ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Вдосконалення процесу агрегації контенту з різних джерел з використанням уніфікованого інтерфейсу»

на здобуття освітнього ступеня магістра
зі спеціальності 121 Інженерія програмного забезпечення
(код, найменування спеціальності)
освітньо-професійної програми «Інженерія програмного забезпечення»
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олег КОРНІЄНКО
(підпис)

Виконав: здобувач вищої освіти групи ПДМ-62

Олег КОРНІЄНКО

Керівник: Олена НЕГОДЕНКО
к.т.н., доцент

Рецензент: Андрій БОНДАРЧУК
д.т.н., професор

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Магістр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
«_____» _____ 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Конієнко Олегу Романовичу

1. Тема кваліфікаційної роботи: «Вдосконалення процесу агрегації контенту з різних джерел з використанням уніфікованого інтерфейсу »

керівник кваліфікаційної роботи Олена НЕГОДЕНКО к.т.н., доцент,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «19» жовтня 2023 р. №145.

2. Строк подання кваліфікаційної роботи «29» грудня 2023 р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, параметри мережі 5G, вимоги до хмарних сервісів.

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

1. Аналіз предметної галузі.
2. Аналіз існуючих інструментів агрегатора.
3. Практична реалізація удосконаленої архітектури системи.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз існуючих інструментів агрегатора новин.
2. Схема Splitter Aggregator.
3. Типова схема роботи агрегатора.
4. Порівняння підходів до збору контенту.
5. Порівняння підходів до зберігання контенту.
6. Порівняння підходів до кешування.
7. Порівняння підходів до індексації.
8. Схема роботи агрегатора з покращенням.
9. Порівняння показників типової і пропонованої архитектур.

6. Дата видачі завдання«19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	
2	Вивчення матеріалів для аналізу існуючих інструментів агрегатора	06.11-12.11.23	
3	Дослідження хмарних технологій	13.11-19.11.23	
4	Аналіз особливостей методів та моделей інтеграції потоків даних	20.11-26.11.23	
5	Дослідження підходів до збору, зберіганню та обробки контенту	27.11-03.12.23	
6	Застосування універсальної мікросервісної архітектури агрегатора	04.12-10.12.23	
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	
8	Розробка демонстраційних матеріалів	21.12-29.12.23	

Здобувач вищої освіти _____ Олег КОРНІЄНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Олена НЕГОДЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 70 стор., 7 табл., 17 рис., 36 джерел.

Мета роботи – покращення показників масштабування шляхом оптимізації архітектури системи.

Об'єкт дослідження – процес агрегації контенту з різних джерел.

Предмет дослідження – моделі керування агрегацією контенту з різних джерел.

Короткий зміст роботи: У роботі проведено аналіз та розробку архітектури для агрегатору контенту. Досліджено способи збору, зберігання, індексації та кешування контенту, що є ключовими для оптимізації обробки великих обсягів даних. Особлива увага приділяється оптимізації швидкості доступу та пропускної здатності системи. В результаті створено архітектуру, яка забезпечує ефективне масштабування агрегатора контенту.

КЛЮЧОВІ СЛОВА: АГРЕГАТОРИ НОВИН, МОДЕЛІ ІНТЕГРАЦІЇ, ХМАРНІ ТЕХНОЛОГІЇ, МАСШТАБУВАННЯ АГРЕГАТОРА КОНТЕНТУ, МІКРОСЕРВІСНА АРХІТЕКТУРА.

ABSTRACT

Text part of the master's qualification work: 70 pages, 17 pictures, 7 table, 36 sources.

The purpose of the work – improving scaling performance by optimizing the system architecture.

Object of research – the process of aggregating content from various sources.
Subject of research – models for managing the aggregation of content from different sources.

Summary of the work: The paper analyzes and develops the architecture for the content aggregator. Methods of content collection, storage, indexing and caching, which are key to optimizing the processing of large volumes of data, have been studied. Special attention is paid to optimization of access speed and system throughput. As a result, an architecture has been created that ensures effective scaling of the content aggregator.

KEYWORDS: NEWS AGGREGATORS, INTEGRATION MODELS, CLOUD TECHNOLOGIES, CONTENT AGGREGATOR SCALING, MICROSERVICE ARCHITECTURE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД	-	База Даних
AMQP	-	Advanced Message Queuing Protocol
REST	-	Representational State Transfer
UI	-	Інтерфейс користувача
NUI	-	Природний інтерфейс користувача
GUI	-	Графічний інтерфейс користувача
EIP	-	Шаблони корпоративної інтеграції
VDI	-	Мережеве налаштування
TUI	-	Відчутний інтерфейс користувача

ВСТУП

У нашу епоху існує велика кількість даних, де прориви та революційні зміни відбуваються через регулярні проміжки часу. Нас оточує чимало різних джерел інформації, які ми використовуємо у повсякденному житті. Одним із них є соціальні мережі. Кожен день люди витрачають години на просування стрічки соціальних мереж. Саме через ці стрічки користувачі отримують базове уявлення про події у світі. У цей час виникає потреба в коректній та швидкій обробці великих обсягів даних. Особливо гостро стоїть питання обробки даних у режимі реального часу, коли рішення повинні прийматися за секунди. І як наслідок, попередня проблема тягне за собою іншу – виникає проблема класифікації якості даних, які споживаються.

Обсяг створюваних даних зростає з кожним днем. Технологічний процес приніс великі зміни у світ зберігання, обробки та сприйняття інформації. У результаті з’явився новий термін, який описує масштаби нового світу – великі дані.

Завдяки великим данім відкриваються нові досягнення у світі технологій. За останні два роки великі дані змінили сам спосіб перегляду та зберігання даних компаніями, зараз вони дозволяють точно маніпулювати великими обсягами даних, які стали доступними для всіх. Досліджено, що щодня створюється 2,5 квінтильйона байтів даних і в майбутньому це число тільки зросте.

Кожна компанія, незалежно від її розміру, генерує дані. Це може бути інформація про клієнтів, дані про співробітників, дані про продажі, трафік користувачів, їх інтереси, переваги. Також усім відомо, що підприємства, фабрики та виробництва будь-якого розміру генерують велику кількість інформації: стан машин, етапи виробництва тощо. Великі дані вже сьогодні змінюють бізнес кількома способами.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

У наш час, коли величезна кількість людей проводять багато часу в Інтернеті, використовують його для отримання необхідної інформації та спілкування з іншими людьми, виникає необхідність створення сервісів, які спрощують ці дії. Пошук новин, статей і цікавого контенту – все це можливо завдяки сучасному Інтернету. З метою економії часу на пошук надійних джерел, актуальним стає питання про можливість фільтрації контенту шляхом його аналізу з різних джерел з використанням уніфікованого інтерфейсу.

1.1 Призначення та типи інтерфейсів користувача

Інтерфейс користувача визначає успіх програмного забезпечення. Кілька факторів, таких як простота використання, ефективність, доступність і естетичність, впливають на те, як користувачі без плутанини орієнтуються в системі, легко знаходять функції та отримують відповідний відгук про свої дії. Для покращення взаємодії з користувачем добре спроектований інтерфейс користувача повинен відповідати всім, тому для компаній вкрай важливо віддавати перевагу дизайну інтерфейсу користувача.

Інтерфейс користувача (UI) — це точка взаємодії між людьми та машинами. Він охоплює візуальні, слухові та тактильні елементи, які дозволяють користувачам взаємодіяти з комп’ютером. Це міст, який полегшує спілкування між людьми та машинами.

Дизайн інтерфейсу користувача створює вигляд інтерфейсу користувача програми або веб-сайту. Крім того, він включає інтерактивність і підключення між різними елементами інтерфейсу користувача. Це важливо, тому що забезпечує взаємодію з користувачем (UX) з подальшим тестуванням дизайну UX.

Дизайн інтерфейсу користувача стає невдалим, якщо користувач не має хорошого досвіду після його використання. Хороший інтерфейс користувача має

бути зручним і зрозумілим для користувачів. Отже, дизайн інтерфейсу користувача спрямований на те, щоб забезпечити споживачам гарну взаємодію з користувачем.

UI та UX — це концепції, які працюють паралельно для створення веб-сайту, послуги чи будь-якого програмного продукту. Інтерфейс користувача (інтерфейс користувача) пов’язаний із візуальними властивостями продукту, що включає зовнішній вигляд, відчуття, дизайн і структуру всіх елементів, з якими потрібно взаємодіяти.

У той час як UX (користувальницький досвід) передбачає поведінкові властивості інтерфейсу точніше, визначає, як користувач може взаємодіяти з вашим продуктом і змушує його відчувати його, використовуючи його. Але вони залежать один від одного. Відсутність одного з них зробить інший марним.

Розробники інтерфейсу користувача виконують багатозадачну роль, яка включає:

По-перше, вони повинні зрозуміти, як люди працюють з інтерактивними візуальними елементами. Потім вони створюють довіру між цими елементами. Вони повинні переконатися, що інтерфейс користувача є привабливим і ефективним, а також відповідає бізнес-цілям. Їм слід регулярно перевіряти дизайн інтерфейсу користувача, враховуючи колір, розмір, масштабованість і проміжки між різними елементами.

Існують декілько типів інтерфейсів користувача. Інтерфейси користувача (UI) мають різні форми, кожна з яких відповідає різним стилям взаємодії та вподобанням користувача. Графічні інтерфейси користувача, графічні інтерфейси користувача з сенсорним екраном, інтерфейси на основі меню, інтерфейси командного рядка, розмовні інтерфейси користувача, звичайні інтерфейси користувача, інтерфейси віртуальної реальності та інтерфейси доповненої реальності є поширеними типами інтерфейсів користувача.

1. Графічний інтерфейс користувача:

Графічний інтерфейс користувача (GUI) — це візуальний інтерфейс, який використовує піктограми, меню та кнопки для взаємодії користувача. Графічний інтерфейс користувача приховує складність дій за простими елементами керування

та пропонує миттєвий візуальний зворотний зв’язок. Однак GUI потребують значних ресурсів енергії та пам’яті. Отже, вони страждають від низької видимості та переважної кількості елементів керування.

Більшість інтерфейсів користувача, згаданих тут, розглядатимуться як інтерфейс GUI. Вони дозволяють користувачам взаємодіяти з комп’ютером за допомогою візуальних елементів, таких як графічні значки, кнопки та вікна, представлені на екрані.

Графічний інтерфейс користувача з сенсорним екраном став дуже популярним після появи планшетів, смартфонів і переносних пристроїв. Цей тип дизайну інтерфейсу користувача зробив революцію в обчислювальних технологіях і став для нас основним способом взаємодії з комп’ютерами. Незважаючи на те, що ця технологія просувається у VR та AR, графічний інтерфейс користувача, ймовірно, стане більш досконалим та інноваційним, оскільки потреба у візуальних елементах завжди буде сильною.

2. Природний інтерфейс користувача:

Природний інтерфейс користувача (NUI) передбачає взаємодії, які імітують жести та поведінку в реальному світі, включаючи дотик, голос, жести руками, рухи тіла та міміку. Природні інтерфейси користувача спрямовані на створення більш інтуїтивного та природного способу взаємодії з машинами.

3. Інтерфейс, керований меню:

Керовані меню інтерфейси (MDUI) покладаються на меню та підменю для навігації. Корисний для початківців завдяки своїй звичності та низькому когнітивному навантаженню. Користувачі переходять між різними екранами, натискаючи пункти меню. Незважаючи на простоту, він може обмежувати параметри, приховувати підменю та займати значний простір на екрані.

Керований меню інтерфейс дозволяє користувачам взаємодіяти з серією меню для навігації файловою структурою або каталогом, щоб знайти функції та функції за допомогою графічного інтерфейсу користувача. Вони можуть надавати логічні структури для складних каталогів і надавати відносно інтуїтивно зрозумілі засоби взаємодії.

MDUI використовуються в автомобілях для навігації налаштуваннями та функціями системи. Так само вони використовуються в годинниках для подібних завдань. Оскільки ці пристрої мають невеликі екрани, але багато функціональних можливостей, меню дозволяє користувачам переходити між ними через поступове розкриття, щоб вони могли продовжувати зосереджуватися на своїх реальних завданнях, не втрачаючи свого місця на пристрої.

4. Інтерфейс користувача терміналу:

Інтерфейс терміналу або інтерфейс командного рядка. Використовується досвідченими користувачами. Це тип дизайну інтерфейсу користувача, який дозволяє вам взаємодіяти з ним, вводячи команди та отримуючи текстовий вихід.

Вони використовуються для системного адміністрування та виконання технічних завдань у проектуванні програмного забезпечення, розробці, безпеці та інших пов’язаних сферах. Надають розробникам програмного забезпечення та системним адміністраторам потужний спосіб давати конкретні команди, щоб допомогти виконати широкий спектр завдань, включаючи керування файлами та каталогами, запуск сценаріїв і програм, а також налаштування параметрів операційної системи.

Інтерфейс командного рядка передбачає текстові взаємодії та популярний серед системних адміністраторів і програмістів. Він пропонує ефективність, масштабованість і можливості автоматизації. Однак інтерфейси командного рядка вимагають певного знання та можуть запропонувати обмежену інтуїтивно зрозумілу зручність використання, що робить вирішальними правильну обробку помилок і документацію.

5. Розмовний інтерфейс користувача:

Розмовні інтерфейси користувача (розмовні інтерфейси користувача) дозволяють користувачам спілкуватися голосом або текстом, нагадуючи людську розмову. Голосовий інтерфейс користувача є підкатегорією розмовного інтерфейсу користувача. Розмовні інтерфейси користувача є універсальними, персональними та адаптуються до поведінки користувача. Однак їм може бракувати візуальних

підказок і вимагати ретельного проектування, щоб імітувати природний потік розмов.

6. Сенсорний графічний інтерфейс користувача:

Сенсорний графічний інтерфейс користувача (сенсорний графічний інтерфейс користувача) поширений у портативних пристроях, таких як смартфони. Користувачі взаємодіють з інтерфейсом, торкаючись, проводячи пальцем і торкаючись, пропонуючи швидке та пряме залучення. Його переваги включають легкість маніпулювання, доступність і адаптованість до різних пристроїв. Однак обмеження розміру елемента керування та випадкові активації можуть бути недоліками.

Багато з раніше згаданих технологій можна вважати безконтактними інтерфейсами. Це категорія інтерфейсу, яка дозволяє користувачам взаємодіяти з системою без фізичного торкання екрана чи пристрою введення. Натомість вони використовують датчики для виявлення жестів, рухів і команд користувача.

Ці інтерфейси досліджуються як способи усунення бар’єру фізичних входів, що робить досвід ігор і розваг менш захоплюючим. Мета полягає в тому, щоб створити простіший та інтуїтивно зрозуміліший спосіб взаємодії людей із системами та технологіями.

Сенсорний інтерфейс вимагає від користувача взаємодії з технологією за допомогою сенсорного екрана або інших чутливих до дотику поверхонь. Сенсорні інтерфейси використовують датчики для виявлення розташування, рухів і тиску пальця або вказівного пристрою користувача на екран і перетворюють це на команди та дії.

Вони стають все більш популярними з появою смартфонів, планшетів і переносних пристроїв. Це чудовий спосіб дозволити користувачам швидко взаємодіяти з візуальними дисплеями, при цьому вони мають увагу та зосередженість на екрані.

7. Інтерфейси віртуальної реальності та доповненої реальності (AR):

Інтерфейси віртуальної реальності (VR) занурюють користувачів у змодельоване середовище, тоді як інтерфейси доповненої реальності (AR)

накладають цифрову інформацію на реальний світ. Інтерфейси VR і AR часто включають жести, рухи голови та контролери для взаємодії [9].

8. Інтелектуальні інтерфейси користувача

Інтелектуальні користувацькі інтерфейси використовують технології штучного інтелекту та машинне навчання, щоб створити персоналізований досвід для користувачів. Технологія використовується для розуміння та інтерпретації намірів і контексту користувача, перш ніж надати індивідуальний або керований досвід.

Прикладами інтелектуальних інтерфейсів користувача можуть бути чат-боти, голосові помічники та системи рекомендацій. Однією з найбільш популярних реалізацій цього типу технології є механізми персоналізації, які тепер вбудовані в багато корпоративних рішень.

9. Пакетний інтерфейс

Пакетний інтерфейс приймає одну команду від користувача та обробляє кілька завдань одночасно без подальшої взаємодії з користувачем. Вони корисні для обробки великих обсягів даних і виконання багатьох завдань, які добре піддаються аналізу або автоматизації.

Ранні комп’ютери працювали на пакетних інтерфейсах, оскільки програмісти передавали комп’ютеру набір інструкцій, а потім дозволяли комп’ютеру працювати для обробки результатів. Більш сучасні приклади включають аудит операцій у фінансових установах або автоматизоване тестування програмного забезпечення.

10. Інтерфейс WIMP

WIMP означає «Windows, значки, меню, покажчик» і означає графічний інтерфейс користувача, який використовує ці елементи, щоб дозволити користувачеві взаємодіяти з системою. Наприклад, ноутбуки та настільні комп’ютери, які використовують мишу та клавіатуру комп’ютера, є прикладами інтерфейсу WIMP.

Хоча це найпопулярніший тип інтерфейсу користувача, він все ще вважається незграбним і неефективним, оскільки це не зовсім ергономічна система.

Через відсутність кращого рішення це найуніверсальніше, але люди зараз досліджують інші шляхи, як голосові та матеріальні інтерфейси.

11. Відчутний інтерфейс користувача (TUI)

Відчутний інтерфейс користувача (TUI) використовує фізичні об’єкти та дотик як основний засіб взаємодії з технологією. На відміну від інтерфейсу WIMP, який використовує мишу та комп’ютер. Матеріальні інтерфейси мають компоненти, які можна торкатися, переміщати та маніпулювати ними, щоб керувати технікою.

TUI зазвичай мають форму настільних дисплеїв та інтерактивних стін, які використовують датчики для інтерпретації дій користувача в команди. Ми бачимо багато таких інтерфейсів у науково-фантастичних фільмах, де герої керують технологіями, переміщуючи об’єкти на столі або взаємодіючи з голограмами.

12. Кінетичний інтерфейс користувача

Кінетичний інтерфейс користувача (KUI) є ще одним прикладом матеріального інтерфейсу користувача, який покладається на жести та рухові команди. Ці інтерфейси використовують камери та датчики глибини для інтерпретації рухів і жестів, щоб створити захоплюючий і природний досвід користувача.

KUI використовуються в іграх, освіті та мистецьких інсталяціях, щоб дозволити користувачам контролювати навколишнє середовище, просто рухаючи своїми тілами та говорячи. Вони особливо корисні для керування середовищами віртуальної реальності та керування роботами за допомогою жестів.

13. Природний інтерфейс користувача

Природні користувальницькі інтерфейси (NUI) поєднують кілька згаданих раніше технологій, щоб створити безперебійний інтерфейс між людиною та технологією. Мета полягає в тому, щоб полегшити людям взаємодію з системами без потреби в традиційних вводах, таких як екрани, клавіатури та миші.

NUI мають багато форм, включаючи голосових помічників та інтерфейси на основі жестів, але вони поєднують їх у спосіб, який може бути невидимим і цілком

природним для користувача. Вони інтерпретують голосові команди та використовують машинне навчання, щоб зрозуміти контекст і запити користувача.

14. Органічний інтерфейс користувача

Органічний інтерфейс користувача (OUI) спрямований на використання вхідних даних, які максимально наближені до органічних організмів і реального світу. Вони імітують органічні форми або системи, такі як клітини, організми та екосистеми. Вони мають елементи дизайну, натхненні формами та поведінкою, які зустрічаються в природі.

OUI найчастіше використовуються для створення ефекту занурення та залучення в мистецьких інсталяціях та інтерактивних медіа. Вони є чудовим навчальним інструментом, і їх можна використовувати для пояснення складних концепцій та ідей у науці та освіті за допомогою візуальних і звукових пейзажів.

15. Інтерфейс користувача на основі форм

Інтерфейс користувача на основі форм — це тип графічного інтерфейсу користувача, який представляє користувачеві. набір форм або шаблонів, які крок за кроком збирають дані користувача. Інтерфейс користувача на основі форм використовує текстові поля, меню, що випадає, і прапорці, щоб фіксувати введені користувачем дані в структурований і логічний спосіб.

Інтерфейси користувача на основі форм корисні в програмах, які вимагають від користувача введення великої кількості даних певним чином. Наприклад, заявка на позику, візу або налаштування операційної системи чи програмного забезпечення.

Не існує єдиного інтерфейсу користувача, який би працював для всіх. Вибір найкращого інтерфейсу користувача залежить від потреб і вподобань користувача. Графічні інтерфейси ідеально підходять для простої взаємодії, графічні інтерфейси для сенсорних екранів для швидкої взаємодії дотиком, керовані меню інтерфейси для простоти, інтерфейси командної команди для ефективності та автоматизації, а розмовні інтерфейси для природної взаємодії.

Крім того, продукти можуть використовувати кілька інтерфейсів користувача: Мультимодальні інтерфейси користувача. Наприклад, iPhone

використовує сенсорний екран і голосовий інтерфейс користувача. Так само пошук Google використовує графічний і голосовий інтерфейси користувача.

1.2 Потенційні проблеми

1. Масштабування.

Масштабування - це здатність системи обробляти зростаюче навантаження шляхом додавання ресурсів. Це дуже важлива характеристика для систем, які повинні працювати з великими обсягами даних та трафіку. Якщо система не масштабується, то з часом вона буде стикатися з перевантаженнями та збоями.

Наприклад, якщо ми проектуємо архітектуру для агрегатора контенту, то слід очікувати, що кількість користувачів та їх активність буде зростати. Відповідно, система повинна витримувати це зростання навантаження.

Масштабованість досягається за допомогою гнучкої архітектури та технологій, що дозволяють легко додавати обчислювальні ресурси, такі як процесори, пам'ять, простір для зберігання даних. Це може бути горизонтальне масштабування, розподілені бази даних, кешування та інші методи.

Загалом, масштабованість визначає здатність системи рости та відповідати потребам бізнесу в майбутньому. Відсутність масштабованості рано чи пізно призведе систему до збоїв та втрати продуктивності. Тому це критично важливий аспект архітектури.

2. Недостатня пропускна здатність

Пропускна здатність - це обсяг даних, який система може обробити за певний проміжок часу. Оптимізація пропускної здатності означає підвищення ефективності системи в плані швидкості обробки даних. Це досягається за рахунок використання різних методів та підходів.

Для такої системи як агрегатор контенту оптимізація пропускної здатності є вкрай важливою, адже користувачі очікують швидкого завантаження даних. Недостатня пропускна здатність призведе до повільної роботи сервісу. Оптимізація

може включати кешування даних, стиснення, використання CDN, асинхронне завантаження ресурсів, балансування навантаження та інші методи.

Головна мета - мінімізувати затримки при обробці запитів користувачів, підвищити швидкість передачі даних, знизити навантаження на сервери.

Загалом, оптимізація пропускної здатності дозволяє системі ефективніше використовувати наявні ресурси та підвищити задоволеність користувачів.

3. Непристосованість до сплесків активності.

Обробка великих навантажень стосується здатності системи працювати стабільно та без збоїв за умови значних обсягів трафіку та запитів від користувачів.

Це критично важливо для сервісів, які можуть стикатися з різким зростанням аудиторії або активності. Наприклад, коли якийсь контент раптом стає популярним в агрегаторі.

Якщо система не спроможна обробляти такі сплески навантаження, це призведе до уповільнення роботи, помилок чи навіть збоїв. Користувачі не зможуть отримати доступ до контенту. Для уникнення таких ситуацій архітектура має бути розроблена з урахуванням можливих пікових навантажень. Це може досягатися різними способами на рівні БД, кешування, балансування тощо.

Головне завдання - забезпечити стабільну роботу сервісу за будь-якого рівня навантаження. Адже це безпосередньо впливає на якість обслуговування користувачів агрегатора контенту.

4. Надійність.

Надійність - це здатність системи виконувати свої функції безперервно і коректно, незважаючи на різні збої чи проблеми. Надійність є вкрай важливою властивістю для такої системи як агрегатор контенту, адже перерви у роботі призведуть до втрати користувачів.

Забезпечення надійності архітектури досягається різними способами на рівні апаратного і програмного забезпечення. Наприклад, реплікацією даних, контролем цілісності, механізмами відмовостійкості та іншими методами.

Головне завдання - мінімізувати ймовірність збоїв та час простою. Користувачі повинні мати стабільний доступ до контенту агрегатора. Тому надійність є ключовим аспектом при проектуванні архітектури.

5. Допущення.

При проектуванні подібної системи варто зробити певні припущення щодо характеру навантаження та використання. Це допоможе сформувати базові вимоги до архітектури. Зокрема, можна очікувати, що співвідношення операцій читання та запису в базу даних складатиме приблизно 10:1 або навіть 100:1. Адже в такій системі користувачі переважно переглядають контент, а оновлення відбуваються порівняно рідко.

Ймовірно, навантаження розподілятиметься нерівномірно, з піками активності вранці та ввечері, коли люди переглядають новини. Також очікується часова локалізація більшості запитів до найсвіжішого контенту.

Водночас популярний «старий» контент, скоріш за все, матиме довгий «хвіст» запитів, хоча й незрівнянно менший за обсягом. Основне навантаження спричинятимуть текстові та графічні дані. Відео та аудіо трохи меншою мірою.

Головні сценарії користувачів - це перегляд списку новин, пошук та перегляд конкретних матеріалів. Очікується глобальна аудиторія з усього світу, отже система має забезпечувати швидкий доступ незалежно від локації. Крім того, варто бути готовим до потенційно швидкого зростання популярності сервісу та відповідного збільшення навантаження.

6. Залежності.

Кількість і характеристики серверів бекенду. Оптимальна кількість серверів залежить від очікуваного навантаження та вартості інфраструктури. Недостатня кількість призведе до перевантаження та збоїв. Занадто багато - до невиправданих витрат. Важливі характеристики: кількість ядер процесора, об'єм ОЗП, пропускна здатність мережі та дисків.

Налаштування кешування впливає на швидкодію та навантаження на БД. Необхідно оптимізувати час життя кешу, максимальний розмір, алгоритми видалення даних. Кеш має вміщувати найбільш часто запитувані дані.

Стратегія індексації та пошуку контенту значною мірою впливає на швидкість генерації сторінок для користувачів. Оптимальна структура індексів, їх розмір та алгоритми пошуку мають забезпечувати мінімальний час видачі результатів пошукових запитів.

Конфігурація БД: кількість ядер, пам'ять, кількість реплік, розподіл даних по серверах, налаштування реплікації та шардингу. Має забезпечувати необхідну пропускну здатність та низьку затримку операцій.

Пропускна здатність та затримки мережі для внутрішніх з'єднань. Мережа не повинна бути "вузьким місцем", що уповільнює роботу. Балансування навантаження між компонентами та центрами обробки даних для розподілу трафіку та підвищення відмовостійкості.

1.3 Постановка задачі

1. Проаналізувати існуючі підходи та архітектуру побудови агрегаторів контенту. Розглянути їх переваги та недоліки.

2. Дослідити підходи до збору та зберігання контенту з різних джерел.

3. Порівняти ефективність різних стратегій кешування та індексації.

4. Запропоновати за результатами аналізу універсальну мікросервісну архітектуру агрегатора, що дозволяє гнучко масштабувати окремі компоненти.

2 АНАЛІЗ ІСНУЮЧИХ ІНСТРУМЕНТІВ АГРЕГАТОРА

2.1 Загальний огляд інструментів агрегаторів новин

У сфері корпоративної інтеграції обмін повідомленнями служить наріжним каменем для полегшення спілкування та обміну даними між програмами та системами. Платформи проміжного програмного забезпечення, орієнтованого на повідомлення (MOM), такі як Apache Kafka, RabbitMQ і IBM MQ, забезпечують надійну інфраструктуру для ефективного управління та обробки повідомлень.

Однак розробка та впровадження складних потоків обміну повідомленнями може бути складним завданням, що потребує ретельного розгляду маршрутизації повідомлень, перетворення та обробки помилок.

Агрегатор служить для об'єднання даних. Агрегатор — це процес із збереженням стану, який збирає (агрегує) пов’язані повідомлення, а коли всі необхідні повідомлення отримані, формує з них кінцеве повідомлення або об’єкт, який передається у вихідний потік для подальшої обробки. На рис.2.1 наведена схема Splitter Aggregator.

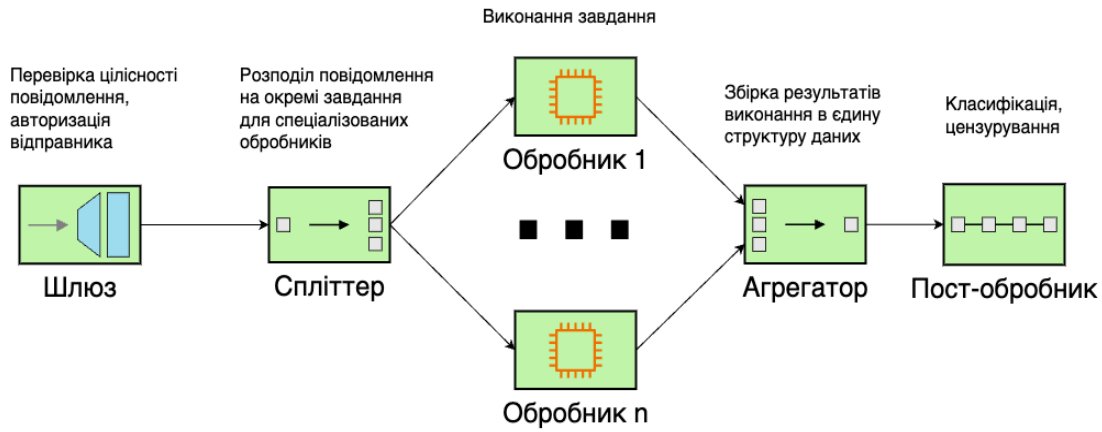


Рис. 2.1. Схема Splitter Aggregator

Наведемо структуру та функціональні можливості роутера спліттера. Шаблон маршрутизатора спліттера складається з двох основних компонентів: спліттера та маршрутизатора. Роздільник відповідає за поділ вихідного повідомлення на меньші підповідомлення або фрагменти на основі попередньо визначеної логіки. Ця логіка може ґрунтуватися на вмісті повідомлення, заголовках повідомлення або зовнішніх факторах, таких як час або тригери подій.

Маршрутизатор, з іншого боку, отримує розділені повідомлення від спліттера та визначає їх кінцеві пункти призначення. Він може використовувати різні стратегії маршрутизації, включаючи кругову маршрутизацію, балансування навантаження або маршрутизацію на основі вмісту повідомлень. Маршрутизатор гарантує, що кожне підповідомлення направляється до відповідного адресата, гарантуючи, що одержувачі повідомлення отримають відповідну інформацію.

Наведемо сценарії використання та переваги Splitter Aggregator. Шаблон Splitter Router знаходить застосування в широкому діапазоні сценаріїв, особливо при обробці великих повідомлень або складних структур даних. Наведемо кілька відомих сценаріїв використання:

Розкладання складних повідомлень: Маршрутизатор Splitter може розбивати велике повідомлення на менші, більш керовані підповідомлення, полегшуючи обробку кількома компонентами або системами.

Паралельна обробка: Розповсюджуючи підповідомлення між кількома адресатами, Splitter Router забезпечує паралельну обробку, забезпечуючи швидшу обробку великих обсягів даних.

Балансування навантаження: він може рівномірно розподіляти вхідні повідомлення між кількома одержувачами, сприяючи ефективному використанню ресурсів і балансуванню навантаження.

Агрегування результатів. Маршрутизатор Splitter можна використовувати в поєднанні з агрегатором для збору та повторного компонування результатів паралельної обробки або для реконструкції складних структур даних.

Шаблон Splitter Router є наріжним каменем Enterprise Integration Patterns, забезпечуючи гнучкий і масштабований механізм для розподілу.

Шаблони корпоративної інтеграції (EIP) виникли як стандартизований підхід до вирішення цих проблем, пропонуючи набір багаторазових шаблонів проектування, які забезпечують спільну мову та методологію для побудови архітектур обміну повідомленнями. Серед цих шаблонів Splitter Router виділяється як універсальна конструкція для поділу одного повідомлення на кілька адресатів або фрагментів і їх маршрутизації відповідно.

Шаблони корпоративної інтеграції (EIP) надають усталений словник і набір інструкцій щодо проектування для побудови цілісної та масштабованої архітектури обміну повідомленнями. Шаблон Splitter Router, фундаментальний компонент EIP, відіграє вирішальну роль у ефективному розподілі повідомлень між кількома пунктами призначення. Це підкреслює універсальність шаблону для вирішення різноманітних потреб інтеграції, таких як декомпозиція складних повідомлень, паралельна обробка та балансування навантаження.

Крім того, він досліджує взаємодію між маршрутизатором Splitter та іншими EIP, демонструючи його потенціал інтеграції в більш широкі системи обміну повідомленнями.

Агрегатор новин — це цифрова платформа, призначена для збору, керування та представлення новинного контенту з різних джерел в одному єдиному інтерфейсі. Діючи як єдине місце для різноманітних новин, ці агрегатори контенту збирають вміст із різноманітних джерел, таких як веб-сайти, блоги та канали соціальних мереж. Мета цих інструментів — запропонувати користувачам повний огляд поточних подій, тенденцій і цікавих тем.

Агрегатори новин використовують алгоритми для сканування, упорядкування та визначення пріоритетів вмісту на основі вподобань користувачів, гарантуючи, що люди отримають індивідуальну стрічку новин. Цей продуманий підхід дає користувачам доступ до широкого діапазону точок зору, забезпечуючи краще розуміння подій, що розгортаються [10].

Однією з ключових особливостей агрегаторів новин є їх здатність економити час для користувачів, яким інакше потрібно було б відвідувати кілька веб-сайтів,

щоб бути в курсі подій. Консолідуючи вміст на одній платформі, агрегатори новин спрощують процес навігації у величезному морі інформації, доступної в Інтернеті.

Ці платформи роблять значний внесок у демократизацію інформації, надаючи можливість користувачам бути добре поінформованими у зручний спосіб. Оскільки цифровий ландшафт продовжує розвиватися, агрегатори новин стають основними інструментами для навігації у всесвіті новин, що постійно розширюється, і бути в курсі подій, які формують наш світ.

Агрегатори новин є незамінними союзниками, пропонуючи безліч переваг, які виходять за межі традиційного споживання новин. Від персоналізованого курування до оновлень у реальному часі та доступності на різних платформах, ці інструменти переосмислюють те, як люди взаємодіють зі світом інформації, що постійно розвивається.

Зосереджуючись на ефективності, налаштуванні та різноманітних перспективах, агрегатори новин не лише спрощують процес споживання новин, але й сприяють більш поінформованому, взаємопов’язаному та залученому світовому співтовариству.

Використовувати ці платформи просто зручність, та стратегічний вибір у навігації в складності нашого багатого на інформацію століття. Наведемо приклади найпопулярнійших інструментів агрегатора новин за 2024 рік в таблиці 2.1.

Таблиця 2.1

Інструментів агрегатора новин

	Платформа	Джерела контенту	Максимальна кількість джерел контенту	Ранжування контенту
Flocker	Веб-додаток	Соціальні мережі	60	За датою публікації
Google News	Веб-додаток, Android, iOS	Сайти з новинами	20,000	За датою публікації
SmartNews	Android, iOS	Сайти з новинами	14,000	За датою публікації
Flipboard	Веб-додаток, Android, iOS	Сайти з новинами, незалежні видавці	4,000	За інтересами користувача
Пропонована система	Веб-додаток	Соціальні мережі, сайти з новинами, незалежні видавці	В залежності від доступних ресурсів	За датою публікації та інтересами користувача

У 2024 році кілька передових інструментів-агрегаторів контенту революціонізують курування та доставку контенту, задовольняючи різноманітні потреби та вподобання, одночасно оптимізуючи досвід споживання контенту: Flipboard, SmartNews,

Tagbox RSS Aggregator — це динамічний інструмент у постійно розширюваному ландшафті курування вмісту. Створений для користувачів, які шукають бездоганний досвід перегляду новин, Tagbox чудово збирає різноманітний вміст із RSS-каналів, пропонуючи єдину платформу для отримання інформації.

Його зручний інтерфейс спрощує навігацію зведеним вмістом, забезпечуючи безпроблемне вивчення останніх новин. Завдяки зосередженню на налаштуванні користувача та оновленнях у реальному часі Tagbox виділяється серед агрегаторів. Оскільки ми досліджуємо найкраще в цій галузі, Tagbox є прикладом ефективності та універсальності, які сучасні агрегатори висувають на перший план, покращуючи спосіб взаємодії користувачів із безліччю джерел новин. За допомогою цього інструменту ви можете вставляти канали RSS на свій веб-сайт, щоб зробити вашу веб-сторінку більш професійною та привабливою наведений на рис.2.1.

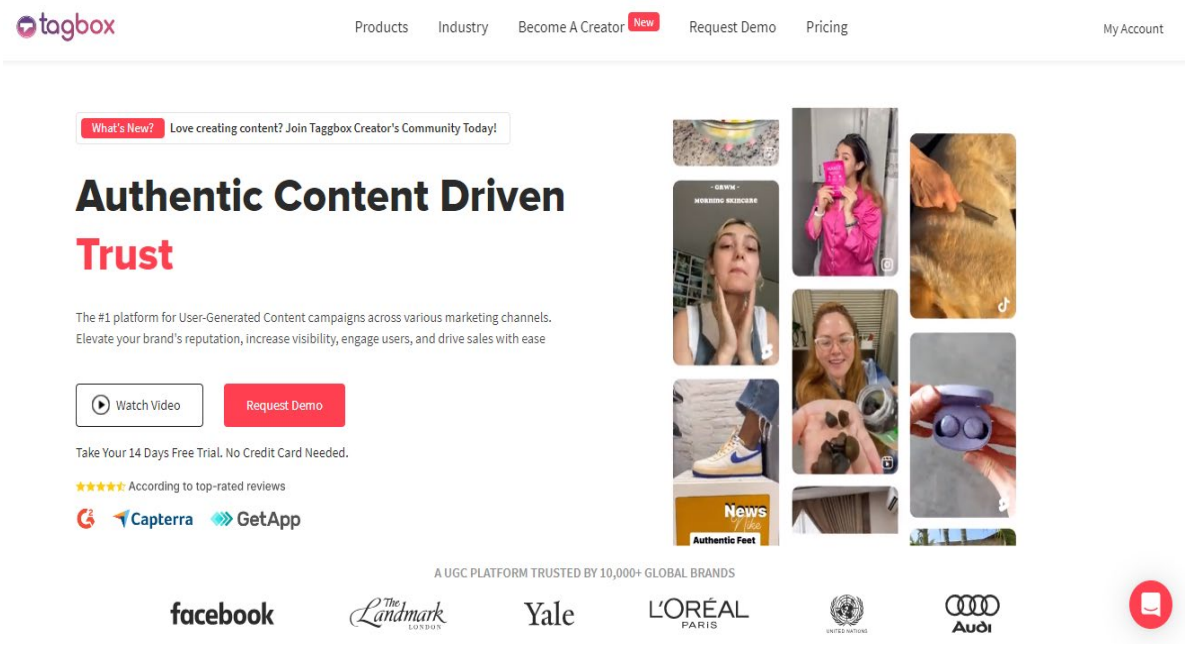


Рис.2.1. Tagbox RSS Aggregator

Flockler постає як динамічна платформа у сфері агрегації вмісту, пропонуючи інноваційні рішення для створення цікавих веб-сайтів і каналів. Цей універсальний інструмент чудово збирає вміст із різних джерел, перетворюючи його на візуально вражаючі дисплеї. Сила Flockler полягає в його здатності курувати та демонструвати створений користувачами контент, створюючи інтерактивну онлайн-присутність, керовану спільнотою.

Завдяки таким функціям, як оновлення та настроювані дисплеї, Flockler дає можливість компаніям і окремим особам підбирати вміст, який відповідає ідентичності їхнього бренду. Незалежно від того, чи використовується він для маркетингових кампаній, подій чи створення спільноти, Flockler виділяється своїм зручним інтерфейсом і можливістю адаптації, що робить його цінним надбанням для тих, хто прагне покращити свою присутність в Інтернеті за допомогою динамічного агрегування вмісту наведений на рис.2.2.

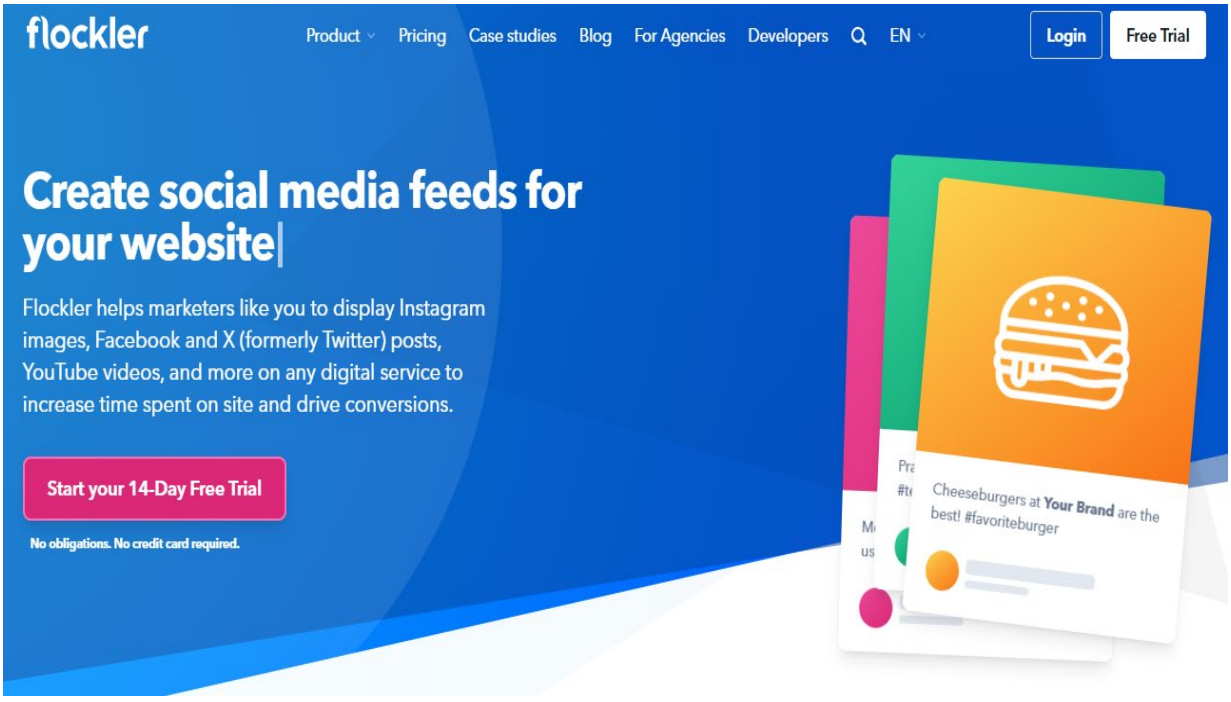


Рис.2.2. Flockler

Як світовий лідер у пошуку інформації Google News є наріжним каменем у світі агрегації новин. Завдяки широкому охопленню та потужним алгоритмам

Новини Google об’єднують вміст новин із безлічі джерел, надаючи користувачам комплексну та персоналізовану стрічку новин.

Цей агрегатор чудово надає оновлення на різноманітні теми, враховуючи індивідуальні вподобання та інтереси. Google News використовує складний підхід, поєднуючи машинне навчання та взаємодію з користувачем, щоб удосконалити та адаптувати процес доставки новин. У великому середовищі агрегаторів новин Новини Google продовжують встановлювати стандарти, демонструючи майстерність і вплив цих платформ у формуванні того, як ми споживаємо інформацію наведений на рис.2.3.

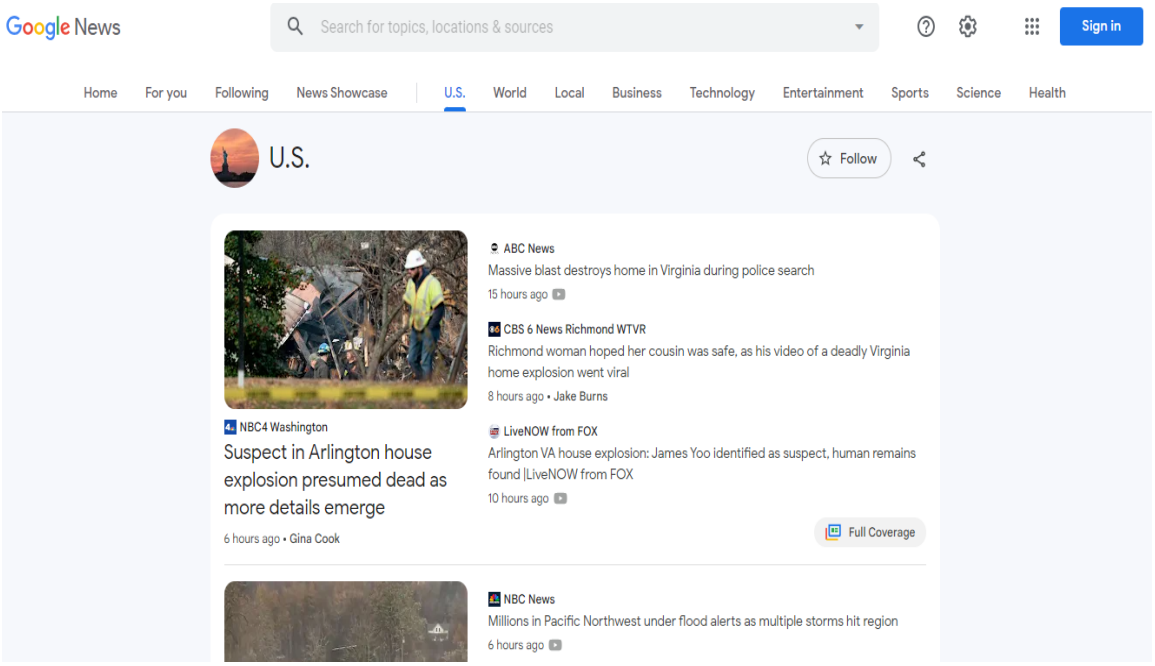


Рис.2.3. Google News

Flipboard: відомий своїм візуально привабливим макетом у журнальному стилі, Flipboard об’єднує вміст на основі вибраних користувачами тем, представляючи його у візуально привабливому форматі, схожому на персоналізований журнал. Він працює, збираючи статті, зображення та відео з багатьох джерел, представляючи їх в інтерактивному макеті, який можна настроювати наведений на рис.2.4.

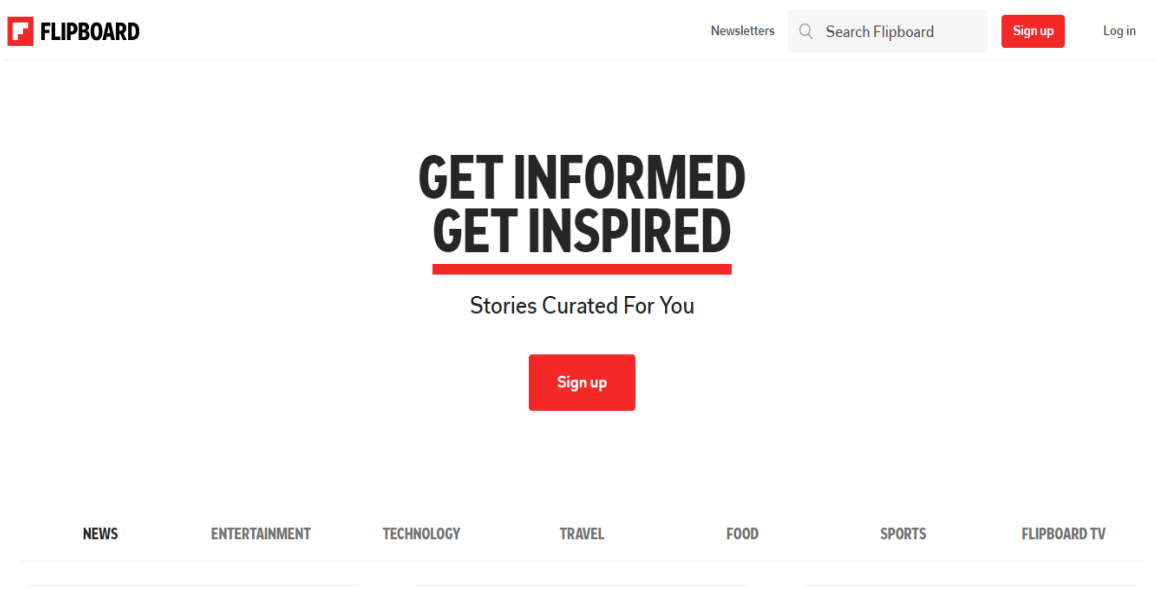


Рис.2.4. Flipboard

SmartNews: відомий своїм інтуїтивно зрозумілим інтерфейсом і персоналізованими рекомендаціями щодо новин, SmartNews збирає новинні статті з авторитетних джерел. Він використовує алгоритми машинного навчання, щоб підбирати статті відповідно до інтересів користувачів, зосереджуючись на швидкості та релевантності наведений на рис.2.5.

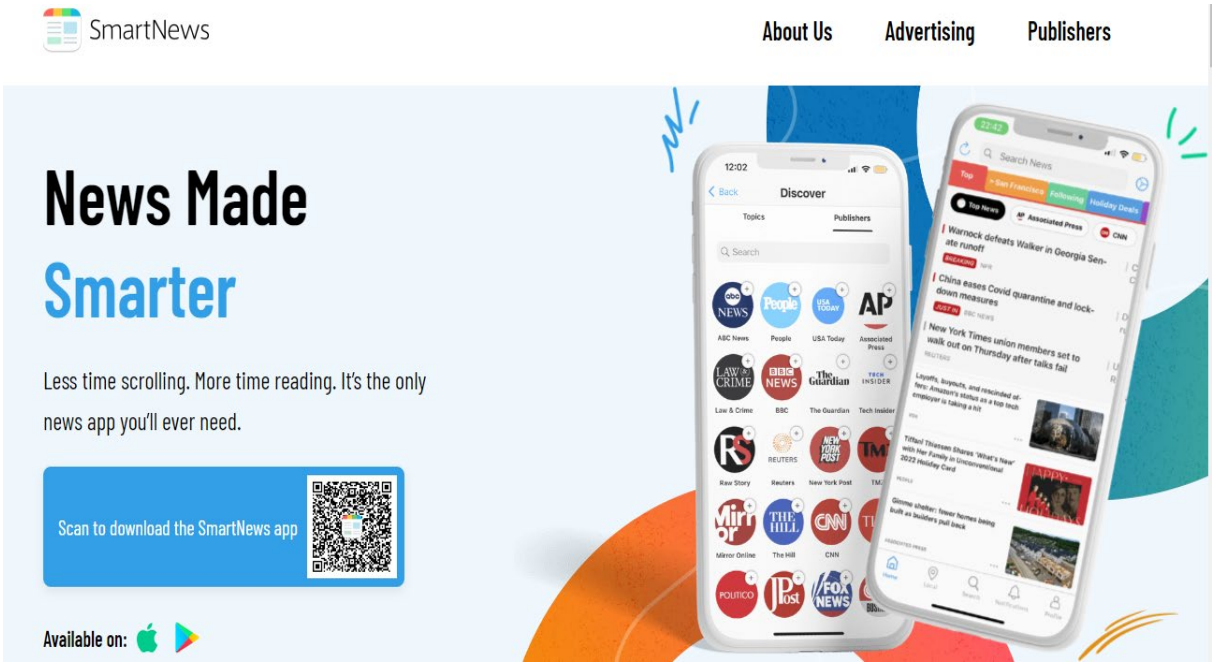


Рис.2.5. SmartNews

2.2 Загальний огляд агрегаторів контенту

Агрегація вмісту служить ключовим механізмом у цифровій сфері, об’єднуючи інформацію з різних джерел в єдину платформу. Він функціонує як центр, збираючи, організовуючи та відображаючи вміст із різноманітних каналів, пропонуючи користувачам спрощений досвід. Цей продуманий підхід надає користувачам централізоване розташування для доступу до актуальної інформації без переходу до кількох джерел.

Процес передбачає автоматичний збір і впорядкування вмісту на основі попередньо визначених критеріїв, таких як ключові слова, теми або налаштування користувача. Це дозволяє отримувати оновлення та гарантує, що користувачі отримають найактуальнішу інформацію. Агрегатори вмісту використовують алгоритми для сортування, класифікації та відображення вмісту в зручному для сприйняття форматі відповідно до вподобань і потреб користувачів.

У 2024 році численні інструменти та платформи дають змогу агрегувати вміст, кожна з яких має унікальні функції. Ці інструменти включають Feedly,

Flipboard і Pocket, серед інших, від складних алгоритмів до зручних інтерфейсів. Вони пропонують настроювані функції, дозволяючи користувачам підбирати вміст, стежити за певними темами та збирати інформацію відповідно до їхніх інтересів або галузевих потреб.

Оскільки цифровий ландшафт продовжує розширюватися, агрегатори контенту відіграють життєво важливу роль у спрощенні споживання інформації, надаючи користувачам єдиний пункт для отримання інформації та оновлення серед величезного простору онлайн-контенту.

Ефективне використання агрегаторів вмісту передбачає дотримання кількох найкращих практик для оптимізації керування та представлення вмісту:

Визначте чіткі цілі: встановіть конкретні цілі для агрегації вмісту, будь то надання інформації про галузь, покращення взаємодії з користувачем або стимулювання залучення [17].

Вибирайте якісні джерела: вибирайте вміст із надійних і різноманітних джерел, щоб забезпечити надійність, точність і комплексне уявлення про інформацію.

Персоналізація та релевантність: персоналізуйте вміст відповідно до інтересів користувачів або галузевих потреб, пропонуючи персоналізовані рекомендації або фільтри для більш релевантної взаємодії з користувачем.

Узгодженість і регулярне оновлення: дотримуйтеся постійного графіка оновлень вмісту, щоб підтримувати актуальність інформації та залучення користувачів. Регулярно оновлюйте та підбирайте вміст, щоб відображати останні тенденції чи розробки.

Оптимізована взаємодія з користувачем: віддавайте перевагу зручним для користувача інтерфейсам, інтуїтивно зрозумілій навігації та швидкості реагування на мобільних пристроях, щоб покращити доступність і залучення користувачів.

Куратор з контекстом: додайте цінність, надаючи контекст, ідеї або короткі описи поряд із підібраним вмістом, спрямовуючи користувачів і додаючи глибини їхньому розумінню.

Відстежуйте та аналізуйте продуктивність: відстежуйте показники залучення користувачів, відгуки та взаємодії, щоб оцінити ефективність стратегій агрегації вмісту. Використовуйте аналітику, щоб уточнювати й покращувати вибір і представлення вмісту.

Будьте в курсі юридичних та етичних міркувань: дотримуйтеся законів про авторське право та етичних практик пошуку вмісту, поважайте права інтелектуальної власності та вказуйте першоджерела.

Використовуючи ці найкращі практики, агрегатори контенту можуть оптимізувати свої платформи, забезпечуючи користувачам підібраний, привабливий і цінний контент, одночасно сприяючи довірі та надійності.

2.3 Огляд типів агрегаторів контенту

Агрегатори вмісту бувають різних форм, кожен з яких адаптований до різних потреб і вподобань у споживанні вмісту:

RSS-агрегатори: ці платформи збирають і відображають вміст із RSS-каналів, що дозволяє користувачам підписуватися на їхні улюблені веб-сайти чи блоги та отримувати оновлення в одній централізованій стрічці.

Агрегатори соціальних медіа: орієнтовані на соціальний вміст, ці агрегатори збирають публікації, зображення та відео з багатьох платформ соціальних медіа, пропонуючи уніфікований погляд на соціальну взаємодію.

Агрегатори новин: спеціально підібрані для вмісту новин, ці платформи збирають статті, заголовки та останні новини від різних видавців і представляють їх у структурованому форматі для легкого доступу.

Спеціальні агрегатори: ці агрегатори зосереджуються на конкретних галузях чи інтересах, зіставляючи вміст, що стосується нішевих тем, як-от технології, фінанси, здоров’я чи розваги.

Агреговані пошукові системи: деякі пошукові системи об’єднують вміст, об’єднуючи результати з кількох пошукових систем або джерел, надаючи користувачам комплексний пошуковий досвід.

Платформи контролю вмісту: ці інструменти дозволяють користувачам керувати вмістом з різних джерел вручну, організовуючи його в колекції або дошки, якими можуть ділитися або мати доступ інші.

Агрегатори електронних бюлетенів: ці агрегатори збирають і доставляють підібраний вміст за допомогою електронних бюлетенів, пропонуючи передплатникам підібрану добірку статей, порад або новин.

Кожен тип агрегатора вмісту задовольняє конкретні переваги вмісту або звички споживання, пропонуючи користувачам зручний і налаштований спосіб централізованого доступу до інформації з різних джерел і її споживання.

Агрегатори вмісту охоплюють різні платформи та галузі, пропонуючи різні підходи до керування та представлення вмісту:

Reddit: Reddit, який часто називають «головною сторінкою Інтернету», служить агрегатором вмісту, де користувачі збирають і діляться вмістом через спільноти (субреддити), присвячені певним темам, сприяючи обговоренню та пошуку вмісту.

Pinterest: відомий як платформа для візуального пошуку, Pinterest збирає вміст за допомогою дошок, які підбирають користувачі, що дозволяє користувачам збирати та ділитися зображеннями, статтями та відео на основі їхніх інтересів.

Новини Google: Новини Google об’єднують статті новин від різних видавців і джерел, представляючи їх у персоналізованій стрічці на основі вподобань користувача та історії веб-перегляду, пропонуючи широкий спектр новинних тем і перспектив.

TikTok: TikTok, платформа для обміну відео, збирає вміст за допомогою коротких відео, створених користувачами або творцями вмісту. Його алгоритмічний канал демонструє відео на основі взаємодії користувачів, сприяючи вірусному вмісту та тенденціям.

Pocket: Pocket діє як агрегатор вмісту, дозволяючи користувачам зберігати статті, веб-сторінки чи відео з різних джерел для подальшого використання. Він забезпечує централізовану бібліотеку, доступну на всіх пристроях.

Новини Apple: Новини Apple об’єднують новини та статті від різних видавців у персоналізовану стрічку на основі вподобань користувачів, надаючи підібраний вміст на різні теми та інтереси.

LinkedIn Pulse: агрегатор контенту LinkedIn демонструє статті, публікації та ідеї від лідерів галузі, впливових людей і публікацій. Він пропонує підібрану стрічку новин на основі професійних інтересів і зв’язків користувачів.

Ці приклади ілюструють різноманітність агрегаторів контенту в соціальних мережах, новинах, візуальному контенті та професійних мережах, які задовольняють уподобання користувачів щодо підібраного та персоналізованого контенту.

Ландшафт агрегації вмісту охоплює широкий спектр, демонструючи інноваційні платформи в різних галузях та інтересах. Кожен агрегатор пропонує

унікальний підхід до відбору та представлення вмісту, починаючи від обговорень спільноти Reddit і закінчуючи візуальними дошками Pinterest і персоналізованими стрічками новин Google.

Такі платформи, як TikTok, процвітають завдяки агрегації короткого відео, тоді як Pocket і Apple News зосереджуються на наданні користувачам персоналізованих бібліотек і підібраних каналів новин. LinkedIn Pulse обслуговує професіоналів, надаючи галузеву інформацію та статті.

Ці приклади підкреслюють еволюцію агрегації контенту, яка відповідає різноманітним уподобанням і потребам у цифрову еру, переповнену інформацією. Пропонуючи персоналізований і підібраний контент, ці агрегатори покращують залучення користувачів, полегшують пошук вмісту та оптимізують споживання інформації.

Серед цього різноманіття агрегатори контенту продовжують переглядати те, як користувачі отримують доступ і взаємодіють із контентом у різних нішах і галузях. Оскільки ці платформи розвиваються, вони наполегливо задовольняють попит на підібраний, персоналізований і захоплюючий контент, формуючи спосіб взаємодії користувачів з інформацією в цифровому середовищі, що постійно розширюється.

3 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ АРХІТЕКТУРИ

3.1 Архітектура в системах оброблення даних

REST – архітектурний підхід до побудови мережевих додатків, що надають доступ до певних ресурсів. Він був вперше описаний Роєм Філдінгом в його докторській роботі в 2000-у році.

REST базується на таких архітектурних особливостях як використання клієнт-серверної моделі, незбереження стану клієнта між запитами, кешування, уніфікований інтерфейс доступу до ресурсів, шарове розміщення компонентів та використання коду-по-запиту [17].

Використання клієнт-серверної архітектури зумовлене тим, що REST з самого початку проектувалася як архітектура для додатків, що будуть працювати на основі протоколу HTTP. Відокремлюючи проблеми користувальницького інтерфейсу від проблем зберігання даних, ми покращуємо переносимість інтерфейсу користувача на різні платформи і покращуємо масштабованість, спрощуючи компоненти сервера. Також цей поділ дозволяє компонентам розвиватися незалежно один від одного .

Наступним обмеженням архітектури REST є незбереження стану клієнта між запитами. Іншими словами, сервер не зберігає ніякої інформації про стан клієнту після надсилання відповіді на запит. Такий підхід дозволяє поліпшити стабільність системи та ефективніше використовувати ресурси сервера. Це не означає, що стан відсутній в системі як такий, але його збереження перекладається на сам клієнт. Саме клієнт відповідальний за передачу стану з кожним запитом, що в свою чергу дозволяє інкапсулювати серверну частину додатку та використовувати розподілювачі навантаження.

Проте, в цього рішення є й свої недоліки. Найбільш помітним з них є надлишкове використання ресурсів мережі. Оскільки клієнту потрібно передавати інформацію про стан з кожним запитом, це призводить до того що одна й та сама

інформація надсилається багато разів, замість того щоб бути збереженою на сервері. Крім того, розміщення стану програми на стороні клієнта зменшує контроль сервера над послідовною поведінкою програми, оскільки програма стає залежною від правильної реалізації семантики в різних версіях клієнта.

Можливість кешувати відповіді сервера це ще одна з ключових особливостей архітектури REST. REST вимагає щоб всі дані були явно відмічені: можна їх кешувати чи ні. Якщо дані можуть бути кешовані, клієнт відповідальний за те щоб скористатися цією перевагою, адже кешування даних дозволяє скоротити (а в деяких випадках уникнути повністю) додаткових інтеракцій з сервером, що в свою чергу поліпшить швидкодію додатка та може частково нівелювати надлишкове використання мережевих ресурсів зумовлене потребою передавати стан з кожним запитом.

І хоча кешування відповідей сервера може поліпшити швидкодію додатка і зробити його більш дружнім до користувача, воно також має і мінуси: кеш потрібно якось інвалідувати. Це одна з фундаментальних проблем при роботі з кешем, що була описана Філом Карлтоном [2]. Клієнту потрібно вирішити в який момент кеш стає неактуальним та потрібно зробити запит до серверу і оновити кеш. Недостатньо часта інвалідація кешу може призвести до розсинхронізації даних між клієнтом та сервером та подальших конфліктів при спробі оновити дані. Надто часта інвалідація кешу ж може повністю нівелювати виграші в швидкодії.

Універсального рішення для цієї проблеми не існує і для кожного додатку потрібен свій підхід. Найпопулярнішими підходами є time-to-live та e-tag. У випадку використання time-to-live, сервер, разом з ресурсом, передає інформацію про мінімальний та/або максимальний час на котрий цей ресурс можна помістити в кеш. Цей підхід дозволяє серверу керувати часом життя для різних ресурсів і ефективніше реалізовувати кешування різних видів ресурсів, хоча й не позбавляє від всіх потенційних проблем пов’язаних з інвалідацією кешу.

При використанні підходу e-tag кожній версії ресурсу назначається певний короткий ідентифікатор (тег). Після внесення будь яких змін до ресурсу, його тег

заміняється на новий. При отриманні відповіді клієнт зберігає цей тег разом з ресурсом і при подальших запитах до цього ресурсу також додає тег до запиту.

Якщо тег в запиті співпадає з тегом ресурсу на сервері, це означає що ресурс не змінювався і серверу не потрібно повертати весь ресурс, клієнт може використовувати кешовану версію. Такий підхід буде ефективним якщо сервер повертає ресурси великого об'єму, тоді використання тегу дозволить скоротити об'єм передачі даних по мережі і як наслідок поліпшить швидкодію додатку. Проте, такий підхід все ще потребує виконання запиту кожен раз, а отже створює затримку поки встановлюється з'єднання та очікується відповідь від сервера.

Центральною особливістю, яка відрізняє архітектуру REST від інших архітектур мережевих додатків, є його акцент на єдиному інтерфейсі між компонентами. Уніфікація інтерфейсу різних компонентів системи робить її розуміння простішим і робить окремі сервіси більш незалежними один від одного. Що в свою чергу позитивно впливає на швидкість розробки та дозволяє сервісам еволюціонувати швидше та незалежно один від одного. Уніфікація інтерфейсу реалізується декількома компонентами: ідентифікація ресурсів в запиті, маніпуляція ресурсом через репрезентацію, самоописні повідомлення, гіпермедіа як двигун стану додатку.

Окремі ресурси ідентифікуються в запитах, наприклад, за допомогою URI у REST веб-сервісах. Самі ресурси концептуально відокремлені від представлень, які повертаються клієнту. Наприклад, сервер може надсилати дані зі своєї бази даних у вигляді HTML, XML або JSON — жодне з яких не є внутрішнім представленням сервера. При цьому клієнт може виконувати маніпуляції над ресурсом за допомогою внесення змін до представлення. Завданням сервера стає правильно обробити ці зміни та внести їх до ресурсу.

Кожне повідомлення містить достатньо інформації, щоб описати, як його обробити. Наприклад, тип даних, що передаються (JSON, XML тощо) вказується в заголовках запиту, що в свою чергу вказує серверу який парсер використовувати щоб правильно обробити дані в запиті.

Використання шарового розміщення компонентів дозволяє додавати в систему проміжні компоненти. При використанні такого підходу клієнт не знає чи спілкується він зараз напряму з сервером чи між ними є ще декілька проміжних компонентів. Такими компонентами можуть бути проксі сервер (або зворотній проксі сервер), балансер навантаження чи WAF. Це дозволяє будувати багатоступінчату систему, де кожен компонент виконує певну задачу, без змін до коду серверу чи клієнта.

Код по запиту є єдиною необов'язковою характеристикою REST. Вона дозволяє передати разом з ресурсом код, котрий буде виконано на клієнті. Це дозволяє більш оперативно вносити зміни в додаток, оскільки деякі клієнти можуть не оновлятися роками. Проте, останнім часом цей підхід використовується все рідше [17].

3.2 Огляд та порівняння розповсюджених архітектур

Окрім REST, для побудови web API часто використовують такі архітектурні підходи як SOAP, GraphQL та RPC. І хоча всі вони допомагають досягти однієї цілі, вони дуже відрізняються в деталях, загальних підходах та популярності.

RPC – це спосіб викликати процедуру або функцію, розташовану на іншому комп'ютері, ніж той, де здійснюється виклик. RPC використовується для створення API, і його переваги включають можливість викликати процедури або функції на різних машинах, а також можливість викликати процедури або функції, які написані різними мовами програмування. RPC також має можливість передавати параметри за посиланням, що може бути корисно при передачі великих обсягів даних.

І хоча RPC може бути використаний для побудови веб-додатків, великої популярності тут він не досяг. Сьогодні RPC використовується зазвичай в розподілених системах для віддаленого виклику функцій на інших машинах та розподілення навантаження.

RPC не накладає майже ніяких обмежень на реалізацію протоколу. Для передачі даних може використовуватися будь-який формат даних. Наприклад, JSON, XML чи ProtoBuffers. Також немає ніяких обмежень при виборі транспорту, котрий використовується для передачі повідомлень: це може бути як і HTTP так і самописні протоколи на основі TCP або UDP.

Одним з наступників архітектури RPC є SOAP. Ця архітектура накладає більше обмежень (architectural constraints). І хоча такі обмеження роблять реалізацію цієї архітектури складнішою для розробника, вони ж роблять цю архітектуру більш стійкою, безпечною та дозволяють їй краще масштабуватися.

Архітектура була вперше розроблена в 1998-му році в Microsoft. Перші версії архітектури не були затверджені як стандарт World Wide Web Consortium, проте вже версія 1.2 була прийнята як стандарт та є останньою версією специфікації архітектури SOAP.

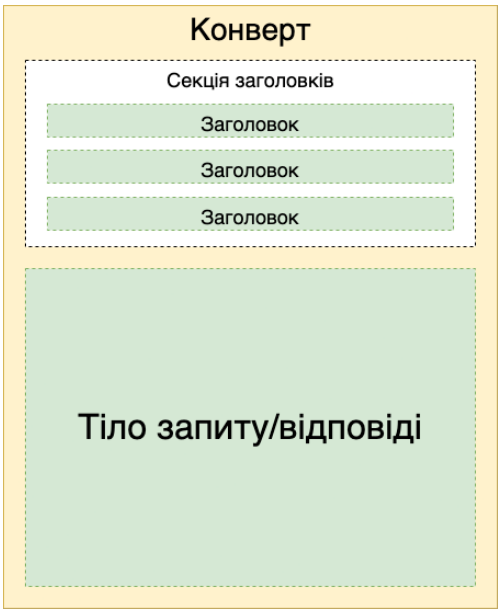


Рис. 3.1. Структура повідомлення в архітектурі SOAP

Ця архітектура є клієнт-серверною та не прив’язана до якогось конкретного протоколу передачі даних. XML використовується як формат передачі даних. Кожне повідомлення, котре в термінах SOAP називається конвертом, завжди містить дві частини: заголовки, котрі містять в собі мета-дані, пов’язані з

повідомленням та тіло, де зберігається основна інформація про запит чи відповідь, як показано на рисунку 3.1 [23].

До переваг SOAP можна віднести його незалежність від протоколу передачі даних, використання XML для передачі даних, і те, що вона добре вписується в модель HTTP, що дозволяє запроваджувати підтримку SOAP без значних змін до клієнтів та проміжних вузлів між клієнтом і сервером (наприклад, файрволів).

Транспорт-нейтральність SOAP робить його придатним для використання з будь-яким транспортним протоколом. Реалізації часто використовують HTTP як транспортний протокол, але можна використовувати й інші популярні транспортні протоколи. Наприклад, SOAP також можна використовувати через SMTP, JMS та AMQP.

В той же час транспорт-нейтральність можна віднести й до мінусів, адже вона означає що SOAP не може користуватися особливостями певних протоколів, що могли б зробити її більш ефективною, а частину з цих особливостей доводиться реалізувати самотужки.

Так як архітектура SOAP використовує XML для обміну повідомленнями, вона може використовувати всі можливості формату XML, що повністю стандартизований та може бути розширений за допомогою використання різних просторів імен (XML Namespaces).

З іншою сторони, при використанні HTTP як транспорту для SOAP, через особливості протоколу ролі сторін «фіксуються» і сервер може тільки відповідати на запити клієнту й не може сам ініціювати запити.

В абревіатурі SOAP є слово «simple», простою цю архітектуру назвати ніяк не можна. Багатослівність протоколу, повільна швидкість синтаксичного аналізу XML і відсутність стандартизованої моделі взаємодії призвели до домінування служб, які використовують протокол HTTP безпосередньо. Наприклад, REST.

GraphQL є наймолодшою архітектурою з згаданих. Вона була розроблена в Facebook в 2012-му та опублікована в 2015-му. А в 2018-му права на GraphQL були передані незалежній організації GraphQL Foundation.

При розробці REST API часто виникає одна й та ж проблема, особливо на пізніх стадіях розвитку продукту: ресурси розростаються настільки, що їх передача по мережі починає відбирати занадто багато ресурсів. При цьому зазвичай клієнту не потрібно отримувати всі дані ресурсу, йому може бути достатньо всього декількох полів.

Цю проблему можна вирішити створивши декілька кінцевих точок, що повертають різні версії ресурсу, наприклад, мінімальну, звичайну та розширену з відповідним набором полів. Проте, таке рішення буде ad hoc, придатним для якогось поодинокого ресурсу.

Адже з розширенням кількості видів ресурсів та кількості клієнтів, кожен з яких може потребувати різного набору полів, такий підхід потребуватиме все більше часу і може нівелювати всі початкові переваги.

GraphQL розроблялася як рішення цієї проблеми. Особливістю GraphQL є те, що клієнт може описувати котрі поля ресурсу йому потрібні, що дозволяє зекономити на ресурсах мережі, якщо клієнту потрібні лише декілька полів з дуже об’ємного ресурсу.

Для використання GraphQL розробники створюють схему для опису всіх можливих даних, які клієнти можуть запитувати через цю службу. Схема GraphQL складається з типів об’єктів, які визначають, який тип об’єкта ви можете запитати та які поля він має. Коли сервер отримує запит, GraphQL перевіряє запит на відповідність схемі і виконує його.

Розробник API зв’язує кожне поле в схемі з функцією, яка під час виконання викликається для отримання значення, котре буде повернуто клієнту. Таким чином GraphQL може повертати дані в різних форматах, наприклад JSON чи XML, а також серіалізувати дані, котрі не можна передати через згадані формати напряму, наприклад DateTime.

GraphQL має й свої недоліки. З використанням GraphQL стає набагато складніше налаштувати кешування, особливо у порівнянні з REST. У REST API ми отримуємо доступ до ресурсів за допомогою URL-адрес, тому можемо кешувати дані на рівні ресурсу, оскільки у нас є URL-адреса ресурсу як ідентифікатор. З

іншого боку, у GraphQL це дуже складно, оскільки кожен запит може відрізнятися, навіть якщо він працює з одним і тим же об’єктом.

3.3 Клієнт-серверна архітектура

У комп’ютерних технологіях термін «клієнт» означає програмне або апаратне забезпечення, яке взаємодіє з сервером з метою отримання даних або виконання певних дій.

Клієнт є важливою частиною архітектури клієнт-сервер. Прикладом клієнтів є веб-браузери. Вони виконують роль веб-клієнтів і надсилають запити на веб-сервер, отримуючи по черзі необхідну веб-сторінку.

ІТ-відділам великих компаній іноді доводиться керувати тисячами пристроїв співробітників. Зазвичай вони роблять це віддалено через корпоративні мережі, підключені до комп’ютера кожного користувача. Терміни «тонкий клієнт» і «товстий клієнт» стосуються цих окремих комп’ютерів.

Обговорення тонкого та товстого клієнта має найбільший сенс у контексті так званої інфраструктури віртуального робочого столу (VDI).

VDI — це мережеве налаштування, де сервер у центрі обробки даних запускає віртуальний «стільний комп’ютер» від імені кожного користувача, використовуючи сховище, операційну систему та ресурси сервера.

У свою чергу окремі користувачі підключаються до своїх віртуальних робочих столів, використовуючи товстий або тонкий клієнт як кінцеву точку. VDI часто використовується великими корпораціями для оптимізації керування віддаленими пристроями та масштабування ІТ-операцій.

Клієнт-серверна архітектура буває двох типів: товста і тонка. Також існують архітектури, які поєднують можливості товстих і тонких клієнтів – гібриди. На рис.3.2 наведено клієнт-серверна архітектура з тонким клієнтом.

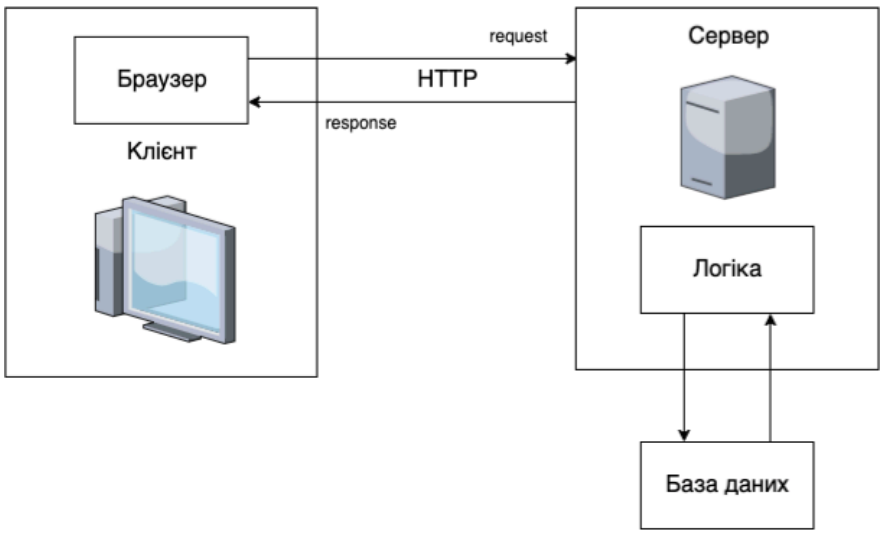


Рис. 3.2 Клієнт-серверна архітектура з тонким клієнтом

Наведемо основні відмінності між тонкими та товстими клієнтами:

1. Вартість

Тонкі клієнти доступніші. Вони покладаються на віддалені сервери для обробки, тому не потребують дорогого локального обладнання.

2. Споживана потужність

Оскільки вони запускають мало локальних програм (якщо такі є), тонкі клієнти споживають менше енергії. Більша компанія може значно зменшити споживання електроенергії та вплив на навколишнє середовище, перейшовши на налаштування на основі тонкого клієнта.

3. Місцеві ресурси

Тонкі клієнти мають обмежені локальні ресурси щодо зберігання даних, обробки та ОС. Вони залежать від мережі, яка їх надає.

4. Залежність від мережі

Через вищезазначене тонким клієнтам потрібне стабільне підключення до мережі. Без нього тонкі клієнти практично марні. Товсті клієнти, з іншого боку, цілком здатні працювати в автономному режимі, використовуючи власне апаратне та програмне забезпечення.

5. Настроюваність

Тонкими клієнтами зазвичай керують дистанційно з обмеженим введенням з боку кінцевого користувача. Товсті клієнти можуть бути налаштовані окремими співробітниками шляхом встановлення необхідного локального програмного забезпечення та програм.

6. IT-ресурси

Товсті клієнти призводять до більш фрагментованого IT-підходу для моніторингу, обслуговування та оновлення різного програмного забезпечення на комп'ютері кожного користувача.

З тонкими клієнтами все розгортається та управляється централізовано, завдяки чому IT-ресурси використовуються набагато ефективніше.

7. Мобільність

При налаштуванні тонкого клієнта «робочий стіл» користувача здебільшого існує в хмарі. Таким чином, він забезпечує кращу мобільність, оскільки користувач може успішно отримати доступ до свого віртуального робочого столу з будь-якої кінцевої точки.

8. Безпека

Товсті клієнти більш схильні до проблем з безпекою, оскільки вони мало контролюють те, що користувачі завантажують і встановлюють на своїх локальних машинах. З тонкими клієнтами ця загроза зведена до мінімуму, оскільки IT-спеціалісти мають більше контролю над розгортанням програмного забезпечення та моніторингом кіберзагроз.

Тонкі клієнти постачаються з чіткими індивідуальними обмеженнями щодо пристроїв порівняно з товстими клієнтами. Тож тонким клієнтам часто віддають перевагу перед товстими клієнтами, коли мова йде про корпоративну IT-інфраструктуру. з точки зору компанії, тонкі клієнти пропонують ряд переваг:

1. Краща масштабованість

Налаштування тонкого клієнта дає IT-менеджерам набагато кращий контроль над встановленням програмного забезпечення, оновленнями та керуванням пристроями. Усе можна обробляти з центрального сервера та надсилати всім користувачам через масове розгортання.

2. Покращена безпека

Завдяки централізованому керуванню програмним забезпеченням і додатками архітектура тонкого клієнта знижує ризики шкідливого програмного забезпечення та інших загроз безпеці з окремих кінцевих пристроїв.

3. Зменшені витрати

Оснащення тисяч співробітників дешевшими тонкими клієнтськими пристроями може значно вплинути на зниження загальних витрат. Подібним чином менше споживання енергії кожним тонким клієнтом призводить до зменшення витрат і зменшення викидів вуглецю для компанії в цілому. На рис.3.3 наведено клієнт-серверна архітектура з товстим клієнтом.

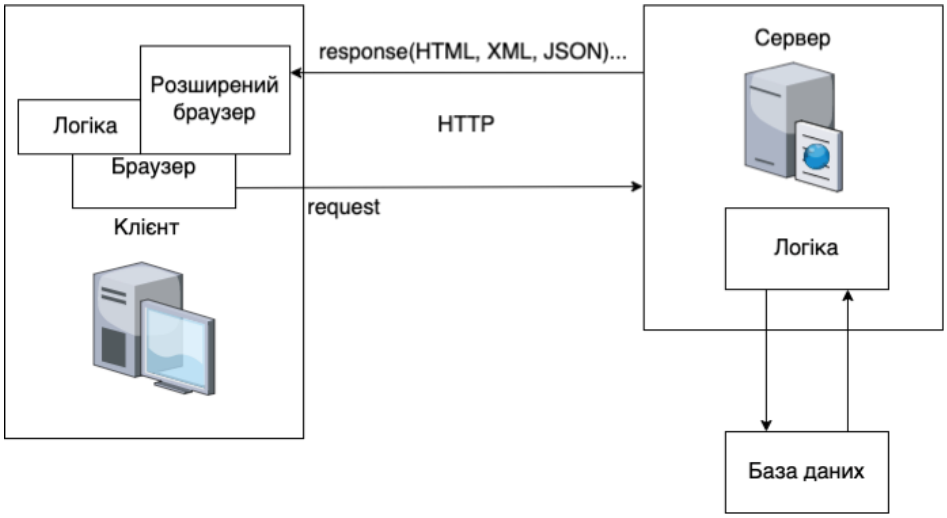


Рис. 3.2 Клієнт-серверна архітектура з товстим клієнтом

Товстий клієнт - це клієнт, який виконує запитувані операції самостійно, так що без допомоги центрального сервера. Центральний сервер у цьому типі архітектури може використовуватися як сховище даних, обробка та відображення яких передається на робочу машину клієнта.

Товстий клієнт - це робоча станція або ПК, який працює під керуванням власної ОС і має повну панель інструментів програмного забезпечення для забезпечення виконання завдань користувачем.

Товстий клієнт виконує основну частину обробки в програмах клієнт/сервер. З товстими клієнтами немає потреби в безперервному обміні даними з сервером, оскільки він переважно передає серверу інформацію про архівне зберігання.

Товсті клієнти набагато більш уразливі до кібератак і порушень безпеки. Це тому, що, по-перше, програми та дані зберігаються на самій машині. А по-друге, IT-команди мають обмежене уявлення про те, що завантажується на ці машини, тобто користувачі можуть легко завантажити шкідливе програмне забезпечення, яке залишиться непоміченим. Товсті клієнти також дорожчі в експлуатації та обслуговуванні, ніж тонкі клієнти.

3.4. Сервіс-орієнтована архітектура

На сьогоднішньому висококонкурентному глобальному ринку підприємства працюють у дуже ліквідному середовищі, де інформація є найбільш стратегічним активом. Швидке реагування на зміни конкуренції, ринкової динаміки та регулятивних повноважень із своєчасним отриманням інформації має вирішальне значення для ефективного функціонування та загального успіху бізнесу. Щоб задовольнити вимоги ринку, що швидко змінюються, підприємства все більше орієнтуються на послуги як у способах взаємодії з клієнтами та партнерами, так і в тому, як вони проектують і створюють свою IT-інфраструктуру.

Оскільки компанії прагнуть забезпечити рентабельність інвестицій за рахунок підвищення гнучкості та оперативності, вони залежать від IT-груп підприємства, щоб знайти нові та економічно ефективні засоби надання нових послуг і сприяти вільному потоку інформації та бізнес-процесів в організації. Наведемо рушії бізнесу, які зробили сервіс-орієнтовану IT- архітектуру економічною реальністю сучасного підприємства:

Впровадження веб-сервісів у галузь для швидкого відкриття та активації нових і застарілих сервісів.

Необхідність створення системно-орієнтованих процесів, що охоплюють програми та користувачів.

Необхідність швидко виставляти процеси як послуги.

Виконання критично важливих процесів безпечно та послідовно, дотримуючись цілісності транзакцій.

Розробка інтегрованих додатків і управління процесами.

Забезпечення високопродуктивного виконання для прямої обробки.

Сервісно-орієнтована архітектура (SOA) стала провідною IT-програмою для реформування інфраструктури, щоб оптимізувати надання послуг і забезпечити ефективне управління бізнес-процесами. Частиною зміни парадигми SOA є фундаментальні зміни в розробці IT-інфраструктури — відхід від інфраструктури додатків до інфраструктури конвергентних послуг.

Сервісно-орієнтована архітектура дозволяє організовувати окремі функції, що містяться в корпоративних програмах, як рівні сумісних спільних «сервісів», заснованих на стандартах, які можна комбінувати та повторно використовувати в складених програмах і процесах.

Крім того, цей архітектурний підхід також дозволяє інтегрувати послуги, що пропонуються зовнішніми постачальниками послуг, в IT-архітектуру підприємства. У результаті підприємства можуть розблокувати ключову бізнес-інформацію в розрізнених бункерах економічно ефективним способом.

Організовуючи IT підприємства навколо служб, а не навколо додатків, SOA допомагає компаніям скоротити час надання послуг і більш гнучко реагувати на швидкі зміни у бізнес-вимогах.

Останніми роками багато підприємств перейшли від пілотних проектів із використанням спеціального впровадження SOA та розширилися до визначеного повторюваного підходу для оптимізованого розгортання SOA на рівні підприємства. Усі рівні архітектури IT SOA стали доступними для надання послуг і включають служби презентацій, бізнес-процеси, бізнес-послуги, служби даних і спільні служби.

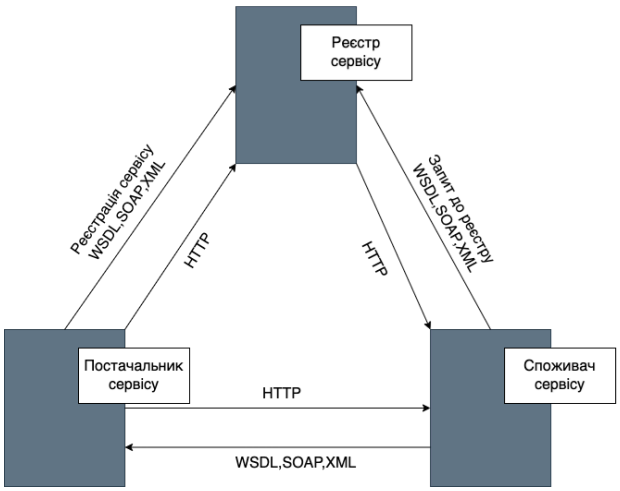


Рис. 3.3 Сервіс-орієнтована архітектура

Сервісно-орієнтована архітектура (SOA) — це стиль проектування програмного забезпечення, де послуги надаються іншим компонентам компонентами програми та обмінюються даними через мережу. Сервісно-орієнтована архітектура (SOA) виникла для оптимізації надання послуг і забезпечення ефективного управління бізнес-процесами.

Принципи SOA не залежать від постачальників та інших технологій. У сервісно-орієнтованій архітектурі ряд служб спілкуються одна з одною через передачу даних або через дві чи більше служб, які координуються одна з одною. Сервісно-орієнтована архітектура дозволяє розподіляти функції, які містяться в корпоративних програмах, щоб бути більш організованими за допомогою рівнів.

Порівняґмо SOA та мікросервіси. SOA та архітектура мікросервісів — це архітектурні стилі, які структурують систему як набір сервісів. Але стилі спілкування різні. Обидва використовують зв’язок між службами завдяки розподіленим службам. Але в SOA, з використанням Enterprise Service Bus, з використанням важких протоколів, таких як протоколи SOAP, WSDL і XSD. Але в мікросервісах, використовуючи брокери повідомлень, або прямий зв’язок між послугами, використовуючи легкі протоколи, такі як REST або gRPC.

Крім того, моделі даних відрізняються, SOA використовує глобальну модель даних і спільні бази даних, але Microservices має поліглотні бази даних із базою даних для шаблону служби.

Ще однією ключовою відмінністю між SOA та архітектурою мікросервісів є розмір сервісів. SOA зазвичай використовується для інтеграції великих, складних, монолітних програм. Хоча сервіси в мікросервісній архітектурі майже завжди набагато менші.

У результаті додаток SOA зазвичай складається з кількох великих сервісів, тоді як додаток на основі мікросервісів зазвичай складається з десятків чи сотень менших сервісів.

Наведемо недоліки SOA. Незважаючи на обіцянки SOA, впровадження цього підходу часто ускладнювало процес і створювало вузькі місця. Витрати на обслуговування стали високими, а проміжне програмне забезпечення ESB – дорогим. Послуги, як правило, були великими. Вони часто ділилися залежностями та сховищем даних. Зрештою, SOA часто призводили до «розподіленої монолітної» структури з централізованими службами, які були стійкі до змін.

Тому ми повинні розвинути нашу архітектуру до архітектури мікросервісів, щоб адаптувати бізнес-адаптацію швидше до виходу на ринок і обробляти великі запити.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ УДОСКОНАЛЕНОЇ АРХИТЕКТУРИ СИСТЕМИ

4.1 Загальний опис запропонованої архітектури

Для побудови сервісу агрегатора контенту було обрано мікросервісну архітектуру, що дозволяє масштабувати окремі компоненти в залежності від навантаження. Кінцевий користувач взаємодіє з додатком через веб-додаток з архітектурою SPA (односторінковий додаток), котрий в свою чергу взаємодіє з бекенд-серверами з використанням REST API. Статичні ресурси (іконки, стилі, html) зберігаються на CDN-сервері, що дозволяє скоротити затримки і досягти максимальної швидкості віддачі статичного контенту.

Бекенд-сервери знаходяться за розподілювачем навантаження, що дозволяє оптимізувати завантаженість кожного сервера. Сервера також географічно розподілені щоб мінімізувати затримки при передачі даних. Для кожного бекенд-сервера, в тому самому датацентрі, буде розташований сервер з базою даних для оптимізації часу на читання даних. Ця база даних буде лише для читання (read-only replica), вона автоматично синхронізується з головною базою даних (master node).

Запис нових даних проходить напряму в головну БД. Базою даних було обрано PostreSQL оскільки вона є достатньо продуктивною, підтримує шардинг і реплікацію і дозволяє зберігати не-структуровані дані, що може стати в нагоді при подальшій розробці.

На окремому сервері працює планувальник, головною задачою котрого є періодична перевірка які стрічки потребують оновлення даних і публікація повідомлень про них в чергу повідомлень. Для планувальника повідомлень було обрано cron, для системи обміну повідомленнями було обрано RabbitMQ через його можливості реплікації та ефективне використання ресурсів. Всі нові стрічки додані в БД відразу додаються в систему обміну повідомленнями, не чекаючи планувальника.

Повідомлення з RabbitMQ будуть спожиті серверами що займаються збором даних з стрічок. Така архітектура дозволяє горизонтально масштабувати кількість серверів що займаються збором даних в залежності від потреби системи.

Код сервісу що буде виконуватися на цих серверах структурований таким чином, що всі імплементації збору даних інкапсульовані за уніфікованим інтерфейсом, таким чином основний код взаємодіє лише з роутером, частиною системи що приховує та ізолює складність системи від зовнішніх компонентів. Роутер, в свою чергу, завдяки даним, котрі він отримав від основного сервісу, обирає яку імплементацію викликати і за потреби адаптує вхідні дані під потреби конкретної імплементації і нормалізує вихідні дані.

Таким чином ми запобігаємо винекненню зв’язків багато-до-багато (many-to-many) серед компонентів системи, що викликало б збільшення складності (complexity) в горизонтальній прогресії при доданні кожного нового типу стрічки. Натомість, ізолюючи імплементації збірщиків даних від решти системи ми отримуємо можливість горизонтально масштабувати і кількість типів стрічок з котрими може працювати система.

Також весь зібраний контент буде класифіковано з використанням нейромережі. Це дозволить в майбутньому розбивати контент за темами і зберігати контент схожих тематик поряд. Така дефрагментація дозволить оптимізувати час на читання і побудову уніфікованої стрічки для кінцевого користувача на основі його інтересів.

Сьогодні своєчасне адаптування існуючого програмно-апаратного забезпечення до нових вимог функціонування бізнесу складає проблему. Звісно, що такі велетенські компанії як Google, Netflix та інші можуть та мають слідкувати й адаптуватися під нескінченно підростаючий клієнто-потік, що несе за собою високе системне навантаження та потреби швидко масштабуватись.

Але, якщо ви працюєте на невеликому підприємстві з відчутно обмеженим бюджетом та невеликою кількістю клієнтів, які відвідують ваш сайт, то можливо зовсім не потрібно вибудовувати швидко масштабовану і велику відмовостійку систему, а краще зважити ризики і оцінити потенційні витрати на підтримку та

експлуатацію великої системи та можливі витрати однієї години простою на місяць/тиждень. Існує величезна вірогідність, що для бізнесу буде дешевше “полежати” одну годину, так як, наприклад, вони зосереджені на офлайн бізнесі.

Розглянємо проблему пошуку можливостей швидкого переходу від монолітної архітектури до мікросервісної архітектури побудови комп’ютерних систем за допомогою технології контейнеризації Docker та оркестровці контейнерів Kubernetes (K8s) для оцінки ефективності їх впровадження залежно від потреб бізнесу. Розгляд даної теми досить актуальний, тому що в даний час функціонування бізнесу залежить від того, наскільки існуючі програмно-апаратні засоби системи автоматизації управління бізнесом зможуть швидко адаптуватися до нових викликів або загроз у нових умовах функціонування бізнесу [18].

Система управління бізнесом складається з різних складових, серед яких програмноапаратне забезпечення, яке визначає ступінь автоматизації та інтелектуалізації процесів управління бізнесом, а також відіграє провідну роль у функціонуванні бізнесу. Ця залежність вказує на критичність (уразливість) системи управління бізнесом під час появи нових вимог функціонування.

Суть цієї проблеми полягає в наступному: відомо, що в загальному плані своєчасність адаптування програмно-апаратного забезпечення бізнесу залежить в першу чергу від архітектури побудови комп’ютерної системи (далі архітектури системи). Архітектурою комп’ютера називається його опис на деякому загальному рівні, що включає опис призначених для користувача можливостей програмування, системи команд, системи адресації, організації пам’яті тощо.

Архітектура визначає принципи дії, інформаційні зв’язки і взаємне з’єднання основних логічних вузлів комп’ютера: процесора, оперативної і зовнішньої пам’яті та периферійних пристроїв. Спільність архітектури різних комп’ютерів забезпечує їхню сумісність з точки зору користувача.

Тому, якщо архітектурою системи вважати набір типів даних, операцій та характеристик кожного окремо взятого рівня (певних структурних компонентів пов’язаних між собою, які задають поведінку всієї системи), то від того як швидко може змінюватися архітектура системи буде залежати час адаптації. Зрозуміло,

адаптація архітектури системи, як процес пристосування до мінливих умов зовнішнього середовища, вимагає нових підходів до принципів побудови архітектури комп'ютерних систем.

Дійсно, історично довгий час у комп'ютерних системах провідне місце займала так звана "монолітна архітектура", в якій, з одного боку, програмні аспекти (наприклад, введення і виведення даних, обробка даних, обробка помилок та інтерфейс користувача) виконували не архітектурно окремі компоненти, а компоненти, що пов'язані між собою.

З іншого боку, застосування багатоядерних процесорів у "монолітній архітектурі" передбачає, що компоненти інтегровані разом до єдиної інтегральної схеми. При даному підході вся система являє собою моноліт, який фізично розташовується на єдиній машині, запускається в одному процесі та виконує по черзі всі бізнес-операції системи управління бізнесом, тобто має зниження продуктивності внаслідок поганого механізму для блокування записів і обробки файлів по локальній мережі.

Тому монолітні архітектури побудови комп'ютерних систем погано працюють з декількома користувачами. Звідси стає зрозуміло, що "монолітна архітектура" погано адаптується до нових вимог щодо розподілення завдань між багатьма користувачами системи управління бізнесом. Тому процес адаптації вимагає пошуку нових підходів до побудови архітектури комп'ютерної системи.

Відомі такі види побудови архітектури комп'ютерної системи: класична архітектура Фон Неймана, монолітна, сервісорієнтована та мікросервісна архітектури. Класична архітектура (фон Нейман) – має пристрій керування, арифметично-логічний пристрій, пам'ять, пристрої вводу-виводу інформації, об'єднані за допомогою каналів зв'язку.

В монолітній архітектурі функціонально помітні аспекти (наприклад, введення і виведення даних, обробка даних, обробка помилок та інтерфейс користувача), не архітектурно окремі компоненти, а компоненти, що пов'язані між собою. Монолітна архітектура може бути централізованою (це класична

архітектура) або розподілена. Розподілена монолітна архітектура – це набір служб, які необхідно розгортати одночасно, щоб створити єдину функцію.

Розподілена монолітна система часто набагато гірша, ніж централізована монолітна система, через складність і вартість координації декількох послуг. Сервісно-орієнтована архітектура (Service-oriented architecture, SOA) – це архітектурний шаблон програмного забезпечення, модульний підхід до розробки програмного забезпечення, заснований на використанні розподілених, слабко пов’язаних замінних компонентів, оснащених стандартизованими інтерфейсами для взаємодії за стандартизованими протоколами.

Архітектурний стиль мікросервісів – це підхід, коли єдиний додаток будується як сукупність невеликих, самодостатніх, незалежних, не тісно пов’язаних сервісів, що спілкуються між собою за допомогою легких механізмів як то HTTP, gRPC, AMQP [19–21].

Необхідність розподілення завдань між багатьма користувачами системи управління бізнесом створило умови появи ідеї побудови SOA. Зміст SOA у тому, що програмне забезпечення складається з набору незалежних сервісів, які фокусуються на власній задачі та може являти собою розподілену систему, в якій відбувається обмін повідомленнями за певним протоколом, що дозволяє працювати будь-яким додаткам на будь-якому апаратному забезпеченні.

Але недоліком SOA є обмеження протоколів обміну даними та розділу системи. Для усунення цього недоліку був запропонований підхід удосконалення SOA, а саме: мікросервісна архітектура (MSA – Measurement System Analysis).

Удосконалення полягає у тому, що за допомогою переходу до мікросервісів бізнес отримує можливість швидко щось змінювати, щоб швидше адаптуватися до змін бізнес-вимог. Метою мікросервісної архітектури є допомога розробникам швидше, безпечніше та якісніше поставляти продукти.

Відокремлені сервіси дозволяють виконувати ітерації швидко і з мінімальним впливом на іншу частину системи. Тобто мікросервіси, які не тісно пов’язані між собою, дають можливість проводити зміни в окремих програмно-апаратних

засобах системи автоматизації управління бізнесом з більшою частотою ітерацій, мінімізуючи вплив змін на решту частин системи.

Звідси головною відмінністю MSA є горизонтальне масштабування – це можливість вирішувати розподілення певного набору задач в умовах різних обставин між множиною комп'ютерних систем, що об'єднані в розподілену мережу. Тому виникає актуальне та своєчасне завдання дослідження проблеми адаптування системи управління за допомогою побудови додатків на базі мікросервісів.

Одним з перспективних напрямків дослідження є дослідження методів спрощення розгортання мікросервісних додатків за допомогою системи керування контейнерами K8S та порівняння цих готових повністю керованих додатків у хмарі (Full managed version in the cloud). Тому метою є дослідження побудови додатків на базі мікросервісів, огляд існуючих платформ для створення мікросервісів та їх порівняння.

Дослідження побудови додатків на базі MSA передбачає знаходження обмежень та доцільності використання цієї архітектури. Відомо, що MSA – це варіант SOA програмного забезпечення, орієнтований на взаємодію наскільки це можливо невеликих, слабо пов'язаних і легко змінюваних модулів – мікросервісів, що набув поширення в середині 2010-х років у зв'язку з розвитком практик гнучкої розробки та технології DevOps.

Технологія DevOps (Development і operations) призначена для ефективної організації процесів створення та оновлення програмних продуктів і послуг. Вона заснована на ідеї тісного взаємозв'язку розробки та експлуатації програмного забезпечення.

Ця методологія активної взаємодії програмістів-розробників з фахівцями інформаційно-технологічного обслуговування надає можливість взаємної інтеграції їхніх робочих процесів один в одного для забезпечення побудови якісного продукту.

Раніше було показано, що монолітна архітектура передбачає побудову системи як єдине ціле. Недоліком монолітної архітектури є те, що будь які зміни,

навіть самі невеликі, потребують перебудови та повторного розгортання всього додатку для адаптації до нових умов функціонування бізнесу. Якщо такі умови змінюються досить часто, то з часом стає складніше зберігати хорошу модульну структуру, тому що зміни логіки одного модуля мають тенденцію впливати на код інших модулів.

Крім того, монолітна архітектура не дозволяє масштабувати програми, тому що різні модулі мають конфліктні вимоги до ресурсів. Наприклад, один модуль може реалізовувати логіку обробки зображень з інтенсивним використанням процесору, а інший модуль може бути вимогливим до використання оперативної пам’яті.

Однак, оскільки ці модулі розгортаються разом, то доведеться йти на компроміс з вибором апаратного забезпечення. Крім того, якщо потрібно масштабувати тільки один модуль, то разом з тим доводиться примусово масштабувати додаток для другого модуля, навіть якщо це не потрібно.

Для усунення цих незручностей мікросервісна архітектура дозволяє кожен мікросервіс розгортати окремо в наслідок того, що в архітектурі мікросервіса кілька слабко пов’язаних служб працюють разом. Кожен сервіс орієнтований на одну мету і має високий ступінь узгодженості поведінки й даних.

Тому, якщо треба змінювати щось в одному з них, то можна розгорнути ці зміни не чіпаючи інших мікросервісів, які можуть продовжувати працювати. Звідси розосереджені завдання веб-проектів, які частіше розробляються як окремі сервіси, котрі можна розгортати та масштабувати окремо [22].

Важливо розуміти, що мікросервіс це: це не сервіс, який має невелику кількість рядків коду або виконує “мікро” завдання. Це помилка, яка походить від назви “мікросервіс” [23].

Це стає можливо внаслідок реалізації принципів моделювання (проектування) мікросервісів у мікросервісній архітектурі. На рис. 4.1наведено Три принципи моделювання (проектування) мікросервісів

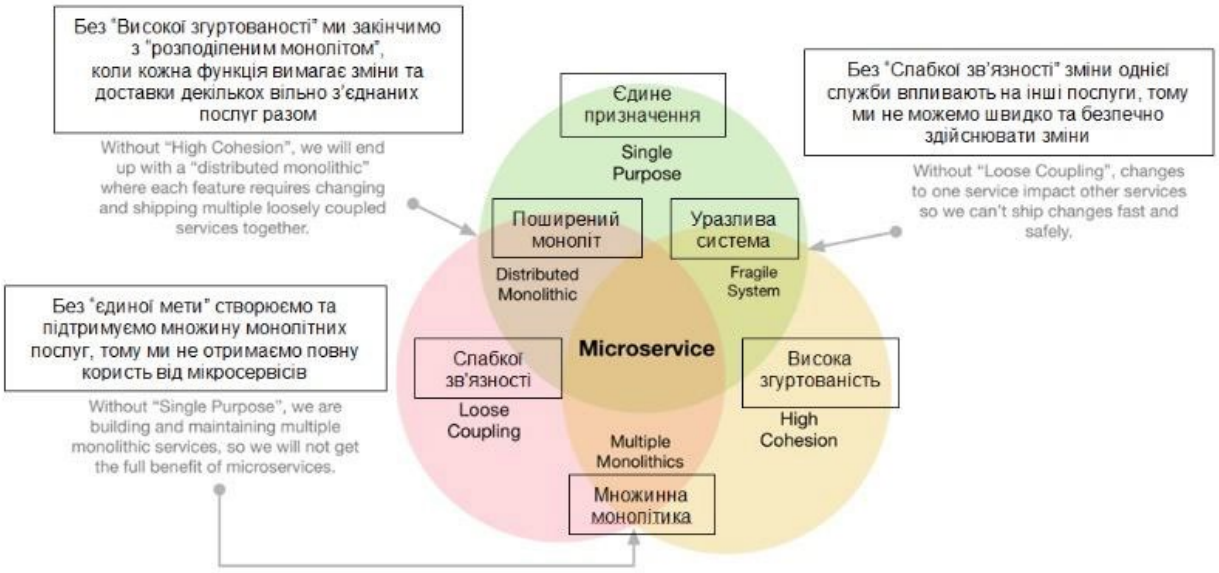


Рис.4.1. Три принципи моделювання (проектування) мікросервісів [23]

Єдина мета – кожен сервіс має зосередитися на одній єдиній меті та робити це добре:

1. Слабкий зв'язок означає, що:

a. Служби мало знають одна про одну;

b. Зміна однієї послуги не повинна вимагати заміни іншої;

c. Зв'язок між сервісами повинен відбуватися тільки через загальнодоступні сервісні інтерфейси;

2. Висока згуртованість – кожен сервіс об'єднує всі пов'язані поведінки і дані разом. Якщо нам потрібно створити нову функцію, всі зміни повинні бути локалізовані тільки для одного сервісу.

Мета мікросервісної архітектури – не створювати якомога більше дрібних сервісів. Послуги можуть бути складними і суттєвими, якщо вони відповідають вищевказаним трьом принципам; це не послуга, яка постійно створюється з використанням нових технологій.

Незважаючи на те, що архітектура мікросервіса дозволяє легше тестувати нові технології, це не є основною метою архітектури мікросервіса. Абсолютно нормально створювати нові сервіси з точно таким технологічним стеком, якщо під

час проектування отримується вигода з роз'єднаних сервісів; це не сервіс, який повинен бути побудований з нуля.

Якщо у вас уже є добре спроектований монолітний додаток, то не потрібно створювати кожен новий сервіс з нуля. Можуть бути можливості витягти логіку безпосередньо з монолітного сервісу.

Таким чином, мікросервісну архітектуру додатків раціонально застосовувати коли інші види архітектури стають вузьким місцем: відносно продуктивності процесів; уповільнення розробки нових продукто-послуг; монолітний додаток ускладнює масштабування системи для конкретних завдань або ізоляцію проблем ресурсів для різних типів завдань; заважає випробувати нові технології (однією з основних переваг мікросервісної архітектури є те, що кожен сервіс може бути побудований з використанням різних технологічних стеків та інтегрований з різними технологіями); моноліт – не дозволяє вибрати кращий інструмент для роботи і, що більш важливо, зробити це швидким і безпечним способом.

Сьогодні багато відомих компаній вирішують проблему масштабування за допомогою MSA. Для прикладу застосування технології MSA – стисло розглянемо технологію Netflix

Компанія Netflix надає послуги потокового відео через Інтернет. Відомо, що якість передачі потокового відео залежить від пропускної здатності каналів мережі MPLS/IP. Ці канали створюють "тунель" між користувачем та сервером. Чим більше таких каналів складаються в "тунель", тим більша ймовірність перевищення 5% вимоги втрати пакетів у каналі потокового відео внаслідок помилок передачі, а це перевищення призводить до зупинки показу відеоконтенту у користувача [24].

Тобто, зрозуміло, що необхідно скорочувати ці "тунелі". Цю проблему компанія Netflix вирішила шляхом розміщення відеоконтенту на серверах компанії Amazon, які розосереджені по всьому світу. Це дозволило суттєво скоротити відстань між сервером та користувачем.

Для користування цим сервісом було створено програмне забезпечення, яке вирішувало з одного боку завдання масштабування управління доступом до відео контенту Netflix на найближчому сервері компанії Amazon, а з іншого передачу

дозволу для зареєстрованих користувачів на сервері компанії Netflix, що знаходиться в Лос-Гатос, Каліфорнія.

Тобто з технічної точки зору компанія Netflix створила бібліотеку відео додатків, які були розосереджені. Бібліотека Netflix доступна для перегляду з більшості сучасних інтернет-браузерів, також існують додатки для мобільних платформ Android, iOS і Windows Phone; приставок Google Chromecast й Apple TV, ігрових консолей Nintendo, PlayStation та Xbox; а також для багатьох моделей телевізорів з функцією Smart TV.

Складність програмного забезпечення Netflix MSA Platform вражає. Воно дозволяє: створювати потрібну конфігурацію серверів; виконувати реєстрацію сервісу та його відкриття; виконувати створення каналів з потрібною якістю передачі потокового відео в мережі MPLS; керувати шлюзами API; виконувати балансування завантаження за допомогою взаємодії між процесами (Inter-process communication, IPC); здійснювати моніторинг послуг у режимі реального часу; забезпечувати фіксацію простою та постачання нуля на час простою; виконувати тестування несправності, застосовувати методи інженерії хаосу для тестування програмного коду тощо.

Впровадження технології Netflix MSA Platform є успішним для розробки нових програмних засобів створення мікросервісів. Сьогодні Netflix можна вважати першопрохідцем розвитку мікросервісів, і їхній підхід став об’єктом дослідження для багатьох інших компаній у всьому світі. Серед них Amazon, eBay, Walmart, Twitter, Spotify, Uber та Medium, Sound Cloud, Stripe, PayPal.

Таким чином, MSA – це архітектурний стиль, за яким єдиний додаток будується як сукупність невеличких сервісів, кожен з яких працює у своєму власному процесі та обмінюється даними з іншими.

Ці сервіси будуються навколо бізнес-потреб і розгортаються незалежно з використанням зазвичай повністю автоматизованого середовища. Сервіси самі по собі можуть бути написані з використанням різних мов й технологій зберігання даних.

Це створює наступні можливості:

1) відсутність обмежень на кількість існуючих сервісів, але кожен сервіс має працювати лише з однією бізнес-задачею;

2) використання стандартизованих протоколів передачі даних (наприклад, HTTP) для обміну інформацією між мікросервісами;

3) для спілкування з іншими мікросервісами кожен сервіс має своє API;

4) всі сервіси можуть бути написані на абсолютно різних мовах програмування та використовувати будь-які бібліотеки;

5) збереження даних децентралізоване, тобто кожен сервіс має свою власну базу даних.

На основі вказаних можливостей можна виділити основні переваги використання MSA:

1) існує досить мала зв’язність між основними компонентами системи та високе зчеплення коду в окремо взятому сервісі;

2) кожен сервіс розгортається незалежно від інших, що надає можливість динамічно вносити зміни та запускати окремо процеси;

3) можна запускати будьяку кількість сервісів на окремих серверах (масштабування системи);

4) має відмовостійкість, тому що при виведенні з ладу одного сервісу інші можуть вірно працювати;

5) можна застосовувати різні мови програмування та технології;

6) розгалуження групи розробників для створення мікросервісів, які є відповідальними за власний сервіс.

До мінусів MSA можна віднести:

1) відносна складність розробки, прямо пропорційно залежить від кількості обраних мов програмування та фреймворків;

2) витрачаються додаткові ресурси на пересилання повідомлень між сервісами та на їхню серіалізацію та десеріалізацію;

3) існують проблеми з версіонуванням;

4) відносно складне інтеграційне тестування;

5) складна система моніторингу;

6) сильно розростається кількість можливих збоїв;

7) додає додаткові складності.

Пропозиції щодо впровадження мікросервісної архітектури Для вирішення питання переходу від монолітної архітектури до мікросервісної спочатку використовувались апаратні засоби віртуалізації такі як KVM, XEN, VMware ESXi тощо, але з часом з'явилися технології віртуалізації рівня операційної системи такі як Docker [25–26] та Kubernetes [27–30].

Відомо, що в апаратній віртуалізації базовий шар – гіпервізор. Цей шар завантажується на сервері та забезпечує взаємодію між апаратним забезпеченням сервера і віртуальними машинами. На відміну від програмної віртуалізації, при апаратній віртуалізації гостьові операційні системи управляються гіпервізором безпосередньо без участі хостової операційної системи.

Апаратна віртуалізація набагато ефективніше програмної, тому що дозволяє за допомогою гіпервізора створювати керовані ізольовані гостьові системи. Гостьова система не залежить від архітектури хостової платформи і реалізації платформи віртуалізації.

Технологія Docker відноситься до віртуалізації рівня операційної системи [25-26].

Технологія Docker дозволяє упакувати зліпок стану системи, виконати його розробку і запустити. З появою технології Docker зацікавленість контейнерами вибухово зросла, тому що розгортати додатки виявилося настільки зручно, що технологію почали використовувати буквально всюди. Розглянемо переваги та недоліки технології Docker.

Переваги технології Docker:

1. Застосування технології Docker дозволяє уникнути конфліктів під час адаптації програмного забезпечення між різними додатками.

Наприклад, є додаток на PHP (Personal Home Page Tools – "Інструменти для створення персональних веб-сторінок" – скриптової мови загального призначення, яка інтенсивно застосовується для розробки веб-додатків).

Додаток на PHP використовує бібліотеку Image Magick для роботи з зображеннями, також цьому додатку потрібні специфічні налаштування php.ini, а сам додаток хоститься за допомогою Apache httpd. Але є проблема: деякі регулярні рутинні задачі реалізовуються запуском Python-скриптів із cron, і бібліотека, яку використовують ці скрипти, конфліктує з версіями бібліотек, що використовуються у цьому додатку.

Технологія Docker дозволяє упакувати цей додаток разом із налаштуваннями, бібліотеками та HTTP-сервером в один контейнер, який обслуговує запити на 80-му порті, а рутинні задачі – перенести в інший контейнер. Все разом буде прекрасно працювати, і про конфлікт бібліотек можна буде забути.

2. Особливістю технології Docker для упаковки кожної програми є те, що типова композиція докерізованного додатку має шари ізоляції (рис. 4.2).

На рисунку 4.2 наведена типова композиція докерізованного додатку, розгорнутого в AWS. Прямокутниками тут позначені шари ізоляції. Найбільший прямокутник це фізична машина.

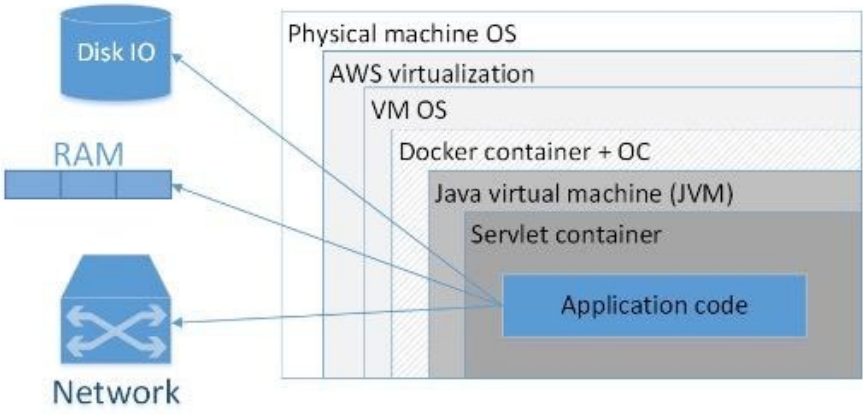


Рис. 4.2. Типова композиція докерізованного додатку [25]

Далі: операційна система (ОС) фізичної машини, амазонівский віртуалізатор, ОС віртуальної машини, докер-контейнер, ОС контейнера, JVM (Java Virtual Machine), Servletконтейнер (якщо це веб-додаток), всередині якого вже міститься код вашої програми.

Недоліки технології Docker:

1. JVM – це, віртуальна машина в Java, яка є завжди.

Додавання сюди ще додаткового Docker-контейнера, необхідне для упаковки кожної програми. Але, по-перше, часто не дає дуже помітної переваги, тому що JVM сама собою вже непогано ізолює від зовнішнього оточення, по-друге, не обходиться даром.

Мова йде про те, що якщо використовуються, наприклад, дискові операції (використання процесора або доступ пам'яті), то технологія Docker додає буквально частки відсотка додаткового часу під час накладних витрат на виконання функції Overhead, але якщо виконується передача по каналам, то виникає мережева затримка (network latency), яка є важливою характеристикою конструкції та продуктивності телекомунікаційної мережі, а ці затримки є цілком відчутними. Вони не гігантські, але в залежності від того, який у вас додаток, можуть зменшувати якість послуги (рис.4.3);

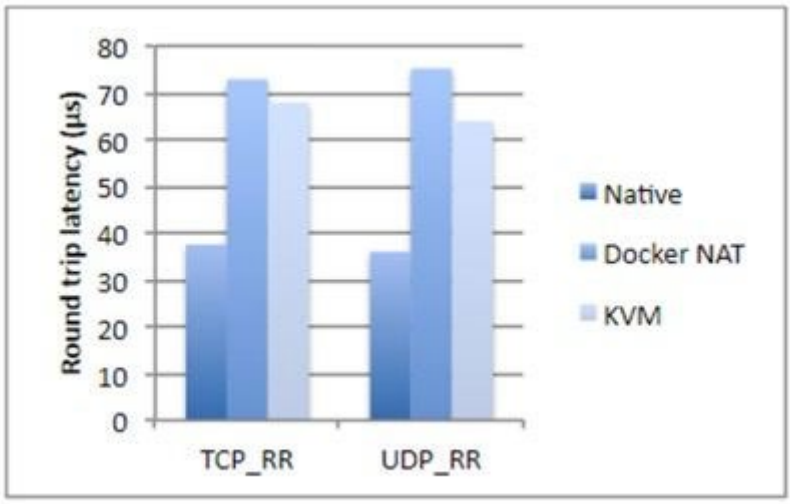


Рис. 4.3. Внесення затримок технологією Docker

2. Наступний недолік технології Docker – з'їдає додаткове місце на диску, займає частину пам'яті, додає час запуску додатку (start-up time);

3. Технологія Docker помітно ускладнює розбір проблем у мережевих протоколах, не лише ростуть затримки, але й змінюються всі таймінги.

Вказані недоліки впровадження технології Docker є некритичними для більшості систем. Але все ж виникають ситуації, коли пам’яті може не вистачати, і сумарний час старту системи, наприклад, що складається з декількох залежних сервісів, може стати досить великим.

У загальному випадку будь-який додаток, чутливий до затримок у мережі в межах до 250–500 мс, краще не докерізувати. Таким чином, у міру того як розробка додатків переміщується в сторону підходу на основі контейнера, стає важливо управляти ресурсами і контролювати їх.

Хмарні сервіси вже дають швидкий доступ до великої кількості віртуальних машин, якими потрібно управляти. Тому без інструменту, який дозволяє запускати контейнери на безлічі хостів, масштабувати і виконувати балансування, вже не обійтися. Таким інструментом є технологія Kubernetes.

Технологія Kubernetes відноситься до віртуалізації рівня операційної системи [27–30].

Розберемося з рішенням, запропонованим Google як технологія Kubernetes. Технологія Kubernetes зараз є провідною платформою надійного планування робочих навантажень для відмовостійких додатків.

Платформа Kubernetes, зараз дуже швидко розвивається. Вона призначена для управління контейнерними додатками Docker і пов’язаними з ними компонентами мережі та сховища.

Kubernetes – це кероване середовище, що спрощує розгортання додатків на основі контейнерів і управління ними. Основна увага в ній приділяється робочим навантаженням додатків, а не базових компонентів інфраструктури.

Переваги технології Kubernetes. З переваг даного підходу можна зазначити наступні:

1. Kubernetes як відкрита платформа дозволяє створювати додатки на будь-якій мові програмування, для будь-якої операційної системи, із застосуванням будь-яких бібліотек і служб повідомлень.

З платформи Kubernetes можна інтегрувати будь-які засоби безперервної інтеграції та безперервного постачання (CI/CD) для планування і розгортання релізів (випусків) продуктів.

Кластер Kubernetes розділяється на два компоненти (рис. 4):

1) головні вузли кластера надають базові служби Kubernetes і оркеструють робочі навантаження додатків;

2) вузли безпосередньо виконують робоче навантаження додатків;

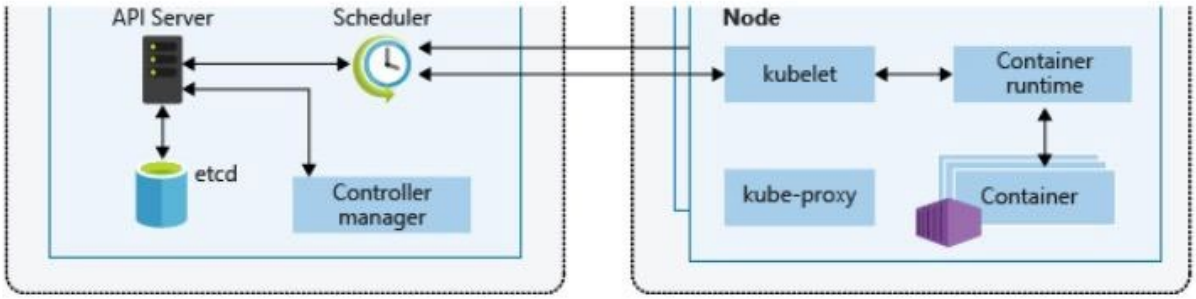


Рис. 4.4. Компоненти кластера Kubernetes [28]

2. У технології Kubernetes реалізується декларативний підхід до розгортання, підкріплений продуманим набором API-інтерфейсів для операцій управління.

Зміст декларативного підходу – це опис того, що потрібно досягти, а не як. Можна створювати і запускати версії сучасних додатків, що масштабуються на базі мікрослужб, при цьому можливості платформи Kubernetes по оркестрації і управлінню використовуються для організації доступності до компонентів програми, що підтримують бізнес-процес.

Технологія Kubernetes підтримує програми без відстеження стану і з відстеженням стану мікросервісної архітектури, що дуже зручно при адаптації додатків на базі мікрослужб;

3. Технологія Kubernetes сама себе відновлює в разі збоїв. Щоб запустити новий сервіс, то потрібний запуск контейнера може бути на будь-якій фізичній машині, він може бути будь-куди перенесеним. Система стає повторюваною.

Практично це виглядає так, якщо у вас все розгорнулося і працює в тестовому оточенні, ви можете з доброю часткою впевненості сказати, що воно розгорнеться в точно таку ж систему і на процесі створення проекту чи творчого задуму (на продакшені).

Нарешті, система починає підтримувати процедуру версіонування. Якщо щось пішло не так, ви можете дістати з розподіленої системи керування версіями Git'a стару конфігурацію і розгорнути все в точності, як було раніше;

4. Технологія Kubernetes є системою оркестрації. Технічне визначення оркестрації – це виконання визначеного робочого процесу: спочатку виконайте A, потім B, потім C.

На відміну від цього, Kubernetes містить набір незалежних, компонуючих процесів управління, які постійно спрямовують поточний стан у потрібний стан. Не має значення, як дістатися від A до C.

Також централізований контроль не потрібен. Це призводить до того, що система простіша у використанні та більш потужна, міцна, еластична і розширювана.

Недоліки Kubernetes:

1. Kubernetes – це не традиційна, всеосяжна система PaaS (Platform as a service – платформа як послуга). Дійсно, PaaS розглядається як модель надання хмарних обчислень, при якій споживач отримує доступ до використання інформаційно-технологічних платформ: операційних систем, систем управління базами даних, зв’язного програмного забезпечення, засобів розробки і тестування розміщених у хмарних провайдерах.

У свою чергу платформа Kubernetes працює на рівні контейнерів, а не на апаратному рівні, тому вона надає деякі загальноприйнятні функції, загальні для PaaS, такі як розгортання, масштабування, балансування навантаження, ведення журналів та моніторинг.

Однак Kubernetes не є монолітним, і ці рішення за замовчуванням є необов’язковими та підключеними. Тому платформа Kubernetes забезпечує

будівельні блоки для побудови інших платформ для розробників, але зберігає вибір та гнучкість користувача там, де це важливо;

2. Kubernetes не обмежує типи підтримуваних програм. Технологія Kubernetes має на меті підтримувати надзвичайно різноманітне навантаження. Якщо програма може працювати в контейнері, вона повинна чудово працювати в Kubernetes;

3. Kubernetes не розгортає вихідний код і не створює вашу програму. Поточні робочі процеси інтеграції, доставки та розгортання (CI/CD) визначаються культурою організації та уподобаннями, а також технічними вимогами;

4. Kubernetes не надає послуг на рівні додатків, таких як середнє програмне забезпечення (наприклад, шини повідомлень), рамки обробки даних (наприклад, Spark), бази даних (наприклад, mysql), кеші, а також розподілені файлові системи зберігання даних (наприклад, Ceph) як вбудовані сервіси.

Такі компоненти можуть працювати на Kubernetes та/або отримувати доступ до них через додатки, що працюють на Kubernetes через портативні механізми, наприклад, Open Broker;

5. Kubernetes не диктує рішення для реєстрації, моніторингу та оповіщення. Він надає деякі інтеграції як доказ концепції та механізми збору та експорту показників;

6. Kubernetes не надає функції конфігурації мовою/системою (наприклад, jsonnet). Він надає декларативний API, на який можуть бути націлені довільні форми декларативних специфікацій;

7. Kubernetes не забезпечує і не приймає жодної комплексної системи конфігурації, обслуговування, управління або самолікування.

На основі виконаного аналізу можна запропонувати наступні пропозиції щодо впровадження мікросервісної архітектури:

1. Контейнери – це хороший спосіб зібрати та запустити програми. У виробничому середовищі потрібно керувати контейнерами, у яких запущені програми, і забезпечувати відсутність простоїв.

Наприклад, якщо контейнер не працює, потрібно запустити інший контейнер. На допомогу приходить Kubernetes. Kubernetes пропонує основу для стійкого запуску розподілених систем. Він піклується про ваші вимоги до масштабування, відмову, схему розгортання тощо. Наприклад, платформа Kubernetes може легко керувати розгортанням каналів для вашої системи;

2. Kubernetes надає: службу виявлення та балансування навантаження. Kubernetes може відкрити контейнер, використовуючи ім’я DNS або використовуючи власну IP-адресу. Якщо трафік до контейнера великий, Kubernetes може завантажувати баланс та розподіляти мережевий трафік так, щоб розгортання було стабільним; оркестрацію зберігання.

Kubernetes дозволяє автоматично монтувати на свій вибір систему зберігання даних, наприклад, локальні сховища, громадські постачальники хмар тощо; автоматизовані розгортання та відкату для відновлення роботи. Можна описати потрібний стан для розгорнутих контейнерів за допомогою Kubernetes, і він може змінити фактичний стан на потрібний стан з контрольованою швидкістю.

Наприклад, ви можете автоматизувати Kubernetes для створення нових контейнерів для розгортання, видалення існуючих контейнерів та прийняття всіх їхніх ресурсів до нового контейнера; автоматичну упаковку у контейнер. Kubernetes дозволяє вказати, скільки потрібно процесорів та пам’яті (ОЗП) кожному контейнеру.

Якщо в контейнерах вказані запити на ресурси, Kubernetes може приймати кращі рішення щодо управління ресурсами для контейнерів; самолікування. Kubernetes перезавантажує контейнери, які виходять з ладу, замінює контейнери, вбиває контейнери, які не відповідають визначеним користувачем вимогам, не показує їх клієнтам, поки вони не будуть готові до обслуговування; таємне та конфігураційне управління.

Kubernetes дозволяє зберігати та керувати конфіденційною інформацією, наприклад паролями, маркерами OAuth та ключами ssh. Ви можете розгортати та оновлювати секрети та конфігурацію програми, не відновлюючи зображення контейнерів та не розкриваючи секрети в конфігурації стека.

Таким чином, було коротко описано, що в залежності від потреб бізнесу можна застосовувати різноманітні технології, які дозволяють підвищувати ефективність роботи компанії.

Головне не забувати, що кожна технологія має своє місце і важливо знати про їхнє існування, але кропітливо підходити до підбору використання тієї чи іншої.

Якщо ви працює на невеликому підприємстві з відчутно обмеженим бюджетом та невеликою кількістю клієнтів, які відвідують ваш сайт, то можливо зовсім не потрібно вибудовувати велику відмовостійку та швидкомасштабовану систему, а краще зважити ризики й оцінити потенційні витрати на підтримку та експлуатацію великої системи, а також можливі витрати при одній годині простою на місяць/тиждень.

Існує величезна вірогідність, що для бізнесу буде дешевше “полежати” одну годину, так як, наприклад, вони зосереджені на офлайн бізнесі.

Головним висновком є те, що не завжди сучасні та модні технології є ключем до вирішення всіх проблем, а при виборі технологій потрібно оцінити ризики, фінанси та потреби компанії у використанні тієї чи іншої систем [18].

4.2 Обрані підходи до збору, зберегання та обробки контенту

В таблиці 4.1 Порівняння підходів до збору контенту наведені порівняні підходи до збору контенту: з використанням веб-скрапінгу (симуляція користувача), через API наданий платформою, з використанням RSS каналів наданих платформою, з соціальних медіа та з використанням краудсорсингу (користувачі самі можуть надсилати контент).

Для цієї роботи було обрано використовувати де можливо API та RSS канали, оскільки це найбільш прості в інтеграції способи, що не потребують великих затрат ресурсів.

В випадку, якщо API та RSS недоступні, було обрано використовувати веб-скрапінг, тобто маскуватися під реального користувача і збирати дані від його імені, але такий підхід потребує більших ресурсів.

Таблиця 4.1

Порівняння підходів до збору контенту

Назва	**Переваги**	**Недоліки**
Веб-скрапінг	Автоматичний збір великих обсягів даних; можливість персоналізації запитів.	Юридичні обмеження та проблеми з авторськими правами; можливість блокування сайтами.
API (Application Programming Interface)	Легальний і надійний доступ до даних; структуровані та актуальні дані.	Обмеження доступу та квоти; залежність від змін у політиці API.
RSS-канали	Простота у використанні; автоматичне оновлення контенту.	Обмеженість у видах контенту; можливе затримання оновлень.
Соціальні медіа	Доступ до широкої аудиторії та різноманітного контенту.	Залежність від алгоритмів платформ; ризик розповсюдження недостовірної інформації.
Краудсорсінг (залучення аудиторії)	Збір унікального та різноманітного контенту; взаємодія з аудиторією.	Потреба у модерації та якісному контролі; можливість поширення неточної інформації.

Для подальшої збірки єдиної стрічки для користувача, контент потрібно десь зберігати. В таблиці 4.2 Порівняння підходів до зберігання контенту наведено порівняння можливих рішень для зберігання контенту.

Для цього було вирішено використовувати SQL базу даних. Сучасні SQL БД добре масштабуються та реплікуються, надають гарантію цілісності даних.

Таким чином, для наших задач вони підходять більше інших баз даних. NoSQL бази даних дозволяють зберігати неструктуровані, жертвуючи гарантіями цілосності цих даних, що є неприйнятним для нашого додатку. S3 та інші файлові сховища підходять для зберігання великих бінарних даних, але погано справляється зі зберіганням та запитами структурованих текстових даних.

DataLake хоч і дозволяє зберігати великі масиви даних, але має обмежені можливості вибірки цих даних та не гарантує консистентності даних, що є небажаним в пропонованій архітектурі.

Таблиця 4.2

Порівняння підходів до зберігання контенту

Назва	**Переваги**	**Недоліки**
SQL Бази Даних	Підходять для комплексних запитів та транзакцій; широко використовуються і підтримують стандартну мову запитів SQL; сильні гарантії цілісності даних.	Вертикальне масштабування може бути дорогим і обмеженим; зміни у структурі даних можуть бути складними.
NoSQL Бази Даних	Горизонтальне масштабування; гнучкість схеми; різноманітність типів (документи, графи, ключ-значення).	Відсутність уніфікованої мови запитів; обмежені можливості управління транзакціями.
Файлові Сховища (наприклад, Amazon S3)	Висока масштабованість і доступність; підходять для зберігання великих об'ємів неструктурованих даних; легкість інтеграції з різними сервісами.	Обмежені можливості для управління метаданими та індексації; не підходить для складних запитів та обробки даних.
Data Lake	Підтримка великої кількості даних; можливість використання даних для аналітики; легка інтеграція з інструментами для обробки великих даних.	Комплексність управління та забезпечення безпеки; можуть виникати складнощі з оптимізацією.

У таблиці 4.3 порівнюються різні способи кешування контенту: на стороні клієнта, на стороні сервера, з використанням CDN та за допомогою бази даних.

Для даної системи агрегації контенту було обрано поєднати кешування на стороні клієнта та з використанням CDN.

Кешування на стороні клієнта дозволяє зменшити навантаження на сервер та покращити час завантаження контенту для повторних відвідувачів.

Використання CDN оптимізує швидкість доставки контенту користувачам завдяки розподіленій мережі серверів та зменшує навантаження на основний сервер.

Ці підходи в сукупності дозволять покращити продуктивність та масштабованість системи. У таблиці 4.3 наведено порівняння підходів до кешування.

Таблиця 4.3

Порівняння підходів до кешування

Назва	**Переваги**	**Недоліки**
Кешування на стороні клієнта	Зменшує навантаження на сервер; покращує час завантаження для повторних відвідувачів.	Контроль та оновлення кешу залежить від клієнта; може викликати проблеми з консистентністю даних.
Кешування на стороні сервера	Забезпечує швидкий доступ до часто запитуваного контенту; контроль та оновлення кешу здійснюється централізовано.	Вимагає додаткових ресурсів на сервері; може виникнути проблема застарілого контенту.
Кешування з використанням CDN	Покращує час завантаження за рахунок географічного розподілу; зменшує навантаження на основний сервер.	Вартість та залежність від стороннього провайдера; може виникнути складність в синхронізації оновлень контенту.
Кешування за допомогою бази даних	Зменшення часу відповіді на запити до бази даних; просте у впровадженні з використанням сучасних СУБД.	Може вимагати значних обчислювальних ресурсів; кеш може стати застарілим при частих оновленнях даних.

У таблиці 4.4 порівнюються різні способи індексації контенту: повнотекстова, на основі метаданих, за допомогою тегів та ієрархічна.

Таблиця 4.4

Порівняння підходів до зберігання індексації

Назва	**Переваги**	**Недоліки**
Повнотекстова індексація	Дозволяє швидко знаходити контент за ключовими словами; підтримує складні запити та пошук за змістом.	Вимагає більше місця для зберігання; може бути ресурсоємною при індексації великих обсягів даних.
Індексація на основі метаданих	Менше вимог до ресурсів та місця для зберігання; швидше індексування.	Менш гнучка для складних запитів; залежить від якості та наявності метаданих.
Індексація за допомогою тегів	Простота у використанні та розумінні; дозволяє легко категоризувати і групувати контент.	Обмежена гнучкість; залежить від точності та послідовності тегування.
Ієрархічна індексація	Добре підходить для структурованого контенту; полегшує навігацію і пошук у великих наборах даних.	Може бути складною в управлінні; менш ефективна для неструктурованого або динамічного контенту.

Для даної системи агрегації контенту було обрано поєднати індексацію на основі метаданих та за допомогою тегів. Індексація за метаданими дозволяє

швидше створювати індекси та економити місце для зберігання, в той час як використання тегів надає гнучкість у категоризації та пошуку контенту.

Повнотекстова індексація була відкинута, оскільки вона вимагає більше ресурсів для обробки та зберігання індексів, що є критичним при великих обсягах даних. Ієрархічна індексація також не підходить, адже контент в системі агрегації є переважно неструктурованим і динамічним, тому гнучкі підходи на основі метаданих і тегів краще підходять для цієї задачі.

Розглянемо типову схему роботи агрегатора та наведемо візуальний вигляд етапів. Фронтенд взаємодіє з бекенд-сервером через REST API для відображення контенту користувачам. Бекенд, у свою чергу, зберігає та обробляє дані в базі даних PostgreSQL. Task runner використовується для автоматизації регулярного завантаження нового контенту з зовнішніх джерел, таких як Spotify, YouTube, RSS-стрічки та інші. Проте, така архітектура має свої недоліки. Уніфікація даних є викликом, оскільки різні джерела часто мають різні формати, що ускладнює їх інтеграцію в єдину систему та представлення користувачам.

Масштабованість також може стати проблемою, оскільки зростання кількості користувачів та джерел вимагає більшої пропускної здатності, обчислювальної потужності та зберігання даних, що може зробити систему дорожчою та складнішою у підтримці. На рис.4.5 наведено Типова схема роботи агрегатора.

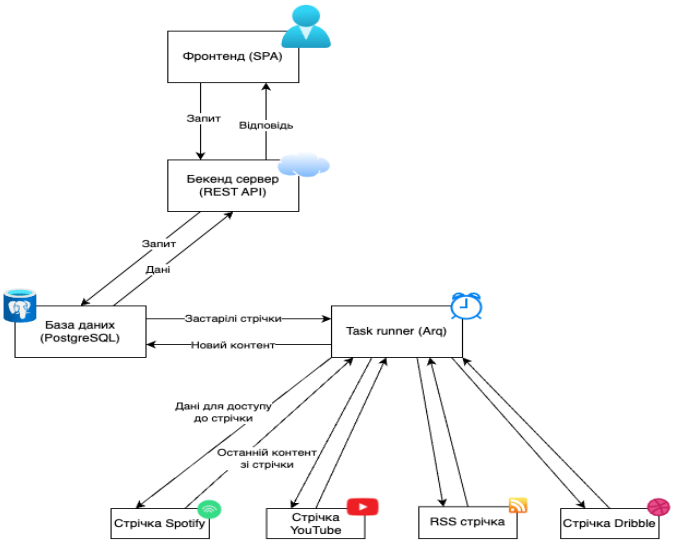


Рис.4.5. Типова схема роботи агрегатора

Розглянемо схему роботи агрегатора з покращенням.

На рис.4.6 зображено Схема роботи агрегатора з покращенням, яка інтегрує різноманітні компоненти для ефективної роботи.

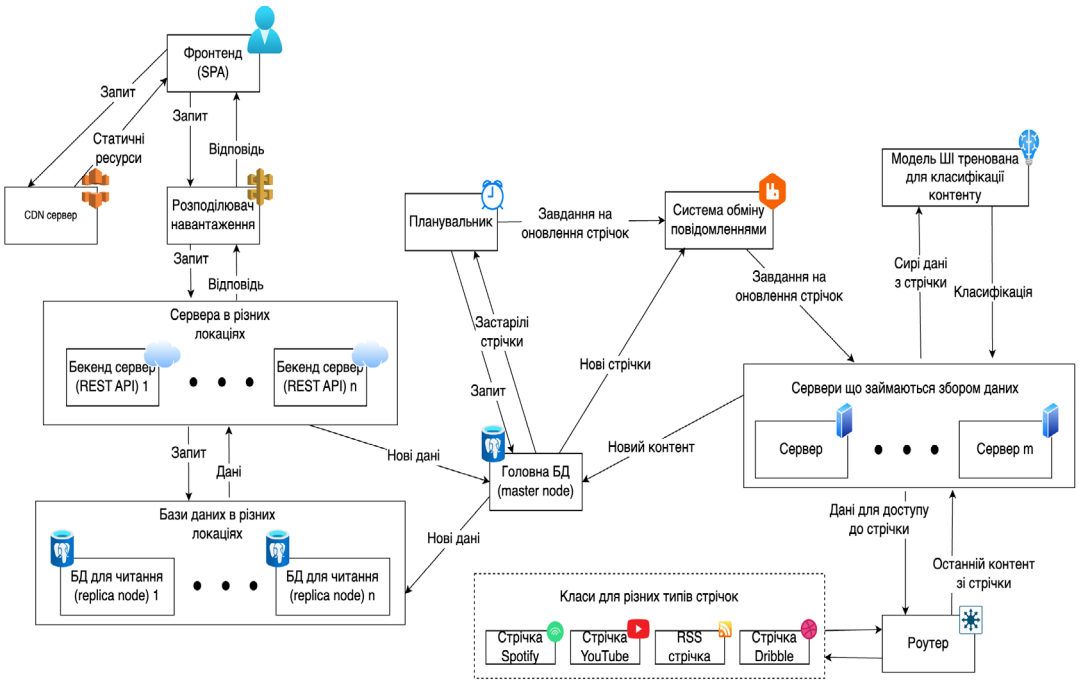


Рис.4.6. Схема роботи агрегатора з покращенням

В основі лежить односторінковий додаток як інтерфейс для кінцевого користувача. Він взаємодіє з бекендом через REST API. Статичні ресурси, такі як зображення та стилі, швидко доставляються через CDN сервер. Розподілювач навантаження гарантує рівномірний розподіл запитів між серверами, розташованими у різних локаціях, забезпечуючи високу доступність та надійність.

Для зменшення навантаження на головний вузол бази даних, запити на читання обробляються через розподілені реплікаційні вузли. Головна база даних координує синхронізацію даних і забезпечує оновлення контенту від зовнішніх джерел. Планувальник автоматизує завдання на оновлення стрічок, в той час як система обміну повідомленнями забезпечує внутрішню комунікацію.

Самим збором займаються окремі сервери, кількість котрих може змінюватись в залежності від кількості стрічок і потрібної частоти оновлень. Виклик коду для збору контенту виконується через роутер, компонент котрий інкапсулює в собі всю складність і відмінності в структурі даних різних стрічок, надаючи уніфікований інтерфейс для решти системи. Така архітектура дозволяє обробляти більші об’єми даних і обслуговувати більшу кількість користувачів.

Діаграма класів системи роутера наведена на рис. 4.7.

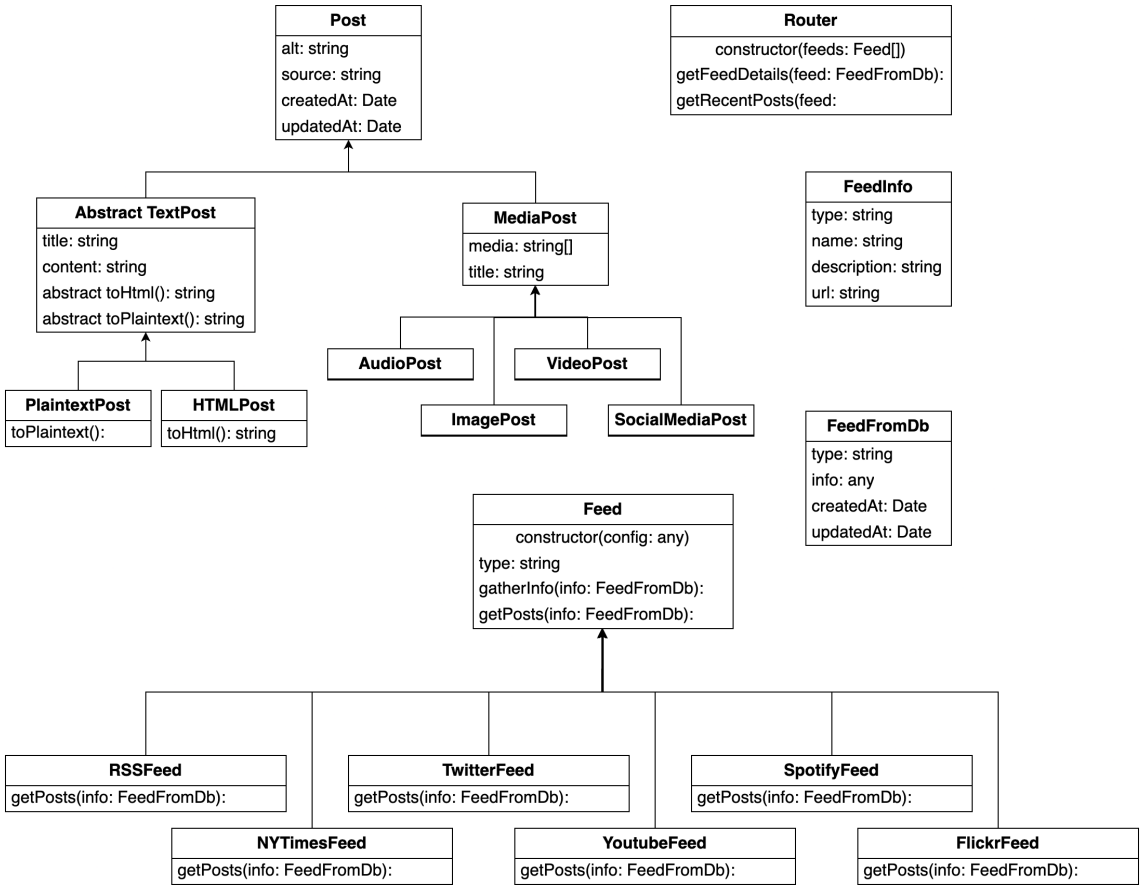


Рис.4.7. Діаграма класів системи роутера

Наведемо код, який займається збором контенту з різних стрічок. Нам вдалося зробити уніфіковану систему, таким чином що імплементація роботи з конкретною платформою схована в окремому класі, котрий реалізує уніфікований інтерфейс і повертає дані в нормалізованому вигляді, що дозволяє решті додатку

не робити якихось додаткових перевірок чи спеціальних дій для роботи з цією стрічкою. Таким чином спрощується підтримка всього додатку і додавання нових типів стрічок.

```
async def update_outdated_feeds(ctx):
    logger.info('Checking feeds for new posts')
    db = ctx['db']
    feeds_raw = db[Feed.collection].find({
        '$where': '!this.updated_at || ((this.updated_at.getTime() + this.update_interval *
60 * 1000) < new ISODate().getTime())'
    })
    feeds = await Feed.from_mongo_cursor(feeds_raw)
    logger.info(f'Feeds to be updated: {len(feeds)}')
    for feed in feeds:
        logger.info(f'Pulling posts for {feed.name}'

        feed_type = get_feed_type_by_id(feed.type)
        posts = await feed_type.get_posts(feed.details)
        await db[Feed.collection].update_one({
            '_id': feed.id,
        }, {
            '$set': {
                'updated_at': datetime.utcnow(),
                'posts': [x.dict() for x in posts],
            }
     })
```

У таблиці 4.5 наведемо порівняння ключових показників продуктивності типової архітектури агрегатори контенту та запропонованої оптимізованої архітектури.

Таблиця 4.5

Таблиця показників типової і пропонованої архітектури

	Типова архітектура	Пропонована архітектура
Середній час відгуку, мс.	560	210
Максимальна кількість запитів в секунду	120	2,400
Максимальна кількість оброблених стрічок за годину	3,000	15,000

Показник середнього часу відгуку важливий, оскільки він характеризує затримку між запитом користувача та отриманням результату. Менший час означає швидшу обробку запитів і кращий досвід для користувача. Максимальна кількість запитів в секунду показує, скільки паралельних запитів система може обробляти, що визначає її пропускну здатність. Більше значення дозволяє обслуговувати більше користувачів одночасно.

Показник максимальної кількості оброблених стрічок за годину характеризує загальну продуктивність системи по збору, аналізу та обробці даних. Більше значення означає вищу пропускну здатність всієї системи в цілому. Отже, запропонована архітектура демонструє суттєве покращення за усіма цими ключовими показниками, що свідчить про її перевагу в термінах продуктивності та масштабованості.

Розроблено прототип хмарного сервісу агрегації новин на основі запропонованих рішень, що демонструє їх працездатність.

На рис.4.4 зображено прототип агрегатора контенту, побудованого за пропонованою архітектурою.

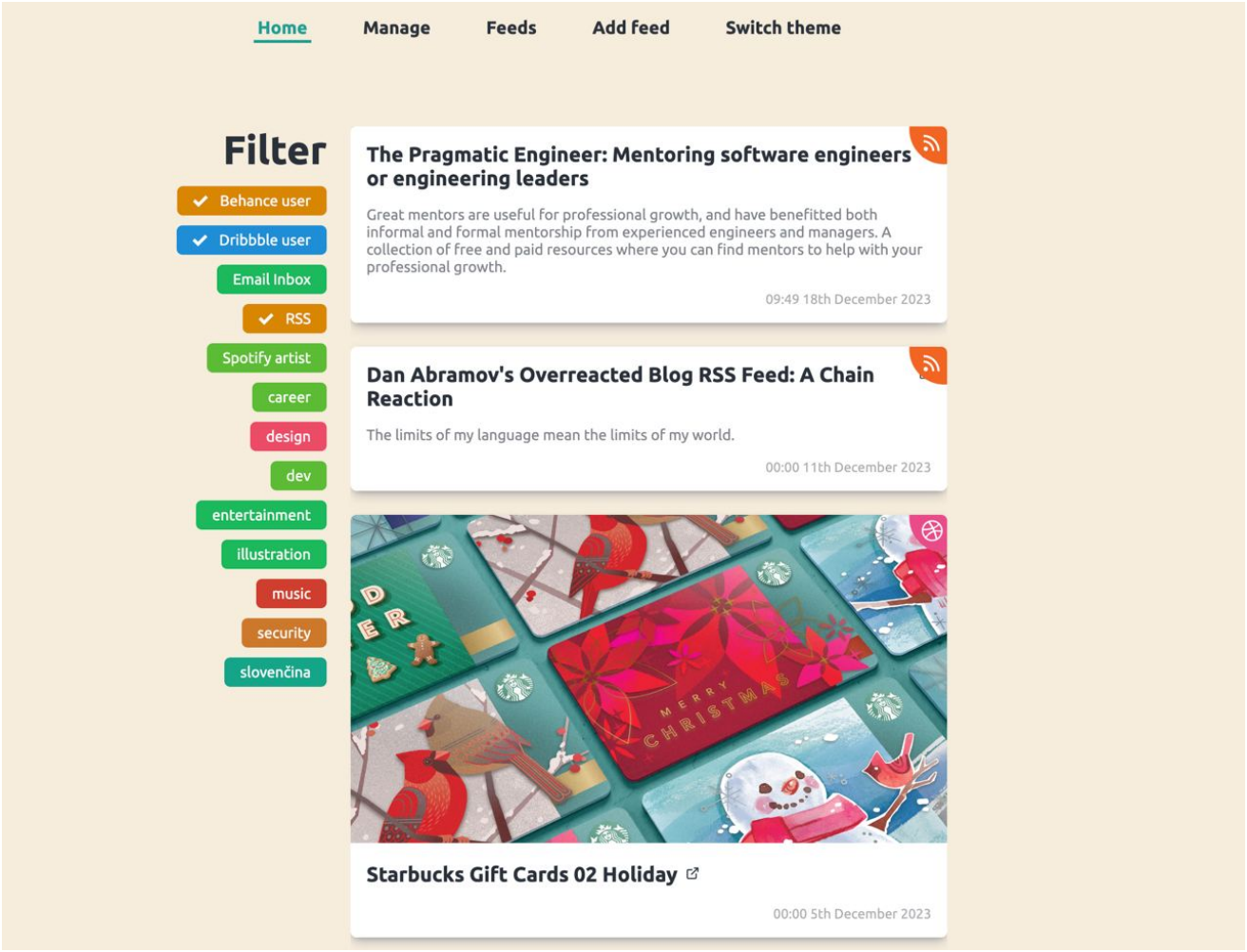


Рис.4.8. Прототип хмарного сервісу агрегації новин

ВИСНОВКИ

В процесі виконання кваліфікаційної роботи, що включала в себе дослідження пов’язані з темою роботи, а саме вдосконалення процесу агрегації контенту з різних джерел з використанням уніфікованого інтерфейсу, було досягнуто наступних результатів:

1. Проаналізовано існуючі підходи та архітектури побудови агрегаторів контенту. Розглянуто їх переваги та недоліки.

2. Досліджено підходи до збору та зберігання контенту з різних джерел.

3. Порівняно ефективність різних стратегій кешування та індексації.

4. Запропоновано за результатами аналізу універсальну мікросервісну архітектуру агрегатора, що дозволяє гнучко масштабувати окремі компоненти.

5. Розроблено прототип хмарного сервісу агрегації новин на основі запропонованих рішень, що демонструє їх працездатність.

ПЕРЕЛІК ПОСИЛАНЬ

1. Балабанов О. С. Аналітика великих даних: принципи, напрямки і задачі (огляд). *Проблеми програмування*. 2019. No 2. С. 47-68. URL: http://dspace.nbuv.gov.ua/xmlui/bitstream/handle/123456789/161487/05-Balabanov.pdf;jsessionid=D60607206FD710CE8DBEAE17F2658C87?sequence=1.

2. Barney N. Amazon Web Services [Електронний ресурс] / Nick Barney // Techtarget. – 2022. – Режим доступу до ресурсу: https://www.techtarget.com/searchaws/definition/Amazon-Web-Services.

3. T.K. B. Machine learning algorithms for social media analysis: A survey [Електронний ресурс] / B. T.K., R. Chandra Sekhara, B. Annushree // ScienceDirect. – 2021. – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/abs/pii/S1574013721000356.

4. Документація «AMQP (Advanced Message Queuing Protocol)» [Електронний ресурс] - Режим доступу: https://www.amqp.org/ 18.10.2023

5. Документація «RabbitMQ»:[Електронний ресурс] - Режим доступу: https://www.rabbitmq.com/ 18.10.2023

6. Article «Why Google Stores Billions of Lines of Code in a Single Repository»: - Rachel Potvin and Josh Levenberg [Електронний ресурс] - Режим доступу: https://dl.acm.org/doi/pdf/10.1145/2854146 18.10.2023

7. "Впровадження архітектур мікросервісів" Боріс Шоль, Лео Штудер, Майк Амундсен

8. "Сервісно-Орієнтована Архітектура: Концепції, Технології та Дизайн" Томас Ерл

9. Клієнт-серверна архітектура [Електронний ресурс] // QATestLab. – 2020. – Режим доступу до ресурсу: https://training.qatestlab.com/blog/technical articles/client-server-architecture/.

10. Агрегація в MongoDB: зменшення сукупного трубопроводу та карти [Електронний ресурс]. – 2016. – Режим доступу до ресурсу: https://uk.myservername.com/aggregation-mongodb.

11. The top programming languages [Електронний ресурс] // GitHub. – 2022. – Режим доступу до ресурсу: https://octoverse.github.com/2022/top programming-languages.

12. "Шаблони проектування сервісів: Фундаментальні рішення для SOAP/WSDL та RESTful веб-сервісів" Роберт Деньо

13. Рейтинг мов програмування 2023 [Електронний ресурс] // Редакція DOU. – 2023. – Режим доступу до ресурсу: https://dou.ua/lenta/articles/language rating-2023/.

14. "Data Structures and Algorithms in Python", Michael T. Goodrich and Roberto Tamassia, 4th edition, Wiley, 2022.

15. "OSPF: The Basics", Juniper Networks, 2023.

16. Tennis Sense: A platform for extracting semantic information from multi-camera tennis data [Електронний ресурс]. Режим https://ieeexplore.ieee.org/document/5201152/authors#authors

17. Shvachych G., Moroz B., Ivaschenko O. , Sushko I., Pobochii I. The implementation features of the aggregation mode of network interface in the multiprocessors computing system. Комп'ютерне моделювання та оптимізація складних систем : Матеріали IV міжнар. науково-практ. конф., м. Дніпро, 1 – 2 листоп. 2018 р. Дніпро. 2018. С. 200 – 202.

18. Ільїн О.Ю., Катков Ю.І., Вергун Д.С., Шашлов А.В. Особливості розгортання мікросервісних додатків за допомогою системи керування контейнерами.

19. The Computer and the Brain (The Silliman Memorial Lectures Series) 3rd Edition by von Neumann, John (Author), Ray Kurzweil (Foreword), Yale University Press; 3 edition, 2012, 136 pages, ISBN-10: 0300181116, ISBN-13: 978-0300181111.

20. Computer Architecture: A Quantitative Approach 5th Edition by John L. Hennessy (Author), David A. Patterson (Author), Morgan Kaufmann; 5 edition (September 30, 2011), 856 pages, ISBN-10: 012383872X, ISBN-13: 978-8178672663.

21. Microservices vs. Service-Oriented Architecture. by Mark Richards. Publisher: O'Reilly Media, Inc.Release Date: April 2016. ISBN: 9781491975657.

22. Ivan Zmerzlyi Microservice architecture – [Електронний ресурс] – 2019 – Режим доступу: https://medium.com/@IvanZmerzlyi/microservice-architecture-f8a382291ff4. Дата доступу: жовтень 2023.

23. Xiao Ma Microservice Architecture at Medium – [Електронний ресурс] – 2019 – Режим доступу: https://medium.engineering/microservice-architecture-at-medium-9c33805eb74f. Дата доступу: жовтень 2023.

24. Netflix MSA Platform – MeltingCon – [Електронний ресурс] – 2019 – https://meltingcon.github.io/2018/assets/files/%EC%A0%95%EC%9C%A4%EC%A7%84.pdf. Дата доступу: жовтень 2023.

25. Docker Cookbook/ by Sébastien Goasguen. Copyright © 2016 Sébastien Goasguen. All rights reserved. Printed in the United States of America. Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

26. Чи завжди потрібні Docker, мікросервіси та реактивне програмування? – [Електронний ресурс] – 2019 – Режим доступу: https://www.dataart.com.ua/news/chi-zavzhdi-potribni-dockermikroservisi-ta-reaktivne-programuvannya. Дата доступу: жовтень 2023.

27. Kubernetes: Up and Running, 2nd Edition \ Dive into the Future of Infrastructure. By Brendan Burns, Kelsey Hightower, Joe Beda – Publisher: O'Reilly Media, Release Date: October 2023. Pages: 278.

28. Cloud Native DevOps with Kubernetes – by Justin Domingus, John Arundel. Publisher: O'Reilly Media, Inc. Release Date: March 2019. ISBN: 9781492040750.

29. Offcial Kubernetes documentation https://kubernetes.io/docs/concepts/overview/what-iskubernetes. Дата доступу: жовтень 2023.

30. Ключевые концепции Kubernetes для службы Azure Kubernetes (AKS) – [Електронний ресурс] – 2019 – Режим доступу: https://docs.microsoft.com/ru-ru/azure/aks/concepts-clustersworkloads. Дата доступу: жовтень 2023.

31. Швачич Г.Г., Соболенко О.В., Іващенко О.В. Режим агрегації каналів мережевого інтерфейсу багатопроцесорних обчислювальних систем. Стратегия качества в промышленности и образовании : Материалы конф.,. г. Варна, 5 – 8 июля 2017 г. Варна, Болгария. 2017. Т. 2. С. 452 – 457.

32. Ivaschenko V., Shvachych G., Ivaschenko O., Busygin, V. Improving the efficiency of multiprocessors system through in-line interface network aggregation. Системні технології. Дніпро. 2018. No 2(115). P. 84 – 93.Книга "Мікросервіси. Патерни розробки та рефакторинг" (Microservices: Design Patterns and Refactoring): авторство - G. Макклелланд, М. Джонсон, С. Хуттон.

33. Cloud-assisted body area networks: State-of-the-art and future challenges – [Електронний ресурс]. Режим доступу: https://www.researchgate net/ publication/271918886_Cloud- assisted body area_networks_State-of-the-art_and_future_challenges.

34. The Fundamentals of Data Warehouse + Data Lake = Lake House. – Режим доступу:https://towardsdatascience.com/the-fundamentals-of-data-warehouse-data-lake-lake

35. Olena Vynokurova, Dmytro Peleshko, Marta Peleshko Hybrid Deep Convolutional Neural Network with Multimodal Fusion. In: Babichev S., Peleshko D., Vynokurova O. (eds) Data Stream Mining & Processing. DSMP 2020. Communications in Computer and Information Science. Springer, Cham. 2020. vol. 1158. pp. 62-78.

36. The Fundamentals of Data Warehouse + Data Lake = Lake House. – Режим доступу: https://towardsdatascience.com/the-fundamentals-of-data-warehouse-data-lake-lake

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ПРЕЗЕНТАЦІЯ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО – КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

МАГІСТЕРСЬКА РОБОТА
«ВДОСКОНАЛЕННЯ ПРОЦЕСУ АГРЕГАЦІЇ КОНТЕНТУ З РІЗНИХ ДЖЕРЕЛ З ВИКОРИСТАННЯМ УНІФІКОВАНОГО ІНТЕРФЕЙСУ»

Виконав: Студент групи ПДМ – 62 Корнієнко Олег Романович
Керівник: к.т.н., доцент кафедри ІПЗ Негоденко Олена Василівна

Київ - 2024

2

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення показників масштабування шляхом оптимізації архітектури системи

Об'єкт дослідження: процес агрегації контенту з різних джерел.

Предмет дослідження: моделі керування агрегацією контенту з різних джерел.

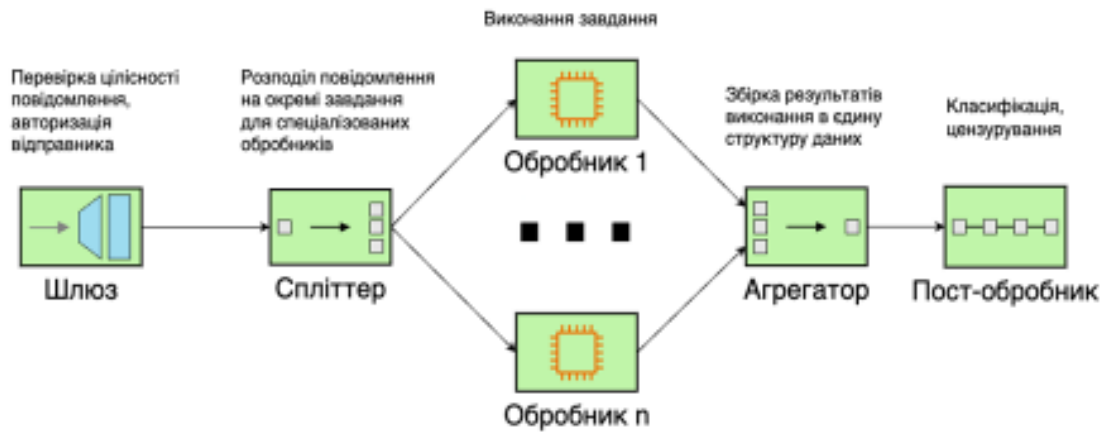
АНАЛІЗ ІСНУЮЧИХ ІНСТРУМЕНТІВ АГРЕГАТОРА НОВИН

3

	Платформа	Джерела контенту	Максимальна кількість джерел контенту	Ранжування контенту
Flocker	Веб-додаток	Соціальні мережі	60	За датою публікації
Google News	Веб-додаток, Android, iOS	Сайти з новинами	20,000	За датою публікації
SmartNews	Android, iOS	Сайти з новинами	14,000	За датою публікації
Flipboard	Веб-додаток, Android, iOS	Сайти з новинами, незалежні видавці	4,000	За інтересами користувача
Пропонована система	Веб-додаток	Соціальні мережі, сайти з новинами, незалежні видавці	В залежності від доступних ресурсів	За датою публікації та інтересами користувача

4

СХЕМА SPLITTER AGGREGATOR



ТИПОВА СХЕМА РОБОТИ АГРЕГАТОРА

5

ПОРІВНЯННЯ ПІДХОДІВ ДО ЗБОРУ КОНТЕНТУ

6

Назва	Переваги	Недоліки
Веб-скрапінг	Автоматичний збір великих обсягів даних; можливість персоналізації запитів.	Юридичні обмеження та проблеми з авторськими правами; можливість блокування сайтами.
API (Application Programming Interface)	Легальний і надійний доступ до даних; структуровані та актуальні дані.	Обмеження доступу та квоти; залежність від змін у політиці API.
RSS-канали	Простота у використанні; автоматичне оновлення контенту.	Обмеженість у видах контенту; можливе затримання оновлень.
Соціальні медіа	Доступ до широкої аудиторії та різноманітного контенту.	Залежність від алгоритмів платформ; ризик розповсюдження недостовірної інформації.
Краудсорсінг (залучення аудиторії)	Збір унікального та різноманітного контенту; взаємодія з аудиторією.	Потреба у модерації та якісному контролі; можливість поширення неточної інформації.

ПОРІВНЯННЯ ПІДХОДІВ ДО ЗБЕРІГАННЯ КОНТЕНТУ 7

Назва	Переваги	Недоліки
SQL Бази Даних	Підходять для комплексних запитів та транзакцій; широко використовуються і підтримують стандартну мову запитів SQL; сильні гарантії цілісності даних.	Вертикальне масштабування може бути дорогим і обмеженим; зміни у структурі даних можуть бути складними.
NoSQL Бази Даних	Горизонтальне масштабування; гнучкість схеми; різноманітність типів (документи, графи, ключ-значення).	Відсутність уніфікованої мови запитів; обмежені можливості управління транзакціями.
Файлові Сховища (наприклад, Amazon S3)	Висока масштабованість і доступність; підходять для зберігання великих об'ємів неструктурованих даних; легкість інтеграції з різними сервісами.	Обмежені можливості для управління метаданими та індексації; не підходить для складних запитів та обробки даних.
Data Lake	Підтримка великої кількості даних; можливість використання даних для аналітики; легка інтеграція з інструментами для обробки великих даних.	Комплексність управління та забезпечення безпеки; можуть виникати складнощі з оптимізацією.

ПОРІВНЯННЯ ПІДХОДІВ ДО КЕШУВАННЯ 8

Назва	Переваги	Недоліки
Кешування на стороні клієнта	Зменшує навантаження на сервер; покращує час завантаження для повторних відвідувачів.	Контроль та оновлення кешу залежить від клієнта; може викликати проблеми з консистентністю даних.
Кешування на стороні сервера	Забезпечує швидкий доступ до часто запитуваного контенту; контроль та оновлення кешу здійснюється централізовано.	Вимагає додаткових ресурсів на сервері; може виникнути проблема застарілого контенту.
Кешування з використанням CDN	Покращує час завантаження за рахунок географічного розподілу; зменшує навантаження на основний сервер.	Вартість та залежність від стороннього провайдера; може виникнути складність в синхронізації оновлень контенту.
Кешування за допомогою бази даних	Зменшення часу відповіді на запити до бази даних; просте у впровадженні з використанням сучасних СУБД.	Може вимагати значних обчислювальних ресурсів; кеш може стати застарілим при частих оновленнях даних.

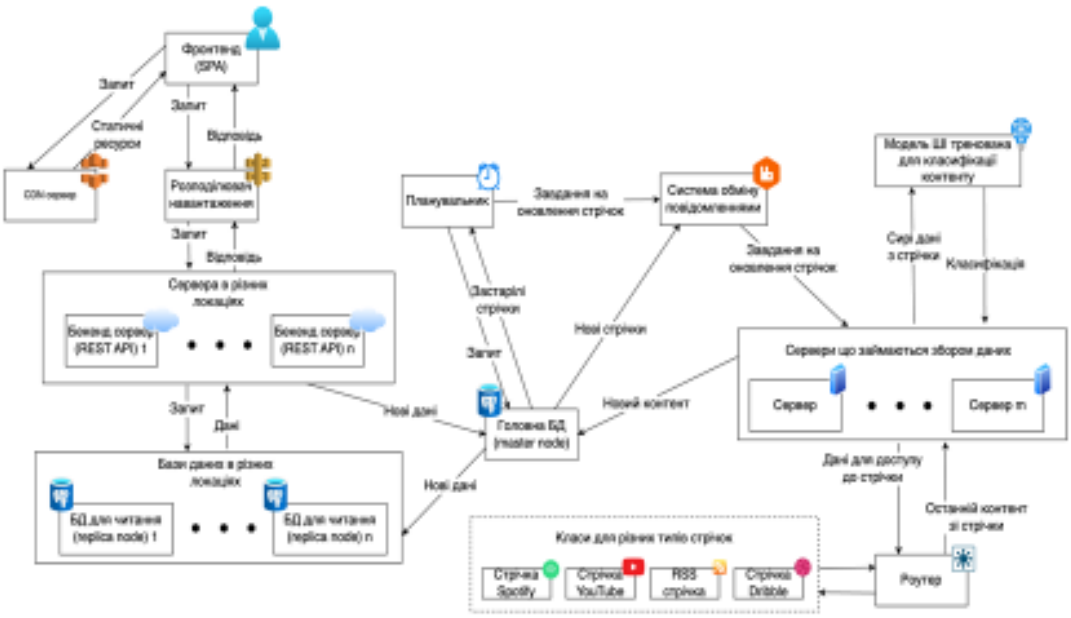
ПОРІВНЯННЯ ПІДХОДІВ ДО ІНДЕКСАЦІЇ

9

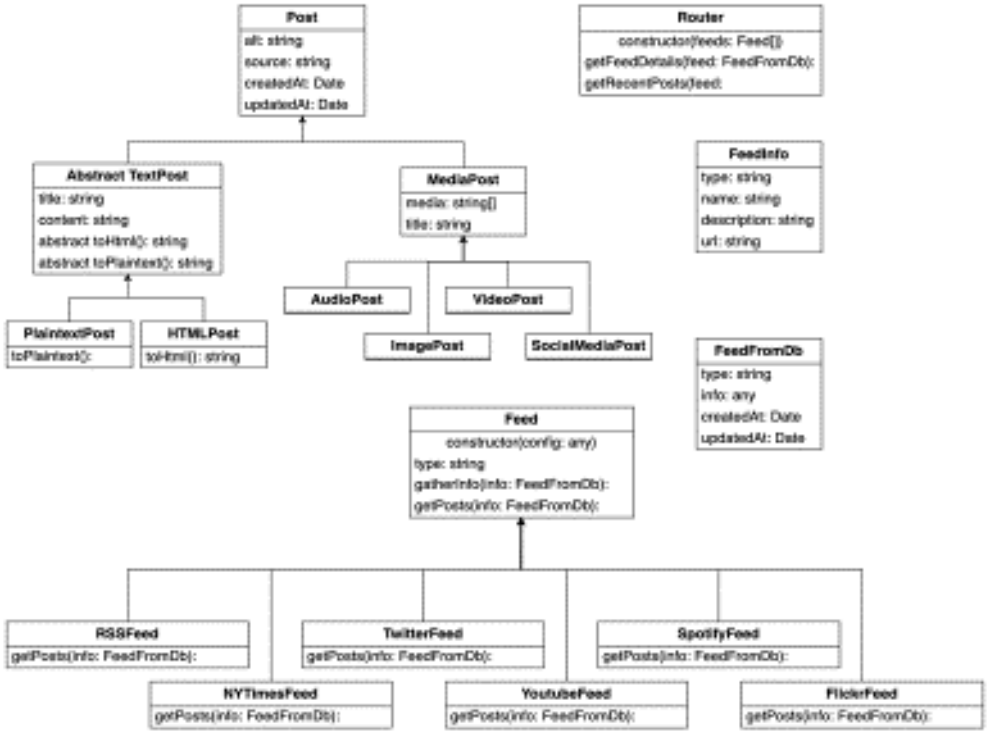
Назва	Переваги	Недоліки
Повнотекстова індексація	Дозволяє швидко знаходити контент за ключовими словами; підтримує складні запити та пошук за змістом.	Вимагає більше місця для зберігання; може бути ресурсоємною при індексації великих обсягів даних.
Індексація на основі метаданих	Менше вимог до ресурсів та місця для зберігання; швидше індексування.	Менш гнучка для складних запитів; залежить від якості та наявності метаданих.
Індексація за допомогою тегів	Простота у використанні та розумінні; дозволяє легко категоризувати і групувати контент.	Обмежена гнучкість; залежить від точності та послідовності тегування.
Ієрархічна індексація	Добре підходить для структурованого контенту; полегшує навігацію і пошук у великих наборах даних.	Може бути складною в управлінні; менш ефективна для неструктурованого або динамічного контенту.

СХЕМА РОБОТИ АГРЕГАТОРА З ПОКРАЩЕННЯМ

10



ДІАГРАМА КЛАСІВ СИСТЕМИ РОУТЕРА 11



ПОРІВНЯННЯ ПОКАЗНИКІВ ТИПОВОЇ І ПРОПОНОВАНОЇ АРХІТЕКТУР 12

	Типова архітектура	Пропонована архітектура
Середній час відгуку, мс.	560	210
Максимальна кількість запитів в секунду	120	2,400
Максимальна кількість оброблених стрічок за годину	3,000	15,000

13

ВИСНОВКИ

1. Проаналізовано існуючі підходи та архітектури побудови агрегаторів контенту. Розглянуто їх переваги та недоліки.

2. Досліджено підходи до збору та зберігання контенту з різних джерел.

3. Порівняно ефективність різних стратегій кешування та індексації.

4. Запропоновано за результатами аналізу універсальну мікросервісну архітектуру агрегатора, що дозволяє гнучко масштабувати окремі компоненти.

14

ПУБЛІКАЦІЇ ТА АПРОБАЦІЯ РОБОТИ

Стаття:

1. Негоденко О.В., Корнієнко О.Р. Вдосконалення процесу агрегації контенту з різних джерел з використанням уніфікованого інтерфейсу // Київ: Наукові записки Державного університету телекомунікацій, 2023. – № 2.

Тези доповідей:

1. Негоденко О.В., Корнієнко О.Р. Агрегації контенту з використанням уніфікованого інтерфейсу // Науково-практична конференція «Проблеми комп'ютерної інженерії». – Київ: ДУТ, 2023.
2. Негоденко О.В., Корнієнко О.Р. Прототип хмарного сервісу агрегації новин//V Всеукраїнська науково-практична конференція «Telecommunication: problem and innovation». – Київ: ДУТ, 2023