Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ШЛЯХИ ІМПЛЕМЕНТАЦІЇ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ В РЕКОМЕНДАЦІЙНИХ СИСТЕМАХ

Текстова частина до курсової роботи за спеціальністю «Комп'ютерні науки» - 122

Керівник курсової роботи
к.т.н., доц. *Олецький О. В.*

_____
(підпис)
"____" _____ 2022 р.

Виконав студент *Волошко М. В.*
"____" _____ 2022 р.

Київ 2022

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри інформатики,
д.т.н, доцент
_____А. М. Глибовець
(підпис)
„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту_____ факультету _____ _____курсу
ТЕМА _____

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. Аналіз поняття рекомендаційної системи
6. Опис необхідних змін в старій версії застосунку
7. Розробка нової версії застосунку
8. Висновки
9. Список використаних джерел

Додатки (за необхідністю)

Дата видачі „___” _____ 2022 р. Керівник _____
(підпис)
Завдання отримав _____

(підпис)

Календарний план виконання курсової роботи

Тема: Шляхи імплементації мікросервісної архітектури в рекомендаційних системах

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	18.10.2021	
2.	Огляд технічної літератури за темою роботи.	5.12.2021	
3.	Аналіз поняття рекомендаційної системи	15.12.2021	
3.	Аналіз принципів та підходів до побудови мікросервісної архітектури у рекомендаційних системах	10.01.2022	
4.	Аналіз Опис необхідних змін в старій версії	25.01.2022	
5.	Створення нової моделі застосунку	15.02.2022	
6.	Виконання практичної частини курсової роботи	01.05.2022	
7.	Написання текстової частини курсової роботи	13.05.2022	
8.	Аналіз отриманих результатів з керівником та попередній захист курсової роботи.	15.05.2022	
9.	Корегування роботи за результатами попереднього захисту.	16.05.2022	

Студент *Волошко М. В.*

Керівник *Олецький О. В.*

“_____”_____

ЗМІСТ

Анотація

Робота присвячена дослідженню аспектів впровадження мікросервісної архітектури у рекомендаційних системах. Були проаналізовані необхідні для старої версії застосунку зміни з метою досягти кращої продуктивності. Було розроблено нову версію застосунку та впроваджені усі необхідні покращення.

Вступ

Рекомендаційні системи зарекомендували себе як перевірене рішення для сприяння підвищенню активності користувачів у системі. Завдяки підтримуванню зацікавленості користувачів у продукті можливо добитися найбільш тривалого користування ним.

Існують певні типи рекомендаційних систем, кожен з яких має свою точність прогнозів, складність імплементації чи переважаючої значимості недоліків, проте такі складнощі будуть виправдані при побудові надійного застосунку.

Минулорічна курсова робота була присвячена створенню застосунку з мікросервісною архітектурою. Цього року було здійснене рішення продовжити розробку того застосунку та здійснити певні зміни, щоб підготувати його до інтеграції сервісу рекомендацій.

Перший розділ присвячено аналізу поняття рекомендаційної системи. Були описані основні характеристики типів рекомендаційних систем, їх переваги та недоліки.

У другому розділі було висвітлено необхідні зміни старої версії застосунку з метою впровадження рекомендаційної системи. Було обґрунтовано рішення стосовно внесення відповідних поправок.

Третій розділ присвячено розробці нової версії застосунку та розкриттю аспектів конкретної імплементації всіх необхідних змін для підготовки застосунку до впровадження сервісу рекомендацій.

Постановка задачі:

1. Провести аналіз поняття рекомендаційної системи. Дослідити різні типи рекомендаційних систем, їх переваги та недоліки.
2. Проаналізувати попередню модель застосунку та обґрунтувати необхідні зміни для нової версії застосунку.

3. Розкрити аспекти імплементації необхідних змін при розробці нової версії застосунку. Впровадити рекомендаційну систему у застосунку з мікросервісною архітектурою.

Перелік прийнятих скорочень

API – прикладний програмний інтерфейс

REST – репрезентативна передача стану

NPM – пакетний менеджер Node.js

ORM – об’єктно-реляційне відображення

РОЗДІЛ 1: АНАЛІЗ ПОНЯТТЯ РЕКОМЕНДАЦІЙНОЇ СИСТЕМИ

1.1 Поняття рекомендаційної системи

Згідно з визначенням у книзі[1], рекомендаційна система – це підклас систем фільтрації інформації, який прагне передбачити оцінку або перевагу, яку користувач може надати елементу. Такі системи пропонують користувачам елементи, яким вони надали б перевагу серед інших. Рекомендаційні системи використовуються у більшості компаній-гігантів зі сфери інформаційних технологій, таких як: Google, Netflix, Facebook, Twitter тощо. Завдяки наданню користувачам релевантних елементів предметної області, користувачі отримують більше задоволення від користування послугами певного бізнесу, що у свою чергу призводить до більшого успіху самого продукту та і компанії в цілому.

1.2 Типи рекомендаційних систем

Рекомендаційні системи, завдяки різноманіттю підходів до обробки інформації, обрахуванню релевантності та здійсненню припущень, поділяються на 3 основні типи: Collaborative filtering, Content-Based filtering та Hybrid.

1.2.1 Collaborative filtering

Під колаборативною фільтрацію розуміється процес знаходження користувачів зі схожими до певного користувача інтересами. Це досягається за допомогою аналізу даних з якими користувачі стикалися, надавали зворотній зв’язок стосовно них та в яких даних він можливо буде зацікавлений у майбутньому. Колаборативна фільтрація реалізується за допомогою принципу «якщо користувачу 1 релевантний елемент А і користувачу 2 релевантний

елемент А, то якщо користувачу 2 релевантний елемент Б, то елемент Б буде релевантним користувачу 1».

1.2.1.1 Переваги типу Collaborative filtering

Головною перевагою рекомендаційних систем типу Collaborative filtering є більш широкий діапазон потенційних релевантних елементів, оскільки суть принципу Collaborative filtering є не чіткий зв’язок релевантності до певних характеристик елементів, а зв’язок між користувачами зі схожими інтересами. Така риса надає перевагу над підходом Content-Based, оскільки у випадку фільтрації базовано на контенті користувачу будуть надаватися рекомендації строго по його вподобаннях.

Черговою перевагою Collaborative filtering є можливість не надавати елементам надлишкової детальної інформації для їх аналізу, оскільки підхід Collaborative filtering не потребує поглиблених характеристик для елементів певної предметної області[2].

1.2.1.2 Недоліки типу Collaborative filtering

Рекомендаційні системи типу Collaborative filtering у зв’язку з принципом своєї роботи мають певні недоліки, такі як масштабованість, розрідженість даних та «холодний старт».

Проблема масштабованості для підходу Collaborative filtering стосується стрімкого росту необхідної обчислювальної потужності машини при збільшенні об’єму даних. Оскільки принцип фільтрації полягає в аналізі усіх користувачів системи та їх переліку релевантних та нерелевантних елементів, велика база користувачів системи робить необхідним обчислити та проаналізувати кожного з них.

Проблема розрідженості даних полягає у малій вибірці частини предметної області, оскільки користувачі в основному надають зворотній зв’язок та

взаємодіють лише з малою часткою усієї предметної області. Так як підхід Collaborative filtering надає рекомендації згідно з аналізом наданої релевантності елементів користувачами, система може рекомендувати менш популярні елементи, в той час як значно популярніші не будуть задіяні.

Проблема «холодного старту» у рекомендаційних системах типу Collaborative filtering полягає у значному зменшенні точності рекомендацій при малому об’єму даних про користувача, оскільки при більш широкому спектрі користувачів зі схожими релевантними елементами алгоритм буде краще аналізувати потенційні рекомендації.

1.2.2 Content-based filtering

Фільтрація базовано на контенті відрізняється від колаборативної фільтрації тим, що їй не потрібні дані про інших користувачів. Весь процес формування рекомендацій проходить навколо аналізу даних конкретного користувача. Цей підхід включає в себе збереження детальних характеристик кожного елементу предметної області для рекомендацій. Процес обчислення рекомендацій включає в себе аналіз характеристик елементів, з якими користувач взаємодіяв чи для яких надавав зворотній зв’язок, з чого потім будується релевантність певних характеристик для користувача і подальше надання рекомендацій.

1.2.2.1 Переваги типу Content-Based filtering

Основною перевагою рекомендаційних систем типу Content-Based filtering є можливість точних рекомендацій при наявному малому об’єму даних про вподобання користувача. Цей підхід вирішує одну з головних проблем рекомендаційних систем типу Collaborative filtering «холодного старту», оскільки може надавати релевантні для користувача елементи базуючись лише на характеристиках самих елементів та мінімальних даних про релевантність для користувача.

Також Content-Based filtering дозволяє швидко підлаштовуватися під зміни вподобань користувача, в той час як Collaborative filtering вимагає тривалого часу для нових обчислень рекомендацій відповідно до інших користувачів з спільними інтересами.

1.2.2.2 Недоліки типу Content-Based filtering

Головним недоліком підходу Content-Based filtering є обмеженість різноманітності релевантних елементів, оскільки користувачу будуть надаватися рекомендації, основані строго на схожих релевантних для користувача елементах. Ця проблема значно знижує значимість рекомендаційних систем цього типу[2], оскільки система не зможе надавати користувачу нових для нього елементів, цим самим знижуючи задоволеність користування системою, що призведе до значних втрат бізнесу.

1.2.3 Hybrid

Гібридні рекомендаційні системи комбінують різні підходи до обчислення рекомендацій для користувача. За допомогою такої комбінації переваги одних підходів компенсують недоліки інших. Більшість гігантів серед компаній у сфері інформаційних технологій використовують гібридні рекомендаційні системи.

1.2.3.1 Переваги типу Hybrid

Основною перевагою рекомендаційних систем типу Hybrid є висока точність рекомендацій. Це досягається за допомогою компенсування недоліків різних підходів за допомогою переваг інших. Різні підходи можна комбінувати різноманітними способами, наприклад, використовувати об’єднаний результат від базованої на контенті фільтрації та колаборативної, внесення одного з них як додатковий крок під час роботи іншого тощо.

1.2.3.2 Недоліки типу Hybrid

Комбінування різних підходів до рекомендацій має необхідність у високій обчислювальній потужності, оскільки кожен з них потребує свого типу даних, що призводить до високої навантаженості системи.

РОЗДІЛ 2: ОПИС НЕОБХІДНИХ ЗМІН В СТАРІЙ ВЕРСІЇ ЗАСТОСУНКУ

2.1 Зміни в моделі застосунку

При подальшій розробці минулорічного застосунку консультаційного сервісу, були внесені певні зміни до самої моделі застосунку. У попередній версії застосунку це були сервіси користувачів, постів, консультацій, рекомендацій, сфер та API-шлюз. У новій версії було змінено основну бізнес-функцію сервісу сфер, сутність спеціальностей може однозначно ідентифікувати сферу, пов’язану з нею. Сутність спеціальностей більш ширше використовується у бізнес-логіці застосунку, тому сервіс сфер було змінено на сервіс спеціальностей. Також було прийнято винести в окремий сервіс логіку, орієнтовану навколо коментарів до постів, оскільки у майбутньому планується активно розвивати його, зокрема додаючи модерацію контенту коментарів. Додатково було змінено назву сервісу користувачів на сервіс автентифікації, оскільки основне його завдання це імплементація автентифікації та авторизації у всьому застосунку, а не виключне здійснення логіки навколо користувачів. Також було видалено сервіс API-шлюз, який слугував єдиною точкою доступу до проекту, містив реєстр сервісів та за допомогою проксі перенаправляв запити до них, оскільки було використано рішення балансувальника навантаження під назвою Ingress Nginx.

2.2 Контейнеризація сервісів

Для нової версії застосунку було вирішено ізолювати середовища виконання для сервісів, що надало певні переваги. Для розробки це дозволило легше та швидше будувати та запускати сервіси, а також зберігати зображення сервісів у хмарному середовищі, за допомогою чого був автоматизований одночасний запуск усіх сервісів. У подальшому розвитку проекту контейнеризація сервісів надає більшого захисту сервісам від зловмисних атак. Контейнери, при майбутньому розвитку застосунку та можливому його розгортанню, дозволять швидко та легко розгортати сервісі у хмарних

провайдерах, індивідуально налаштувати життєвий цикл та правила масштабованості кожного з них [3]. Також робота з контейнерами дозволяє використовувати такий інструмент, як Kubernetes, який надає можливості для налаштування ізольованого спілкування між сервісами, правилами їх запуску, перезапуску, зупинки тощо. Завдяки інструменту Kubernetes було використано зображення сервісу Ingress Nginx, який виступає у застосунку у ролі балансувальника навантаження та налаштовано перенаправляв запити із зовнішнього середовища до доступних ізольованих сервісів. За допомогою Kubernetes були створені Deployments у контексті об’єктів Kubernetes. Їхня основна задача налаштовано створювати, запускати, та видаляти Pods, які в контексті Kubernetes містять у собі контейнери. Для налаштування роботи Deployments використовуються маніфести, де вказуються основні налаштування та принцип роботи. Для кожного Pods також були створені Services типу Cluster IP, які слугують засобом внутрішньої комунікації Pods усередині кластеру Kubernetes. Ще одним Service у застосунку також є вищезгаданий балансувальник навантаження Ingress Nginx, який є об’єктом Kubernetes Service типу Load Balancer.

2.3 Спосіб комунікації між сервісами

У попередній версії застосунку використовувався синхронний спосіб комунікації сервісів між собою за допомогою протоколу HTTP. Синхронна комунікація легка в імплементації і за допомогою архітектури REST надає можливість побудувати якісну архітектуру комунікації сервісів. Проте сам принцип синхронності у комунікації між мікросервісами несе в собі проблему в очікуванні відповіді від сервісу для подальшої обробки даних, що значно затримує роботу всього застосунку. Також синхронний спосіб комунікації між мікросервісами створює залежності між сервісами, що порушує головну ідею мікросервісів – автономність кожного з них. У випадку якщо один з сервісів з якоїсь причини буде недоступним, сервіс, який надсилав запит на дані з

недоступного сервісу, також може вийти з ладу або просто не нести за собою будь-якої користі. Один з головних принципів мікросервісів – це незалежність кожного з сервісів, тому у новій версії застосунку було зроблене рішення реалізувати асинхронну комунікацію між сервісами.

Асинхронна комунікація між сервісами в першу чергу дозволяє позбутися залежності між мікросервісами, що дозволяє кожному з них функціонувати автономно. Завдяки цьому, якщо якийсь з сервісів вийде з ладу, це не зупинить роботу всього застосунку загалом, а лише якась певна частина функціоналу буде тимчасово недоступна для користувача, в той час як він все ще зможе користуватися іншими функціями якогось бізнесу. Іншою перевагою асинхронного спілкування між сервісами є швидкість обробки даних на сервері. Це досягається завдяки позбавленню від необхідності очікування на результати кожного запиту до якогось сервісу всередині мікросервісного застосунку. Наприклад, якщо користувач надсилає запит на реєстрацію до сервісу користувачів, застосунок оброблює лише ті дані, якими він сам володіє, і відразу надсилає йому відповідь. У разі використання синхронної комунікації між мікросервісами, при такому сценарії задля забезпечення інших сервісів даними про нового користувача, сервіс користувачів повинен був би індивідуально для кожного сервісу надіслати запит з інформацією про кожного користувача, дочекатися відповіді від сервісу задля забезпечення цілісності даних у разі якогось збою іншого сервісу і лише після цього надіслати відповідь користувачу. При невеликій кількості залежних сервісів, малому об’єму даних та достатній обчислювальній потужності, такий підхід до комунікації між сервісами був би задовільним і мав би перевагу над асинхронним спілкуванням. У випадку асинхронного спілкування, зокрема використовуючи підхід автобусу подій, цей сценарій виконання завдання має інший вигляд. Після повної обробки даних про нового користувача сервіс користувачів надіслав би подію про створення нового користувача з необхідними даними про користувача до інших сервісів, яким потрібно зберігати у себе сутність користувачів і в подальшому обробляти. Вся логіка навколо обробки помилок та алгоритмі дій при нефункціонуванні якогось

сервісу лягає на сам автобус подій, в той час як сервіс користувачів після надсилання події матиме змогу відразу надіслати відповідь користувачу, а сервіси, яким необхідні дані про користувачів, оброблять у себе інформацію про нового користувача незалежно від сервісу користувачів відразу або як тільки знову почнуть функціонувати.

Попри такі переваги асинхронного спілкування між сервісами у мікросервісній архітектурі над синхронним способом, реалізація асинхронного спілкування значно складніша, ніж синхронного. Найбільшою проблемою асинхронного спілкування є проблема паралельності подій. У застосунках з великою базою користувачів одночасно може надходити велика кількість подій для обробки у хаотичному порядку. При стандартному підході до обробки асинхронних подій, застосунок наражається на критичне порушення цілісності даних, що може призвести до критичних проблем у бізнесі в цілому. Неможливо перелічити усі можливі сценарії випадків, у яких події приходитимуть не у правильному порядку. Найбільш типовим сценарієм таких випадків є імплементація самих бібліотек для використання подій, оскільки не є реальним забезпечити безвідмовно правильне надсилання подій у відповідному порядку, оскільки при багатьох одночасних подіях сервіс може просто не відслідкувати правильну чергу цих подій і надіслати їх у неправильному порядку. Також у зв’язку з можливими збоями у роботі самих мікросервісів, навіть при правильному порядку вхідних подій сам мікросервіс може не обробити подію і при його відновленні він почне обробку подій з неправильного порядку. У новій версії застосунку було використано підхід оптимістичного контролю паралельності, який полягає у тому, що у разі збою в обробці вхідної події, скасовується транзакція і подія заново відсилається в обробник подій для подальшого її повторного відсилання. У новій версії застосунку цей підхід було реалізовано за допомогою ведення версій кожного необхідного об’єкту у базі даних. Надіслані події містять відповідну версію об’єкту і при невідповідності версій буде скасована обробка події і вона буде надіслана назад до автобуса подій.

РОЗДІЛ 3: РОЗРОБКА НОВОЇ ВЕРСІЇ ЗАСТОСУНКУ

3.1 Контейнеризація застосунку та використання Kubernetes

3.1.1 Docker

Контейнеризація мікросервісів була реалізована за допомогою сервісу Docker. Кожен мікросервіс містить файл Dockerfile з налаштуваннями для побудови зображення сервісу (рисунок 3.1). За допомогою цього файлу вказується необхідне зображення середовища виконання, файли, які переносяться з локальної машини у контейнер і місце призначення на контейнери, куди вони будуть занесені, а також команда для підготовки контейнеру на етапі перед запуском і команда виконання у самому контейнері.

Після побудови зображень мікросервісів, кожен з них публікується на онлайн-сервісі DockerHub для подальшого їх використання в інструменті для керування контейнерами Kubernetes. Приклад інтерфейсу онлайн-сервісу DockerHub для розміщеного мікросервісу, який відповідає за надання рекомендацій користувачу, наведено у додатку А.

Dockerfile

```
1  FROM node:alpine
2
3  WORKDIR /app
4
5  COPY package.json .
6  RUN npm install
7  COPY . .
8
9  CMD ["npm", "start"]
```

Рисунок 3.1 – Приклад Dockerfile у сервісі автентифікації

3.1.2 Kubernetes

3.1.2.1 Використані об’єкти Kubernetes

Інструмент для керування контейнерами Kubernetes надає користувачу можливість створювати об’єкти різних типів для конкретних призначень.

При розробці нової моделі застосунку було використано Deployments, які слугують для налаштованого автоматичного керування подами у контексті Kubernetes, що у свою чергу містять контейнери, вказані користувачем. Приклад маніфесту для об’єкту Kubernetes типу Deployment вказано в додатку Б.1. За допомогою нього досягається автоматичне запускання реплік подів у кількості 1, вказуємо ім’я для середовища кластеру Kubernetes, а також налаштовуємо характеристики подів, яку будуть створені, такі як ім’я поди, її зображення, а також вказуємо змінні середовища для контейнера сервісу авторизації.

Для взаємодії подів з іншими елементами кластеру інструменту для керування контейнерами Kubernetes, також були створені об’єкти Kubernetes типу Service. Існують різні типи об’єктів Service, але в застосунку було використано 2: Cluster IP та Load Balancer. Об’єкт Service типу Load Balancer представляє собою вільно доступне рішення Ingress Nginx. За допомогою нього кластер інструменту Kubernetes, де знаходяться усі сервіси застосунку, може спілкуватися із зовнішніми джерелами, оскільки по стандарту увесь кластер та контейнери є повністю ізольованими. У маніфесті (додаток Б.2) для цього об’єкту Service типу Load Balancer було вказано назву хоста застосунку, а також правила, які вказують про перенаправлення запитів до відповідних об’єктів Service типу Cluster IP по вказаних шляхах. Для кожного об’єкту Deployment додатково створюються об’єкти Service типу Cluster IP, які надаюсь можливість сервісам комунікувати між собою всередині кластера, при цьому не відкриваючи доступ із зовнішнього середовища. У маніфесті для об’єктів Service типу Cluster IP вказується назва подів та порти, по яким буде відбуватися внутрішнє спілкування (рисунок 3.2).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: consultations-srv
spec:
  selector:
    app: consultations
  type: ClusterIP
  ports:
    - name: consultations
      protocol: TCP
      port: 3000
      targetPort: 3000
```

Рисунок 3.2 – маніфест об’єкту Service типу Cluster IP для сервісу консультацій

3.1.2.2 Skaffold

Допоміжний інструмент для автоматизації запуску, будуванню та спостеріганню за змінами у коді у зображеннях контейнерів Skaffold був використаний у розробці нової версії застосунку задля пришвидшення процесу запуску усіх необхідних об’єктів інструменту для керування контейнерами Kubernetes. Також важливою складовою цього рішення є автоматичне видалення усіх запущених об’єктів інструменту Kubernetes після зупинки команди запуску інструменту Skaffold, оскільки робота з великою кількістю запущеними контейнерами, об’єктами Kubernetes, а також віртуальної машини самого інструменту Docker потребує значних обчислювальних можливостей робочої машини і за допомогою автоматичної зупинки та видалення непотрібних елементів допомагає пришвидшити процес розробки в цілому. У маніфесті (рисунок 3.3) для налаштування інструменту Skaffold вказується шлях до маніфестів об’єктів інструменту Kubernetes, а також вказуються артефакти – елементи масиву, які відповідають за запуск елементів інструменту Kubernetes, а також містять у собі назву зображення контейнера, файл Dockerfile, необхідний для будування зображень контейнерів і вказується шлях для спостерігання за змінами у файлах, щоб забезпечити гаряче перевантаження усіх запущених контейнерів та об’єктів інструменту Kubernetes.

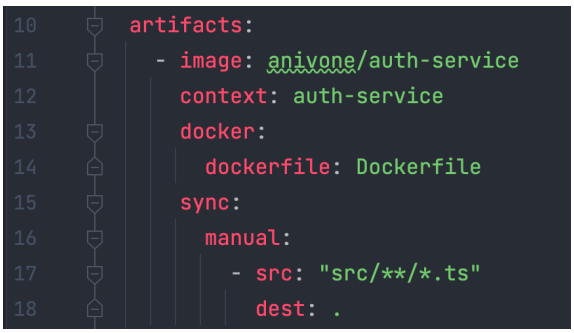


Рисунок 3.3 – Приклад написання артефакту для сервісу автентифікації у маніфесті для інструменту Skaffold

3.2 Спільна бібліотека NPM

При розробці нової версії застосунку було здійснено рішення створити спільну бібліотеку NPM. Мотивацією такого рішення було запровадження фіксованих елементів коду для усіх мікросервісів застосунку, оскільки імпортування локальних файлів з одної точки доступу для різних мікросервісів порушило б принцип незалежності кожного сервісу у мікросервісній архітектурі. Структуру спільної бібліотеки NPM у застосунку можна побачити у папці під назвою «common» на рисунку 3.4. На поточній версії застосунку було прийнято рішення винести абстрактні класи для слухачів та видавців подій, обгортка навколо клієнту для автобусу подій задля зручного користування у контексті Express.js, допоміжні функції, а також інтерфейси подій та перераховування назв предметів для автобусу подій. Всі зміни до спільної бібліотеки публікуються у публічні організації на офіційному сайті NPM з метою використання спільної бібліотеки за допомогою її імпортування у відповідних мікросервісах, де її елементи необхідні для роботи сервісів. Публікація змін відбувається у скрипті у файлі package.json, при якому усі зміни вносяться до інструменту контролю версій Git, змінюється версія бібліотеки, проводиться будування проекту і подальша його публікація.

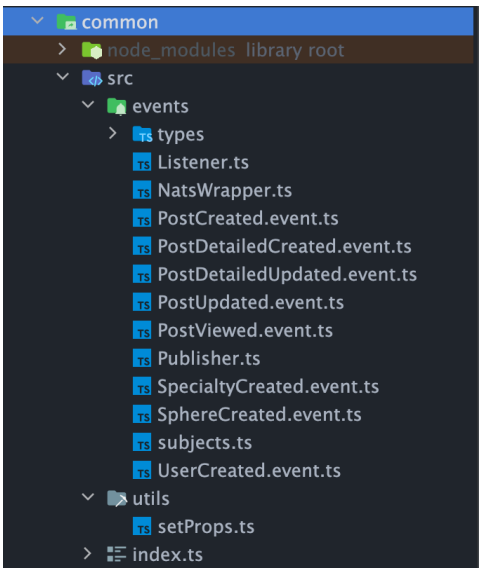


Рисунок 3.4 – Вміст спільної бібліотеки NPM у застосунку

3.3 Реалізація асинхронного спілкування

3.3.1 NATS Streaming Server

Інструмент для реалізації обміну повідомленням NATS Streaming Server є досить новим рішенням і був обраний з метою познайомитися з його функціоналом, ефективністю та порівняти з іншими відомими рішеннями, наприклад, Kafka або RabbitMQ. Інструмент NATS Streaming Server був інтегрований у нову версію застосунку за допомогою створення об’єктів Kubernetes типу Deployment і Service, а також завдяки використанню вільно доступного рішення у вигляді зображення на онлайн-сервісі DockerHub. Інструмент NATS Streaming Server налаштовується за допомогою написання маніфестів, приклад яких зображено на рисунку 3.5. При заданні шаблону контейнеру, який буде створено автоматично за допомогою об’єкту типу Deployment у контексті Kubernetes, вказуються аргументи для налаштування самого інструменту NATS Streaming Server. Серед них вказується значення для портів, налаштування поведінки перевірки дієздатності сервісів, які підключені до NATS Streaming Server, а також назву кластера застосунку у контексті NATS Streaming Server.

При розробці нової версії застосунку було використано бібліотеку «nats-streaming-server» для взаємодії з інструментом NATS Streaming Server. З метою зручного користування функціоналом бібліотеки, а також для правильного структурованого його використання у застосунку в контексті Express.js, було розроблено обгортку навколо бібліотеки (додаток Б.3). За допомогою цієї обробку реалізовано єдиний екземпляр класу NatsWrapper, який дозволяє використовувати одну і ту саму змінну клієнта інструменту NATS Streaming Server під назвою Stan. Змінна ініціалізується за допомогою методу connect, який приймає аргументами необхідні для бібліотеки nats-streaming-server налаштування для здійснення підключення до інструменту NATS Streaming Server.

```
spec:
  containers:
    - name: nats
      image: nats-streaming:0.17.0
      args:
        [
          '-p',
          '4222',
          '-m',
          '8222',
          '-hbi',
          '5s',
          '-hbt',
          '5s',
          '-hbf',
          '2',
          '-SD',
          '-cid',
          'consultation-platform',
        ]
```

Рисунок 3.5 – Приклад маніфесту для налаштування інструменту NATS Streaming Server

3.3.2 Імплементація слухачів та видавців подій

Для слухачів та видавців подій були розроблені відповідні абстрактні класи Listener (додаток Г.1) та Publisher (додаток Г.2). Вони входять до спільної бібліотеки NPM під назвою «common».

Слухачі мають містити всі необхідні налаштування для виклику методу listen клієнта Stan бібліотеки nats-streaming-server. Також при наслідуванні

абстрактного класу Listener, нащадки мають визначити логіку обробки подій у методі onMessage. За допомогою узагальненого класу і наслідуванню типу від інтерфейсу Event досягається безпечність типів та уникнення можливих помилок у даних при компіляції. У кожному нащадку абстрактного класу слухача реалізується метод onMessage (рисунок 3.6, а), який відповідає за здійснення певної логіки у момент отримання події. Ключовим елементом у цьому методі є виклик методу екземпляру класу Message бібліотеки nats-streaming-server під назвою ack, який повідомляє інструменту NATS Streaming Server, що подія була оброблена успішно і інструмент може видаляти копію події, яка знаходиться у нього в пам'яті. Цей процес є надзвичайно важливим у вирішенні проблеми паралельності подій та імплементації підходу Optimistic Concurrency Control.

Абстрактний клас видавця подій визначає у собі необхідні налаштування для здійснення публікацій подій до інструменту NATS Streaming Server. При наслідування абстрактного класу видавця потрібно вказати назву предмету для відправки події (рисунок 3.6, б). Для безпосереднього надсилання події необхідно викликати метод екземпляру класу видавця publish та передати дані допустимого типу.

```
10  export class PostCreatedListener extends Listener<PostCreatedEvent> {
11      subject: PostCreatedEvent["subject"] = Subjects.PostCreated;
12      queueGroupName: string = queueGroupName;
13
14      async onMessage(data: PostCreatedEvent["data"], msg: Message) {
15          const post = PostModel.build(data);
16          await post.save();
17
18          msg.ack();
19      }
20  }
```

(а)

```
export class PostCreatedPublisher extends Publisher<PostCreatedEvent> {
  subject: PostCreatedEvent["subject"] = Subjects.PostCreated;
}
```

(б)

Рисунок 3.6 – Приклади реалізації нащадків абстрактного слухача *(а)* та видавця *(б)*

3.3.3 Реалізація Optimistic Concurrency Control

Оскільки при розробці нової версії застосунку було здійснено рішення перейти від синхронного способу комунікації між мікросервісами до асинхронного, застосунок зіткнувся з проблемою паралельності подій, які несли критичні помилки цілісності даних застосунку. З метою вирішення цієї проблеми, було обрано підхід Optimistic Concurrency Control. Для запобігання обробки подій у неправильному порядку при асинхронному типу спілкування між мікросервісами було вирішено здійснювати ведення версій кожного екземпляру сутності бази даних MongoDB.

Для реалізації ведення версій було обрано бібліотеку mongoose-update-if-current, яка підключається до ORM Mongoose як плагін та здійснює автоматичне версіонування кожного об’єкту необхідних сутностей.

Для кожного об’єкту сутностей – документів у контексті MongoDB – було додано метод пошуку по колекції бази даних по ідентифікаційному коду і версії (рисунок 3.7). За допомогою нього, а також вищезгаданого методу екземплярів класу Stan бібліотеки nast-streaming-server інструменту NATS Streaming Server під назвою ack, здійснюється вирішення проблеми паралельності подій. У разі невідповідності версій сутності у сервісі та версії, переданої у події, сервіс не обробить успішно подію і не надішле інструменту NATS Streaming Server інформацію про те, що копію події можна буде видаляти зі свого сховища.

```
78  PostSchema.statics.findByEvent = async (event: {
79    id: string;
80    version: number;
81  }) => {
82    return PostModel.findOne( filter: {
83      _id: event.id,
84      version: event.version - 1,
85    });
86  };
```

Рисунок 3.7 – Імплементація методу findByEvent на прикладі сервісу постів

3.4 Сервіс рекомендацій

3.4.1 Content-Based filtering

При розробці нової версії застосунку, для сервісу рекомендацій було вирішено імплементувати 2 підходи до побудови рекомендаційних систем. Першим з них було реалізовано Content-Based filtering. Оскільки кожен корисний для рекомендацій елемент пов’язаний зі спеціальністю, було вирішено здійснювати аналіз релевантності елементів користувача по спеціальностях переглянутих елементів користувачем. Також в алгоритмі будується рейтинг релевантності кожної спеціальності для окремого користувача і рекомендовані елементи будуть надходити у спадаючому по релевантності списку.

3.4.2 Collaborative filtering

Також при розробці нової версії застосунку, для сервісу рекомендацій було реалізовано колаборативну фільтрацію. Рішення імплементувати обидва підходи було прийняте для можливої майбутньої їх комбінації для компенсації недоліків один одного. Колаборативна фільтрація також будує рейтинг релевантності інших користувачів відносно релевантності спеціальностей, переглянутих конкретним користувачем. Колаборативна фільтрація також надсилає результат у вигляді спадаючого по релевантності списку.

3.4.3 Підготовка тестових даних

При закінченні роботи об’єктів Deployment у середовищі Kubernetes, усі дані, які належали їм, також видаляються, тому з метою підготовки переліку необхідних сутностей та здійсненню необхідних дій для демонстрації результату рекомендацій, було написано допоміжний файл скрипт, який надсилає запити до застосунку та ініціалізує початковий стан баз даних у всіх сервісах. У таблиці 3.8 наведено стан початкових даних та вказується кількість постів певної спеціальності, переглянутих певним користувачем.

	Кількість переглядів користувачем 1	Кількість переглядів користувачем 2	Кількість переглядів користувачем 3
Web Development	3	3	0
Design	2	2	1
Martial Arts	0	0	2
Machine Learning	0	0	0

Таблиця 3.8 – Загальний вигляд початкових тестових даних

3.4.4 Результати рекомендацій

Сервіс рекомендацій надає 2 кінцеві точки для доступу до рекомендацій. Після ініціалізації початкових даних за допомогою файлу-скрипту, у першого користувача буде найбільший пріоритет для спеціальності «Web Development», після якої по пріоритету буде спеціальність «Design». Результат запиту на рекомендації типу Content-Based filtering можна побачити у додатку Д.1. Оскільки для користувача 1 першим пріоритетом буде «Web Development», а другим «Design», то алгоритм зробить припущення, що у користувача 2 найбільш схожі пріоритети, як у користувача 1, а у користувача 3 також буду схожі, але з меншим коефіцієнтом. Результати колаборативної фільтрації у додатку Д.2.

Висновки

У роботі було досліджена та описана суть поняття рекомендаційної системи, висвітлено ідею різних підходів, а також їхні переваги та недоліки. Також було проаналізовано можливість комбінації декількох підходів і можливі недоліки у такому рішенні.

Було досліджено модель та архітектуру застосунку попередньої версії. Здійснено аналіз щодо актуальності їх для завдання по впровадженню її у рекомендаційній системі. Наведено перелік змін та обґрунтовано причини, чому вони насправді потрібні і що принесуть новій версії застосунку.

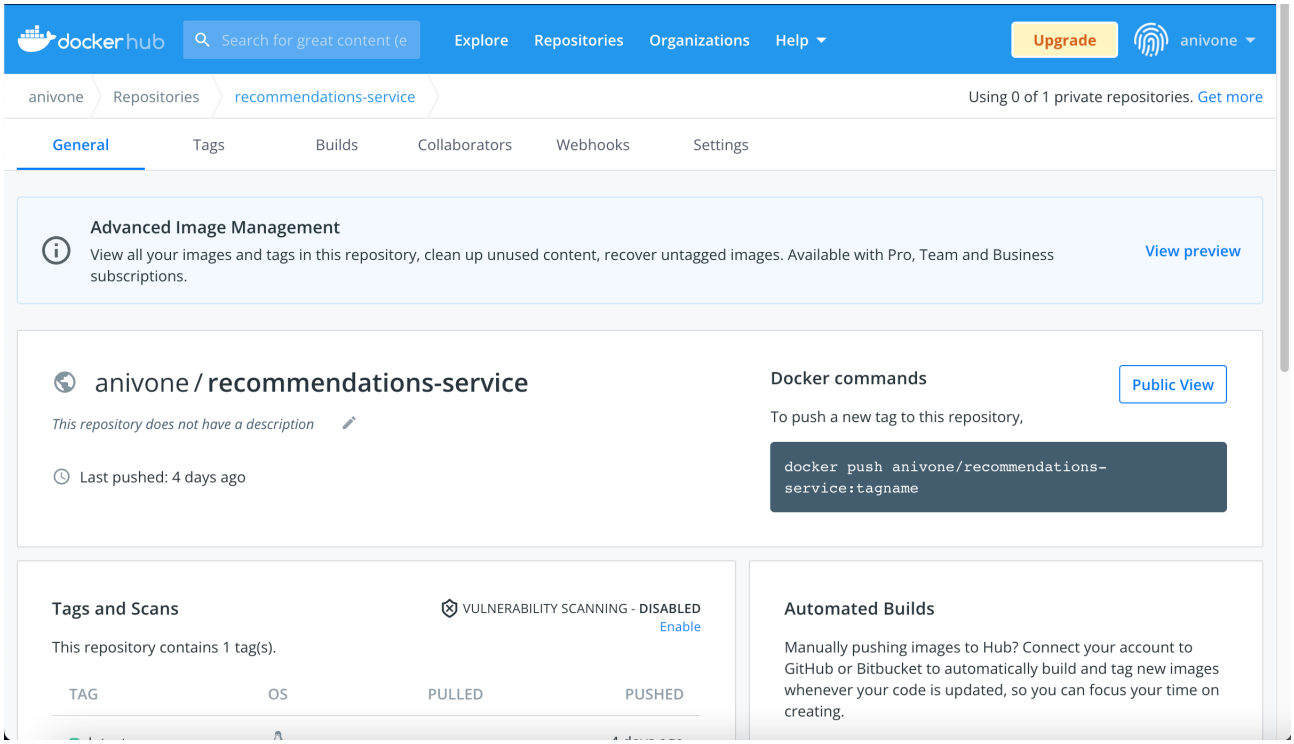
На основі дослідження необхідних змін старої версії, було розроблено застосунок нової версії, який містить усі проаналізовані рішення проблем старої версії. Було детально описано основні моменти розробки та впровадження оновлень застосунку.

Список використаних джерел

1. Ricci F. Introduction to Recommender Systems Handbook / F. Ricci, L. Rokach, B. Shapira // Recommender Systems Handbook / F. Ricci, L. Rokach, B. Shapira., 2011. – С. 11–35.
2. Isinkaye O. F. Recommendation systems: Principles, methods and evaluation [Електронний ресурс] / O. F. Isinkaye, O. Y. Folajimi, A. B. Ojokoh – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/pii/S1110866515000341.
3. Rajeev G. The Benefits of Containerization and What It Means for You [Електронний ресурс] / G. Rajeev, P. Szmrecsanyi. – 2019. – Режим доступу до ресурсу: https://www.ibm.com/cloud/blog/the-benefits-of-containerization-and-what-it-means-for-you.

Додаток А

Розміщене зображення рекомендаційного сервісу застосунку на онлайн-сервісі DockerHub



Додаток Б.1

Приклад маніфесту Kubernetes для сервісу автентифікації

auth-depl.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-depl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
    spec:
      containers:
        - name: auth
          image: anivone/auth-service
          env:
            - name: NATS_CLIENT_ID
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: NATS_URL
              value: 'http://nats-srv:4222'
```

Додаток Б.2

Налаштування об’єкту Service типу Load Balancer Ingress Nginx

ingress-srv.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-service
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  rules:
    - host: consultation.com
      http:
        paths:
          - path: /api/auth/?(.*)
            pathType: Prefix
            backend:
              service:
                name: auth-srv
                port:
                  number: 3000
          - path: /api/posts/?(.*)
            pathType: Prefix
            backend:
              service:
                name: posts-srv
                port:
                  number: 3000
          - path: /api/comments/?(.*)
            pathType: Prefix
            backend:
```

Додаток В

Імплементація обгортки навколо клієнта інструменту NATS Streaming Server

NatsWrapper.ts

```ts
import nats, { Stan } from "node-nats-streaming";

class NatsWrapper {
  private _client?: Stan;

  get client() {
    if (!this._client) {
      throw new Error("Cannot access NATS client before connecting!");
    }
    return this._client;
  }

  connect(clusterId: string, clientId: string, url: string) {
    this._client = nats.connect(clusterId, clientId, opts: { url });

    return new Promise<void>( executor: (resolve, reject) => {
      this.client.on( eventName: "connect", listener: () => {
        console.log("Connected to NATS!");
        resolve();
      });
      this.client.on( eventName: "error", listener: (err) => {
        reject(err);
      });
    });
  }
}

export const natsWrapper = new NatsWrapper();
```

Додаток Г.1

Імплементація абстрактного слухача подій

Listener.ts

```ts
interface Event {
  subject: Subjects;
  data: any;
}

export abstract class Listener<T extends Event> {
  abstract subject: T["subject"];
  abstract queueGroupName: string;

  abstract onMessage(data: T["data"], msg: Message): void;

  protected client: Stan;
  protected ackWait = 5 * 1000;

  constructor(client: Stan) {
    this.client = client;
  }

  subscriptionOptions() {
    return this.client
      .subscriptionOptions()
      .setDeliverAllAvailable()
      .setManualAckMode(true)
      .setAckWait(this.ackWait)
      .setDurableName(this.queueGroupName);
  }

  listen() {
    const subscription = this.client.subscribe(
      this.subject,
      this.queueGroupName,
      this.subscriptionOptions()
    );

    subscription.on( eventName: "message", listener: (msg: Message) => {
      console.log(`Message received: ${this.subject} / ${this.queueGroupName}`);

      const parsedData = this.parseMessage(msg);
      this.onMessage(parsedData, msg);
    });
  }

  parseMessage(msg: Message) {
    const data = msg.getData();
    return typeof data === "string"
      ? JSON.parse(data)
      : JSON.parse(data.toString( encoding: "utf8"));
  }
}
```

Додаток Г.2

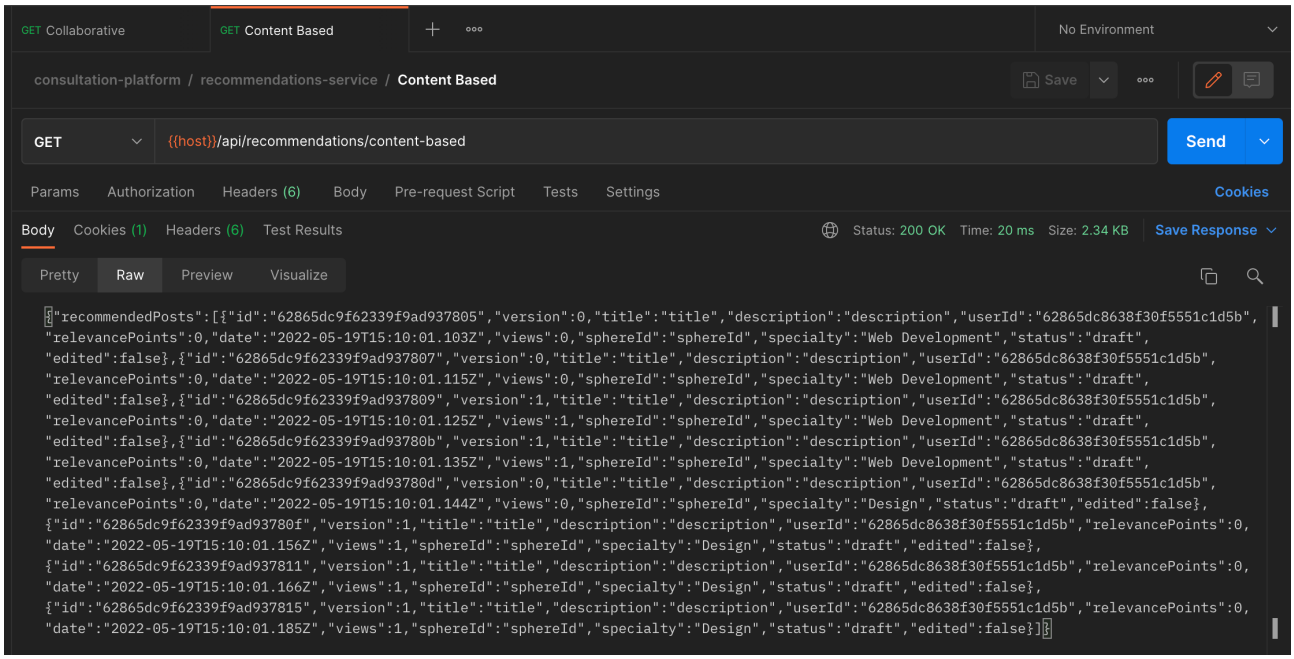
Імплементація абстрактного видавця подій

Listener.ts × Publisher.ts ×

```ts
import ...

interface Event {
  subject: Subjects;
  data: any;
}

export abstract class Publisher<T extends Event> {
  abstract subject: T["subject"];
  protected client: Stan;

  constructor(client: Stan) {
    this.client = client;
  }

  publish(data: T["data"]): Promise<void> {
    return new Promise( executor: (resolve, reject) => {
      this.client.publish(this.subject, JSON.stringify(data), callback: (err : Error | undefined ) => {
        if (err) {
          return reject(err);
        }
        console.log("Event published to subject", this.subject);
        resolve();
      });
    });
  }
}
```

Додаток Д.1

Результат запиту на отримання рекомендацій за підходом Content-Based filtering



Додаток Д.2

Результат запиту на отримання рекомендацій за підходом Collaborative filtering