Київський національний торговельно-економічний університет
Кафедра інженерії програмного забезпечення та кібербезпеки

ВИПУСКНИЙ КВАЛІФІКАЦІЙНИЙ ПРОЕКТ

на тему:

«Розробка інформаційної системи на основі мікросерверної архітектури з використанням Cloud-середовища»

Студента 2 курсу, 6 групи,
спеціальності 121
«Інженерія програмного
забезпечення»,
спеціалізації «Інженерія
програмного забезпечення» _____ Мосійчука Євгена
підпис студента Володимировича

Науковий керівник
кандидат педагогічних наук,
старший викладач кафедри
інженерії програмного
забезпечення та кібербезпеки _____ Жирова Тетяна
підпис керівника Олександрівна

Гарант освітньої програми
доктор технічних наук,
професор кафедри інженерії
програмного забезпечення та
кібербезпеки _____ Криворучко Олена
підпис гаранта Володимирівна

КИЇВ-2020

Київський національний торговельно-економічний університет

Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення та кібербезпеки
Освітній ступінь магістр
Спеціальність 121 «Інженерія програмного забезпечення»

Затверджую
Зав. кафедри інженерії програмного забезпечення та кібербезпеки
Криворучко О. В.
8 листопада 2019 р.

Завдання
на випускний кваліфікаційний проект студентові

Мосійчуку Євгену Володимировичу
(прізвище, ім'я, по батькові)

1. Тема випускного кваліфікаційного проекту *«Розробка інформаційної системи на основі мікросерверної архітектури з використанням Cloud-середовища»*

Затверджена наказом ректора від "13" грудня 2019 р. № 4304

2. Строк здачі студентом закінченого проекту 01 грудня 2020

3. Цільова установка та вихідні дані до проекту

Мета проекту теоретичне обґрунтування стратегій і напрямків розвитку мікросервісної архітектури і моделей систем сервісів для створення конкретних прикладних додатків у зв'язку з загальною тенденцією переносу ІТ-рішень у хмарне середовище.

Об'єкт дослідження розміщення і контейнеризація додатків з мікросервісною архітектурою у хмарному середовищі.

Предмет дослідження розробка додатку на основі мікросервісної архітектури та розгорнення його у хмарному середовищі.

4. Консультанти проекту із зазначенням розділів, які консультують:

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

5. Зміст випускного кваліфікаційного проекту (перелік питань за кожним розділом)

ВСТУП

РОЗДІЛ 1 ОСОБЛИВОСТІ ПОБУДОВИ СУЧАСНИХ ВЕБ ДОДАТКІВ

1.1. МОНОЛІТНА АРХІТЕКТУРА

1.2. СЕРВІСНО-ОРІЄНТОВАНА АРХІТЕКТУРА

1.3. МІКРОСЕРВІСНА АРХІТЕКТУРА

1.4. ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА АРХІТЕКТУРНИХ ПІДХОДІВ

1.5. ВИСНОВКИ ДО РОЗДІЛУ 1

РОЗДІЛ 2 АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1. ХАРАКТЕРИСТИКА І ПРИНЦИПИ ПОБУДОВИ МІКРОСЕРВІСІВ

2.2. ВИЯВЛЕННЯ СЕРВІСІВ

2.3. СТРАТЕГІЇ РОЗГОРТАННЯ ДОДАТКІВ

2.4. ВИСНОВОК ДО РОЗДІЛУ 2

РОЗДІЛ 3 ВИБІР ПРОГРАМНИХ ЗАСОБІВ ТА ПРОЕКТУВАННЯ МІКРОСЕРВІСНОГО ДОДАТКУ

3.1. ПРОГРАМНІ ЗАСОБИ ДЛЯ СТВОРЕННЯ СЕРВЕРНОЇ ЧАСТИНИ

3.2. ПРОГРАМНІ ЗАСОБИ ДЛЯ СТВОРЕННЯ КЛІЄНТСЬКОЇ ЧАСТИНИ

3.3. КОНТЕЙНЕРИЗАЦІЯ ДОДАТКІВ ЗА ДОПОМОГОЮ DOCKER

3.4. АРХІТЕКТУРА ДОДАТКУ

3.5. ХМАРНЕ СЕРЕДОВИЩЕ AMAZON WEB SERVICES

3.6. ВИСНОВКИ ДО РОЗДІЛУ 3

РОЗДІЛ 4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ

4.1 ПРОГРАМНА РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ ЧАСТИНИ

6. Календарний план виконання проекту

№ пор.	Назва етапів випускного кваліфікаційного проекту	Строк виконання етапів проекту	
		за планом	фактично
1	2	3	4
1.	*Вибір теми випускної кваліфікаційної роботи*	*20.09.2019*	*20.09.2019*
2.	*Розробка та затвердження завдання на проект магістра*	*08.11.2019*	*08.11.2019*
3.	*Вступ та перелік літературних джерел*	*24.01.2020*	*24.01.2020*
4.	*Наукова стаття*	*01.09.2020*	*01.09.2020*
5.	*Технічне завдання*	*28.02.2020*	*28.02.2020*
6.	*Розділ 1. Особливості побудови сучасних веб додатків*	*25.06.2020*	*25.06.2020*
7.	*Розділ 2. Аналіз мікросервісної архітектури*	*07.09.2020*	*07.09.2020*
8.	*Розділ 3. Вибір програмних засобів та проектування мікросервісного додатку*	*19.10.2020*	*19.10.2020*
9.	*Розділ 4. Опис програмної реалізації інформаційної системи*	*20.10.2020*	*20.10.2020*
10.	*Програма та методика тестування*	*21.10.2020*	*21.10.2020*
11.	*Керівництво користувача*	*23.10.2020*	*23.10.2020*
12.	*Висновки та пропозиції*	*30.10.2020*	*30.10.2020*
13.	*Здача випускного кваліфікаційного проекту на кафедру (перша перевірка)*	*05.11.2020*	*05.11.2020*
14.	*Підготовка автореферату та презентації доповіді*	*05.11.2020*	*05.11.2020*
15.	*Попередній захист випускного кваліфікаційного проекту*	*25.11.2020-27.11.2020*	*25.11.2020-27.11.2020*
16.	*Зовнішнє рецензування випускної кваліфікаційної роботи*	*01.12.2020*	*01.12.2020*
17.	*Підготовка до публічного захисту випускної кваліфікаційної роботи*	*10.12.2020-11.12.2020*	

7. Дата видачі завдання «8» листопада 2019 р.

8. Науковий керівник випускного кваліфікаційного проекту Жирова Т. О.
(прізвище, ініціали, підпис)

9. Гарант освітньої програми Криворучко О. В.
(прізвище, ініціали, підпис)

10. Завдання прийняв до виконання студент Мосійчук Є. В.
(прізвище, ініціали, підпис)

11. Відгук керівника випускного кваліфікаційного проекту

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Науковий керівник випускного кваліфікаційного проекту

_____
(підпис, дата)

Відмітка про попередній захист _____Жирова Т. О._____
(ПІБ, підпис, дата)

12. Висновок про випускний кваліфікаційний проект

Випускний кваліфікаційний проект студента __Мосійчука Є. В.__
(прізвище, ініціали)

може бути допущена до захисту екзаменаційній комісії.

Гарант освітньої програми _____Криворучко О. В._____
(прізвище, ініціали, підпис)

Завідувач кафедри _____Криворучко О. В._____
(підпис, прізвище, ініціали)

«_____»_____20 ___ р.

АНОТАЦІЯ

Дана робота присвячена дослідженню основних підходів до побудови мікросервісної архітектури. Метою роботи є дослідження існуючих концепцій для створення мікросервісних систем, ознайомлення з наявними програмними платформами та інструментами, які дозволяють швидко та ефективно розгортати дані системи. А також реалізація тестового додатка для підтвердження працездатності даного архітектурного підходу.

Розробка серверної частини виконана у середовищі розробки IntelliJ IDEA, а клієнтської – в JetBrains WebStorm. Обрана мова програмування для серверної частини – Java, клієнтська частина розроблена на React JS. Збирання проектів виконувалося за допомогою інструмента автоматизації Gradle та Webpack. Контейнеризація додатку була здійснена за допомогою Docker та розгорнуто за допомогою хмарного провайдера Amazon Web Services.

Готовий програмний комплекс CinemaBooking було успішно протестовано відповідно до функціональних вимог.

Ключові слова: мікросервісна архітектура, хмарне середовище, контейнеризація, розгортання додатків.

ABSTRACT

This work is devoted to the study of basic approaches to the construction of microservice architecture. The aim of the work is to study the existing concepts for the creation of microservice systems, to get acquainted with the existing software platforms and tools that allow you to quickly and efficiently deploy these systems. As well as the implementation of a test application to confirm the efficiency of this architectural.

The server side development was done in IDE IntelliJ IDEA and the client partition – in JetBrains WebStorm. The chosen programming language for the server part is Java, the client part is developed with React JS. Project packaging was done

using the automation tool Gradle and Webpack. The application was containerized using Docker and deployed using the cloud provider Amazon Web Services.

The CinemaBooking software package was successfully tested according to functional requirements.

Keywords: microservices architecture, cloud environment, containerization, application deployment.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SOA – сервіс-орієнтована архітектура.

DDD – Предметно-орієнтоване проєктування.

API – Прикладний програмний інтерфейс.

REST – англ. Representational State Transfer, «передача репрезентативного стану».

VM – віртуальна машина.

ВСТУП

Тема дипломної роботи «Розробка інформаційної системи на основі мікросерверної архітектури з використанням Cloud-середовища». Цілю дипломної роботи є розробка клієнт серверного застосування онлайн резервації квитків до кінотеатру на основі мікросервісної архітектури та розгорнути додаток в хмарному середовищі.

Актуальність. Інформаційні технології – це галузь знань, яка пропонує широкий спектр інструментів, фреймворків та мов програмування. Крім того, усі вони миттєво вдосконалюються та модернізуються, щоб відповідати вимогам сучасних клієнтів та ринку. Вибір правильних інструментів може допомогти легко виконати завдання та принести користь як програмісту, так і клієнту; однак, з іншого боку, через поганий вибір технологій компанія може втратити чимало часу та коштів. Тому вибір технологій та архітектурних рішень важливий як ніколи.

В останні роки тенденція використання мікросервісів набула популярності, коли підприємства прагнуть, щоб їх програмне забезпечення було більш гнучким та масштабованим. У мікросервісній архітектурі додаток розбивається на колекцію слабко пов'язаних служб.

Використовуючи мікросервісний підхід можна: збільшити доступність бізнес-додатків в кілька разів; протягом тижня встановлювати десятки оновлень, не ризикуючи порушити працездатність корпоративної інформаційної системи; розробляти оновлення паралельно силами кількох команд, що створюють нові версії незалежно один від одного.

Мета дослідження: теоретичне обґрунтування стратегій і напрямків розвитку мікросервісної архітектури і моделей систем сервісів для створення конкретних прикладних додатків у зв'язку з загальною тенденцією переносу IT-рішень у хмарне середовище.

Об'єкт дослідження: розміщення і контейнеризація додатків з мікросервісною архітектурою у хмарному середовищі.

Предмет дослідження: розробка додатку на основі мікросервісної архітектури та розгорнення його у хмарному середовищі.

Відповідно до мети дослідження поставлені наступні *завдання*:

1) огляд існуючих стратегій створення сучасних веб-додатків;

2) дослідити технологію віртуалізації та контейнеризації додатків для подальшого розгортання в хмарному середовищі.

3) обґрунтувати вибір технологій і засобів для створення мікросервісної архітектури, яка слугуватиме платформою для майбутнього додатка;

4) на основі отриманих даних створити тестовий веб додаток для демонстрації основних засад мікросервісної архітектури.

5) розгорнути додаток в хмарному середовищі і описати основні проблеми з якими може зіштовхнутися розробники програмного забезпечення.

РОЗДІЛ 1.
ОСОБЛИВОСТІ ПОБУДОВИ СУЧАСНИХ ВЕБ-ДОДАТКІВ

1.1. Монолітна архітектура

У програмній інженерії моноліт відноситься до єдиної неподільної одиниці. Концепція монолітного програмного забезпечення полягає в тому, що різні компоненти програми поєднуються в єдину програму на одній платформі. Зазвичай, монолітний додаток складається з бази даних, користувацького інтерфейсу на стороні клієнта та додатка на стороні сервера. Усі частини програмного забезпечення уніфіковані, а всі його функції керуються в одному місці.

Монолітна архітектура зручна для роботи невеликим командам, саме тому багато стартапів обирають такий підхід при створенні програми. Компоненти монолітного програмного забезпечення взаємопов'язані та взаємозалежні, що допомагає програмному забезпеченню бути самостійним. Ця архітектура є традиційним рішенням для побудови додатків, але деякі розробники вважають його застарілим.

Серед переваг монолітного підходу можна назвати:

– легший процес розробка та розгортання додатку. Існує безліч інструментів, які можна інтегрувати для полегшення розробки. Крім того, всі дії виконуються з одним каталогом, що забезпечує полегшення розгортання. Маючи монолітне ядро, розробникам не потрібно розгортати зміни або оновлення окремо, оскільки вони можуть це робити відразу та економити багато часу;

– менше проблем з наскрізною функціональністю, такою як аудиторські перевірки, запису логів, обмеження швидкості тощо. Завдяки єдиній кодовій базі монолітні програми імплементують ці функції набагато простіше;

– краща продуктивність. При правильній побудові монолітні програми, як правило, ефективніші, ніж програми на базі мікросервісів. Додатку з мікросервісною архітектурою може знадобитися зробити 40 викликів API для 40 різних мікросервісів, щоб завантажити кожен екран, що, очевидно, призводить до зниження продуктивності. Монолітні програми, у свою чергу, дозволяють пришвидшити зв'язок між програмними компонентами завдяки спільному коду та пам'яті.

Недоліки монолітної архітектури:

– з часом більшість продуктів розвиваються і збільшуються, і їх структура стає розмитою. База коду починає виглядати громіздкою і стає важкою для розуміння та модифікації, особливо для нових розробників. Також стає важче знайти побічні ефекти та залежності. Зі зростанням бази коду знижується якість, а інтегроване середовище розробки (IDE) стає перевантаженим;

– важко застосовувати нові технології. Якщо потрібно додати якусь нову технологію до свого додатка, розробники можуть зіткнутися з перешкодами для прийняття. Додавання нової технології означає перезапис усієї програми, що є дорогим та трудомістким.

– обмежена гнучкість. Потрібно знову повністю розгортати додаток після кожного оновлення.

1.2. Сервісно-орієнтована архітектура

SOA (service-oriented architecture - сервісно-орієнтована архітектура) - це архітектурний підхід для визначення, зв’язування та інтеграції багаторазових бізнес сервісів, які мають чіткі межі та свою функціональну зону відповідальності. Подібні сервіси взаємодіють між собою, щоб забезпечити просту передачу даних, може включати два або більше сервісів. Складність

кожного сервісу в SOA, як правило, дуже низька, і вони спілкуються між собою за допомогою набору API.

SOA (service-oriented architecture - сервісно-орієнтована архітектура) - це архітектурний підхід для визначення, зв’язування та інтеграції багаторазових бізнес сервісів, які мають чіткі межі та свою функціональну зону відповідальності. Подібні сервіси взаємодіють між собою, щоб забезпечити просту передачу даних, може включати два або більше сервісів. Складність кожного сервісу в SOA, як правило, дуже низька, і вони спілкуються між собою за допомогою набору API.

Кожен сервіс в SOA містить інтеграцію коду та даних, необхідних для виконання повної, дискретної бізнес функції (наприклад, перевірка кредиту клієнта, обчислення щомісячного платежу за позикою або обробка заявки на іпотеку). Сервісні інтерфейси забезпечують вільне з'єднання, тобто їх можна викликати, мінімально знаючи про те, як реалізується інтеграція на нижчих рівнях, або взагалі не знаючи. Сервіси використовуються за допомогою стандартних мережевих протоколів - таких як SOAP / HTTP або JSON / HTTP - для надсилання запитів на читання або зміну даних. Служби публікуються таким чином, що дозволяє розробникам швидко знаходити їх і використовувати їх повторно для складання нових додатків.

Переваги сервісно-орієнтованої архітектури:

– Повторне перевикористання сервісів. Завдяки самодостатньому та слабо пов'язаному характеру функціональних компонентів у сервісно-орієнтованих додатках, ці компоненти можуть бути використані повторно в декількох додатках, не впливаючи на інші сервіси;

– Легше підтримувати. Оскільки кожен сервіс є незалежною одиницею, його легко оновлювати та підтримувати, не завдаючи шкоди іншим сервісам. Наприклад, великими корпоративними програмами можна простіше керувати, якщо їх розбити на сервіси;

– Висока надійність. Невеликі сервіси легше налагоджувати та тестувати, ніж величезні фрагменти коду, як у монолітному підході. Це, в свою чергу, робить продукти на основі SOA більш надійними;

– Можливість паралельної розробки. Оскільки сервісно-орієнтована архітектура складається з шарів, вона підтримує паралельність у процесі розробки. Незалежні сервіси можна розвивати паралельно і одночасно завершувати.

Недоліки сервісно-орієнтованої архітектури:

– Складне управління. Основним недоліком сервісно-орієнтованої архітектури є її складність. Кожен сервіс повинен забезпечити своєчасну доставку повідомлень. Кількість цих повідомлень одночасно може перевищувати мільйон, що робить великим викликом оркестрацію всіх сервісів;

– Великі інвестиційні витрати. Розробка SOA вимагає великих попередніх вкладень людських ресурсів, технологій та розробки;

– Більше навантаження і збільшений час відгуку. У SOA кожна взаємодія між службами супроводжується повною валідацією всіх вхідних параметрів. В результаті навантаження стає надзвичайно важким, що, в свою чергу, подовжує час відгуку.

1.3. Мікросервісна архітектура

Мікросервіси – це архітектурний підхід до створення додатків. Як архітектурний фреймворк, мікросервіси розподіляються та вільно поєднуються, тому зміни однієї команди не порушать всю програму. Вигода від використання мікросервісів полягає в тому, що команди розробників можуть швидко створювати нові компоненти програм для задоволення мінливих потреб бізнесу.

Мікросервіси вирішують виклики монолітних систем, будучи максимально модульними. У найпростішій формі вони допомагають створити додаток як набір невеликих сервісів, кожен з яких працює у своєму власному процесі та може бути розгорнутий незалежно. Ці сервіси можуть бути написані різними мовами програмування та можуть використовувати різні методи

зберігання даних. Хоча це призводить до розробки масштабованих та гнучких систем, вона потребує динамічного перетворення. Мікросервіси часто підключаються через API і можуть використовувати багато однакових інструментів та рішень, які виросли в екосистемі RESTful та вебсервісу.

Переваги мікросервісної архітектури:

– Різноманіття технологій. За допомогою системи, що складається з декількох сервісів, що співпрацюють, є можливість використовувати різні технології всередині кожного. Це дозволяє підібрати правильний інструмент для кожної задачі;

– Масштабування. У випадку використання монолітної архітектури необхідно масштабувати цілу систему. Використовуючи мікросервісну архітектуру, можна обійтися масштабуванням лише тих сервісів, які цього потребують. Таким чином, компанія може зекономити чимало грошей та апаратних ресурсів.

– Зручність використання в хмарному середовищі. На цей момент існує чимало хмарних платформ, які надають готові рішення і послуги для розгортання програмного забезпечення різної величини і складності. Дані платформи надають широкий спектр послуг, а головне допомагають автоматизувати та полегшити процеси під час розробки, розгортання та моніторингу додатків. Це дозволяє якомога швидше приносити прибуток для замовників.

Серед недоліків можна назвати:

– Тестування мікросервісної архітектури стає складнішим і виснажливим;

– Дана архітектура вносить додаткову складність, оскільки розробникам доводиться перейматися через відмовостійкість, затримки в мережі та мати справу з різноманітними форматами комунікації між сервісами, а також з балансуванням навантаження;

– Імплементація розподіленої системи вимагає більше зусиль і досвіду розробників;

– Зі збільшенням кількості сервісів інтеграція та управління цілою системою значно ускладнюється;

– Оптимізація до змін. Оскільки індивідуальні сервіси мають невеликі розміри, значно легше керувати витратами на заміну їх на кращу реалізацію або взагалі видалення.

– Завдання, які потребують взаємодії декількох сервісів без використання розподілених транзакції мало того, що є саме по собі складним завданням, також потребує комунікації між командами, які володіють сервісами.

1.4. Порівняльна характеристика архітектурних підходів

Розглянувши вище наведені особливості архітектурних підходів до проектування інформаційних систем, створимо порівняльну характеристику за визначеними критеріями.

Теорема CAP (теорема Брюєра) –твердження про те, що в будь-якій реалізації розподілених обчислень можливо забезпечити не більше двох з трьох наступних властивостей (табл. 1.1.):

– узгодженість даних (consistency) – гарантія того, що кожен вузол розподіленого кластера повертає однаковий, найновіший, успішний запис;

– доступність (availability) – кожен невідмовний вузол повертає відповідь на всі запити на читання та запис за резонний проміжок часу;

– стійкість до розподілу (partition tolerance) – система продовжує функціонувати і підтримує гарантії узгодженості, незважаючи на розподіленість системи на кілька ізольованих сегментів.

Таблиця 1.1

ХАРАКТЕРИСТИКА CAP

Архітектура	**Узгодженість**	**Доступність**	**Стійкість до розподілу**

Моноліт	Так	Ні	Ні
SOA	Ні	Так	Так
Мікросервіс	Ні	Так	Так

Вертикальне масштабування (scaling up) – збільшення кількості доступних для ПО ресурсів шляхом збільшення потужності з серверів. У цьому випадку перевагу віддають потужнішому апаратному забезпеченню.

Горизонтальне масштабування (scaling out) – збільшення кількості серверів, об'єднаних в кластер серверів або процесів за рахунок ОС.

У табл. 1.2. показано допустимість типу масштабування у залежності від виду архітектури.

Таблиця 1.2.

Характеристика по типу масштабування

Архітектура	**Вертикальне**	**Горизонтальне**
Моноліт	Так	Ні
SOA	Так	Так
Мікросервіс	Так	Так

У табл. 1.3. вказана приблизна оцінка вимог апаратних ресурсів інфраструктури до архітектурного рішення.

Позначення: S – small (невисокі вимоги), M – middle (середні вимоги), L – large (значні вимоги).

Типи ресурсів:

– Пам'ять (RAM);

– Ресурси процесора (CPU);

– Місце на диску (HDD / SSD);

– Комунікація (LAN).

Таблиця 1.3.

Характеристика по типам ресурсів

Архітектура	**RAM**	**CPU**	**HDD/SSD**	**LAN**
Моноліт	M	M	M	S
SOA	L	L	S	L
Мікросервіс	L	L	S	L

Таблиця 1.4.

ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА АРХІТЕКТУРНИХ ПІДХОДІВ

	Мікросервіси	**SOA**	**Моноліт**
1	**2**	**3**	**4**
Архітектура	Сервіси побудовані з невеликих компонентів і формально виражаються за допомогою бізнес-орієнтованих API.	Сервіси можуть варіюватися в розмірах, починаючи від невеликих додатків і закінчуючи дуже великими корпоративними.	Монолітні програми з часом стають доволі громіздкими і виникає ситуація, коли розуміння всієї програми в цілому стає важким завданням.
Зручність використання	Сервіси, які використовуються за допомогою стандартного протоколу, наприклад RESTful API, та використовуються іншими сервісами та програмами.	Сервіси, які використовуються за допомогою стандартного протоколу, наприклад SOAP, і використовуються іншими сервісами – використовують проміжне програмне забезпечення обміну повідомленнями.	Обмежене повторне використання здійснюється в монолітних додатках.

Масштабованість	Сервіси існують як незалежні артефакти розгортання і можуть масштабуватися незалежно від інших сервісів.	Залежність між сервісами та підкомпонентами може спричинити проблеми масштабування.	Масштабування монолітних додатків часто може бути проблемою.

Продовження таблиці 1.4.

1	2	3	4
Гнучкість	Менш незалежні блоки полегшують управління збіркою / релізом, тим самим, надаючи високу операційну гнучкість.	Залежність між сервісами та підкомпонентами може спричинити проблеми масштабування.	Масштабування монолітних додатків часто може бути проблемою.
Розробка	Розробка сервісів дискретно дозволяє розробникам використовувати необхідні засоби для розробки задля вирішення поставленого завдання.	Багаторазові компоненти та стандартні практики допомагають розробникам у реалізації.	Монолітні програми реалізуються за допомогою єдиного стека розробок (тобто JEE або .NET), що може обмежити доступність «правильного інструменту для роботи».

1.5. Висновки до розділу 1

Високі темпи змін і висока складність можуть бути факторами, які змушують вибрати архітектуру мікросервіса. Можливість безперервно розробляти та розгортати великі, складні програми буде тільки сприяти активному росту бізнесу.

В свою чергу, коли ще немає чіткої предметної області, почати з моноліту буде правильним рішенням, проте варто робити служби модульними. Це полегшить завдання, якщо буде вирішено розділити моноліт на кілька мікросервісів.

РОЗДІЛ 2.
АНАЛІЗ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1. Характеристика і принципи побудови мікросервісів

Архітектура програмного додатка – це його високий рівень структура, яка складається із компонентів та залежностей між ними. Архітектура програми є багатовимірною, тому існує кілька способів описати її. Причина, по якій архітектура важлива, полягає в тому, що вона визначає атрибути якості програмного забезпечення програми. Традиційно метою архітектури є масштабованість, надійність та безпека. Але сьогодні важливо, щоб архітектура також забезпечувала швидку та безпечну розробку програмного забезпечення. Архітектура мікросервісу – це архітектурний стиль, який надає додатку високу ремонтопридатність, незалежність розгортання та тестування компонентів.

Сервіс – це самостійний, програмний компонент, який можна незалежно розгорнути та який реалізує деякі корисні функції. Сервіс має API, який забезпечує доступ клієнтів до його функціональності. Існує два типи операцій: команди та запити. API складається з команд, запитів та подій. Команда створені, щоб виконувати дії та оновляти дані. Запит – це операція для отримання даних. Сервіс також може публікувати події, які споживають його клієнти. API сервісу інкапсулює його внутрішню реалізацію. Таким чином, архітектура мікросервісу забезпечує модульність програми.

Декомпозиція згідно з можливостями бізнесу

Однією зі стратегій створення архітектури мікросервісу є декомпозиція за можливостями бізнесу. Згідно з концепцією моделювання бізнес-архітектури, бізнес-можливості - це те, що бізнес робить, щоб генерувати цінність. Набір

можливостей для даного бізнесу залежить від виду діяльності. Наприклад, можливості страхової компанії, як правило, включають андерайтинг, управління претензіями, виставлення рахунків, дотримання вимог тощо. Можливості інтернет-магазину включають управління замовленнями, управління складами, доставка тощо.

Бізнес можливості організації визначаються шляхом аналізу цілі, структури та бізнес-процесів організації. Кожен ділову можливість можна сприймати як послугу як сервіс. Специфікація сервісу складається з різних компонентів, включаючи вхідні дані, результати та опис міжсервісної взаємодії. Наприклад, для страхового андерайтингу вкладом є заявка споживача, а результатом є схвалення та ціну за послугу.

Ділові можливості часто орієнтовані на конкретний бізнес-об'єкт. Наприклад, бізнес-об’єкт «Претензія» є фокусом можливості управління претензіями. Можливість часто можна розкласти на під-можливості. Наприклад, можливість управління претензіями має кілька під-можливостей, включаючи управління інформацією про претензії, перевірку претензій та управління платежами.

Ключовою перевагою організації сервісів навколо бізнес можливостей є те, що оскільки вони стабільні, отримана архітектура також буде відносно стабільною. Окремі компоненти архітектури можуть змінюватися в міру того, як змінюється аспект бізнесу, але архітектура залишається незмінною.

Декомпозиція згідно з субдоменом

DDD (domain-driven design - предметно-орієнтоване проєктування) – це підхід до побудови складних програмних додатків, який зосереджений на розробці об’єктно-орієнтованої моделі домену. Доменний підхід фіксує знання про домен у формі, яка може бути використана для розв'язання проблем у цьому домені. Він визначає словниковий запас, який використовує команда, що DDD називає Ubiquitous Language (єдина мова). Модель домену чітко зображається при розробці та реалізації програми. DDD має дві концепції, які надзвичайно

корисні при застосуванні архітектури мікросервісу: субдомени та обмежений контекст.

DDD значно відрізняється від традиційного підходу до бізнес моделювання, яке створює єдину модель для всього підприємства. У такій моделі існує, наприклад, єдине визначення кожного суб'єкта господарювання, наприклад, замовник, замовлення тощо. Проблема такого моделювання полягає в тому, що отримання згоди від різних частин організації щодо однієї моделі є монументальним завданням. Крім того, це означає, що з точки зору кожної окремої частини організації модель є надто складною для їх потреб. До того, модель домену може заплутати, оскільки різні частини організації можуть використовувати один і той самий термін для різних концепцій або різні терміни для одного і того ж поняття. DDD уникає цих проблем, визначаючи безліч моделей доменів, кожна з явною сферою застосування.

DDD визначає окрему модель домену для кожного субдомену. Субдомени ідентифікуються з використанням того самого підходу, що і визначення ділових можливостей: аналізуйте бізнес та визначайте різні галузі знань. Кінцевий результат, швидше за все, буде субдомени, подібні до ділових можливостей. Приклади субдоменів включають прийняття замовлень, управління замовленнями, управління фінансами.

DDD називає область дії доменної моделі обмеженим контекстом. Обмежений контекст включає артефакти коду, що реалізують модель. При використанні архітектури мікросервісу кожен обмежений контекст є сервісом або, можливо, набором сервісів. Можна створити архітектуру мікросервісу, застосувавши DDD та визначивши сервіси для кожного субдомену. На рисунку 2.1. показано, як субдомени відображаються через сервіси, кожен з яких має свою власну модель домену.

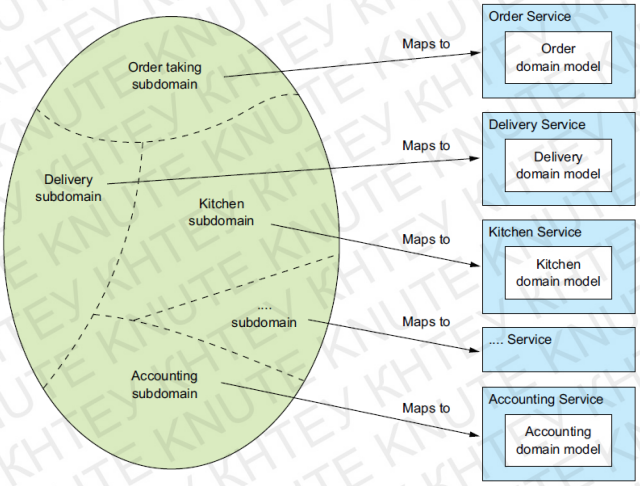

Рис. 2.1. Кожен субдомен домену програми відображається у сервісі, який має власну модель домену

Міжпроцесовий зв'язок в мікросервісній архітектурі

У монолітній програмі компоненти викликають один одного, використовуючи методи рівня мови або виклики функцій. Об'єкти працюють в рамках одного і того ж процесу. Найбільша проблема при переході від монолітної програми до програми на базі мікросервісів полягає у зміні механізму зв'язку. Перехід від викликів методів, що виконуються в процесі, до викликів сервісів через RPC призведе до великої кількості повідомлень та неефективного міжсервісного спілкування, яке буде погано працювати в розподілених середовищах. Проблеми правильного проектування розподіленої системи достатньо відомі, що існує навіть канон, відомий як "Помилки розподілених обчислень", який містить переліки припущень, які розробники часто роблять, переходячи від монолітного до розподіленого дизайну.

Додаток на основі мікросервісів - це розподілена система, що працює на декількох процесах або послугах, зазвичай навіть на декількох серверах або хостах. Кожен екземпляр сервісу, як правило, є процесом. Тому служби повинні взаємодіяти, використовуючи міжпроцесорний протокол зв'язку, такий як HTTP, AMQP або двійковий протокол, такий як TCP, залежно від природи кожного сервісу.

Існує безліч стилів взаємодії між клієнтом та сервісом. Як показано в таблиці 2.1, їх можна класифікувати за двома вимірами. Перший вимір полягає у тому, чи є взаємодія один-до-одного або один-до-багатьох:

– один-до-одного - кожен запит клієнта обробляється рівно однією службою.

– один до багатьох - кожен запит обробляється багатьма службами.

Другий вимір – це синхронна чи асинхронна взаємодія:

– синхронна – клієнт очікує своєчасної відповіді від служби і може бути заблокований, поки чекає;

– асинхронна – клієнт не блокується, і відповідь, якщо така є, не обов'язково негайно надіслана.

Таблиця 2.1.

Характеристика міжсервісної взаємодії

	Один-до-одного	**Один-до-багатьох**
Синхронно	Запит / відповідь	–
Асинхронно	Асинхронний запит / відповідь. Одностороння нотифікація	Публікація / підписка. Публікація / асинхронна відповідь.

Типи взаємодії один-до-одного:

– Запит / відповідь - клієнт послуги робить запит до сервісу і чекає відповіді. Клієнт очікує, що відповідь надійде своєчасно. Це може заблокувати клієнта під час очікування. Це стиль взаємодії, який, як правило, призводить до тісної взаємодії сервісів;

– Асинхронний запит / відповідь - клієнт послуги надсилає запит сервісу, який відповідає асинхронно. Клієнт не блокується, оскільки сервіс може не надсилати відповідь протягом тривалого часу;

– Одностороння нотифікація – клієнт сервісу надсилає запит, але відповіді не очікується і не надсилається.

Нижче наведено різні типи взаємодій один-до-багатьох:

– Публікація / підписка – клієнт публікує повідомлення, яке споживають сервіси які підписалися на повідомлення;

– Публікація / асинхронна відповідь - клієнт публікує повідомлення із запитом, а потім чекає певний час відповідей від зацікавлених служб.

Кожен сервіс, як правило, використовує комбінацію цих стилів взаємодії.

2.2. Виявлення сервісів

У традиційній програмі, що працює на фізичному обладнанні, мережеві розташування екземплярів сервісів, як правило, статичні. Наприклад, додаток може зчитувати розташування мережі з конфігураційного файлу, який час від часу оновлюється. Але в сучасному хмарному мікросервісному додатку все не так просто. Сучасні додатки набагато динамічніші. Екземпляри сервісів мають динамічно призначені мережеві розташування. Більше того, набір екземплярів служби динамічно змінюється через автоматичне масштабування, збої та оновлення. Отже, клієнтський код повинен використовувати виявлення послуги (service discovery).

Існує дві основні схеми виявлення сервісів: виявлення на стороні клієнта та виявлення на стороні сервера.

Під час використання виявлення сервісів на стороні клієнта – клієнт відповідає за визначення мережевих розташувань доступних екземплярів сервісів та балансування навантаження на них. Клієнт робить запит до реєстру сервісів, який є базою даних доступних екземплярів сервісів. Потім клієнт використовує алгоритм балансування навантаження для вибору одного з доступних екземплярів сервісу та робить запит.

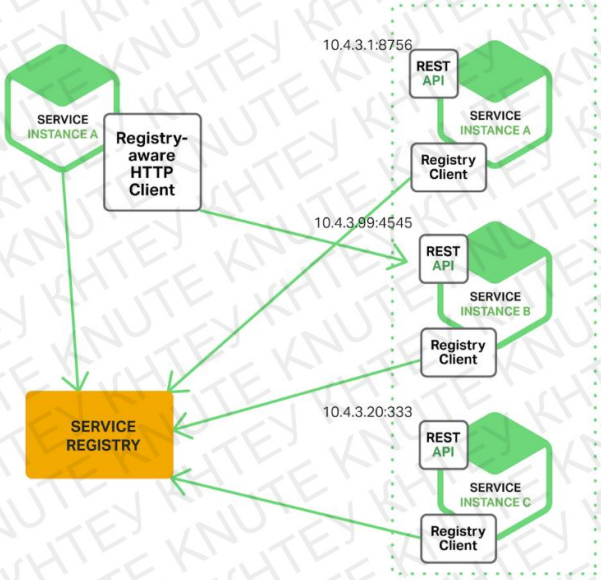

Рис. 2.2. Виявлення сервісів на стороні клієнта

Шаблон виявлення на стороні клієнта має ряд переваг та недоліків. Цей шаблон є відносно простим, і, крім реєстру сервісів, інших складних компонентів немає. Окрім того, оскільки клієнт знає про наявні екземпляри сервісів, він може приймати розумні рішення щодо балансування навантаження, специфічні для програми, наприклад. Одним із суттєвих недоліків цієї моделі є те, що вона створює тісний зв'язок клієнта з реєстром послуг. Також необхідно впровадити логіку виявлення сервісів на стороні клієнта для кожної мови програмування та фреймворку, що використовується клієнтами сервісу.

Інший підхід до виявлення сервісів - це шаблон виявлення на стороні сервера. Наступна схема показує структуру цього шаблону.

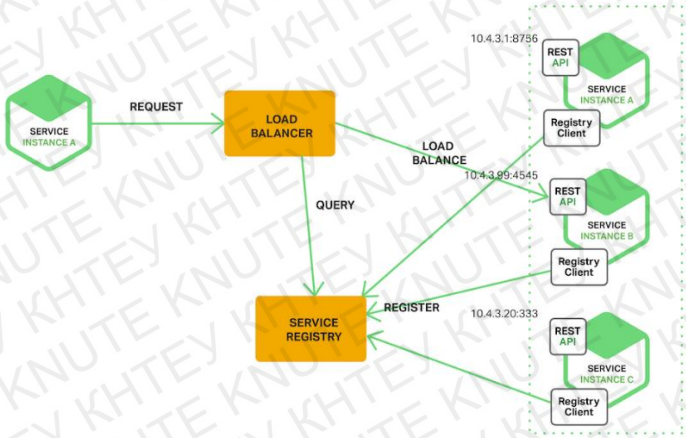

Рис. 2.3. – Виявлення сервісів на стороні серверу

Клієнт робить запит до послуги через балансир навантаження (load balancer). Балансир навантаження запитує реєстр сервісів і направляє кожен запит до доступного екземпляра сервісу. Як і при виявленні на стороні клієнта, облік екземплярів сервісів ведеться в реєстрі служб.

Шаблон виявлення на стороні сервера має ряд переваг та недоліків. Однією з великих переваг цього шаблону є те, що деталі виявлення абстрагуються подалі від клієнта. Клієнти просто роблять запити до балансира навантаження. Це позбавляє від необхідності впроваджувати логіку виявлення для кожної мови програмування та фреймворку, що використовується клієнтами сервісів. Однак цей шаблон також має деякі недоліки. Якщо середовище розгортання не забезпечує балансування навантаження, потрібно імплементувати ще один високо доступний системний компонент, який потрібно налаштувати та керувати ним самостійно.

2.3. Стратегії розгортання додатків

Додаток з мікросервісною архітектурою складається з десятків або навіть сотень послуг. Сервіси написані різними мовами та структурами. Кожен із мікросервісів являє собою міні додаток зі своїми специфічними вимогами до розгортання, ресурсів, масштабованості та моніторингу. Наприклад, потрібно запустити конкретну кількість екземплярів кожного сервісу в залежності від навантаження на цей сервіс. Крім того, кожному екземпляру потрібно забезпечити відповідні ресурси процесора, пам'яті та I/O.

Розгортання мікросервісу має бути швидким, надійним та економічно ефективним.

Стратегія з декількома екземплярами на одному сервері

Одним із способів розгортання мікросервісів є використання декількох екземплярів сервісу на один сервер. Використовуючи цю стратегію, використовують один або декілька фізичних або віртуальних серверів і запускають декілька екземплярів сервісу на кожному. Багато в чому це

традиційний підхід до розгортання додатків. Кожен екземпляр сервісу виконується на добре відомому порту на одному або декількох хостах.

Однією з головних переваг цього шаблону розгортання є відносно ефективне використання ресурсів. Кілька екземплярів сервісу спільно використовують сервер та його операційну систему. Ще ефективніше, якщо процес або група процесів запускає кілька екземплярів сервісу, наприклад, кілька веб-додатків, що використовують один і той же сервер Apache Tomcat та JVM. Ще однією перевагою цього шаблону є те, що розгортання екземпляра сервісу відбувається відносно швидко.

Одним з основних недоліків є те, що ізоляція між екземплярами сервісів майже або повністю відсутня, якщо кожен екземпляр сервісу не є окремим процесом. Хоча можна спостерігати за використанням ресурсів кожним екземпляром сервісу, ви не можете обмежувати ресурси, які використовує кожен екземпляр. Екземпляр, який вийшов з ладу може споживати всю пам’ять або ж усі ресурси центрального процесора хоста.

Ще однією суттєвою проблемою цього підходу є те, що операційна команда, яка розгортає сервіс, повинна знати конкретні деталі, як це зробити. Сервіси можуть бути написані різними мовами та використовувати різні фреймворки, тому існує безліч деталей, якими команда розробників повинна ділитися з операційною командою. Це збільшує ризик помилок під час розгортання.

Віртуалізація на виділеному сервері

Кожен екземпляр сервісу - це віртуальна машина (наприклад, екземпляр EC2), яка запускається за допомогою образу VM. Наступна схема показує структуру цього шаблону:

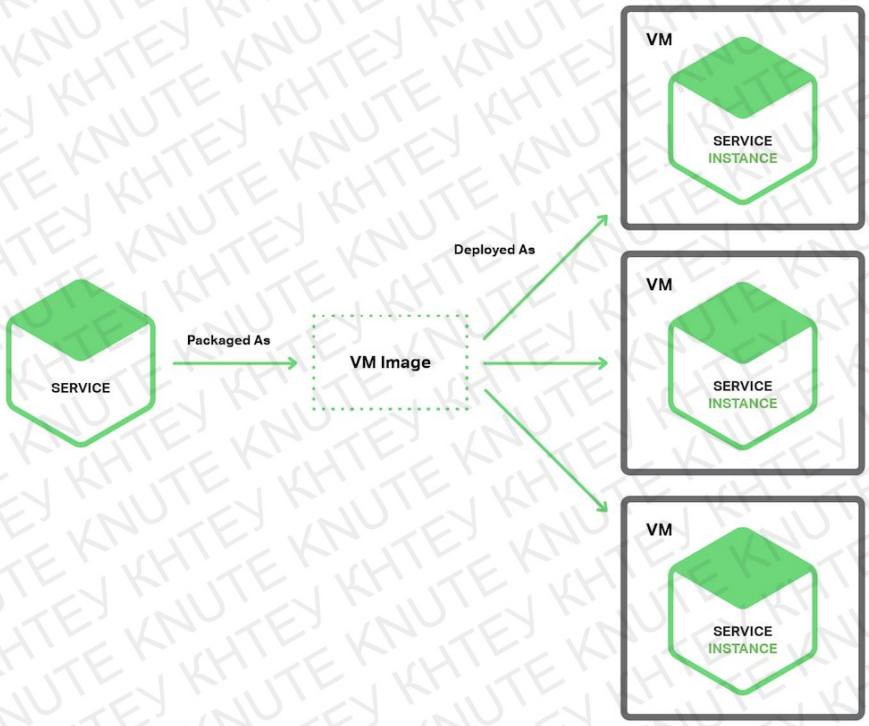

Рис. 2.4. Віртуалізація на виділеному сервері

Це основний підхід, який Netflix використовує для розгортання свого сервісу потокового відео. Netflix пакує кожен зі своїх сервісів як EC2 AMI за допомогою Aminator. Кожен запущений екземпляр служби є екземпляром EC2.

Існує безліч інструментів, які можна використовувати для створення власних віртуальних машин. Можна налаштувати сервер безперервної інтеграції (CI) (наприклад, Jenkins), щоб викликати Aminator для пакетування сервісів як EC2 AMI. Packer.io – ще один варіант автоматизованого створення образів віртуальної машини. На відміну від Aminator, Packer.io підтримує різноманітні технології віртуалізації, включаючи EC2, DigitalOcean, VirtualBox та VMware.

Віртуалізація на виділеному сервері має ряд переваг. Основною перевагою віртуальних машин є те, що кожен екземпляр служби працює в повній ізоляції.

Ще однією перевагою розгортання ваших мікросервісів як віртуальних машин є те, що ви можете використовувати хмарну інфраструктуру. Такі хмарні сервіси, як AWS, надають такі корисні функції, як балансування навантаження та автомасштабування.

Великою перевагою розгортання вашої служби як віртуальної машини є те, що вона інкапсулює технологію реалізації вашої послуги. Після упаковки послуги як віртуальної машини вона стає чорною скринькою. API управління віртуальною машиною стає API для розгортання служби. Розгортання стає набагато простішим і надійнішим.

Однак віртуалізація на виділеному сервері має деякі недоліки. Один недолік - менш ефективне використання ресурсів. Кожен екземпляр служби має накладні витрати на всю віртуальну машину, включаючи операційну систему. Більше того, у типовому загальнодоступному IaaS віртуальні машини мають фіксований розмір, і цілком можливо, що віртуальна машина буде недостатньо використана.

Ще одним мінусом цього підходу є те, що розгортання нової версії служби, як правило, відбувається повільно. Крім того, віртуальні машини, як правило, повільно створюють екземпляри, знову ж через їх розмір. Крім того, для запуску операційної системи зазвичай потрібен певний час.

Контейнеризація на виділеному сервері

При використанні стратегії розгортання екземпляру служби за допомогою контейнеризації, кожен екземпляр служби виконується у своєму власному контейнері. Контейнери - це механізм віртуалізації на рівні операційної системи. Контейнер складається з одного або декількох процесів, що працюють у пісочниці. З точки зору процесів, вони мають власний простір імен портів та кореневу файлову систему. Ви можете обмежити пам’ять контейнера та ресурси ЦП. Деякі реалізації контейнерів також мають обмеження швидкості вводу-виводу. Приклади засобів контейнеризації включають Docker та Solaris Zones.

Рисунок 2.5. показує структуру шаблону контейнеризація на виділеному сервері.

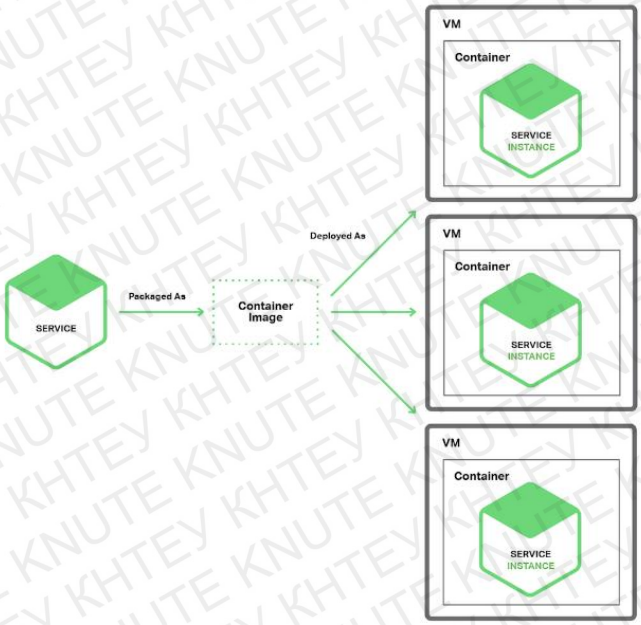

Рис. 2.5. Контейнеризація на виділеному сервері

Щоб використовувати цей шаблон, необхідно упаковувати сервіси як образи контейнера. Образ контейнера - це образ файлової системи, що складається з програм та бібліотек, необхідних для запуску сервісу. Деякі зображення контейнерів складаються з повної кореневої файлової системи Linux. Інші більш легкі. Наприклад, для розгортання служби Java ви створюєте образ контейнера, що містить середовище виконання Java, можливо, сервер Apache Tomcat та скомпільовану програму Java.

Переваги контейнерів подібні до переваг віртуальних машин. Вони ізолюють ваші екземпляри служб один від одного. Ви можете легко контролювати ресурси, що споживаються кожним контейнером. Однак, на відміну від віртуальних машин, контейнери є легкою технологією. Зображення контейнерів зазвичай створюються дуже швидко. Контейнери також запускаються дуже швидко, оскільки відсутній тривалий механізм завантаження ОС. Коли контейнер запускається, запускається служба.

Є деякі недоліки використання контейнерів. Хоча контейнерна інфраструктура швидко розвивається, вона не настільки розвинена, як інфраструктура для віртуальних машин. Крім того, контейнери не настільки

захищені, як віртуальні машини, оскільки контейнери спільно використовують ядро головної ОС.

Ще одним недоліком контейнерів є те, якщо не використовується контейнерне рішення, таке як Google Container Engine або Amazon EC2 Container Service (ECS), необхідно самостійно адмініструвати контейнерну інфраструктуру та, можливо, інфраструктуру віртуальної машини, на якій вона працює.

Крім того, контейнери часто розгортаються в інфраструктурі, яка має ціну за VM. Отже, ймовірно, будуть додаткові витрати на надмірне забезпечення віртуальних машин для обробки стрибків навантаження.

2.4. Висновок до розділу 2

У цьому розділі було проаналізовано стратегії декомпозиції додатків на мікросервіси, основні стратегії розгортання мікросервісів, нюанси міжсервісної взаємодії.

Мікросервісний підхід, дозволяє швидко розробляти та легко масштабувати додатки у залежності від виявленого навантаження та бізнес-вимог, що робить даний шаблон одним із найкращих рішень для бізнесу будь-яких масштабів, на меті у якого стрімкий ріст та розвиток.

РОЗДІЛ 3.
ВИБІР ПРОГРАМНИХ ЗАСОБІВ ТА ПРОЕКТУВАННЯ МІКРОСЕРВІСНОГО ДОДАТКУ

3.1. Програмні засоби для створення серверної частини

Для реалізації серверної частини буде використано мову програмування Java разом з фреймворком Spring, а саме з його компонентами Spring Boot, Spring Data, Spring Security та Spring Cloud Netflix.

Spring Framework забезпечує комплексну модель програмування та конфігурації для сучасних корпоративних програм на базі Java та на будь-якій платформі розгортання.

Також фреймворк Spring надає важливий компонент для розробки мікросервісів – Spring Cloud Netflix, який забезпечує інтеграцію Netflix OSS для програм Spring Boot за допомогою автоконфігурації та прив'язки до Spring Environment та інших ідіом моделі програмування Spring. За допомогою кількох простих анотацій можна швидко увімкнути та налаштувати загальні шаблони у вашому додатку та побудувати великі розподілені системи із перевіреними боями компонентами Netflix. Запропоновані шаблони включають виявлення служб (Eureka), автоматичний вимикач (Hystrix), інтелектуальну маршрутизацію (Zuul) та балансування навантаження на стороні клієнта (Ribbon).

Eureka – це сервіс, що базується на REST (Representational State Transfer), який в основному використовується в хмарі AWS для пошуку служб з метою балансування навантаження та відновлення після відмови серверів проміжного рівня.

Hystrix – це бібліотека для контролю за мережевими затримками та відмовами сервісів, призначена для ізоляції точок доступу до віддалених систем, служб та сторонніх бібліотек, зупинки каскадного збою та забезпечення стійкості в складних розподілених системах, де збій неминучий.

Zuul – це маршрутизатор на базі JVM та балансир навантаження на стороні сервера від Netflix, яка забезпечує динамічну маршрутизацію, моніторинг, стійкість, безпеку тощо.

Ribbon - це бібліотека IPC на стороні клієнта, яка перевірена у хмарі. Він надає наступні функції:

– Балансування навантаження;
– Відмовостійкість;
– Підтримка декількох протоколів (HTTP, TCP, UDP) в асинхронній та реактивній моделі;
– Кешування.

Для автоматизації збірки та пакування мікросервісів використовувався Gradle - це інструмент автоматизації побудови для розробки багатомовного програмного забезпечення. Він контролює процес розробки в завданнях компіляції та упаковки для тестування, розгортання та публікації. Підтримувані мови включають Java (Kotlin, Groovy, Scala), C / C ++, JavaScript.

3.2. Програмні засоби для створення клієнтської частини

Клієнтська частина додатку реалізована як SPA (англ. Single Page Application) за допомогою React JS.

SPA – це додаток, що працює в браузері, якому не потрібно перезавантажувати сторінку під час використання. Приклади SPA: Facebook, Google Maps, Gmail, Twitter, Google Drive або навіть GitHub.

Головна перевага односторінкових додатків - це їх швидкість. Більшість потреб SPA (HTML + CSS + Scripts) завантажуються під час запуску програми, і їх не потрібно перезавантажувати під час використання. Єдине, що змінюється,

це дані, які передаються на сервер і з нього. Як результат, програма дуже чуйно реагує на запити користувача і не повинна весь час чекати на зв'язок клієнт-сервер.

React.js – це бібліотека JavaScript з відкритим кодом, яка використовується для побудови користувальницьких інтерфейсів спеціально для односторінкових додатків.

У React, замість використання звичайного JavaScript для створення шаблонів, він використовує JSX. JSX - це простий JavaScript, який дозволяє прописувати HTML і використовує синтаксис HTML-тегу для візуалізації підкомпонентів. Синтаксис HTML обробляється у виклики JavaScript React Framework. Однак справа тут не лише в зручності - використання JSX для оновлення DOM призводить до значного покращення продуктивності сайту та ефективності розробки.

React має віртуальний DOM, який є копією фактичного DOM і зберігається в пам'яті браузера у вигляді об'єкта javascript. Віртуальний DOM (VDOM) – це концепція програмування, де ідеальне, або «віртуальне», представлення інтерфейсу зберігається в пам'яті та синхронізується з «реальним» DOM. Цей підхід дозволяє декларативний API React: Ви повідомляєте React, у якому стані має знаходитись UI, і він забезпечує, щоб DOM відповідав цьому стану. Це абстрагує обробку атрибутів, обробку подій та оновлення DOM вручну.

Для збірки проекту використовується Webpack - це статичний обробник модулів для сучасних JavaScript додатків. Коли webpack обробляє додаток, він створює граф залежностей, який відображає кожен модуль, який потрібен проекту, і генерує один або кілька пакетів.

NGINX слугує сервером для зберігання статичних HTML/CSS/JS файлів. NGINX - це безкоштовний високопродуктивний HTTP-сервер із зворотним кодом та зворотний проксі-сервер, а також проксі-сервер IMAP / POP3. NGINX

відомий своєю високою продуктивністю, стабільністю, багатим набором функцій, простою конфігурацією та низьким споживанням ресурсів.

3.3. Контейнеризація додатків за допомогою Docker

Docker - це програмна платформа для створення додатків на основі контейнерів - невеликих та легких середовищ виконання, які спільно використовують ядро операційної системи, але працюють ізольовано одне від одного.

Контейнери надають усі переваги віртуальних машин, включаючи ізоляцію додатків, масштабованість та одноразовість. Але додатковий рівень абстракції (на рівні ОС) пропонує важливі додаткові переваги:

- Менша вага: На відміну від віртуальних машин, контейнери не несуть навантаження цілого екземпляра ОС - вони включають лише процеси ОС та залежності, необхідні для виконання коду;
- Більша ефективність використання ресурсів: за допомогою контейнерів можна запустити в кілька разів більше копій програми на тому ж обладнанні, аніж допомогою віртуальних машин. Це може зменшити хмарні витрати.
- Покращена продуктивність розробки: порівняно з віртуальними машинами, контейнери швидше та простіше розгортати, надавати та перезапускати. Це робить їх ідеальними для використання в конвеєрах безперервної інтеграції та безперервної доставки (CI / CD), а також краще підходить для команд розробників, які застосовують практики Agile та DevOps.

Кожен контейнер Docker починається з простого текстового файлу, що містить інструкції щодо побудови образу контейнера Docker. DockerFile автоматизує процес створення образу Docker. По суті, це список команд, які виконуватиме Docker Engine для складання образу контейнера.

Образ Docker містять виконуваний вихідний код програми, а також усі інструменти, бібліотеки та залежності, які код програми повинен запускати як контейнер.

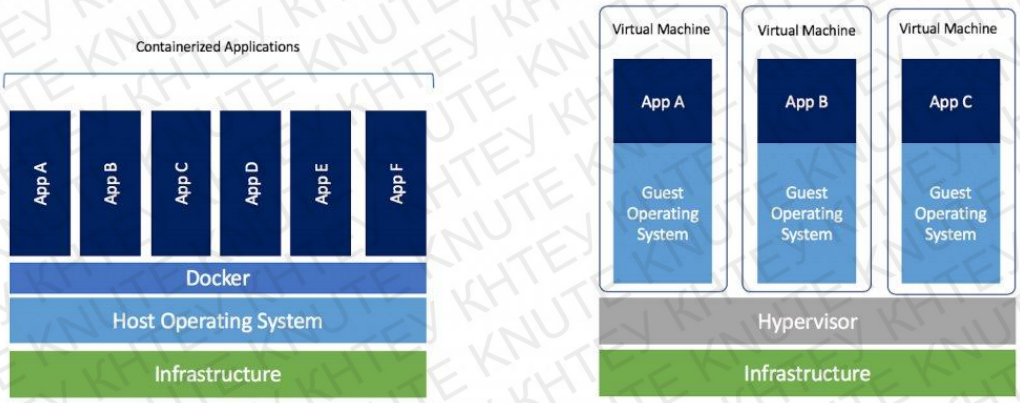

Рис. 3.1. Порівняння віртуальних машин з контейнерами Docker

3.4. Архітектура додатку

CinemaBooking – це додаток створений на основі мікросервісної архітектури, який надає користувачам можливість замовити квитки до кінотеатру онлайн.

На рисунку 3.2. зображена архітектура майбутнього додатку CinemaBooking.

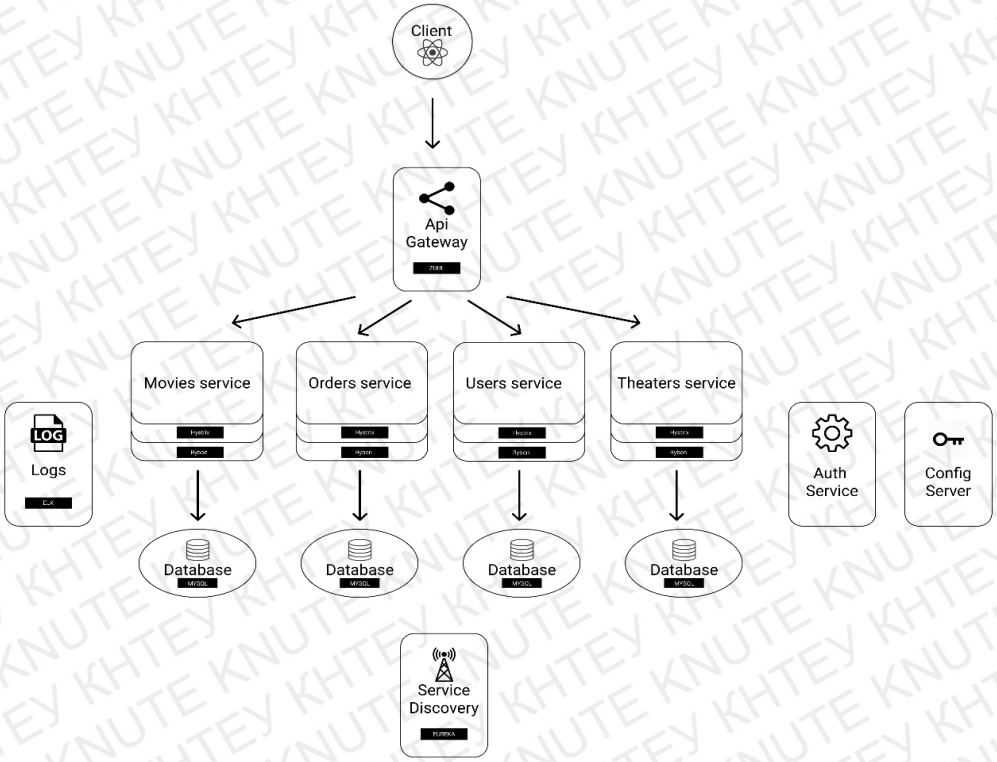

Рис. 3.2. Архітектура додатку CinemaBooking

Додаток складатиметься з 4 основних мікросервісів:

- Movies service: зберігає та надає основну інформацію про фільми;

- Users service: відповідає за зберігання інформації про користувачів, реєстрацію та оновлення даних;

- Orders service: відповідає за функціональність пов'язану з бронюваннями квитків на сеанси;

- Theaters service: містить інформацію про кінотеатри, зали, місця та сеанси.

Кожен мікросервіс є Spring Boot додатком та має свою базу даних яка відповідає домену, який покриває кожен конкретний мікросервіс. Для усіх мікросервісів використовується СКБД MySQL. Здебільшого, якщо один сервіс має зберігати інформацію про об'єкт з іншого домену, то в базі даних зберігається ідентифікатор як посилання на об'єкт в іншій базі даних і при необхідності можна дістати необхідну інформацію з іншого сервісу. Наприклад, об'єкт «Замовлення» з Orders Service зберігає лише ID користувача з Users service. Але іноді частина інформації може реплікуватися між сервісами, щоб уникати низки зайвих мережевих запитів. Наприклад, Theaters service частково дублює інформацію про фільми з Movies service, щоб при запиті на отримання прем'єр на цей тиждень зайвий раз не звертатися за певною інформацією до Movies service.

Клієнтом виступає SPA розроблене на React JS. Клієнт має лише відомості про API Gateway, і при необхідності дістати інформацію від мікросервісів звертається саме до нього. Той в свою чергу, знаючи усі IP адреси мікросервісів, перенаправляє запити до необхідних сервісів.

API Gateway – це інструмент управління API, який знаходиться між клієнтом та колекцією серверних сервісів. API Gateway діє як зворотний проксі-сервер, щоб приймати всі API запити, агрегувати різні служби, необхідні для їх виконання, і повертати відповідний результат.

Інформацію про мікросервіси та їх екземпляри надає сервіс виявлення служб Eureka, в якій кожен мікросервіс під час створення реєструється.

Окрім основних мікросервісів в нас також є допоміжні сервіси, такі як Authentication service (відповідає за аутентифікацію користувачів), Logging service (оснований на ELK, відповідає за агрегацію логів), Configuration service (містить загальну конфігурацію, а також конфігурацію необхідну для окремих сервісів).

3.5. Хмарне середовище Amazon Web Services

Amazon Web Services (AWS) - це один із найкращих варіантів розгортання додатків на базі мікросервісів завдяки різноманітності пакетів IaaS, PaaS, SaaS та SDK, пропонованих цією хмарною платформою.

AWS пропонує широкий спектр будівельних блоків для підтримки програмного забезпечення будь-якої складності та масштабу:

- Обчислювальна потужність: AWS EC2 Elastic Container Service та AWS Lambda Server Server;
- Зберігання: безпечне сховище (Amazon S3) та Amazon ElastiCache;
- Бази даних: Amazon RDB, Amazon Aurora, Amazon DynamoDB та десятки інших;
- Мережа: Amazon Service Discovery і AWS App Mesh, AWS Elastic Load Balancing, Amazon API Gateway і AWS Route 53 для DNS;
- Обмін повідомленнями: Amazon SQS для черги повідомлень та SNS для публікації та сповіщень;
- Моніторинг: AWS Cloudtrail для моніторингу API та Amazon CloudWatch для моніторингу інфраструктури;
- DevOps та CI / CD: Amazon Container Image Repository (Amazon ECR) та інші інструменти DevOps для включення робочих процесів CI / CD.

AWS дозволяє налаштувати необхідну архітектуру та вибрати будь-яку необхідну операційну систему, мову програмування, фреймворк, базу даних або

інші необхідні послуги. Це полегшує міграцію існуючих програм і дозволяє швидко створювати нові програми з нуля.

Основна архітектура мікросервісу на AWS виглядає так:

1. Екземпляри клієнтської частини працюють на AWS CloudFront CDN; статичний вміст зберігається на Amazon S3.
2. Вхідний трафік надходить на Amazon Automatic Load Balancer (ALB), який направляє його до кластера Kubernetes з контейнерами Docker, що запускають мікросервіси на Amazon ECS.
3. Дані кешуються ElastiCache і зберігаються в будь-якій базі даних, таких як Aurora, RDS або DynamoDB.

3.6. Висновки до розділу 3

В цьому розділі було описано програмні засоби для розробки, контейнеризації та розгортання мікросервісного додатка CinemaBooking.

Для реалізації серверної частини буде використано мову програмування Java разом з фреймворком Spring, а саме з його компонентами Spring Boot, Spring Data, Spring Security та Spring Cloud Netflix.

Клієнтська частина додатку реалізована як SPA (англ. Single Page Application) за допомогою React JS.

Усі компоненти додатку будуть контейнеризовані за допомогою програмної платформи Docker і розміщені в хмарному середовищі за допомогою сервісів Amazon Web Services.

РОЗДІЛ 4.
ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ

4.1. Програмна реалізація клієнтської частини

Клієнтська частина реалізована як SPA на React JS.

Основні сторінки сайту:

– Головна сторінка: має меню для переходу на інші частини сайту, можливість логіну чи реєстрації, слайдер з кіно прем'єрами цього тижня, селектор з вибором необхідного кінотеатру;

– Сторінка «Скоро в прокаті»: містить список фільмів для яких незабаром почнеться продаж квитків;

– Сторінка з інформацією про фільм: містить основну інформацію про фільм, таку як назва, жанр, трейлер, основні актори. Також на сторінці присутній селектор для вибору сеансу в обрану дату. Користувач може обрати день і час із запропонованих варіантів і перейти до придбання квитків;

– Сторінка з вибором місць: клієнту демонструється схема залу, типи місць, позначено, які місця зарезервовані, які вільні. Користувач може обрати собі місця і перейти до сплати;

– Сторінка оплати: клієнт вводить основні платіжні дані, після оплати отримує квитанцію на пошту.

– Сторінка профілю: сторінка містить список попередніх замовлень користувача.

Структура кодової бази зображена на рисунку 4.1.

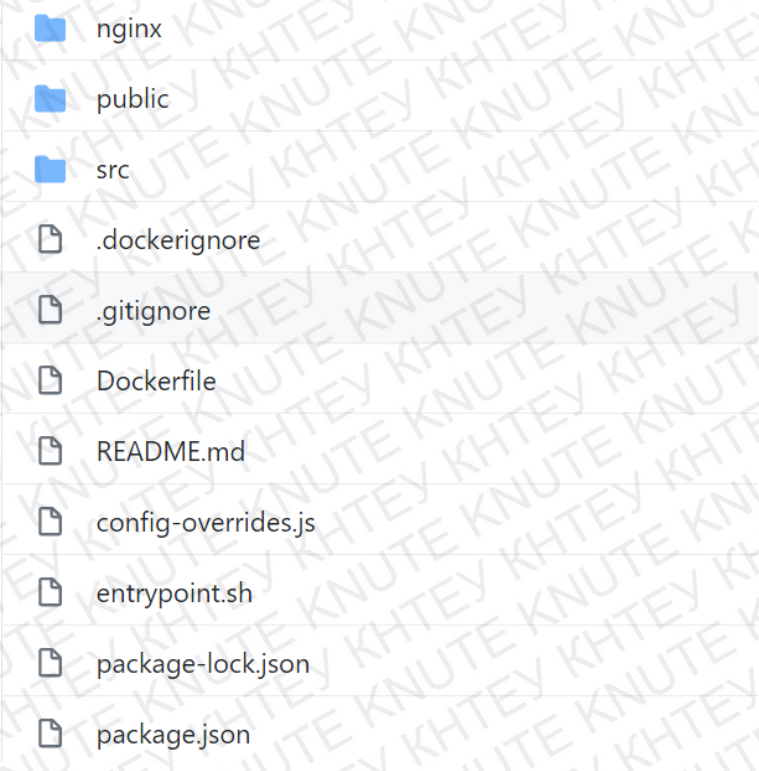

Рис. 4.1. Структура кодової бази клієнтської частини

Файл package.json містить основні залежності проекту, конфігурацію та скрипти для запуску режиму розробки, або ж генерації файлів, які будуть використовуватися для розгортання додатку.

Папка src містить вихідний код додатку. В ній містяться файли стилів, розмітки та Javascript файли з React компонентами. React додатки, зазвичай, розбивають на компоненти та контейнери. Контейнери переважно групують у собі набір компонентів та контролюють їх, а також зберігають стан програми.

Компоненти –це багаторазові блоки, які можуть бути використанні в різних частинах сайту для відображення певної інформації.

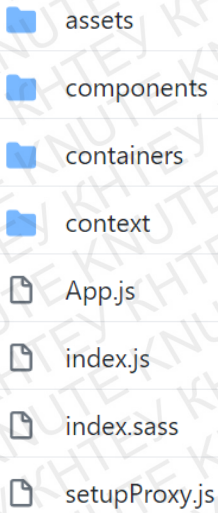

Рис. 4.2. Вміст папки src

App.js основний файл додатка. В ньому зберігаються імпорти для деяких бібліотек, файлів стилів. В ньому міститься основна структура проекту і це той компонент, який відповідає за те, яка саме сторінка буде відображена.

React Router використовується для обробки маршрутів у веб-програмі, використовуючи динамічну маршрутизацію. Динамічна маршрутизація відбувається, коли програма відображається на вашому комп'ютері, на відміну від старої архітектури маршрутизації, де маршрутизація обробляється у конфігурації поза запущеною програмою. Маршрутизатор React реалізує компонентний підхід до маршрутизації. Він надає різні компоненти маршрутизації відповідно до потреб програми та платформи.

Для контейнеризації клієнтської частини є файл Dockerfile. В ньому прописані інструкції для побудови і підготовки вихідних файлів, та їх встановлення на сервер NGINX. Вміст файлу:

```
FROM node:14.15.0-alpine as build
WORKDIR /app
ENV PATH /app/node_modules/.bin:$PATH
COPY package.json ./
COPY package-lock.json ./
RUN npm ci --silent
RUN npm install react-scripts@3.4.1 -g --silent
```

```
COPY . ./
RUN npm run build

FROM nginx:stable-alpine
COPY --from=build /app/build /usr/share/nginx/html
COPY nginx/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

4.2. Програмна реалізація серверної частини

Для реалізації серверної частини додатку CinemaBooking буде використано мову програмування Java разом з фреймворком Spring, а саме з його компонентами Spring Boot, Spring Data, Spring Security та Spring Cloud Netflix.

Додаток складається з 4 основних мікросервісів:

- Movies service;
- Users service;
- Orders service;
- Theaters service.

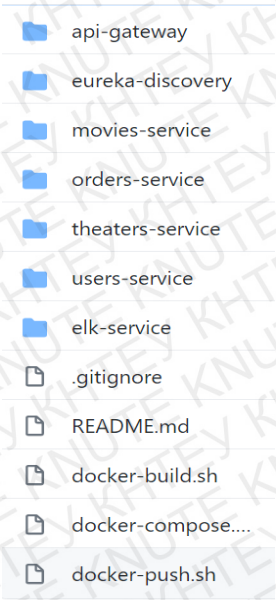

Рис. 4.3. Файлова структура проекту з мікросервісами

Кожен мікросервіс містить вихідні файли, конфігураційний файл Gradle для автоматизованої збірки проекту та файл Dockerfile для контейнеризації додатку.

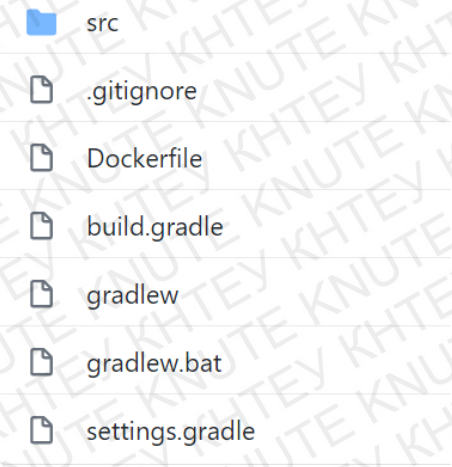

Рис. 4.4. Типова структура мікросервісного проекту

Кожен мікросервіс розділений на такі частини:

– Service layer: частина, яка відповідає за бізнес логіку;
– Data access layer: відповідає за доступ до бази даних;
– Controller layer: обробляє усі HTTP запити.

Сервіс eureka-discovery відповідає за реєстрацію усіх екземплярів мікросвервісів. Щоб мікросервіс був зареєстрований він має бути Eureka клієнтом.

```java
@SpringBootApplication
@EnableDiscoveryClient
public class UsersServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(UsersServiceApplication.class, args);
    }

    @Bean
    public ModelMapper modelMapper() {
        ModelMapper modelMapper = new ModelMapper();
        modelMapper.getConfiguration().setMatchingStrategy(STRICT);
        return modelMapper;
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

}
```

Рис. 4.5. Приклад Eureka клієнта

Api-gateway є точку входу до системи, яка вміє взаємодіяти із іншими мікросервісами використовуючи як REST. У якості конкретної імплементації обрано Netflix Zuul.

Під час роботи з мікросервісами важливо пам'ятати про можливість конфігурації змінних так, щоб не доводилось змінювати код сервісу. Саме для таких цілей було розроблено мікросервісний патерн «Configuration server». Цей сервер є сховищем конфігураційних змінних. Кожен сервіс залежить залежати від нього та звантажує свої 48 конфігурації. Окрім того, конфігураційний сервер надає можливість уникнути дуплікації конфігураційних змінних, якщо вони використаються у декількох сервісах. У якості конкретної реалізації патерну обрано Cloud Configuration Server. Імплементацією є розроблений config-service, що відповідає за керування конфігурацією та розповсюджує відповідні налаштування для кожного з компонентів системи.

Кожен мікросервіс має Dockerfile для подальшої контейнеризації додатку. Оскільки використано Spring Boot фреймворк, результатом компіляції та збірки проекту є артефакт з вмонтованим веб-сервером та усіма залежностями, які потрібні для незалежного запуску мікросервісу. Залишається написати спеціальний файл для декларації середовища запуску контейнеру з сервісом

(див. рис. 4.6). Він залишається актуальним для всіх сервісів системи без винятку.

```
FROM adoptopenjdk/openjdk11:alpine-jre
WORKDIR /opt/app
ARG JAR_FILE=build/libs/*.war
COPY ${JAR_FILE} theater-service.war
ENTRYPOINT ["java","-jar","theater-service.war"]
```

Рис. 4.6. Dockerfile для середовища запуску сервісу

4.3. Висновки до розділу 4

Отже, згідно з Технічним завданням та на основі спроектованого архітектури програмного продукту було написано саме програмний код з використанням об’єктно-орієнтованої мови програмування Java. Розробку програмного продукту було розподілено на етапи згідно з визначеними мікросервісами, їх функціями та зв’язками один з одним. Всі етапи розробки було виконано, програмний продукт протестовано.

ВИСНОВКИ ТА ПРОПОЗИЦІЇ

Згідно з завданням випускного кваліфікаційного проекту було реалізовано і спроектовано інформаційну систему для продажу квитків до кінотеатру на основі мікросервісної архітектури, мікросервіси контейнеризовано і розміщенно в хмарному середовищі.

У ході виконання дипломного проекту було проаналізовано складові та компоненти веб-інфраструктури, принципи взаємодії клієнт-сервер, визначено базові технології і служби за допомогою яких організовано роботу веб-додатків.

Було проведено аналіз і дослідження принципів побудови мікросервісної архітектури. Проведено порівняльну характеристику мікросервісного підходу із іншими шаблонами проектування, такими як: монолітна архітектура та сервісно-орієнтована архітектура.

Проведено аналіз базових підходів у розгортанні додатків, а саме із використанням апаратної віртуалізації й програмної контейнеризації, було проаналізовано основі принципи роботи та переваги і недоліки кожної із стратегій.

У якості програмного забезпечення для контейнеризації було обрано Docker, як сучасний і широко застосований продукт для керування контейнерами й образами.

На фінальному етапі було розроблено інформаційну систему для продажу квитків до кінотеатру на основі мікросервісної архітектури. Для реалізації серверної частини було використано мову програмування Java разом з фреймворком Spring, а саме з його компонентами Spring Boot, Spring Data, Spring Security та Spring Cloud Netflix.

Клієнтська частина додатку реалізована як SPA (англ. Single Page Application) за допомогою React JS.

Усі компоненти додатку будуть контейнеризовані за допомогою програмної платформи Docker і розміщені в хмарному середовищі за допомогою сервісів Amazon Web Services.

Отже, мета випускного кваліфікаційного проекту: теоретичне обґрунтування стратегій і напрямків розвитку мікросервісної архітектури і моделей систем сервісів для створення конкретних прикладних додатків у зв'язку з загальною тенденцією перенесення IT-рішень у хмарне середовище – досягнута. Завдання, що були поставлені для виконання мети проекту – виконано.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chris Richardson Microservices Patterns / Chris Richardson. – Shelter Island : Maning, 2019. – 490 с.
2. Sam Newman Building Microservices / Sam Newman. – Boston: O`Really, 2016. – 260 с.
3. Ian Dees Seven More Languages in Seven Weeks. Languages That Are Shaping the Future / Bruce Tate, Fred Daoud, Jack Moffitt, Ian Dees. — The Pragmatic Bookshelf, 2015. — 320 с.
4. Programming Java / Zigurd Mednieks, G. Blake Meike, Masumi Nakamura. – O`Reilly, 2011. - 736 с.
5. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans – Addison-Wesley, 2003 – p. 560
6. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.

Інтернет-ресурси

7. Monolithic vs. Microservices Architecture [Електронний ресурс] – Режим доступу: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59.
8. SOA (Service-Oriented Architecture) [Електронний ресурс] – Режим доступу: https://www.ibm.com/cloud/learn/soa.
9. The Scale Cube [Електронний ресурс] – Режим доступу: https://microservices.io/articles/scalecube.html.

10. Service Discovery in a Microservices Architecture [Електронний ресурс] – Режим доступу: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.
11. Spring Framework [Електронний ресурс] – Режим доступу: https://spring.io/projects/spring-framework/.
12. Guide to Implementing Microservices Architecture On AWS [Електронний ресурс] – Режим доступу: https://relevant.software/blog/microservices-on-aws/#Why_AWS.
13. Hystrix project on Github [Електронний ресурс] – Режим доступу: https://github.com/Netflix/Hystrix.
14. Introduction to Spring Cloud Netflix – Eureka [Електронний ресурс] – Режим доступу: https://www.baeldung.com/spring-cloud-netflix-eureka.
15. Service Discovery in a Microservices Architecture [Електронний ресурс] – Режим доступу: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.
16. Communication in a microservice architecture [Електронний ресурс] – Режим доступу: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

ТЕХНІЧНЕ ЗАВДАННЯ

1. Введення:

1.1. Призначення.

ПЗ призначене для компанії, що займається продажем квитків до кінотеатрів.

Мета створення.

Головною метою системи онлайн-бронювання квитків є надання альтернативного та зручного способу для покупця придбати квитки в кінотеатри в будь-який час і в будь-якому місці для збільшуючи прибуток.

Область дії.

Область дії програмного продукту – мережа інтернет.

1.2. Короткий огляд.

Інформаційна система для продажу квитків до кінотеатру.

2. Загальний опис:

2.1. Повне найменування інформаційної системи.

«Інформаційна система бронювання квитків до кінотеатру CinemaBooking».

2.2. Скорочене найменування інформаційної системи.

«CinemaBooking»,

2.3. Функції продукт (короткий опис).

При розробці інформаційної системи для придбання квитків до кінотеатру необхідно реалізувати наступні функції:

а) Перегляд інформації про фільми, сеанси, зали та місця в кінотеатрах.

б) Реєстрація користувачів та зберігання основної інформації про них.

в) Розрахунок вартості замовлення.

г) Зберігання інформації про замовлення користувачів.

2.4. Потенційні користувачі системи.

Потенційними користувачами системи є поціновувачі перегляду фільмів в кінотеатрах.

2.5. Характеристики користувача.

Потенційний користувач має мати доступ до мережі інтернет.

3. Вимоги:

3.1. Вимоги до інтерфейсу.

Інтерфейс програмного продукту повинен бути зручним та інтуїтивно зрозумілим.

Вимоги до розміщення об’єктів на екрані:

а) Об’єкти, що по своїй ролі можна віднести до основних, потрібно групувати

в центрі екрану;

б) Другорядні об’єкти – по периферії;

в) Кнопки групувати відповідно до їх функціонального призначення, не розташовувати кнопки (що відносяться до різних груп) поруч один з одним.

Вимоги до виведення інформації на дисплей:

Будь-яка взаємодія користувача з програмним забезпеченням повинна відбуватися в окремому вікні. Тобто, якщо користувачеві необхідно здійснити дію, що не зв’язана напряму з його поточною роботою – повинно відкриватися нове вікно.

Вимоги до мови взаємодії\спілкування з користувачем:

а) Відповідати потребам і задачам користувачів;

б) Відповідати призначенню і особливостям програмного продукту;

в) Бути зручною і легкою у використанні, ефективною в діяльності;

г) Відповідати нормам предметної області.

3.2. Функціональні вимоги.

3.2.1 Інформаційне середовище.

Інформація, з якою працюватимуть користувачі програмного продукту, зберігатиметься в базах даних (що також буде розроблятися) мікросервісів відповідно до домену.

База даних буде сформована на основі даних кожного домену. База даних повинна мати реляційну структуру.

Вхідними даними для роботи з програмним продуктом є дані описаної вище бази даних.

Вихідними даними слугуватимуть заповнена інформація про працівника та звіти.

3.2.2 Функції програмного забезпечення.

При розробці інформаційна система бронювання квитків до кінотеатру CinemaBooking необхідно реалізувати наступне:

Створити серверну та клієнтську частину. Серверна частина буде складатися з мікросервісів, які відповідають доменам бізнес-моделі. Окрім основних мікросервісів мають бути присутні служби, які відповідають за додаткові функції, наприклад, виявлення та реєстрація сервісів, API Gateway.

Основні функції інформаційної системи бронювання квитків до кінотеатру CinemaBooking;

1. Реєстрація - Якщо клієнт хоче забронювати квиток, тоді він / вона повинні бути зареєстровані, незареєстрований користувач не може забронювати квиток.

2. Вхід - Клієнт входить у систему, вводячи дійсний ідентифікатор користувача та пароль для бронювання квитка.

3. Перегляд інформації про фільми - Користувач повинен мати можливість переглянути список фільмів, що йдуть у вибраному кінотеатрі.

4. Перегляд сидіння - Клієнту показується 2D-зображення місць, з яких обрані бажані місця.

5. Оплата – Має бути налаштована тестова версія оплати для клієнта кредитною карткою.

6. Вихід - Після оплати або перегляду фільму клієнт вийде з системи.

7. Створити квиток - Після замовлення система може сформувати переносний файл документа (.pdf), а потім надіслати одну копію на електронну адресу замовника.

3.3. Нефункціональні вимоги (надійність, доступність, безпека та ін.).

3.3.1 Параметри, що характеризують ступінь відповідності програмного забезпечення призначенням.

Основним параметром, що характеризує ступінь відповідності програмного забезпечення призначенню є виконання функцій, що описані в п. 3.2.2.

3.3.2 Вимоги до надійності.

Основними показниками надійності програмного забезпечення є:

а) Безвідмовність програмного забезпечення;

б) Відновлюваність програмного забезпечення.

Необхідно проводити тестування функціональності програмного продукту після завершення написання програмного коду для кожного функціонального блоку. Також, при завершенні написання програмного коду потрібно провести тестування додатку, врахувавши всі можливі варіанти використання.

Позитивним результатом оцінки та контролю показників надійності буде вважатися відсутність помилок та відмов при тестуванні.

ПРОГРАМА ТА МЕТОДИКА ТЕСТУВАННЯ

Необхідно виконати тестування інформаційну систему для продажу квитків до кінотеатру. Для успішного проведення тестування, слід виконати наступні тест-кейси:

1. Реєстрація на сайті.
2. Робота з карткою співробітника.
3. Робота зі звітами.

Кроки для виконання тест-кейсів та очікуваний результат описані в

Таблиця Т. 1

ПОРЯДОК ТЕСТУВАННЯ

Тест-кейс	Кроки для виконання	Очікуваний результат
Реєстрація на сайті	Користувач потрапляє на головну сторінку.	Буде відкрита головна сторінка у вікні браузера.
	Натискання кнопки «Login».	Буде відкритий модальне вікно «Аутентифікації».
	Натискання кнопки «Register».	Буде відкрита форма «Реєстрації».
	Користувач заповнює усі дані та натискає кнопку «Register»	Після підтвердження відправки форми користувач побачить повідомлення про успішну реєстрацію.
Аутентифікація користувача та бронювання місць	Користувач потрапляє на головну сторінку.	Буде відкрита головна сторінка у вікні браузера.
	Натискання кнопки «Login».	Буде відкритий модальне вікно «Аутентифікації».
	Користувач заповнює усі дані та натискає кнопку «Login»	Після підтвердження відправки форми користувач побачить повідомлення про

Тест-кейс	Кроки для виконання	Очікуваний результат
		успішну аутентифікацію.
	Вибір фільму.	Користувача буде перенаправлено на сторінку з інформацією про фільм.
	Вибір дати.	Після вибору дати, користувачу буде показано список сеансів на обрану дату.
	Вибір сеансу.	Після вибору сеансу користувача буде перенаправлено на сторінку з вибором місць.
	Перехід на сторінку з вибором місць.	На цій сторінці користувач має побачити схему розсаджування в залі кінотеатру, вільні та зарезервовані місця.
	Вибір місця	Після вибору місця користувача буде перенаправлено на сторінку оплати.

Якщо в результати виконання тест-кейсів будуть аналогічними очікуваному результату, тестування можна вважати успішним.

КЕРІВНИЦТВО КОРИСТУВАЧА

В інформаційній системі для продажу квитків до кінотеатру користувачеві можна використовувати наступні функції:

1. Реєстрація - Якщо клієнт хоче забронювати квиток, тоді він / вона повинні бути зареєстровані, незареєстрований користувач не може забронювати квиток.

2. Вхід - Клієнт входить у систему, вводячи дійсний ідентифікатор користувача та пароль для бронювання квитка.

3. Перегляд інформації про фільми - Користувач повинен мати можливість переглянути список фільмів, що йдуть у вибраному кінотеатрі.

4. Перегляд сидіння - Клієнту показується 2D-зображення місць, з яких обрані бажані місця.

5. Оплата – Має бути налаштована тестова версія оплати для клієнта кредитною карткою.

6. Вихід - Після оплати або перегляду фільму клієнт вийде з системи.

7. Створити квиток - Після замовлення система може сформувати переносний файл документа (.pdf), а потім надіслати одну копію на електронну адресу замовника.

Після відкриття сайту в браузері користувач може використовувати меню для загальної навігації по сторінках сайту, використовувати кнопки для відкриття модальних вікон чи здійснення певних операцій.

Щоб мати можливість обирати сеанс, місця та створювати замовлення користуває має бути обов’язково авторизованний.

					КНТЕУ 121– 06-14.МР		
Зм.	*Аркуш*	*№ докум.*	*Підпис*	*Дата*	*Розробка інформаційної системи на основі мікросерверної архітектури з використанням Cloud-середовища*	*Стадія*	*Аркуш*	*Аркушів*
Зав. каф.		Криворучко О.В.		24.01.20		КК	56	47
Керівник		Жирова Т.О.		24.01.20				
Гарант		Криворучко О.В		24.01.20	*Керівництво користувача*	Факультет інформаційних технологій, 2м курс, 6 група		
Розробив		Мосійчук Є.В.		24.01.20				

ДОДАТКИ

Додаток А

«Лістинг програмного коду для класу MoviesServiceImpl»

```
@Service
public class MoviesServiceImpl implements MoviesService {
    private MoviesRepository moviesRepository;

    @Autowired
    public MoviesServiceImpl(MoviesRepository moviesRepository) {
        this.moviesRepository = moviesRepository;
    }

    @Override
    public List<MovieData> findMovies(boolean active) {
        return ObjectMapperUtils.mapAll(moviesRepository.findAllByActive(active),
MovieData.class);
    }

    @Override
    public List<MovieData> findMoviesByTheaterId(boolean active, long theaterId) {
        return
ObjectMapperUtils.mapAll(moviesRepository.findAllByActiveAndShowingsHallTheat
erId(active, theaterId), MovieData.class);
    }
}
```

Додаток Б

«Лістинг програмного коду файлу конфігурації NGNIX»

```
server {

 listen 80;

 location / {
  root   /usr/share/nginx/html;
  index  index.html index.htm;
  try_files $uri $uri/ /index.html;
 }

 location /api/ {
    rewrite ^/api/(.*) /$1 break;
    proxy_pass ${API_GATEWAY};
 }

 error_page   500 502 503 504  /50x.html;

 location = /50x.html {
  root   /usr/share/nginx/html;
 }

}
```

Додаток В

«Лістинг програмного коду Dockerfile»

```
FROM adoptopenjdk/openjdk11:alpine-jre
WORKDIR /opt/app
ARG JAR_FILE=build/libs/*.war
COPY ${JAR_FILE} theater-service.war
ENTRYPOINT ["java","-jar","theater-service.war"]
```

«Лістинг програмного коду для класу HallsServiceImpl»

```
@Service
public class HallsServiceImpl implements HallsService {
   private HallsRepository hallsRepository;

   @Autowired
   public HallsServiceImpl(HallsRepository hallsRepository) {
      this.hallsRepository = hallsRepository;
   }

   @Override
   public List<HallData> findAllBy(long movieId, LocalDate date, long theaterId) {
      List<HallData> hallData = ObjectMapperUtils
            .mapAll(hallsRepository.findAllByTheaterIdAndShowingsDate(theaterId,
date),
                  HallData.class);

      hallData.forEach(hall -> hall.getShowings().forEach(showingData ->
showingData.setHall(null)));
      return hallData;
   }
}
```