Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Трофименко Дениса Олександровича*
(ПІБ)

академічної групи *122М-19-1*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

на тему: *Розробка та оптимізація методів та алгоритмів автоматичного управління розгорнутими контейнерізованими веб-додатками та сервісами*

____________________ *Д.О. Трофименко*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	Проф. Мещеряков Л.І.			
економічний	Доц. Касьяненко Л.В.			

Рецензент				

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп’ютерних систем
(повна назва)

_______________ І.М. Удовик
(підпис) (прізвище, ініціали)

« » ____________ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *122 Комп’ютерні науки*
(код і назва спеціальності)

студенту *122М-19-1* Трофименко Денису Олександровичу
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Розробка та оптимізація методів та алгоритмів автоматичного управління розгорнутими контейнерізованими веб-додатками та сервісами*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888 -с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об’єкт досліджень – процеси автоматизації налаштування роботи розгорнутих веб-додатків.

Предмет досліджень – моделі та методи автоматичного управління контейнерізованими додатками та їх сервісами.

Мета роботи – підвищення ефективності автоматичного управління розгорнутими веб-додатками, сервісами та їх навантаженнями у середі Kubernetes.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні методів автоматичного масштабування та управління розгорнутими веб-додатками.

Практична цінність результатів полягає у тому, що запропоновані в роботі методи та алгоритми підвищують ефективність автоматизації управління та роботу контейнерізованих розгорнутих веб-додатків та сервісів.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	01.09.2020-30.09.2020
Побудова моделі методів та алгоритмів автоматизації засобами Java та пошук оптимальних шляхів вирішення задачі оптимізації	01.10.2020-31.10.2020
Створення на базі існуючих алгоритмів автоматизації управління розгорнутими веб-додатками методів для вирішення задачі оптимізації автоматичного управління веб-додатками	01.11.2020-07.12.2020

Завдання видав ______________ *Мещеряков Л.І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання ______________ *Трофименко Д.О.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *01.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *10.12.2020*

РЕФЕРАТ

Пояснювальна записка: 82 с., 18 рис., 3 дод., 60 джерел**.**

Об'єкт дослідження: процеси автоматизації налаштування роботи розгорнутих контейнерізованих веб-додатків та сервісів.

Предмет дослідження: моделі та методи управління контейнерізованими розгорнутими веб-додатками та їх сервісами.

Мета магістерської роботи: підвищення ефективності управління розгорнутими веб-додатками, сервісами та їх навантаженнями у середі Kubernetes.

Методи дослідження. Для виконання поставлених завдань були використані методи створення операторів та користувацьких ресурсів, теорії баз даних, методи пакування та контейнерізації додатків.

Наукова новизна результатів дипломної роботи полягає в удосконаленні методів автоматичного масштабування та управління розгорнутими веб-додатками та сервісами.

Практичне значення роботи. Результати роботи, отримані в ході дослідження, можуть застосовуватися під час автоматизації управління розгорнутими веб-додатками та сервісами.

У розділі «Економіка» проведено розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПО і тривалості його розробки.

Список ключових слів: Kubernetes, Docker, DockerHub, image-образ, контейнер, управління автоматизацією, контейнеризація веб-додатків, под, деплоймент, кластер, нода.

ABSTRACT

Explanatory note: 82 p., 16 fig., 3 applications, 60 sources.

Object of research: Processes of automating the configuration of deployed containerized web applications and services.

Subject of research: Models and methods of managing containerized deployed web applications and services.

Purpose of Master's thesis: Increase the efficiency of managing deployed web applications, services, their loads in the Kubernetes environment.

Research methods. To perform the tasks, we used methods of creating operators and custom resources, database theory, methods of packaging and containerization applications.

Originality of research consists in the improvement of the methods of automatic scaling and managing deployed web applications and services.

Practical value of the results the results of the research can be used to automate the management of deployed containerized web applications and services.

In the Economics section we calculated the complexity of software development and the costs of software development, the duration of the actual development are calculated.

Keywords: Kubernetes, Docker, DockerHub, image, container, managing the automatization, containerization of web applications, pod, deployment, service, cluster, node.

ЗМІСТ

ВСТУП…………………………………………………………………….. 9

РОЗДІЛ 1. АВТОМАТИЗАЦІЯ РОБОТИ РОЗГОРНУТИХ ВЕБ-ДОДАТКІВ……………………………….……………………. 12

1.1. Основні відомості з автоматизації роботи розгорнутих веб-додатків….. 12

1.2. Автоматизація управління контейнерізованими веб-додатками за допомогою платформи Kubernetes.. 13

1.2.1 Види розгортання веб-додатків.. 13

1.2.2 Структура платформи Kubernetes………….. 17

1.3. Основні компоненти платформи та їх призначення...…….............. 21

1.3.1. Поди……………………………………………...………….. 21

1.3.2. Сервіси та деплойменти. Керування подами…………………...…. 26

1.4. Висновки до першого розділу……………………………………… 31

РОЗДІЛ 2. КОНТРОЛЕРИ, ОПЕРАТОРИ ТА ВЛАСНІ РЕСУРСИ НА ПЛАТФОРМІ K8s. ОСНОВНІ МОЖЛИВОСТІ АВТОМАТИЗАЦІЇ УПРАВЛІННЯ ВЕБ-ДОДАТКАМИ.. 33

2.1. Основні функції K8s для автоматизації роботи контейнерізованих веб-додатків.. 33

2.2. Роль контролерів у процесах автоматизації та управління додатками на платформі K8s……………………………………… 35

2.3. Розширення платформи та розробка власних ресурсів…………… 38

2.4. Висновки до другого розділу……………………………………….. 43

РОЗДІЛ 3. РОЗРОБКА ТА ОПТИМІЗАЦІЯ АВТОМАТИЧНОГО УПРАВЛІННЯ РОЗГОРНУТИМИ ВЕБ-ДОДАТКАМИ.............................. 45

3.1. Шаблон автоматизації оператор…………………………………….. 45

3.2. Розробка власного шаблону автоматизації роботи веб-додатків та власного ресурсу………………………………………………… 46

3.1. Висновки до третього розділу...…………………………………….. 57

РОЗДІЛ 4. ЕКОНОМІЧНИЙ РОЗДІЛ.. 59

4.1. Визначення трудомісткості розробки програмного забезпечення... 59

4.2 Витрати на створення програмного забезпечення…………………. 63

4.3 Маркетингові дослідження……………………………………….... 64

4.4 Економічна ефективність…………………………………………….. 66

ВИСНОВКИ………………………………………………………………... 68

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ………………………………. 71

Додаток А. ЛІСТИНГ ПРОГРАМИ……………………………………. 76

Додаток Б. ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ…………… 81

Додаток В. ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ…………... 82

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

K8s – Kubernetes

YAML – Yet Another Markup Language

CRD – Custom Resource Definition

ОС – Операційна система

API – Application Programming Interface

JSON – JavaScript Object Notation

REST – Representational State Transfer

CRUD – Create Read Update Delete Operations

ВСТУП

Актуальність дослідження. Однією з найбільших проблем для підприємств, які застосовують практики DevOps та хмарні можливості, є підтримка загальних та послідовних середовищ протягом усього життєвого циклу додатку.

Контейнери вирішили проблему переносимості додатків, упакувавши всі необхідні залежності в images, і таким чином підтримують узгодженість між хмарними платформами та архітектурою мікросервісів. Kubernetes - це фактичний стандарт для організації та розгортання контейнерів. Користувачі IT та сфери бізнесу можуть зосередити свої зусилля на розробці додатків, а не на інфраструктурі, використовуючи контейнери та Kubernetes.

Kubernetes дозволяє користувачам вибрати найкраще місце для запуску програми, виходячи з бізнес-потреб. Для деяких додатків визначальним фактором буде масштаб та охоплення публічної хмари. Для інших такі фактори, як локалізація даних, безпека чи інші проблеми вимагають локального розгортання.

Поточні рішення можуть бути складними, що змушує команди з’єднувати усі частини разом за рахунок витрат часу та грошей. Це може призвести до зменшення вибору, змушуючи користувачів вибирати між локальними та загальнодоступними хмарними провайдерами. Рішення щодо управління контейнерами може подолати ці виклики.

Разом з тим, незважаючи на великі обсяги можливостей у області автоматизації управління розгорнутими веб-додатками, необхідно відзначити недолік методів та алгоритмів, присвячених сучасним методам автоматизації управління, які поєднували б у собі демонстрацію переваг окремих методів та власних шаблонів автоматизації при вирішенні конкретних завдань і перевірку їх ефективності на сучасному обладнанні. У даній роботі ми плануємо заповнити цю прогалину.

Мета дослідження полягає у підвищенні ефективності управління розгорнутими контейнерізованими веб-додатками та сервісами у середі Kubernetes.

Завдання дослідження. Для досягнення поставленої мети в роботі сформульовані і вирішені такі завдання:

1. Викласти принципи контейнеризації та розгортання додатків;
2. Визначити основні можливості та функції платформи K8s;
3. Дослідити особливості автоматизації управління розгорнутими додатками;
4. Вивчити підходи до автоматизації у рамках додатків Docker та K8s;
5. Проаналізувати існуючі інструменти автоматизації управління розгорнутими додатками, які доступні на платформі K8s;
6. Розробити оператори-контролери та шаблони власних ресурсів і інтегрувати їх з існуючою платформою.

Об'єкт дослідження: процеси автоматизації налаштування роботи розгорнутих веб-додатків.

Предмет дослідження: моделі та методи управління контейнерізованими розгорнутими додатками та їх сервісами.

Методи дослідження. Для виконання поставлених завдань були використані методи розробки операторів та ресурсів, теорії баз даних, методи пакування та контейнерізації додатків.

Наукова новизна результатів дипломної роботи полягає в удосконаленні методів автоматичного масштабування та управління розгорнутими веб-додатками та сервісами.

Практичне значення роботи. Результати роботи, отримані в ході дослідження, можуть застосовуватися під час автоматизації управління розгорнутими веб-додатками та сервісами.

Особистий внесок автора:

1. Наукові результати роботи отримані автором самостійно.
2. Вибір методів досліджень і технологій реалізації;

3. Реалізація алгоритмів та методів автоматичного управління розгорнутими веб-додатками;

4. Розробка теоретичної частини роботи, в якій досліджені і знання про існуючі методи та алгоритми автоматизації і досягнення найкращих результатів по взаємодії з власними ресурсами та шаблонами;

5. Оцінка отриманих результатів.

Структура і обсяг роботи. Робота складається з вступу, трьох розділів і висновків. Містить 80 сторінки, в тому числі 82 сторінок тексту основної частини з 18 рисунками, списку використаних джерел з 60 найменуваннями на 3 сторінках, 3 додатка на 9 сторінках.

РОЗДІЛ 1
АВТОМАТИЗАЦІЯ РОБОТИ РОЗГОРНУТИХ ВЕБ-ДОДАТКІВ

1.1. Основні відомості з автоматизації роботи розгорнутих веб-додатків

До недавніх пір, інфраструктура та розробка веб-додатків були неминуче переплетені. По мірі зростання складності розробки додатків, розробники стикалися з прірвою, між розробкою та операціями, пов'язаними з проблемою доставки додатків на сервера. Проблеми виникали у коректному функціонуванні додатків без належного управління конфігурацією. Конфігурації додатків часто суперечили власним цілям розробників щодо безпеки, масштабованості та доступності. Проблема гнучкої доставки полягала у тому, що конфігурація додатків та конфігурація інфраструктури повинні бути досягнуті спільно, але мають чітку визначену власність та чітке розділення ролей. На додачу до цього був класичний випадок синдрому "це працює для мене", коли розробники часто скаржились на те, що їх програмне забезпечення справно працювало у власному налаштованому середовищі розробки, але поводилося по-різному, коли його розгортали на налаштованому IT-середовищі.

Щоб вирішити ці проблеми, галузь звернулася до принципу ідемпотентності. У випадку управління конфігурацією, таким кінцевим результатом буде бажана конфігурація середовища програми. Коли навколишнє середовище відхиляється від бажаної конфігурації, є можливість використати цей принцип, щоб забезпечити виправлення дрейфу та повернення середовища до бажаного стану. Хоча теорія ідемпотентності вирішувала деякі питання, вона не вирішувала основні завдання. Розробники мали змогу дізнатися, коли щось змінилося, і лише оновлювати те, що потрібно було оновити. Цю складність стало можливим уникнути з появою контейнерів. Замість того, щоб міняти щось на місці, розробники могли б зараз розгорнути незмінні повністю налаштовані образи контейнерів і лише замінити старі контейнери новими оновленими. Таким чином, фокус перемістився з ідемпотентності на інший важливий

принцип: незмінність. Отже, як тільки програма була упакована в образ контейнера разом із залежностями та конфігураціями, з неї можна було створити будь-яку кількість однакових контейнерів.

З популярністю контейнерів, K8s став найпопулярнішою платформою для організації контейнерів. Додатки, упаковані як контейнери, можуть бути розгорнуті в будь-якому середовищі K8s, що працює де завгодно, і що програма буде поводитися однаково завдяки незмінності. Це, здавалося, поклало кінець проблемам конфігурації, згаданим раніше, оскільки відбулося фундаментальне зрушення з чітким розділенням проблем між операційними системами середовища виконання та розгортанням додатків. Тож розробники розгортанням додатків могли зосередитись на таких речах, як кластерна інфраструктура, управління пропускною спроможністю, моніторинг інфраструктури, аварійне відновлення на рівні кластера, мережева безпека, надмірність зберігання даних тощо. З іншого боку, розробники додатків зосередились на створенні image контейнерів, написанні скриптів (Kubernetes маніфестує YAML) для розгортання та конфігурації додатків. Фактична інфраструктура стала вже не так важлива для доставки та розгортання, оскільки її гарно абстрагували за допомогою K8s.

1.2. Автоматизація управління контейнерізованими веб-додатками за допомогою платформи Kubernetes

1.2.1. Види розгортання веб-додатків

Kubernetes - це портативна розширювана платформа з відкритим вихідним кодом для управління контейнерізованими робочими навантаженнями та сервісами, що спрощує як декларативне налаштування, так і автоматизацію. У платформи є велика, швидко розвинута екосистема. Сервіси, підтримка та інструменти Kubernetes широко доступні. Google відкрив вихідний код Kubernetes в 2014 році. Kubernetes засновується на десятилітньому досвіді роботи Google із масштабними робочими навантаженнями, у співпраці з найкращими в своєму класі ідеями та практичними повідомленнями. Щоб краще

зрозуміти потужність та можливості Kuberntes, ми роздивилися усі види розгортання додатків, що існували.

Традиційна ера розгортання: раніше організації запускали додатки на фізичних серверах. Не було ніякого способу визначити граничні ресурси для додатків на фізичному сервері, і це викликало проблеми з розподілом ресурсів. Наприклад, якщо кілька додатків виконуються на фізичному сервері, це може бути випадком, коли один додаток буде займати більшу частину ресурсів, а в результаті інші додатки будуть працювати гірше. Вирішенням цієї проблеми був старт кожного додатку на іншому фізичному сервері. Це не було масштабовано, оскільки ресурси використовувались не повністю, тому що для організації було важко підтримувати безліч фізичних серверів.

Ера віртуального розгортання: у якості рішень була представлена віртуалізація. Вона дозволяє запускати кілька віртуальних машин (ВМ) на одному фізичному сервері. Віртуалізація ізолює додатки між віртуальними машинами та забезпечує певний рівень безпеки, оскільки інформація про один додаток не може бути доступна в іншому додатку. Віртуалізація дозволяє краще використовувати ресурси на фізичному сервері та забезпечує найкращу масштабованість, оскільки додаток можна легко додати або оновити, завдяки цьому зниженню затрат на обладнання та багато іншого. За допомогою віртуалізації можна передати набір фізичних ресурсів у кластер одноразових віртуальних машин. Кожна віртуальна машина представляє собою повноцінну машину, на якій виконуються всі компоненти, включаючи власну операційну систему, поверх віртуалізованого обладнання.

Ера контейнерів: контейнери, схожі на віртуальні машини, але у них немає властивостей ізоляції для спільного використання операційної системи (ОС) між додатками. Тому контейнери вважаються легкими. Як і віртуальна машина, контейнер має власну файлову систему (рис. 1.1), процесор, пам'ять, просторовий процес та багато іншого. Оскільки вони не пов'язані з базовою інфраструктурою, вони переносяться між хмарищами та дистрибутивами ОС.

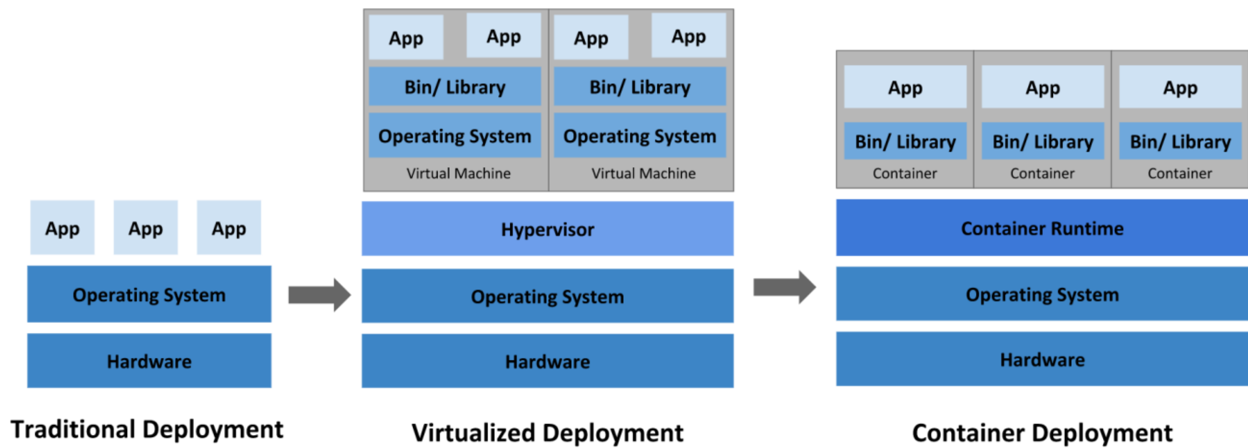

Рис. 1.1. Види розгортання додатків

Контейнери стали популярними із-за таких додаткових переваг як:

- Гібке створення та розробка додатків: простота та ефективність створення образотворчого контейнера за порівнянням з використанням засобів віртуальної роботи;
- Непреривна розробка, інтеграція та розробка: забезпечує надійну та часту збірку та розвернення обробки контейнерів із швидкими та простими відкатами (завдяки невизначеності образів);
- Розділення завдань між Dev і Ops: створення образів контейнерів, що додаються до часу збору / релізу, а не під час розробки, тим самим окремими додатками від інфраструктури;
- Відстежування охоплює не тільки інформацію та метрики на рівні ОС, а також інформацію про робочі можливості додатків та інші сигнали;
- Ідентична навколишня середа при розробці, тестуванні та релізі: на ноутбуках працює таким же чином, як і в хмарищах;
- Переносимість хмарищних та операційних систем: працює на Ubuntu, RHEL, CoreOS, on-prem, Google Kubernetes Engine та в іншому місці;
- Управління, орієнтоване на додатки: підвищує рівень абстракції від запуску ОС на віртуальному обладнанні, до запуску додатків в ОС із використанням логічних ресурсів;

- Слабозв’язані, розподільні, гнучкі, виделені мікросервіси: замість монолітного стека на одній великій виделеній машині, додатки, розбиті на більші мелки незалежних частин, які можна динамічно розвертати та керувати;
- Ізоляція ресурсів: пропозиція про виготовлення додатків;
- Грамотне використання ресурсів: висока ефективність та компактність.

Контейнери - відмінна можливість зв’язати та запустити додатки. У виробничій среді необхідно керувати контейнерами, які запускають додатки та гарантують відсутність простоїв. Наприклад, якщо контейнер виходить із строю, необхідно запустити інший контейнер. Було б набагато простіше, якщо така поведінка оброблялася б системою. Ось тут Kubernetes приходить на допомогу. Kubernetes надає фреймворк для гнучкої роботи розподілених систем. Він займається масштабуванням і обробкою помилок у додатках, пропонує шаблони розробки та багато іншого. Наприклад, Kubernetes може легко керувати канареечним перетворенням системи. Kubernetes надає такі можливості:

- Моніторинг сервісів та розподілення навантажень. Kubernetes може виявити контейнер, використовуючи ім'я DNS або власний IP-адрес. Якщо трафік у контейнері високий, Kubernetes може збалансувати навантаження та розподілити мережевий трафік, щоб перетворення стало стабільним;
- Оркестрація сховищ. Kubernetes дозволяє вам автоматично зміцнити систему зберігання за вашим вибором, таку як локальне сховище, провайдери загальнодоступного обласного управління та багато іншого;
- Автоматичне розвернення та відкати. Використовуючи Kubernetes можна описати бажаний стан розгорнутих контейнерів та змінити фактичний стан на бажаний. Наприклад, ви можете автоматизувати Kubernetes для створення нових контейнерів для розробки, удалення існуючих контейнерів та розподілу всіх їх ресурсів у новому контейнері;
- Автоматичне розподілення навантажень. Ми надаємо кластер Kubernetes, який він може використовувати для запуску контейнерних завдань.

Ми вказуємо Kubernetes, суто ЦП та пам'ять (ОЗУ) вимагає кожного контейнера. Kubernetes може розміщувати контейнери на наших таксах, щоб найбільш ефективно використовувати ресурси;

- Самоконтроль Kubernetes перезапускає показ контейнерів, замінює та завершує роботу контейнерів, які не проходять певний користувач перевірки роботоздатності, і не показує своїх клієнтів, тому вони не будуть готові до обслуговування;
- Управління конфіденційною інформацією та конфігурацією Kubernetes може зберігати та керувати конфіденційною інформацією, як-от паролі, OAuth-токени та ключі SSH. Ви можете розробити та оновити конфіденційну інформацію та конфігурацію додатків без змін образок контейнерів та нерозкриту конфіденційну інформацію в конфігурації стека.

1.2.2. Інфраструктура платформи Kubernetes

При розгортанні Kubernetes уся робота виконується у кластері. Кластер Kubernetes складається з набору машин, так званих вузли, які запускають контейнеризовані додатки. Кластер має як мінімум один робочий вузол. У робочих вузлах розміщені поди (pods), що є компонентами програми. Площина управління керує робочими вузлами і подами в кластері. У промислових середовищах площина управління зазвичай запускається на декількох комп'ютерах, а кластер, як правило, розгортається на кількох вузлах, гарантуючи відмовостійкість і високу надійність. Нижче показана діаграма кластера Kubernetes з усіма пов'язаними компонентами (рис. 1.2.). Компоненти панелі управління відповідають за основні операції кластера (наприклад, планування), а також обробляють події кластера (наприклад, запускають новий під, коли поле replicas розгортання не відповідає необхідній кількості реплік). Компоненти панелі управління можуть бути запущені на будь-якій машині в кластері. Однак для простоти сценарію налаштування зазвичай запускають усі компоненти панелі управління на одному комп'ютері і в той же

час не дозволяють запускати призначені для користувача контейнери на цьому комп'ютері.

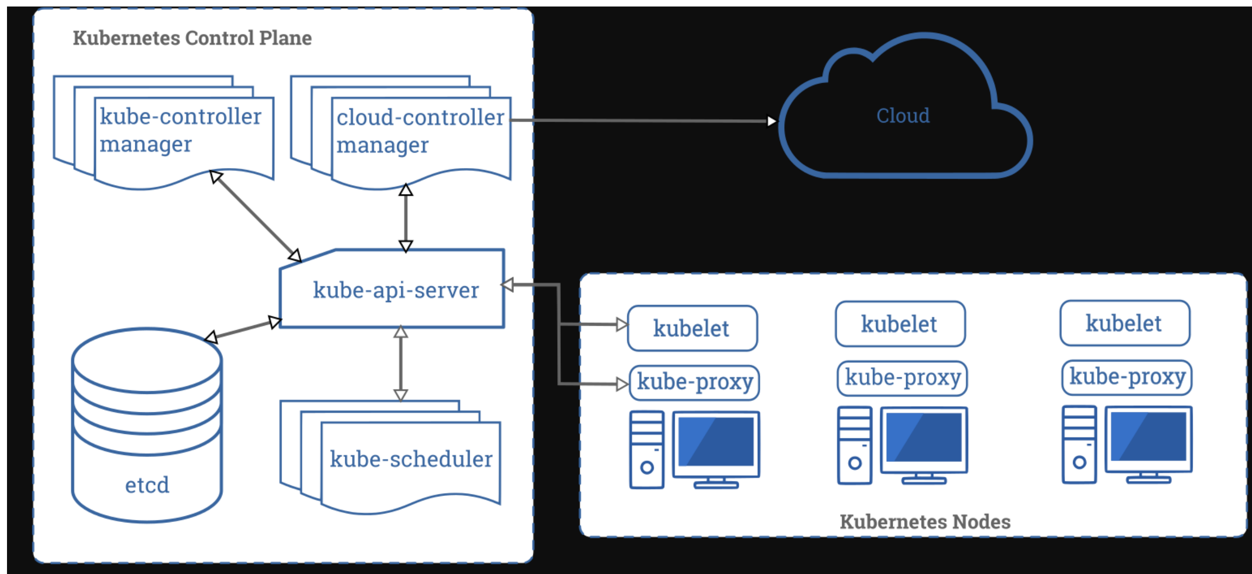

Рис. 1.2. Кластер у Kubernetes

Kube-apiserver - компонент Kubernetes панелі управління, який представляє API Kubernetes. API-сервер - це клієнтська частина панелі управління Kubernetes. Основний реалізацією API-сервера Kubernetes є kube-apiserver. kube-apiserver призначений для горизонтального масштабування, тобто розгортання на кількох подах. Надає можливість запустити кілька екземплярів kube-apiserver і збалансувати трафік між цими подами. Etcd - розподілене і високонадійні сховище даних в форматі "ключ-значення", яке використовується як основне сховище всіх даних кластера в Kubernetes. Kube-sheduler - компонент площини управління, який відстежує створені поди без прив'язаного вузла і вибирає вузол, на якому вони повинні працювати. При плануванні розгортання подів на вузлах враховуються безліч чинників, включаючи вимоги до ресурсів, обмеження, пов'язані з апаратними/програмними політиками, приналежності та неналежність вузлів/подів, місцезнаходження даних, граничних термінів.

Kube-controller-manager - компонент control plane, запускає процеси контролера. Кожен контролер в свою чергу є окремимим процесом, і для

спрощення всі такі процеси скомпільовані в один двійковий файл і виконуються в одному процесі. Ці контролери включають у себе:

- Контролер вузла (Node Controller): повідомляє і реагує на збої вузла;
- Контролер реплікації (Replication Controller): підтримує правильну кількість подів для кожного об'єкта контролера реплікації в системі;
- Контролер кінцевих точок (Endpoints Controller): заповнює об'єкт кінцевих точок (Endpoints), тобто пов'язує сервіси (Services) та поди (Pods);
- Контролери облікових записів і токенів (Account & Token Controllers): створюють стандартні облікові записи і маркери доступу API для нових просторів імен.

Cloud-controller-manager запускає контролери, які взаємодіють з основними хмарними провайдерами. Двійковий файл cloud-controller-manager - це альфа-функціональність, що з'явилися в Kubernetes 1.6. cloud-controller-manager запускає тільки цикли контролера, що відносяться до хмарного провайдера. Ви можете відключити цикли контролера, встановивши прапор --cloud-provider зі значенням external при запуску kube-controller-manager. За допомогою cloud-controller-manager код як хмарних провайдерів, так і самого Kubernetes може розроблятися незалежно один від одного. У попередніх версіях код ядра Kubernetes залежав від коду, призначеного для функціональності хмарних провайдерів. У майбутніх випусках код, специфічний для хмарних провайдерів, повинен підтримуватися самим хмарним провайдером і компонуватись з cloud-controller-manager під час запуску Kubernetes. Наступні контролери залежать від хмарних провайдерів:

- Контролер вузла (Node Controller): перевіряє хмарний провайдер, щоб визначити, чи був видалений вузол в хмарі після того, як він перестав працювати;
- Контролер маршрутів (Route Controller): налаштовує маршрути в основній інфраструктурі хмари;

- Контролер сервісів (Service Controller): створює, оновлює і видаляє балансувальник навантаження хмарного провайдера;
- Контролер томи (Volume Controller): створює, приєднує і монтує томи, а також взаємодіє з хмарним провайдером для оркестрації томів.

Компоненти вузла працюють на кожному вузлі, підтримуючи роботу подів і середовища виконання Kubernetes. Kubelet - агент, який працює на кожному вузлі в кластері. Він стежить за тим, щоб контейнери були запущені в поді. Утиліта kubelet приймає набір PodSpecs, і гарантує працездатність і справність визначених у них контейнерів. Агент kubelet не відповідає за контейнери, які не створені Kubernetes.

Kube-proxy - мережевий проксі, який працює на кожному вузлі в кластері, і який реалізує частину концепції сервіс. Kube-proxy конфігурує правила мережі на вузлах. За допомогою них вирішуються мережеві підключення до подів зсередини і зовні кластера. Kube-proxy використовує рівень фільтрації пакетів в операційнії системі, якщо він доступний. В іншому випадку, kube-proxy сам обробляє передачу мережевого трафіку. Середовище виконання контейнера - це програма, призначена для виконання контейнерів. Kubernetes підтримує кілька середовищ для запуску контейнерів: Docker, containerd, CRI-O, і будь-яка реалізація Kubernetes CRI (Container Runtime Interface). Доповнення використовують ресурси Kubernetes (DaemonSet, Deployment і т.д.) для розширення функціональності кластера. Оскільки доповнення охоплюють весь кластер, ресурси відносяться до простору імен kube-system. Хоча інші доповнення не є строго обов'язковими, проте при цьому у всіх Kubernetes-кластерів повинен бути кластерний DNS, так як багато прикладів припускають його наявність. Кластерний DNS - це DNS-сервер поряд з іншими DNS-серверами в оточенні, який оновлює DNS-записи для сервісів Kubernetes. Контейнери, запущені за допомогою Kubernetes, автоматично включається до інших DNS-сервер в свої DNS.

Dashboard - це універсальний веб-інтерфейс для кластерів Kubernetes. За допомогою цієї панелі, користувачі можуть управляти та усувати помилки

кластера та програм, які потребують кластер. Моніторинг ресурсів контейнера записує загальні метрики щодо контейнерів у вигляді часових рядів в центральній базі даних і пропонує користувальницький інтерфейс для перегляду цих даних. Механізм логування кластера відповідає за збереження логів контейнера в централізованому сховищі логів з можливістю їх пошуку або перегляду.

1.3. Основні компоненти платформи та їх призначення

1.3.1. Поди

Поди (Pods) - це найменші одиниці обчислювальної техніки, які можна створити та керувати ними в Kubernetes. Под - це група одного або декількох контейнерів із загальними ресурсами зберігання або мережі та специфікацією запуску контейнерів. Вміст подів завжди розміщується спільно та за розкладом і працює у спільному контексті. Под моделює специфічний для програми «логічний хост»: він містить один або кілька контейнерів програм, які відносно тісно пов'язані. У нехмарних контекстах програми, що виконуються на одній фізичній або віртуальній машині, є аналогами хмарних програм, що виконуються на одному логічному хості. Окрім контейнерів додатків, под може містити контейнери init, які запускаються під час запуску подів. Також под має змогу вводити ефемерні контейнери для налагодження, якщо кластер це пропонує. Спільний контекст поду - це набір просторів імен Linux, cgroups та потенційно інших аспектів ізоляції - тих самих речей, які ізолюють контейнер Docker. У контексті поду, окремі додатки можуть застосовувати подальші субізоляції.

З точки зору концепцій Docker, под схожий на групу контейнерів Docker зі спільними просторами імен та спільними томами файлової системи. Зазвичай не потрібно створювати поди безпосередньо, навіть одиночні. Натомість вони створюються, використовуючи такі ресурси робочого навантаження, як розгортання або робота. Якщо подам потрібно відстежувати стан, розглядається

ресурс StatefulSet. Поди в скупченні Kubernetes використовуються двома основними способами:

- Поди, що запускають один контейнер. Модель "один контейнер на один под" - найпоширеніший варіант використання Kubernetes. У цьому випадку можливо уявити под як обгортку навколо одного контейнера. Kubernetes керує подами, а не керує контейнерами безпосередньо;
- Поди, на яких запущено кілька контейнерів, які повинні працювати разом. Под може інкапсулювати додаток, що складається з декількох розташованих одночасно контейнерів, які тісно пов'язані між собою і потребують спільного використання ресурсів. Ці спільно розташовані контейнери утворюють єдину цілісну одиницю обслуговування - наприклад, один контейнер, що обслуговує дані, що зберігаються у спільному томі для загального користування, тоді як окремий контейнер оновлює ці файли. Под обертає ці контейнери, ресурси зберігання та ефемерну мережеву ідентичність у єдине ціле.

Кожен под призначений для запуску одного екземпляра програми. Якщо є необхідність масштабувати додаток горизонтально (щоб забезпечити більше загальних ресурсів, запустивши більше екземплярів), потрібно використовувати декілька подів, по одному для кожного екземпляра. У Kubernetes це зазвичай називають реплікацією. Реплікаційні підсистеми зазвичай створюються та керуються у вигляді групи за допомогою ресурсу робочого навантаження та його контролера. ODS призначені для підтримки кількох взаємодіючих процесів (як контейнери), що утворюють цілісну одиницю обслуговування. Контейнери у поді автоматично розташовуються та плануються спільно за допомогою однієї фізичної або віртуальної машини в кластері. Контейнери можуть обмінюватися ресурсами та залежностями, спілкуватися між собою та координувати, коли та як вони припиняються. Наприклад, може бути контейнер, який виконує роль веб-сервера для файлів у спільному томі, і окремий контейнер "Sidecar", який оновлює ці файли безпосередньо за допомогою віддаленого джерела.

Деякі підсистеми мають контейнери ініціалізації, а також контейнери додатків. Контейнери Init запускаються та завершуються до запуску контейнерів програм. Поди - неділена одиниця в платформі Kubernetes. При створенні розвертування в Kubernetes створюються поди з контейнерами всередині (рис. 1.3.). Кожен под-об'єкт пов'язаний з вузлом, під час якого він розміщений, і залишається там до завершення роботи (відповідно до стратегії перезапуску) або знищення. У разі невідповідності такої же дії буде розподілено на інших доступних вузлах у кластері.

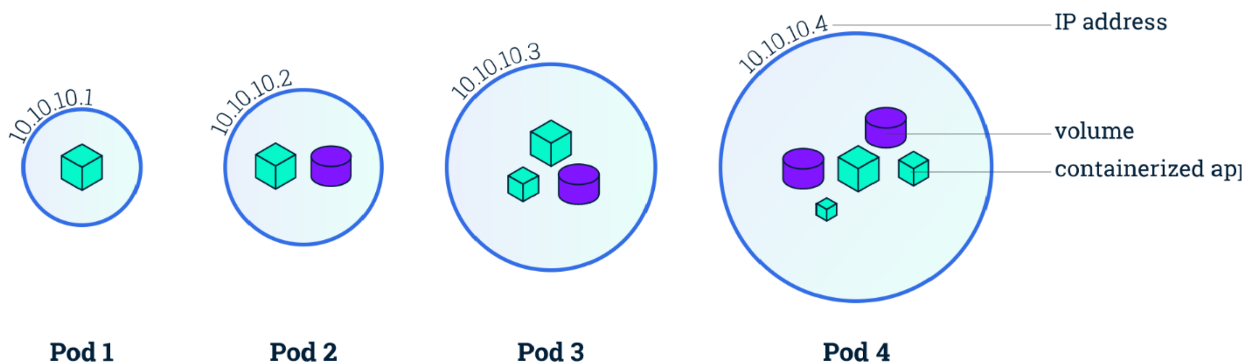

Рис. 1.3. Схема компоненту под

Поди розроблені як відносно швидкоплинні одноразові сутності. Коли под створюється (безпосередньо користувачем або власне контролером), новий под планується запуститися на вузлі у кластері. Под залишається на цьому вузлі до тих пір, поки не закінчить виконання, об’єкт буде видалений, под знищений за браком ресурсів або вузол не вийде з ладу. Можливо використовувати ресурси робочого навантаження, щоб створити та управляти кількома подами. Контролер ресурсу обробляє реплікацію та розгортання та автоматичне загоєння у випадку відмови поду. Наприклад, якщо вузол виходить з ладу, контролер помічає, що поди на цьому вузлі перестали працювати, і створює заміну під. Планувальник розміщує замінник поду на здоровий вузол. Контролери ресурсів робочого навантаження створюють підсистеми з шаблону підсистеми та керують цими підсистемами від імені користувача.

PodTemplates - це специфікації для створення подів, і вони включаються в ресурси робочого навантаження, такі як розгортання, завдання та DaemonSets. Кожен контролер ресурсу робочого навантаження використовує PodTemplate усередині об’єкта робочого навантаження для створення фактичних подів. PodTemplate є частиною бажаного стану будь-якого ресурсу робочого навантаження, який використовується для запуску програми.

Зміна шаблону поду або перехід на новий шаблон не має прямого впливу на вже існуючі поди. Якщо змінюється шаблон модуля для ресурсу робочого навантаження, цей ресурс повинен створити запасні модулі, які використовують оновлений шаблон. Наприклад, контролер StatefulSet гарантує, що запущені поди відповідають поточному шаблону поду для кожного об'єкта StatefulSet. Якщо редагується StatefulSet, щоб змінити його шаблон поду, StatefulSet починає створювати нові поди на основі оновленого шаблону. Врешті-решт, всі старі поди замінюються новими, і оновлення завершено. Под може вказувати набір спільних томів пам'яті. Усі контейнери у поді можуть мати доступ до спільних томів, дозволяючи цим контейнерам обмінюватися даними. Томи також дозволяють постійним даним у поді зберігатись, якщо один із контейнерів усередині потребує перезапуску. Кожному підрозділу присвоюється унікальна IP-адреса для кожної родини адрес.

Кожен контейнер у поді має спільний доступ до простору імен мережі, включаючи IP-адресу та мережеві порти. Усередині поду (і лише тоді) контейнери, які належать поду, можуть спілкуватися між собою за допомогою localhost. Коли контейнери у поді спілкуються з сутностями за межами поду, вони повинні координувати спосіб використання спільних мережевих ресурсів (наприклад, портів). У межах поду, контейнери мають спільну IP-адресу та простір порту, і можуть знаходити один одного через localhost. Контейнери в поді можуть також спілкуватися між собою, використовуючи стандартні міжпроцесорні комунікації, такі як семафори SystemV або спільна пам'ять POSIX. Контейнери в різних підрозділах мають різні IP-адреси і не можуть спілкуватися за допомогою IPC без спеціальної конфігурації. Контейнери, які

хочуть взаємодіяти з контейнером, що працює в іншому модулі поду, можуть використовувати IP-мережі для спілкування. Контейнери всередині поду бачать, що ім'я хосту системи є таким самим, як і налаштоване ім'я для поду. Будь-який контейнер в поді може вмикати привілейований режим, використовуючи привілейований прапор у контексті безпеки специфікації контейнера. Це корисно для контейнерів, які хочуть використовувати адміністративні можливості операційної системи, такі як маніпулювання мережевим стеком або доступ до апаратних пристроїв. Процеси в привілейованому контейнері отримують майже ті самі привілеї, які доступні для процесів поза контейнером.

Статичні поди управляються безпосередньо демоном kubelet на певному вузлі, без того, щоб сервер API їх спостерігав. У той час як більшістю подів є керованою площиною управління (наприклад, розгортання), для статичних подів кубелет безпосередньо контролює кожен статичний под (і перезапускає його, якщо він не працює). Статичні поди завжди пов'язані з одним кубелетом на певному вузлі. Основне використання статичних подів - запуск самообслуговуваної площини управління: іншими словами, використання кубелета для нагляду за окремими компонентами площини управління. Kubelet автоматично намагається створити дзеркальний под на сервері Kubernetes API для кожного статичного поду.

Поди слідують за визначеним життєвим циклом, починаючи з фази очікування, переходячи через запуск (рис. 1.4.), якщо принаймні один з його основних контейнерів починає працювати нормально, а потім через фази "успіх" або "помилка", залежно від того, закінчився який-небудь контейнер у поді. Поки працює под, kubelet може перезапустити контейнери для обробки якихось несправностей. У межах поду, Kubernetes відстежує різні стани контейнерів і визначає, які дії вжити, щоб под знову став здоровим. В API Kubernetes, поди мають як специфікацію, так і фактичний статус. Статус об'єкта под складається з набору умов поду. Поди плануються лише один раз у житті. Як тільки под було призначено для вузла, він працює на цьому вузлі, доки не зупиниться або не припиниться. Як і окремі контейнери додатків, поди вважаються відносно

ефемерними (а не довговічними) сутностями. Структури створюються, їм присвоюється унікальний ідентифікатор (UID) та плануються вузли, де вони залишаються до завершення (відповідно до політики перезапуску) або видалення. Якщо вузол вмирає, підсистеми, призначені для цього вузла, планується видалити після періоду очікування.

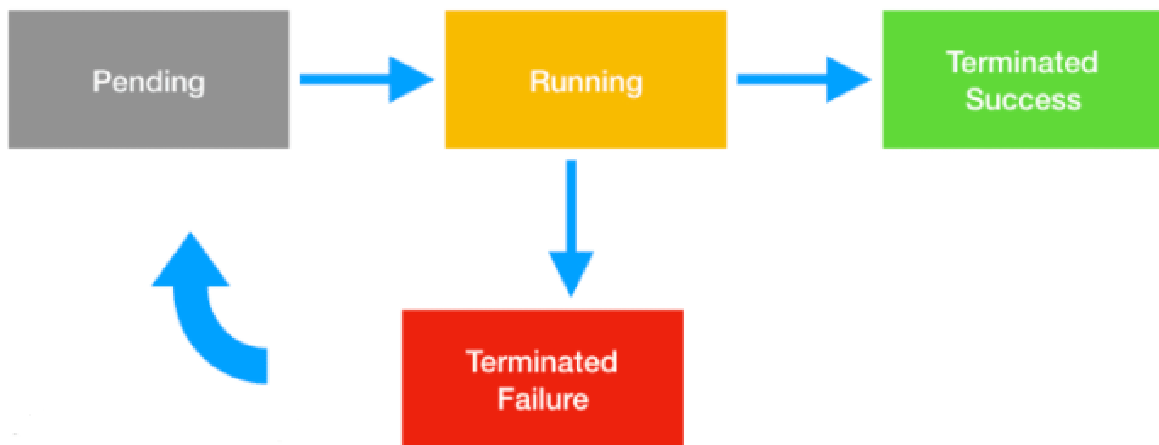

Рис. 1.4. Життєвий цикл поду

Поди самі по собі не відновлюються. Якщо под заплановано на вузол, який потім виходить з ладу, або якщо сама операція планування не вдається, под видаляється; так само, под не витримає виселення через брак ресурсів або обслуговування ноди. Kubernetes використовує абстракцію вищого рівня, яка називається контролером, і керує роботою з управління відносно одноразовими екземплярами поду.

1.3.2. Сервіси та деплойменти. Керування подами

Сервіси – це абстрактний спосіб виставити програму, що працює на наборі подів, як послугу мережі. З Kubernetes не потрібно модифікувати програму, щоб використовувати незнайомий механізм виявлення служби. Kubernetes надає подам їх власні IP-адреси та єдине DNS-ім'я для набору подів, і може

балансувати навантаження між ними. Поди Kubernetes створюються та знищуються відповідно до стану кластера. Поди - це непостійні ресурси. Якщо використовується Deployment для запуску програми, він може динамічно створювати та знищувати поди. Кожен под отримує свою власну IP-адресу, однак у розгортанні набір подів, запущених за один момент часу, може відрізнятися від набору подів, що запускає цю програму мить пізніше. Це призводить до проблеми: якщо якийсь набір подів (назвемо їх "backends") надає функціональність іншим подам (назвемо їх "frontends") всередині кластера, як інтерфейси дізнаються та відстежують, до якої IP-адреси потрібно підключитися, щоб інтерфейс міг використовувати внутрішню частину робочого навантаження? Для цього вводяться сервіси. У Kubernetes сервіс - це абстракція, яка визначає логічний набір подів та політику, за допомогою якої можна отримати до них доступ (іноді цей шаблон називається мікросервісом). Набір подів, на які спрямована послуга, зазвичай визначається селектором. Наприклад, бекенд для обробки зображень без стану, який працює з 3 репліками. Ці репліки є взаємозамінними - інтерфейсам не важливо, яку бекенд вони використовують. Хоча фактичні поди, що складають серверний набір, можуть змінюватися, клієнтам зовнішнього інтерфейсу не повинно бути відомо про це, а також їм не слід відстежувати сам набір backends.

Абстракція служби дозволяє це роз'єднання. Якщо можливо використовувати API Kubernetes для виявлення служби у додатку, можливо надіслати запит на сервер API для кінцевих точок, які оновлюються при зміні набору подів у службі. Для інших програм Kubernetes пропонує способи розміщення мережевого порту або балансу завантаження між додатком та серверними підсистемами. Служба в Kubernetes - це об’єкт REST, схожий на под. Як і всі об'єкти REST, k8s дозволяє розмістити визначення служби на сервері API, щоб створити новий екземпляр. Ім'я об'єкта служби має бути дійсним ім'ям DNS-мітки. Наприклад, припустимо, є набір подів, які сервіс прослуховує на TCP-порту 9376 і містить назву app = MyApp:

//

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
//
```

Ця специфікація створює новий сервісний об'єкт з назвою "my-service", який націлений на TCP-порт 9376 на будь-якому пристрої з міткою app = MyApp. Kubernetes присвоює цій службі IP-адресу (іноді її називають "кластерною IP"), яку використовують проксі-сервери служби. Контролер селектора послуг постійно сканує підстанції, що відповідають його селектору, а потім публікує будь-які оновлення об'єкта кінцевої точки, який також називається "моя служба". Визначення портів у подах мають імена, і є можливість посилатися на ці імена в атрибуті targetPort служби. Це працює, навіть якщо в службі є суміш подів, що використовують одне налаштоване ім'я, з однаковим мережевим протоколом, доступним через різні номери портів. Це пропонує велику гнучкість для розгортання та вдосконалення додатків.

Деплойменти (deployments) представляють собою набір з декількох однакових подів (рис. 1.5.) без унікальних ідентифікаційних даних. Розгортання запускає кілька реплік додатку та автоматично замінює будь-які екземпляри, які не працюють або не реагують. Таким чином, розгортання допомагають забезпечити доступність одного або декількох екземплярів програми для обслуговування запитів користувачів. Розгортаннями керує контролер розгортання Kubernetes. Розгортання використовують шаблон поду, який містить специфікацію для своїх подів. Специфікація поду визначає, як повинен виглядати кожен под: які програми повинні запускатися всередині його контейнерів, які обсяги слід монтувати поду, його мітки тощо. Коли шаблон розгортання поду змінюється, нові поди автоматично створюються по одному.

Деплойменти добре підходять для програм, які використовують томи ReadOnlyMany або ReadWriteMany, встановлені на декількох репліках, але не

підходять для робочих навантажень, що використовують томи ReadWriteOnce. Для додатків із підтримкою стану, що використовують томи ReadWriteOnce, використовуються StatefulSets. StatefulSets призначені для розгортання додатків, що містять стани, та кластерних програм, які зберігають дані у постійному сховищі, наприклад, на постійних дисках Compute Engine. StatefulSets підходять для розгортання програм Kafka, MySQL, Redis, ZooKeeper та інших, що потребують унікальних, стійких ідентифікаційних даних та стабільних імен хостів. Після створення розгортання гарантує, що бажана кількість подів працює і буде доступною постійно. Розгортання автоматично замінює поди, які виходять з ладу або витісняються з їх вузлів. Нижче наведено приклад файлу маніфесту розгортання у форматі YAML:

```
//
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
//
```

У цьому прикладі створено деплоймент з назвою nginx. Деплоймент створює три репліки поду, для котрих він відкриває порт 80. Специфікація шаблону поду вказує, що на кожному поді працює по одному контейнеру, з nginx-образом. Щоб оновити розгортання, необхідно зробити зміни у специфікації шаблону поду розгортання. Внесення змін до поля специфікації автоматично запускає випуск оновлення. Для цього можливо використовувати kubectl, API Kubernetes або меню GKE Workloads в Google Cloud Console.

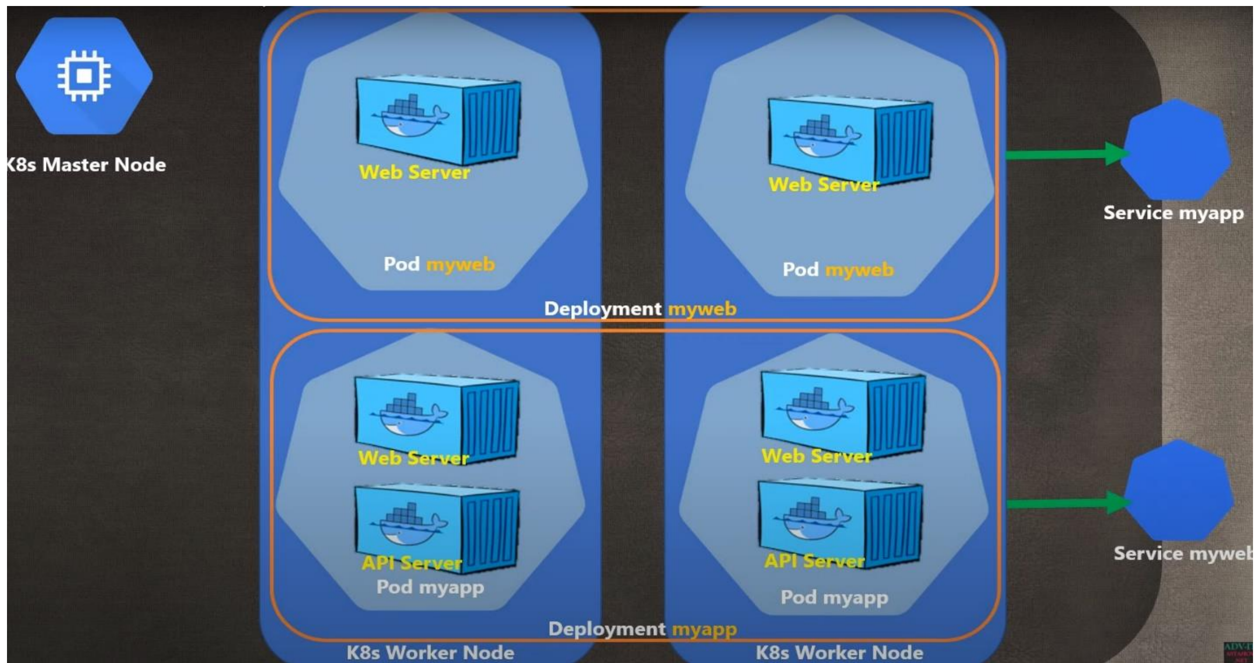

Рис. 1.5. Структура деплойменту та сервісу

За замовчуванням, коли розгортання запускає оновлення, розгортання зупиняє структури, поступово зменшує кількість структур до нуля, потім зливає та припиняє структури. Потім розгортання використовує оновлений шаблон поду для створення нових подів. Старі поди не видаляються, поки не запускається достатня кількість нових подів, а нові поди не створюються, поки не буде видалено достатню кількість старих подів. Щоб побачити, в якому порядку виводяться та видаляються поди, необхідно запустити kubectl з описом розгортань.

Розгортання можуть гарантувати, що працює щонайменше на одну менше ніж бажано кількості реплік, причому щонайменше один под недоступний. Подібним чином, розгортання можуть забезпечити роботу щонайбільше більше на одну, ніж бажана кількість реплік, причому щонайменше на один под більше, ніж бажано. Для відкату оновлення застосовується команда відміни розгортання kubectl. Є можливість використовувати паузу розгортання kubectl, щоб тимчасово зупинити розгортання. Частіше за все, деплоймент виповнює одне з наступних завдань: масштабування розгортання, автомасштабування

розгортання за допомогою об'єкту HorizontalPodAutoscaler, видалення розгортання.

Розгортання можуть перебувати в одному з трьох станів протягом його життєвого циклу: прогресуючим, завершеним або невдалим. Прогресуючий стан вказує на те, що розгортання виконує свої завдання, наприклад, масштабує свої поди. Завершений стан вказує на те, що розгортання успішно виконало свої завдання, всі його підрозділи працюють за останньою специфікацією і доступні, а старі підстанції досі не працюють. Помилковий стан вказує на те, що розгортання зіткнулося з однією або кількома проблемами, які заважають йому виконувати свої завдання. Деякі причини включають недостатню квоту або дозволи, помилки витягування images, діапазони обмежень або помилки виконання.

1.4. Висновки до першого розділу

У першому розділі було розглянуто основні проблеми, які виникали під час автоматизації роботи веб-додатків. Проаналізовано усі види розгротрань додатків, плюси та мінуси кожного з них. Досліджено платформу Kuvernetes, її мету, функції та призначення. Розглянуто основні компоненти та іх функції, інфраструктуру платформи.

З виходом платформи Kubernetes, проблема гнучкої доставки додатків на сервера у контейнерах, їх управління та масштабування, була вирішена. Проблем з коректним функціонуванням додатків без належного управління конфігурацією більше не було. Kubernetes (K8s) – це відкрите програмне забезпечення для автоматизації розгортання, масштабування та управління контейнеризованими додатками. K8s групує контейнери, які створюють додаток, у логічні одиниці для простішого управління та виявлення. Kubernetes пропонує можливість масштабування будь-якого рівня, контейнери с додатками можуть буди розгорнуті будь-де, у будь-якому середовищі. На платформі K8s можна виділити 3 основних типи компонентів:

- Поди (pods) - це найменші одиниці обчислювальної техніки, які можна створити та керувати ними в Kubernetes. Pod - це група одного або декількох контейнерів із загальними ресурсами зберігання / мережі та специфікацією запуску контейнерів;
- Розгортання (deployments) - це набір з декількох однакових подів без унікальних ідентифікаційних даних. Розгортання запускає кілька реплік додатку та автоматично замінює будь-які екземпляри, які не працюють або не реагують. Таким чином, розгортання допомагають забезпечити доступність одного або декількох екземплярів програми для обслуговування запитів користувачів;
- Сервіси (services) – це абстракція, яка визначає логічний набір подів та політику, за допомогою якої можна отримати до них доступ.

РОЗДІЛ 2
КОНТРОЛЕРИ, ОПЕРАТОРИ ТА ВЛАСНІ РЕСУРСИ НА ПЛАТФОРМІ K8S. ОСНОВНІ МОЖЛИВОСТІ АВТОМАТИЗАЦІЇ УПРАВЛІННЯ ВЕБ-ДОДАТКАМИ

2.1. Основні функції платформи для автоматизації роботи контейнерізованих веб-додатків

У K8s є безліч команд для управління, автоматизації та розподілення навантажень у контейнерізованих веб-додатках. Основні команди платформи представлено у цьому розділі.

Команда get відображає один або кілька ресурсів кластера. Ця команда друкує таблицю найважливішої інформації про всі ресурси, включаючи вузли кластера, запущені поди, контролери реплікації та служби.

Щоб перелічити всі поди з детальною інформацією:

kubectl get po -o wide

Щоб відобразити поди у всіх просторах імен:

kubectl get po --all-namespaces

Щоб перерахувати всі простори імен вузла:

kubectl get namespace

Щоб відобразити под із вказаною назвою у вихідному форматі YAML:

kubectl get po <podname> -o yaml

Команда create створює ресурс кластера з файлу або вводу. Якщо дескриптор ресурсу вже існує (файл YAML або JSON), ви можете створити ресурс із файлу, виконавши таку команду:

kubectl create -f filename

Команда expose виставляє ресурс як нову службу Kubernetes. Можливі ресурси включають под, контролер реплікації, службу та розгортання:

kubectl expose deployment deployname --port=81 --type=NodePort --target-port=80 --name=service-name

Щоб запустити певний image в кластері:

kubectl run deployname --image=nginx:latest

Команда set налаштовує об'єктні ресурси. Щоб змінити image розгортання з іменем, зазначеним у імені розгортання, на образ 1.0:

kubectl set image deploy deployname containername=containername:1.0 Команда edit редагує ресурс із редактора за замовчуванням. Щоб оновити под:

kubectl edit po po-nginx-btv4j

Команда delete видаляє ресурси за назвою або міткою ресурсу. Щоб видалити под із мінімальною затримкою:

kubectl delete po podname --now

kubectl delete -f nginx.yaml

kubectl delete deployment deployname

rolling-update - оновлює запущену службу з нульовим простоєм. Поди поступово замінюються новими. За раз оновлюється один под. Старий под видаляється лише після того, як новий под запущено. Нові поди повинні відрізнятися від старих подів за назвою, версією та маркуванням. В іншому випадку повідомлення про помилку буде повідомлено:

kubectl rolling-update poname -f newfilename

kubectl rolling-update poname -image=image:v2

Якщо під час поновлення оновлення виникає яка-небудь проблема, використовується команда з позначкою -rollback, щоб скасувати поновлення оновлення та повернутися до попереднього стану:

kubectl rolling-update poname -rollback

Команда rollout управляє випуском ресурсу. Щоб перевірити статус деплойменту певного розгортання:

kubectl rollout status deployment/deployname

Щоб переглянути історію деплойменту певного розгортання:

kubectl rollout history deployment/deployname

Щоб повернутися до попереднього розгортання: (за замовчуванням ресурс відкочується до попередньої версії)

kubectl rollout undo deployment/test-nginx

Команда scale встановлює новий розмір для ресурсу, регулюючи кількість реплік ресурсів:

kubectl scale deployment deployname --replicas=newnumber

Команда автомасштабування (autoscale) автоматично вибирає та встановлює кількість подів. Ця команда задає діапазон для кількості реплік подів, що підтримуються контролером реплікації. Якщо занадто багато подів, контролер реплікації припиняє зайві модулі. Якщо їх замало, контролер реплікації запускає більше модулів:

kubectl autoscale deployment deployname --min=minnumber --max=maxnumber

Команда переадресації портів перенаправляє один або кілька локальних портів на под. Щоб прослуховувати порти 5000 і 6000 локально, пересилаючи дані на/з портів 5000 і 6000 у поді:

kubectl port -forward podname 5000:6000

Команда proxy створює проксі-сервер між localhost та сервером API Kubernetes. Щоб увімкнути API REST HTTP на головному вузлі:

kubectl proxy -accept-hosts= '.*' -port=8001 -address= '0.0.0.0'

2.2. Роль контролерів у процесах автоматизації та управління додатками на платформі K8s

У Kubernetes контролери - це цикли управління, які спостерігають за станом кластера, а потім вносять або вимагають змін, де це необхідно. Кожен контролер намагається перемістити поточний стан кластера ближче до бажаного стану. Контролер відстежує принаймні один тип ресурсу Kubernetes. Ці об'єкти мають поле специфікації, яке представляє бажаний стан. Контролери для цього ресурсу відповідають за наближення поточного стану до бажаного стану. Контролер може здійснити дію самостійно, частіше в Kubernetes контролер надсилатиме на сервер API повідомлення, які мають корисні побічні ефекти.

Контролер роботи - це приклад вбудованого контролера Kubernetes. Вбудовані контролери управляють станом, взаємодіючи з сервером API кластера.

Робота - це ресурс Kubernetes, який запускає поди, або, можливо, декілька подів, для виконання завдання, а потім зупинки. Коли контролер завдання бачить нове завдання, він переконується, що у кластері на наборі вузлів запускається потрібна кількість подів, щоб виконати роботу. Натомість контролер завдання вказує серверу API створювати або видаляти підсистеми. Після створення нового завдання бажаним станом є виконання цього завдання. Контролер завдання робить поточний стан для цього завдання ближчим до бажаного стану: створюючи структури, які виконують роботу, яку необхідно для цього завдання, щоб завдання було ближче до завершення. Контролери також оновлюють об'єкти, які їх налаштовують. Наприклад: як тільки робота буде виконана для завдання, контролер завдання оновить цей об’єкт завдання, щоб позначити його як готовий. На відміну від роботи, деяким контролерам потрібно вносити зміни поза кластером. Якщо використовується цикл керування, щоб переконатися, що у кластері достатньо вузлів, тоді цей контролер має щось поза поточним кластером (рис. 2.1.), щоб створити нові вузли, коли це потрібно. Контролери, які взаємодіють із зовнішнім станом, знаходять бажаний стан із сервера API, а потім взаємодіють безпосередньо із зовнішньою системою, щоб наблизити поточний стан у відповідність. Важливим моментом тут є те, що контролер вносить певні зміни для досягнення бажаного стану, а потім повідомляє про поточний стан назад на сервер API кластера. Інші цикли керування можуть спостерігати за цими звітними даними та здійснювати власні дії.

З кластерами Kubernetes площина управління опосередковано працює з інструментами управління IP-адресами, службами зберігання, хмарним постачальником APIS та іншими службами, розширюючи K8s для реалізації цього. В якості принципу свого дизайну K8s використовує безліч контролерів, кожен з яких керує певним аспектом стану кластера. Найчастіше певний цикл управління (контролер) використовує один вид ресурсу як бажаний стан і має інший тип ресурсу, який йому вдається здійснити у бажаному стані.

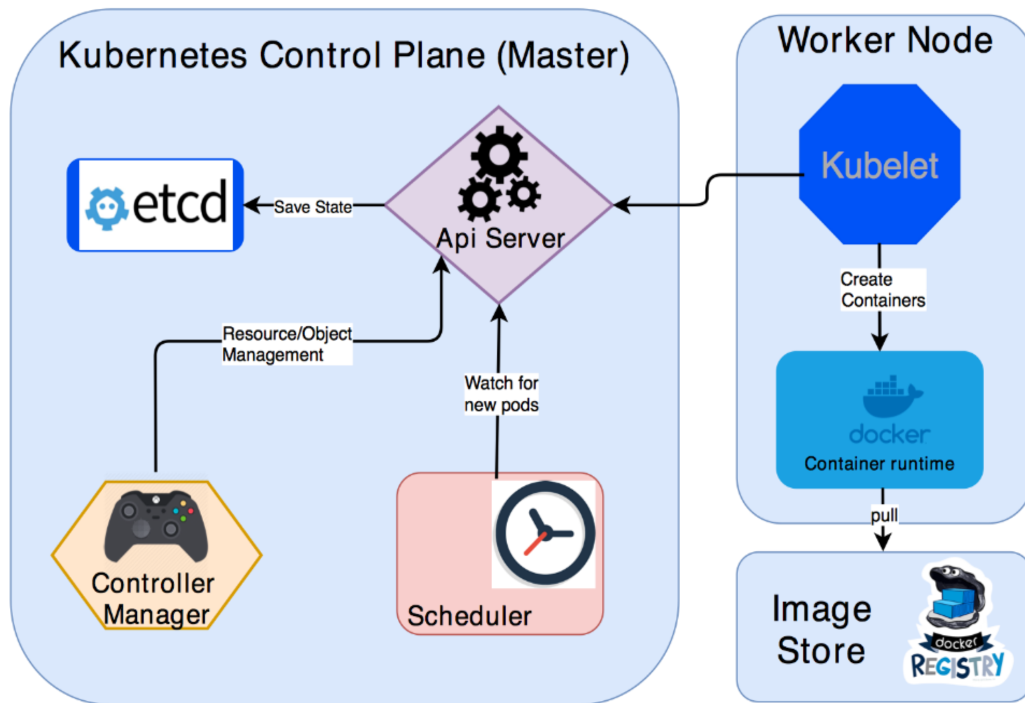

Рис. 2.1. Принцип роботи контролера у Kubernetes

Корисно мати прості контролери, а не один монолітний набір контурів управління, які взаємопов’язані. Контролери можуть вийти з ладу, тому Kubernetes розроблений, щоб це дозволити. Може бути кілька контролерів, які створюють або оновлюють один і той самий об’єкт. За лаштунками контролери Kubernetes переконуються, що вони звертають увагу лише на ресурси, пов’язані з їхнім контрольним ресурсом. Якщо існує розгортання та завдання, вони обидва створюють поди. Контролер завдання не видаляє підсистеми, створені розгортанням, оскільки є інформація (мітки), яку контролери можуть використовувати для розрізнення цих підсистем. Kubernetes постачається з набором вбудованих контролерів, які працюють всередині kube-controller-manager. Ці вбудовані контролери забезпечують важливу основну поведінку. Контролер розгортання та контролер завдань - це приклади контролерів, які входять до складу самого Kubernetes ("вбудовані" контролери). Kubernetes дозволяє запускати еластичну площину управління, так що якщо будь-який із вбудованих контролерів вийде з ладу, інша частина площини управління прийме на себе роботу.

2.3. Розширення платформи та розробка власних ресурсів.

Kubernetes можна легко налаштувати та розширити. Як результат, рідко виникає необхідність подавати виправлення до коду проєкту K8s. Підходи до налаштування можна розділити на конфігурацію, яка передбачає лише зміну прапорів, локальних файлів конфігурації або ресурсів API, та розширення, які передбачають запуск додаткових програм або послуг. Прапори та файли конфігурації не завжди можуть бути зміненими у розміщеній службі K8s або розповсюдженні з керованою інсталяцією. Коли вони мінливі, вони, як правило, змінюються лише адміністратором кластера. Крім того, вони можуть бути змінені в майбутніх версіях K8s, і для їх налаштування може знадобитися перезапуск процесів. З цих причин їх слід використовувати лише тоді, коли інших варіантів немає.

Розширення - це програмні компоненти, які розширюються та глибоко інтегруються з K8s. Вони адаптують його для підтримки нових типів та нових видів обладнання. Більшість адміністраторів кластера використовуватимуть розміщений або розподілений екземпляр K8s. Як результат, більшості користувачів K8s не потрібно буде встановлювати розширення, а меншій кількості потрібно буде створювати нові. K8s призначений для автоматизації шляхом написання клієнтських програм. Будь-яка програма, яка читає та / або пише в API K8s, може забезпечити корисну автоматизацію. Автоматизація може працювати як на кластері, так і поза ним. Дотримуючись вказівок K8s, є можливість написати високодоступну та надійну автоматизацію. Зазвичай автоматизація працює з будь-яким кластером K8s, включаючи розміщені кластери та керовані інсталяції.

Спеціальні ресурси - це розширення API Kubernetes. Ресурс - це кінцева точка в API K8s, що зберігає колекцію об’єктів API певного типу, наприклад, вбудований ресурс под містить колекцію об’єктів подів. Спеціальні ресурси не обов’язково доступні в установці K8s за замовчуванням. Вони являють собою налаштування конкретної установки K8s. Однак багато основних функцій

побудовані з використанням власних ресурсів, що робить Kubernetes більш модульним (рис. 2.2.). Спеціальні ресурси можуть з'являтися та зникати в запущеному кластері за допомогою динамічної реєстрації, а адміністратори кластера можуть оновлювати власні ресурси незалежно від самого кластера. Після встановлення користувацького ресурсу користувачі можуть створювати та отримувати доступ до його об’єктів за допомогою kubectl, як це роблять для вбудованих ресурсів, таких як поди. Спеціальні ресурси самостійно дозволяють зберігати та отримувати структуровані дані. Коли поєднується власний ресурс із користувацьким контролером, користувацькі ресурси надають справжній декларативний API.

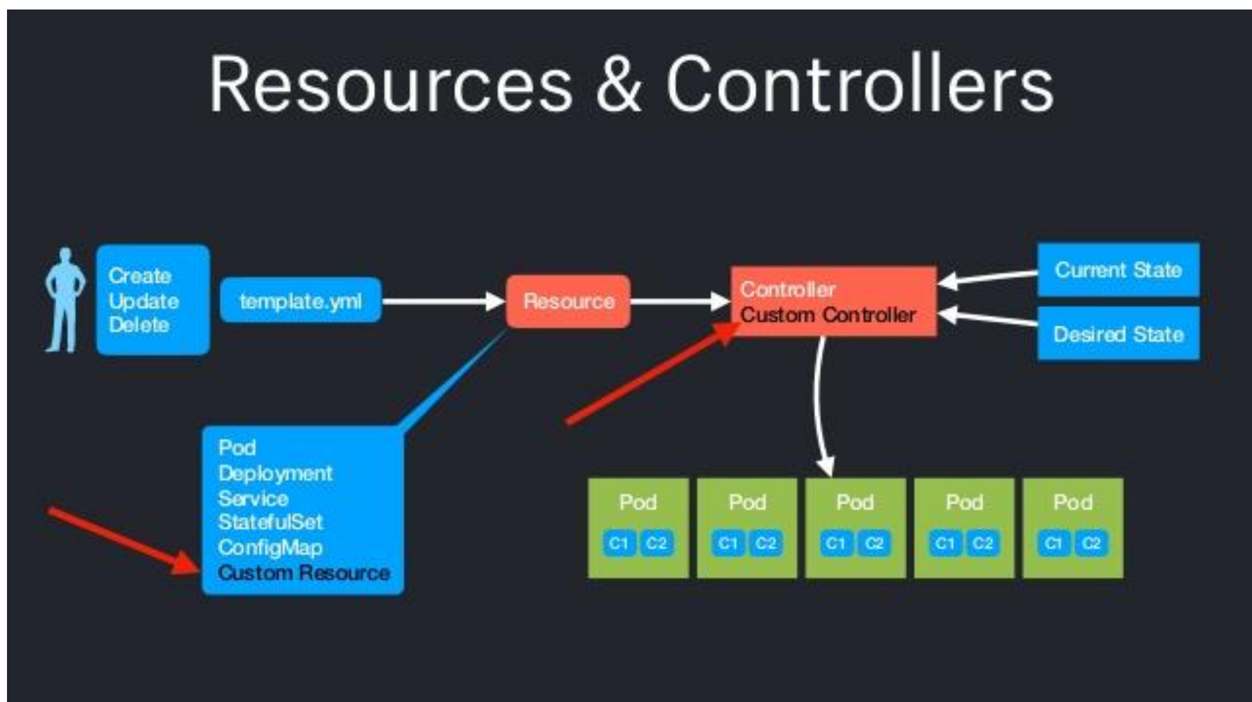

Рис. 2.2. Спеціальні ресурси у Kubernetes

Декларативний API дозволяє оголосити або вказати бажаний стан ресурсу і намагається підтримувати поточний стан об'єктів Kubernetes в синхронізації з бажаним станом. Контролер інтерпретує структуровані дані як запис бажаного користувачем стану і постійно підтримує цей стан. Можливо розгорнути та оновити власний контролер на запущеному кластері, незалежно від життєвого циклу кластера. Спеціальні контролери можуть працювати з будь-якими

ресурсами, але вони особливо ефективні в поєднанні з користувацькими ресурсами. Шаблон оператора поєднує користувацькі ресурси та користувацькі контролери.

Створюючи новий API, необхідно вирішити, чи слід агрегувати його API з кластерними API K8s, або нехай власний (кастомний) API стоїть окремо. Необхідність агрегування API виникає, якщо:

- Власний API є декларативним;
- Необхідно щоб нові типи були читабельними та записаними за допомогою kubectl;
- Необхідно переглядати власні нові типи в інтерфейсі Kubernetes,
- Якщо розробляється новий API;
- Власні ресурси обмежуються кластером або просторами імен кластера;
- Необхідно повторно використовувати функції підтримки Kubernetes API.

Перевага автономному API надається, якщо:

- Власний API не відповідає декларативній моделі;
- Підтримка kubectl не потрібна;
- Підтримка інтерфейсу Kubernetes не потрібна;
- Вже є програма, яка обслуговує власний API і працює добре;
- Необхідно мати певні шляхи REST, щоб бути сумісними з уже визначеним REST API;
- Ресурси з кластером або простором імен погано підходять, потрібен контроль над особливостями шляхів використання ресурсів.

Декларативний API зазвичай складається з відносно невеликої кількості відносно невеликих об’єктів (ресурсів). Об'єкти визначають конфігурацію програм або інфраструктури (рис. 2.3.). Об'єкти зазвичай оновлюються порівняно рідко. Основними операціями над об’єктами є CRUD-у (створення,

читання, оновлення та видалення). Транзакції між об’єктами не потрібні: API представляє бажаний стан, а не точний стан.

Імперативні API не є декларативними. Ознаки того, що API може бути не декларативним, включають:

- Віддалені виклики процедур (RPC);
- Безпосереднє зберігання великих обсягів даних; наприклад,> кілька кБ на об'єкт або> 1000 с об'єктів;
- Необхідний доступ до високої пропускної здатності (10 секунд запитів на секунду);
- Зберігання данних кінцевих користувачів (наприклад, зображення, ідентифікаційні дані тощо) або інші великомасштабні дані, що обробляються програмами;
- Основні дії на об'єктах не є CRUD;
- API непросто моделюється як об'єкти;
- Kubernetes пропонує два способи додати власні ресурси до кластера:
- CRD прості і можуть бути створені без будь-якого програмування.
- Агрегація API вимагає програмування, але дозволяє більше контролювати поведінку API, наприклад, як зберігаються дані та перетворюють між версіями API.

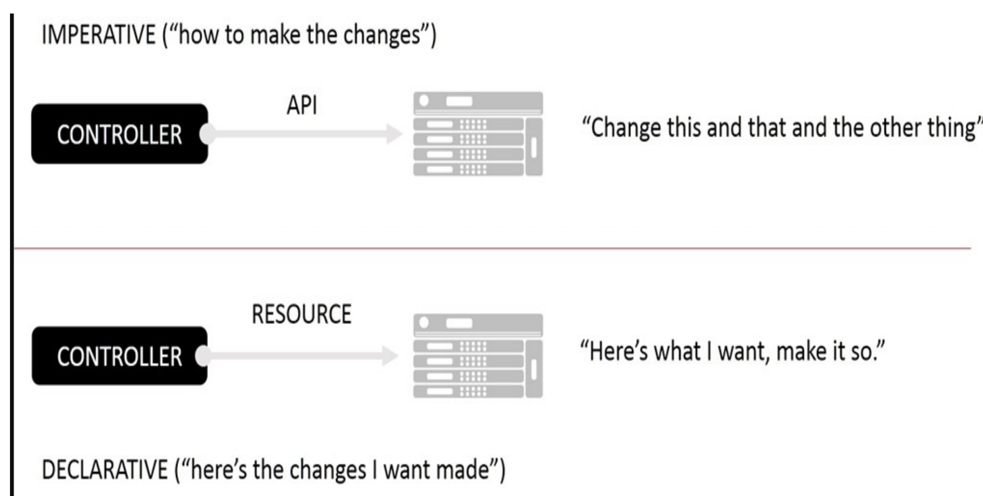

Рис. 2.3. Різниця між декларативним та імперативним API

Kubernetes пропонує ці два варіанти для задоволення потреб різних користувачів, так що ні зручність використання, ні гнучкість не порушуються.

Агреговані API - це підлеглі сервери API, які розташовані за основним сервером API, який діє як проксі. Це розташування називається API Aggregation (AA). CRD дозволяють користувачам створювати нові типи ресурсів без додавання іншого сервера API. Незалежно від того, як вони встановлені, нові ресурси називаються спеціальними ресурсами, щоб відрізнити їх від вбудованих ресурсів Kubernetes.

Ресурс API CustomResourceDefinition дозволяє визначити власні ресурси. API Kubernetes обслуговує та обробляє сховище власного ресурсу. Ім'я об'єкта CRD має бути дійсним іменем субдомену DNS. Це позбавляє від написання власного сервера API для обробки користувацького ресурсу, але загальний характер реалізації означає, що є менша гнучкість, ніж при агрегуванні серверів API. Зазвичай для кожного ресурсу в API Kubernetes потрібен код, який обробляє запити REST та керує постійним зберіганням об’єктів.

Основний сервер Kubernetes API обробляє вбудовані ресурси, такі як підсистеми та служби, а також може в цілому обробляти власні ресурси через CRD. Рівень агрегації дозволяє надавати спеціалізовані реалізації для власних ресурсів шляхом написання та розгортання власного автономного сервера API. Основний сервер API делегує запити на користувацькі ресурси, якими обробляються, роблячи їх доступними для всіх своїх клієнтів. CRD простіші у використанні. Агреговані API є більш гнучкими. Як правило, CRD добре підходять, якщо: є декілька подів, або використовується власний ресурс. CRD легше створювати, ніж агреговані API. Хоча створення CRD не додає автоматично жодних нових точок відмови (наприклад, викликаючи запуск стороннього коду на сервері API), пакети (наприклад, діаграми) або інші пакети інсталяції часто містять CRD, а також розгортання сторонній код, який реалізує бізнес-логіку для нового користувацького ресурсу.

Встановлення агрегованого сервера API завжди передбачає запуск нового розгортання. Спеціальні ресурси споживають місце для зберігання так само, як і

ConfigMaps. Створення великої кількості користувацьких ресурсів може перевантажити простір для зберігання сервера API. Агреговані сервери API можуть використовувати те саме сховище, що і основний сервер API. CRD завжди використовують ту саму аутентифікацію, авторизацію та журнал аудиту, що і вбудовані ресурси вашого сервера API.

Клієнтські бібліотеки Kubernetes можна використовувати для доступу до власних ресурсів, але не всі бібліотеки підтримують власні ресурси. Бібліотеки Go та Java це роблять. Додавши власний ресурс, є змога отримати до нього доступ за допомогою:

- Kubectl;
- Динамічного клієнту kubernetes;
- Власного клієнту REST.

2.4. Висновки до другого розділу

У другому розділі ми з'ясували, що в основі самого Kubernetes є великий набір контролерів, які гарантують, що конкретний ресурс знаходиться (і залишається) у бажаному стані, продиктованому заявленим визначенням. Якщо ресурс відхиляється від бажаного стану, контролер запускається для здійснення необхідних дій, щоб повернути стан ресурсу туди, де він повинен бути. Коли ми масштабуємо розгортання, ми фактично надсилаємо запит на сервер API із новою бажаною конфігурацією. Сервер API у відповідь публікує зміни для всіх своїх абонентів подій (будь-який компонент, який прослуховує зміни на сервері API). Таким чином, контролер розгортання створює один або кілька модулів подів для відповідності новому визначенню. Нове створення поду - це сама по собі нова зміна, яку сервер API також транслює до слухачів подій. Отже, якщо є якісь дії, які повинні ініціюватися при створенні нового поду, вони запускаються автоматично. Сервер API публікує лише нове визначення. Це не дає інструкцій контролеру або слухачам подій про те, як вони повинні діяти. Реалізація залишається за контролером. Хоча власних контролерів Kubernetes, таких як

Deployments, StatefulSets, Services, Job тощо, досить для самостійного вирішення більшості потреб додатків, в деяких випадках потрібно застосувати власний контролер.

Kubernetes дозволяє нам розширити та використовувати існуючу функціональність без необхідності ламати або змінювати вихідний код K8s. З моменту виникнення K8s контролери вважалися способом, яким розробники можуть розширити функціональність Kubernetes, надаючи нову поведінку. Підходи до налаштування можна розділити на конфігурацію, яка передбачає лише зміну прапорів, локальних файлів конфігурації або ресурсів API, та розширення, які передбачають запуск додаткових програм або послуг.

Хоча Golang зарекомендував себе як надійну мову програмування системи, він не є обов’язковим для використання при розробці користувацьких контролерів.

РОЗДІЛ 3
РОЗРОБКА ТА ОПТИМІЗАЦІЯ АВТОМАТИЧНОГО УПРАВЛІННЯ РОЗГОРНУТИМИ ВЕБ-ДОДАТКАМИ

3.1. Шаблон автоматизації оператор

Оператор - це патерн розробки для автоматизації роботи більш складних додатків або частин інфраструктури шляхом створення подів (контейнерів), які працюють у Kubernetes та мають вбудовану логіку управління. Ця логіка називається контуром управління. Цикл управління може бути реалізований за допомогою Java або будь-якої іншої мови програмування загального призначення, яка може спілкуватися з API K8s. Вхідними даними для циклу керування є об'єкти, створені на сервері Kubernetes API (рис. 3.1.), який підтримує власні схеми для цих об'єктів. Вони називаються спеціальними (кастомними) ресурсами.

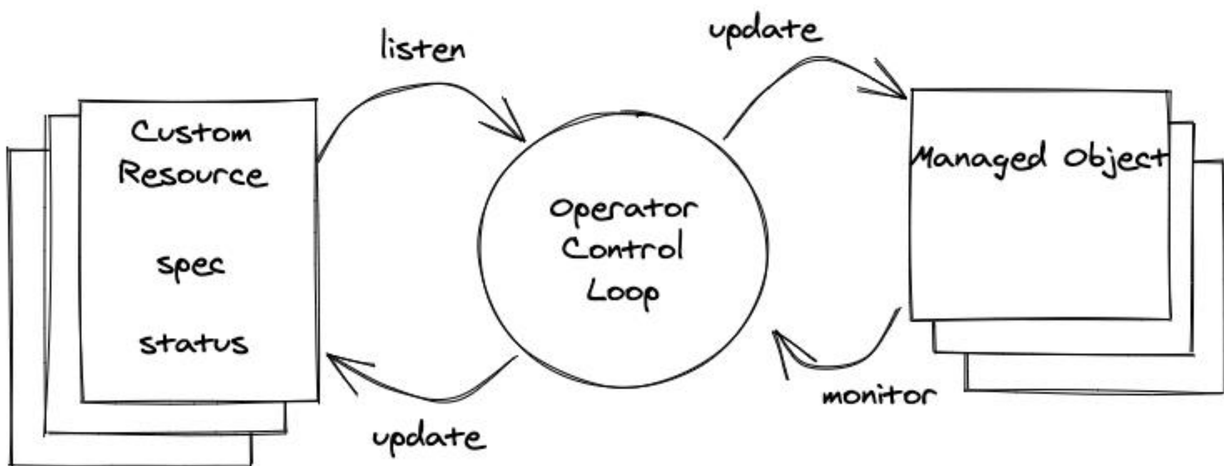

Рис. 3.1. Схема спеціальних ресурсів

Оператори можуть використовуватись для автоматизації у таких випадках:

- Розгортання додатку за вимогою;
- Отримання та відновлення резервних копій стану цього додатка;

- Обробка та оновлення коду програми разом із відповідними змінами, такими як схеми бази даних або додаткові налаштування конфігурації;
- Імітація повного або часткового збою у кластері, щоб перевірити його стійкість.

Найпоширеніший спосіб розгортання оператора - це додавання Custom Resource Definition та пов'язаного з ним контролера до кластера. Контролер, як правило, працює поза площиною управління, подібно до того, як запускається будь-яка контейнерна програма. Після розгортання оператора можливо використовувати його, додаючи, змінюючи або видаляючи тип ресурсу, який використовує оператор (рис. 3.2.). Якщо в екосистемі немає оператора, який реалізує потрібну поведінку, можливо створити свій власний.

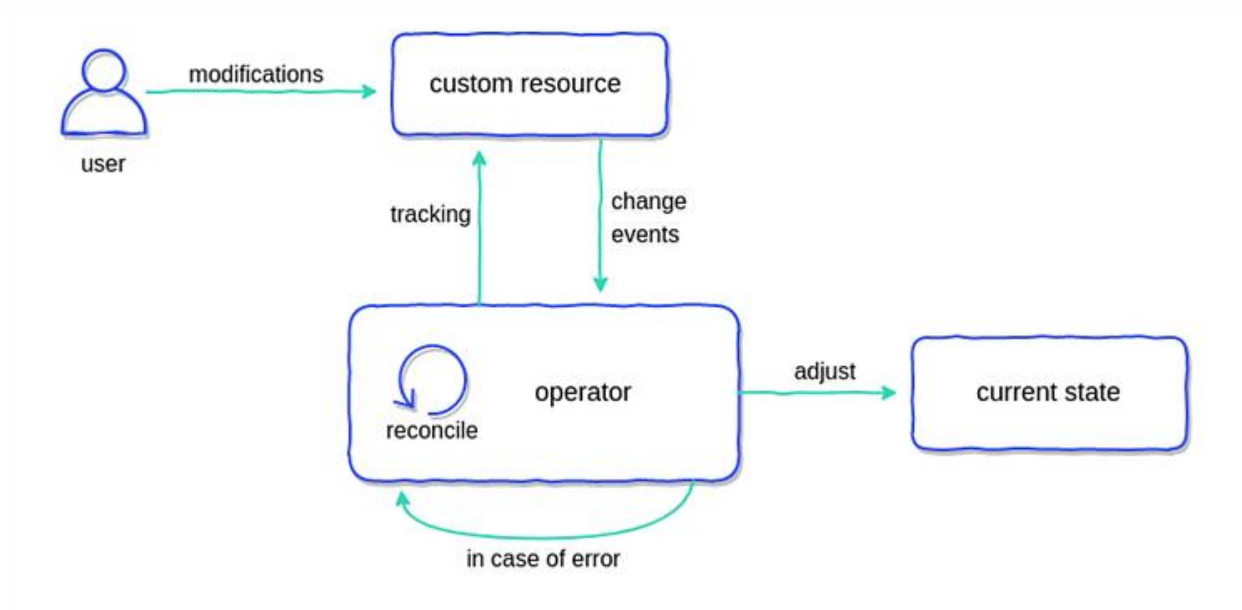

Рис. 3.2. Принцип роботи оператору

3.2. Розробка власного шаблону автоматизації роботи веб-додатків та власного ресурсу

Типовим сценарієм автоматизації інфраструктури є надання баз даних для додатків. Ми будемо надавати бази даних MySQL. Індивідуальні бази даних на

сервері MySQL називаються схемами. Для того, щоб не створювати новий сервер баз даних для кожної програми, а створювати нову схему необхідно:

1. Створити нове визначення спеціальних ресурсів (Custom Resource Definition) під назвою MySQLSchema, щоб дозволити користувачам Kubernetes створювати нові спеціальні ресурси цього типу;
2. Впровадити логіку, яка буде спостерігати за API Kubernetes для нових об'єктів MySQLSchema, і створити фактичну схему в MySQL, або видалити її за потреби.

Створеному MySQLSchemaOperator потрібно буде зробити наступне:

- Прослухати зміни будь-яких ресурсів MySQLSchema із сервера API, використовуючи Fabric8;
- Зв'язати ресурси MySQLSchema із набором класів Java;
- Узгодити стан ресурсів MySQLSchema з фактичним станом у кластері баз даних MySQL. Якщо в Kubernetes існує ресурс з назвою «mydb», оператор повинен переконатися, що створена схема;
- Оновити спеціальний ресурс URL-адресою новоствореної схеми бази даних, щоб вказати на успішне створення схеми;
- Створити ConfigMap та Secret із URL-адресою та обліковими даними для доступу до бази даних. Потім вони можуть бути замаплені з додатком, який використовує базу даних як змінні середовища або томи.

Використання fabric8 зумовлено тим, що цей клієнт забезпечує доступ до главних API-інтерфейсів K8s через вільний DSL та має плагін Maven для запуску та створення образів Docker.

Файл MySQLSchema має наступний вигляд:

```
apiVersion: "mysql.sample.javaoperatorsdk/v1"
kind: MySQLSchema
metadata:
 name: mydb
spec:
 encoding: utf8
//
```

Основним компонентом java-operator-sdk є оператор-фреймворк. Оператор-фреймворк:

- Обгортає fabric8 та налаштовує його для прослуховування змін у зазначених спеціальних ресурсах, таким чином приховуючи шаблонний код, необхідний для цього;
- Забезпечує чистий інтерфейс для реалізації циклу узгодження для певного типу ресурсу;
- Графіки змінюють події, які повинні виконуватися ефективно. Фільтрування застарілих подій та паралельне виконання непов'язаних подій.

Клієнт fabric8 буде обробляти зв'язок з API Kubernetes. Фреймворк оператора обробляє управління подіями, отриманими через fabric8, і викликає логіку контролера. Це серце оператора, де буде жити бізнес-логіка забезпечення MySQLSchema. Наша логіка, в свою чергу, використовуватиме бібліотеку mysql-connector для спілкування з сервером MySQL (рис. 3.3.).

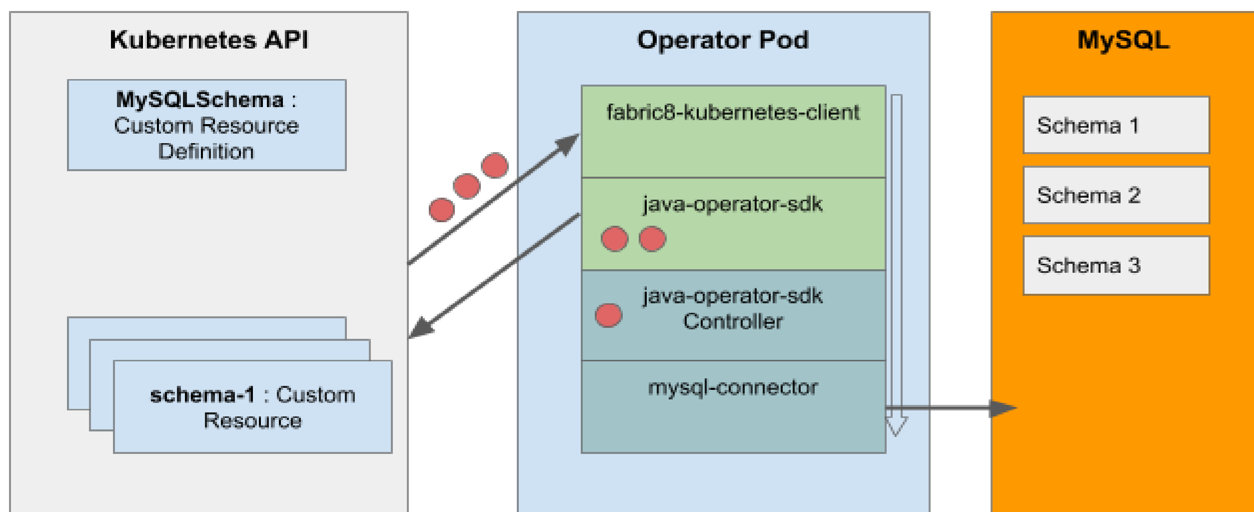

Рис. 3.3. Зв'язок Kubernetes API та MySQL

Спершу необхідно зв'язати Kubernetes з нашим спеціальним ресурсом. Це робиться за допомогою спеціального визначення ресурсів (CRD)

```
//
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
 name: schemas.mysql.sample.javaoperatorsdk
 spec:
  group: mysql.sample.javaoperatorsdk
  version: v1
  scope: Namespaced
```

```
  names:
   plural: schemas
   singular: schema
   kind: MySQLSchema
  validation:
   openAPIV3Schema:
     type: object
     properties:
       spec:
         type: object
         properties:
           encoding:
             type: string
//
```

Kubernetes дбає лише про те, щоб ім'я було зареєстровано. Ми зареєструємо CRD у Kubernetes, використовуючи просту команду apply із файлом yaml, що містить зазначене вище визначення:

kubectl apply -f crd.yaml

Дані з спеціальних ресурсів необхідно відобразити на об’єкти цих Java-класів. Для цього Fabric8 використовує бібліотеку зіставлення об'єктів у фоновому режимі. Для цього ми взаємодіємо з бібліотекою fabric8, розширюючи клас CustomResource. Для реалізації циклу управління необхідно реалізувати інтерфейс ResourceController, який має лише два методи:

```
//
public interface ResourceController<R extends CustomResource> {
   UpdateControl createOrUpdateResource(R resource, Context<R> context);
   boolean deleteResource(R resource);
}
//
```

Завданням операторської структури є виклик цих методів щоразу, коли створюється, оновлюється або видаляється спеціальний об’єкт ресурсу в API Kubernetes. Ця логіка полягає в управлінні MySQL, використовуючи поточну версію спеціального ресурсу схеми як вхід. У цьому полягає сенс Java-Operator-SDK: дозволяючи нам зосередитися на бізнес-логіці спеціального оператора. Коли спеціальний ресурс буде створено або оновлено, оператор:

- Перевірить, чи вже існує схема бази даних;
- Створить схему, якщо вона не існує в базі даних Mysql;
- Оновить поле стану спеціального ресурсу за допомогою URL-адреси бази даних і встановить статус на СТВОРЕНО;

- Створить ConfigMap і Secret, які можна підключити до програми за допомогою бази даних.

Важливим є те, що ми не діємо залежно від типу події (створення / оновлення). Ми навіть не отримуємо цю інформацію як вхід, тому замість цього ми завжди перевіряємо, чи створена схема у Mysql. Це важливий момент у підході оператора до забезпечення: необхідно завжди перевіряти реальний стан спеціальних ресурсів. Оператори мають справу зі станом, який важко контролювати і який може бути змінений різними процесами, тому їм доводиться робити дуже мало припущень, щоб бути надійними. У нашому випадку ми перевіряємо, чи існує схема, і створюємо її, якщо ні. Видалення ресурсів обробляється іншим способом. Воно обробляється особливим чином, використовуючи фіналізатори у фоновому режимі.

Після того, як клас контролера готовий, потрібно ініціалізувати його: створити об'єкт оператора та додати до нього будь-які контролери. Зазвичай запускається один контролер в операторі, але можливо кілька з них упакувати в один процес Java Virtual Machine (JVM).

Для побудови та розгортання нашого оператора на Kubernetes нам потрібно виконати наступні кроки:

- Налаштувати властивість реєстру docker у ~ / .m2 / settings.xml;
- Створити образ Docker і перенести його до віддаленого реєстру: mvn package dockerfile:build dockerfile:push;
- Створити CustomResourceDefinition на кластері: kubectl apply -f crd.yaml;
- Створити RBAC об’єкти: kubectl apply -f rbac.yaml;
- Розгорнути оператор: kubectl apply -f operator-deployment.yaml

Для створення образу Docker, використано плагін dockerfile-maven-plugin.

```
//
<plugin>
   <groupId>com.spotify</groupId>
   <artifactId>dockerfile-maven-plugin</artifactId>
   <version>1.4.12</version>
   <configuration>
       <repository>${docker-registry}/mysql-schema-operator</repository>
```

```
        <tag>${project.version}</tag>
        <buildArgs>
            <JAR_FILE>${project.build.finalName}.jar</JAR_FILE>
        </buildArgs>
    </configuration>
</plugin>
//
```

Docker-image файл оператора:

```
//
FROM openjdk:12-alpine
ENTRYPOINT ["java", "-jar", "/usr/share/operator/operator.jar"]
ARG JAR_FILE
ADD target/${JAR_FILE} /usr/share/operator/operator.jar
//
```

Ця збірка також передбачає, що артефакт, створений пакетом maven, є виконуваним jar-файлом. Наш оператор буде розроблений як звичайне розгортання на Kubernetes. Це гарантує, що завжди буде запущено єдиний екземпляр оператора. Стратегія оновлення «повторне створення» гарантує, що стара версія буде відключена, перш ніж нова версія буде запущена під час оновлення.

```
//
apiVersion: v1
kind: Namespace
metadata:
  name: mysql-schema-operator
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql-schema-operator
  namespace: mysql-schema-operator
spec:
  selector:
    matchLabels:
      app: mysql-schema-operator
  replicas: 1 # ми завжди запускаємо лише одну репліку оператору для уникнення дублікату обробки подій
  strategy:
    type: Recreate # на протязі апгрейду оператор буде знищено до того, як новая версія вийде, щоб не допустити розгортання двох об’єктів одночасно
    metadata:
      labels:
        app: mysql-schema-operator
    spec:
      serviceAccount: mysql-schema-operator # вказано ServiceAccount під яким дозволами RBAC буде виконуватись оператор
      containers:
      - name: operator
        image: ${DOCKER_REGISTRY}/mysql-schema-operator:${OPERATOR_VERSION}
        imagePullPolicy: Always
        ports:
        - containerPort: 80
```

```
        env:
        - name: MYSQL_HOST
          value: mysql.mysql # припускаємо що сервер MySQL працює в просторі
імен під назвою "mysql" на Kubernetes
        - name: MYSQL_USER
          value: root
        - name: MYSQL_PASSWORD
          value: password
        readinessProbe:
          httpGet:
            path: / #перевірка статусу  додатку, коли повертає 200 оператор
працює без помилок
          initialDelaySeconds: 1
          timeoutSeconds: 1
        livenessProbe:
          httpGet:
            path: /health # коли ендпоінт не повертає 200, оператор не працює та
примусово перезапускається
            port: 8080
          initialDelaySeconds: 30
          timeoutSeconds: 1
//
```

Підключення до сервера баз даних MySQL налаштовується за допомогою змінних середовища. Ім'я хоста ‘mysql.mysql’ відноситься до бази даних Mysql, що працює на Kubernetes у просторі імен mysql. MySQL буде призначено лише ефемерне сховище, тому воно втратить усі дані, коли модуль буде знищено.

kubectl apply -f k8s/mysql.yaml

Більшість програм, що працюють на Kubernetes, не потребують доступу до API Kubernetes. Однак оператори - потребують, оскільки вони працюють із ресурсами, визначеними та оновленими в API.

Ресурс ClusterRole визначає групу дозволів. Він надає доступ до виконання всіх операцій над ресурсами MySQLSchema, а також до переліку та отримання CustomResourceDefinitions. Оператору потрібно буде спостерігати за всіма ресурсами схеми кластера, а також оновлювати їх при зміні стану. Що стосується CustomResourceDefinitions, оператору потрібно отримати власний CRD при запуску, щоб отримати деякі метадані:

```
//
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
 name: mysql-schema-operator
rules:
- apiGroups:
 - mysql.sample.javaoperatorsdk
 resources:
```

```
 - schemas
 verbs:
 - "*"
- apiGroups:
 - apiextensions.k8s.io
 resources:
 - customresourcedefinitions
 verbs:
 - "get"
 - "list"
//
```

Це ServiceAccount, під капотом котрого буде виконуватися оператор. Це просто назва, яка пов’язує запущений модуль оператора з дозволами, які він має на сервері API.

```
//
apiVersion: v1
kind: ServiceAccount
metadata:
 name: mysql-schema-operator
 namespace: mysql-schema-operator
//
```

Схема, що пояснює взаємозв'язок між різними об'єктами RBAC представлена на рисунку 3.4.

ClusterRoleBinding підключає ServiceAccount до ClusterRole. Таким чином, будь-який под, що працює під оператором mysql-schema-ServiceAccount, прийматиме ClusterRole з тим самим іменем та дозволами, описаними вище.

```
//
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: operator-admin
subjects:
- kind: ServiceAccount
 name: mysql-schema-operator
 namespace: mysql-schema-operator
roleRef:
 kind: ClusterRole
 name: mysql-schema-operator
 apiGroup: ""
//
```

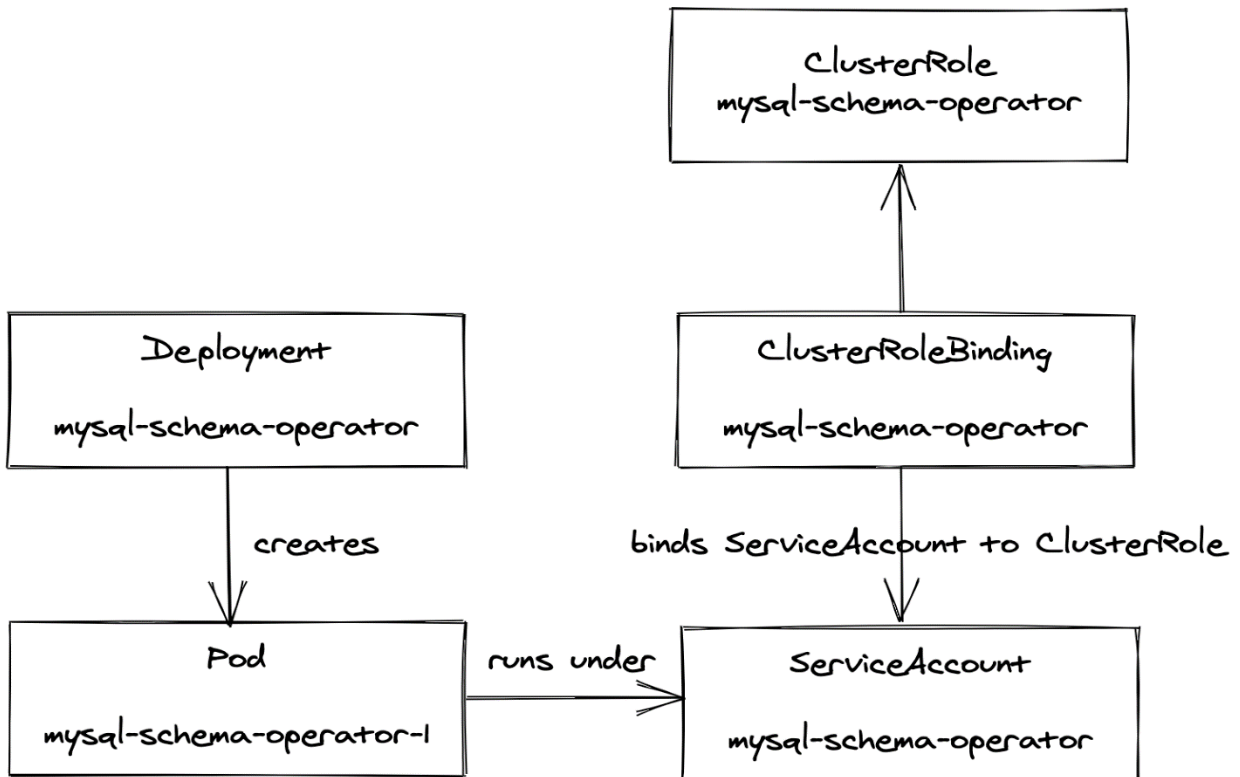

Рис. 3.4. Взаємозв'язок між компонентами RBAC

ClusterRoleBinding підключає ServiceAccount до ClusterRole. Таким чином, будь-який под, що працює під оператором mysql-schema-ServiceAccount, прийматиме ClusterRole з тим самим іменем та дозволами, описаними вище.

Для розробки власного оператора та власних ресурсів спочатку необхідно пакувати Java-код у jar-файл (рис. 3.5.):

mvn clean package

Для подальшої розробки необхідно створити Docker image нашого Dockerfile (рис. 3.6.):

docker build .

```
[WARNING]   - META-INF/NOTICE
[WARNING] jakarta.xml.bind-api-2.3.2.jar, jaxb-api-2.3.0.jar define 114 overlapping classes and resources:
[WARNING]   - META-INF.versions.9.javax.xml.bind.ModuleUtil
[WARNING]   - javax.xml.bind.Binder
[WARNING]   - javax.xml.bind.ContextFinder
[WARNING]   - javax.xml.bind.ContextFinder$1
[WARNING]   - javax.xml.bind.ContextFinder$2
[WARNING]   - javax.xml.bind.ContextFinder$3
[WARNING]   - javax.xml.bind.ContextFinder$4
[WARNING]   - javax.xml.bind.ContextFinder$5
[WARNING]   - javax.xml.bind.DataBindingException
[WARNING]   - javax.xml.bind.DatatypeConverter
[WARNING]   - 104 more...
[WARNING] jakarta.activation-api-1.2.1.jar, jakarta.xml.bind-api-2.3.2.jar define 2 overlapping resources:
[WARNING]   - META-INF/LICENSE.md
[WARNING]   - META-INF/NOTICE.md
[WARNING] commons-lang3-3.5.jar, commons-text-1.4.jar, javax.annotation-api-1.3.2.jar, jaxb-api-2.3.0.jar define 1 overlappi
[WARNING]   - META-INF/LICENSE.txt
[WARNING] log4j-api-2.13.3.jar, log4j-core-2.13.3.jar, log4j-slf4j-impl-2.13.3.jar define 1 overlapping resource:
[WARNING]   - META-INF/DEPENDENCIES
[WARNING] commons-lang3-3.5.jar, commons-text-1.4.jar define 1 overlapping resource:
[WARNING]   - META-INF/NOTICE.txt
[WARNING] maven-shade-plugin has detected that some class files are
[WARNING] present in two or more JARs. When this happens, only one
[WARNING] single version of the class is copied to the uber jar.
[WARNING] Usually this is not harmful and you can skip these warnings,
[WARNING] otherwise try to manually exclude artifacts based on
[WARNING] mvn dependency:tree -Ddetail=true and the above output.
[WARNING] See http://maven.apache.org/plugins/maven-shade-plugin/
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing C:\Users\Denis\Desktop\java-operator-sdk-master\java-operator-sdk-master\samples\mysql-schema\target\mysql-
ysql-schema-sample-1.4.1-SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.178 s
[INFO] Finished at: 2020-12-12T19:46:28+02:00
[INFO] ------------------------------------------------------------------------
```

Рис. 3.5. Створення jar-файлу додатку

Для подальшої розробки необхідно зареєструвати акаунт на Dockerhub та відіслати свій image-образ у свій приватний репозиторій (рис. 3.7.), щоб потім K8s мав змогу скачати цей образ з хабу (рис. 3.8.) та продовжити роботу.

Для цього виповнимо ряд команд:

docker login

docker push image name

```
[+] Building 0.1s (2/2) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 32B
failed to solve with frontend dockerfile.v0: failed to create LLB definition: Dockerfile parse error line 1: unknown instruction: DFROM

C:\Users\Denis\Desktop\java-operator-sdk-master\java-operator-sdk-master\samples\mysql-schema>docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
social_webapp       latest              48588bdf02d0        3 weeks ago         490MB
postgres            latest              f51c55ac75ed        3 weeks ago         314MB

C:\Users\Denis\Desktop\java-operator-sdk-master\java-operator-sdk-master\samples\mysql-schema>docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
social_webapp       latest              48588bdf02d0        3 weeks ago         490MB
postgres            latest              f51c55ac75ed        3 weeks ago         314MB

C:\Users\Denis\Desktop\java-operator-sdk-master\java-operator-sdk-master\samples\mysql-schema>docker build .
[+] Building 24.5s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 198B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/openjdk:12-alpine
 => [internal] load build context
 => => transferring context: 31.89MB
 => [1/2] FROM docker.io/library/openjdk:12-alpine@sha256:fecd532eaee349b4d9e329148e99de77ffaf803e66e184a0e4d6b946bb97ffa3
 => => resolve docker.io/library/openjdk:12-alpine@sha256:fecd532eaee349b4d9e329148e99de77ffaf803e66e184a0e4d6b946bb97ffa3
 => => sha256:9716b977a99b5983f66063ce42eaa529af94f6265b6908558041581c3ae5b4ac 197.66MB / 197.66MB
 => => sha256:fecd532eaee349b4d9e329148e99de77ffaf803e66e184a0e4d6b946bb97ffa3 433B / 433B
 => => sha256:37b8b402893091d9120401a3d9c87d81a6fa967d96ca81e06a80d01605845c79 741B / 741B
 => => sha256:0c68e7c5b7a0cb1612ea7b14c460d1f165ae7250b8aa7a0e5e53ae6cdc846310 3.44kB / 3.44kB
 => => sha256:6c40cc604d8e4c121adcb6b0bfe8bb038815c350980090e74aa5a6423f8f82c0 2.75MB / 2.75MB
 => => extracting sha256:6c40cc604d8e4c121adcb6b0bfe8bb038815c350980090e74aa5a6423f8f82c0
 => => extracting sha256:9716b977a99b5983f66063ce42eaa529af94f6265b6908558041581c3ae5b4ac
 => [2/2] ADD target/ /usr/share/operator/operator.jar
 => exporting to image
 => => exporting layers
 => => writing image sha256:9e03855c12ef5d9783344520b0dbac184ee7e413c580b6b5b36d2a9f9efb2d36
```

Рис. 3.6. Створення image-образу Dockerfile

```
C:\Users\Denis\Desktop\java-operator-sdk-master\java-operator-sdk-master\samples\mysql-schema>docker push 14882281488228322/trofimenko:latest
The push refers to repository [docker.io/14882281488228322/trofimenko]
3cbd44dfbcaa: Pushed
e10fdc20c652: Mounted from library/openjdk
503e53e365f3: Mounted from library/openjdk
latest: digest: sha256:86acf14ed3eb4e49414e4471239e69bfbd962033a9f721c867ed15c5b4c1383c size: 953
```

Рис. 3.7. Відправка image-образу на DockerHub

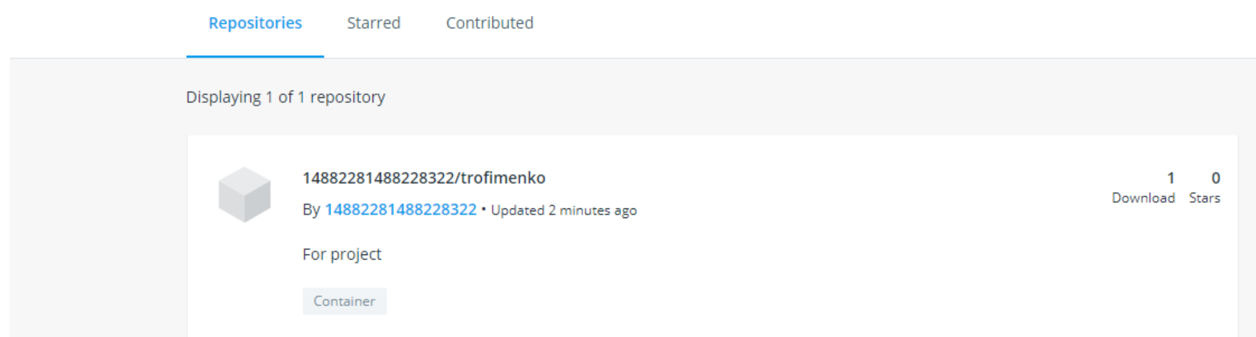

Рис. 3.8. Образ у приватному репозиторію

Після розгортання перевіримо, чи працює оператор:

kubectl get pods -n mysql-schema-operator

Ми побачимо, що один под працює та готовий до роботи (рис. 3.9).

```
→  mysql-schema git:(master) ✗ kubectl get pods -n mysql-schema-operator
NAME                                     READY   STATUS    RESTARTS   AGE
mysql-schema-operator-957bd9d6d-h9tqn   1/1     Running   0          42h
```

Рис. 3.8. Розгорнутий працюючий оператор

Після цього ми можемо створити спеціальний ресурс MySQLSchema, застосувавши відповідний файл yaml. Для встановлення назви схеми його необхідно редагувати:

kubectl apply -f k8s/example.yaml

Після створення ресурсу MySQLSchema оператор повинен підключити сервер MySQL і створити справжню схему та користувача бази даних, який може отримати до неї доступ. Він повинен створити секрет Kubernetes з обліковими даними для доступу до бази даних, яку ми можемо включити в розгортання програми.

Результатом розробки став гнучкий та повністю автоматизований процес надання баз даних. Ми показали, як реалізувати такий процес, використовуючи шаблон оператора на Kubernetes та логіку на Java.

3.3. Висновки до третього розділу

Метою даного розділу була розробка та оптимізація методів та алгоритмів автоматичного управління розгорнутими контейнерізованими веб-додатками та сервісами. Ми досягли цього, розробивши патерн оператор на Java, який виступає аналогом контролеру у Kubernetes, та успадковується платформою K8s і вдало інтегрується у неї. Оператор виповнює команди, які направлені на підтримку бажаного стану об’єктів, він слідкує за ними увесь час та намагається виправити ситуацію, у якій стан об’єктів виходить із бажаного стану. Підтримка бажаного стану виповнюється за рахунок базових команд створення, оновлення

та знищення. Оператори - це програмні розширення Kubernetes, які використовують власні ресурси для управління програмами та їх компонентами. Оператори слідують принципам K8s, зокрема контуру управління.

Оператор слідкує та підтримує бажаний стан власних ресурсів, які було створено із Custom Resource Definition. Власні ресурси - це розширення API Kubernetes, які забезпечує місце, де ми можемо зберігати та отримувати структуровані дані - бажаний стан нашої програми. Оператор постійно відстежує події кластера, що стосуються певного типу користувацьких ресурсів. Коли оператор отримує будь-яку інформацію, він вживає заходів для регулювання кластера Kubernetes або зовнішньої системи до бажаного стану як частини свого циклу узгодження в користувацькому контролері. Спеціальний ресурс розширює можливості K8s, додаючи нові види об’єктів, які можуть бути корисними під час автоматизації розгортання веб-додатків та сервісів.

РОЗДІЛ 4
ЕКОНОМІЧНИЙ РОЗДІЛ

4.1. Визначення трудомісткості розробки програмного забезпечення

Початкові дані:

1. передбачуване число операторів програми – 1900;
2. коефіцієнт складності програми – 1,9;
3. коефіцієнт корекції програми в ході її розробки – 0,08;
4. годинна заробітна плата програміста– 80грн/год;
5. коефіцієнт збільшення витрат праці в наслідок недостатнього опису задачі – 1,2;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,0;
7. вартість машино-години ЕОМ –15 грн/год.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д \text{, людино-годин,} \quad (4.1)$$

де to- витрати праці на підготовку й опис поставленої задачі (приймається 70 людино-годин);

tи - витрати праці на дослідження алгоритму рішення задачі;

ta- витрати праці на розробку блок-схеми алгоритму;

tп-витрати праці на програмування по готовій блок-схемі;

tотл-витрати праці на налагодження програми на ЕОМ;

tд - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у програмному забезпеченні, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p),$$

де q - передбачуване число операторів (1900);

c - коефіцієнт складності програми (1,9);

p - коефіцієнт корекції програми в ході її розробки (0,08).

Звідси умовне число операторів в програмі:

$$Q = 1,9 \cdot 1900 \cdot (1 + 0,08) = 3898,8 \text{ людино-годин},$$

Витрати праці на вивчення опису задачі t визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75..85) \cdot k}, \text{ людино-годин},$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності. При стажі роботи від 2 до 3 років він складає 1,0.

Приймемо збільшення витрат праці внаслідок недостатнього опису завдання не більше 50% (B = 1,2). З урахуванням коефіцієнта кваліфікації k = 1,0, отримуємо витрати праці на вивчення опису завдання:

$$t_u = (3898,8 \cdot 1,2) / (80 \cdot 1,0) = 58,48 \text{ людино-годин}$$

Витрати праці на розробку алгоритму рішення задачі визначаються за формулою:

$$t_a = \frac{Q}{(20\ldots25)\cdot k}$$, людино-годин, (4.2)

де Q – умовне число операторів програми;

k – коефіцієнт кваліфікації програміста.

Підставивши відповідні значення в формулу (4.2), отримаємо:

t_a= 3898,8 / (22 · 1,0) = 177,2 людино-годин.

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20\ldots25)\cdot k}$$, людино-годин.

t_n = 3898,8 / (22 · 1,0) = 177,2 людино-годин.

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5)\cdot k}$$, людино-годин.

$t_{отл}$ = 3898,8 / (4 · 1,0) = 974,7 людино-годин.

- за умови комплексного налагодження завдання:

$$t^k_{отл} = 1{,}5\cdot t_{отл}$$, людино-годин.

$t^{k}_{отл}$= 1,5 · 974,7 = 1462 людино-годин.

Витрати праці на підготовку документації визначаються за формулою:

$$t_{д} = t_{др} + t_{до}$$, людино-годин,

де tдр-трудомісткість підготовки матеріалів і рукопису:

$$t_{др} = \frac{Q}{(15..20) \cdot k}$$, людино-годин,

tдо - трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0,75 \cdot t_{др}$$, людино-годин.

Підставляючи відповідні значення, отримаємо:

$t_{др}$ = 3898,8 / (17 · 1,0) = 229,34 людино-годин.

$t_{до}$ = 0,75 · 229,34 = 172 людино-годин.

$t_{д}$ = 229,34 + 172 = 401,34 людино-годин.

Повертаючись до формули (4.1), отримаємо повну оцінку трудомісткості розробки програмного забезпечення:

t = 70 + 58,28 + 177,2 + 177,2 + 974,7 + 401,34 = 1858,72 людино-годин.

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ $К_{ПО}$ включають витрати на заробітну плату виконавця програми $З_{ЗП}$ і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн.}$$

Заробітна плата виконавців визначається за формулою:

$$З_{ЗП} = t \cdot C_{ПР}, \text{ грн,}$$

де: t - загальна трудомісткість, людино-годин;

СПР - середня годинна заробітна плата програміста, грн/година

З урахуванням того, що середня годинна зарплата програміста становить 60 грн / год, отримуємо:

$$З_{ЗП} = 1858{,}72 \cdot 80 = 148697{,}6 \text{ грн.}$$

Вартість машинного часу, необхідного для налагодження програми на ЕОМ, визначається за формулою:

$$З_{мв} = t_{отл} \cdot C_{мч}, \text{ грн,} \quad (4.3)$$

де tотл - трудомісткість налагодження програми на ЕОМ, год;

Смч - вартість машино-години ЕОМ, грн/год (15 грн/год).

Підставивши в формулу (4.3) відповідні значення, визначимо вартість необхідного для налагодження машинного часу:

$$З_{мв} = 974{,}7 \cdot 15 = 14620{,}5 \text{ грн.}$$

Звідси витрати на створення програмного продукту:

$$K_{ПО} = 148697{,}6 + 14620{,}5 = 163318{,}1 \text{ грн.}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}, \text{ міс.}$$

де Bk- число виконавців (дорівнює 1);

Fp - місячний фонд робочого часу (при 40 годинному робочому тижні Fp=176 годин).

Звідси витрати на створення програмного продукту:

$$T = 1858{,}72 / (1 \cdot 176) \approx 10{,}5 \text{ міс.}$$

4.3. Маркетингові дослідження ринку збуту розробленого програмного продукту

Використання алгоритмів контейнерізації додатків - це один з найбільш динамічно зростаючіх сегментів IT-ринку на сьогодні, який орієнтовано на активне корпоративне використання багатьох послуг. Один з методів цього процесу - розробка системних оркестрацій контейнерів, яка використовує функції запуску, налаштування та адміністрування цих інструментів віртуалізації.

Розроблене програмне забезпечення та його методи і алгоритми можуть бути використані в багатьох системах та для безлічі задач. Оскільки розроблене програмне забезпечення розширює функціонал Kubernetes, його можуть використовувати мільйони користувачів та розробників. Розроблене програмне забезпечення можна використовувати у таких областях як інформаційні технології та сервіси, фінансові послуги, медицинські послуги, банківські та телекомунікаційні послуги та інші. Програмне забезпечення платформи вже використовують більш ніж у 20000 компаній, найчастіше використовується невеликими компаніями з 50-200 співробітниками та доходами від 1 до 10 мільйонів доларів. Такі компанії найчастіше зустрічаються у США та в галузі комп'ютерного програмного забезпечення.

Згідно з офіційною статистикою, компанії у таких країнах як США, Великобританія, Канада, Германія, Індія та Франція найчастіше використовують методи контейнеризації під час своїх розробок та досліджень. Розроблені методи та алгоритми програмного забезпечення вирішують та надають повністю автоматизований процес надання баз даних, що є дуже необхідним для будь-якої сучасної компанії, що має кілька команд розробників.

Серед конкурентів розробленого програмного забезпечення можна виділити такі інструменти як: VMware vCenter, VMware Horizon, VMware vMotion, Red Hat OpenShift, Citrix Hypervisor, Microsoft System Center Virtual Machine Manager, Pivot3, Docker Swarm, Shippable, AppSense.

За результатами дослідження, сьогодні контейнери вже отримали визнання у 31% корпоративного ринку. Згідно з прогнозами, світовий обсяг продажів контейнерів найближчим часом буде продовжено. Якщо в 2016 році він оцінювався в розмірі 762 мільйонів доларів, то до 2022 року має досягти рівня 4 мільярдів доларів, що відповідає середньорічному темпу зростання на рівні 40%. Втім, незважаючи на таку активну динаміку зростання, цей сегмент поки ще залишається порівняно малим в порівнянні із загальним ринком хмарних послуг, який враховує і інші напрямки, такі як системи віртуалізації, приватні PaaS, засоби автоматизації і управління хмарними послугами.

Більш ніж 2225 компаній вже використовують послуги платформи, яку було поширено та удосконалено під час дипломної роботи. Такі компанії як Google, Shopify, Slack, Robinhood, Delivery Hero, Stack, Nubank та інші вже використовують платформу у своїх технологічних стеках.

Серед переваг розробленого програмного забезпечення над аналогами можна виділити найбільший та найпотужніший функціонал, потужну відказоустійкість продукту, безкоштовну повну версію, покращення контейнерізації та пакування додатків. Кожен контейнер є повторюваним, стандартизація від включення залежностей означає, що отримується однакова поведінка у будь якому середовищі, де додаток розгортається. Контейнери відокремлюють програми від базової інфраструктури хоста. Це полегшує розгортання в різних середовищах хмари або ОС. Найкращі налаштування для роботи з робочими навантаженнями. Користувачу не потрібно керувати кожним елементом безпосередньо. Натомість є можливість використовувати ресурси робочого навантаження, які будуть керувати набором елементів від імені користувача. Ці ресурси налаштовують контролери, які переконуються, що запущено потрібну кількість потрібних видів елементів, щоб відповідати вказаному стану.

4.4. Оцінка економічної ефективності впровадження програмного забезпечення

Як було зазначено вище, у ході дослідження було оптимізовано методи та алгорими для особистого використання, тож дане програмне забезпечення не було призначено для впровадження на підприємстві. Отже, ми не можемо обчислити економічну ефективність цього продукту, але ми можемо визначити його соціальний ефект.

Серед соціальних ефектів розробленого програмного забезпечення можна виділити наступні:

- Покращений моніторинг сервісів та розподілення навантажень;

- Оркестрація сховищ. Автоматична зміна системи зберігання за вибором;
- Автоматичне розвернення та відкати;
- Вдосконалене втоматичне розподілення навантажень;
- Самоконтроль. ПЗ перезапускає контейнери, замінює та завершує роботу контейнерів;
- Управління конфіденційною інформацією та конфігурацією.

ВИСНОВКИ

У процесі дослідження були отримані наступні результати. При вивченні принципів автоматизації управління розгорнутими веб-додатками було розглянуто можливості платформи Kubernetes, де основним визначальним архітектурним параметром є взаємодія двигуну Kubernetes та контейнерізованих додатків. Додатки пакуються у контейнери разом з усіма необхідними інструментами та залежностями за домомогою програми Docker.

Серед основних можливостей платформи K8s можна виділити:

1. Автоматизація розгортання додактів;
2. Автоматизація масштабування додатків;
3. Автоматизація управління додатками.

Серед переваг використання Kubernetes для автоматизації управління розгорнутими веб-додатками можна виділити наступні функції:

4. Виявлення та балансування навантаження;
5. Оркестрація усіх видів сховищ;
6. Автоматичне розгортання та відкати;
7. Автоматичне відновлення та рестарт контейнерів, заміна новими контейнерами пошкоджених;
8. Управління секретами та конфігурацією.

Для визначення ефективних способів автоматизації управління розгорнутими веб-додатками було запропоновано розбивати задачу на наступні етапи: аналіз, дизайн, реалізація, налагодження та тестування.

Було визначено, що для оптимізації управління розгорнутими контейнерізованими веб-додатками необхідно:

1. Розробка власного оператора, який буде виконувати роль контролера та стежити за користувацькими ресурсами.

2. Створення власних (користувацьких) ресурсів, які будуть у повній мірі описувати додаток та будуть керуватися оператором.

3. Інтегрування користувацьких ресурсів та контролеру з платформою Kubernetes.

Метою роботи була розробка та оптимізація методів та алгоритмів автоматичного управління розгорнутими контейнерізованими веб-додатками та сервісами. Ми досягли цього, розробивши патерн оператор на Java, який виступає аналогом контролеру у Kubernetes, та успадковується платформою K8s і вдало інтегрується у неї. Оператор виповнює команди, які направлені на підтримку бажаного стану об'єктів, він слідкує за ними увесь час та намагається виправити ситуацію, у якій стан об'єктів виходить із бажаного стану. Підтримка бажаного стану виповнюється за рахунок базових команд створення, оновлення та знищення. Оператори - це програмні розширення Kubernetes, які використовують власні ресурси для управління програмами та їх компонентами. Оператори слідують принципам K8s, зокрема контуру управління.

Оператор слідкує та підтримує бажаний стан власних ресурсів, які було створено із Custom Resource Definition. Власні ресурси - це розширення API Kubernetes, які забезпечує місце, де ми можемо зберігати та отримувати структуровані дані - бажаний стан нашої програми. Оператор постійно відстежує події кластера, що стосуються певного типу користувацьких ресурсів. Коли оператор отримує будь-яку інформацію, він вживає заходів для регулювання кластера Kubernetes або зовнішньої системи до бажаного стану як частини свого циклу узгодження в користувацькому контролері. Спеціальний ресурс розширює можливості K8s, додаючи нові види об'єктів.

З огляду на основне призначення даної роботи – розробка та оптимізація методів та алгоритмів автоматичного управління розгорнутими контейнерізованими веб-додатками та сервісами, – ми вибрали автоматизацію управління базами даних у розгорнутому веб-додатку для демонстрації розроблених методів.

Наш оператор було розміщено як розгортання на Kubernetes. Це забезпечило постійно запущений єдиний екземпляр оператора. Стратегія

оновлення «відтворити» забезпечує попереднє вимкнення старої версії перед запуском нової версії під час оновлення.

З зростаючою кількістю веб-додатків, складністю їх архітектури та сервісів, необхідність автоматизації управління розгорнутими веб-додатками тільки посилюється. Тенденція на нові методи та алгоритми автоматизації управління буде лиши посилюватися.

Ми вважаємо, що узагальнення отриманого практичного досвіду буде корисно для більш глибокого розуміння суті автоматизації управління, її необхідності та можливостей, а також полегшить перехід до вирішення складніших завдань.

На наш погляд, проведене дослідження буде корисно для усіх розробників, які розробляють та створюють додатки, та для розробників, які управляють та адмініструють розгорнуті додатки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Rahul Sharma Traefik API Gateway for Microservices / With Java and Python Microservices Depolyed in Kubernetes, 2018 – 120 p.

2. Scott Surovich, Marc Boorshtein Kuberenetes and Docker / An Enterprise Guide: Effectively containerize applications, integrate enterprise systems, and scale applications in your enterprise, 2017. – P. 140-180.

3. Saibal Ghosh Docker Demystified: Learn How to Develop and Deploy Applications Using Docker (English Edition), 2016 – 15 p.

4. Richard Bullington-McGuire Docker for Developers: Develop and run your application with Docker containers using DevOps tools for continuous delivery, 2018 – 35 p.

5. Elton Stoneman Learn Docker in a Month of Lunches 1st Edition, 2019 – 50 p.

6. Andrew Block Learn Helm: Improve productivity, reduce complexity, and speed up cloud-native adoption with Helm for Kubernetes, 2017 – 100 p.

7. Geraldine A. Van der Auwera Genomics in the Cloud: Using Docker, GATK, and WDL in Terra 1st Edition, 2018 – 321 p.

8. Edwin M Sarmiento The SQL Server DBA’s Guide to Docker Containers: Agile Deployment without Infrastructure Lock-in 1st ed. Edition, 2016 – 222 p.

9. Nills Franssens Hands-On Kubernetes on Azure: Automate management, scaling, and deployment of containerized applications, 2nd Edition, 2018 – 123 p.

10. Dijkstra E. How do we tell truths that might hurt? / E. Dijkstra // Selected Writings on Computing: A Personal Perspective. – 1982. – № 1(23). – P. 89-131.

11. Барри Берд Программирование на Java для чайников, 3-е издание = Beginning Programming with Java For Dummies, 3rd Edition. — М.:«Диалектика», 2013. — 384 с. Encyclopedia of Artificial Intelligence. 2nd ed. / [S. C. Shapiro editor]. – New York: John Wiley & Sons, 1992. – Volume 1. – p. 434.

12. Fox G. C. Solving Problems on Concurrent Processors / G. C. Fox. – Prentice Hall, Englewood Cliffs, NJ, 1988. – 416 p.

13. Герберт Шилдт.Java. Полное руководство. Java SE 7 = Java 7: The Complete Reference. — 8-е изд. — М.: Вильямс, 2012. — 1104 с.Freeman B. NET 4.5 Parallel Extensions Cookbook / B. Freeman. – Published by Packt Publishing Ltd., UK. – 320 p.

14. Miles Price Docker The Comprehensive Beginners Guide to Take Control of Docker Programming 2018. – 365 p.

15. Geist G. A., Beguelin A., Dongarra J., Jiang W., Manchek B., Sunderam V. PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing. MIT Press, 1994. – 740 p.

16. Ghosh S. Distributed Systems. An Algorithmic Approach / S. Ghosh. – CRC, 2007. – 389 p.

17. Nills Franssens Hands-On Kubernetes on Azure: Automate management, scaling, and deployment of containerized applications, 2nd Edition, 2018 – 124 p.

18. Herlihy M., Kozlov D., Rajsbaum S. Distributed computing through combinatorial topology. – Elsevier Inc., Waltham, 2014. – 310 p.

19. Hillar G. C. Professional Parallel Programming with Java / G. C. Hillar. – Wiley Publishing, Inc., Indianapolis, Indiana, 2011. – 547 p.

20. Miles Price Docker The Comprehensive Beginners Guide to Take Control of Docker Programming 2018. – 365 p.

21. Johnson E. E. Completing an MIMD Multiprocessor Taxonomy // Computer Architecture News, 1988. – Vol. 16. – № 2. – P. 44-48.

22. Marshall D. Parallel Programming with Microsoft Visual Studio 2010 Step by Step / D. Marshall. – O’Reilly Media, Inc., 2011. – 249 p.

23. McCool M., Robison A.D., Reinders J. Structured Parallel Programming: Patterns for Efficient Computation. – Morgan Kaufmann, 2012. – 433 p.

24. Midkiff S. P. Automatic Parallelization. An Overview of Fundamental Compiler Techniques / S. P. Midkiff. – Morgan & Claypool, 2012. – 170 p.

25. Miller R., Boxer L. Algorithms Sequential & Parallel: A Unified Approach / R. Miller, L. Boxer. – Cengage Learning, 2012. – 448 p.

26. Murray, T. Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art / T. Murray. // International Journal of Artificial Intelligence in Education. – 1999.– № 10 – P. 98-129.

27. Padua D. Encyclopedia of Parallel Computing / D. Padua. – Springer, New York, Dordrecht, Heidelberg, London, 2011. – 2196 p.

28. Parhami B. Introduction to Parallel Processing. Algorithms and Architectures / B. Parhami. – Kluwer, 2002. – 557 p.

29. Pfister G. P. In Search of Clusters / G. P. Pfister. – Prentice Hall PTR, 1998. – 314 p.

30. Rahman M. C# Deconstructed / M. Rahman. – Apress, 2014. – 172 p.

31. Rauber T., Rünger G. Parallel Programming: for Multicore and Cluster Systems / T. Rauber, G. Rünger. – Springer, 2013. – 522 p.

32. Ringler R. C# Multithreaded and Parallel Programming / R. Ringler. – PACKT Published, 2015. – 344 p.

33. Stroustrup B. The C++ Programming Language 4th Edition / B. Stroustrup. Prentice Hall, 2013. – 1281 p.

34. Narendranath Reddy Thota Mastering Hyperledger Fabric: Master The Art of Hyperledger Fabric on docker, docker swarm and Kubernetes, 2018 – 25 p.

35. Stephen Fleming Continuous Delivery Handbook: Non Programmer's Guide to DevOps, Microservices and Kubernetes 2018. – 108 p.

36. Stephen Fleming DevOps And Microservices Handbook: Non-Programmer's Guide to DevOps and Microservices, 2018. – 503 p.

37. Brandon Shaw Docker Step-by-Step: The Ultimate Guide From Beginner to Expert. Learn & Master The Platform and Containerize, Create, Deploy and Run Your Application Like a Professional,2019 г. – 344 p.

38. Mr Daniel Jones the Ultimate Beginners Guide to Learn Docker Programming 2017, № 1. – 5 p.

39. Буч Г. Объектно–ориентированный анализ и проектирование с примерами приложений / Г. Буч. – 3–е издание, –М.: ООО "И. Д. Вильямс", 2008. – 720 с.

40. Шуп Р. Изучаем ActionScript 3.0. От простого к сложному / Р. Шуп, З. Россер. – СПб.: Символ-Плюс, 2009. – 495 с.

41. Nick Williamsin the Ultimative Beginners Guide to Starting with and Mastering Docker Fast!, 2016. – 184 p.

42. Narendranath Reddy Thota Mastering Hyperledger Fabric: Master The Art of Hyperledger Fabric on docker, docker swarm and Kubernetes, 2018 – 25 p.

43. Голдштейн С., Зурбалев Д., Флатов И. и др. Оптимизация приложений на платформе .NET. – М.: ДМК Пресс, 2014. – 522 с.

44. Deepu K Sasidharan Full Stack Development with JHipster: Build full stack applications and microservices with Spring Boot and modern JavaScript frameworks, 2nd Edition, 2015 – 80 p.

45. Mattew N., Stones R. Beginning Databases with PostgreSQL: From Novice to Professional. 2nd Edition. – Apress, 2005. – 665 p.

46. Эккель Б. Философия Java. Библиотека программиста. 4-е изд.; пер. с англ. – СПб.: Питер, 2009. – 640 с.

47. Шилдт Г. Полный справочник по Java. – М.: Вильямс, 2009. - 1040 с Cay S. Horstman Core Java Volume 1 - Fundamentals, 2018. — 824 c.

48. John Casey Create a Customized Build Process in Maven, 2009. – 1328 p.

49. Daryl Wood Gerber Sifting Through Clues. – 2019. – 451 p.

50. Michael Charge Docker Easy: The Complete Guide on Docker World for Beginners, 2015 – 45 p.

51. Jordan Lioy Kubernetes: Build and Deploy Modern Applications in a Scalable Infrastructure. The Complete Guide to the Most Modern Scalable Software Infrastructure. (Docker & Kubernetes), 2016 – 100 p.

52. Deepu K Sasidharan Full Stack Development with JHipster: Build full stack applications and microservices with Spring Boot and modern JavaScript frameworks, 2nd Edition, 2018 – 142 p.

53. Narendranath Reddy Thota Mastering Hyperledger Fabric: Master the Art of Hyperledger Fabric on docker, docker swarm and Kubernetes, 2017 – 14 p.

54. Arun Kumar Docker: A complete beginner's guide, 206 – 124 p.

55. Christian Leornardo Docker: Docker for the Absolute Beginner, 2018 – 41 p.

56. Marko Luksa Kubernetes in Action: 1st Edition, 2019 – 272 p.

57. Brayden Smith Kubernetes: A Step-by-Step Guide to Learn and Master Kubernetes, 2018 – 162 p.

58. Bilgin Ibryam Rolan Huss Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications, 2019 – 197 p.

59. Eva Tuczai Asena Hertz Managing Kubernetes Perfomance at Scale, 2019 – 87 p.

60. James Turnbull The Docker Book: Containerization is the new virtualization, 2018 – 502 p.

ДОДАТОК А

ЛІСТИНГ ПРОГРАМИ

Клас SchemaController

```
@Controller(crdName = "schemas.mysql.sample.javaoperatorsdk")
public class SchemaController implements ResourceController<Schema> {
  static final String USERNAME_FORMAT = "%s-user";
  static final String SECRET_FORMAT = "%s-secret";

  private final Logger log = LoggerFactory.getLogger(getClass());

  private final KubernetesClient kubernetesClient;

  public SchemaController(KubernetesClient kubernetesClient) {
    this.kubernetesClient = kubernetesClient;
  }

  @Override
  public UpdateControl<Schema> createOrUpdateResource(Schema schema, Context<Schema> context) {
    try (Connection connection = getConnection()) {
      if (!schemaExists(connection, schema.getMetadata().getName())) {
        try (Statement statement = connection.createStatement()) {
          statement.execute(
              format(
                  "CREATE SCHEMA `%1$s` DEFAULT CHARACTER SET %2$s",
                  schema.getMetadata().getName(), schema.getSpec().getEncoding()));
        }

        String password = RandomStringUtils.randomAlphanumeric(16);
        String userName = String.format(USERNAME_FORMAT, schema.getMetadata().getName());
        String secretName = String.format(SECRET_FORMAT, schema.getMetadata().getName());
        try (Statement statement = connection.createStatement()) {
          statement.execute(format("CREATE USER '%1$s' IDENTIFIED BY '%2$s'", userName, password));
        }
        try (Statement statement = connection.createStatement()) {
          statement.execute(
              format("GRANT ALL ON `%1$s`.* TO '%2$s'", schema.getMetadata().getName(), userName));
        }
        Secret credentialsSecret =
            new SecretBuilder()
                .withNewMetadata()
                .withName(secretName)
                .endMetadata()
                .addToData(
                    "MYSQL_USERNAME", Base64.getEncoder().encodeToString(userName.getBytes()))
                .addToData(
                    "MYSQL_PASSWORD", Base64.getEncoder().encodeToString(password.getBytes()))
                .build();
        this.kubernetesClient
            .secrets()
```

```
            .inNamespace(schema.getMetadata().getNamespace())
            .create(credentialsSecret);

        SchemaStatus status = new SchemaStatus();
        status.setUrl(
            format(
                "jdbc:mysql://%1$s/%2$s",
                System.getenv("MYSQL_HOST"), schema.getMetadata().getName()));
        status.setUserName(userName);
        status.setSecretName(secretName);
        status.setStatus("CREATED");
        schema.setStatus(status);
        log.info("Schema {} created - updating CR status", schema.getMetadata().getName());

        return UpdateControl.updateStatusSubResource(schema);
      }
      return UpdateControl.noUpdate();
    } catch (SQLException e) {
      log.error("Error while creating Schema", e);

      SchemaStatus status = new SchemaStatus();
      status.setUrl(null);
      status.setUserName(null);
      status.setSecretName(null);
      status.setStatus("ERROR");
      schema.setStatus(status);

      return UpdateControl.updateCustomResource(schema);
    }
  }

  @Override
  public DeleteControl deleteResource(Schema schema, Context<Schema> context) {
    log.info("Execution deleteResource for: {}", schema.getMetadata().getName());

    try (Connection connection = getConnection()) {
      if (schemaExists(connection, schema.getMetadata().getName())) {
        try (Statement statement = connection.createStatement()) {
          statement.execute(format("DROP DATABASE `%1$s`", schema.getMetadata().getName()));
        }
        log.info("Deleted Schema '{}'", schema.getMetadata().getName());

        if (userExists(connection, schema.getStatus().getUserName())) {
          try (Statement statement = connection.createStatement()) {
            statement.execute(format("DROP USER '%1$s'", schema.getStatus().getUserName()));
          }
          log.info("Deleted User '{}'", schema.getStatus().getUserName());
        }

        this.kubernetesClient
            .secrets()
            .inNamespace(schema.getMetadata().getNamespace())
            .withName(schema.getStatus().getSecretName())
```

```
            .delete();
      } else {
        log.info(
            "Delete event ignored for schema '{}', real schema doesn't exist",
            schema.getMetadata().getName());
      }
      return DeleteControl.DEFAULT_DELETE;
    } catch (SQLException e) {
      log.error("Error while trying to delete Schema", e);
      return DeleteControl.NO_FINALIZER_REMOVAL;
    }
  }

  private Connection getConnection() throws SQLException {
    return DriverManager.getConnection(
        format(
            "jdbc:mysql://%1$s:%2$s?user=%3$s&password=%4$s",
            System.getenv("MYSQL_HOST"),
            System.getenv("MYSQL_PORT") != null ? System.getenv("MYSQL_PORT") : "3306",
            System.getenv("MYSQL_USER"),
            System.getenv("MYSQL_PASSWORD")));
  }

  private boolean schemaExists(Connection connection, String schemaName) throws SQLException {
    try (PreparedStatement ps =
        connection.prepareStatement(
            "SELECT schema_name FROM information_schema.schemata WHERE schema_name = ?")) {
      ps.setString(1, schemaName);
      try (ResultSet resultSet = ps.executeQuery()) {
        return resultSet.first();
      }
    }
  }

  private boolean userExists(Connection connection, String userName) throws SQLException {
    try (PreparedStatement ps =
        connection.prepareStatement("SELECT User FROM mysql.user WHERE User = ?")) {
      ps.setString(1, userName);
      try (ResultSet resultSet = ps.executeQuery()) {
        return resultSet.first();
      }
    }
  }
}
```

Yaml-файл deployment.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: mysql-schema-operator
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: mysql-schema-operator
  namespace: mysql-schema-operator
spec:
  selector:
    matchLabels:
      app: mysql-schema-operator
  replicas: 1
  strategy:
    type: Recreate на протяжении обновления оператор будет уничтожен до того, как начнет работать новая версия, чтобы избежать работающих одновременно 2 операторов разных верисй
  template:
    metadata:
      labels:
        app: mysql-schema-operator
    spec:
      serviceAccount: mysql-schema-operator # уточняется ServiceAccount под которым RBAC разрешения оператора будут выполняться
      containers:
      - name: operator
        image: ${DOCKER_REGISTRY}/mysql-schema-operator:${OPERATOR_VERSION}
        imagePullPolicy: Always
        ports:
        - containerPort: 80
        env:
        - name: MYSQL_HOST
          value: mysql.mysql # предполагается, что MySQL запущен в namespace "mysql" в Kubernetes
        - name: MYSQL_USER
          value: root
        - name: MYSQL_PASSWORD
          value: password
        readinessProbe:
          httpGet:
            path: /health # когда возвращает статус 200, оператор работает в нормальном  режиме
            port: 8080
          initialDelaySeconds: 1
          timeoutSeconds: 1
        livenessProbe:
          httpGet:
            path: /health # когда этот эндпоинт не возвращает статус 200, это означает, что оператор не работает и он автоматически перезагружается
            port: 8080
          initialDelaySeconds: 30
          timeoutSeconds: 1
```

Клас MySQLSchemaOperator

```
public class MySQLSchemaOperator {

  private static final Logger log = LoggerFactory.getLogger(MySQLSchemaOperator.class);

  public static void main(String[] args) throws IOException {
```

```
    log.info("MySQL Schema Operator starting");

    Config config = new ConfigBuilder().withNamespace(null).build();
    KubernetesClient client = new DefaultKubernetesClient(config);
    Operator operator = new Operator(client);
    operator.registerControllerForAllNamespaces(new SchemaController(client));

    new FtBasic(new TkFork(new FkRegex("/health", "ALL GOOD!")), 8080).start(Exit.NEVER);
  }
}
```

ДОДАТОК Б

ВІДГУК
керівника економічного розділу
на кваліфікаційну роботу магістра
на тему: «Розробка та оптимізація методів та алгоритмів автоматичного управління розгорнутими контейнерізованими веб-додатками та сервісами»
студента групи 122м-19-1 Трофименко Дениса Олександровича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім'я файла	**Опис**
Пояснювальні документи	
Диплом_Трофименко.doc	Пояснювальна записка до магістерської роботи. Документ Word.
Диплом_Трофименко.pdf	Пояснювальна записка до до магістерської роботи в форматі PDF
Програма	
Program.rar	Архів. Містить коди програми і откомпільовану програму
Презентація	
Презентація_Трофименко.ppt	Презентація до магістерської роботи