Київський національний торговельно-економічний університет
Кафедра програмної інженерії та кібербезпеки

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА (ПРОЕКТ)

на тему:

«ПРОЕКТУВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ПОБУДОВИ КОРПОРАТИВНИХ WEB-ДОДАТКІВ»

Студента 2 курсу, 5 групи,
спеціальності 6.050103
“Програмна інженерія”
спеціалізації «Інженерія програмного забезпечення» _____ підпис студента — Александренко Андрій Анатолійович

Науковий керівник
кандидат технічниих наук,
доцент _____ підпис керівника — Рассамакін Володимир Якович

Гарант освітньої програми
доктор технічних наук,
професор _____ підпис керівника — Криворучко Олена Володимирівна

КИЇВ – 2019

Київський національний торговельно-економічний університет

Факультет *ФОАІС* Кафедра програмної інженерії та інформаційних систем
Освітній ступінь магістр
Спеціальність інженерія програмного забезпечення
Спеціалізація / освітня програма _____

Затверджую
Зав. кафедри Криворучко О.В.
«___» _____20__ р.

Завдання
на випускну кваліфікаційну роботу студентові

Александренко Андрію Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема випускної кваліфікаційної роботи
«Проектування мікросервісної архітектура побудови корпоративних web-додатків»

Затверджена наказом ректора від "7" листопада 2018 р. № 4183
2. Строк здачі студентом закінченого роботи (проекту)_____
3. Цільова установка та вихідні дані до роботи (проекту)

Мета роботи - *створення web-додатку з мікросервісною архітектурою використанням технологій середовища Docker*

Об'єкт дослідження - *проектування корпоративних web–додатків.*

Предмет дослідження - *створення web-додатку з мікросервісною архітектурою.*

4. Консультанти по роботі із зазначенням розділів, які консультують:

Розділ	Консультант (прізвище, ініціали)	Підпис, дата	
		Завдання видав	Завдання прийняв

5. Зміст випускної кваліфікаційної роботи (перелік питань за кожним розділом)

ВСТУП

РОЗДІЛ 1. ТЕХНОЛОГІЇ РОЗРОБКИ WEB-ДОДАТКІВ

1.1. Генезис Web-додатків та технології їх розробки.

1.2. Концепції розробки web-додатків

1.3. Проблеми під час розробки web-додатків

1.4. Висновки до розділу 1

РОЗДІЛ 2. ОСНОВНІ ПІДХОДИ І ТЕХНОЛОГІЇ РОЗРОБКИ МІКРОСЕРВІСНИХ WEB-ДОДАТКІВ

2.1. Опис підходу

2.2. Переваги підходу

2.3. Проблемні задачі розробки мікросервісних додатків

2.4. Середовище розробки

2.5. Тестування

2.6. Моніторинг системи

2.7. Висновки до розділу 2

РОЗДІЛ 3. РОЗРОБКА WEB-ДОДАТКУ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1. Мова програмування та фреймворк

3.2. Проектування архітектури додатку

3.3. Розгортування додатку в середовищі docker

3.4. Середовище для роботи додатку

3.5. Аналіз розробленого продукту

3.6. Висновки до розділу 3

ВИСНОВКИ ТА ПРОПОЗИЦІЇ

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

ДОДАТКИ

6. Календарний план виконання роботи

№ пор.	Назва етапів випускної кваліфікаційної роботи	Строк виконання етапів роботи	
		за планом	фактично
1	2	3	4
1.	*Вибір теми випускної кваліфікаційної роботи*	*25.09.2018*	
2.	*Розробка та затвердження завдання на проект магістра (Наказ №185 від 07.11.18)*	*07.11.2018*	
3.	*Вступ та перелік літературних джерел*	*28.02.2019*	
4.	*Розробка технічного завдання*	*22.03.2019*	
5.	*Розділ 1.* (*Назва розділу 1*)	*18.04.2019*	
6.	*Розділ 2.* (*Назва розділу 2*)	*24.05.2019*	
7.	*Розділ 3.* (*Назва розділу 3*)	*21.06.2019*	
8.	*Розділ 4.* (*Назва розділу 4, при необхідності, можливе об'єднання розділів 3 ти 4*)	*20.09.2019*	
9.	*Розробка програми та методики тестування*	*18.10.2019*	
10.	*Написання наукової статті*	*27.09.2019*	
11.	*Керівництво користувача*	*23.10.2019*	
12.	*Висновки та пропозиції*	*01.11.2019*	
13.	*Здача випускної кваліфікаційної роботи на кафедру (перша перевірка)*	*13.11.2019*	
14.	*Підготовка автореферату та презентації доповіді*	*14.11.2019*	
15.	*Попередній захист випускної кваліфікаційної роботи*	*14.11.2019 – 15.11.2019*	
16.	*Здача зброшурованої випускної кваліфікаційної роботи*	*25.11.2019*	
17.	*Зовнішнє рецензування випускної кваліфікаційної роботи*	*26.11.2019*	
18.	*Підготовка до публічного захисту випускної кваліфікаційної роботи*	*02.12.2019-04.12.2019*	

7. Дата видачі завдання **«26» вересня 2018 р.**

8. Науковий керівник випускної кваліфікаційної роботи

_____

(прізвище, ініціали, підпис)

9. Гарант освітньої програми _____

(прізвище, ініціали, підпис)

10. Завдання прийняв до виконання студент _____

(прізвище, ініціали, підпис)

11. Відгук керівника випускної кваліфікаційної роботи

____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____
____

Науковий керівник випускної кваліфікаційної роботи ____________________

(підпис, дата)

Відмітка про попередній захист ____

(ПІБ, підпис, дата)

12. Висновок про випускну кваліфікаційну роботу

Випускна кваліфікаційна робота студента ____________________________

(прізвище, ініціали)

може бути допущена до захисту екзаменаційній комісії.

Гарант освітньої програми ____

(прізвище, ініціали, підпис)

Завідувач кафедри ____

(підпис, прізвище, ініціали)

Анотація

Проведені дослідження по визначенню оптимальної архітектури для розробки корпоративних web-додатків, з урахуванням того фактору, що розроблений програмний продукт потребує постійної підтримки та оновлення в процесі його використання.

Проаналізовані та виділені переваги використання мікросервісного підходу. Показано, що кожний мікросервіс має власну апаратну базу і розгортується в окремій операційній системі незалежно один від одного. Це дає змогу використовувати будь-яку мову програмування та будь-який фреймворк в межах одного сервісу.

Здійснена програмна розробка корпоративного веб-додатку з мікросервісною архітектурою, в рамках якої досліджені базові функції автоматизатора розгортання контейнерів – docker. В ньому було створено 3 контейнери с базою даних MySql та двума мікросервісами. Встановлено, що обгрунтованим вибором буде мова програмування Java і фреймворк Spring з використанням менеджера автоматизації програмних продуктів Maven

Практична апробація результатів показала, що вибір мікросервісної архітектури є зваженим рішенням для розробки великих за розміром додатків, або ж додатків з великим навантаженням.

Annotation

Studies have determined that the optimal architecture for the development of enterprise web-applications, taking into account the fact that the developed software product requires constant support and updating in the process of its use.

Advantages of using microservice approach are analyzed and highlighted. The analysis has shown that each microservice has its own code and hardware base and deployed in a separate operating system independently of each other microservise. This allows using any programming language and any framework within a single service.

The software development of a corporate web application with microservice architecture has been performed, basic functions of the container deployment automator - docker has been explored. There are three created containers with MySql database and two microservices. It is determined that Java programming language and Spring framework using Maven software automation manager will be a reasonable choice for developing application based on the microservice architecture.

Practical validation of the results has shown that microservice architecture is a optimal solution for the development of large applications or high-load applications.

Технічне завдання

1. НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Мікросервісний Web-додаток.

Галузь застосування: інформаційні технології.

2. ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на ВКР, затверджене кафедрою програмної інженерії та інформаційних систем КНТЕУ.

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для аналізу переваг використання мікеросервісної архітектури при розробці корпоративних web-додатків на прикладі простої системи зберігання записів користувачів.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

1) використання мікросервісної архітектури при побудові додатку;
2) можливість розширення структури web-ресурсу;
3) комунікація між сервісами по HTTP протоколу;
4) використання бази даних;
5) розгортування додатку у середовищі docker.

Розробку виконано на мові програмування Java з використанням фреймворку Spring та менеджером середовищ docker.

5. ВИМОГИ ДО ПРОЕКТНОЇ ДОКУМЕНТАЦІЇ

У процесі виконання роботи повинна бути розроблена наступна документація:

1) пояснювальна записка;
2) програма та методика тестування;
3) керівництво користувача.

6. ЕТАПИ ПРОЕКТУВАННЯ

Вивчення літератури за тематикою роботи.

Розробки та узгодження технічного завдання.

Розробки структури web-додатку.

Розробки дизайну сторінок та графічних елементів.

Програмна реалізація web-ресурсу.

Тестування web-ресурсу.

Підготовка матеріалів текстової частини роботи.

Підготовка матеріалів графічної частини роботи.

Оформлення технічної документації роботи.

7. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Тестування розробленого програмного продукту виконується відповідно до “Програми та методики тестування”.

ЗМІСТ

					КНТЕУ 121– 05-01.МР	
Зм.	*Аркуш*	*№ докум.*	*Підпис*	*Дата*	*Проектування мікросервісної архітектура побудови корпоративних web-додатків*	*Стадія* / *Аркуш* / *Аркушів*: з / *2* / 51
Зав. каф.		Криворучко О.В.				
Керівник		Рассамакін В.Я.				
Гарант		Криворучко О.В.			*Зміст*	Факультет обліку, аудиту та інформаційних систем, 2м курс, 5 групи
Розробив		Александренко А.А.				

ВСТУП

Вплив глобальної комп'ютерної мережі Internet на сучасний світ не має історичних аналогів. Його сьогоднішній день - це початок епохи електронного проникнення в усі сфери людського життя, це щось більше, ніж просто маркетингова кампанія, це основа нової філософії і нової ділової стратегії.

Цілком логічно припустити, що і з точки зору реклами продукції або послуги Інтернет - найбільш значимий ресурс. Більшість сучасних людей користуються Інтернетом, як найбільш доступним джерелом інформації.

Web-технологія повністю перевернула уявлення про роботу з інформацією, та й з комп'ютером взагалі. Виявилося, що традиційні параметри розвитку обчислювальної техніки - продуктивність, пропускна здатність, ємність запам'ятовуючих пристроїв не враховували головного "вузького місця" системи - інтерфейсу з людиною. Застарілий механізм взаємодії людини з інформаційною системою стримував впровадження нових технологій і зменшував вигоду від їх застосування. І тільки коли інтерфейс між людиною і комп'ютером був спрощений до природності сприйняття звичайною людиною, послідував безпрецедентний вибух інтересу до можливостей обчислювальної техніки.

Створення Web-сайтів є однією з найважливіших технологій розробки ресурсів Internet. Хороший сайт, вбираючи в себе всю корисну інформацію, є найкращою візитною карткою і комерційної фірми і освітнього закладу, працюючи на них в будь-який час доби.

З кожним роком web-додатки стають все більшими та складними. Сьогодні важко знайти сучасний web-додаток, який би показував нові дані тільки після оновлення сторінки браузера чи не мав би хоча б однієї анімації. Браузерам

необхідно виконувати велику кількість операцій таких як відображення розмітки та підключення стилів та обробка інформації у фоновому режимі, зміна відображення елементів в залежності від дій користувача

Актуальність

Актуальність обраної теми полягає в дослідженні та визначенні найбільш досконалої та кращої архітектури корпоративних web-додатків, з урахуванням того фактору, що розроблений програмний продукт потребує постійної підтримки та оновлення після його виходу в світ. Метою власника продукту є отримання більшого прибутку за менших витрат - це можливо лише за правильно спроектованої архітектури додатку, так як легкість внесення змін до існуючих бізнес-функцій та простота додавання нових бізнес-функцій до існуючої системи є основними витратами при пітримці додатку після його релізу.

Метою роботи випускної кваліфікаційної роботи (проекту) є створення web-додатку з мікросервісною архітектурою використанням технологій середовища Docker.

Об‘єкт дослідження: проектування корпоративних web–додатків.

Предмет дослідження: створення web-додатку з мікросервісною архітектурою з допомогою мови програмування Java.

Завдання дослідження:

- дослідження існуючих технологій та рішень для створення додатків на базі мікросервісного підходу;
- обгрунтування вибору мікросервісної архітектури, розробка технічного завдання;
- проектування та створення корпоративного web-додатку в середовищі докер с використанням мови програмування Java;
- апробація та тестування розробленого програмного продукту;
- розробка інструкції користувача;

- створення технічного звіту;
- формулювання висновків та пропозицій;

Методи, методики і технології розробки:

В роботі використовуються:

- сучасні методи об’єктно-орієнтованого програмування
- мова програмування Java та фреймворки для цієї мови
- методи розробки мікросервісів,
- платформа для розробки та експлуатації додатків docker
- база даних MySql.

Результат роботи: web-додаток, побудований на мікросервісній архітектурі.

РОЗДІЛ 1
ТЕХНОЛОГІЇ РОЗРОБКИ WEB-ДОДАТКІВ

1.1. Генезис Web-додатків та технології їх розробки

Протягом всієї історії інформаційних технологій однією з найважливіших проблем була боротьба за повторне використання програмного коду. Першим засобом, що реалізує ідею повторно використання, були процедури і функції - спочатку вбудовані, а потім оформленні як модулі. Модуль являє собою одиницю програмного коду, що окремо компілюється, окремо зберігається та має власний інтерфейс. Інтеграція модулів у єдиний додаток реалізується завдяки механізмам виклику процедур в мовах програмування і механізмам збірки в завантажувачах і редакторах зв'язків. Розробка модулів, реалізує типові технологічні і прикладні функції, що значно полегшує роботу програміста, зменшує обсяг коду і прискорює процес розробки та впровадження програмного продукту, що є важливою і конкурентоздатною перевагою. Розвинені бібліотеки модулів, що надають майже вичерпний набір засобів для вирішення певних класів технологічних або прикладних задач, отримали назву каркасів. Застосування каркасів в ідеалі перетворює створення нового додатка в збірку програми з готових елементів каркаса.

Наступним значним етапом боротьби за повторне використання коду можна вважати появу компонентної архітектури. Компонента - це модуль, який реалізує стандартний інтерфейс. Стандартизація інтерфейсу забезпечує взаємозамінність компонентів і, що найважливіше, взаємодію компонент з проміжним програмним забезпеченням - каркасами, інструментальними засобами розробки,

контейнерами. Останнє кардинальним чином змінює процеси розробки та функціонування додатків, значно прискорюючи їх створення та впровадження. Контейнер являє собою деяке проміжне програмне забезпечення, яке підтримує виконання компонентів. Кажуть, що, реалізуючи стандартний інтерфейс, компонент укладає контракт з контейнером: компонент 12 «зобов'язується» забезпечувати деяку певну поведінку (виконує стандартні методи), а контейнер «за це» надає йому підтримку низького рівня, наприклад, управління життєвим циклом компоненти (певна послідовність виклику її методів), безпека, зв'язок між компонентами, управління транзакціями тощо.

Виконання контейнером технологічних функцій низького рівня позбавляє програміста від їх реалізації і дозволяє йому зосередити свої зусилля на прикладній задачі. Особливе значення контрактні відносини між компонентами і контейнером набувають в технологіях розподілених додатків. Частини розподіленого додатку, що виконуються в різних вузлах розподіленої системи, є компонентами, і їх функціонування і взаємодія, що підтримується відповідним контейнером. Інтеграційну функцію, таким чином, виконує контейнер, що підтримує той чи інший тип контракту. Найбільш розвиненими платформами розподілених додатків є Microsoft .NET і Java 2 Enterprise Edition (J2EE). Остання базується на відкритих стандартах і не залежить від апаратної платформи і операційного середовища. Основним виробником засобів технології .NET є фірма Microsoft. Технологія J2EE має більший спектр виробників, в силу відкритості своїх стандартів. Лідерами є фірми Oracle, IBM, Sun, Red Hat, багато засобів цієї технології поширюються вільно або мають відкритий код.

Хоча компонентна архітектура забезпечує швидку і зручну інтеграцію компонентів, це стосується тільки компонентів, які були розроблені у відповідності зі специфікаціями даної платформи. Однак сучасні вимоги ринку надзвичайно посилюють вимоги до термінів створення і впровадження систем і додатків інформаційних технологій. Починаючи з середини 1990-х років відзначається безліч випадків, коли розробка і реалізація проекту інформаційної системи «з нуля»

закінчувалася крахом, тому що, по-перше, витрати на розробку не виправдовували себе, по-друге, до моменту завершення проекту, умови, на підставі яких були складені специфікації проекту, істотно змінювалися.

Необхідність швидкого реагування підприємств на мінливі умови привели фірму IBM до концепції бізнесу за вимогою. За визначенням IBM, бізнес за вимогою (on demand business) - це підприємство, бізнес-процеси якого щільно 13 інтегровані всередині всієї компанії, а також з бізнес-процесами партнерів, постачальників, споживачів, що дає підприємству можливість гнучко і швидко реагувати на мінливі вимоги покупців, зміни ринку і зовнішні впливи. Інформаційні системи, що підтримують бізнес за вимогою, повинні швидко відповідати вимогам того бізнесу, який вони обслуговують. Інтеграція процесів усередині підприємства має на увазі, що окремі процеси підприємства вже автоматизовані, в розробці яких, могли використовуватися різні мови програмування, операційні системи та платформи проміжного програмного забезпечення. Розробка всіх або деякої частини додатків наново відповідно до єдиної прийнятої платформи є занадто довгою і дорогою справою. Підприємство має максимально використовувати наявні активи інформаційних технологій, щоб знизити витрати і зменшити терміни впровадження. Також при розробці нових або модифікації наявних додатків доцільно максимально використовувати вже існуючі компоненти. При інтеграції з інформаційними системами партнерів, постачальників, споживачів тощо, приведення всіх компонентів системи до єдиній платформи просто неможливе. Необхідність вирішення цих проблем заснувала концепцію сервісно-орієнтованої архітектури.

Програмні засоби, що реалізують компоненти бізнес-процесів, оформляються як сервіси. Сервіс - це програмний компонент, який реалізує закінчену функцію надання або обробки даних, що переводить їх з одного цілісного стану в інший. Основною відмінністю сервісу від звичайної компоненти є стандартний і переносний незалежний інтерфейс. Клієнт, що звертається до сервісу, не зобов'язаний нічого знати про подробиці реалізації сервісу: якою мовою

і якою моделі програмування він створений, на яких апаратних засобах, в якому операційному середовищі, на якій платформі проміжного програмного забезпечення він виконується. Сервісно-орієнтована архітектура, таким чином, дозволяє компонувати бізнес-процеси з компонентів, виконаних на різних платформах (корпоративних J2EE, компонентів .NET, окремих додатків), представляти у вигляді сервісів і повторно використовувати в нових бізнес-процесах успадковані компоненти. В останньому випадку 14 успадкуванні компоненти не змінюються, але для них робиться оболонка, що адаптує їх інтерфейси до стандартів сервісів.

Необхідно звернути увагу на те, що оформлення процесу як сервісу не має будь-яких вимог до розробки процесу, а лише вимоги до його інтерфейсу. Щоб успішно виконувати свої інтегрують функції, платформа, яка втілює сервісноорієнтовану архітектуру, повинна відповідати таким вимогам: • забезпечувати специфікації інтерфейсу, які були б прийняті, якщо не всіма, то більшістю розробників компонентів; • використовувати загальноприйняті протоколи для взаємодії сервісів і клієнтів; • не використовувати в інтерфейсі які-небудь складні та / або закриті формати представлення інформації; • не вимагати для своєї підтримки дорогого або ресурсномісткого програмного забезпечення.

1.2. Концепції розробки web-додатків

Існує лише декілька підходів до розробки.

Монолітний підхід є найстарішою моделлю проектування ПО оскільки саме з неї і почалася розробка всього програмного забезпечення. В рамках даного підходу складної структури web-додатки як такої може не бути: сервер зберігає всю бізнес-логіку, а база даних - дані необхідні серверу для роботи [1].

Як правило подібні програми не відрізняються складністю в розробці і її великою вартістю на ранніх етапах, коли список необхідного функціоналу не відрізняється великою кількістю рядків, а помилкові дії досить просто виправляються на ранніх етапах. Нова функціональність додається легко і швидко.

Однак з плином часу або в поспіху за датою випуску цілком вірогідне зростання ризику припуститися помилки, яка перетворюється в «технічний борг», який необхідно буде виправити після випуску продукту. Однак з плином часу при монолітному підході, таких «помилок» може накопичитися дуже багато і таким чином, підтримувати монолітну систему в довгостроковій перспективі дуже складно і дорого, тому що команди розробників можуть змінюватися і виходить так, що простіше зробити «латочку», ніж розібратися як працює та чи інша частина програми.

Проблеми з підтримкою монолітного додатки можна пояснити і тим, що рано чи пізно невеликий список функціоналу в системі поповнюється новими вимогами та ідеями, які вимагаю внесення змін до вже існуючого код, що вже відбивається в останньому реченні попереднього абзацу. Крім того, в рамках web-додатки однією з проблем «моноліту» є масштабованість. Всі складові системи розташовані в одній точці, а тому потужність самої точки повинна бути відповідною для підтримки сервера і бази даних в робочому стані.

Приклад монолітної архітектури можна знайти в будь-якому додатку, навіть включає в себе різні класи. Варто відзначити, що в цей момент може здатися, що додаток в цей момент ставати модульним, проте все зовсім не так. Монолітний додаток по праву може включати модулі, однак, вони повністю або частково залежать один від одного.

Модульна архітектура на відміну від монолітної має на увазі розбиття всього функціоналу програми на окремі модулі, кожен з яких відповідає за певну частину функціоналу програми таким чином, що модульну архітектуру можна описати як сукупність безлічі монолітних модулів всередині однієї програми.

Кожен модуль програми є функціонально незалежним від іншого, а тому його застосування в різних ділянках коду не викликатиме збоїв в працездатності, по суті це надає простоту рефакторінга і перенесення меж модуля.

Варто відзначити, що при зміні одного модуля, інші модулі порушені не будуть, а тому спрощується завдання дебагу. Все, що потрібно, це підлаштувати

інші модулі, що використовують функціонал зміненого модуля під нові можливості і вимоги останнього, якщо зміни в ньому можуть призвести до збою в роботі програми.

Крім того, всі модулі взаємодіють один з одним в синхронному порядку, оскільки знаходяться на одній апаратній частині, а при необхідності для модуля можна виділити окрему базу даних або ж частина даних, з якими цей модуль буде працювати.

Прикладом можуть виступати сучасні PHP фреймворки застосовуються для швидкої розробки web-додатків. В даних фреймворками кожен створений клас-модуль має функціоналом конкретної спрямованості і ніяк не взаємодіє безпосередньо з тією частиною програми, для якої не був призначений.

Прикладом модуля може виступати клас Router, який би використовувався для визначення оброблюваних в додатку маршрутів. Будь-які не зазначені маршрути приводили б користувача на сторінку з кодом помилки 404.

Варто відзначити, що розробка додатків на основі модульної архітектури на перших етапах вже трохи вище за вартістю, ніж розробка аналогічного монолітного додатки, оскільки при внесенні нового функціоналу завжди повинен вставати питання про те, який конкретний модуль буде реалізовувати цей функціонал. Однак зростання вартості утримання подібних додатків не такий різкий, що дозволяє підтримувати дані додатка досить довго.

Сервіси є окремі цілком самодостатні модулі, що володіють власною апаратною базою, а саме розташовуються на окремому сервері. Крім того, вони можуть володіти власною базою даних, а оскільки вони розташовуються на окремих апаратних пристроях, навіть якщо віртуально, то і взаємодія між сервісами здійснюється асинхронно, що може надати як певні переваги, так і деякі недоліки. Однак, одним з ключових переваг SOA це те, що вона надає можливість незалежного масштабування компонентів, маючи на увазі збільшення потужностей тільки того сервісу, який цього потребує, не зачіпаючи апаратну частину інших.

Можливість розробки цілісного додатки сервіс-орієнтованим підходом, надає можливість розробки сервісів на різних мовах програмування. Простий приклад: сервіс доставки повідомлень може бути написаний на NodeJS, де мовою програмування використовується JavaScript, в той час як основна програма, що використовує цей сервіс, може складатися з зв'язки React на клієнтської частини і PHP на серверної частини. Для взаємодії цих двох частин необхідно тільки правильно використовувати програмний інтерфейс сервісу в основній частині програми.

Розробка сервісів відбувається окремо один від одного, а тому зміни в коді одного сервісу буде впливати тільки на нього, якщо тільки не змінилася видача даних в програмному інтерфейсі сервісу, що відбувається вкрай рідко, оскільки з форматом виведення визначаються ще на перших етапах розробки сервісу.

Однак розробка програми сервіс-орієнтованим підходом не настільки проста. Справа в тому, що необхідно строго і чітко окреслити межі функціоналу того чи іншого сервісу. Крім того, оскільки кожен сервіс розробляється на окремій апаратної частини, то перевірити компілятором того чи іншого сервісу передані дані з іншої частини програми буде неможливо, оскільки схеми взаємодії між сервісами додатки в явному вигляді не існує і сервіс знає тільки про самого себе. Знати про інших сервісах він може тільки те, що ми самі вказали, а тому обробка помилок - це також додаткова проблема, якої варто приділити увагу при розробці додатків сервіс-орієнтованим підходом, бо всі скоєні помилки при використанні архітектурного підходу можуть серйозним чином позначитися на всій працездатності системи. Для підтвердження того, що до сервіс-орієнтованого підходу потрібно підходити з особливою обережністю далі наведено приклад.

В одній фінансовій установі несправно відпрацьовував сервіс оплати. Клієнт оплачує курси, а сервіс, який надає ці курси, отримує повідомлення від фінансової установи - «гроші були переведені», і надає доступ клієнту до курсів, однак, на одному з етапів, один з сервісів, який обробляв переказ грошей відмовив з невідомої причини, що призвело до відкоту транзакцій і гроші повертаються на

рахунок програміста, що призвело до того, що клієнт отримав доступ до курсів, при цьому сервіс, який надав їх клієнту, гроші за це не отримав.

В рамках сервіс-орієнтованого підходу особливого поширення набув термін «оркестровки», який передбачає автоматичне розміщення, координацію і управління складними комп'ютерними системами і службами. Оркестровка описує те, як різні сервіси повинні взаємодіяти один з одним, використовуючи для цього обмін повідомленнями, включаючи бізнес-логіку і послідовність дій. Одним з основних орієнтирів у виборі архітектурного підходу є вартість підтримки програми при тому чи іншому підході.

З сказаного вище можна зробити висновок, що монолітний підхід надає досить швидкий старт і вимагає найменше витрат, однак вартість обслуговування додатків, розроблених на основі даного підходу зростає досить швидко.

Модульний підхід вимагає певних витрат на старті, однак зростання вартості обслуговування не такий різкий, що дозволяє підтримувати дані системи набагато довше. Нарешті, SOA-підхід вимагає серйозних витрат на старті, що включає в себе досить комплексне планування того як буде розвиватися система. На підставі всього сказаного можна сказати, що кожен архітектурний підхід дає свої переваги і недоліки.

Монолітний підхід дає непоганий старт, він дозволяє почати з низької вартості розробки, а також прощає помилки на початкових етапах розробки: все можна швидко змінити і поправити. Однак подібні додатки мають тенденцію зростати в ціні підтримки швидше ніж аналоги, розроблені на основі інших архітектурних підходів. При цьому, межа між монолітним і модульним додатком з першого погляду не завжди видно, однак вона інтуїтивно зрозуміла: з одного боку модульний додаток пишеться модулями з самого початку, кожен модуль має строго описаний функціонал з заданими межами і його можна застосовувати в різних ситуаціях, з іншого боку монолітне додаток може складатися з різних класів і компонентів, які будуть залежати один від одного і від даних якими володіють

інші модулі, крім того дані компоненти будуть частково реалізовувати функціонал вже існуючий в іншій частині програми, але за іншою схемою.

Сервіс-орієнтований підхід досить дорогий на перших етапах, однак при правильному підході дозволяє заощадити величезні фінансові вкладення на наступних етапах. Однак цей шлях варто вибирати великим компаніям, які мають достатню кількість ресурсів для проведення точних розрахунків і планування того, як буде розвиватися надалі їх розробка.

На рис. 1.2. зображено схеми комунікацій у додатках з монолітним та мікросервісним підходами. Добре видно, що всі комунікації у монолітномі підході йдуть через 1 сервес, проте при мікросервісному підході цих серверів 4, що повищує максимальну кількість оброблених запитів до додатку. Якщо ж щось трапиться з будь-яким елементом архітектури у монолітномі підході – не буде працювати весь додаток, проте в такій же ситуації в мікросервісній архітектурі, система не зазнає великого впливу.

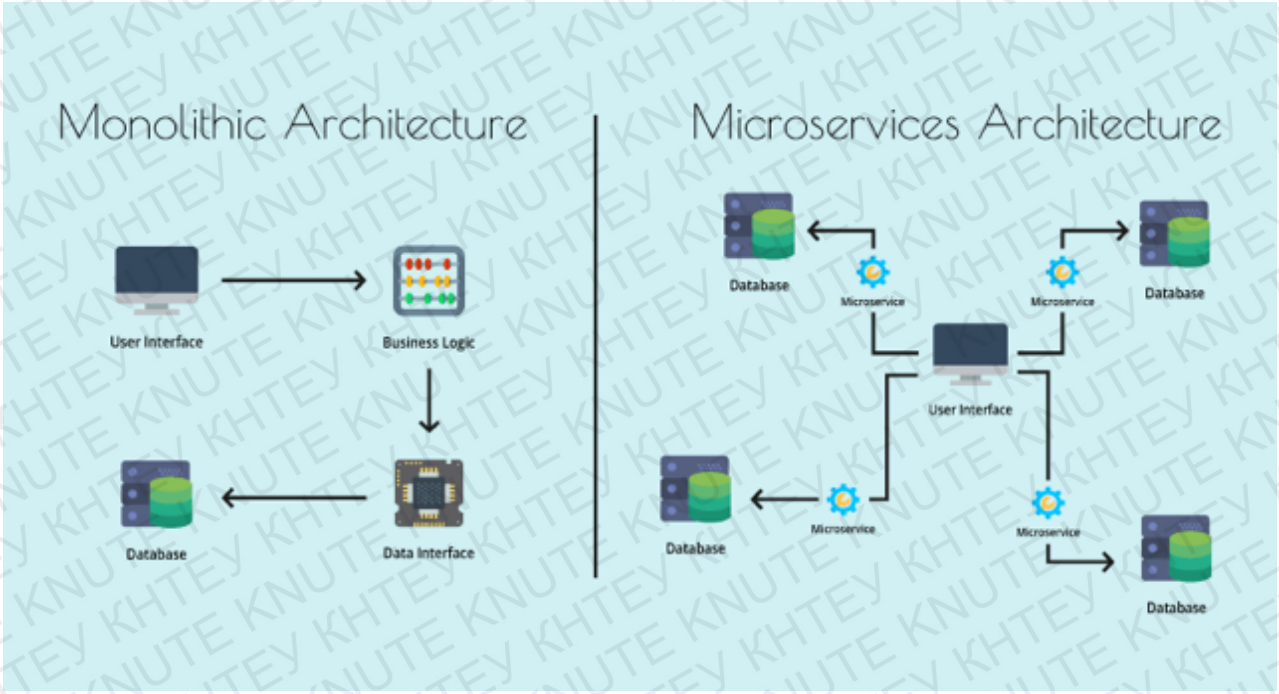

Рис. 1.2 Схема комунікацій у монолітній та мікросервісній архітектурах.

Таблиця 2.1 містить порівняльні характеристики монолітної та мікросервісної архітектур.

Таблиця 2.1 Порівняння мікросервісної та монолітної архітектур

Порівняння мікросервісної та монолітної архітектур

Мікросервісна архітектура	*Монолітна архітектура*
Кожний компонент у додатку маленький за розміром	Величезний розмір кодової бази
Маленькі сервіси швидші	Займає більше часу
Навіть якщо 1 сервіс не працює, це не впливає на інші	Якщо не працює хоч щось – не працює все
Всі сервіси не зв'язані між собой і децентралізовані	Всі сервіси(внутрішні) пов'язані між собою
Скейлінг дуже простий – можно збільшити кількість окремо взятих сервісів	Скейлінг можливий лише за умови підняття ще одного додатку
Може використовуватися багато мов програмування мають необхідності	Використовується лише 1 мова програмування
Команди що працюють над всім додатком не мають знати про всі сервіси, так як розроблюють лише свій	Всі команди, що працюють мають знати багато про додаток

1.3. Проблемні задачі розробки мікросервісних додатків

Веб-розробка прискорюється агресивно. Популярні і зручніші інтерфейси користуються попитом. Що стосується розробки успішного web-додатку, існує ряд факторів, що визначають цей успіх. Клієнти хочуть знати різні аспекти вашого продукту, такі як вартість, зовнішній вигляд та співвідношення ціни та якості. Щоб дізнатися про деталі компанії, клієнти можуть завітати на веб-сайт компанії, мобільні додатки та платформи соціальних медіа. Таким чином, важливо, як ви

взаємодієте та реагуєте на клієнтів. Можна виділити наступні моменти, на яких потрібно зосередити увагу при розробці web-додатку:

- *інтерфейс користувача і досвід користувача.* Десятиліття тому Інтернет був зовсім іншим місцем. Смартфонів не існувало. Простіший web-додаток, орієнтований на клієнтів, зараз дуже очікуваний. Іноді найменші елементи інтерфейсу роблять найбільший вплив. В епоху смартфонів веб-сайти повинні бути досить чуйними на менших екранах. Якщо веб-програми не задовольняють користувача, важко зберегти його лояльність клієнта до веб-сайту. Навігація по веб-сайту - ще одна частина, якою розробники часто нехтують. Інтуїтивна навігація створює кращий досвід користувача для відвідувача веб-сайту. Інтуїтивна навігація приводить аудиторію до інформації, яку вони шукають, без навчання. І коли навігація є інтуїтивно зрозумілою, відвідувачі можуть без особливих зусиль знаходити інформацію, створюючи бездоганний досвід, який заважає їм відвідувати конкурентів.

- *масштабованість*. Масштабованість - це ні продуктивність, ні користь використання обчислювальної потужності та пропускної здатності. Йдеться про врівноваження навантаження між серверами, отже, коли навантаження збільшується, можуть бути додані додаткові сервери, щоб збалансувати це. Не можно просто кидати все навантаження на один сервер, але треба спроектувати програмне забезпечення таким чином, щоб воно могло працювати на кластері серверів. Мікросервісна архітектура може допомогти покращити масштабованість, коли додається все більше серверів. Підхід дає можливість легко змінюватись. Мікросервісна архітектура - це дизайн, в якому компоненти додатків надають послуги іншим компонентам через протокол зв'язку, в основному по мережі.

- *продуктивність*. Як правило, прийнято, що швидкість web-сайту має головне значення для успішного web-сайту. Коли бізнес розміщений в Інтернеті, рахується кожна секунда. Повільні web-додатки - це збій. Як результат, клієнти покидають web-сайт, завдаючи таким чином шкоди доходам та репутації бізнесу. Кажуть, подумайте про ефективність спочатку, перш ніж розробляти web-додаток.

Деякі проблеми щодо продуктивності - це погано написаний код, неоптимізовані бази даних, некерований ріст даних, стрибки трафіку, поганий розподіл навантаження, конфігурація за замовчуванням, проблемні послуги сторонніх служб тощо. Мережа розповсюдження контенту (CDN) - це глобально розподілена мережа проксі сервери, розгорнуті в декількох центрах обробки даних. Це означає, що замість одного веб-сервера для web-сайту використовувати мережу серверів. Однією з переваг CDN є те, що запити на сервері будуть перенаправлені на різні сервери, що врівноважують трафік, файли розділені на різні CDN, тому не буде черги і чекати завантаження різних файлів, таких як зображення, відео, текст тощо.

- *безпека.* У час розпалу дизайну та досвіду користувачів безпекою web-додатків часто нехтують. Але безпеку слід враховувати протягом усього життєвого циклу розробки програмного забезпечення, особливо коли програма стосується життєво важливої інформації, такої як платіжні дані, контактна інформація та конфіденційні дані. Існує багато речей, які слід враховувати, якщо мова йде про безпеку web-додатків, такі як відмова в сервісних атаках, безпека даних користувачів, несправності в базі даних, несанкціонований доступ до обмежених частин web-сайту тощо, Фішинг, підробка між сайтами, введення снарядів, викрадення сеансу, ін'єкція SQL, переповнення буфера тощо. Web-сайт повинен бути ретельно спроектований і захищений, щоб бути недоступним для цих проблем безпеки.

- *знання платформи, мови та фреймворку розробки.* Фреймворк - це полегшення використання для мов розробки: вони підвищують продуктивність, пропонують бібліотеки кодування та розширюють можливості, тому розробникам не потрібно робити web-додатки, що кодують вручну, з нуля. Фреймворки пропонують такі функції, як моделі, API, фрагменти коду та інші елементи для розробки динамічних web-додатків. Деякі фреймворки мають жорсткий підхід до розвитку, а деякі - гнучкі. Найпоширеніші приклади веб-фреймів - PHP, ASP.Net, Ruby on Rails та J2EE. Веб-платформи пропонують бібліотекам клієнтів будуватись на існуючих структурах, необхідних для розробки web-додатку чи веб-сайту. Нова

функціональність може бути додана через зовнішній API. Розробники та власники малого бізнесу повинні чітко розуміти потреби своєї компанії, пов’язані з розробкою web-сайтів та додатків. Для доставки інформації та присутності в Інтернеті потрібна буде проста веб-платформа, така як WordPress або Squarespace, але для продажу цього продукту потрібна платформа електронної комерції, наприклад Magento, Shopify. WooCommerce або BigCommerce). Вибираючи ідеальну платформу, слід також враховувати технічні навички, криву навчання, ціноутворення, варіанти налаштування та аналітику.

1.4. Висновки до розділу 1

Тож вибираючи архітектуру треба звертати увагу на багато параметрів. Одним з найважливіших є розмір або величина web-додатку, так як від цього залежить вибір. Для маленьких додатків кращим вибором буде монолітний підхід, оскільки для такої розробки не потрібно багатьох «нових» знань з області мікросервісної архітектури, не потрібне налаштування багатьох середовищ, також вартість початкової розробки і життєвої підтримки є дуже невелика.

Але для великих web-додатків більш раціональним вибором буде мікросервісна архітектура. Найбільші її переваги наступні:

- всі сервіси володіють власною апаратною базою і розташовуються на окремих серварах;
- незалежна масштабованість сервісів;
- незалежна кодова база – сервіси можна розробляти незалежно один від одного, зміни одного сервісу не будуть спливати на інші сервіси;
- можливість розробки сервісів на різних мовах програмування;
- використання окремих баз даних для окремих сервісів.

Проте використання мікросервісної архітектури накладає деякі складності і обмеження на додаткок. Виникає потребність у оркестровці – обслуговуванні системи, яка керує підняттям, масштабованістю та життєвим циклом контейнерів,

у яких піднімаються всі сервіси. Серед прикладів можна виділити OpenShift, Kubernetes(K8S), docker [7].

Також цей архітектурний підхід накладає певні обмеження на спосіб спілкування між сервісами. Серед найпопулярніших сьогодні – HTTP та RPC. Більш поширеним є HTTP, так його простіше реалізовувати, у багатьох мов програмування є вже готові рішення для цього способу комунікацій і такий спосіб легше підтримувати у оркестраторах.

З часом вартість підтримки будь-якого за розміром додатку буде зростати. В разі монолітної архітектури цей рост буде набагато більш, ніж при використання мікросервісного підходу, так як розробники будуть мати справу не с одною великою кодовою базою, а с декількома маленькими.

Тому, мікросервісний підхід є більш доцільним при розробці великих web-додатків і з часом – при життевій підтримці продукту, ці перевагу будуть все більш помітні.

РОЗДІЛ 2
ОСНОВНІ ПІДХОДИ І ТЕХНОЛОГІЇ РОЗРОБКИ МІКРОСЕРВІСНИХ WEB-ДОДАТКІВ

2.1 Опис підходу

Мікросервісна архітектура надає цілий ряд технічних переваг, які сприяють швидкості розвитку та якості продукції в програмних проектах, а також сприяють загальній спритності бізнесу.

Мікросервісний підхід - це техніка розробки програмного забезпечення, яка структурує додаток як сукупність послуг, що з'єднуються. Кожна послуга є самодостатньою і повинна реалізувати можливості єдиного бізнесу. Мікросервісна архітектура призначена для подолання проблем, збоїв та збою більшої кількості додатків. Мікросервіси надають можливість додати стійкість до системи, щоб компоненти могли граціозно працювати з шипами та помилками. Завдяки цьому, кожен із зацікавлених сторін може зосередитися виключно на одному конкретному елементі загальної програми, зі своїм власним стилем програмування, не турбуючись про інші компоненти [2]. Спілкування в мікросервісах може здійснюватися без особливих зусиль, оскільки вони без стану і в чітко визначеному інтерфейсі (запит і відповідь не залежать).

Якщо програмне забезпечення розробляється за допомогою методології мікросервісів, це включає в себе постійне розгортування, що призведе до меншого часу простою на риноку. Оскільки мікросервіси є агностичними для пристроїв та платформ, це дозволяє розробляти додатки, які забезпечують покращений досвід роботи користувачів на більшості платформ, включаючи веб, мобільний, IoT,

планшети, носіння та багато іншого. На рис. 2.1 схематично зображено web-додаток, побудований на мікросервісній архітектурі.

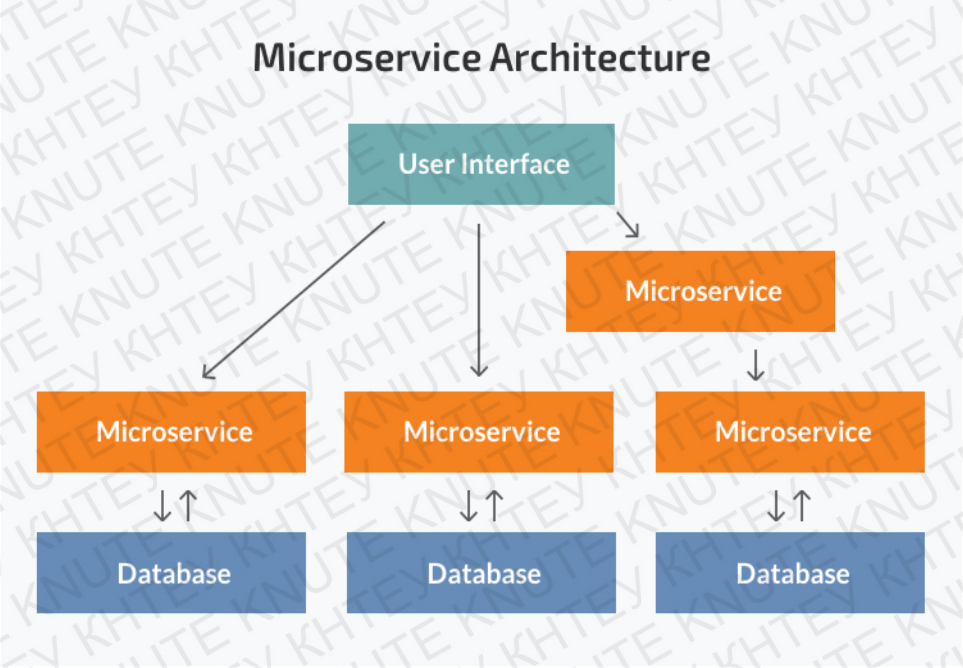

Рис. 2.1 Схематичне зображення мікросервісів

Наприклад: Walmart Canada використовував монолітну архітектуру до 2012 року! Компанія зіткнулася з помилками в обробці 6 мільйонів переглядів сторінок/хвилину, що забирало більше часу і призводило до менших продажів. Через такі проблеми вони відновили свою архітектуру програмного забезпечення на мікросервіси і виявили миттєвий результат і високий коефіцієнт конверсії протягом ночі. Час простою було скорочено, і компанія змогла використовувати більш дешеві сервери x86, а не дорогий товарний товар, що призвело до економії витрат від 20% до 50%.

Філософія мікросервісної архітектури схожа на філософію Unix, тобто "Зробіть одне і зробіть це добре". Характеристики:

- компонентизація для виконання єдиного функціоналу;
- організовано відповідно до можливостей бізнесу;
- зосереджується на продуктах, які не обробляються;
- децентралізоване управління та управління даними;

- обслуговування еластичне, пружне, компоноване, мінімальне та повноцінне.

2.2 Переваги підходу

Чому компанії з розробки програмного забезпечення повинні інвестувати в архітектуру мікросервісів?

- покращує ізоляцію несправностей: у мікросервісній архітектурі розробники знають, де саме потрібно шукати проблеми, які потрібно вирішити. Якщо зачіпається один модуль, його можна легко знести або вирішити без пошкодження інших частин програм; збільшення доступності програми Це абсолютно суперечить монолітному застосуванню; вихід з ладу одного компонента може знищити всю програму. Наприклад, мобільні ігрові програми (побудовані на монолітній архітектурі), які містять різні компоненти, такі як оплата, логін, плеєр, історія та інші. Якщо певний компонент почне витрачати більше місця в пам'яті, це вплине на всю програму, що призведе до поганого досвіду користувача;
- легко змінювати стек технологій: за допомогою мікросервісів компанія з розробки програмного забезпечення може спробувати новий стек або новітні технології на конкретних компонентах, щоб підвищити зручність використання та скористатися більшими перевагами на рівні програми. Оскільки проблем із залежністю немає, розробники програмного забезпечення можуть уникати використання певних технологічних стеків, якщо вони не забезпечують постійного користувальницького досвіду. При такій постійній модернізації ваша система не скоро стане застарілою. На Рис. 2.2 зображено приклад використання багатьох мов;

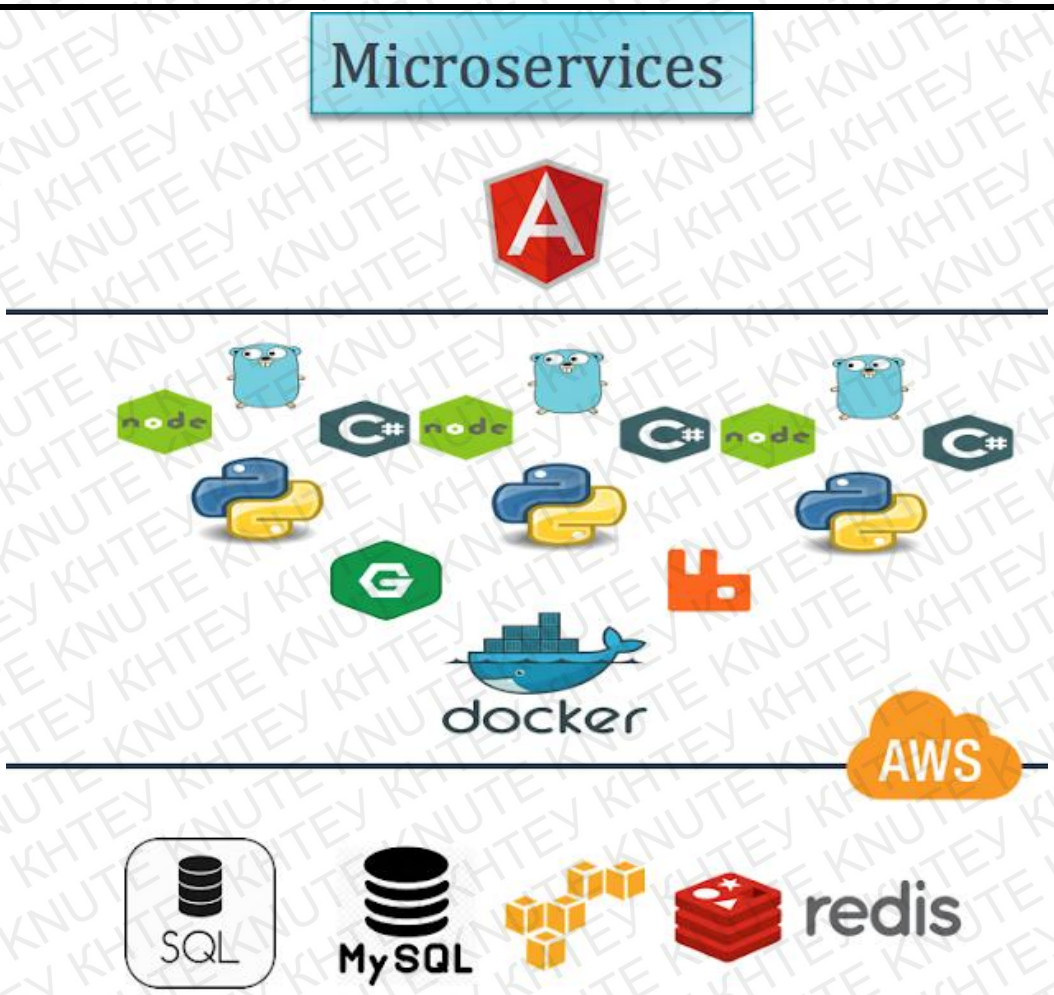

Рис. 2.2 Приклад використання багатьох мов для розробки мікросервісів в одному додатку

- забезпечує масштабованість: мікросервіси масштабують деталі, які мають проблеми з продуктивністю, та використовують апаратне забезпечення, що найкраще відповідає вимогам обслуговування. Оскільки кожна служба є окремим компонентом, масштабування може бути розгорнуто за допомогою більшої кількості контейнерів, що дозволяє більш ефективно планувати потужність, зменшувати вартість ліцензування та відповідне обладнання. Компоненти критичної служби можуть бути розміщені на декількох серверах для збільшення доступності та продуктивності, не впливаючи на продуктивність іншої служби. Ця масштабованість призводить до кращого досвіду клієнтів та збільшує економію витрат [8];

- вирівняні з організацією: якщо організація використовує мікросервіси, розмір команди може бути визначений відповідно до необхідних завдань. Крім того, команди можуть бути розбиті на менші групи та можуть зосередитись на окремих компонентах програми. Оскільки кінцевою метою є задоволення клієнтів та великий досвід роботи, команда не поділяється на команду інтерфейсу, команду бази даних тощо. Наприклад, якщо команда, що працює в ОАЕ, обробляє три послуги, тоді як команда, що працює в Каліфорнії, обробляє п'ять служб, кожна команда, що працює в Каліфорнії та ОАЕ, може випустити та розгорнути різні функції незалежно. Ці міжфункціональні команди працюють над виконанням однієї єдиної функціональності, розбиваючи сили між командою, сприяючи кращій співпраці;

- підвищення продуктивності та швидкості: за допомогою мікросервісів легко вирішити проблеми продуктивності та швидкості. Різна команда працює над різними компонентами одночасно, не чекаючи, коли одна команда виконає завдання. Це призводить до прискорення забезпечення якості, оскільки кожну мікропослугу можна перевірити індивідуально. Інші зацікавлені сторони можуть працювати над покращенням уже розроблених компонентів, тоді як інші програмісти працюють над іншими. Це збільшує швидкість і призводить до більш швидкого випуску продукту.

2.3 Складності у розробці

Просто тому, що все виглядає вишукано, це не означає, що воно ідеально підходить для програмної галузі; у нього є потенційні болі, які також потрібно вирішити:

- оскільки мікросервіс фокусується на розподілених системах та незалежному обслуговуванні, кожен запит повинен ретельно оброблятися між модулями. Може статися, що одна з служб не відповідає, що змушує розробників написати додатковий код, щоб уникнути збою.

- тестування програми на базі мікросервісу може бути болючим завданням, оскільки кожну залежну службу потрібно підтвердити перед початком тестування. Зі збільшенням кількості послуг складності не залишаються за лаштунками! Ведення вкладок на всіх сервісах стає недоцільним, оскільки можуть виникати помилки в базі даних, затримка мережі, проблеми кешування тощо. Отже, необхідне тестування на витривалість та усунення несправностей стає обов'язковим.

- кожна служба залежить від власного API та платформи, а відстеження всього може бути важкою роботою. Лідеру потрібно стежити за кількома об'єктами та керувати всією інфраструктурою, оскільки, якщо будь-яка служба не працює, слідкувати за проблемами стає нудно. Таким чином, необхідний надійний моніторинг.

- при дебагу, за допомогою монолітних додатків ви можете додати гачки налагодження в код і логічно переходити через кожен шар виконання, щоб виявити проблемні області. Коли ви маєте справу з декількома а то й сотнею окремих мікросервісів, що розмовляють один з одним із слабо визначеними API, ви не можете так просто зрозуміти у якому місці програма дає збій.

- розробка розподілених систем може бути складною. Оскільки зараз все є незалежним сервісом, ви повинні ретельно обробляти запити, що подорожують між вашими модулями. В одному з таких сценаріїв розробники можуть бути змушені писати додатковий код, щоб уникнути збою. З часом ускладнення виникнуть, коли віддалені дзвінки відчувають затримку.

- кілька баз даних та управління транзакціями можуть бути болючими.

- розгортання мікросервісів може бути складним. Можливо, їм потрібна координація між декількома службами, яка може бути не такою простою, як розгортання war-архіву в контейнері.

2.4 Середовище розробки

П'ятнадцять років тому єдиним варіантом було б встановити та запустити мікросервіси на фізичному сервері під управлінням операційної системи. Але цей підхід був би сьогодні надзвичайно марнотратним, враховуючи величезну кількість потужних серверів, що працюють зараз. Щоб обійти це, ви можете розглянути можливість запуску декількох служб на одному екземплярі операційної системи, але це ризикує виникнути конфліктними версіями бібліотеки та компонентами додатків, не маючи на увазі той факт, що одна несправність мікросервісу може вплинути на доступність інших. Запуск мікросервісу на голому металі не є привабливим варіантом.

Наступним очевидним вибором є поділ фізичного сервера на багато віртуальних серверів (віртуальних машин або VM), що дозволяють декільком середовищам проживання на одному сервері. Віртуалізація - це зріла, налагоджена технологія, більшість підприємств вже інвестували у віртуальну інфраструктуру, і більшість постачальників хмарних технологій використовують VM як основу для своїх пропозицій щодо інфраструктури як послуги (IaaS). Але це має серйозні обмеження, коли справа стосується запуску мікросервісів.

Найкращий вибір для роботи архітектури додатків для мікросервісів - контейнери додатків. Контейнери містять легке середовище виконання програми, представляючи послідовне програмне середовище, яке може слідкувати за програмою від робочого столу розробника до тестування до остаточного розгортання, а також можна запускати контейнери на фізичних або віртуальних машинах.

Отже, оскільки контейнери дають змогу існувати в декількох середовищах виконання на одному екземплярі операційної системи, кілька компонентів додатків можуть співіснувати в одному середовищі VM. Крім того, за допомогою Linux ви можете використовувати контрольні групи (групи) для ізоляції повного

середовища виконання для певного набору коду додатків, гарантуючи, що кожен має приватне середовище і не може впливати на роботу інших додатків [3].

Ця здатність ізолювати розробників від необхідності відокремлення коду програми на окремі VM, дозволяє використати більше час на написання/покращення коду програми, аніж на налаштування додатку для запуску на певній ОС.

Результат: ви отримуєте більшу величину обробки з існуючого обладнання. Наслідки цього можуть бути різними, оскільки характеристики застосування різні. Наприклад, деяким потрібно велика потужність обробки, а інші генерують багато мережевого трафіку. При розумно розміщеному робочому навантаженні, користувачі контейнерів можуть максимально використовувати рівні використання всіх ресурсів сервера, а не просто завантажувати їх декількома програмами, що вивішують процесор, які залишають невикористану деяку мережу.

За допомогою цього типу ізоляції тепер можна розмістити декілька мікросервісів на одному сервері. Функціональність cgroup гарантує, що жодна служба не може перешкоджати іншій, тоді як ефективність контейнера дозволяє підвищити швидкість використання сервера.

Однак є одне застереження: слід запустити мікросервіси у надмірній конфігурації, щоб підвищити стійкість, і переконайтеся, що вони не опиняються в бічних контейнерах на тому ж фізичному сервері, оскільки це перешкоджає меті надмірності. Хоча можна керувати розміщенням контейнерів вручну, щоб запобігти їх розміщенню, набагато краще використовувати систему управління контейнерами, наприклад Kubernetes або OpenShift, який дозволяє використовувати політику для диктування розміщення контейнерів.

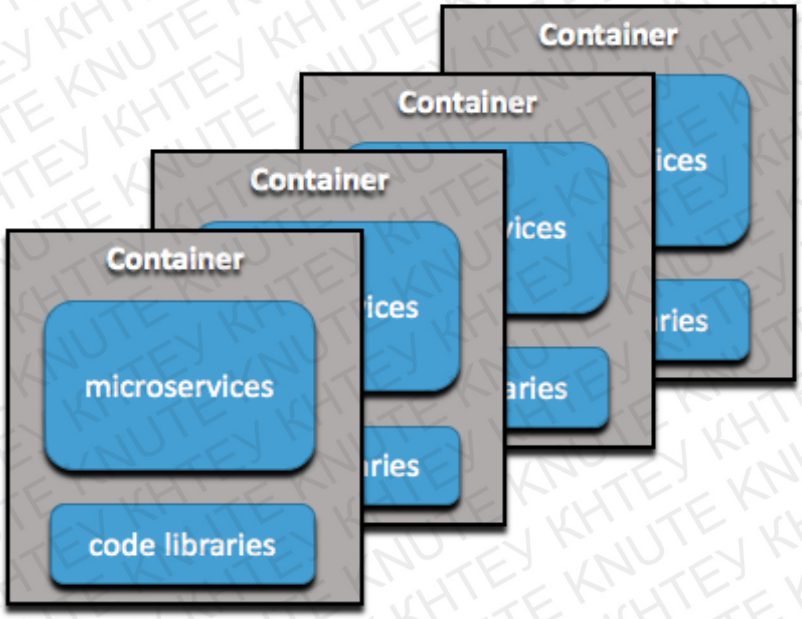

Рис. 2.3 Деплой мікросервісів у контейнерах.

На рис. 2.3 зображено приклад існування мікросервісів у контейнерах. Ми бачимо, що всі контейнери є незалежними один від одного та мають свій набір мікросервісів та бібліотек для успішного функціонування.

2.5 Тестування

Мікросервіси (порівняно з монолітом) потребують додаткових кроків, так як керують кількома сховищами, кожний з сервісів зі своєю схемою бази даних. Але виклики можуть бути глибшими за це.

Ось кілька основних проблем, пов'язаних із тестуванням мікросервісів:

- доступність: Оскільки різні команди можуть керувати власними мікросервісами, забезпечити наявність мікросервісу (або, що ще гірше, намагатися знайти час, коли всі мікросервіси будуть доступні одразу), важко;
- фрагментарне та цілісне тестування: Мікросервіси створені для роботи поодинці та разом з іншими сервісами з нещільним зв’язком. Це означає, що розробникам потрібно протестувати кожен компонент ізольовано, а також протестувати все разом;

- прогалини в знаннях: Зокрема, з інтеграційним тестуванням, кожен, хто проводить тести, повинен мати чітке розуміння кожної служби, щоб ефективно писати тестові справи;

- Різні версії API потребують нового тестування і нових інтеграційних тестів;

Загальні підходи до тестування мікросервісів:

- unit-тести: мікросервіс може бути меншим за визначенням, але за допомогою такого тестування ви можете перейти ще більш детального тестування. Тест блоку фокусується на найменшій частині тестуваного програмного забезпечення, щоб з'ясувати, чи працює цей компонент як слід.

- інтеграційне тестування: за допомогою тестів на інтеграцію ви робите те, що вам здається: тестуйте взаємодію між компонентами для виявлення проблем, здійснюєте перевірку шляхів зв'язку через підсистему, щоб перевірити на предмет неправильних припущень, які має кожен сервіс щодо взаємодії з іншими сервісами.

- End-to-end тестування: І останнє, але не менш важливе - це тестування в кінці, яке, як було сказано раніше, може бути складним завданням. Це тому, що він включає тестування кожної рухомої частини мікросервісу, гарантуючи, що вона може досягти цілей, для яких ви побудували.Такі тести за визначенням більші, займають більше часу. Щоб мати низькі витрати і уникати затримки часу, потрібно дотримуватися тестування поступово, коли всі інші засоби тестування вичерпані - як остаточний штамп забезпечення якості.

2.6 Моніторинг системи

Моніторинг залишається важливою частиною управління будь-якою IT-системою, тоді як проблеми, пов'язані з моніторингом мікросервісів, особливо унікальні. Прикладом є те, як традиційні монолітні системи, розгорнуті як єдиний виконуваний файл або бібліотека, мають різні точки відмови та залежності, ніж ті, що розгорнуті з мікросервісною архітектурою.

Важливо зрозуміти моніторинг в цілому і чим він відрізняється для мікросервісів. Програми на основі мікросервісів мають різні та більш інтенсивні

вимоги до моніторингу. Моніторинг є найважливішим елементом систем управління мікросервісами, оскільки чим складнішим стає ваше програмне забезпечення, тим складніше зрозуміти його ефективність та усунути неполадки. Однак, враховуючи кардинальні зміни в постачанні програмного забезпечення, моніторинг потребує капітального ремонту для успішної роботи в мікросервісному середовищі. Далі представлено чотири принципи моніторингу мікросервісів:

- відстежуйте контейнери, а не те, що працює всередині них. Контейнери набули популярності як будівельні блоки мікросервісів. Швидкість, портативність та ізоляція контейнерів полегшили розробникам модель мікросервісу. Контейнери - це чорні скриньки для більшості систем, які живуть навколо них. Але коли справа стосується роботи, моніторингу та усунення несправностей із службою, чорні скриньки ускладнюють загальні дії, що змушує задуматися: що працює в контейнері? Як працює програма / код? Виплітає важливі спеціальні показники? З точки зору DevOps, вам потрібна глибока видимість всередині контейнерів, а не просто знати, що деякі контейнери існують;
- використовуйте системи оркестрування для оповіщення про продуктивність контейнерів. Ознайомлення з операційними даними в контейнерному середовищі є новим завданням. Метрики одного контейнера мають набагато нижче граничне значення, ніж сукупна інформація з усіх контейнерів, що складають функцію чи послугу. Особливо це стосується інформації на рівні додатків, наприклад, які запити найповільніші або які URL-адреси відповідають з найбільшою кількістю помилок, але також застосовується до моніторингу на рівні інфраструктури, наприклад, які контейнери служб використовують найбільше ресурсів за винятком розподілених частот процесора;
- будьте готові до послуг, які є еластичними та багатоповерховими. Еластичні послуги, безумовно, не нова концепція, але швидкість змін набагато швидша в середовищі, де є контейнери, ніж віртуалізовані середовища. Швидко мінливі середовища можуть спричинити хаос у крихких системах моніторингу. Але вони мають адаптуватися до змін без втручання людини;

- моніторингу API. У мікросервісних середовищах API - є єдиними елементами служби, які піддаються впливу інших команд, тому їх моніторинг є важливим. Моніторинг API може приймати різні форми, наприклад, корисно розуміти найбільш часто використовувані кінцеві точки як функцію часу. Це дозволяє командам бачити, чи змінилося щось помітне у використанні сервісів, чи це пов'язано зі зміною дизайну чи зміною користувача.

2.7 Висновки до розділу 2

Тож мікросервісна архітектура – прекрасний вибір для розробки великих web-додатків через те, що велика кількість людей може одночасно працювати у різних командах над різними сервісами ніяк не заважаючи один одному. Для розробки можуть використовуватися різні мови та технології. Мікросервіси дуже легко масштабувати, так як вони ізольовані від всього та підстроювати під структуру своєї організації.

Легка масштабованість забезпечена ізоляцією мікросервісів, так як зазвичай вони розгортуються у контейнерах – маленький віртуальних серверах, які можеть бути розгорнуті на 1му фізичному сервері або віртуальній машині. Контейнери налаштовуються окремо під сервіси, а можуть бути однакові – це залежить від необхідності серверів у різному забезпеченні. Для полегшення контролю контейнерів використовуються оркестратори - OpenShift, Kubernets.

Важливим моментов роботи з мікросервісами є тестування. Воно стає важчим, через існування багатьох кодових баз і багатьох інтеграцій с базами даних, кешами та іншими продуктами.

Ніколи не можно забувати про моніторинг мікросервісного додатку. Це є нелегким лише при першому використанні і установці, але воно того варте. Завдяки своєчасним сповіщенням команда зможе вчасно зреагувати на якісь події і зробити щось для того, щоб система працювала справно як і раніше.

В результаті проведеного аналізу існуючих концепцій і підходів щодо створення корпоративних web-додатків, розроблено технічне завдання на web-

додаток з мікросервісною архітектурою з використанням технологій середовища Docker та мови програмування Java (додаток Б).

РОЗДІЛ 3
РОЗРОБКА WEB-ДОДАТКУ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1 Мова програмування та фреймворк

Серед багатьох мов програмування, які підходять для створення web-корпоративних додатків Java є одним з найкращих варіантів, оскільки сьогодні ця мова є однією з найпопулярніших, через те, що вона мультиплатформенна, об'єктно-орієнтована, має безліч фреймворків, для Java реалізовано дуже багато різних паттернів не лише програмування, а й мікросервісних паттернів [4]. Насправді, багато з найбільших імен у галузі інтернету, що займаються неймовірним трафіком, використовують Java, такі як Netflix, Amazon, Google, eBay, Twitter та LinkedIn.

Серед відомих фреймворків - Spring Boot, Jersey, Dropwizard, Play Framework та Restlet. Spring Boot - це найкращий та найбільш часто використовуваний фреймворк для побудови мікросервісних web-додатків [11]. Перевага Spring Boot полягає в тому, що він має велику інфраструктуру та багато попутніх проектів для полегшення розробки мікросрвісних додатків. Серед попутніх проектів:

- Spring Cloud Eureka;
- Spring Cloud Config;
- Spring Cloud Sleuth;
- Spring Cloud Hystrix;

Будь-який сервіс, написаний на Java, буде скомпільований та зібраний у jar архіви, потім на основі centos образу будуть створені dockerfile, на основі яких

будуть створені та запущені контейнери у докері. База даних MySql також буде запущена в окремому контейнері.

На рис 3.1 зображено загальну схему функціонування мікросервісів с використанням докеру. Контейнери спілкуються між собой та з базою даних, насправді, всередині контейнерів розгорнуті мікросервіси, які спілкують з іншим мікросервісами або ж базою даних.

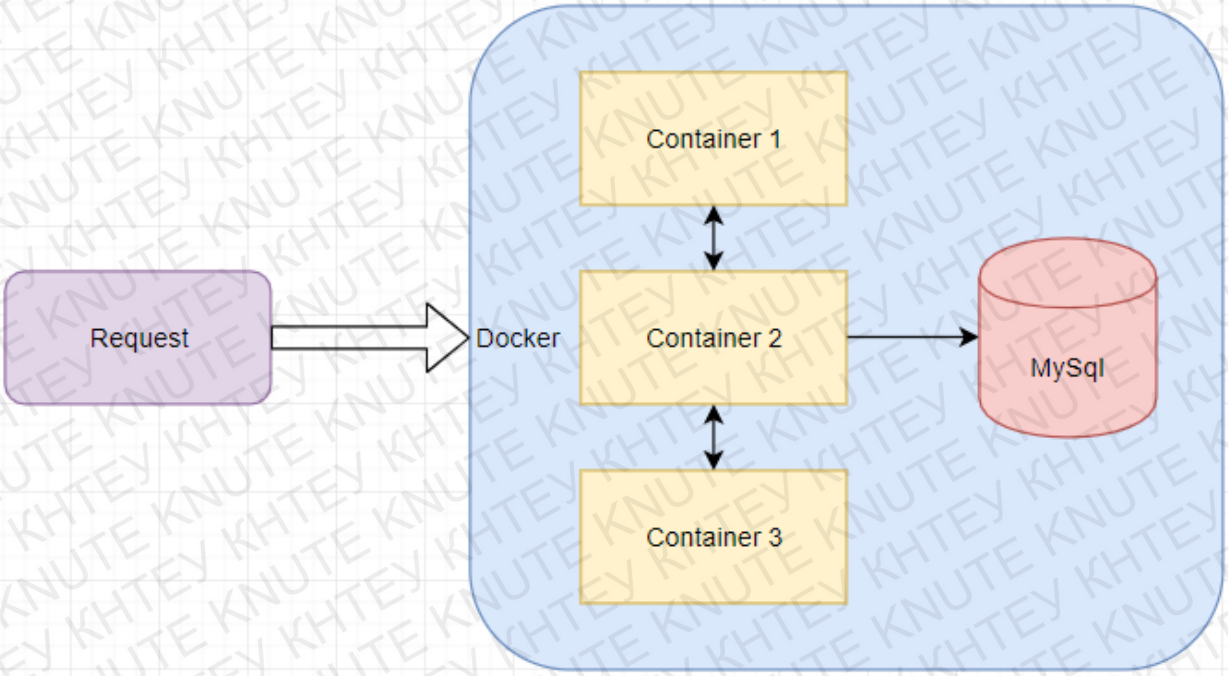

Рис. 3.1 Схема функціонування мікросервісного додатку в середовищі докер.

Більшість додатків з використанням Spring Boot побудовані однаково. Всі додатки будуються з використанням засобу автоматизації роботи с програмними проетами – Maven [10]. Завдяки йому не потрібно буде вручну закачувати бібліотеки, які потрібні проекту, а лише вказати їх як залежності у файлі «pom.xml». Приклад файлу наведено у додатку А.

3.2 Проектування архітектури додатку

Web-додаток буде складатися з 2х сервісів: Web-service та user-service. Web-service буде отримувати запити від клієнта, створювати запити до user-service, отримувати відповідь, обробляти її і якимось чином відповідати клієнту. До user-service буде підключена база даних MySql для зберігання даних про користувачів. Схема обробки запитів сображена на рисунку 3.2.

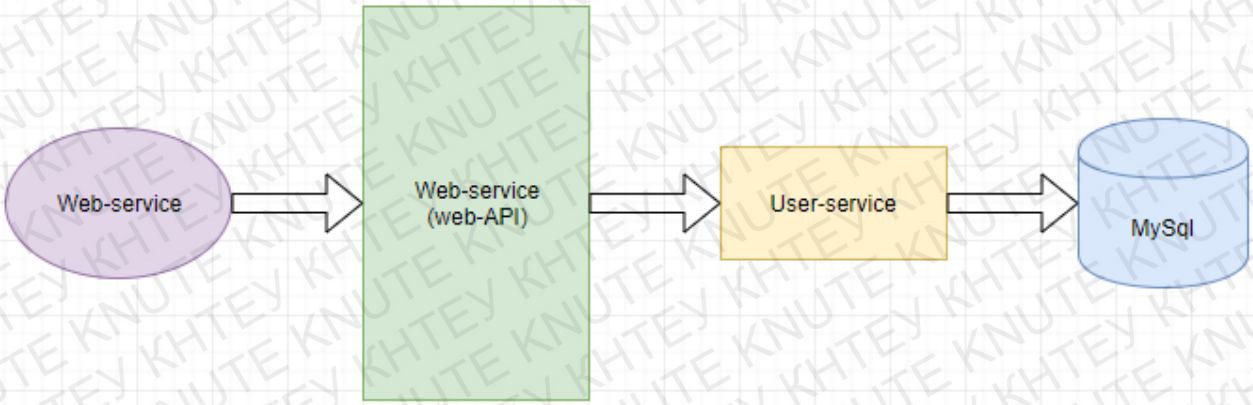

Рис. 3.2 Схема обробки запитів додатку.

Як ми бачимо, запит буде приходити на Web-service, який являє собой сайт з декількома сторінками. Одна додає користувачів, інша видаляє користувачів.

Web-service будуе мати ієрархію зображену на рис 3.3, а User-service ієрархію зображену на рис. 3.4.

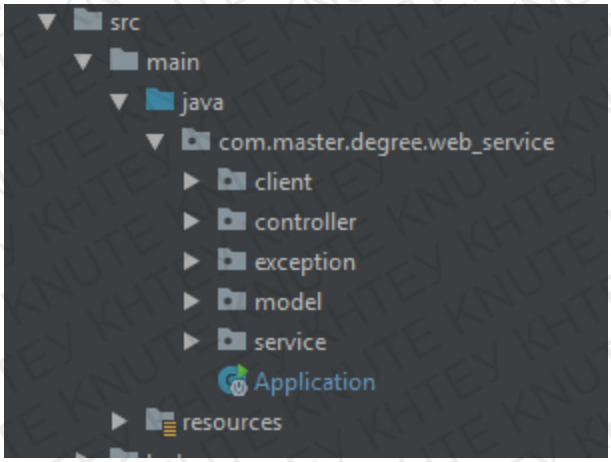

Рис. 3.3 Ієрархія пакетів Web-service.

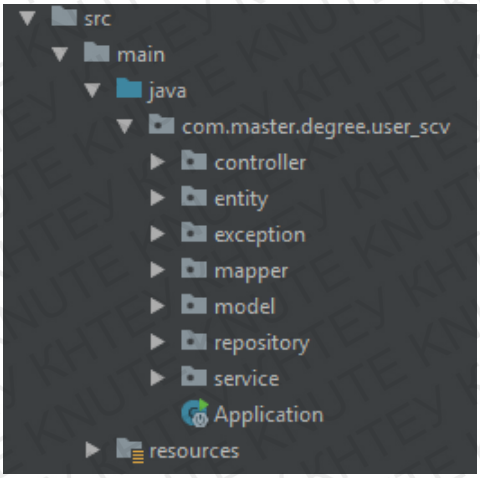

Рис. 3.4 Ієрархія пакетів User-service.

Як видно, вони дуже схожі між собой, але відрізняться пакетом, у якому містять всі інші пакеті «web_service» у Web-service та «user_svc» у User-service.

Web-service має пакет client, який має класи для комунікацій с іншими сервісами. А пакети «entity» та «repository» в User-service мають класи для роботи з базою даних.

Обидва сервіси мають пакет «controller», який містить класи-контроллери, які декларують Http Rest Api. Приклад коду контроллеру з user-service наведено у додатку В.

У User-service доступні наступні ендпоінти:

- «/users» Get метод – отримати всіх користувачів;
- «/users/{email}» Get метод – отримати користувачи по емейлу;
- «/users» Post метод – зберігти користувача;
- «/user/{email}» Delete метод –видалити користувача по емейлу;

Web-service інстанціює настпуні:

- «/web/users» Get метод – запит на отримання сторінки с відображенням користувачів;
- «/web/signup» Get метод – запит на отримання сторінки с відображенням форми додання користувачів;
- «/users» Get метод – отримати всіх користувачів;
- «/users» Post метод – створити користувача та повернути сторінку с вже створеними користувачами;
- «/users/delete/{email}» Delete метод – видалення користувача по емейлу;

Після компіляції сервісів, створюються jar архіви, з яких будуються docker-образи командою «docker build –t user-svc .». Команду потрібно виконувати у корневій директорії, але також потрібно мати Dockerfile – файл з інструкціями по створенню docker-image. Він виглядає наступним чином:

```
FROM centos:7
RUN yum install -y \
    java-1.8.0-openjdk \
    java-1.8.0-openjdk-devel
ENV JAVA_HOME /etc/alternatives/jre
COPY target/user-service-1.0-SNAPSHOT.jar /app/user-svc.jar
CMD ["/usr/bin/java", "-jar", "-Dspring.profiles.active=default", "/app/user-svc.jar"]
```

Тут сказано, що образ буде побудовано на основі centos образу Linux з установкою openjdk в систему, копіюванням jar архіву в образ(в centos) і стартову команду при запуску контейнера.

3.3 Розгортування додатку в середовищі docker

Після того, як обидва сервіси були скомпільовані і зібрані в jar архіви командою «mvn clean package», створюються docker образи, з яких створюються контейнери. Контейнери з сервісами створюються командами:

« docker run --name web-svc -p8080:8080 --net mynet web-svc» та

« docker run --name user-svc -p8081:8081 --net mynet user-svc ».

Команда «run» створює новий контейнер, параметр «--name web-svc» вказує ім’я контейнера, параметр «-p 8080:8080» буде перенаправляти трафік, який прийде на порт 8080 машини, на якій запущений докер, і направить його до порту 8080 контейнера, параметр «--net mynet» вказує в якій мережі треба запускати контейнер, якщо не вказати цей параметр, то контейнери не зможуть бачити один одного. Ім’я контейнера – унікальний ідентифікатор контейнера, по цьому імені можна робити HTTP запити до мікросервіса всередині контейнера. Тобто user-service можна викликати з web-service по «http://user-svc».

База даних розгортується в окремому контейнері командою

«docker run --name=sql –p 3306:3306 -d mysql/mysql-server». Після старту контейнерів, виконання команди «docker ps» окажи запущені контейнери з інформацією про них. На рис. 3.5 зображено вивод команди «docker ps».

```
C:\Users\Andrushka>docker ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS
      PORTS                                NAMES
fd75452802cd        user-svc             "/usr/bin/java -jar …"   34 hours ago        Up 2 minutes
      0.0.0.0:8081->8081/tcp               user-svc
b0f4acabefd7        login-svc            "/usr/bin/java -jar …"   34 hours ago        Up 2 minutes
      0.0.0.0:8080->8080/tcp               web-svc
605a1e8bc788        mysql/mysql-server   "/entrypoint.sh mysq…"   4 days ago          Up 3 minutes (healt
hy)   0.0.0.0:3306->3306/tcp, 33060/tcp   sql
```

Рис. 3.5 Вивід команди «docker ps».

3.4 Демонстрація мікросервісного підходу роботи додатку

Першою сторінкою є сторінка з відображенням вже існуючих користувачів.

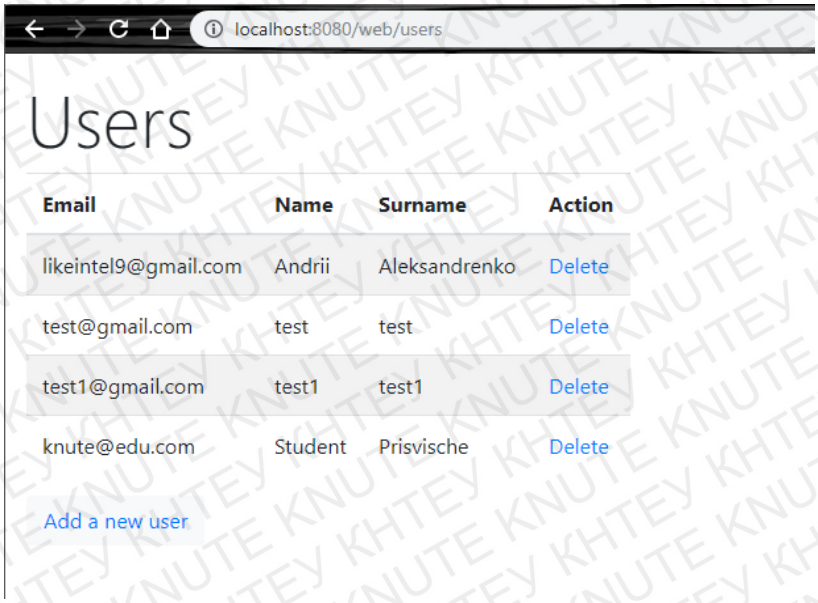

Рис. 3.6 Сторінка виводу користувачів

Вона показує існуючих користувачів в базі даних. Виводиться 3 поля: «Email», «Name» , «Surname» і можливість видалити користувача.

На рис. 3.7 зображена сиквенс-діаграма запитів, вона показує який сервіс робитиь який запит відносно інших запитів в межах одного бізнес процесу, наразі це вивід існуючих користувачів в браузері.

Клієнт іцініює запит до web-service, який в свою чергу робить запит до user-service, який дістає інформацію з бази даних та повертає її. На схемі видно, як запит проходить через кожний сервіс.

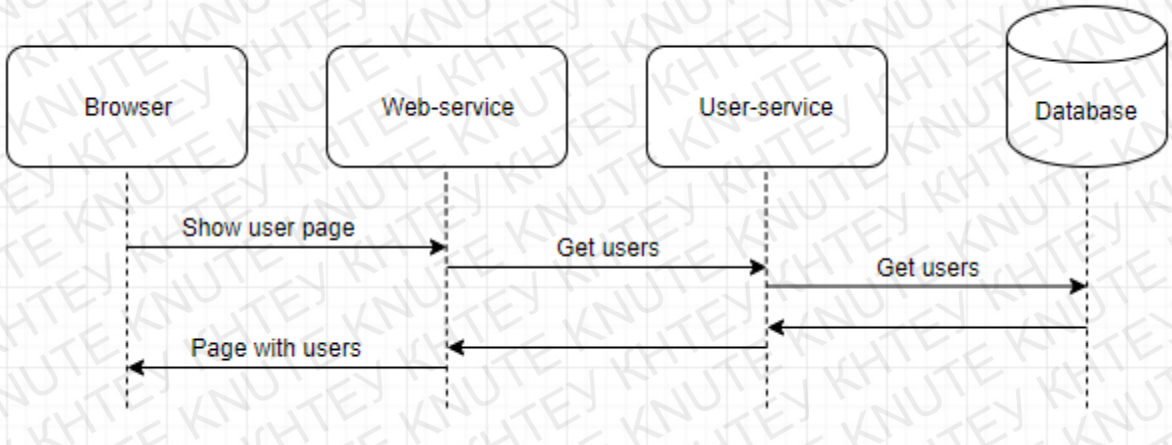

Рис. 3.7. Діаграма запитів процесу «Список користувачів».

Наступний бізнес процес – «Створення користувача». Сторінказображена на рис. 3.8. Результатом процесу буде додання нового запису до бази даних. При успішному збереженні запису, web-service поверне сторінку с існуючими користувачами. Але якщо буде спроба створити нового користувача з емейлом, який вже є в базі даних – буде повернута помилка в браузер.

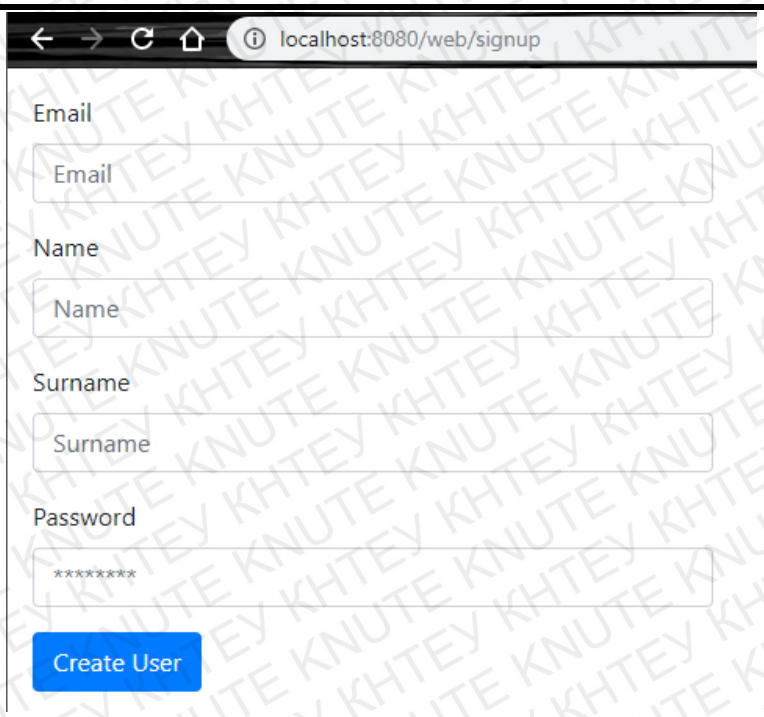

Рис. 3.8 Сторінка додання користувача

На рис 3.9 зображено помилку, яка вказує, що такий емейл вже існує в базі даних.

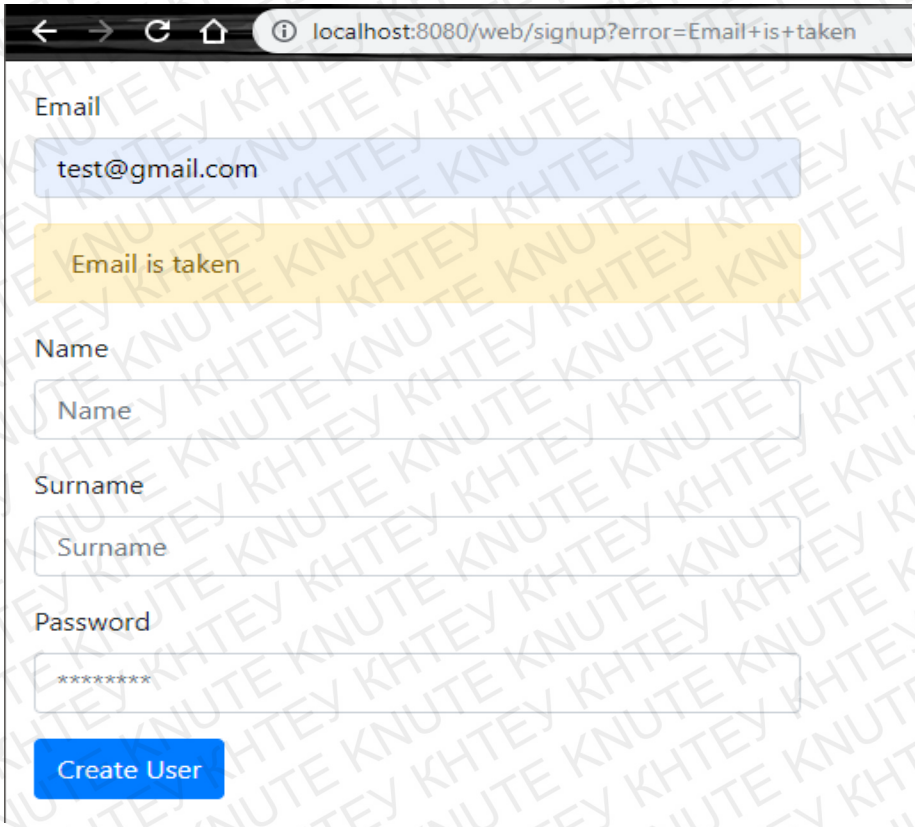

Рис. 3.9. Помилка при доданні користувача з існуючим емейлом.

На рис. 3.10 відображено діаграму запитів бізнес процесу «Створення користувача». 2 варінти сображені червоним та синім блоками, перший поверне помилку, інший поверне сторінку с користувачами.

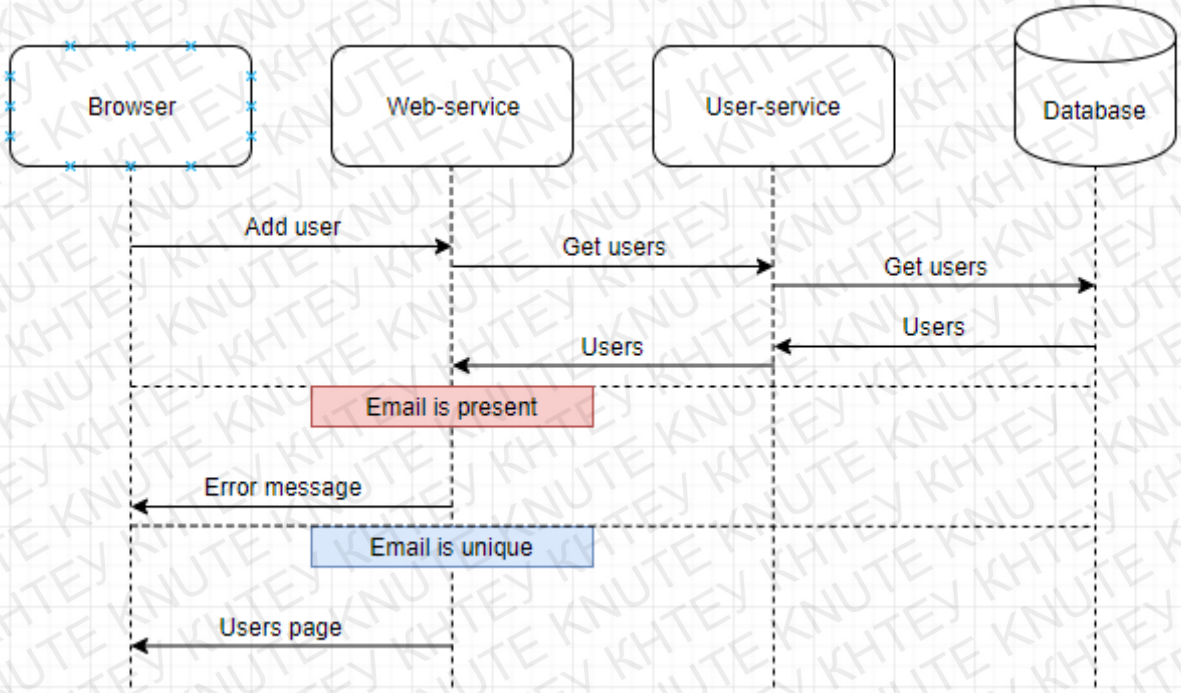

Рис. 3.10. Діаграма запитів процесу «Створення користувача».

Останнім є процес «Видалення користувача». Діаграма зображена на рис. 3.11 показує послідовність запитів. Процес видаляє запис с бази даних та повертає сторінку с існуючими користувачами.

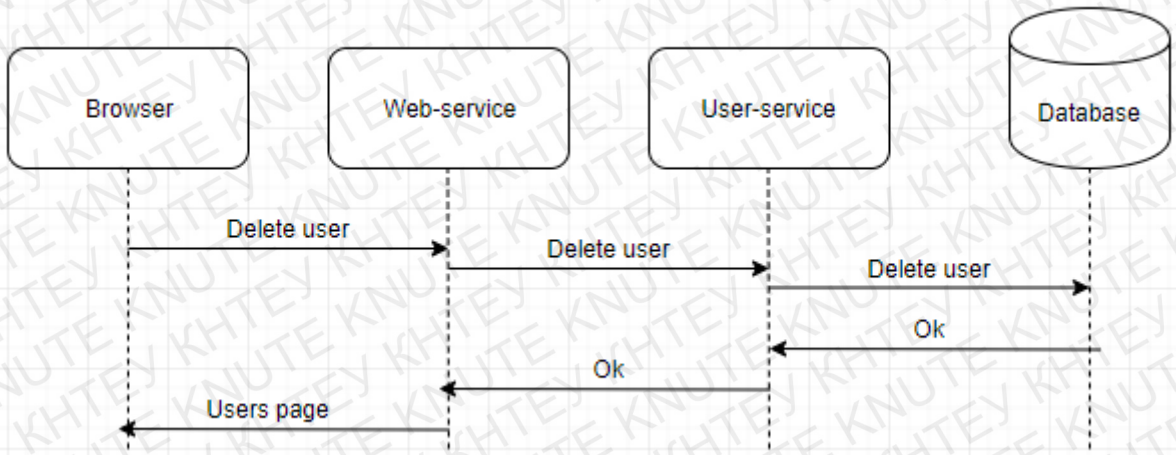

Рис. 3.11 Діаграма запитів процесу

3.5 Аналіз розробленого продукту

Даний додаток був розроблений с використанням мікросервісного підходу, що дуже добре видно по діаграмам бізнем процесів. Бізнес процес складається з декількох запитів від сервіса до сервісу, щоб виконати 1 певну дію.

В цьому додатку існує лише 2 сервіси, проте в багатьох великих web-додатках кількість мікросервісів може перевищути 100 одиниць. Це дозволяє розпределити навантаження за рахуванок великої кількості мікросервісів, що дуже позитивно впливає на максимально можливу кількість потенційно оброблених запитів. При необхідності, кількість будь-якого з сервісів можна збільшити, щоб додаток був в змозі обробити підвищену кількість запитів. Часто, це робиться автоматично, базуючись на показаннях використання пам’яті та навантаження на процесор. Використання докеру легко дозволяє змінитив версію розгорнутого додатку, просто створивши новий докер образ та запустивши контейнер з використанням щойно створеного образу.

3.6 Висновки до розділу 3

У розділі було розроблено web-додаток с використанням мікросервісної архітектури дотримуючись поставлених цілей та методологій, згідно технічного

завдання, проведено аналіз архітектури. Було використано мову програмування Java та фреймоворк Spring Boot для зменшення кількості написаного коду. Було застосоване середовище docker для розгортування сервісів.

Було розібрано та виділено переваги використання мікросервісного підходу на прикладі створеного додатку. Після виконання аналізу додатку, можна зробити висновок, що вибір мікросервісної архітектури є зваженим рішенням дял розробки великих за розміром додатків, або ж додатків з великим навантаженням.

ВИСНОВКИ ТА ПРОПОЗИЦІЇ

Результати роботи пройшли апробацію та опубліковані в збірнику наукових статей студентів [5].

В успіху будь-якого web-додатку велику роль грає вибрана архітектура, оскільки вона впливає на такі фактори як швидкість роботи додатку, максимально можливу кількість одночасних запитів, легкість розробки, вартість розробки та підтримки.

В даній роботі після виконання аналізу було досліджено, що для великих за розміром та сильно навантажених додатків мікросервісна архітектура підходить краще інших, оскільки додаток, розроблений на основі цієї архітектури здатний забезпечити велику продуктивність та працювати під великими навантаженнями довгий час. Код мікросервісів набагато легше підтримувати та доповнювати новою функціональністю, порівняно с монолітним додатком, оскільки код всього додатку розбитий на логічні мікросервіси з своєю кодовою базою, тому одночасно декілька команд розробників можуть працювати над певним мікросервісом без необхідності знати, як працюють інші мікросервіси.

Було досліджено, що кожний мікросервіс має власну апаратну базу і розгортується в окремій операційній системі незалежно один від одного. Це дає змогу використовувати будь-яку мову програмування та будь-який фреймворк в межах одного сервісу.

В результаті роботи було досліджено базові функції автоматизатору розгортання контейнерів – docker. В ньому було створено 3 контейнери с базою даних MySql та двума мікросервісами.

					КНТЕУ 121– 05-01.МР		
					Проектування мікросервісної архітектура побудови корпоративних web-додатків	*Стадія*	*Аркуш*	*Аркушів*
Зм.	*Аркуш*	*№ докум.*	*Підпис*	*Дата*		ВП	*45*	51
Зав. каф.		Криворучко О.В.						
Керівник		Рассамакін В.Я.						
Гарант		Криворучко О.В.			*Висновки*	Факультет обліку, аудиту та інформаційних систем, 2м курс, 5 групи		
Розробив		Александренко А.А.						

Розглянуто основні засоби розробки, доведено і обгрунтовано правильність вибору цих засобів для розробки додатку. Встановлено, що обгрунтованим

вибором буде мова програмування Java і фреймворк Spring з використанням менеджеру автоматизації програмних продуктів Maven. Фреймворк надзвичайно полегшує швидкість створення додатку з нуля, а Maven дозволяє компілювати весь проект та закачувати всі необхідні бібліотеки відразу, не витричаючи час на цю роботу.

Результатом даної роботи є створений корпоративний web-додаток у середовищі докер з використання мови програмування Java. Результатом його апробації є підтвержений висновок , що мікросервісна архітектура є раціональним вибором при розробці навантажених web-додатків.

В цілому розробка web-додатків є перспективним направлінням, оскільки вплив інтернет технологій на наше життя збільшується з кожним роком, рівень попиту на нові інформаційні системи, медіа-системи та їх покращення постійно зростає і лише ті портали, які можуть надати інформацію без перешкод, переможуть у конкуренції.

Щодо подальшого вдосконалення система може бути доповнена системою моніторингу. Додання системи збирання метрик Prometheus, та систему відображення зібраних метрик Grafana.

Результати роботи пройшли апробацію на Всеукраїнській науково-практичній конференції «Безпека соціально-економічних процесів в кіберпросторі», 27 березня 2019 року місто Київ [5] та були опубліковані у журналі Software Engineering комп’юрених систем [6].

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sam Newman – Monolith to microservices - O'Reilly Media; 2019 – 44 с.
2. Sam Newman – Building microservices - O'Reilly Media; 2014 – 17 с.
3. Morgan Kaufmann - Virtual Machines: Versatile Platforms for Systems and Processes; 1 edition, 2005 – 51 с.
4. Herbert Schildt - Java: The Complete Reference, Ninth Edition. / Herbert Schildt - McGraw-Hill Education; 9 edition, 2014 – 1312 с.
5. Безпека мікросервісних web-додатків: журнал Безпека соціально-економічних процесів в кіберпросто / Александренко А.А, Рассамакін В.Я. – К.: Київ. нац. торг. н-т, 2019 – 201-202 с.
6. Software engineering комп'ютерних систем - зб. наук. ст. студ. відп. ред. – Александренко А.А. Київ. Київ. нац. торг. -екон. ун-т. 2019 6-10 с.

Інтернет ресурси

7. Docker – Why docker? [Електронний ресурс]. – Режим доступу: https://www.docker.com/why-docker .
8. Microservices benefits [Електронний ресурс]. – Режим доступу : https://skelia.com/articles/5-major-benefits-microservice-architecture/
9. Spring Boot - Microservices with Spring [Електронний ресурс]. – Режим доступу: https://spring.io/blog/2015/07/14/microservices-with-spring.
10. Maven - What is maven [Електронний ресурс]. – Режим доступу : https://maven.apache.org/what-is-maven.html.
11. Microservices – What are microservices? [Електронний ресурс]. – Режим доступу: [Електронний ресурс]. – Режим доступу: https://microservices.io/.

ДОДАТКИ

Лістинг файлу pom.xml ДОДАТОК А

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.5.RELEASE</version>
    <relativePath/>
  </parent>
  <groupId>com.master.degree</groupId>
  <artifactId>user-service</artifactId>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
```

```
    <artifactId>mysql-connector-java</artifactId>
      <version>8.0.16</version>
    </dependency>
    <dependency>
      <groupId>org.mapstruct</groupId>
      <artifactId>mapstruct</artifactId>
      <version>1.3.1.Final</version>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <version>1.18.10</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
```

```
            <annotationProcessorPaths>
              <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>1.3.1.Final</version>
              </path>
              <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.10</version>
              </path>
            </annotationProcessorPaths>
            <compilerArgs>
              <compilerArg>
                -Amapstruct.defaultComponentModel=spring
              </compilerArg>
            </compilerArgs>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

```
@AllArgsConstructor
@RestController
@RequestMapping(path = "/users")
public class UserController {
    private UserService userService;
    @GetMapping(path = "/{email}")
    public User getByEmail(@PathVariable("email") String email) {
        return userService.getByEmail(email);
    }
    @GetMapping
    public List<User> getAll() {
        return userService.getAll();
    }
    @PostMapping
    public User save(@RequestBody UserSaveDto userDto) {
        return userService.save(userDto);
    }
    @DeleteMapping("/{email}")
    public void delete(@PathVariable("email") String email) {
        userService.delete(email);
    }
}
```

Київський національний торговельно-економічний університет
Кафедра програмної інженерії та кібербезпеки

РЕФЕРАТ

ВИПУСКНИЙ КВАЛІФІКАЦІЙНИЙ ПРОЕКТ

на тему:

«Проектування мікросервісної архітектури побудови корпоративних web-додатків»

Студента 2м курсу, 5 групи,
Спеціальності 6.050103
«Програмна інженерія»
спеціалізації «Інженерія програмного забезпечення»

_____
підпис студента

Александренко
Андрій
Анатолійович

Науковий керівник
кандидат технічних наук,
доцент

_____
підпис керівника

Рассамакін
Володимир Якович

КИЇВ – 2019

Актуальність теми. Метою власника продукту є отримання більшого прибутку за менших витрат - це можливо лише за правильно спроектованої архітектури додатку, так як легкість внесення змін до існуючих бізнес-функцій та простота додавання нових бізнес-функцій до існуючої системи є основними витратами при пітримці додатку після його релізу. Тому постає проблема вибору найбільш досконалої та кращої архітектури корпоративних web-додатків.

Мета: створення web-додатку з мікросервісною архітектурою використанням технологій середовища Docker.

Об'єкт дослідження: проектування корпоративних web–додатків.

Предмет дослідження: створення web-додатку з мікросервісною архітектурою з допомогою мови програмування Java.

Завдання випускного кваліфікаційного проекту:

- дослідження існуючих технологій та рішень для створення додатків на базі мікросервісного підходу;
- обгрунтування вибору мікросервісної архітектури, розробка технічного завдання;
- проектування та створення корпоративного web-додатку в середовищі докер с використанням мови програмування Java;
- апробація та тестування розробленого програмного продукту;
- розробка інструкції користувача;
- створення технічного звіту;
- формулювання висновків та пропозицій;

Методи дослідження:

- сучасні методи об'єктно-орієнтованого програмування
- мова програмування Java та фреймворки для цієї мови
- методи розробки мікросервісів,
- платформа для розробки та експлуатації додатків docker
- база даних MySql;
- компілятор Intellij IDEA.

Структура та обсяг роботи. Випускний кваліфікаційний проект складається зі вступу, трьох розділів, висновків та пропозицій, списку використаних джерел. Загальний обсяг роботи – 52 сторінки, які містять загалом 15 рисунків та 1 таблицю.

РОЗДІЛ 1. ТЕХНОЛОГІЇ ПОБУДОВИ WEB-ДОДАТКІВ

З часом вартість підтримки будь-якого за розміром додатку буде зростати. В разі монолітної архітектури цей рост буде набагато більш, ніж при використання мікросервісного підходу, так як розробники будуть мати справу не с одною великою кодовою базою, а с декількома маленькими.

Тому, мікросервісний підхід є більш доцільним при розробці великих web-додатків і з часом – при життевій підтримці продукту, ці перевагу будуть все більш помітні.

РОЗДІЛ 2. ОСНОВНІ ПІДХОДИ І ТЕХНОЛОГІЇ РОЗРОБКИ МІКРОСЕРВІСНИХ WEB-ДОДАТКІВ

Мікросервісна архітектура – прекрасний вибір для розробки великих web-додатків через те, що велика кількість людей може одночасно працювати у різних командах над різними сервісами ніяк не заважаючи один одному. Для розробки можуть використовуватися різні мови та технології. Мікросервіси дуже легко масштабувати, так як вони ізольовані від всього та підстроювати під структуру своєї організації.

Розроблено технічне завдання на створення web-додатку з мікросервісною архітектурою з використанням технологій середовища Docker, мови програмування Java та фреймворку Spring Boot.

РОЗДІЛ 3. РОЗРОБКА WEB-ДОДАТКУ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

З використанням бібліотеки Element, було розроблено зручний та зрозумілий інтерфейс для користувача клієнтської частини ІС. Щоб візуалізувати розпорядок дня для кожного з кабінетів, було підключено та налаштовано бібліотеку Timetable. Створено інструкцію користування з описом усіх функцій інформаційної системи.

ВИСНОВКИ ТА ПРОПОЗИЦІЇ

Відповідно до поставлених завдань, можна зробити наступні висновки.

В даній роботі після виконання аналізу було досліджено, що для великих за розміром та сильно навантажених додатків мікросервісна архітектура підходить краще інших, оскільки додаток, розроблений на основі цієї архітектури здатний забезпечити велику продуктивність та працювати під великими навантаженнями довгий час. Код мікросервісів набагато легше підтримувати та доповнювати новою функціональністю, порівняно с монолітним додатком, оскільки код всього додатку розбитий на логічні мікросервіси з своєю кодовою базою, тому одночасно декілька команд розробників можуть працювати над певним мікросервісом без необхідності знати, як працюють інші мікросервіси.

Результатом даної роботи є створений корпоративний web-додаток у середовищі докер з використання мови програмування Java. Результатом його апробації є підтвержений висновок , що мікросервісна архітектура є раціональним вибором при розробці навантажених web-додатків.

Щодо подальшого вдосконалення система може бути доповнена системою моніторингу. Додання системи збирання метрик Prometheus, та систему відображення зібраних метрик Grafana.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sam Newman – Monolith to microservices - O'Reilly Media; 2019 – 44 с.
2. Sam Newman – Building microservices - O'Reilly Media; 2014 – 17 с.
3. Morgan Kaufmann - Virtual Machines: Versatile Platforms for Systems and Processes; 1 edition, 2005 – 51 с.
4. Herbert Schildt - Java: The Complete Reference, Ninth Edition. / Herbert Schildt - McGraw-Hill Education; 9 edition, 2014 – 1312 с.
5. Безпека мікросервісних web-додатків: журнал Безпека соціально-економічних процесів в кіберпросто / Александренко А.А, Рассамакін В.Я. – К.: Київ. нац. торг. н-т, 2019 – 201-202 с.
6. Software engineering комп'ютерних систем - зб. наук. ст. студ. відп. ред. – Александренко А.А. Київ. Київ. нац. торг. -екон. ун-т. 2019 6-10 с.

Інтернет ресурси

7. Docker – Why docker? [Електронний ресурс]. – Режим доступу: https://www.docker.com/why-docker .
8. Microservices benefits [Електронний ресурс]. – Режим доступу : https://skelia.com/articles/5-major-benefits-microservice-architecture/
9. Spring Boot - Microservices with Spring [Електронний ресурс]. – Режим доступу: https://spring.io/blog/2015/07/14/microservices-with-spring.
10. Maven - What is maven [Електронний ресурс]. – Режим доступу : https://maven.apache.org/what-is-maven.html.
11. Microservices – What are microservices? [Електронний ресурс]. – Режим доступу: [Електронний ресурс]. – Режим доступу: https://microservices.io/.

РЕЦЕНЗІЯ

на випускну кваліфікаційну роботу студента факультету обліку, аудиту та інформаційних систем 2м курсу, 5 групи Александренко Андрія Анатолійовича на тему:

«Проектування мікросервісної архітектури побудови корпоративних web-додатків»

Актуальність теми, представленої на рецензію випускної кваліфікаційної роботи визначається дослідженням та визначенням найбільш досконалої та кращої архітектури корпоративних web-додатків, з урахуванням того фактору, що розроблений програмний продукт потребує постійної підтримки та оновлення після його виходу в світ.

Метою власника продукту є отримання більшого прибутку за менших витрат - це можливо лише за правильно спроектованої архітектури додатку, так як легкість внесення змін до існуючих бізнес-функцій та простота додавання нових бізнес-функцій до існуючої системи є основними витратами при пітримці додатку після його релізу.

В даній роботі чітко визначені актуальність, мета, об’єкт, предмет та завдання дослідження.

Проаналізовані концептуальні особливості, та методи розробки web-додатків. Визначені та здійснений порівняльний аналіз існуючих видів архітектур, їх вплив на вартість підтримки та розробки нового функціоналу після релізу додатку.

На основі проведеного аналізу, з урахуванням визначеної загальної концепції, щодо створення web-додатків здійснена розробка технічного завдання і її подальша успішна реалізація.

Матеріали випускної кваліфікаційної роботи викладені автором логічно і послідовно, з аргументованими висновками і дотриманням науково-технічного стилю викладення.

В роботі приведений список використаних літературних джерел.

Вагомим є те, що результати проведених досліджень пройшли апробацію на Всеукраїнській науково-практичнії конференції, за результатами якої студент Александренко А.А. має 1 публікацію.

Суттєвих зауважень по змісту та недоліків по оформленню робота не має.

В цілому можна констатувати, що за своїм змістом і отриманим результатам випускна кваліфікаційна робота студента Александренко А.А. виконана на високому рівні відповідає існуючим вимогам та може бути допущена до захисту, а її автору присвоєна кваліфікація освітнього ступеня «магістр» напряму підготовки 6.050103 «Програмна інженерія».

Рецензент,
Науковий керівник
кандидат технічниих наук, Демідов П.Г.
доцент