за ряду факторів, які можуть ускладнити процес та вплинути на якість інтелектуального аналізу. Деякі з найбільш поширених проблем включають:

1. Різноманітність джерел: публікації змі можуть бути розміщені на різних джерелах, таких як новинні сайти, блоги, соціальні медіа та інші. Кожен джерело може мати власний формат, структуру та правила публікації, що ускладнює процес їх збору та стандартизації.

2. Об'єм і швидкість змі: інформація про змі постійно змінюється і поширюється великими обсягами. Збір та обробка цієї великої кількості даних може стати складним завданням. Крім того, потрібно забезпечити швидкий доступ до актуальної інформації, оскільки змі можуть мати надзвичайно короткий термін актуальності.

3. Якість та достовірність даних: публікації змі можуть бути супроводжені неточностями, помилками або неправдивою інформацією. Це може вплинути на точність та достовірність інтелектуального аналізу, особливо якщо система покладається на автоматичну обробку даних.

4. Різнорідність форматів та мов: публікації змі можуть бути написані на різних мовах та мати різні формати (текст, зображення, відео тощо). Аналіз таких різнорідних даних вимагає спеціальних підходів та алгоритмів для виявлення та розуміння змісту.

5. Сентимент аналіз та семантичний аналіз: розуміння сентименту або настрою, вираженого в публікаціях змі, може бути складним

завданням. Людська мова часто використовує вирази, ідіоми, сарказм та інші суб'єктивні елементи, які можуть змінювати семантику та емоційний зміст тексту. Аналіз сентименту вимагає вдосконалених алгоритмів для виявлення та класифікації позитивного, негативного або нейтрального настрою.

6. Обробка неструктурованих даних: багато публікацій змі містять неструктуровану інформацію, таку як вільний текст, що ускладнює їх обробку та аналіз. Виявлення тематик, категоризація та витягування ключової інформації з таких даних вимагає використання методів обробки природної мови та машинного навчання.

7. Виклики збереження та обробки великого обсягу даних: зібрані публікації змі можуть виявитися великим обсягом даних, що вимагає потужних обчислювальних ресурсів та ефективних алгоритмів збереження, індексування та пошуку. Коректне зберігання та швидкий доступ до цих даних є важливими аспектами розробки системи.

Враховуючи вищезазначені проблеми, розробка та реалізація системи для збору та інтелектуального аналізу публікацій змі вимагає комплексного підходу та використання різних технологій і методів. Наявність бази даних для збереження інформації, пошукової системи для швидкого пошуку і т.п.

## Огляд існуючих систем і рішень

### Ground News

Ground News є онлайн-платформою, яка надає інструменти для аналізу новинового покриття з різних джерел ЗМІ. Її основна мета - допомогти користувачам отримати об'єктивну та розгорнуту картину новин, а не обмежуватися однобічними джерелами і поглядами.

Основні риси та функції Ground News:

1. Медіа-рейтинги: Ground News використовує методологію рейтингування ЗМІ, щоб визначити їх політичну спрямованість та біас. Вони використовують алгоритми та дані для категоризації новинних джерел і визначення їх позицій.
2. News Blindspot: Ця функція відображає новинні історії, які можуть бути недостатньо покриті або проігноровані певними ЗМІ. Ground News стежить за різними перспективами, щоб користувачі могли бачити ширший обсяг інформації.
3. Стеження за новинами: Ground News дозволяє користувачам стежити за конкретними новинними темами і отримувати сповіщення про пов'язані статті. Це дозволяє бути в курсі останніх подій та аналізувати різні точки зору.
4. Контекст та аналіз: Платформа надає контекстну інформацію до новинних статей, включаючи історію та подібні статті. Вона також забезпечує можливість порівняти покриття та біас різних ЗМІ.

Ground News дозволяє користувачам отримувати більш об'єктивну картину новин, досліджувати різні погляди та отримувати контекст до новинних статей. Вона може бути корисною для тих, хто бажає побачити ширший обсяг інформації та оцінити різні позиції щодо актуальних подій.

## Google News API

Google News API - це API, яке дозволяє розробникам отримувати доступ до новинних статей і сторінок з різних джерел ЗМІ. За допомогою Google News API, розробники можуть отримати доступ до останніх новин, розподіленних за категоріями та мовами, а також виконувати пошук новин за ключовими словами, визначеними датами та іншими параметрами.

Основні функції та можливості Google News API:

1. Отримання новинних статей: Розробники можуть отримувати останні новини з різних джерел ЗМІ, включаючи заголовки, текст статей, URL, зображення та інші метадані.
2. Категорії та мови: API надає можливість отримувати новини, розподілені за категоріями (наприклад, спорт, технології, наука) та мовами (наприклад, англійська, французька, німецька).
3. Пошук новин: Розробники можуть виконувати пошук новин за ключовими словами, визначеними датами, місцезнаходженням та іншими параметрами, щоб отримати релевантні статті.
4. Автоматичне розпізнавання мови: API може автоматично розпізнавати мову новинних статей, що дозволяє розробникам фільтрувати результати за мовою.

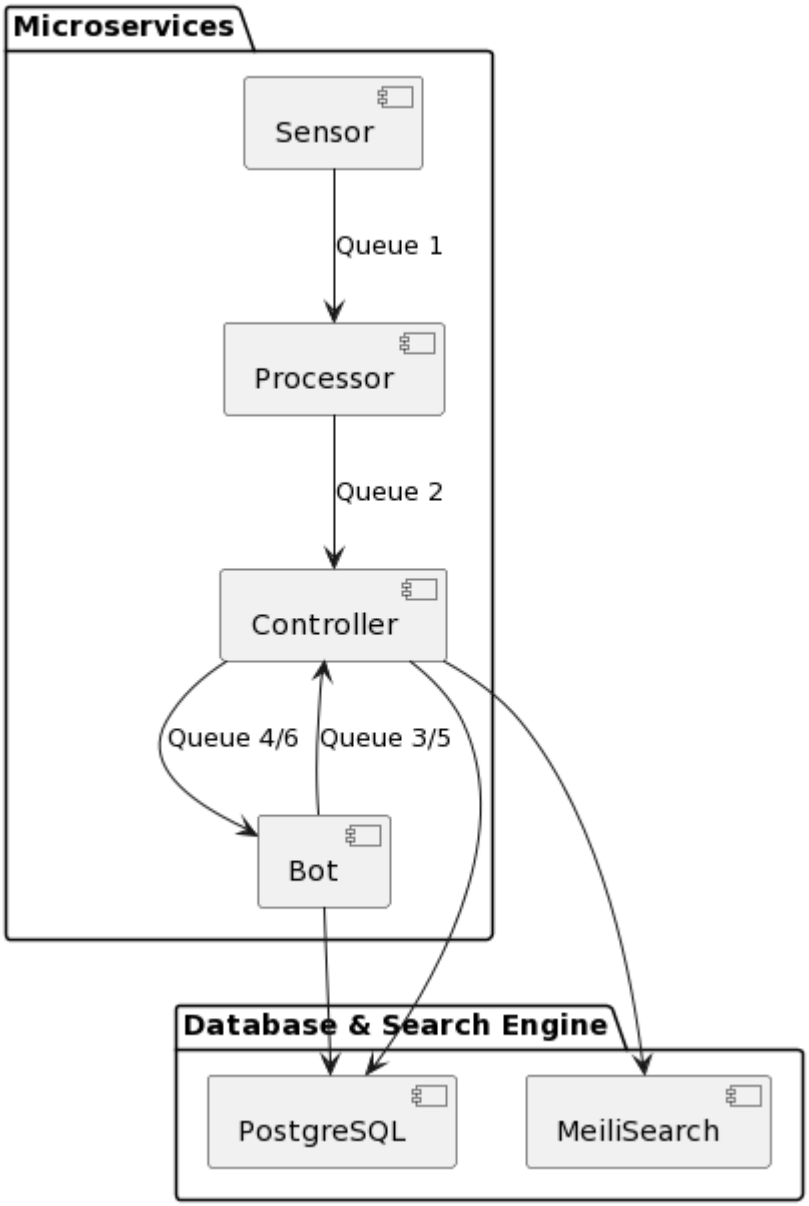
Google News API забезпечує розробникам доступ до великого обсягу новинних даних з різних джерел ЗМІ. Вона може бути використана для створення новинних агрегаторів, аналітичних платформ та інших додатків, які потребують доступу до свіжих новинних матеріалів.

## Розділ 2

### Функціональні вимоги до системи

1. Збір публікацій: Система повинна мати реалізований базовий спосіб здійснювати збір публікацій з різних джерел ЗМІ через RSS-канали, а також можливість кастомізації через написання плагінів для того, щоб так само збирати інформацію з соціальних медіа(наприклад, парсити їх сайти) та інших джерел інформації.
2. Стандартизація публікацій: Система повинна автоматично стандартизувати отримані публікації, щоб забезпечити їх єдність у форматі, структурі та метаданих.
3. Аналіз контенту: Система повинна проводити інтелектуальний аналіз публікацій, згідно з заданим пайплайном.
4. Індексація та пошук: Система повинна індексувати зібрані публікації та надавати можливість користувачам здійснювати ефективний пошук і фільтрацію за ключовими словами, датами, джерелами та іншими параметрами.
5. Моніторинг та сповіщення: Система повинна надавати можливість моніторити нові публікації та надсилати користувачам сповіщення або звіти про виявлені зміни, тренди чи інші важливі події.
6. Інтеграція зі сторонніми інструментами: Система повинна бути здатна інтегруватися з іншими інструментами аналізу даних, базами даних, пошуковими системами(той же google News API) або зовнішніми додатками для покращення аналітичних можливостей.

**Схема мікросервісної архітектури**



## Проектування системи

Для досягнення поставленої мети система розроблена на мікросервісній архітектурі, що дозволяє розбити функціональність на невеликі окремі компоненти, що працюють незалежно один від одного. Кожен мікросервіс відповідає за певну функцію, таку як збір публікацій, аналітичні обчислення, взаємодію з користувачами тощо. Використання мікросервісів сприяє гнучкості системи, легкості масштабування та підтримки різних технологій у кожному компоненті.

У процесі розробки системи було використано такі технології та інструменти, як база даних PostgreSQL, пошукова система Meilisearch та месседж брокер RabbitMQ. Ці компоненти були обрані з метою забезпечення ефективного зберігання даних, швидкого пошуку та передачі повідомлень між компонентами системи.

Структура системи включає такі мікросервіси:

1. Мікросервіс "Sensor": Цей компонент відповідає за збір публікацій з різних джерел. Він слухає RSS-канали та отримує нові публікації. Після отримання публікації, мікросервіс стандартизує її формат та надсилає до черги номер 1 для подальшої обробки.

2. Мікросервіс "Analytics": Цей компонент виконує аналітичні обчислення над отриманими публікаціями. Він отримує публікацію з черги номер 1, проводить сентимент аналіз для визначення настрою тексту та генерує короткий зміст. Результати обчислень записуються у чергу номер 2.

3. Мікросервіс "Bot": Цей компонент представляє собою телеграм-бота, який дозволяє користувачам вводити пошукові запити. Він отримує запити від користувачів та надсилає їх до черги номер 3 для подальшого опрацювання.

4. Мікросервіс "Controller": Цей компонент відповідає за керування системою. Він слухає чергу номер 2 і записує отримані результати аналітичних обчислень у пошукову систему Meilisearch. Крім того, він слухає чергу номер 3, виконує пошук по запитам користувачів та надсилає відповіді у чергу номер 4 для подальшої доставки до телеграм-бота.

Ця архітектура системи дозволяє забезпечити ефективне збирання та аналіз публікацій змі. В результаті роботи мікросервісів система здатна автоматично збирати публікації з різних джерел і стандартизувати їх

формат для подальшого аналізу. Мікросервіс "Analytics" виконує сентимент аналіз тексту та генерує краткий зміст, що допомагає користувачам швидко отримувати узагальнену інформацію про публікації. Мікросервіс "Bot" дозволяє користувачам шукати публікації за допомогою пошукових запитів і отримувати відповіді від системи.

Важливо відзначити переваги розробленої системи. Перш за все, мікросервісна архітектура дозволяє розділити функціональність на окремі компоненти, що спрощує розробку, супровід і масштабування системи. Кожен мікросервіс може бути розгорнутий та масштабований незалежно, що дозволяє ефективно використовувати ресурси. Крім того, пошукова система Meilisearch дозволяє швидко виконувати пошук та індексацію.

Проте, система також має свої обмеження. Одним з них є обмежена масштабованість архітектури. Якщо кількість сервісів та їх взаємозв'язки стають дуже складними, це може призвести до ускладнення розробки та супроводу системи. Крім того, інтеграція різних компонентів системи може бути складною і вимагати додаткових зусиль.

Незважаючи на обмеження, розроблена система має потенціал для подальшого розвитку. Можливості розширення включають:

1. Розширення джерел збору публікацій: систему можна розширити для збору публікацій з інших джерел, таких як соціальні мережі, блоги, новинні портали тощо. Це дозволить збирати більш широкий обсяг інформації та забезпечити більш повну картину змі.

2. Удосконалення аналітичних можливостей: систему можна розширити для використання більш складних аналітичних методів, таких як виявлення тематик, класифікація текстів, виявлення згадок про певні сутності тощо. Це дозволить отримувати більш детальну та контекстну інформацію з публікацій.

3. Покращення інтерфейсу користувача: можливо розширити функціонал телеграм-бота, додати можливості налаштування підписок, збереження публікацій, сповіщення про нові змі тощо. Також можна розробити веб-інтерфейс для зручного доступу до системи з будь-якого пристрою.

4. Використання машинного навчання: можливо впровадити алгоритми машинного навчання для автоматичного виявлення важливих публікацій, розуміння контексту, рекомендацій тощо. Це може покращити якість аналізу та забезпечити більш персоналізовані результати для користувачів.

5. Моніторинг та аналіз продуктивності: можливо впровадити механізми моніторингу та аналіз продуктивності системи, щоб забезпечити її ефективну роботу. Це може включати моніторинг ресурсів, швидкості виконання операцій, обсягу даних та інших метрик. Такий моніторинг допоможе виявляти потенційні проблеми та оптимізувати роботу системи для забезпечення високої продуктивності.

Остаточна розроблена система зі збору та інтелектуального аналізу публікацій змі має значний потенціал у сферах, де інформація є ключовим ресурсом. Вона може знайти застосування в медіа, фінансових установах, дослідницьких організаціях та багатьох інших галузях, де необхідно відстежувати змі та проводити аналіз текстової інформації. Подальший розвиток системи та впровадження запропонованих розширень можуть зробити її ще більш потужним і корисним інструментом для користувачів.

## Sensor

Компонент "Sensor" є одним з мікросервісів системи для збору та інтелектуального аналізу публікацій ЗМІ. Основна функція цього компоненту полягає в зборі публікацій з різних джерел ЗМІ та їх передачі для подальшої обробки.

Основні характеристики та функціонал компоненту "Sensor" включають:

1. Підписка на джерела: Компонент "Sensor" має базову можливість підписуватися на RSS-канали. Це дозволяє системі отримувати свіжі публікації з ЗМІ. За допомогою плагінів можна розширити функціонал до, наприклад, читання фейсбуку або різних жж
2. Збір публікацій: Після підписки на джерела, компонент "Sensor" здійснює періодичний збір публікацій з цих джерел. Він виконує запити до веб-сайтів, отримує RSS-стрічки, щоб отримати актуальну інформацію.
3. Обробка отриманих публікацій: Після збору публікацій, компонент "Sensor" проводить попередню обробку цих публікацій. Він видаляє непотрібні елементи, такі як непотрібні теги або додаткові відступи, і структурує дані для подальшої обробки.
4. Передача для подальшої обробки: Коли публікації вже оброблені, компонент "Sensor" передає їх до компоненту аналітики для проведення інтелектуального аналізу контенту.
5. Обробка помилок та повторна спроба: Компонент "Sensor" має механізми обробки помилок та повторної спроби. Якщо виникають проблеми з отриманням публікацій з якого-небудь джерела або з'єднанням, компонент "Sensor" може виконувати додаткові кроки для обробки таких ситуацій. Наприклад, він може записувати помилки до журналу, сповіщати адміністратора системи або автоматично спробувати повторити запит до джерела пізніше.

Компонент "Sensor" може бути горизонтально масштабованим, що означає, що можна запускати багато екземплярів цього компонента паралельно для обробки великої кількості джерел і публікацій. Це дозволяє системі бути ефективною та масштабованою для збору публікацій з різних джерел ЗМІ.

### Processor

Компонент "Processor" виконує обробку публікацій, що надходять, за допомогою пайплайну обробки. Один з важливих аспектів цього компонента - можливість розширення функціоналу користувачем шляхом додавання нових етапів обробки.

Користувач проекту може додати нові етапи обробки до пайплайну компонента "Processor". Це дозволяє використовувати як сторонні моделі машинного навчання, так і власний написаний звичайний код для розширення можливостей системи.

Створення нових етапів обробки може включати такі кроки:

1. Інтеграція сторонніх моделей: Користувач може додати новий етап обробки, використовуючи готові моделі машинного навчання або бібліотеки для виконання певних завдань, таких як визначення емоційного відтінку тексту, виявлення ключових слів або класифікація тематик. Це дозволяє використовувати потужні інструменти, розроблені іншими командами або дослідниками.
2. Написання власного коду: Користувач також може написати власний код для виконання специфічних завдань обробки. Це дає можливість налаштувати обробку публікацій згідно зі своїми потребами та враховувати специфіку домену або конкретні вимоги.

Користувач може визначити послідовність етапів обробки, які мають бути виконані для кожної публікації. Пайплайн обробки дозволяє передавати

результати одного етапу до наступного, створюючи послідовну обробку даних.

За допомогою компонента "Processor" та можливості розширення функціоналу користувач може створити гнучку систему обробки публікацій.

### Controller

Після того, як компонент "Processor" опрацює публікації і згенерує результати аналізу, компонент "Controller" виконує наступні дії:

1. Запис до бази даних: Компонент "Controller" відповідає за зберігання оброблених публікацій та результатів аналізу у базу даних. Він здійснює запис отриманих даних від "Processor" у відповідні таблиці бази даних, забезпечуючи їх доступність для подальшого пошуку та використання.
2. Обробка запитів на пошук: Компонент "Controller" відповідає за обробку запитів на пошук, які надійшли від користувача через інтерфейс або іншим способом. Він виконує пошук у базі даних, використовуючи підходящі алгоритми та запити, та повертає результати користувачеві.
3. Керування індексацією та пошуковою системою: Компонент "Controller" здійснює керування індексацією публікацій та взаємодію з пошуковою системою. Він забезпечує оновлення індексу після кожного запису до бази даних, щоб забезпечити актуальність результатів пошуку.
4. Керування налаштуваннями: Компонент "Controller" дозволяє користувачу змінювати налаштування системи, такі як параметри пошуку, фільтри, сортування тощо. Він забезпечує можливість

налаштування системи відповідно до вимог користувача і здійснює зберігання цих налаштувань для подальшого використання.

**Bot**

Компонент "Bot" взаємодіє з користувачем через Telegram, забезпечує можливість здійснення пошуку по збережених публікація.

Основні функції компонента "Bot" включають:

1. Прийом запитів користувача: Компонент "Bot" отримує запити від користувача через месенджер-платформу. Це може бути запит на пошук певної теми, отримання оновлень або запит на отримання підсумку публікації.
2. Обробка запитів: Компонент "Bot" аналізує та обробляє запити, що надійшли від користувача. Він використовує різні хендлери для команд, які надішле юзер.
3. Виконання пошуку: Після обробки запиту компонент "Bot" створює запит на пошук у базі даних, використовуючи компонент "Controller" для отримання результатів аналізу. Він може враховувати фільтри та критерії, задані користувачем, для забезпечення точних і релевантних результатів пошуку.
4. Генерація відповідей: Компонент "Bot" слухає чергу, до якої компонент "Controller" надсилає публікації(результати пошуку). Відповіді можуть бути структуровані та форматовані для зручного сприйняття користувачем.

Компонент "Bot" має дві основні команди: "Пошук" та "Підписка". Давайте розглянемо їх більш детально:

1. Команда "Пошук": Користувач може викликати команду "Пошук", вказавши певні критерії або ключові слова. "Bot" отримує цей запит і

передає його до компонента "Controller" для обробки. "Controller" виконує пошук у базі даних і повертає результати аналізу, які відповідають запиту. Після отримання результатів, "Bot" генерує відповідь зі списком публікацій, які задовольняють запиту, і відправляє цю відповідь користувачеві через месенджер-платформу.

2. Команда "Підписка": Користувач може використовувати команду "Підписка", щоб зберегти пошуковий запит. Коли користувач викликає цю команду і вказує пошуковий запит, "Bot" зберігає цей запит для подальшого використання. Компонент "Controller" постійно перевіряє нові публікації, які надходять до системи, і порівнює їх з збереженими пошуковими запитами користувачів. Якщо публікація відповідає одному з збережених запитів, "Controller" надсилає цю публікацію в окрему чергу, яку "Bot" слухає.

Коли "Bot" отримує нотифікацію про нову публікацію в черзі, він відправляє нотифікацію з цією публікацією користувачеві через месенджер-платформу. Це дозволяє користувачу отримувати сповіщення про нові публікації, які відповідають їхнім підпискам.

Таким чином, компонент "Bot" забезпечує можливість користувачам здійснювати пошук публікацій та підписуватися на певні критерії пошуку для отримання нотифікацій про нові публікації, що задовольняють їхнім інтересам.

Компонент "Bot" взаємодіє з компонентом "Controller", щоб забезпечити ефективну асинхронну обробку запитів користувача та надання релевантних відповідей.

## Meilisearch

Meilisearch - це потужний пошуковий двигун з високою швидкістю та точністю. Він пропонує простоту використання через добре задокументований API та доступні SDK для різних мов програмування. Meilisearch дозволяє розширювати та масштабувати пошукові додатки, контролювати релевантність результатів і працювати з великими обсягами даних. Цей пошуковий двигун відмінно підходить для проектів, де потрібна ефективна пошукова функціональність.

Між RedisSearch, ElasticSearch та **Meilisearch** вибір був здійснений навмання.

**Розділ 3**

## Реалізація системи

### Shared

Пакет "Shared" в проекті містить спільні компоненти, які використовуються різними модулями. Основні складові пакету "Shared" включають наступні:

1. DTO (Data Transfer Objects): Це класи, що представляють дані, які передаються між різними компонентами системи. DTO використовуються для передачі даних між мікросервісами, дозволяючи стандартизувати формат даних.

2. ORM (Object-Relational Mapping) моделі: Це класи, які відповідають таблицям у базі даних та забезпечують взаємодію з базою даних.

3. Модуль "Util": Цей модуль містить різні утилітарні функції, які використовуються в різних частинах проекту. Одна з них - функція import_all_from_package, яка дозволяє імпортувати всі файли з певного пакету. Це корисна функція, яка використовується в мікросервісах "Processor" та "Sensor", щоб користувач проекту міг легко розширити функціонал системи, додавши нові файли та класи до відповідних пакетів(див. 17с.).

Пакет "Shared" є спільним ресурсом, який дозволяє забезпечити повторне використання коду та стандартизацію даних та компонентів, що спрощує розробку та підтримку проекту.

```
def import_all_from_package(package_name):
    package_path = package_name.replace('.', '/')  # Convert package name to path format
    package_dir = os.path.dirname(__import__(package_name).__file__)  # Get package directory

    # Iterate over the files in the package directory
    for file_name in os.listdir(package_dir):
        if file_name.endswith('.py') and file_name != '__init__.py':  # Exclude __init__.py and non-Python files
            module_name = file_name[:-3]  # Remove the .py extension
            module_path = f'{package_path}.{module_name}'  # Construct the module path

            module = importlib.import_module(module_path)  # Import the module
            globals()[module_name] = module  # Add the module to the global namespace
```

**принцип реалізації DI наступний:**

marker_decorator.py

```
class Marker:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        self.marked_classes = []

    @staticmethod
    def mark_class(cls):
        Marker._instance.marked_classes.append(cls)
        return cls
```

```
Marker()
```

test_mod/classes.py :

```
from test.dec import Marker


@Marker.mark_class
class F1:
    pass

@Marker.mark_class
class F2:
    pass
```

main.py:

```
if __name__ == '__main__':
    package_name = "test_mod"
    import_all_from_package(package_name)

    print(Marker._instance.marked_classes)
```

Сервіс для роботи з rabbitmq також лежить в shared, оскільки використовується у всіх модулях. Він реалізованих з допомогою асинхронної бібліотеки aio_pika

RabbitMqService.py:

```
class RabbitMqService:
 def __init__(self):
 self.config = get_rabbit_config_from_env()

 async def send_data_to_queue(self, queue_name: str, message: dict):
        connection = await aio_pika.connect_robust(
                host=self.config.host,
                port=self.config.port,
                login=self.config.username,
                password=self.config.password
        )
        channel = await connection.channel()
        exchange = channel.default_exchange
        await exchange.publish(
                aio_pika.Message(body=json.dumps(message).encode()),
                        routing_key=queue_name
                )
        await connection.close()
        print(f"Message sent to queue: {queue_name}")

 async def consume_queue(self, queue, callback):
        async with aio_pika.connect_robust(
                host=self.config.host,
                port=self.config.port,
                login=self.config.username,
                password=self.config.password,
        ) as connection:
                channel = await connection.channel()
                queue = await channel.declare_queue(queue)

        async with queue.iterator() as queue_iter:
```

```
            async for message in queue_iter:
                async with message.process():
                    asyncio.create_task(callback(message))
```

### Проектування і реалізація мікросервісу "Sensor"

Для реалізації мікросервісу "Sensor" використовується Python та деякі бібліотеки. Нижче наведено опис використовуваних бібліотек та їх функціоналу:

1. Бібліотека feedparser: Ця бібліотека дозволяє отримати потік посилань на публікації з використанням RSS-каналів. Вона забезпечує зручний спосіб отримання та обробки даних з різних джерел новин або блогів.
2. Бібліотека selenium: Ця бібліотека використовується для автоматизації веб-браузера. У даному контексті вона буде використовуватись для відкриття посилань на публікації та отримання HTML-коду веб-сторінки, де розміщена публікація.
3. Бібліотека newspaper: Ця бібліотека дозволяє витягти текст публікації з HTML-коду веб-сторінки. Вона надає зручні методи для парсингу вмісту публікацій з різних джерел, забезпечуючи витягування заголовків, тексту, дат публікацій та іншої корисної інформації.

"Sensor" реалізовано з використанням підходу, що дозволяє користувачам розширювати його функціонал шляхом створення класів-нащадків.

Описаний підхід передбачає такі кроки:

1. Абстрактний клас AbstractListener містить метод article_iterator, який представляє ітератор для отримання публікацій. Цей клас визначає основний інтерфейс, який мають використовувати всі класи-нащадки.
2. Для управління класами-нащадками та їх реєстрації використовується сінглтон-клас SensorDIContainer (від DI-Dependency Injection). Цей клас має метод add_sensor, який приймає клас-нащадок AbstractListener та додає його до списку

зареєстрованих сенсорів. Цей список зберігається у DIContainer і використовується для створення інстансів сенсорів за необхідності.

3. Щоб розширити функціонал мікросервісу "Sensor", користувач повинен створити свій клас-нащадок, який успадковує клас AbstractListener. Клас-нащадок повинен бути помічений декоратором add_sensor, щоб його можна було зареєструвати у DIContainer. Клас-нащадок потрібно розмістити в пакеті sensors.

Після цього їх класи будуть автоматично зареєстровані у DIContainer і доступні для використання у системі.

Цей підхід дозволяє забезпечити гнучкість та розширюваність мікросервісу "Sensor", дозволяючи користувачам легко додавати нові сенсори та налаштовувати їх функціонал згідно зі своїми потребами.

При старті мікросервісу "Sensor" виконується наступна послідовність дій:

1. Викликається функція import_all_from_package, яка імпортує всі модулі з пакету sensors. Це дозволяє автоматично завантажити всі наявні класи-сенсори.

```
if __name__ == '__main__':
    import_all_from_package('listeners')
```

2. З DIContainer отримується список класів-нащадків AbstractListener, які були зареєстровані раніше. Для кожного класу створюється об'єкт (інстанс) за допомогою конструктора.
3. Ініціалізується інстанс класу SensorService, якому передається список створених об'єктів класів-сенсорів. Цей сервіс використовується для керування процесом збору публікацій.

```
if __name__ == '__main__':
    import_all_from_package('listeners')
    sensor_service = SensorService(SensorDIContainer.get_cls(),
```

```
RabbitMqService())
    asyncio.run(sensor_service.run())
```

4. SensorService ітерується по усіх класах-сенсорах і викликає метод run для кожного з них. Метод run у кожному з них пише в спільну (in-memory)чергу потік об'єктів типу ArticleDTO.
5. Отримані об'єкти ArticleDTO публікуються в черзі RabbitMQ

Таким чином, мікросервіс "Sensor" забезпечує збір публікацій за допомогою різних класів-сенсорів, реєструється і керується через DIContainer, а результати збору публікацій передаються для подальшої обробки в системі за допомогою RabbitMQ.

Приклад коду парсеру представлено нижче:

```
class Parser:
    def __init__(self):
        opts = webdriver.FirefoxOptions()
        opts.add_argument('-headless')
        self.driver = webdriver.Firefox(options=opts)

    @staticmethod
    async def parse_rss_feed(link: str, etag: str) -> FeedParserDict:
        return await feedparser.parse(link, etag=etag)

    @staticmethod
    def parse_date(date_str) -> int:
        date_obj = datetime.strptime(date_str, "%a, %d %b %Y %H:%M:%S %z")
        unix_timestamp = int(time.mktime(date_obj.timetuple()))
        return unix_timestamp

    def get_article_text(self, link: str) -> str:
        self.driver.get(link)
        article = newspaper.Article(link)
```

```
    article.download(self.driver.page_source)
    article.parse()
    return article.text

def entry_to_article_dto(self, entry) -> ArticleDto:
    link = entry['link']
    text = self.get_article_text(link)
    published = self.parse_date(entry['published'])
    return ArticleDto(link, text, published)
```

```python
@SensorDIContainer.add_sensor
class RssListener(AbstractListener):
    def __init__(self, queue):
        super(RssListener, self).__init__(queue)
        self.config = dotenv_values()

        self.links = init_link_storage(self.config)
        self.parser = Parser()

    async def run(self):
        async for link in self.links.link_iterator():
            asyncio.create_task(self.process_link(link))

    def main(self):
        asyncio.run(self.run())  # todo

    async def process_link(self, link):
        rss_parse_result = await self.parser.parse_rss_feed(link,
self.links.get_etag(link))
        for entry in rss_parse_result['entries']:
            try:
                article_dto = self.parser.entry_to_article_dto(entry)
                self.push_article(article_dto)
            except:
                log('error')
        self.links.set_etag(link, rss_parse_result.etag)
```

### Проектування і реалізація мікросервісу "Processor"

В мікросервісі "Processor" дії аналогічні до "Sensor", але з використанням класів-процесорів:

1. Існує абстрактний клас Handler, який має метод process. Цей метод приймає словник, сформований з обєкту типу ArticleDTO і виконує обробку цієї публікації. Кожен клас-нащадок Handler також має додатковий метод get_key, який повертає ключ, за яким записується в словник результатів сам результат обробки.
2. Пакет, де розташовані класи-процесори, називається handlers.
3. Клас ProcessorDIContainer має декоратор add_processor, який додає клас, помічений ним і унаслідуваний від класу Processor в список зареєстрованих процесорів.
4. Ініціалізується інстанс класу ProcessorService, якому передається список створених об'єктів класів-процесорів. Цей сервіс використовується для керування процесом обробки публікацій.
5. ProcessorService ітерується по отриманих з черги ArticleDTO і, для кожного з них, перетворює його на словник. Потім сервіс ітерується по списку переданих йому при ініціалізації об'єктів процесорів і додає результат виклику методу process по ключу, отриманому з методу get_key процесора, в словник.

Класи-наслідники класу Processor в мікросервісі "Processor" мають гнучкість в обробці тексту, що дозволяє їм використовувати власний код або завантажені моделі з бібліотеки Hugging Face. Це дає користувачам можливість розширювати функціонал сервісу шляхом створення нових класів-наслідників і налаштування їх для власних потреб.

Якщо користувач бажає використовувати власний код для обробки тексту, він може написати власний клас-наслідник Processor, реалізувати методи process і get_key для своїх потреб і додати цей клас у DIContainer.

У разі використання завантажених моделей з Hugging Face, користувач може написати клас-наслідник, який використовує попередньо натреновану модель для обробки тексту. Наприклад, використовуючи бібліотеку transformers в Python, користувач може імпортувати модель і використовувати її у своєму класі-наслідникуProcessor. Метод process цього класу буде виконувати необхідні дії для обробки тексту з використанням завантаженої моделі.

Ця гнучкість дозволяє користувачам використовувати власний код або використовувати готові моделі для обробки тексту в мікросервісі "Processor", забезпечуючи широкі можливості для розширення функціоналу і задоволення індивідуальних потреб користувачів.

Сам клас Pipeline має метод add_handler через який на самому початку додаються ініціалізовані інстанси класів наслідників Handler.

```python
class Handler:
    def __init__(self, key: str):
        self.__key = key

    def get_key(self):
        return self.__key

    def process(self, inp: dict):
        pass


class Pipeline:
    def __int__(self):
        self.handlers = []

    def add_handler(self, handler: Handler):
        self.handlers.append(handler)

    def process(self, input: dict):
        for handler in self.handlers:
            input[handler.get_key()] = handler.process(input)
```

**Приклади хендлерів, для пайплайну:**

**1. простий підрахунок кількості слів**

```
@ProcessorDIContainer.add_processor
class WordCounter(Handler):
    def __init__(self):
        super().__init__("words cnt")

    def process(self, input: str):
        return self.summarizer(input, min_length=3, max_length=128)
```

**2. використання моделі з huggingface для генерації summary**

```
@ProcessorDIContainer.add_processor
class Summarizer(Handler):
    def __init__(self):
        super().__init__("summary")
        tokenizer = AutoTokenizer.from_pretrained("google/mt5-large")
        model =
AutoModelForSeq2SeqLM.from_pretrained("SGaleshchuk/mT5-sum-news-ua"
)
        self.summarizer = pipeline("summarization", model=model,
tokenizer=tokenizer, framework="pt")

    def process(self, input: str):
        return self.summarizer(input, min_length=3, max_length=128)
```

**Проектування і реалізація мікросервісу "Controller"**:

Мікросервіс "Controller" відіграє важливу роль у системі, керуючи роботою і координацією інших мікросервісів. Його основною відповідальністю є обробка запитів користувача та управління потоком даних.

Задля спрощення роботи з чергами, пов’язаними з Controller задамо клас-обгортку для дефолтного RabbitMqService:

```
class ControllerRabbitMqService(RabbitMqService):
    def __init__(self):
        super().__init__()
        config = get_config()
        self.search_requests_queue =
config.get('RABBIT_SEARCH_REQUESTS_QUEUE_NAME')
        self.search_result_queue =
config.get('RABBIT_SEARCH_RESULT_QUEUE_NAME')
        self.notifications_queue =
config.get('RABBIT_NOTIFICATION_QUEUE_NAME')
        self.article_queue = config.get('RABBIT_ARTICLE_QUEUE_NAME')

    async def publish_notification(self, notification):
        await self.send_data_to_queue(self.notifications_queue,
notification.to_dict())

    async def publish_search_result(self, search_result_dto):
        await self.send_data_to_queue(self.search_result_queue,
search_result_dto.to_dict())

    async def consume_article_updates_task(self, article_callback):
        cb = lambda message:
article_callback(ArticleDto.from_json(message.body))
        await self.consume_queue(self.article_queue, cb)

    def consume_search_updates_task(self, search_request_callback):
```

```
        cb = lambda message:
search_request_callback(SearchRequestDto.from_json(message.body))
        await self.consume_queue(self.article_queue, cb)
```

Для роботи з пошуковою системою напишемо сервіс MeilisearchService:

```
class MeilisearchService:
    def __init__(self, engine_url: str, tmp_index_name: str, main_index_name:
str):
        self.engine_url = engine_url
        self.lock = asyncio.Lock()
        self.client = Client(main_index_name)
        self.tmp_index = self.client.index(tmp_index_name)
        self.main_index = self.client.index(main_index_name)

    async def save_article_to_tmp_index(self, article):
        await self.lock.acquire()
        self.tmp_index.add_documents([article])
        self.lock.release()

    async def remove_article_from_tmp_index(self, doc_id):
        await self.lock.acquire()
        self.tmp_index.delete_document(doc_id)
        self.lock.release()

    async def search_for_updates(self, request_phrase: str):
        await self.lock.acquire()
        result = self.tmp_index.search(request_phrase)
        self.lock.release()
        return result['hits']

    async def search(self, request_phrase: str):
        await self.lock.acquire()
        result = self.main_index.search(request_phrase)
        self.lock.release()
        return result

    async def save_article(self, article):
        await self.lock.acquire()
        self.main_index.add_documents([article])
```

```
    self.lock.release()
```

І, власне, сам клас з логікою:

```
class Worker:
    def __init__(self):
        config = get_config()
        self.search_engine_service =
MeilisearchService(config.get('ENGINE_URL'),
config.get('TMP_INDEX_NAME'),
                                           config.get('MAIN_INDEX_NAME'))
        self.queue_service = ControllerRabbitMqService()
        self.queries_provider = QueryProvider()

    async def article_callback(self, article: ArticleDto):
        queries = self.queries_provider.get_queries()
        await
self.search_engine_service.save_article_to_tmp_index(article.to_dict())

        for key in queries.keys():
            updates = await self.search_engine_service.search_for_updates(key)
            if updates is not None:
                for update in updates:
                    notification = NotificationDto(queries[key], update['link'], key)
                    await
self.search_engine_service.remove_article_from_tmp_index(update['id'])
                    await self.queue_service.publish_notification(notification)
        await self.search_engine_service.save_article(article.to_dict())

    async def search_request_callback(self, search_request: SearchRequestDto):
        updates = await self.search_engine_service.search(search_request.query)
        links = [update['link'] for update in updates]
        await self.queue_service.publish_search_result(
            SearchResultDto(tg_id=search_request.tg_id,
query=search_request.query, links_list=links))

    async def article_processing_task(self):
        await
self.queue_service.consume_article_updates_task(self.article_callback)
```

```
    async def search_requests_processing_task(self):
        await
self.queue_service.consume_search_updates_task(self.search_request_callback)
```

Проектування і реалізація мікросервісу "Bot":

Мікросервіс "Bot" може бути проектований і реалізований з використанням бібліотеки aiogram. Основна функціональність мікросервісу "Bot" включає наступне:

1. Мікросервіс "Bot" використовує бібліотеку aiogram для роботи з Telegram.
2. Команда "search": Мікросервіс "Bot" має команду "search", яка дозволяє користувачам виконувати пошук публікацій за пошуковим запитом. Користувачі можуть ввести пошуковий запит, і мікросервіс "Bot" обробляє цей запит і виконує пошук відповідних публікацій.
3. Команда "add_monitoring_query": Мікросервіс "Bot" також має команду "add_monitoring_query", яка дозволяє користувачам додавати моніторингові запити. Користувачі можуть вказати критерії пошуку, за якими вони бажають отримувати нотифікації про нові публікації.
4. Взаємодія з базою даних: Мікросервіс "Bot" взаємодіє з базою даних, наприклад, PostgreSQL, для збереження і отримання інформації про користувачів і моніторингові запити. Це дозволяє зберігати дані про підписки користувачів і використовувати їх для надсилання нотифікацій про нові публікації.

```
bot = Bot(token=os.getenv("TELEGRAM_BOT_TOKEN"))
dp = Dispatcher(bot)

rabbitmq_service = RabbitMqService()
bot_service = BotService(rabbitmq_service)
```

```
async def process_notification(notification_dto: NotificationDto):
    for tg_id in notification_dto.tg_ids:
        await bot.send_message(tg_id,
NOTIFICATION.format(notification_dto.query, notification_dto.link))


async def process_search_result(search_result: SearchResultDto):
    await bot.send_message(search_result.tg_id,
construct_search_result_msg(search_result))


async def process_add_monitoring_query_input(message: types.Message):
    await bot_service.process_add_monitoring_query(message.text,
message.chat.id)


async def process_search_input(message: types.Message):
    await bot_service.process_search_request(message.text, message.chat.id)


@dp.message_handler(Command("search"))
async def search(message: types.Message):
    await Search.waiting_for_input.set()
    await message.answer("Please enter your query:")


@dp.message_handler(Command("add_monitoring_query"))
async def add_monitoring_query(message: types.Message):
    await AddMonitoringQuery.waiting_for_input.set()
    await message.answer("Please enter your query:")
```

```
@dp.message_handler(state=AddMonitoringQuery.waiting_for_input)
async def process_add_monitoring_query(message: types.Message, state:
FSMContext):
    await state.finish()
    await process_add_monitoring_query_input(message)
    await message.answer("Query added to monitoring list.")


@dp.message_handler(state=Search.waiting_for_input)
async def process_search(message: types.Message, state: FSMContext):
    await state.finish()
    await process_search_input(message.text)
    await message.answer("Search query sent.")
```

**Запуск бота:**

```
if __name__ == '__main__':
    await asyncio.gather(dp.start_polling(),
                         rabbitmq_service.consume_notifications(process_notification),

rabbitmq_service.consume_search_results(process_search_result))
```

**Контейнеризація і розгортання проекту**

Контейнеризація та розгортання проекту були реалізовані з використанням Docker, що є стандартним підходом і дозволяє забезпечити легкість управління залежностями та розгортанням середовища.

1. Docker є платформою для контейнеризації додатків, яка дозволяє упаковувати програмне забезпечення та всі його залежності у стандартизовані контейнери. Контейнери дозволяють запускати додатки відокремлено від операційної системи та незалежно від середовища розробки. Використовуючи Docker, ви можете забезпечити консистентність та переносимість вашого проекту між різними середовищами.
2. Docker Compose: Docker Compose є інструментом для оркестрації та управління багатоконтейнерними додатками. Використовуючи YAML-файл конфігурації, ви можете визначити сервіси, мережі та залежності вашого проекту. Docker Compose дозволяє легко створювати, запускати та зупиняти контейнери вашого проекту з однієї команди.

Розгортання: За допомогою вище описаних інструментів ви можете легко розгортати ваш проект на багатьох середовищах, включаючи локальну машину, сервери в хмарі або контейнерні платформи, такі як Kubernetes. Ви можете використовувати Docker-образи для створення однорідного середовища розгортання та забезпечити простоту масштабування вашого проекту.
Нижче представлений код конфігураційного файлу Docker Compose

```yaml
version: '3.9'

services:
 sensor:
  build:
   context: .
   dockerfile: sensor/Dockerfile
  depends_on:
   - rabbitmq
   - postgres
   - meilisearch
  env_file:
   - microservice1/.env
  restart: always

 bot:
  build:
   context: .
   dockerfile: bot/Dockerfile
  depends_on:
   - rabbitmq
   - postgres
   - meilisearch
  env_file:
   - bot/.env
  restart: always

 processor:
  build:
   context: .
   dockerfile: processor/Dockerfile
  depends_on:
   - rabbitmq
   - postgres
   - meilisearch
  env_file:
   - processor/.env
```

```
    restart: always

  rabbitmq:
    image: rabbitmq:3.9-alpine
    env_file:
      - rabbitmq/.env
    ports:
      - '5672:5672'
      - '15672:15672'
    restart: always

  postgres:
    image: postgres:13-alpine
    env_file:
      - postgres/.env
    volumes:
      - postgres-data:/var/lib/postgresql/data
    ports:
      - '5432:5432'
    restart: always

  meilisearch:
    image: getmeili/meilisearch:latest
    env_file:
      - meilisearch/.env
    volumes:
      - meilisearch-data:/data.ms
    ports:
      - '7700:7700'
    restart: always

volumes:
  postgres-data:
  meilisearch-data:
```

## Резюме

Система, розглянута вище, є комплексним рішенням для збору, аналізу та обробки публікацій ЗМІ. Вона включає мікросервіси "Sensor", "Processor" та "Bot", які працюють разом для забезпечення повного циклу обробки інформації.

Мікросервіс "Sensor" відповідає за збір публікацій, використовуючи різні джерела і бібліотеки для отримання, парсингу та зберігання даних. "Processor" виконує обробку публікацій, застосовуючи налаштовані етапи обробки, включаючи моделі машинного навчання. "Bot" забезпечує комунікацію з користувачем і повідомляє про результати пошуку та нотифікації.

Система має потенціал для подальшого розвитку, зокрема, розширення джерел публікацій, покращення аналітичних можливостей, вдосконалення взаємодії з користувачем, розширення підтримки мов та інтеграцію з іншими системами.

Для зручності розгортання та управління системою пропонується використовувати Docker та docker-compose для контейнеризації та оркестрації мікросервісів.

Усі ці елементи разом створюють потужну систему, яка може бути використана для збору, аналізу та обробки публікацій ЗМІ з метою отримання цінної інформації та нотифікації користувачів.

**Список використаних джерел**


1. https://cloud.google.com/docs – Документація GCP
2. https://newspaper.readthedocs.io/en/latest/ – Бібліотека парсингу публікацій
3. https://www.rabbitmq.com/documentation.html – Документація RabbitMq
4. https://huggingface.co/ – Ерзац гітхабу для тих, хто в ML
5. https://docs.docker.com/reference/ – Документація Docker
6. https://core.telegram.org/bots/api – Документація Telegram Bot Api
7. https://www.meilisearch.com/docs – Документація Meilisearch і відповідної бібліотеки для роботи з цією пошуковою системою
8. https://docs.peewee-orm.com/en/latest/ – Документація ORM фреймворку, використаного в цій роботі

**Код:**

**https://gitlab.com/p4lp471n3/newsparser**