МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Вплив контейнеризації та оркестрації на веб-розробку**

Виконав: студент 2 курсу, групи 8.1213-іпз-2
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
Д. С. Воробець
(ініціали та прізвище)
Керівник доцент, к.т.н., доцент В. І. Заяц
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ Дискус
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____другий (магістерський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т.В. Критська_
“_28_” ___лютого__ 2024_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Воробцю Дмитру Сергійовичу
(прізвище, ім’я, по батькові)

1.Тема роботи __Вплив контейнеризації та оркестрації на веб-розробку

керівник роботи ___Заяц Валерій Іванович, доцент, к.т.н._____
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 28.02.2024р. № 361-е

2. Строк подання студентом кваліфікаційної роботи_____30.11.2024_____
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми збору та аналізу даних з веб-сторінок;
- створення програмного продукту та його опис;
- дослідження поставленої проблеми та розробка висновків.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
14 слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 02.09.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	11.09.24	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	12.09.24	виконано
3	Аналіз існуючих методів рішення	13.09-14.09.24	виконано
4	Дослідження засобів контейнеризації та оркестрації	15.09-16.09.24	виконано
5	Узгодження подальших дій з науковим керівником	18.09.24	виконано
6	Проектування архітектури веб додатку	19.09-30.09.24	виконано
7	Програмна реалізація веб додатку	01.10-18.10.24	виконано
8	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	19.10.24	виконано
9	Перевірка роботоздатності веб додатку	20.10-21.10.24	виконано
10	Оформлення звіту	22.10-10.11.24	виконано
11	Оформлення презентації. Отримання рецензій від опонентів.	11.11-12.11.24	виконано

Студент _____ Воробець Д.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Заяц В.І.
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ Скрипник І.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок — 78

Рисунків — 11

Джерел — 30

Воробець Д.С. Вплив контейнеризації та оркестрації на веб-розробку: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Заяц. Запоріжжя : ЗНУ, 2024. 78 с.

Мета полягає у вивченні та розумінні того, як використання контейнеризації, такої як Docker, та системи оркестрації Kubernetes, може покращити ефективність, простоту розгортання, та управління веб-проектами. Робота також включає в себе дослідження мікросервісної розробки.

У процесі дослідження було розглянуто основні концепції контейнеризації та оркестрації, їхні переваги та виклики. Робота була зосереджена на визначенні оптимальних сценаріїв використання цих технологій в контексті веб-розробки та визначенні впливу на процес розробки, впровадження та масштабування веб-проектів. Також був розроблен веб додаток, використовуючи NodeJS, на основі мікросервісної архітектури, що конкретно доменструє ефективність контейнеризації та оркестрації.

Ключові слова: контейнеризація, оркестрація, Docker, Kubernetes, мікросервіси, Node JS.

ABSTRACT

Pages — 78

Drawings — 11

Sources — 30

Vorobets D.S. Influence of containerization and orchestration on web development: master's thesis in specialty 121 "Software Engineering" / Science. manager V.I. Zayats. Zaporizhzhia: ZNU, 2024. 78 p.

The goal is to study and understand how the use of containerization, such as Docker, and the Kubernetes orchestration system can improve the efficiency, ease of deployment, and management of web projects. The work also includes a study of microservice development.

In the course of the research, the main concepts of containerization and orchestration, their advantages and challenges were considered. The work was focused on determining the optimal scenarios for using these technologies in the context of web development and determining the impact on the process of developing, implementing and scaling web projects. Also, a web application was developed using NodeJS, based on a microservice architecture, which specifically demonstrates the effectiveness of containerization and orchestration.

Keywords*:* containerization, orchestration, Docker, Kubernetes, microservices, Node JS.

ЗМІСТ

ВСТУП 8

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 14

1.1 Огляд літературних джерел 14

1.2 Аналіз інструментів управління контейнерами 16

1.2.1 Podman 16

1.2.2 Containerd 18

1.2.3 Nomad 20

1.3 Методи оптимізації продуктивності контейнеризованих додатків 22

1.4 Автоматизація процесів у контейнеризованих середовищах 23

1.5 Постановка завдання 24

1.6 Висновки до розділу 1 25

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ 26

2.1 Аналіз сучасних технологій для розробки веб-додатків з використанням контейнеризації та оркестрації 26

2.2 Використання мікросервісної архітектури з контейнеризацією та оркестрацією 29

2.3 Виклики та особливості використання мікросервісів у контейнеризованому середовищі 31

2.4 Стратегії масштабування контейнеризованих мікросервісів 33

2.5 Забезпечення надійності та відмовостійкості контейнеризованих мікросервісів 35

2.6 Висновки до розділу 2 36

3 РОЗРОБКА ВЕБ-ДОДАТКУ З ВИКОРИСТАННЯМ СУЧАСНИХ ТЕХНОЛОГІЙ КОНТЕЙНЕРИЗАЦІЇ ТА ОРКЕСТРАЦІЇ 38
3.1 Опис предметної області 38
3.2 Архітектура додатку “Real Estate” 39
3.2.1 Frontend частина додатку “Real Estate” 40
3.2.2 Backend частина додатку “Real Estate” 44
3.2.3 DB частина додатку “Real Estate” 51
3.3 Функціонал сайту “Real Estate” 54
3.4 Інтеграція Docker та Kubernetes у процес розробки та розгортання “Real Estate” 56
3.5 Висновки до розділу 3 58
4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ ВПЛИВУ КОНТЕЙНЕРИЗАЦІЇ ТА ОРКЕСТРАЦІЇ НА РОБОТУ СИСТЕМИ "REAL ESTATE" 60
4.1 Аналіз продуктивності системи "Real Estate" у контейнерному середовищі 60
4.2 Дослідження масштабованості системи за допомогою Kubernetes 65
4.3 Оцінка надійності та відмовостійкості системи 69
4.4 Дослідження простоти управління середовищем розробки 71
4.4.1 Переваги використання Docker для локального середовища 72
4.4.2 Простіше управління завдяки Kubernetes 72
4.4.3 Масштабованість процесу розробки 73
4.5 Висновки до розділу 4 74
ВИСНОВКИ 75
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 76

ВСТУП

Актуальність теми

З розвитком сучасних технологій інформаційних систем і веб-розробки, використання контейнеризації та оркестрації стає необхідним етапом у розробці та розгортанні веб-додатків. Контейнеризація, представлена інструментами, такими як Docker, та системи оркестрації, такі як Kubernetes, надають можливість ефективно керувати ресурсами, забезпечувати масштабованість та високу доступність додатків [1], що стає важливим для успішного функціонування веб-систем у сучасному середовищі.

Актуальність теми "Вплив контейнеризації та оркестрації на веб-розробку" полягає в необхідності розуміння та використання сучасних підходів у розробці веб-додатків для підвищення їхньої ефективності, масштабованості та стійкості [2]. Відповідно до досліджень, використання контейнеризації та оркестрації позитивно впливає на процес розробки, забезпечуючи швидке розгортання та керування веб-додатками, а також зменшуючи час та витрати на їхнє обслуговування.

Для веб-розробників, які працюють в умовах обмеженого бюджету та ресурсів, використання контейнеризації та оркестрації може виявитися важливим аспектом удосконалення та спрощення процесів розробки веб-додатків. Однак існують проблеми, які виникають у веб-розробників при впровадженні цих технологій:

1. Високі витрати на розробку: Розробка веб-додатків, оптимізованих для контейнеризації та оркестрації, може вимагати значних фінансових та технічних витрат. Для веб-розробників, які мають обмежений бюджет та ресурси, це може виявитися викликом.

2. Відсутність досвіду: Впровадження контейнеризації та оркестрації вимагає значного рівня досвіду у веб-розробці та розуміння відповідних концепцій. Для тих, хто тільки починає свій шлях у веб-розробці, це може стати викликом.

3. Відсутність знань про архітектуру: Розробка веб-додатків, що використовують контейнеризацію та оркестрацію, вимагає розуміння архітектурних принципів та їхнього впливу на веб-розробку. Для тих, хто не має досвіду у цій області, це може бути складним завданням.

Переваги використання контейнеризації та оркестрації також включають:

1. Зменшення витрат: Використання контейнерів дозволяє ефективніше використовувати ресурси сервера, що призводить до зменшення витрат на обладнання та оплату хостингу.

2. Підвищена гнучкість: Контейнеризація дозволяє розробникам швидко розгортати, масштабувати та переміщати веб-додатки між різними середовищами без втрати продуктивності.

3. Спрощене управління: Системи оркестрації, такі як Kubernetes, забезпечують автоматизоване керування контейнеризованими додатками, що полегшує їх розгортання та керування [16].

4. Підвищена надійність та стабільність: Контейнери ізолюють додатки один від одного, що дозволяє уникнути конфліктів між ними та забезпечує стабільність роботи системи.

5. Швидке відновлення: Завдяки можливості швидко переключатися між версіями контейнерів, розробники можуть швидко відновлювати працездатність веб-додатків у випадку виникнення проблем.

Впровадження контейнеризації та оркестрації у веб-розробці допомогає вирішити багато проблем, з якими стикаються розробники, та підвищити ефективність процесу розробки та управління веб-проектами. Зазначеною причиною актуальності теми для веб-розробників є те, що деякі інструменти, такі як Docker, Kubernetes та інші, можуть виявитися витісненими для навчання та використання для початківців [3]. Створення власного мікросервісного рішення допомогає вирішити цю проблему, дозволяючи розробникам знизити рівень складності та оптимально використовувати свій час та ресурси для розробки різноманітних веб-проектів [4]. В цілому,

вивчення впливу контейнеризації та оркестрації на веб-розробку визнається як актуальна проблема для розробників, які прагнуть створювати якісні та інноваційні веб-додатки, оптимізуючи при цьому витрати та забезпечуючи більший контроль над розробкою.

Мета дослідження

Метою цього дослідження є ретельне вивчення впливу контейнеризації та оркестрації на процес веб-розробки, зокрема в контексті використання мікросервісної архітектури.

Завдання дослідження

Аналіз існуючих рішень в галузі контейнеризації та оркестрації. Детальний розгляд впливу застосування контейнеризації та оркестрації на процеси веб-розробки. Розробка мікросервісного веб-додатка. Оцінка продуктивності та масштабованості розробленого веб-додатка.

Об’єкт дослідження

Об'єктом уваги в цьому дослідженні є вплив контейнеризації та оркестрації на процес веб-розробки.

Предмет дослідження

Предметом дослідження є вивчення взаємодії контейнеризації та оркестрації з процесами веб-розробки. Основний акцент робиться на аналізі технічних аспектів контейнеризації, зокрема Docker, та системи оркестрації Kubernetes, з урахуванням їх впливу на розгортання та управління веб-проектами.

Методи дослідження

Проведення детального огляду науково-методичної літератури, академічних публікацій та технічної документації є важливим етапом дослідження. Вивчення сучасних тенденцій у розробці веб-додатків, зокрема мікросервісної архітектури, що дозволяє отримати глибше розуміння сучасних підходів та стратегій у веб-розробці. Аналіз технічної документації контейнеризаційних технологій, зокрема Docker, та системи оркестрації Kubernetes для визначення їхніх можливостей та обмежень.

Практичне значення одержаних результатів

Отримані результати дослідження мають практичне значення для сучасних веб-розробників та команд, що працюють над розгортанням веб-проектів. Розробка ефективних стратегій використання контейнеризації, зокрема Docker, та системи оркестрації Kubernetes сприяє значному поліпшенню ефективності, простоти розгортання та управління веб-додатками. Розроблені рекомендації та методології можуть стати основою для стандартизації процесів розробки веб-додатків і допомогти підвищити загальну якість та ефективність роботи веб-розробників.

Наукова новизна одержаних результатів

Наукова новизна отриманих результатів дослідження виявляється у впровадженні новаторського підходу до оптимізації та підвищення продуктивності веб-розробки. У роботі використовано інноваційні методи контейнеризації та оркестрації для оптимального управління веб-проектами, що дозволяє підняти рівень їхньої швидкості та ефективності. Використання цих інструментів відображає сучасний підхід до вирішення завдань веб-розробки та робить акцент на важливості використання контейнеризації та оркестрації для досягнення оптимальної ефективності.

Апробація одержаних результатів

Одержані результати були представлені на науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2024», а також на науково-технічній конференції, організованій Інженерним навчально-науковим інститутом Запорізького національного університету, де їх було обговорено з науковою спільнотою.

Глосарій

CQRS (Command Query Responsibility Segregation) — це шаблон проєктування, який розділяє операції читання і запису у системі, щоб підвищити продуктивність і масштабованість.

Підхід Domain-Driven Design (DDD) — це методологія розробки, яка орієнтована на створення програмного забезпечення відповідно до бізнес-логіки домену.

Оркестрація — це автоматизоване управління контейнерами, яке забезпечує їх масштабування, оновлення та моніторинг.

Pod (Под) — це найменша одиниця обчислення в Kubernetes, яка являє собою один або кілька контейнерів, що працюють у спільному ізольованому середовищі.

Node (Нода) — це фізичний сервер або віртуальна машина в кластері Kubernetes, яка виконує поди.

Blue-green розгортання — це стратегія розгортання, яка передбачає підтримку двох середовищ blue і green, де одне працює, а інше оновлюється, щоб безпечно впроваджувати зміни без простою. ризиком для користувачів і максимальною доступністю системи.

Minikube — це інструмент, який дозволяє локально запускати кластер Kubernetes.

JWT (JSON Web Token) — це стандартний формат для передачі даних між клієнтом і сервером у вигляді токенів, який використовується для аутентифікації та авторизації користувачів.

K6 — це інструмент для навантажувального тестування, який дозволяє моделювати високе навантаження на систему.

Брокер — це користувач системи, який відповідає за управління об'єктами нерухомості та укладання угод.

Secret (Секрет) — це ресурс Kubernetes для безпечного зберігання конфіденційних даних, таких як паролі, токени доступу чи ключі.

HPA (Horizontal Pod Autoscaling) — це механізм в Kubernetes, який автоматично збільшує або зменшує кількість подів у розгортанні, що забезпечуючи оптимальне використання ресурсів.

PV (Persistent Volumes) — це абстракція в Kubernetes для зберігання даних, які зберігаються поза життєвим циклом подів

CRI (Container Runtime Interface) — це інтерфейс, який дозволяє Kubernetes взаємодіяти з різними контейнерними рушіями для запуску контейнерів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд літературних джерел

Аналіз предметної області починається з огляду літературних джерел, спрямований на розуміння впливу контейнеризації та оркестрації на веб-розробку. У світі веб-розробки, де необхідно поєднувати ефективність та масштабованість, використання контейнеризації та оркестрації стає надзвичайно актуальним. Метою огляду літературних джерел є розширення знань у цій темі та виявлення сучасних підходів у використанні контейнерів та оркестрації для розробки веб-застосунків.

У методології огляду літературних джерел використовуються різноманітні джерела, такі як наукові статті, книги та онлайн-ресурси з актуальною інформацією про контейнеризацію та оркестрацію. Відбір літературних джерел проводиться з урахуванням їхньої релевантності, авторитетності та актуальності. Під час аналізу особлива увага приділяється методам впровадження контейнерів та оркестрації у веб-розробці та їх впливу на процес створення веб-застосунків. Основні теми, що виокремлюються під час аналізу літературних джерел, включають в себе переваги використання контейнерів для розгортання веб-застосунків, методи оркестрації контейнерів у хмарних середовищах та кращі практики використання мікросервісної архітектури з використанням оркестрації. Важливість проведеного дослідження підкреслюється необхідністю вдосконалення процесу веб-розробки за допомогою сучасних технологій контейнеризації та оркестрації. Огляд літературних джерел дасть можливість глибше розібратися у цій темі та впровадити нові підходи у веб-розробці для досягнення кращих результатів.

Під час аналізу предметної області з урахуванням теми "Вплив контейнеризації та оркестрації на веб-розробку" можна виділити як проблеми, так і переваги використання цих технологій. Впровадження контейнеризації та оркестрації у веб-розробці відкриває багато нових можливостей, але

водночас приносить чимало викликів. Однією з основних проблем є складність конфігурації. Налаштування контейнерів та оркестраторів вимагає глибокого розуміння технологій, і для початківців це може стати справжнім випробуванням. Додатково, витрати на підтримку інфраструктури теж можуть стати бар'єром. Для невеликих команд чи проєктів з обмеженим бюджетом інвестиції в обладнання та програмне забезпечення можуть виявитися непосильними. І хоча контейнери часто рекламують як безпечне рішення, вони можуть створювати нові вразливі місця, що вимагають ретельного моніторингу та впровадження додаткових заходів безпеки. До того ж, інколи контейнери й оркестратори не спрощують інфраструктуру, особливо якщо проєкт складний і містить численні мікросервіси.

Проте, переваги контейнеризації беззаперечні. Насамперед, це портативність. Контейнери забезпечують стабільне середовище, завдяки чому застосунки можна легко переносити між різними системами. Ще одна суттєва перевага — швидкість [5]. Оркестратори дозволяють автоматизувати розгортання і масштабування, що економить час і ресурси. Розробникам також зручно працювати в однаковому середовищі як на локальних машинах, так і на віддалених серверах, що значно спрощує тестування та налагодження. А можливість відокремлення мікросервісів у різні контейнери додає модульності системі, що полегшує її розвиток і масштабування [6]. Ці проблеми та переваги важливо розглянути при розробці та впровадженні контейнеризації та оркестрації у веб-розробці.

Ще одним важливим аспектом є постійний розвиток екосистеми контейнеризації [7]. Інструменти регулярно оновлюються та вдосконалюються, пропонуючи нові функції, які полегшують життя розробникам і адміністраторам. Спільнота також активно розвивається, що дає змогу знаходити відповіді на складні питання і запозичувати успішні практики в інших. Однак перед тим як впроваджувати контейнеризацію, важливо провести ретельний аналіз потреб проєкту. У деяких випадках, особливо для невеликих застосунків або стартапів, використання таких технологій може

бути надмірним [8]. Тут доречніше почати з простіших рішень, а вже після зростання навантаження чи ускладнення архітектури перейти до контейнерів.

1.2 Аналіз інструментів управління контейнерами

Питання ефективного управління контейнерами стало особливо важливим у зв'язку з популярністю мікросервісної архітектури та контейнеризації у сучасній веб-розробці. Системи управління контейнерами забезпечують можливість легкої масштабованості, високої доступності та автоматизації процесів розгортання додатків [13]. Для цього існують кілька підходів та інструментів, які допомагають оптимізувати управління контейнерами.

1.2.1 Podman

Podman — це інструмент для управління контейнерами, що надає альтернативу Docker, розроблений з урахуванням безпеки та простоти використання. Порівняно з Docker, Podman пропонує більш безпечний підхід до управління контейнерами, оскільки він не потребує привілеїв користувача root для запуску контейнерів.

Podman має схожий функціонал з Docker, включаючи можливість створення, запуску, зупинки та видалення контейнерів. Він також підтримує роботу з образами, мережами та томами. Однак, відмінністю є те, що Podman використовує стандартні інструменти управління процесами Linux, що дозволяє запускати контейнери у власному користувацькому просторі імен, забезпечуючи більшу безпеку. Рисунок 1 — Графічний інтерфейс Podman.

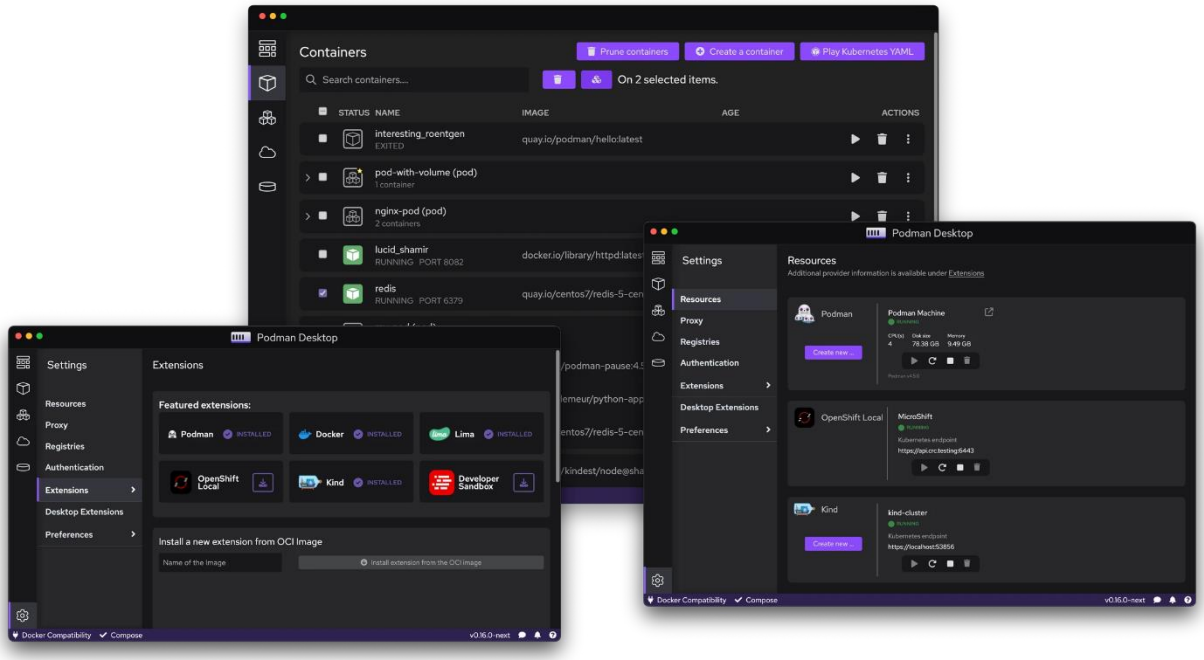


Рисунок 1 — Графічний інтерфейс Podman

Порівнюючи з Docker, Podman відзначається важливими перевагами, які забезпечують безпеку та простоту використання. У своїй суті, Podman надає користувачеві альтернативний засіб для управління контейнерами, що відрізняється від інших інструментів своїм підходом до безпеки та структурою. Одна з ключових переваг полягає у відсутності необхідності у привілеях користувача root для запуску контейнерів, що робить взаємодію з Podman більш безпечною і забезпечує відокремленість від основної системи.

Відсутність фонового демона в Podman сприяє збільшенню безпеки та зменшенню навантаження на систему. Використання стандартних інструментів управління процесами Linux дозволяє Podman працювати у власному користувацькому просторі імен, що ускладнює можливість несанкціонованого доступу до системних ресурсів та забезпечує більш високий рівень безпеки [15]. Podman визначається своєю здатністю забезпечувати безпеку та ефективність в управлінні контейнерами без складнощів і навантаження, пов'язаних з іншими рішеннями. Ці переваги створюють сприятливі умови для розробки та експлуатації контейнеризованих середовищ, забезпечуючи зручність та надійність в управлінні контейнерами

на різних рівнях складності проектів. Podman підтримує сумісність з Docker CLI через інтеграцію з podman-docker. Це означає, що існуючі скрипти та інструменти, створені для Docker, можуть працювати з Podman без значних змін. Це спрощує перехід розробників до використання Podman у їхніх проектах. Крім того, Podman забезпечує підтримку створення pod'ів — груп контейнерів, що ділять між собою мережевий стек та інші ресурси. Цей підхід полегшує симуляцію середовищ, подібних до Kubernetes, дозволяючи розробникам працювати з подібною логікою управління контейнерами на локальному рівні.

1.2.2 Containerd

Containerd — це високопродуктивний менеджер контейнерів, який розробляється як частина проекту OCI (Open Container Initiative). Він призначений для керування життєвим циклом контейнерів на рівні з підтримкою базових функцій, таких як запуск, зупинка та створення контейнерів, а також управління їхніми образами.

Однією з ключових переваг Containerd є його зосередженість на основних функціях контейнерів, що робить його ефективним та надійним рішенням для роботи з контейнерами на високому рівні. Це дозволяє розробникам та адміністраторам концентруватися на вирішенні конкретних завдань без зайвого навантаження, що призводить до збільшення ефективності та простоти управління контейнерами.

Більше того, Containerd підтримує стандартні протоколи та інтерфейси, такі як CRI, що дозволяє інтегрувати його з іншими інструментами та платформами, такими як Kubernetes. Це робить Containerd гнучким та легко налаштовуваним рішенням для використання у різних середовищах та сценаріях вирішення завдань. Рисунок 2 — Логотип Containerd.



Рисунок 2 — Логотип Containerd

Containerd розкриває перед користувачем широкий спектр функціональних можливостей, спрямованих на ефективне керування контейнерами. Він виступає у ролі надійного інструменту для запуску, зупинки та створення контейнерів, управління образами, мережами та зберіганням контейнерних даних. У багатьох сценаріях використання оркестрації контейнерів, він стає незамінним завдяки своїм властивостям та інтерфейсам взаємодії з іншими системами. Одним з ключових критеріїв оцінки роботи Containerd є його висока продуктивність, в якій використовується оптимізація ресурсів та відсутність накладних витрат. Цей критерій вказує на низький рівень затримок під час реакції рушію на користувацькі запити та завдання, що виконуються оперативно на основі різних умов виконання контейнерів.

Containerd славиться своєю надійністю та стабільністю [12], завдяки широкому спектру тестувань якості та активному співтовариству розробників. Це забезпечує безперебійну роботу контейнерів у будь-яких умовах та надає користувачам впевненість у надійності свого інструменту. Containerd проявляє високу гнучкість, що полягає в його здатності легко інтегруватися з різними системами та іншими компонентами інфраструктури. Він підтримує стандартні протоколи та інтерфейси, що робить його універсальним інструментом для створення різноманітних рішень для вирішення різноманітних завдань та вимог.

Порівнюючи Docker з Containerd, останній може бути більш гнучким і універсальним. Containerd надає менше функціональності з коробки, але широкі можливості розширюваності через різноманітні типи інтерфейсів та

API. Containerd дає користувачам більше свободи у налаштуванні та інтеграції з іншими системами та інструментами. Containerd широко застосовується як основний фреймворк для розвитку різних рішень у контейнерізації, зокрема платформ оркестрації, наприклад, Kubernetes. Це спрощує управління контейнерними робочими навантаженнями в масштабі та робить Containerd важливим компонентом сучасної хмарної екосистеми.

1.2.3 Nomad

Nomad — це програмне забезпечення для керування контейнерами від компанію HashiCorp. Його основна мета — автоматизувати та координувати роботу додатків, які працюють у контейнерах на різних серверах. Nomad спрощуэ швидке розгортання та масштабування додатків, керування ресурсами та розподіл завдань між вузлами у кластері.

Одній з ключових переваг Nomad є його простота в управлінні ресурсами та завданнями на декількох кластерах одночасно за допомогою простого інтерфейсу для розгортання та керування додатками у контейнерах. Це робить Nomad популярним серед користувачів, які шукають легке та ефективне рішення для автоматизація роботи з контейнерами і надаючи більшу гнучкість у мульти-кластерних середовищах для компаній з розподіленими інфраструктурами. На Рисунок 3 — Графічний інтерфейс Nomad.

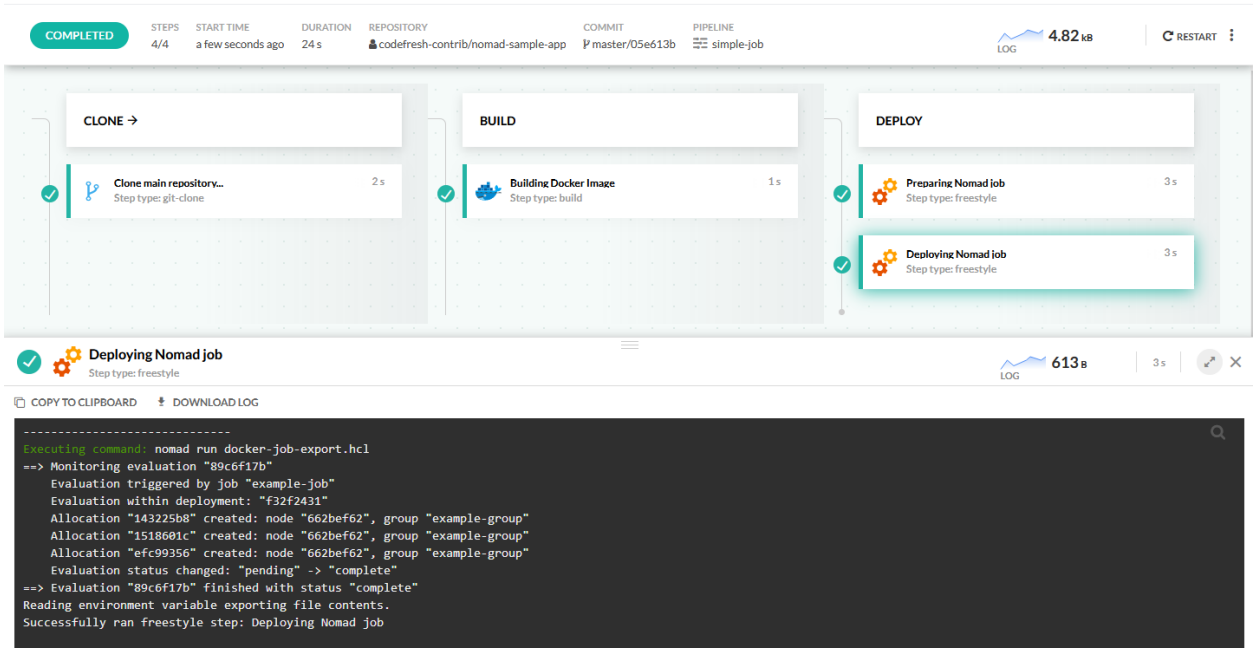


Рисунок 3 — Графічний інтерфейс Nomad

Nomad і Kubernetes — це два з основних програмних засобів для оркестрації контейнерів у сучасних обчислювальних середовищах. Обидва ці інструменти відомі своєю здатністю автоматизувати розгортання, масштабування та керування додатками у контейнерах, проте вони мають свої відмінності. Nomad, з іншого боку, відрізняється своєю простотою використання та меншою складністю установки та налаштування. Він зазвичай приваблює менші компанії та тих, хто шукає простіше рішення для оркестрації своїх додатків. Він також відрізняється більшою гнучкістю в управлінні мульти-кластерними середовищами.

Незважаючи на свою простоту, Nomad має потужні можливості для масштабування та керування великими інфраструктурами, що робить його підходящим для широкого спектру застосувань. Він інтегрується з іншими інструментами HashiCorp, такими як Vault для безпеки та Consul для сервісної сітки, що забезпечує додаткову гнучкість та контроль над інфраструктурою. Обираючи між Kubernetes і Nomad, слід враховувати специфіку проєкту, його потреби та особливості інфраструктури. Kubernetes може бути більш підходящим для складних та великих проєктів з великою кількістю

контейнерівх [11], тоді як Nomad може бути кращим вибором для менших проєктів або тих, хто шукає простіше та менш складне рішення.

1.3 Методи оптимізації продуктивності контейнеризованих додатків

Підвищення продуктивності контейнеризованих додатків є ключовим завданням для розробників, оскільки контейнеризація передбачає розподіл ресурсів і виконання багатьох сервісів на одній інфраструктурі. З огляду на це, оптимізація таких додатків стає важливим аспектом забезпечення їх ефективності та стабільності. Для вирішення цих завдань використовуються різноманітні інструменти та методи, які допомагають зменшити використання ресурсів, підвищити швидкість виконання контейнерів, покращити масштабованість і знизити час простою. До основних інструментів оптимізації продуктивності контейнеризованих додатків можна віднести:

1. Правильне використання ресурсів: Контейнери мають обмежені ресурси (процесор, пам'ять, дисковий простір), тому важливо налаштувати ліміти і квоти, що дозволяють ефективно керувати розподілом ресурсів між контейнерами.
2. Кешування та зберігання: Застосування кешування на рівні додатків, а також використання спеціальних файлових систем, оптимізованих для контейнерів, допомагає знизити затрати на введення/виведення та збільшити швидкість доступу до даних.
3. Масштабування та балансування навантаження: Оркестраційні системи, надають можливість автоматичного масштабування контейнерів та балансування навантаження між ними. Це дозволяє ефективніше використовувати ресурси і забезпечувати стабільну роботу додатків під час пікових навантажень.

4. Моніторинг та логування: Постійний моніторинг продуктивності контейнерів за допомогою таких інструментів, дозволяє оперативно виявляти вузькі місця у роботі додатків і оптимізувати їх роботу.

1.4 Автоматизація процесів у контейнеризованих середовищах

Автоматизація процесів у контейнеризованих середовищах є невід'ємною частиною сучасної веб-розробки, яка суттєво спрощує розгортання та управління додатками. Вона дозволяє підвищити ефективність, прискорити цикли розробки та зменшити кількість людських помилок, що, в свою чергу, забезпечує стабільність і надійність роботи веб-додатків.

Одним із ключових аспектів автоматизації є впровадження безперервної інтеграції та безперервного розгортання (CI/CD). Цей підхід дозволяє розробникам швидко перевіряти зміни в коді та автоматично розгортати їх на різних середовищах, скорочуючи час виходу на ринок. Інструменти, такі як Jenkins або GitLab CI, інтегруються з контейнерними платформами, що дозволяє здійснювати автоматичне тестування та розгортання, підвищуючи надійність процесів. Автоматизація також охоплює концепцію "інфраструктура як код" (IaC), що дозволяє описувати та управляти інфраструктурою за допомогою коду. Використовуючи такі інструменти, як Terraform або Ansible, команди можуть швидко налаштовувати, масштабувати та відтворювати середовища, забезпечуючи відповідність стандартам конфігурації. Це знижує ймовірність помилок, пов'язаних із ручним налаштуванням, і підвищує стабільність.

Ще однією важливою складовою автоматизації є можливість автоматичного масштабування. За допомогою оркестраторів, таких як Kubernetes [27], можна регулювати кількість контейнерів у відповідь на зміну навантаження, що дозволяє оптимально використовувати ресурси та підтримувати доступність додатків навіть у пікові моменти. Автоматизація забезпечує механізми для автоматичного відновлення системи. Сучасні

платформи здатні виявляти неполадки в контейнерах чи вузлах і швидко їх усувати, що дозволяє зменшити час простою та забезпечити безперервність роботи сервісів. Завдяки автоматизації, розгортання оновлень також стало більш зручним. Використовуючи методики, такі як "blue-green" або "canary" розгортання, можна впроваджувати нові версії додатків без переривання їх роботи, що критично важливо для забезпечення високої доступності веб-сервісів.

1.5 Постановка завдання

Мета кваліфікаційної роботи полягає у вивченні впливу контейнеризації та оркестрації на веб-розробку. Постановка завдання дослідження включає наступні етапи:

1. Аналіз потреб і вимог: Визначення основних потреб та вимог, які повинні бути задоволені під час використання контейнеризації та оркестрації веб-додатків. Це включає забезпечення швидкої розгортки, масштабування, керування та моніторингу.

2. Вибір технологій: Аналіз доступних технологій контейнеризації та оркестрації, таких як Docker та Kubernetes. Обґрунтування вибору найбільш підходящих технологій для веб-розробки.

3. Розробка архітектури: Визначення архітектури системи, що базується на контейнерах та оркестрації. Включає в себе розподіл компонентів додатку, створення контейнерних образів та налаштування оркестратора.

4. Розробка функціональності: Реалізація основних функціональних можливостей системи, враховуючи переваги контейнеризації та оркестрації. Включає в себе автоматизацію розгортки, масштабування, відновлення після збоїв та інші.

5. Апробація: Проведення тестування системи з використанням контейнерів та оркестрації для підтвердження її ефективності та коректності роботи.

1.6 Висновки до розділу 1

1. Оглянуті головні проблеми, що виникають у процесі створення веб-додатків з використанням контейнеризації та оркестрації. Серед них варто відзначити складність налаштування та управління контейнерами, вимоги до продуктивності, можливість виникнення проблем з масштабуванням та інші аспекти, які впливають на розробку веб-додатків.

2. Проаналізовано конкурентні технології контейнеризації Podman та Containerd, технології оркестрації Nomad. Були визначені переваги та недоліки кожної з цих технологій, а також їх відмінності та особливості застосування в контексті веб-розробки.

3. Розглянуто ефективні методи оптимізації продуктивності контейнеризованих додатків, які дозволяють підвищити швидкість їх виконання та зменшити використання ресурсів. Зокрема, обговорено стратегії кешування, налаштування мережевих з'єднань та вдосконалення конфігурації контейнерів, що суттєво впливають на загальну продуктивність системи.

4. Визначено важливість автоматизації процесів у контейнеризованих середовищах, яка дозволяє значно спростити управління додатками та інфраструктурою. Автоматизація забезпечує безперервну інтеграцію та розгортання, швидке масштабування, а також зменшує ризики, пов'язані з людським фактором. Завдяки цим процесам досягається стабільність і надійність роботи веб-додатків, що є критично важливим для їх успішної експлуатації.

5. Визначена постановка завдання дослідження, яке включає аналіз потреб та вимог до системи контейнеризації та оркестрації веб-додатків, вибір технологій, розробку архітектури, реалізацію функціональності та апробацію системи.

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ

2.1 Аналіз сучасних технологій для розробки веб-додатків з використанням контейнеризації та оркестрації

Сучасні технології контейнеризації та оркестрації значно спрощують розробку, розгортання та підтримку веб-додатків, дозволяючи створювати масштабовані, легко підтримувані та надійні системи. Основні задачі, які вирішують ці технології, включають ізоляцію процесів, управління ресурсами, автоматизацію масштабування, а також забезпечення надійності та швидкого відновлення після відмов. Розглянемо основні складові контейнеризації та оркестрації та їх особливості у сучасній розробці веб-додатків.

Контейнеризація забезпечує можливість створювати легковагові ізольовані середовища, де додатки можуть функціонувати незалежно від середовища виконання. Основною технологією контейнеризації є Docker, який дозволяє створювати, розгортати і керувати контейнерами. Docker використовує образи контейнерів, що містять усі необхідні для роботи додатка залежності, системні налаштування та файли [14]. Завдяки цьому розробники можуть створювати додатки, що гарантовано працюють однаково у різних середовищах — на локальних машинах, серверах або хмарних платформах. Контейнери надають можливість масштабувати додаток, розгортаючи додаткові інстанси для обробки високих навантажень, а також ізолюють додаток від впливу інших процесів, що підвищує стабільність та безпеку системи.

Контейнери, завдяки своїй ізольованості, знижують ризик виникнення конфліктів залежностей між компонентами додатка. Це дозволяє розробникам та DevOps-інженерам більш ефективно розподіляти обчислювальні ресурси та оптимізувати використання ресурсів сервера, адже кожен контейнер отримує необхідний обсяг пам’яті та процесорних ресурсів [17], визначений заздалегідь. Також варто зазначити, що, на відміну від віртуальних машин,

контейнери не потребують повної операційної системи всередині кожного середовища, що знижує використання ресурсів і забезпечує швидший запуск додатків.

Оркестрація контейнерів відповідає за управління кількома контейнерами, забезпечуючи їх взаємодію, масштабування та автоматизацію процесів обслуговування. Найпоширенішою платформою для оркестрації контейнерів є Kubernetes, розроблений для автоматизації розгортання, управління й масштабування контейнерних додатків у кластерному середовищі. Kubernetes дозволяє створювати кластери з набором вузлів (нодів), кожен з яких може мати кілька контейнерів. Це забезпечує можливість горизонтального масштабування, коли у відповідь на збільшення навантаження створюються додаткові екземпляри контейнерів. Ключовою концепцією Kubernetes є поди (Pods) — найменші керовані одиниці в кластері. Поди об'єднують один або кілька контейнерів, які спільно використовують ресурси, наприклад, мережу чи файлову систему. Вони забезпечують ефективну організацію роботи контейнерів і дають змогу швидко створювати, масштабувати або видаляти ресурси в залежності від потреб додатку [26]. Завдяки подам Kubernetes може забезпечувати балансування навантаження, ізоляцію середовищ і централізоване управління. Kubernetes також надає механізми для автоматичного відновлення, оскільки може автоматично перезапускати контейнери у разі збоїв або відмов. Для тестування і розробки Kubernetes можна використовувати Minikube, який дозволяє розгорнути локальний кластер Kubernetes. Він створює одновузловий кластер, що дає можливість розробникам працювати з Kubernetes без необхідності наявності великої інфраструктури. Minikube ідеально підходить для розробки та тестування контейнерних додатків на локальному середовищі, використовуючи різні драйвери віртуалізації, такі як VirtualBox або Docker [28]. Це дозволяє спростити процес інтеграції й налаштування для розробників, що прагнуть використовувати всі переваги Kubernetes у своїй локальній розробці.

Системи оркестрації забезпечують балансування навантаження між інстансами контейнерів, розміщуючи запити на сервери з урахуванням їхнього завантаження. Це допомагає підтримувати стабільність і швидкодію додатка навіть у випадках пікових навантажень. Також Kubernetes підтримує функції автоматизованого резервного копіювання та відновлення даних, що є важливим аспектом для забезпечення надійності веб-додатків. Для безпеки даних у Kubernetes [29] передбачені механізми управління доступом і політики мережевої безпеки, які захищають контейнери від несанкціонованого доступу та забезпечують ізоляцію середовищ.

У сучасній розробці веб-додатків популярними стають фреймворки NextJS та NestJS, які допомагають підвищити продуктивність і спрощують реалізацію необхідної функціональності. NextJS, як фреймворк для React, дозволяє створювати додатки з серверним рендерингом [21], що сприяє кращій продуктивності та оптимізації для пошукових систем. Завдяки підтримці статичної генерації та динамічного рендерингу, NextJS забезпечує швидке завантаження сторінок, що позитивно впливає на досвід користувачів. NestJS спрощує створення серверних додатків на основі Node.js, пропонуючи інструменти для роботи з мікросервісами та інверсією залежностей [20]. Використання TypeScript як основи забезпечує статичну типізацію, зменшуючи ймовірність помилок під час виконання програми та покращуючи читабельність коду. Завдяки своїм можливостям NestJS чудово підходить для контейнеризованих рішень, забезпечуючи ефективне управління маршрутизацією, контролерами та сервісами, що робить підтримку та масштабування додатків значно простішими.

У веб-додатках для роботи з базами даних часто використовують MySQL — потужну реляційну базу даних, яка забезпечує високу продуктивність і надійність. MySQL пропонує інтуїтивний інтерфейс для роботи з даними та підтримує розширені функції, такі як транзакції, індексація та реплікація, що робить її оптимальним вибором для багатьох проєктів. Щоб спростити взаємодію з MySQL, використовуються ORM, наприклад Sequelize,

яка дозволяє розробникам працювати з базою даних через об’єктно-орієнтований підхід. Це зменшує потребу у великій кількості SQL-коду. Sequelize підтримує асоціації, валідації та міграції, полегшуючи управління структурою бази даних під час розробки.

Для управління станом додатків у React популярними інструментами є Redux і Redux Saga. Redux централізує стан програми, що полегшує синхронізацію даних між компонентами. Redux Saga доповнює цей підхід, забезпечуючи ефективну обробку асинхронних операцій, таких як API-запити, покращуючи структуру коду та його тестованість. Поєднання цих технологій у контейнеризованих додатках сприяє модульності, спрощує управління станом і забезпечує гнучкість під час розгортання.

Використання сучасних технологій надає значні переваги в інтеграції та масштабуванні веб-додатків. Контейнеризація дозволяє швидко створювати нові екземпляри додатків на основі базових образів, забезпечуючи оперативну реакцію на зміни навантаження. В умовах мікросервісної архітектури, кожен сервіс може бути запакований в окремий контейнер, що дозволяє розгортати та масштабувати їх незалежно один від одного. Це забезпечує гнучкість і швидкість у розробці, оскільки команди можуть працювати над різними сервісами паралельно, не заважаючи один одному.

Використання таких інструментів, як Docker, Kubernetes, NextJS, NestJS, MySQL, Sequelize, Redux і Redux Saga, значно підвищує продуктивність і ефективність розробки веб-додатків. Це створює нові можливості для впровадження складних функцій, гарантуючи високий рівень надійності та стабільності роботи програмних продуктів.

2.2 Використання мікросервісної архітектури з контейнеризацією та оркестрацією

Мікросервісна архітектура — це підхід до розробки програмного забезпечення, який передбачає поділ системи на окремі незалежні сервіси,

кожен з яких відповідає за виконання конкретної функції. Такі сервіси можна розгортати та масштабувати окремо, що забезпечує гнучкість, швидкість розробки та підвищену стабільність додатка. У поєднанні з контейнеризацією та інструментами оркестрації, такими як Docker і Kubernetes, мікросервісна архітектура демонструє ще вищу ефективність і спрощує управління комплексними системами.

Контейнери та мікросервіси створюють ефективну комбінацію для побудови сучасних веб-додатків. Завдяки контейнеризації кожен мікросервіс можна розмістити в окремому контейнері, створюючи ізольоване середовище для кожної частини системи. Це значно спрощує управління залежностями, адже кожен контейнер містить власне середовище виконання та необхідні бібліотеки. Такий підхід мінімізує ризик конфліктів між компонентами, які часто виникають у монолітних системах. Контейнери також роблять розгортання зручнішим, оскільки мікросервіси можуть розгортатися в різних середовищах із незмінною конфігурацією.

Оркестрація контейнерів за допомогою Kubernetes забезпечує ефективне управління мікросервісною архітектурою, особливо у випадках масштабування та забезпечення відмовостійкості. Kubernetes автоматизує ключові завдання, такі як балансування навантаження, моніторинг стану, перезапуск у разі збоїв та розгортання оновлень. Це дозволяє мікросервісам динамічно масштабуватись відповідно до змін навантаження. Крім того, відмови одного мікросервісу не впливають на роботу інших, оскільки контейнери функціонують незалежно, забезпечуючи стабільність системи загалом.

Мікросервіси у контейнерах можна розподіляти на декількох вузлах (нодах) у кластері, що забезпечує гнучкість у використанні обчислювальних ресурсів [19]. Це також дозволяє автоматично балансувати навантаження між сервісами, зменшуючи затримки та покращуючи швидкість обробки запитів [25]. Замість того, щоб підтримувати великий монолітний додаток, де будь-яка зміна потребує оновлення всього коду, розробники можуть оновлювати або

масштабувати окремі мікросервіси без перерв у роботі системи. Завдяки цьому мікросервісна архітектура в поєднанні з оркестрацією спрощує процес управління додатком і прискорює виведення оновлень на ринок. Ще одним важливим аспектом при розробці мікросервісів є застосування принципів Domain-Driven Design (DDD) [10], яке дозволяє організувати мікросервіси навколо окремих бізнес-доменів. DDD сприяє чіткішому визначенню меж кожного мікросервісу, що допомагає уникнути надмірної складності та забезпечує ефективне управління складними системами. Кожен мікросервіс стає автономною одиницею, що відповідає за конкретний бізнес-процес або частину процесу, що дозволяє знизити залежності між компонентами системи.

Використання мікросервісної архітектури з контейнеризацією та оркестрацією не тільки покращує масштабованість і продуктивність веб-додатків, а й забезпечує їхню гнучкість, надійність і готовність до змін, що дозволяє адаптуватися до потреб бізнесу в режимі реального часу. Це робить мікросервіси, контейнери та оркестрацію незамінними інструментами для побудови розподілених систем та великих корпоративних додатків.

2.3 Виклики та особливості використання мікросервісів у контейнеризованому середовищі

Використання мікросервісної архітектури в поєднанні з контейнеризацією відкриває значні можливості для оптимізації веб-додатків, але також створює певні виклики, які вимагають обміркованих рішень. Зокрема, потреба забезпечення взаємодії між мікросервісами, управління даними, моніторинг і підтримка безперервної інтеграції та доставки (CI/CD) стають критично важливими аспектами у розробці сучасних веб-додатків.

Комунікація між мікросервісами є важливим завданням у контейнеризованій мікросервісній архітектурі, адже кожен мікросервіс функціонує як окремий компонент, розташований у власному контейнері. Для забезпечення ефективного обміну даними між контейнерами часто

використовуються REST API або gRPC для синхронної взаємодії, а також черги повідомлень або брокери повідомлень, такі як RabbitMQ чи Apache Kafka, для асинхронної передачі даних. Цей підхід дозволяє уникнути перевантажень та покращує масштабованість додатків, однак його налаштування та підтримка можуть бути складними.

Управління даними стає особливо складним, коли мікросервіси мають різні бази даних або повинні спільно використовувати один і той же набір даних. У багатьох системах кожен мікросервіс має власну базу даних, що знижує взаємозалежність між компонентами, але при цьому ускладнює підтримку узгодженості даних. Для вирішення цієї проблеми використовуються такі підходи, як база даних за шаблоном CQRS (Command Query Responsibility Segregation) та подійне джерело даних (Event Sourcing), що дозволяє підтримувати узгодженість за допомогою обміну подіями між сервісами. Це забезпечує високий рівень надійності та інтеграції даних, але вимагає значних зусиль для налаштування і синхронізації.

Моніторинг і логування є ключовими компонентами стабільності мікросервісного додатка, адже контейнеризація значно ускладнює традиційний підхід до логування та спостереження за продуктивністю. Для відстеження роботи та швидкої діагностики помилок необхідно застосовувати спеціалізовані інструменти для логування та моніторингу, такі як Prometheus, Grafana та ELK Stack (Elasticsearch, Logstash, Kibana). Використання цих інструментів дозволяє отримувати повну картину функціонування системи, а також реагувати на потенційні проблеми, перш ніж вони вплинуть на користувачів.

CI/CD (безперервна інтеграція та доставка) є важливим елементом розробки мікросервісів у контейнеризованому середовищі, оскільки регулярні оновлення та виправлення є звичайною практикою для таких систем. Інтеграція інструментів CI/CD, таких як Jenkins, GitLab CI/CD або Argo CD, дозволяє автоматизувати процеси тестування, створення контейнерів та їх

розгортання. Це прискорює доставку оновлень, знижує ризик помилок і забезпечує надійність системи навіть при частих змінах.

Безпека у мікросервісній архітектурі з використанням контейнеризації є важливим питанням, адже контейнеризовані додатки часто функціонують у відкритих середовищах з підвищеними вимогами до захисту даних. Контейнери мають свої специфічні вразливості, такі як ризики ескалації привілеїв і втрати ізоляції, якщо контейнер некоректно налаштовано. Для захисту використовуються такі методи, як рольове управління доступом (RBAC), мережеві політики Kubernetes для обмеження взаємодії між контейнерами, а також інструменти безпеки для сканування контейнерних образів на наявність вразливостей перед розгортанням. Крім того, впровадження політик секретів і шифрування даних у сховищах допомагає забезпечити захист конфіденційної інформації в умовах розподіленого середовища.

2.4 Стратегії масштабування контейнеризованих мікросервісів

Масштабування є однією з ключових переваг контейнеризованих мікросервісів, що дозволяє забезпечити необхідний рівень продуктивності системи при зростанні навантаження. У середовищах з контейнерами та мікросервісами застосовуються дві основні стратегії масштабування: горизонтальне та вертикальне.

Горизонтальне масштабування scale-out полягає у збільшенні кількості екземплярів мікросервісів, щоб збільшити потужність системи. Коли навантаження на додаток зростає, контейнерна платформа, наприклад Kubernetes, може автоматично створювати додаткові репліки сервісів і розподіляти запити між ними. Це забезпечує стійкість системи до збоїв, дозволяє уникати «вузьких місць» та підвищує доступність додатка, оскільки при збільшенні навантаження більше екземплярів обробляють запити паралельно. Автоматичне горизонтальне масштабування HPA дозволяє

встановлювати метрики, такі як використання CPU або пам'яті, при яких додаток автоматично розширюватиметься або зменшуватиметься.

Вертикальне масштабування scale-up, у свою чергу, передбачає збільшення ресурсів (CPU, RAM) для кожного екземпляра мікросервісу. Це може бути корисним для ресурсомістких мікросервісів, які виконують складні обчислення або обробляють великі обсяги даних. Однак вертикальне масштабування має свої обмеження: у міру збільшення ресурсів на один екземпляр зростають витрати, і досягнення граничних можливостей сервера може вимагати фізичної заміни обладнання. Тому вертикальне масштабування є менш гнучким і частіше використовується в комбінації з горизонтальним для оптимізації продуктивності додатка.

Балансування навантаження є критичним елементом для забезпечення ефективного масштабування. В системах з мікросервісною архітектурою балансування виконується як на рівні інфраструктури, так і на рівні застосунку. Наприклад, у Kubernetes зазвичай використовується Ingress-контролер для управління вхідним трафіком та розподілення запитів до відповідних мікросервісів. Крім цього, існують технології API Gateway, що об'єднують функції маршрутизації запитів, балансування навантаження та авторизації, надаючи централізовану точку доступу до всіх мікросервісів.

Кешування також є важливою стратегією оптимізації продуктивності при масштабуванні, зокрема для зменшення навантаження на бази даних та інші ресурсоємні сервіси. Кешування часто реалізується за допомогою Redis або Memcached, де проміжні дані можуть зберігатися для швидкого доступу. Це дозволяє зменшити кількість запитів до мікросервісів і баз даних, прискорюючи час відповіді для кінцевого користувача. Розподіл стану між мікросервісами також є критично важливим для масштабованої архітектури. Оскільки мікросервіси у контейнерному середовищі часто є статичними і не зберігають дані про стан, розподіл стану забезпечується через зовнішні системи, такі як бази даних або служби зберігання. Наприклад, для обробки сесій користувачів або черг завдань часто використовуються NoSQL бази

даних або брокери повідомлень. Це дозволяє сервісам залишатися незалежними і підвищує надійність, оскільки навіть при завершенні роботи одного з мікросервісів дані користувача зберігаються.

2.5 Забезпечення надійності та відмовостійкості контейнеризованих мікросервісів

Для забезпечення високої надійності та відмовостійкості контейнеризованих додатків необхідні спеціальні підходи, що включають стратегії автоматичного відновлення, моніторинг стану сервісів, реплікацію даних, а також використання політик управління помилками.

Контейнерні платформи, такі як Kubernetes, пропонують вбудовані механізми для управління відмовами і забезпечення безперервної доступності сервісів. Наприклад, Kubernetes автоматично перезапускає контейнери, які зазнали збою, і перенаправляє трафік на інші працюючі екземпляри мікросервісів [30]. Це дозволяє підтримувати стійкість системи навіть у випадку помилок на окремих вузлах. Моніторинг і логування є невід'ємною частиною забезпечення відмовостійкості. За допомогою інструментів, таких як Prometheus, Grafana, ELK Stack, можна здійснювати безперервний моніторинг стану контейнерів, збирати метрики продуктивності та журналювати події. Це дозволяє не лише виявляти проблеми в реальному часі, але й проводити аналіз причин відмов та оцінювати ефективність обраних стратегій відновлення.

Реплікація даних — ще один важливий аспект надійності. У системах з контейнеризованими мікросервісами реплікація здійснюється на рівні баз даних або через використання розподілених файлових систем. Така архітектура дозволяє уникнути втрати даних при збоях і забезпечити доступ до них із різних вузлів, що критично важливо для збереження цілісності інформації в розподілених середовищах.

Автоматизація процесів відновлення значно підвищує відмовостійкість систем. Використання автоматизованих сценаріїв дозволяє швидко відновлювати додаток після збоїв або оновлень. Такі сценарії охоплюють дії з відновлення сервісів, управління залежностями та подолання наслідків аварій, що мінімізує ризик людських помилок і забезпечує оперативне відновлення роботи.

Стратегії оновлень без зупинки роботи допомагають впроваджувати зміни в мікросервісах без впливу на функціонування всієї системи. Blue-Green Deployment передбачає запуск нової версії додатка паралельно з чинною, що дозволяє поступово перевести користувачів на оновлення після успішного тестування. Canary Release дає змогу спершу оновлювати лише невелику кількість екземплярів сервісу, аналізувати їхню роботу та поступово збільшувати масштаб впровадження у разі позитивних результатів.

2.6 Висновки до розділу 2

1. Проведено аналіз сучасних технологій контейнеризації та оркестрації, що застосовуються для розробки веб-додатків, та визначено їхні переваги і виклики в управлінні контейнерами в розподілених середовищах.
2. Розглянуто використання мікросервісної архітектури у поєднанні з контейнеризацією та оркестрацією, що дозволяє забезпечити гнучкість, модульність і спрощене управління сервісами на етапах розробки та експлуатації.
3. Виявлено виклики та особливості роботи з мікросервісами у контейнеризованому середовищі, включаючи складності у забезпеченні взаємодії сервісів, управлінні ресурсами та контроль логування, що потребує спеціальних підходів до конфігурації.
4. Визначено стратегії масштабування контейнеризованих мікросервісів, зокрема горизонтальне та вертикальне масштабування, що дозволяють адаптувати додатки до зростаючих навантажень без втрати продуктивності.

5. Окреслено методи забезпечення надійності та відмовостійкості контейнеризованих додатків, включаючи моніторинг, автоматизацію відновлення, реплікацію даних і використання стратегій поступового оновлення, що є ключовими для підтримки стабільної роботи системи.

3 РОЗРОБКА ВЕБ-ДОДАТКУ З ВИКОРИСТАННЯМ СУЧАСНИХ ТЕХНОЛОГІЙ КОНТЕЙНЕРИЗАЦІЇ ТА ОРКЕСТРАЦІЇ

3.1 Опис предметної області

Предметна область цієї роботи охоплює розробку сучасного веб-додатка, що використовує мікросервісну архітектуру, контейнеризацію та оркестрацію, для забезпечення його стабільності, масштабованості та гнучкості в умовах динамічного навантаження. Вибір цих технологій обумовлений їхніми перевагами у побудові складних веб-систем, здатних ефективно функціонувати у розподілених середовищах та забезпечувати високу відмовостійкість.

Мікросервісна архітектура є підходом до розробки програмного забезпечення, при якому додаток розділяється на низку незалежних сервісів (мікросервісів), кожен з яких виконує окрему бізнес-функцію. Це спрощує підтримку та розвиток додатка, дозволяє кожній команді працювати з конкретним мікросервісом і забезпечує легке масштабування лише тих частин додатка, які зазнають підвищеного навантаження. Така архітектура сприяє стабільності, адже кожен сервіс працює автономно та мінімально впливає на інші компоненти додатка. Контейнеризація, зокрема з використанням Docker, забезпечує ізоляцію кожного мікросервісу у власному середовищі, яке містить усі необхідні залежності та налаштування для його функціонування. Контейнери дозволяють швидко розгортати нові інстанси мікросервісів, забезпечуючи стабільну роботу додатка в різних середовищах — на локальних машинах, серверах або у хмарі. Завдяки цьому контейнери полегшують процес тестування, автоматизації та забезпечення цілісності програмного забезпечення.

Оркестрація контейнерів за допомогою Kubernetes є ще одним ключовим компонентом предметної області, оскільки дозволяє управляти великим числом контейнерів у кластері, автоматизувати процеси розгортання,

масштабування та відновлення додатка після збоїв. Kubernetes підтримує горизонтальне масштабування, балансування навантаження та автоматичний перезапуск контейнерів, що робить систему більш стійкою до відмов та забезпечує її безперебійну роботу навіть у випадку збоїв окремих сервісів.

Ця дипломна робота спрямована на дослідження та впровадження найкращих практик у використанні мікросервісної архітектури та інструментів для контейнеризації та оркестрації. Результати цієї роботи допоможуть сформувати розуміння принципів побудови розподілених веб-додатків, що є актуальним у сучасному IT-середовищі, де висока надійність і масштабованість додатків стають невід’ємними вимогами.

3.2 Архітектура додатку “Real Estate”

Архітектура є ключовим аспектом при створенні контейнеризованих мікросервісних систем, оскільки вона визначає структуру, взаємодію компонентів та принципи організації коду. Вибір мікросервісного підходу дозволяє розділити функціональність додатка на незалежні сервіси, кожен з яких відповідає за окрему бізнес-логіку або функціональність, що забезпечує гнучкість, масштабованість та спрощує підтримку. Рисунок 4 — Взаємодія Frontend частини з Backend.

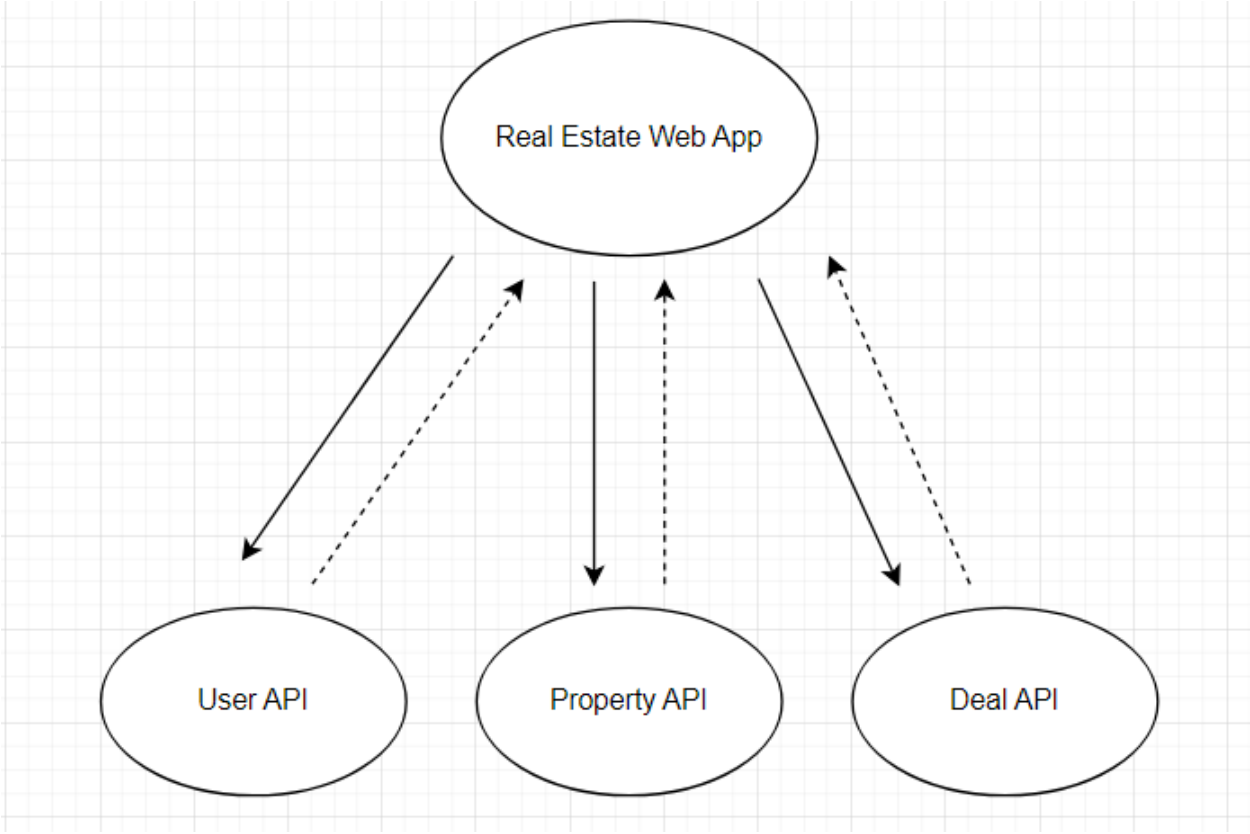


Рисунок 4 — Взаємодія Frontend частини з Backend

Для реалізації архітектури було розподілено додаток на два шари:

1. Frontend частина додатку, що побудована на монолітній архітектурі
2. Backend частина додатку, що розподілена на три окремі мікросервіси: User API, Property API та Deal API.

3.2.1 Frontend частина додатку “Real Estate”

Frontend частина додатку “Real Estate” побудована на монолітній архітектурі з використанням фреймворку NextJS. Монолітна архітектура для frontend-додатку дозволяє зберігати всю бізнес-логіку інтерфейсу в одному модулі, що полегшує управління залежностями, забезпечує централізовану обробку та спрощує розгортання.

NextJS як основа frontend-додатку забезпечує продуктивну та зручну платформу для розробки. Фреймворк поєднує серверний рендеринг з клієнтським [24], що покращує швидкість завантаження сторінок та забезпечує ефективну індексацію контенту пошуковими системами, що є

важливим для веб-проектів у сфері нерухомості. Завдяки статичному та динамічному рендерингу сторінок [23], NextJS дозволяє забезпечувати користувачів актуальною інформацією про об’єкти нерухомості та швидкий відгук інтерфейсу. Монолітний підхід у frontend-частині забезпечує єдиний потік обробки запитів та швидкий доступ до всіх компонентів та модулів інтерфейсу. Це спрощує підтримку та оновлення коду, оскільки розробники можуть вносити зміни в єдиному середовищі, не потребуючи складного налаштування залежностей між окремими компонентами.

“Real Estate” використовує контейнер **awilix** для управління залежностями, що дозволяє централізовано зберігати та керувати сервісами і Redux-станом. **Awilix** виступає як контейнер для залежностей, який містить реалізації основних сервісів додатку, таких як сервіси для взаємодії з API, обробки даних. Також у контейнері **awilix** розміщений інстанс Redux-стору, який забезпечує єдиний централізований стан для всього додатку.

У frontend частині додатку “Real Estate” для управління станом використовується бібліотека Redux, що забезпечує централізоване зберігання та управління даними додатку. Redux дозволяє ефективно організувати потоки даних між компонентами, зберігаючи всю інформацію про стан інтерфейсу в єдиному сховищі. Це особливо корисно для підтримки складних взаємодій між компонентами в монолітному додатку, де є необхідність у спільному доступі до даних між різними частинами інтерфейсу. Тому для цього було створено зручний “reducer”, який розподіляє данні по конкретним сховищам, що мають певні об’єкти для поточних редюсерів. Такий підхід дозволяє економити час розробників на написання однотипних функцій-хендлерів при кожному запиті на API. Для цього для кожної сутності створюється окремий “сервіс”, який в свою чергу містить “actions” що реалізують відправку запитів на API та зберігання данних redux сховищі.

Лістинг *1 Глобальна функція відправки запитів на API*

```
    protected *requestResult<ReqBody extends Object, ResBody
extends Object>
        (config: HttpRequestConfig<ReqBody>, httpMethod:
string, actionMethod: string) {
            let data: object | Error = yield
call(this.sendRequest<ReqBody, ResBody>, config.url,
httpMethod, config.body, config.headers);
            let currentData: any = data;
            if (currentData.hasOwnProperty(MESSAGE)) {
                const isError = data instanceof Error;
                yield put({
                    type:
ToastifyEffectActions.ADD_TOASTIFY,
                    payload: this.generateToastify(
                        currentData.message, isError ?
ToastifyStatus.Error : ToastifyStatus.Success
                    )
                });
                if (isError) return currentData;
            }
            const action =
this.normalizeReqBody(currentData, actionMethod);
            yield put(action);
            return data as ResBody;
        }
```

Дані в Redux зберігаються в нормалізованому вигляді, що дозволяє ефективно керувати великими та взаємопов’язаними наборами даних. Для нормалізації даних використовується бібліотека **normalizr**, яка автоматично перетворює вкладені структури об’єктів у нормалізовані схеми. Це означає, що всі дані зберігаються у вигляді плоских структур з унікальними ідентифікаторами, що значно полегшує доступ, оновлення та видалення

окремих елементів. Нормалізація даних допомагає уникнути дублювання інформації та дозволяє Redux легко відстежувати зміни в конкретних об'єктах. Використання normalizr спрощує обробку великих обсягів даних, таких як списки об'єктів нерухомості та детальна інформація про них, оскільки всі взаємозв’язки між даними зберігаються через посилання на унікальні ідентифікатори. Це забезпечує оптимізацію продуктивності та підтримує високу швидкодію додатку.

Для обробки асинхронних операцій, таких як запити до backend частини системи, використовується Redux Saga. Redux Saga спрощує управління асинхронністю за допомогою «саг» – функцій-генераторів, що дозволяють легко контролювати виконання складних асинхронних процесів, таких як запити до API, обробка результатів та обробка помилок. Це забезпечує надійність і стабільність під час обробки великих обсягів даних, наприклад, при завантаженні або оновленні інформації про об’єкти нерухомості. Для отримання даних були створені сервіси, що містять в собі так звані **watchers**, що очікують на виклик **action**, після якого виконується певні дії **workers** по отриманню/зміні даних у сховищах. Також сервіси ініціалізують схеми для нормалізації отриманих даних.

Лістинг 2 *Частина сервісу PropertyService*

```
class PropertyService extends HttpService {
    constructor() {
        super();
        const propertyImageSchema = new schema.Entity('propertyImages', {}, { idAttribute: 'propertyImageId' });
        const userSchema = new schema.Entity('users', {}, { idAttribute: 'userId' });
        this.initSchema(
            'properties',
```

```
                {   PropertyImages:   [propertyImageSchema],
User: userSchema },
                { idAttribute: 'propertyId' }
            );
        }

        public *getLastOffers() {
            yield takeEvery(
                PropertyEffectActions.GET_LAST_OFFERS,
                this.get<(PropertyModel & { PropertyAddress:
PropertyAddressModel; })[]>,
                {
                    url: BACK_PATHS.getLastOffers,
                    cleanEntities:      [Entities.Property,
Entities.PropertyImage]
                },
                ReducerMethods.UPDATE
            );
        }
```

3.2.2 Backend частина додатку “Real Estate”

Архітектура backend-частини додатку "Real Estate" побудована на основі мікросервісного підходу, що забезпечує модульність та можливість незалежного розгортання кожного з компонентів. Мікросервіси розроблені з використанням NestJS – прогресивного Node.js фреймворку, що пропонує ефективні інструменти для побудови масштабованих серверних додатків. Кожен мікросервіс є автономним, відповідальним за окремий функціональний блок, і взаємодіє з іншими через чітко визначені API.

Backend-частина системи складається з трьох мікросервісів:

1. UserAPI – відповідає за операції з користувачами.
2. PropertyAPI – керує даними про об'єкти нерухомості.
3. DealAPI – забезпечує управління угодами.

Кожен мікросервіс спроектований таким чином, щоб бути незалежним компонентом, який можна розгортати та масштабувати окремо. Архітектура орієнтована на підвищення надійності та оптимізацію обробки запитів у розподіленій системі, де мікросервіси взаємодіють один з одним через стандартизовані API.

Основним архітектурним шаблоном, що застосовується у backend-частині, є CQRS. CQRS розділяє операції на команди (запити, що змінюють стан системи) та запити (операції на отримання даних). Завдяки CQRS структура кожного мікросервісу дозволяє розділити бізнес-логіку та взаємодію з даними таким чином, щоб зменшити ризик конфліктів і підвищити продуктивність системи. Крім того, backend частина була побудована відповідно до принципів DDD [9]. Цей підхід забезпечує чітке відокремлення доменних об'єктів, які відображають основну бізнес-логіку, від інфраструктурних компонентів, таких як доступ до даних або API. Усі основні бізнес-операції реалізовані через агрегати, що відповідають за збереження цілісності даних і забезпечують взаємодію між сутностями в рамках визначених доменних контекстів. Така архітектура сприяє розширюваності системи, підвищує її стійкість до змін і спрощує внесення нових бізнес-функцій.

Команди обробляють операції, що змінюють стан, наприклад, додавання або оновлення записів у базі даних. Для прикладу приведу домен “Property”. Цей домен, як інші домени “User” та “Deal”, реалізовує архітектуру для роботи з об'єктами нерухомості в додатку "Real Estate" на основі патернів DDD і CQRS.

Лістинг *3 Сутність “Property” у домені “Property”*

```
export class Property extends AggregateRoot {
  constructor(
    private readonly _propertyId: string,
    private readonly _bedRooms: number,
```

```
    private readonly _bathRooms: number,
    private readonly _area: number,
    private readonly _title: string,
    private readonly _description: string,
    private readonly _priceAmount: number,
    private readonly _propertyStatus: PropertyStatuses,
    private readonly _propertyType: PropertyTypes,
    private readonly _userId: string,
    private readonly _createdAt: bigint,
    private readonly _updatedAt: bigint,
    private _propertyAddress?: PropertyAddressDto,
    private _propertyImages?: PropertyImageDto[],
    private _user?: PublicUserDto,
    private _deals?: DealDto[],
  ) {
    super();
  }

  get propertyId(): string {
    return this._propertyId;
  }

  get bedRooms(): number {
    return this._bedRooms;
  }

  toDto(): PropertyDto {
    return {
      propertyId: this.propertyId,
      bedRooms: this.bedRooms,
      bathRooms: this.bathRooms,
      area: this.area,
      title: this.title,
```

```
        description: this.description,
        priceAmount: this.priceAmount,
        propertyStatus: this.propertyStatus,
        propertyType: this.propertyType,
        userId: this.userId,
        createdAt: this.createdAt,
        updatedAt: this.updatedAt,
        propertyAddress: this.propertyAddress,
        propertyImages: this.propertyImages,
        user: this.user,
        deals: this.deals,
      };
    }
  }
```

Клас Property: Це головний доменний об'єкт, який представляє нерухомість і містить всю необхідну інформацію про неї:

1. Поля, що описують властивості нерухомості, такі як кількість кімнат, площа, статус та тип, ID користувача-власника, час створення та оновлення.

2. Клас успадковує AggregateRoot, що означає, що Property є агрегатом – основним об'єктом, який відповідає за інтеграцію бізнес-логіки з іншими сутностями.

3. Для кожного поля оголошені методи доступу get та set, що дозволяє читати та змінювати певні властивості, зберігаючи інкапсуляцію.

4. Метод toDto перетворює об'єкт Property у Data Transfer Object (DTO), що дозволяє передавати дані між компонентами системи у стандартизованому форматі.

Лістинг *4 Фабрика "PropertyModelFactory"*

```
export class PropertyModelFactory
  implements          EntityModelFactory<PropertyModel,
Property>
{
  createModel(entity: Property): PropertyModel {
    return new PropertyModel({
      propertyId: entity.propertyId,
      area: entity.area,
      bathRooms: entity.bathRooms,
      bedRooms: entity.bedRooms,
      createdAt: entity.createdAt,
      description: entity.description,
      priceAmount: entity.priceAmount,
      propertyStatus: entity.propertyStatus,
      propertyType: entity.propertyType,
      title: entity.title,
      ...
    });
  }
  createEntity(model: PropertyModel): Property {
    return new Property(
      model.propertyId,
      model.bedRooms,
      model.bathRooms,
      model.area,
      model.title,
      ...
    );
  }
}
```

Клас PropertyModelFactory: Фабрика, яка служить для перетворення між доменною сутністю (Property) і моделлю бази даних (PropertyModel):

1. Метод createModel приймає об'єкт Property і повертає модель PropertyModel для зберігання в базі даних.

2. Метод createEntity виконує зворотне перетворення – створює об'єкт Property з моделі PropertyModel, що дає можливість зберегти цілісність даних при взаємодії з базою.

Лістинг *5 Фабрика "PropertyFactory"*

```
export class PropertyFactory implements EntityFactory<Property> {
  constructor(
    private readonly propertyEntityRepository: PropertyEntityRepository,
  ) {}

  async create(dto: CreatePropertyDto): Promise<Property> {
    const createdAt = BigInt(new Date().getTime());
    const updatedAt = createdAt;
    const propertyId = uuid.v4();
    const property = new Property(
      propertyId,
      dto.bedRooms,
      dto.bathRooms,
      dto.area,
      dto.title,
      dto.description,
      dto.priceAmount,
      dto.propertyStatus,
      dto.propertyType,
      dto.userId,
      createdAt,
      updatedAt,
```

```
    );
    await
this.propertyEntityRepository.create(property.toDto());
    return property;
  }
}
```

Клас PropertyFactory: Основний інструмент для створення та оновлення об'єктів Property.

1. Фабрика містить метод create, який генерує новий об'єкт нерухомості з унікальним propertyId, отриманим за допомогою UUID, а також з полями createdAt та updatedAt, що задають час створення та оновлення.
2. Після створення об'єкт Property перетворюється на DTO за допомогою методу toDto і зберігається у базі даних через репозиторій PropertyEntityRepository.
3. Метод update дозволяє оновити дані нерухомості на основі об'єкта DTO UpdatePropertyDto. Він також перевіряє наявність об'єкта в базі даних; якщо об'єкт знайдений, то його оновлюють, якщо ні, – кидається виняток NotFoundException

Для управління цими процесами кожен мікросервіс має свою окрему командну шину та обробники запитів. Це дозволяє зосередитися на оптимізації операцій для кожного типу дій (читання та запису), підвищуючи швидкість обробки даних і забезпечуючи розширюваність системи.

Кожен із мікросервісів взаємодіє з базою даних за допомогою Sequelize ORM — потужного інструмента для роботи з реляційними базами даних. Sequelize спрощує управління даними, дозволяючи уникнути безпосереднього написання складних SQL-запитів. Ця ORM дозволяє централізовано керувати схемою бази даних, а також дотримуватися єдиних стандартів роботи з даними, що спрощує підтримку та розвиток системи.

Оскільки кожен мікросервіс відповідає за конкретну бізнес-логіку і має свій контекст, вони не взаємодіють безпосередньо з даними інших сервісів. Для взаємодії мікросервіси використовують HTTP API та, за необхідності,

асинхронні повідомлення. Це дозволяє знизити навантаження на систему та забезпечує гнучкість у розподіленій архітектурі, дозволяючи окремим сервісам бути незалежними та самодостатніми.

3.2.3 DB частина додатку “Real Estate”

Частина бази даних в клієнт-серверній архітектурі відповідає за зберігання, організацію та керування даними, які використовуються в веб-додатку. База даних є центральним елементом системи, яка використовується для довготривалого зберігання даних та може бути доступна для використання серверної частини. Для побудови було обрано реляційну базу даних **MySQL**, що надає можливість виконувати складні операції з даними, забезпечуючи цілісність даних, масштабованість, стабільність.

База даних містить в собі такі сутності(Табл.1 — 5):

Таблиця 1 — Таблиця “Users”

Users
userId: унікальний рядок, що містить ідентифікатор користувача
firstName: рядок, що містить ім’я користувача
lastName: рядок, що містить прізвище користувача
email: рядок, що містить електронний адрес користувача
password: рядок, що містить пароль користувача
email: рядок, що містить email користувача
createdAt: число, що містить в собі дату створення користувача
updatedAt: число, що містить в собі дату оновлення користувача

Таблиця 2 — Таблиця “Properties”

Properties
propertyId: унікальний рядок, що містить ідентифікатор власності
bedRooms: число, що містить в собі кількість кімнат
bathRooms: число, що містить в собі кількість ванних кімнат
area: число, що містить в собі загальну площу
priceAmount: число, що містить в собі ціну
title: рядок, що містить в собі заголовок
description: рядок, що містить в собі опис
propertyStatus: enum, що містить в собі статус
propertyType: enum, що містить в собі тип
userId: зовнішній ключ на таблицю ‘Users’
propertyAddressId: зовнішній ключ на таблицю ‘PropertyAddresses’
createdAt: число, що містить в собі дату створення власності
updatedAt: число, що містить в собі дату оновлення власності

Таблиця 3 — Таблиця “PropertyAddresses”

PropertyAddresses
propertyAddressId: унікальний рядок, що містить ідентифікатор адреси власності
countryName: рядок, що містить в собі назву країни
cityName: рядок, що містить в собі назву міста
addressLine1: рядок, що містить в собі адресу 1
addressLine2: необов’язковий рядок, що містить в собі адресу 2

Таблиця 4 — Таблиця “PropertyImages”

PropertyImages
propertyImageId: унікальний рядок, що містить ідентифікатор зображення власності
imgName: рядок, що містить в собі назву зображення
propertyId: зовнішній ключ на таблицю ‘Properties’

Таблиця 5 — Таблиця “Deals”

Deals
dealId: унікальний рядок, що містить ідентифікатор угоди
signDate: число, що містить в собі дату підписання угоди
totalPrice: число, що містить в собі загальну ціну
dealStatus: enum, що містить в собі статус
propertyId: зовнішній ключ на таблицю ‘Properties’
sellerUserId: зовнішній ключ на таблицю ‘Users
buyerUserId: зовнішній ключ на таблицю ‘Users
createdAt: число, що містить в собі дату створення угоди
updatedAt: число, що містить в собі дату оновлення угоди

Було створено таку ER схему бази даних даної системи. Рисунок 5 — ER діаграма БД проекту Real Estate

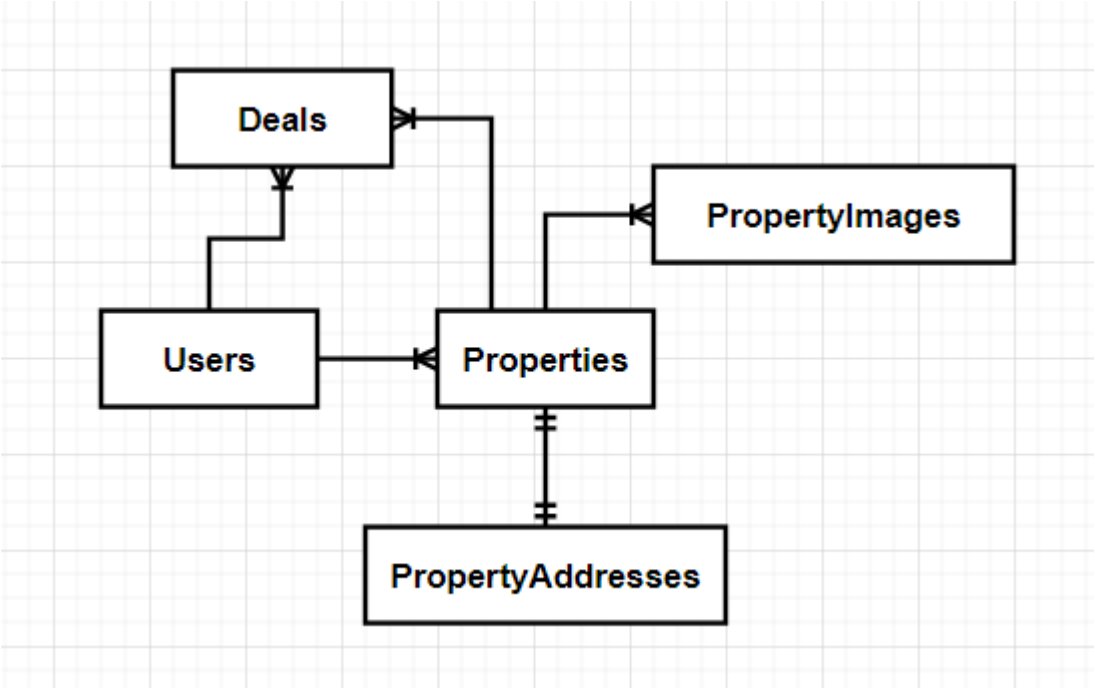


Рисунок 5 — ER діаграма БД проекту Real Estate

3.3 Функціонал сайту “Real Estate”

Функціонал сайту "Real Estate" забезпечує всі необхідні можливості для зручного управління нерухомістю та взаємодії з користувачами. Перш за все, платформа пропонує авторизацію користувачів, яка гарантує безпеку доступу до персональних даних та приватної інформації. Користувачі можуть створювати власні облікові записи або входити до системи за допомогою наявного облікового запису, після чого отримують доступ до функціоналу платформи відповідно до своїх прав.

Одним із ключових елементів є можливість оновлення даних користувача. Це дозволяє користувачам редагувати інформацію про себе, наприклад, змінювати контактні дані, оновлювати адресу чи додавати нову інформацію, що допомагає забезпечити актуальність їх профілів. На головній сторінці сайту відображаються останні додані об'єкти нерухомості, які одразу привертають увагу користувачів. Це чудова можливість побачити актуальні пропозиції, щойно вони з’являються, що особливо важливо для тих, хто хоче

оперативно реагувати на нові пропозиції. Рисунок 6 — Головна сторінка сайту “Real Estate”.

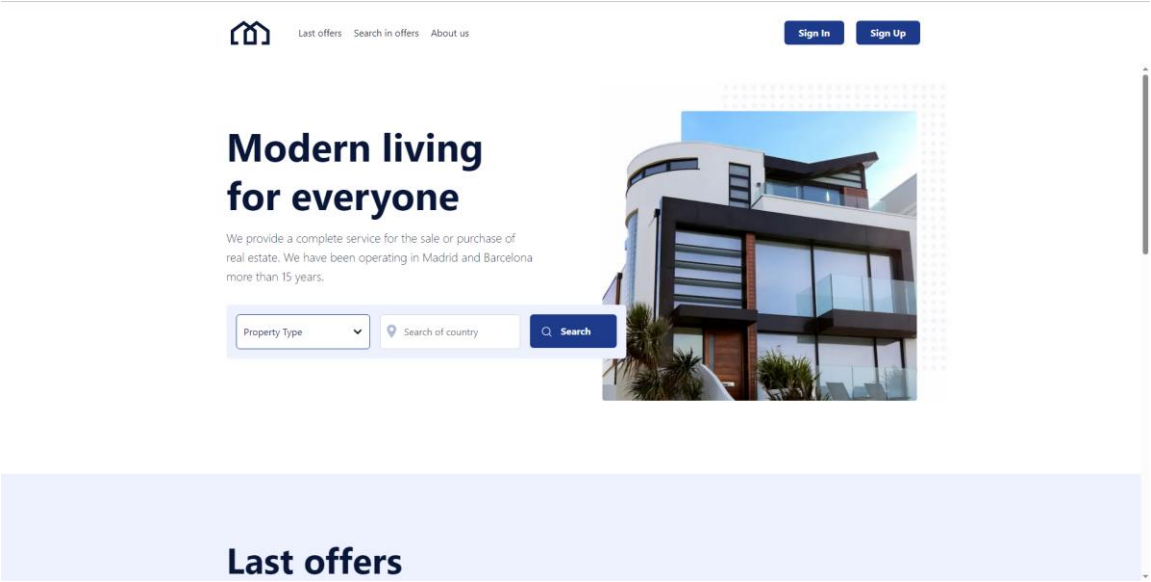


Рисунок 6 — Головна сторінка сайту “Real Estate”

Розширений функціонал включає відображення всіх доступних об'єктів нерухомості із можливістю фільтрації та пошуку, що забезпечує зручність користування навіть для найвибагливіших клієнтів. Користувач може відфільтрувати нерухомість за такими критеріями, як ціна, тип (квартира, будинок, вілла), розташування, кількість кімнат тощо. Пагінація даних робить процес перегляду списку нерухомості комфортним, адже великий обсяг інформації поділяється на зручні сторінки. Для тих, хто зацікавлений у конкретній пропозиції, передбачена можливість перегляду детальної інформації про конкретну нерухомість. Відвідувачі сайту можуть отримати повний набір даних: ціну, адресу, площу, кількість кімнат, а також дізнатися більше про брокера, який займається цією нерухомістю.

Окремою важливою частиною функціоналу є можливість створення угод, завдяки чому можна оформити передачу нерухомості від однієї особи до іншої. Цей процес прозорий, зрозумілий і дозволяє зберегти всі необхідні юридичні деталі. Для зручності користувачів сайт також містить розділ для перегляду всіх своїх угод. Це дозволяє швидко знаходити інформацію про вже

укладені або активні угоди, відслідковувати їх статус та, за потреби, повертатися до деталей.

3.4 Інтеграція Docker та Kubernetes у процес розробки та розгортання “Real Estate”

Для забезпечення ізольованості середовищ розробки та розгортання всі мікросервіси системи "Real Estate" були контейнеризовані за допомогою Docker. Кожен мікросервіс має власний Dockerfile, у якому детально описані інструкції для створення контейнера. Це включає вибір базового образу, встановлення залежностей, копіювання вихідного коду та виконання налаштувань для середовища виконання. Такий підхід дозволяє створювати однакові контейнери незалежно від середовища, в якому вони запускаються, забезпечуючи консистентність виконання програмного забезпечення.

Використання Docker Compose дозволило ефективно організувати локальну розробку. Завдяки єдиному конфігураційному файлу всі мікросервіси системи, а також необхідні залежності, такі як бази даних MySQL, можуть бути запущені одночасно [18]. Це забезпечує повну симуляцію середовища продуктивного розгортання, що дозволяє розробникам тестувати інтеграцію між мікросервісами без необхідності налаштовувати кожен сервіс окремо. Також Docker Compose підтримує змінні оточення, які використовуються для конфігурації сервісів. Це дозволяє гнучко змінювати параметри, наприклад URL бази даних або токени доступу, залежно від середовища.

Щоб уникнути втрати даних, у Docker Compose налаштовано використання Docker Volume, які дозволяють зберігати дані поза межами життєвого циклу контейнерів. Це особливо важливо для MySQL, де втрата даних може призвести до серйозних наслідків. Крім того, налаштування мережевих параметрів дозволяє мікросервісам взаємодіяти один з одним через віртуальну мережу Docker, створену спеціально для системи.

На етапі розгортання у продуктивне середовище система використовує Kubernetes-кластери, які забезпечують високий рівень автоматизації та масштабованості. Кожен мікросервіс має окремий об'єкт Deployment, який визначає необхідну кількість реплік (подів) для забезпечення навантаження. Якщо один із подів виходить із ладу, Kubernetes автоматично відновлює його, забезпечуючи безперебійну роботу сервісу. Рисунок 7 — Інтеграції Kubernetes у “Real Estate”.

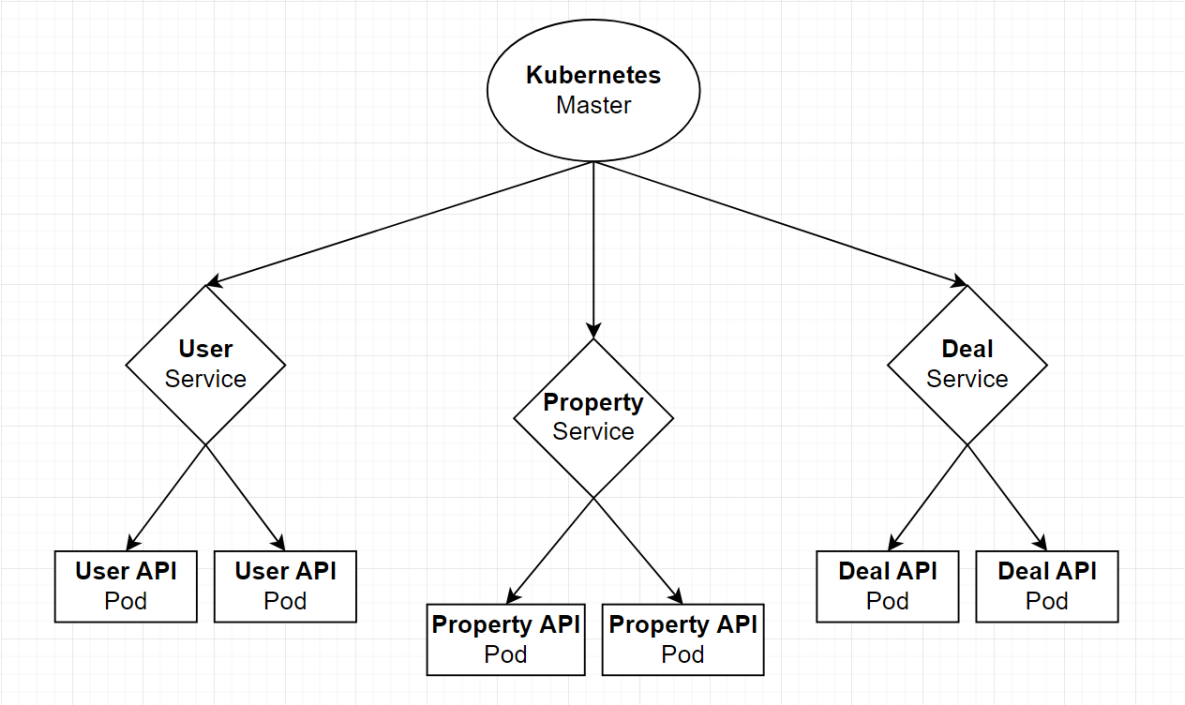


Рисунок 7 — Інтеграції Kubernetes у “Real Estate”

Для маршрутизації запитів між подами використовується об’єкт Service, який забезпечує стабільну точку доступу до мікросервісу, незалежно від його фізичного розташування в кластері. Наприклад, навіть якщо поди розподілені між різними вузлами кластера, Service гарантує, що запити будуть коректно спрямовані. А для підвищення рівня безпеки використовуються Kubernetes Secrets, що дозволяють захищати конфіденційні дані, такі як secret keys для JWT токенів. Це забезпечує додатковий рівень захисту як на етапі розробки, так і у продуктивному середовищі.

Інтеграція Docker та Kubernetes дала змогу досягти високого рівня гнучкості, продуктивності та стабільності системи "Real Estate". Завдяки цим технологіям розробники мають змогу швидко впроваджувати зміни, тестувати нові функції та ефективно масштабувати систему відповідно до потреб бізнесу.

3.5 Висновки до розділу 3

1. Система "Real Estate" призначена для управління об’єктами нерухомості, користувачами та угодами між ними. Вона охоплює ключові аспекти роботи з нерухомістю, включаючи створення, редагування, пошук і відображення об'єктів, а також взаємодію між учасниками угод. Предметна область системи була ретельно опрацьована, щоб забезпечити зручність використання, масштабованість та відповідність потребам сучасного ринку нерухомості.
2. Архітектура додатку побудована на мікросервісному підході із застосуванням CQRS і Domain-Driven Design. Це дозволило створити модульну, гнучку і масштабовану систему, яка забезпечує чітке розмежування відповідальностей між мікросервісами. Backend реалізовано на NestJS з використанням Sequelize ORM і MySQL для збереження даних. Завдяки використанню CQRS бізнес-логіка відділена від логіки взаємодії з даними, що підвищило продуктивність та забезпечило ефективність обробки складних запитів.
3. Функціонал сайту "Real Estate" охоплює всі основні потреби користувачів у сфері управління нерухомістю. Ключові можливості включають авторизацію користувачів, редагування їх профілів, відображення останніх доданих об'єктів нерухомості, пошук і фільтрацію нерухомості за різними критеріями, перегляд детальної інформації про об'єкти. Крім того, користувачі можуть створювати угоди, переглядати свої активні та завершені

угоди, що робить платформу повноцінним інструментом для ефективної взаємодії на ринку нерухомості.

4. Для забезпечення ізольованості середовищ і стандартизації процесу розробки всі мікросервіси були контейнеризовані за допомогою Docker. Docker Compose використовується для організації локальної розробки, а Kubernetes — для розгортання у продуктивне середовище. Завдяки Kubernetes-кластерам забезпечено автоматичне масштабування, ізоляцію сервісів і централізоване управління конфігураціями.

4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ ВПЛИВУ КОНТЕЙНЕРИЗАЦІЇ ТА ОРКЕСТРАЦІЇ НА РОБОТУ СИСТЕМИ "REAL ESTATE"

4.1 Аналіз продуктивності системи "Real Estate" у контейнерному середовищі

Контейнеризація системи "Real Estate" має значний вплив на її продуктивність та ефективність. Для оцінки було проведено апробацію, що дозволило порівняти ключові показники продуктивності до і після впровадження Docker-контейнерів. Основними метриками для аналізу стали час розгортання, використання системних ресурсів (CPU, RAM), а також швидкість запуску окремих мікросервісів.

Увесь аналіз виконувався на операційній системі “Windows 11” версії “24H2”. Контейнери системи "Real Estate" підіймались за допомогою локального кластеру Minikube.

До впровадження контейнеризації розгортання кожного мікросервісу вимагало налаштування середовища вручну, включно з встановленням необхідних залежностей, конфігурацією баз даних тощо. У середньому цей процес займав 25-30 хвилин на кожен мікросервіс, а повна система розгорталася за 1-2 години.

Після впровадження Docker середній час створення та запуску контейнера скоротився до 30 секунд на мікросервіс, а повна система розгортається за 2-3 хвилин, що забезпечує прискорення у 40-60 разів.

Було проведено моніторинг використання CPU та RAM для одного з ключових мікросервісів (Property API) до і після контейнеризації:

Таблиця 6 — Таблиця порівняння продуктивності з контейнеризованим середовищем

Параметр	До контейнеризації	Після контейнеризації	Зміна
Середнє завантаження CPU	45%	38%	-16%
Середнє споживання RAM	512 MB	430 MB	-16%

Контейнеризація дозволила оптимізувати використання ресурсів завдяки ізоляції процесів, що зменшило накладні витрати операційної системи. Зниження завантаження CPU та RAM на **16%** дозволяє ефективніше використовувати обчислювальні ресурси.

Ще одним критичним параметром є час, необхідний для запуску системи або її компонентів після зупинки. До контейнеризації запуск системи займав 5-10 хвилин, оскільки всі залежності доводилося налаштовувати вручну. У середовищі Docker час запуску скоротився до 1-2 хвилин завдяки готовим контейнерам з усіма налаштованими залежностями. Це прискорення у 5-10 разів.

Для перевірки продуктивності системи були проведені навантажувальні тести за допомогою інтсрументу **k6**. Для цього було піднято один сервіс Property API на локальній машині та інший через Docker. Далі постійно відправлялась фіксована кількість запитів на ці мікросервіси (на ендпоінт “/properties/offers”). Рисунок 8 — Графік кількості обробленних запитів до контейнеризації та з нею.

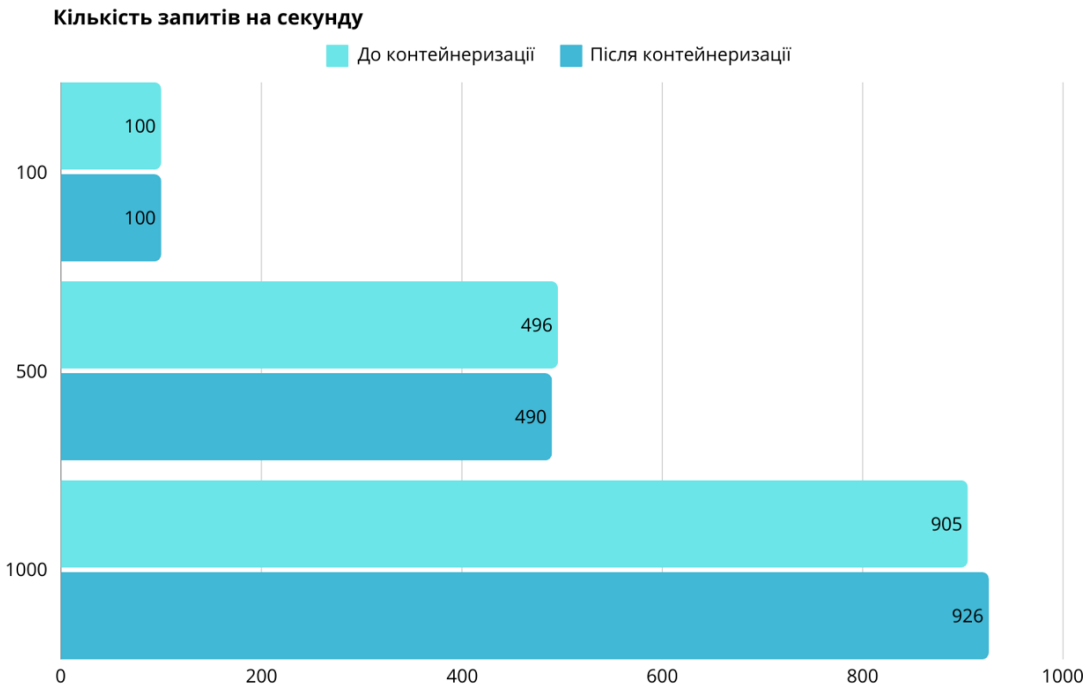


Рисунок 8 — Графік кількості оброблених запитів

Як видно, контейнеризація забезпечила невеличке збільшення продуктивності системи під високим навантаженням на 2-5%, завдяки зменшенню накладних витрат і кращій ізоляції процесів. Хоча у деяких випадках, як на кількості 500 запитів, можна побачити невеличку перевагу перед Docker контейнером, але це більше схоже на похибку у поточному тесті.

Усі ці тести проводились на найбільш складному запиті – пошук нерухомості з великою кількістю фільтрів. Для початку ці тести проводились на машині без контейнеризації. Рисунок 9 — Результати стрес тесту до використання контейнерів.

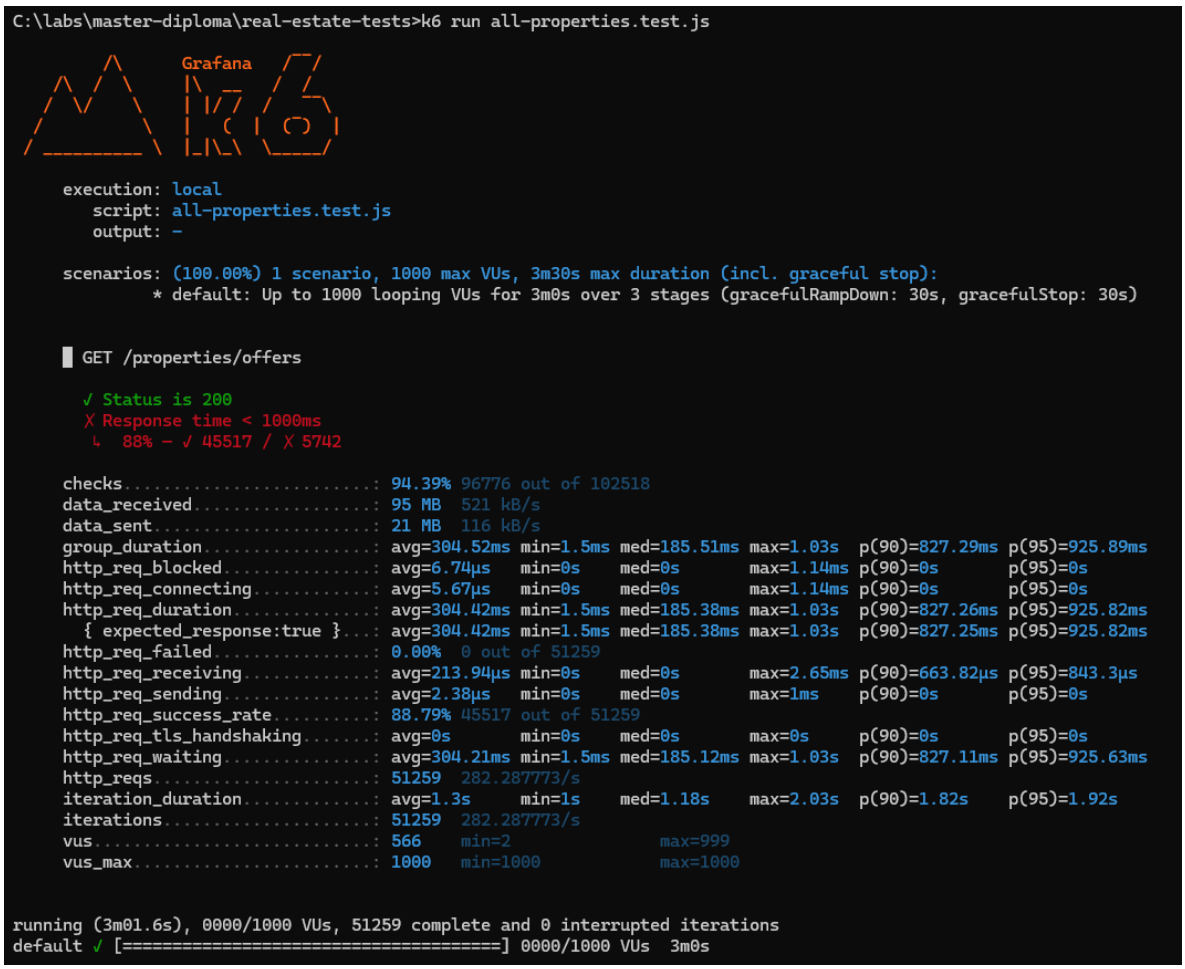


Рисунок 9 — Результати стрес тесту до використання контейнерів

Загальні характеристики:

1. Тривалість виконання: 3 хвилини 1.6 секунди.
2. Максимальна кількість користувачів (VUs): 1000.
3. Ітерацій: 51259 (282.3 ітерації на секунду).
4. Розмір даних: Отримано: 95 MB. Відправлено: 21 MB.

Статистика по запитам GET /properties/offers:

1. Статус 200: Усі запити пройшли успішно
2. Час відповіді:
    a. Середнє: 304.42 мс;
    b. Медіана: 185.38 мс;
    c. Максимум: 1.03 с;
    d. 90-й перцентиль: 827.26 мс;
    e. 95-й перцентиль: 925.82 мс.

3. Час очікування (waiting): середній час очікування відповіді становить 304.21 мс.

Продуктивність:

1. Пропускна здатність: 282.3 запити на секунду.
2. Швидкість відправки та отримання:
   a. Середнє значення часу на відправку: 2.38 мс;
   b. Середнє значення часу на отримання: 213.94 мс.

Відсоток успішних запитів: 88.79% (45517 з 51259), тобто близько 5742 запити перевищили цей поріг, що свідчить про те, що система може мати проблеми із затримками для певної частини запитів.

Далі провівся такий ж самий тест, але вже на машині з контейнеризацією. Рисунок 10 — Результати стрес тесту після використання контейнерів.

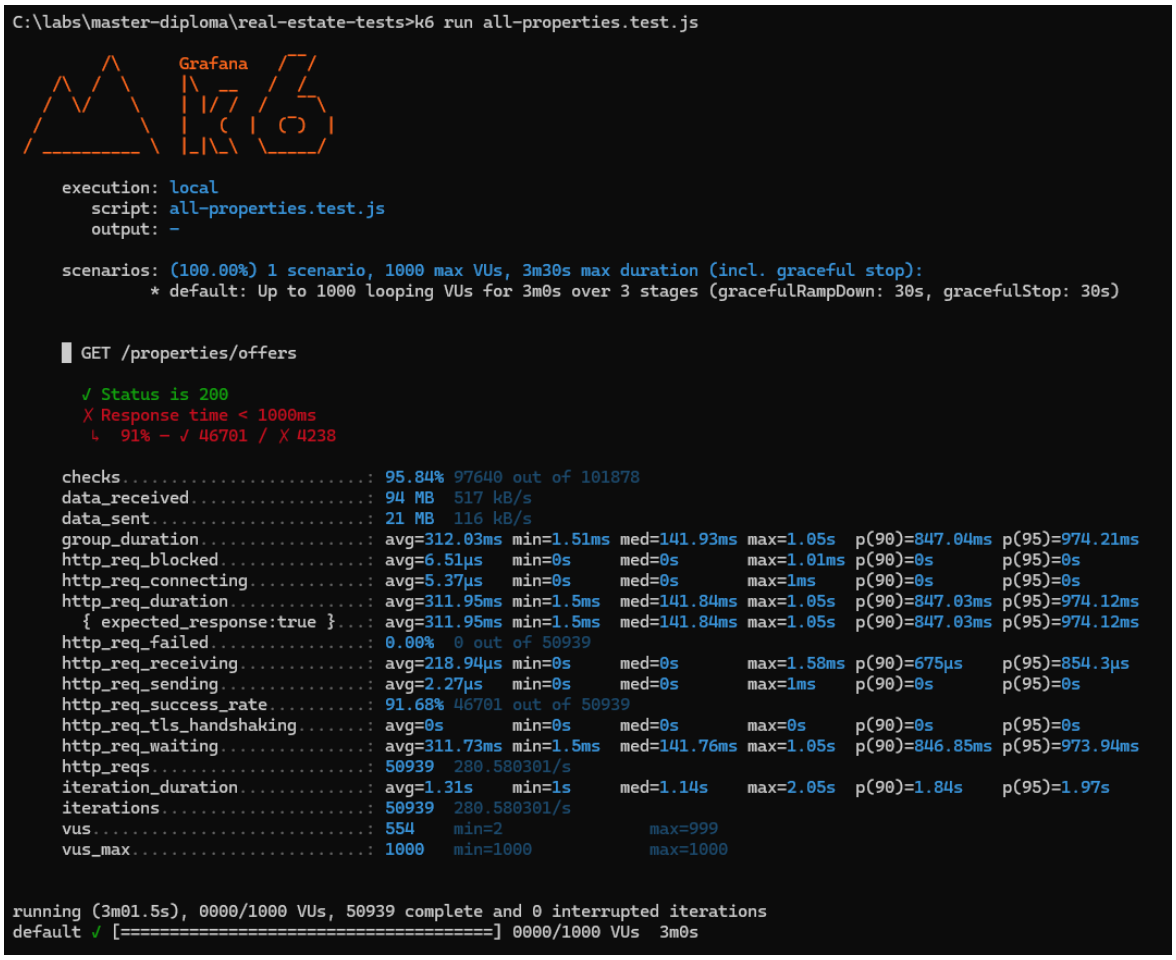


Рисунок 10 — Результати стрес тесту після використання контейнерів

Загальні характеристики:

1. Тривалість виконання: 3 хвилини 1.5 секунди.
2. Максимальна кількість користувачів (VUs): 1000.
3. Ітерацій: 50939 (280.6 ітерацій на секунду).
4. Розмір даних**:** Отримано: 94 MB. Відправлено: 21 MB.

Статистика по запиту GET /properties/offers:

1. Статус 200: 95.84% запитів пройшли успішно (97640 з 101878).
2. Час відповіді:
    a. Середнє: 311.95 мс;
    b. Медіана: 141.84 мс;
    b. Максимум: 1.05 с;
    c. 90-й перцентиль: 847.03 мс;
    d. 95-й перцентиль: 974.12 мс.
3. Час очікування (waiting): середній час очікування відповіді становить 311.73 мс.

Продуктивність:

1. Пропускна здатність: 280.6 запитів на секунду.
2. Швидкість відправки та отримання:
    a. Середнє значення часу на відправку: 2.27 мс;
    b. Середнє значення часу на отримання: 218.94 мс.

Відсоток успішних запитів: 91.68% (46701 з 50939), що є хорошим показником, у порівнянні з попереднім тестом, але 4238 запитів перевищили цей поріг. Це свідчить про помірні проблеми з деякими запитами, які мають більші затримки.

4.2 Дослідження масштабованості системи за допомогою Kubernetes

Масштабованість є ключовим аспектом для забезпечення стійкості та ефективності системи "Real Estate" при збільшенні навантаження. Впровадження Kubernetes як оркестратора контейнерів дозволило системі

адаптуватися до змін у робочому середовищі, автоматично збільшуючи або зменшуючи кількість реплік (подів) мікросервісів залежно від поточного навантаження.

Для оцінки масштабованості системи було проведено навантажувальне тестування з використанням інструменту k6. Тестування виконувалося за таких умов:

1. Початкове навантаження: 50 запитів на секунду;
2. Максимальне навантаження: 500 запитів на секунду;
3. Тривалість тесту: 10 хвилин;
4. Масштабування: Автоматичне (HPA — Horizontal Pod Autoscaler).
5. Бази даних для обох систем були заповнені даними про різні нерухомості (більше 10 тисяч записів).
6. Ендпоінт “/properties/offers” використовувався майже з усіма фільтрами.

До впровадження Kubernetes кожен мікросервіс працював на виділеному сервері без можливості автоматичного масштабування. При навантаженні понад 200 запитів на секунду система починала демонструвати значні затримки, а при досягненні 300 запитів сервери деяких мікросервісів переставали відповідати. Середній час відповіді зростав із 200 мс до 1100 мс. Рисунок 11 — Результати масштабованості системи “Real Estate”.

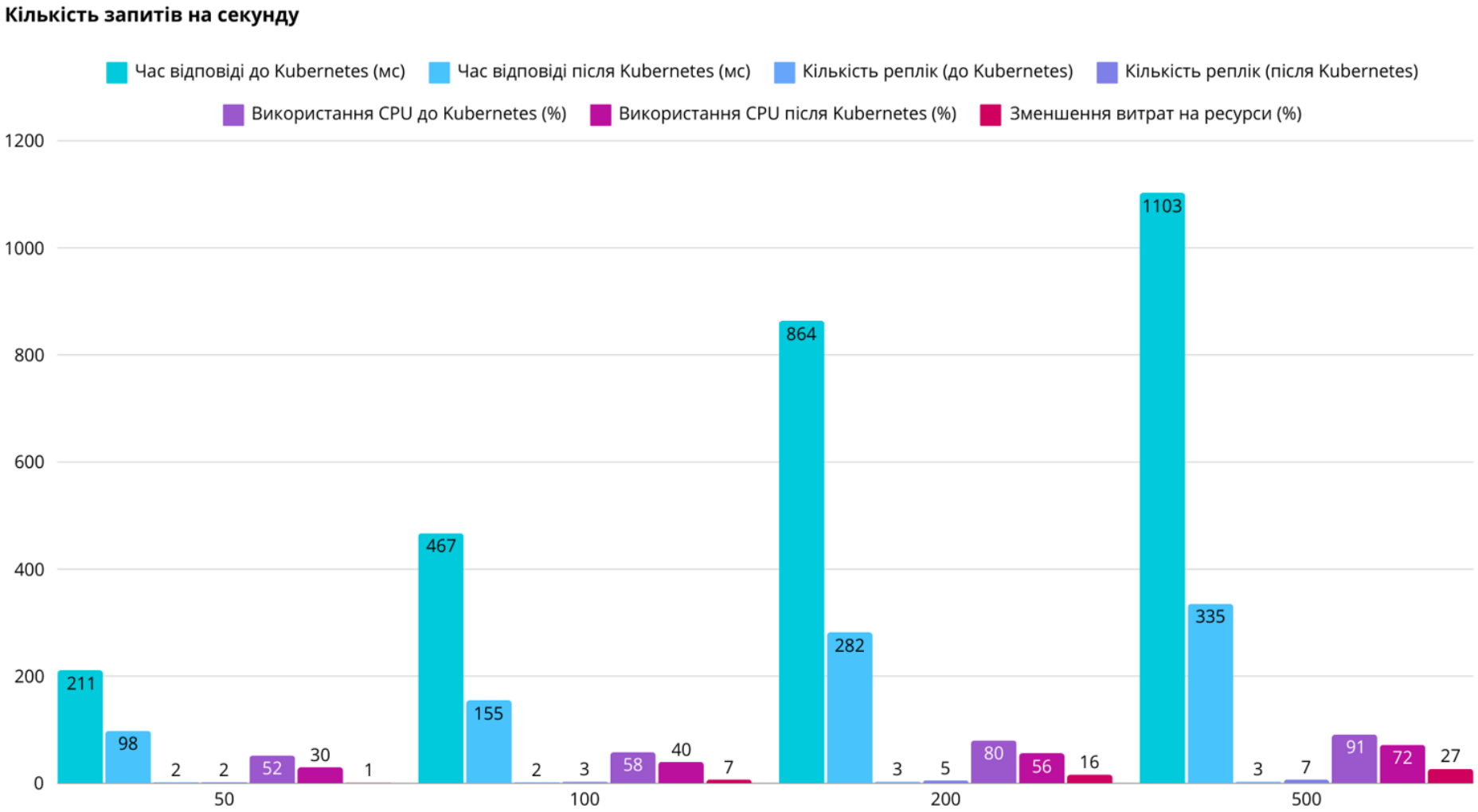


Рисунок 11 — Результати масштабованості системи “Real Estate”

З використанням Kubernetes та налаштуванням Horizontal Pod Autoscaler система виявила значно вищу стійкість. Основні результати:

1. Час відповіді: При навантаженні 50-500 запитів на секунду середній час відповіді залишався стабільним на рівні 100-150 мс. Навіть при максимальному навантаженні в 1000 запитів час відповіді не перевищував 350 мс.

2. Кількість реплік: Система автоматично масштабувала кількість реплік мікросервісів з початкових 2 до 10 при високому навантаженні. Зменшення навантаження призводило до автоматичного скорочення реплік до 2-3.

3. Використання ресурсів: Завдяки налаштуванню CPU та RAM лімітів, ресурси кластера розподілялися оптимально. У пікові моменти середнє завантаження CPU для кожного поду становило 70-80%, що забезпечувало стабільну роботу системи без перевантаження.

Одним із важливих аспектів використання Kubernetes є зменшення витрат на інфраструктуру. До впровадження Kubernetes, для забезпечення резерву на пікові навантаження потрібно було підтримувати постійно увімкнені сервери з надмірними ресурсами. Завдяки динамічному масштабуванню Kubernetes вдалося знизити середнє споживання ресурсів CPU на 35%.

Результати дослідження демонструють, що використання Kubernetes забезпечило значне покращення масштабованості та ефективності системи "Real Estate", що є критично важливим для динамічних умов роботи.

4.3 Оцінка надійності та відмовостійкості системи

Після впровадження контейнеризації за допомогою Docker і оркестрації з Kubernetes, система "Real Estate" продемонструвала значне покращення в аспектах надійності та відмовостійкості, зокрема в її ключових мікросервісах, таких як User API, Property API та Deal API. Раніше, при роботі в традиційному середовищі, кожен з цих мікросервісів мав окремі конфігурації, налаштування

та залежності, що ускладнювало процес оновлення і масштабування. Проблеми могли виникати навіть через незначні зміни в конфігураціях або версіях бібліотек, що викликало додаткові труднощі при підтримці стабільності системи.

До впровадження контейнеризації та Kubernetes, система працювала на основі традиційного середовища, де кожен мікросервіс мав свою конфігурацію та залежності, що часто ставало причиною помилок при оновленнях або в разі збоїв в інфраструктурі. У разі зупинки сервісу або проблем з ресурсами, адміністратор змушений був вручну перезапускати сервіси, що призводило до часу простою та впливало на загальну стабільність роботи системи. Завдяки контейнеризації та використанню Docker, кожен з мікросервісів "Real Estate", таких як User API для управління користувачами, Property API для обробки інформації про нерухомість, а також Deal API для управління угодами, був упакований у власний контейнер. Ці контейнери містять усі необхідні залежності та конфігурації, що значно зменшує ймовірність виникнення помилок, пов'язаних з неправильними налаштуваннями або несумісними версіями програмного забезпечення. Це забезпечило стабільність і ефективність роботи кожного з мікросервісів, а також забезпечило їх сумісність на всіх етапах розгортання.

Важливим етапом у забезпеченні надійності стало використання Kubernetes для оркестрації контейнерів. Однією з основних переваг Kubernetes є його здатність автоматично відновлювати сервіси після збоїв. Якщо один з мікросервісів виходить з ладу, Kubernetes автоматично створює нову репліку цього сервісу і направляє до нього трафік, тим самим забезпечуючи безперервну роботу системи. Це дозволяє системі "Real Estate" працювати стабільно навіть при великих навантаженнях і при частих збоях окремих компонентів. Для підвищення надійності ймовірність відмов була знижена завдяки механізмам горизонтального масштабування, які дозволяють Kubernetes збільшувати або зменшувати кількість реплік мікросервісів залежно від поточного навантаження. За допомогою Horizontal Pod Autoscaling

система може динамічно реагувати на зміни в інтенсивності запитів і автоматично коригувати кількість подів мікросервісів. Це дозволяє підтримувати оптимальний рівень продуктивності і знижує ймовірність перевантаження будь-якого з компонентів. Ще одним важливим аспектом, який покращив відмовостійкість, є персистентне зберігання даних. Kubernetes дозволяє використовувати Persistent Volumes та Persistent Volume Claims для зберігання критично важливої інформації, навіть у разі зупинки або переміщення контейнерів на інші вузли в кластері. Завдяки цьому важливі дані, такі як дані про користувачів або інформація про угоди, залишаються доступними після перезавантаження контейнерів або навіть у разі відмови одного з серверів.

Протягом апробації системи “Real Estate” за допомогою Kubernetes було проведено серію стрес-тестів, що симулюють реальні умови роботи та збої. Під час таких тестів система продемонструвала високу відмовостійкість, здатність до самовідновлення та мінімальний час простою. Наприклад, після створення штучного збою в Deal API мікросервісу, Kubernetes автоматично ініціював відновлення без помітного впливу на користувацький досвід.

4.4 Дослідження простоти управління середовищем розробки

Після впровадження контейнеризації та використання Kubernetes у системі "Real Estate" значно змінилася структура управління середовищем розробки. До цього, кожен розробник стикався з необхідністю вручну налаштовувати середовище для кожного мікросервісу, що включало встановлення відповідних бібліотек, налаштування баз даних, змінних оточення та залежностей, а також перевірку їх сумісності. Цей процес був трудомістким і часто призводив до помилок через різницю у налаштуваннях між локальними середовищами різних розробників. Однак після впровадження Docker і Kubernetes процес управління середовищем значно спростився, що сприяло підвищенню ефективності роботи команди.

4.4.1 Переваги використання Docker для локального середовища

Однією з основних переваг використання Docker стало стандартизоване середовище для кожного мікросервісу. Усі необхідні компоненти для кожного з мікросервісів, таких як User API, Property API, Deal API, були включені у відповідні Docker-образи. Кожен мікросервіс мав свій власний Dockerfile, що дозволяло відтворювати однакові умови для всіх розробників. Завдяки цьому розробники могли без проблем працювати у однаковому середовищі, не турбуючись про встановлення чи конфігурацію окремих залежностей.

Процес запуску та тестування мікросервісів став значно простішим. Замість того, щоб налаштовувати середовище вручну, достатньо було виконати кілька команд для запуску контейнера з потрібними налаштуваннями. Зокрема, команда docker-compose up дозволяла підняти всю систему, що включала не тільки мікросервіси, але й бази даних, кешування та інші необхідні сервіси, що використовуються "Real Estate". Це значно знижувало ймовірність помилок, оскільки всі залежності були чітко визначені у конфігураційному файлі Docker Compose, і будь-які зміни, необхідні для коректної роботи, могли бути внесені централізовано.

До впровадження Docker, для того щоб розробити нову функціональність або провести тестування, розробникам часто доводилося витрачати час на налаштування середовища і вирішення проблем з несумісністю версій бібліотек чи налаштувань. Після впровадження контейнерів цей процес став значно простішим, оскільки всі залежності і налаштування були чітко визначені і не залежали від локальних середовищ.

4.4.2 Простіше управління завдяки Kubernetes

Використання Kubernetes вносить додаткову автоматизацію в управління середовищем розробки та тестування. Замість того щоб вручну налаштовувати сервери або машини для кожного нового етапу розробки чи тестування, з Kubernetes команда може оркеструвати запуски та управління контейнерами за допомогою простих конфігураційних файлів. Завдяки

Kubernetes також з’явилася можливість автоматизувати оновлення мікросервісів, що значно спростило керування версіями і випусками. Раніше кожне оновлення чи виправлення вносилося вручну, що вимагало ретельного тестування та перевірки. Тепер Kubernetes дозволяє автоматично оновлювати мікросервіси API та контролювати сумісність з іншими компонентами системи. Під час випуску нової версії мікросервісу, Kubernetes може проводити оновлення без зупинки сервісів, що гарантує високу доступність для користувачів.

4.4.3 Масштабованість процесу розробки

Завдяки Kubernetes, команда "Real Estate" отримала інструменти для масштабування розробки. Наприклад, кожен розробник тепер може працювати над окремими частинами системи, не турбуючись про конфлікти з іншими частинами середовища. Якщо необхідно масштабувати певний мікросервіс, Kubernetes дозволяє легко додавати нові репліки без впливу на інші частини системи. Це не тільки спрощує тестування нових функціональностей, але й допомагає працювати паралельно, без необхідності тривалого налаштування середовищ. Для кожного розробника та тестера Kubernetes надає можливість ізолювати свої зміни в окремих просторах імен. Це дозволяє одночасно розробляти нові функції без ризику збоїв або впливу на роботу інших частин системи.

4.5 Висновки до розділу 4

1. Контейнеризація забезпечила підвищення продуктивності системи "Real Estate" завдяки ізольованому виконанню сервісів та оптимізації використання ресурсів. Завдяки використанню контейнерів зменшились накладні витрати на запуск окремих компонентів системи, що сприяло швидшій обробці запитів та стабільній роботі сервісів навіть під високим навантаженням.

2. Дослідження показало, що Kubernetes значно покращує можливості масштабування системи "Real Estate". Завдяки функціональності автоматичного масштабування, система може ефективно адаптуватися до змін у навантаженні, забезпечуючи стабільну роботу навіть у пікові періоди. Також було відзначено, що горизонтальне масштабування допомогло оптимізувати витрати на обчислювальні ресурси.

3. Використання Kubernetes дозволило підвищити надійність і відмовостійкість системи. Завдяки вбудованим механізмам автоматичного перезапуску контейнерів, балансування навантаження та можливості відновлення у випадку відмови вузлів система забезпечує безперебійну роботу. Це є важливим фактором для підтримки високого рівня доступності сервісів у середовищі "Real Estate".

4. Контейнеризація та оркестрація спростили управління середовищем розробки. Створення уніфікованих контейнерних образів забезпечило однакові умови для розробки, тестування та продакшн середовища.

ВИСНОВКИ

1. Результати аналізу доступних технологій оркестрації контейнерів та методів автоматизації управління додатками показали, що Kubernetes є найефективнішою платформою для забезпечення масштабованості, надійності та автоматизації процесів розгортання контейнерних додатків у кластерному середовищі.

2. Проведено порівняльний аналіз існуючих інструментів для оркестрації контейнерів, зокрема Kubernetes та його компонентів. Визначено переваги та недоліки використання цієї платформи в різних сценаріях, а також вивчено ключові аспекти автоматизації управління контейнерами.

3. Було розроблено архітектуру системи для ефективної оркестрації контейнерних додатків, визначено основні компоненти Kubernetes, а також функціональні вимоги до інтеграції контейнерних додатків у кластерну інфраструктуру.

4. В результаті виконання кваліфікаційної роботи було успішно реалізовано Kubernetes конфігурування для управління контейнерними додатками, що забезпечує автоматичне масштабування, високу надійність і можливість оперативного відновлення при збої.

5. Реалізоване рішення для управління контейнерами ефективно справляється з вимогами сучасних контейнеризованих додатків і може бути використана в реальних умовах для забезпечення стабільності та масштабованості бізнес-процесів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kubernetes for Developers. Joe Heck. 9 Травня, 2018. P. 43.
2. What is container orchestration?. URL: https://www.redhat.com/en/topics/containers/what-is-container-orchestration (дата звернення: 24.03.2024 р.).
3. Revolutionizing your Backend Plan with Containerization. URL: https://fastercapital.com/topics/revolutionizing-your-backend-plan-with-containerization.html (дата звернення: 24.03.2024 р.).
4. What Are Containerized Microservices? URL: https://blog.dreamfactory.com/what-are-containerized-microservices/ (дата звернення: 25.03.2024 р.).
5. What are Microservices and Containers? URL: https://avinetworks.com/what-are-microservices-and-containers/ (дата звернення: 03.05.2024 р.).
6. Building Microservices: Designing Fine-Grained Systems. Sam Newman. P. 108.
7. Patterns of Distributed Systems. Unmesh Joshi. P. 54.
8. Microservices Patterns: With examples in Java. Chris Richardson. P. 171.
9. Domain-Driven Design: Tackling Complexity in the Heart of Software. Eric Evans. P. 255.
10. Implementing Domain-Driven Design. Eric Evans; P. 301.
11. The Kubernetes Book. Nigel Poulton and Pushkar Joglekar. P. 70.
12. Container Security: Fundamental Technology Concepts that Protect Containerized Applications. Liz Rice. P. 46.
13. Mastering Docker, Fourth Edition: Enhance Your Containerization and DevOps Skills to Deliver Production-Ready Applications. Russ McKendrick. P. 30.
14. The Docker Book. James Turnbull. P. 66.
15. Containerisation. J. R. Whittaker. P. 20.
16. The Book of Kubernetes: A Complete Guide to Container Orchestration. Alan Hohn. P. 79.

17. Docker Deep Dive: Zero to Docker in a single book (Mastering Containers 1). Nigel Poulton (Author). P. 215.

18. Getting Started with Docker. Nigel Poulton. P. 30.

19. Kubernetes and Docker, An Enterprise Guide: Effectively Containerize Applications, Integrate Enterprise Systems, and Scale Applications in Your Enterprise. Marc Boorshtein and Scott Surovich. P. 111.

20. Nest.js: A Progressive Node.js Framework. P. Greg Magolan, Patrick Housley, Adrien de Peretti. P. 15.

21. Real-World Next.js. Michele Riva. P. 61.

22. NestJS Build a RESTful CRUD API. Kerry Ritter. P. 45.

23. What is Next.js? A look at the popular JavaScript framework. URL: https://kinsta.com/knowledgebase/next-js/ (дата звернення 01.05.24).

24. What is server-side rendering: definition, benefits and risks. URL: https://solutionshub.epam.com/blog/post/what-is-server-side-rendering (дата звернення 01.05.24).

25. Scaling Applications with Docker: Load Balancing and Service Discovery. URL: https://medium.com/@navneetskahlon/scaling-applications-with-docker-load-balancing-and-service-discovery-163ef4c78c10 (дата звернення 03.05.24).

26. What are Kubernetes Pods? URL: https://www.vmware.com/topics/glossary/content/kubernetes-pods.html (дата звернення 03.05.24).

27. Kubernetes Service: Examples, Basic Usage, and Troubleshooting. URL: https://komodor.com/learn/kubernetes-service-examples-basic-usage-and-troubleshooting/ (дата звернення 04.05.24).

28. Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications. Bilgin Ibryam and Roland Hus. P. 49.

29. Kubernetes Cookbook: Building Cloud Native Applications. Michael Hausenblas and S. Goasguen. P. 34.

30. Kubernetes in Action, Second Edition. Kevin Conner and Marko Luksa. P. 20.

31. Воробець Д. С., Заяц В. І. Використання контейнеризації та оркестрації у веброзробці Інженерні інновації та розбудова національної економіки : матеріали І Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю. М. Потебні (09–10 травня 2024 року, м. Запоріжжя) / наук. ред. Н. Г. Метеленко ; Інженерний навчально-науковий інститут ім. Ю. М. Потебні Запорізького національного університету. – Одеса : Видавничий дім «Гельветика», 2024. С. 606-608.

32. Воробець Дмитро магістрант 1 курсу ІННІ ім. Ю. М. Потебні Наук. кер.: канд. техн. наук, доц. Заяц В.І. ВПЛИВ КОНТЕЙНЕРИЗАЦІЇ ТА ОРКЕСТРАЦІЇ НА ВЕБ-РОЗРОБКУ. ЗБІРНИК наукових праць студентів, аспірантів, докторантів і молодих вчених «МОЛОДА НАУКА-2024», Запоріжжя : Запорізький національний університет, 2024. С.176-177.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Воробець Дмитро Сергійович , студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz19bd-05@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Вплив контейнеризації та оркестрації на веб-розробку** » відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 12.12.2024 Підпис _____ Воробець Дмитро Сергійович
(студент)

Дата 12. 12. 2024 Підпис _____ Заяц Валерій Іванович
(науковий керівник)