ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра комп‘ютерних наук

Пояснювальна записка

до бакалаврської роботи
на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА KUBERNETES ОПЕРАТОРУ ДЛЯ АВТОМАТИЧНОЇ РОЗГОРТКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»**

Виконав: студент 4 курсу, групи КНД-42
спеціальності
122 Комп'ютерні науки
(шифр і назва спеціальності)
Лещинський А. Г
(прізвище та ініціали)
Керівник Вишнівський В.В.
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

Навчально-науковий інститут Телекомунікацій

Кафедра Комп'ютерних наук і технологій
Ступінь вищої освіти - «Бакалавр»
Напрям підготовки - 122 - "Комп'ютерні науки"

ЗАТВЕРДЖУЮ
Завідувач кафедри
Телекомунікаційних технологій
_____А.В. Бондаренко
“____” _____2021 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТУ

Лещинський Антон Геннадійович
(прізвище, ім‘я, по батькові)

1.Тема роботи: «Розробка Kubernetes оператору для автоматичної розгортки програмного забезпчення»_____
_____

Керівник роботи_Вишнівський Віктор Вікторович, завідувач кафедри__,
(прізвище, ім‘я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від “ ”_____2021 року №__.

2. Строк подання студентом роботи_____

3. Вхідні дані до роботи:
Віртуалізація, Контейнеризація, Kubernetes, DNS, Мікросервісна архітектура, Load Balancing, Service Discovery

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
4.1 Аналіз контейнеризації.
4.2 Аналіз Kubernetes.
4.3 Розробка Kubernetes оператору.
4.4 Використання Kubernetes оператору.

5. Перелік графічного матеріалу

1.

2.

3.

4.

24

6. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз віртуалізації		
2	Аналіз контейнеризації		
3	Аналіз Kubernetes	……..	
4	Розробка Kubernetes оператору	……..	
5	Використання Kubernetes оператору	……..	
6	Оформлення роботи	……..	

Студент _____ Лещинський А.Г.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Вишнівський В.В.
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 55 с., 1 дод., 6 джерел.

KUBERNETES, КОНТЕЙНЕРИ, ВІРТУАЛІЗАЦІЯ, LOAD BALANCING, SERVICE DISCOVERY, MYSQL.

Об`єкт дослідження – розробка Kubernetes оператору для автоматичної розгортки програмного забезпечення

Предмет дослідження – Kubernetes

Мета роботи – дослідження Kubernetes для подальшого використання та розробки операторів

Методи дослідження – технології віртуалізації, контейнеризації та Kubernetes.

Проведений аналіз сучасних технологій контейнеризації, детальний аналіз структури та компонентів Kubernetes, а також аналіз існуючих операторів MySQL.

На основі результатів виконаних досліджень розроблено Kubernetes оператор за допомогою Operator SDK на Golang.

Упровадження розробленого ПЗ дозволяє швидко розгортати Mysql\MariaDB кластера на Kubernetes.

ЗМІСТ

Вступ

У наш час однією з найбільш актуальних галузей є хмарні обчислення. Як невеликі стартапи, так і величезні продуктові компанії використовують хмари замість купівлі або оренди датацентрів. Лідери цієї галузі намагаються створити найбільш відмовостійкий та зручний сервіс для своїх клієнтів. З кожним роком створюються нові рішення щодо забезпечення найкращого сервіса серед конкурентів.

Окрім того, поширення знайшла модель надання хмарних обчислень PaaS (Platform as a Service – "Платформа як послуга"), при якій споживач отримує готовий комплект програмного забезпечення – операційні системи, бази даних, засоби розробки та тестування, а вся інфраструктура, мережі, сервера, системи збереження забезпечуються та контролюється провайдером.

Розглянемо найбільш важливі критерії для споживача хмарних сервісів:

- Uptime. Відмовостійкість та надійність сервісу.
- Multiregional Availability. Доступність серед регіонів світу.
- Security. Безпека, стійкість до взломів та різних атак, контролювання прав серед користувачів (юзерів) системи.
- Performance. Продуктивність, зокрема можливість розподілити навантаження на сервіс між декількома серверами для забезпечення найкращої продуктивності незалежно від кількості споживачів.
- Ease of Deployment and Scalability. Легкість деплойменту (розгортки) та масштабування продукту.

Для забезпечення цих критеріїв було створено немало технологій. Серед них важливими є *віртуалізація* та *контейнеризація.* Обидві використовуються для відокремлення та емуляції середовища виконання, а також для підвищення переносимості додатків. Віртуалізація – більш ретельний процес емуляції та звичайно використовується для цілих систем, а контейнеризація – більш легкий, і може використовуватися навіть для кожного додатку.

Відокремлений у контейнері додаток називають контейнеризованим. Зі створенням цієї технології з’явились інструменти для швидкого деплойменту таких додатків. Одним з таких є Docker Compose, що дозволяє за допомогою декількох конфігураційних файлів розгорнути повністю робочу сервісну сітку. Наприклад, запустити бекенд, фронтенд та базу даних лише однією командою менш чим за хвилину.

Для величезних сервісних сіток, а також для задовільнення раніше згаданих критеріїв були створені системи оркестрації. Найбільш відомою у наш час системою оркестрації є Kubernetes, що і є об'єктом дослідження цієї роботи.

К кінцю 2010 років Kubernetes став неофіційним стандартом для хмарних систем базованих на контейнеризованих додатках, зокрема мікросервісів та набув підтримки серед найбільших поставників хмарних послуг, таких як Amazon (AWS), Microsoft (Azure), Google (GCP), HPE (HPE Container Platform).

Альтернативою цього продукту є Docker Swarm. Він легший для розуміння, проте є пропрієтарним, а також менш використовуваним у величезних проектах.

1 Аналіз контейнеризації

1.1 Віртуалізація

Незважаючи на то, що віртуалізація не обов'язковою частиною контейнеризації, а також Kubernetes, я вважаю що для розуміння цих технологій, треба мати базове розуміння саме віртуалізації.

Віртуалізація надає можливість запуску віртуальних машин. З точки зору користувача, віртуальна машина не відрізняється від окремого серверу або комп'ютеру. Це задовольняється за допомогою емуляції або мостування (bridging – делегування функцій до фізичного пристрою) віртуальних пристроїв, як процесор, пам'ять, сховища даних, пристроїв вводу тощо.

Для забезпечення продуктивності роботи використовується гіпервізор. За допомогою режиму гіпервізору, процесор підтримує одночасну роботу декількох операційних систем тобто однієї хостову та декількох гостевих (що працюють на віртуальних машинах) на низькому уровні. Без гіпервізору ці операції виконувались програмно, що збільшувало час виконання кожної з них у декілька разів. На щастя, підтримка гіпервізору є частиною більшості сучасних x86 процесорів.

Використання віртуальних машин досить поширене. Провайдери хостингів використовують їх для поділення одного виділеного серверу на декілька віртуальних. З точки зору безпеки, це виключає можливість отримати доступ до хостової машини, а також у випадку перенавантаження віртуальної машини зменшується шанс перенавантаження самого серверу, так як ресурси резервуються. Окрім того, з'являється можливість виділити точну кількість ресурсів за запитом споживача.

Інший приклад – це використання віртуальних машин для розробки та тестування. Можливість запустити одну або декілька віртуальних машин на одному комп'ютері з середою розробки надає перевагу за використання окремих пристроїв,

при цьому можливо емулювати майже будь-яку архітектуру процесорів та запустити будь-яку операційну систему.

З мого досвіду, ці два приклади широко використовуються кожен день у великих та малих компаніях.

1.2 Контейнеризація

Контейнеризація - це більш сучасна технологія, оскільки завдяки підтримці зі сторони хостової операційної системи, вона лише відокремлює середовище, застосовуючи теж саме ядро операційної системи. Завдяки цьому, створення контейнерів набагато ефективніше за створення віртуальних машин, майже без втрат у продуктивності в порівнянні зі звичайною системою.

Варто помітити, що контейнери на відміну від віртуальних машин, не емулюють жодний пристрій. Замість цього, на уровні ядра операційної системи здійснюється контроль доступу до них.

Саме тому, контейнер однієї операційної системи може лише працювати на такий же системі. Проте деякі реалізації дозволяють використовувати віртуальне ядро операційної системи для підтримки контейнерів іншої ОС, наприклад Docker on Windows дозволяє запускати Linux контейнери.

Перевагою контейнерів є можливість досить швидко розгорнути контейнеризований додаток за допомогою образу. При цьому образ звичайно включає в себе файли гостевої операційної системи та файли додатку, іноді також файли необхідних бібліотек. Деякі образи можуть бути досить легкими. Наприклад, образ MySQL серверу важить лише ~ 150 MB.

Найбільш відомою на цей час реалізацією контейнерів є containerd (також відома як частина Docker Engine до відокремлення). Окрім того є інші реалізації, такі як LXC (Linux Containers), CRI-O. Варто згадати, що деякі з них можуть бути взаємозамінними завдяки підтримуванню загальних форматів (наприклад, OCI - Open Container Initiative).

Для більшого розуміння контейнерів можна розібрати завдяки чому вони працюють на тій або іншій операційній системі. Прикладом буде Linux Containers, проте Windows контейнери працюють досить схоже.

Більшість функціоналу контейнеризації забезпечено завдяки неймспейсам (namespace/простір імен). Вони дозволяють відокремити певні функції для групи процесів.

Найбільшу роль виконують контрольні групи (CGroups, Control Groups), Деякі додаткі мають контролюватися та обмежуватися заради стабільності, і в деякої степені, безпеки. Дуже часто баги або просто поганий код можуть нашкодити, а іноді навіть зруйнувати цілу екосистему. На щастя, є засіб тримати ці додатки під контролем. Контрольні групи – це функціонал, що обмежує, веде облік та ізолює CPU, пам’ять, I/O дисків та використання мережі для одного або деяких процесів. Уперше розроблена Google, технологія контрольних груп була пізніше додана до ядра Linux.

Основна мета контрольних груп – забезпечити єдиний інтерфейс контролю процесів.

Контрольні групи забезпечують наступне:

- Обмежування ресурсів – група може бути законфігурована не перевищувати вказану кількість пам’яті, ядр процесору або обмежена до деяких периферійних девайсів.
- Пріоритизація – одна або декілька груп можуть бути законфігуровані зменшувати використання ядр процесору, або швидкості I\O в користь більш важливим процесам.
- Моніторинг – використання ресурсів може бути виміряний та відстежений.
- Контроль – група процесів може бути заморожена, призупинена або перезапущена.

Програмне забезпечення для використання контрольних групп може бути використано окремо від засобів контейнеризації.

Окрім контрольних групп, є інші неймспейси:

- pid неймспейс відповідає за нумерацію та ієрархію процесів. При створенні нового pid неймспейсу, процес, що запускається першим, отримує PID 1, а усі наступні процесси стають дочірніми до нього. Усі інші процеси становляться невидимими для юзера контейнеру.
- Networking неймспейси дозволяють запускати програми на будь-якому порті без конфліктів с використаними у хостовій системі або інших неймспейсах.
- Mount неймспейс дозволяє монтувати та демонтувати файлові системи не зачіпаючи хостову файлову систему. Тобто можливо замонтувати інші девайси, як більше так і менше. Наприклад, замонтувати USB флешку тільки для контейнеру або тільки для хосту.
- User неймспейс відокремлює усіх юзерів хостової системі від контейнеру. Таким чином, окрім root, контейнера система не містить жодного юзеру, окрім створених окремо для неї.

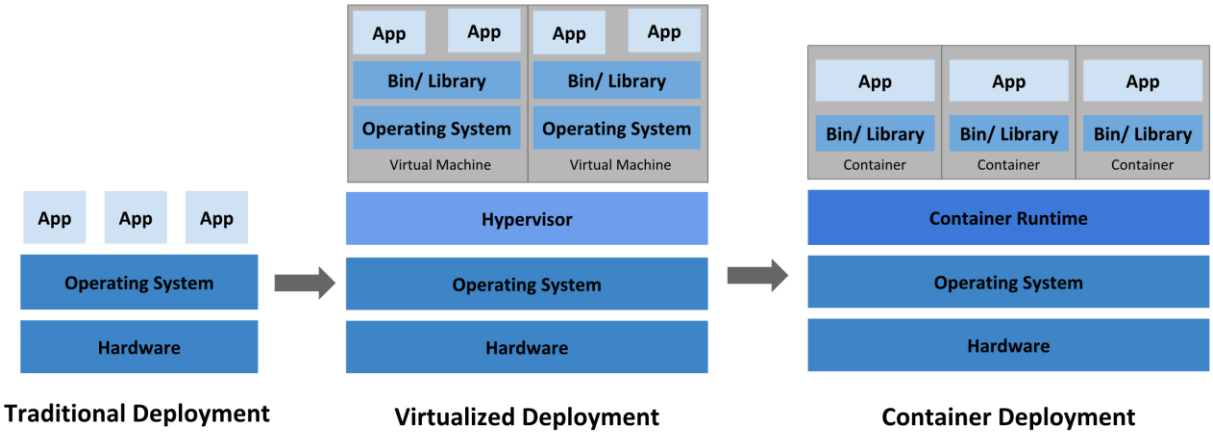
Наступним не менш важливим компонентом контейнеризації є Union FS – каскадно-об'єднані файлові системи, що дозволяють файлам та каталогам ізольованих файлових систем (що називають бранчами або гілками) бути об’єднаними у одну. Таким чином, для користувачів цієї файлової системи, будуть доступні файли з кожної з них. Також для таких бранчів вказується пріоритет. Це дозволяє з’ясувати який з файлів буде доступний якщо декілька систем містять файли з одним іменем.

Різноманітні бранчі можуть знаходитися у режимах “читання-запис” або “тільки читання”. Таким чином можна з'ясувати які файлові системи повинні отримати файл при запису. Окрім того, файлова система може виглядати змінною, але насправді не змінюватися. Це задовольняється за допомогою копіювання при запису, коли запис одночасно відбувається для усіх змінюваних файлових систем.

Використання Union файлових систем або їх альтернатив залежить від двигуну контейнеризації. Наприклад, Docker використовує Aufs для слоювання образів. Окрім того, існує overlayfs тощо.

Інша величезна перевага деяких двигунів контейнеризації – слоювання образів. Хоча це не обов'язковий елемент контейнеризації, але на наш час досить стандартизований. Прикладом цього є найбільш відомий двигун контейнеризації – Docker. Він поділяє процес створення імеджів на послідовні кроки. Першим кроком звичайно є базовий імедж (або scratch якщо треба використовувати порожній). Іншими можуть бути будь-які команди, такої як виконання скрипту, або додавання файлів з хостової машини, на якій створюється образ. Після виконання команди усі змінені файли відокремлюються від існуючих до цього кроку, стискаються у один файл, що називається layer (шар). Таким чином імедж містить декілька шарів файлів, що при використанні монтуються у одну файлову систему завдяки Aufs. Команди (кроки) містяться у файлі з іменем Dockerfile. Якщо не змінюючи Докерфайл (або додані файли), повторно запустити побудову імеджу, вона відбудеться майже миттєву отже ніяких змін не було. Якщо змінити лише один крок, усі попередні кроки будуть використовувати вже побудовані шари. В підсумку, ця технологія дозволяє не тільки зменшити час побудови, але й розмір кінцевого образу, оскільки зберігаються лише різні шари.

1.3 Порівняння



Для підсумування використання різних підходів деплойменту додатків, можна порівняти традиційний, віртуалізований та контейнеризований деплоймент.

При використанні традиційного підходу, на сервері встановлена операційна система та виконуються один або декілька додатків.

При використанні віртуалізації на хостовій операційній системі запускаються декілька віртуальних машин, на кожної з яких – операційні система та додатки.

При використанні контейнеризації на хостовій оперраційній системі запускається двигун контейнеризації, за допомогою якого виконуються контейнеризовані додатки.

2 Аналіз Kubernetes

2.1 Вступ до Kubernetes

Контейнери – це чудовий спосіб збірки та запуску ваших додатків. У продакшені важливим фактором є можливість програмних або апаратних помилок, що приводить до часу простою (downtime). Якщо контейнер крашнеться, інший контейнер має запуститись. Не простіше було би забезпечити це за допомогою готової системи?

Для цього був створений Kubernetes. Kubernetes надає фреймворк для виконання розподілених систем с забезпеченням стійкості, займається масштабуванням та відмовостійкістю додатків, надає методи деплойменту та величезну кількість інших переваг:

- Service discovery та Load Balancing – Kubernetes може надавати доступ до контейнеру за допомогою DNS імені або IP адреси. При звищенному трафіку на контейнер, Kubernetes може балансувати навантаження і розподіляти трафік для стабильного деплойменту.
- Оркестрація сховищ – Kubernetes дозволяє автоматично монтувати системи збереження даних на вибір, будь-то локальні сховища, або хмарні провайдери.
- Автоматичний розподіл ресурсів – Після додавання до Kubernetes кластеру з вузлів для запуску контейнеризованих задач ви можете вказати необхідну кількість ресурсів (CPU, пам’ять, сховища) для кожного контейнеру, після чого Kubernetes вмістить усі контейнери для найкращого використання ресурсів.
- Самолікування – Kubernetes перезапускає контейнери що провалюються, вбиває та заміняє контейнери що не відповідають на Health Check (скрипт, що перевіряє працездатність додатку), а також перекриває доступ доки контейнер не буде готов до роботи.

- Менеджмент секретів та конфігурації – Kubernetes дозволяє зберігати чутливу інформацію, як паролі, токени або ключі за допомогою секретів, а також конфігурації без перебудування образів контейнерів, а також без видавання секретів за межі.

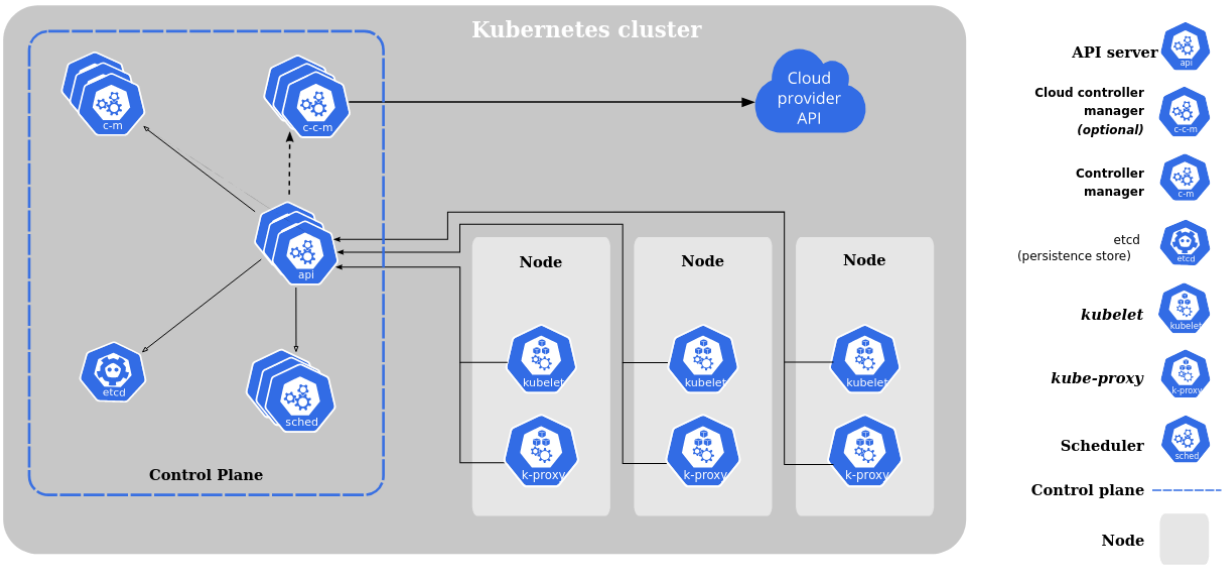
2.2 Компоненти Kubernetes

Після деплою Kubernetes, ви отримуєте *кластер.*

Kubernetes кластер містить машини, що називаються *вузлами* або *нодами.* Саме на них виконуються контейнеризовані додатки. Кожен кластер має принаймні одну воркер (worker) вузол.

Вузли хостять *Поди (Pods)* що є компонентом application workload (об'єму додатків), та містять один або декілька контейнерів. Контрольна площина (control plane) – шар оркестрації контейнерів, що контролює вузли, поди та життєвий цикл контейнерів у них. У продакшені звичайно контрольна площина виконується серед декількох комп'ютерів, а кластер має декілька вузлів для забезпечення помилкостійкості та високої доступності.

Наступна діаграмма відображає усі компоненти у св'язці.



2.3 Компоненти контрольної площини

Компоненти контрольної площини відповідають за базові операції кластеру, а також оброблює події кластеру. Наприклад, якщо кількість реплік не відповідає очікуваному, запускає додаткові поди.

kube-apiserver

Сервер API – компонент, що надає доступ до контрольної площини через програмний інтерфейс. Основною реалізацією цього серверу є kube-apiserver. Він спроектований для горизонтального масштабування, тобто деплоймент у декілька екземплярів.

etcd

Розподілене та високонадійне сховище даних у форматі key-value, що використовується як основне сховище даних кластеру в Kubernetes.

kube-scheduler

Компонент контрольної площини що відсліджує создані поди без прив’язаних вузлів та обирає вузол для їх роботи. При плануванні деплойменту под, враховуються багато чинників, у тому числі вимоги до ресурсів, обмеження пов’язані з різними апаратними та програмними політиками, належенню або неналеженню вузлів (affinity\anti-affinity) тощо.

kube-controller-manager

Менеджер процессів *контроллера* – контролюючого циклу що відстежує загальний стан через АПИ сервер та вносить зміни для задовільнення очікуємого стану.

З точки зору логіки кожен контроллер це окремий процесс, проте для зменшення ускладненності, вони звичайно скомпільовані в один бінарний файл та запускаються як один процесс.

Серед основних контроллерів:

- **Контроллер вузлів** — відповідальний за помічання та реагування коли вузли виходять зі строю.
- **Контроллер завдань** (jobs) – слідкує за Job об’єктами являються одноразовими задачами (task), та створюють поди для виконання цих задань.
- **Контроллер кінцевих точок** (endpoint) – заселяє об’єкти кінцевих точок (тобто об’єдиняє Сервіси та Поди).
- Та інші.

2.4 Компоненти вузлів

Компоненти вузлів запускаються на кожному із вузлів, підтримують запущені поди та надають Kubernetes середовище виконання.

kubelet

Агент що виконується на кожній ноді у кластері. Він впевнюється що контейнери виконуються всередині Под.

Kubelet бере множину PodSpecs (специфікація под) що надаються завдяки різним механізмам та впевнюється що контейнери описані у цих PodSpecs виконуються у робочому стані. Kubelet не контролює контейнери що не були створені Kubernetes.

kube-proxy

Kube-proxy - це мережевий проксі що виконується на кожній з нод у кластері, реалізуючи концепт сервісів (Service) у Kubernetes.

Kube-proxy підтримує мережеві правила на вузлах. Ці мережеві правила дозволяють мережеву коммунікацію до под з мережевих сесій зсередини або зовні кластеру.

Kube-proxy використовує шар фільтрування пакетів операційної системи, якщо він доступний. В іншому випадку він контролює та переводить трафік сам.

Container runtime

Контейнерне середовище — це програмне забезпечення відповідальне за виконання контейнерів.

Kubernetes підтримує декілька контейнерних середовищ: Docker, containerd, CRI-O. Перші два вже були описані в розділі контейнеризації. CRI-O – це легка реалізація створена під Kubernetes від авторів Kubernetes. Окрім того, підтримуються

будь-які реалізації створені за Kubernetes CRI (Container Runtime Interface) – інтерфейсом контейнерного середовища.

2.5 Addons

Addons (аддони\доповнення) використовують об’єкти Kubernetes для реалізації окремого функціоналу кластеру.

DNS

На відміну від інших аддонів, усі Kubernetes кластери повинні мати кластерний DNS, так як уся внутрикластерна коммуникація залежить від цього.

Кластерний DNS – це окремий DNS сервер, що надає DNS записи для Kubernetes сервісів.

Kubernetes контейнери автоматично включають DNS сервер в їх DNS пошуки.

Web UI (Dashboard)

Dashboard (приборна панель) надає веб інтерфейс до кластерів та дозволяє користувачам контролювати и виправляти додатки що виконуються у кластеры, а також сам кластер.

Cluster-level Logging

Логування на кластерному уровні — це механізм що відповідає за збереження логів контейнерів до центрального сховища логів з інтерфейсом пошуку та перегляду.

2.6 Kubernetes об’єкти

Kubernetes об’єкти (або ресурси) – це persistent entities (абстрактна модель що зберігається після перезавантаженні системи) в Kubernetes системі. Kubernetes використовує ці об’єкти щоб представляти стан кластеру. У тому числі:

- Які контейнеризовані додатки виконуються і на яких вузлах
- Які ресурси доступні для цих додатків
- Політики щодо того як ці додатки повинні поводитися при рестарті, апгрейдах та помилках
- Інші залежно від типу об'єкту

Kubernetes об’єкт це інформація щодо бажання – після створення об’єкту Kubernetes система буде постійно працювати та перевіряти що цей об’єкт існує. Створюючи об’єкт ви заявляєте системі як ви бажаєте бачити додатки на своєму кластері. Тобто це бажаний стан вашого кластеру.

Для роботи з Kubernetes об’єктами (створюванні, зміні або видаленні) треба використовувати Kubernetes API. Основний спосіб – використання kubectl – інтерфейсу командної строки, що робить Kubernetes API запроси за вас. Інакше можливо використовувати клієнтські бібліотеки (наприклад, client-go для Golang).

2.7 Специфікація та Статус об’єктів

Майже кожен Kubernetes об’єкт містить два поля-об’єкти (поля, значенням которих є об’єкт, що складається з інших іменованих полей): *spec* та *status*.

Створюючи об’єкт що містить *spec*, ви вказуєте характеристики, що очікуєте від ресурсів, тобто це – *бажаний стан* ресурсу. Він змінюється лише користувачем, або іноді при додаванні полей за замовчуванням.

Після створення та протягом усієї роботи, Kubernetes система створює та оновлює поле *status*, що відображає *теперішній стан.* Контрольна площина завжди активно намагається приблизити *теперішній стан* до *бажаного.*

Наприклад, в Kubernetes, Deployment (деплоймент) – це об'єкт, що представляє додаток що виконується на кластері. При створенні Деплойменту, можливо вказати поле *replicas* у об’єктному полі *spec*, зі значенням кількості реплік додатку. Система прочитає специфікацію та запустить три екземпляри бажаного додатку, після чого обновить статус. Якщо будь-який екземпляр відмовляє, система реагує на різницю між статусом та специфікацією та виправляє стан – наприклад, замінюючи екземпляр.

2.8 Опис Kubernetes об'єктів

Створюючи об'єкт в Kubernetes, треба вказати специфікацию об’єкту що описує бажаний стан, а також базову інформацію таку як ім’я об’єкту. При використанні Kubernetes API для створення об’єкту, API запит повинен містити цю інформацию у вигляді JSON як тіло запиту. Проте, частіше для полегшення читання цієї інформації, kubectl дозволяє використовувати .yaml файл, після чого він конвертує у JSON та робить запит.

Наприклад, наступний .yaml файл описує Kubernetes Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

У ньому вказано що бажано створити NGINX додаток, з двома репліками, з вказаним образом контейнеру, а також порт 80 (http), що відкриється для внутрішнього доступу.

2.9 Імена об’єктів та їх ID

Кожен об’єкт кластеру має ім’я унікальне для кожного типу ресурсів, а також UID унікальний серед об'єктів кластеру.

Наприклад, ви можете мати лише одну Поду з іменем *someapp* у рамках одного неймспейсу, проте ви можете також мати інші ресурси з цім іменем, наприклад Деплоймент *someapp*.

Окрім того, є не унікальні вказуємі користувачом атрибути – лейбли та аннотації (labels and annotations).

2.10 Namespaces (неймспейси)

Kubernetes підтримує декілька віртуальних кластерів розташовані на одному фізичному кластері – неймспейси, що дозволяють групувати інші об’єкти.

Вони призначені для роботи у середовищах з великою кількістю користувачів розподілених по командам або проектам. Для кластерів з лише кількома користувачами, необхідності користуватися неймспейсами звичайно нема.

Неймспейси надають рамки для імен об'єктів. Ресурси мають мати унікальне ім'я для кожного типу лише серед одного неймспейсу. Неймспейси не можуть знаходитися один в одному та кожний Kubernetes об’єкт може бути лише в одному неймспейсі.

Більшість Kubernetes об’єктів належать до неймспейсів. Але самі неймспейси, що є об’єктами, глобальні. Окрім цього, глобальними є ноди та деякі інші ресурси. Об’єкти що належать до неймспейсів називаються namespaced.

2.11 Контролери

В робототехніці та автоматизації є термін контрольного циклу – нескінченного циклу що регулює стан системи.

Прикладом цього циклу є термостат у комнаті.

Встановлюючи температуру задається бажаний стан. Актуальна температура кімнати – це теперішній стан. Термостат намагається змінити актуальну температуру у сторону бажаного стану виключаючи та включаючи пристрої.

Kubernetes використовує подібний паттерн контрольних циклів, що слідкують за станом кластеру та робить зміни якщо вони потрібні. Кожен контролер намагається змінити стан ближче до бажаного.

2.12 Workload ресурси

Workload – це додаток, що запускається на Kubernetes. Незалежно від того, один це компонент або декілька працюючих разом, на Kubernetes вони виконуються в подах. Под може містити як один так і декілька контейнерів.

Кожний под має свій життєвий цикл. Якщо на ноді критична помилка, усі поди провалюються. Для Kubernetes ця помилка значить що Под більше не придатний і треба замінити його новим.

Замість того, щоб робити це вручну, можно використовувати Workload ресурси. Ці ресурси конфігурють контроллери, що впевнюються, що певна кількість под виконуються у правильному стані.

Серед таких ресурсів є:

- Deployment та ReplicaSet – використовуються для Stateless додатків, де будь-який под може бути заміненим при необхідності. ReplicaSet створює поди, а Deployment створює ReplicaSet. Використовувати Деплоймент має більше сенсу, адже він автоматично відображує усі зміни в специфікації подів, перестворюючи ReplicaSet та підтримує декілька RollingStrategy – стратегії заміни реплік.
- StatefulSet – використовуються для Stateful додатків, де великий сенс має реплікація даних.
- DaemonSet – створює Поди для кожної ноди, що може бути фундаментальною функцією для створення якихось засобів зв'язку між кластерами.
- Job та CronJob – задачі, що мають виконатися та зупинитися. Job – одноразові задачі, що виконуються одразу після створення, а CronJob виконуються за розкладом.

2.13 Сервіси

Абстрактний спосіб відкрити додаток, що виконується на подах як мережевий сервіс.

З Kubernetes нема необхідності змінювати код додатку щоб підтримувати невідомий Service Discovery механізм. Kubernetes надає Подам окремий IP адрес, а також DNS ім'я для групи подів. Крім того він може автоматично розподіляти навантаження між ними.

Kubernetes поди створюються та знищуються щоб відповідати стану кластеру. На відміну від звичних серверів, поди - непостійні ресурси. При використанні Деплойменту, поди створюються за знищуються динамічно.

Хоча кожний под отримує свій IP адрес, у Деплойменті, група подів що виконується у цей час, може відрізнятися пізніше.

Це призводить до проблеми: якщо група подів використовує функціонал інших подів кластеру, як вони будуть слідкувати які IP адреса актуальні. Прикладом є бекенди та фронтенди звичайного веб додатку, де фронтенд повинен робити запити до бекенду.

Для цього можно використовувати *сервіси*.

В Kubernetes, сервіси – це абстракція що визначає колекцію подів та політики доступа до них. Колекція подів звичайно визначається законфігурованим селектором.

Окрім того, для відокремлення додатків один від одного, було введено принцип, при якому залежні сервіси не повинні мати ніякої інформації щодо стану інших сервісів. Тобто, для фронтенду немає ніякої різниці який саме екземпляр бекенду використовувати. Цей принцип надається сервісами.

2.14 Ingress

Окрім Сервісів, Kubernetes надає інший ресурс, що займається перенаправленням трафіку – Ingress. Він дозволяє відкривати додатки до зовнішньої мережі. У рамках цієї роботи я не буду використовувати Ingress, але для просунутого рівня конфігурації доступу це важливий компонент.

2.15 Custom Resources, controllers, operators

Ресурси – це кінцеві точки Kubernetes API, що зберігають та надають доступ до об'єктів певного типу, наприклад вбудований ресурс *pods* містить коллекцію подів.

Кастом ресурси – розширення до Kubernetes API для задовільнення додаткових функцій. Багато основних Kubernetes функцій є кастом ресурсами, що робить Kubernetes модульним. Кастом ресурс становиться ресурсом після додавання його формату, що зберігається як вбудований ресурс CustomResourceDefinition (CRD) і містить опис об'єктів за допомогою OpenAPI. Після цього вони стають доступними так само як вбудовані ресурси, наприклад через kubectl.

Кастом ресурси надають можливість зберігати та отримувати структуровану та стандартизовану інформацію. Як і будь-який ресурс, його об’єкт лише відображає бажаний стан кластеру. Для того, щоб його забезпечити, треба створити *Custom Controller* – контроллер, що міститься на кластері як окремий додаток та як і звичайний контролер, слідкує та виправляє стан відповідної частини кластеру. Вони можуть працювати без кастом ресурсів, проте разом вони надають величезний потенціал. Таке поєднання називається *оператором*.

2.16 Operator SDK

Operator SDK – це засіб для створення операторів. Хоча він не обов’язковий для цієї мети, він є досить швидким способом розробки порівняно зі звичайним Golang проектом з Kubernetes Golang API.

Основною функцією, що я використовував, була генерація boilerplate кода, наприклад функції копіювання об'єктів, або регістрацію контролерів. Крім того, він генерує CustomResourceDefinition на основі Golang структур. Саме тому, замість вказування OpenAPI .yaml файлів, я буду вказувати Golang структури, а також .yaml

приклади. Окрім того, я не буду вказувати сгенеровані частини коду (окрім випадків, де вони грають велику роль).

2.17 MySQL\MariaDB оператор

Однією із найбільш активно використовуваних баз даних у наш час є MySQL, а також її форки (відгалуження) MariaDB та Percona Server.

Kubernetes оператори можливо розробляти для будь-яких додатків. Я обрав MySQL\MariaDB оператор, оскільки протягом півтора років своєї праці з Kubernetes, неодночасно помічав, що в відкритому доступі немає функціонального оператору з підтримкою MariaDB. Так в мене з’явилася ідея розробити оператор, що може підтримувати не тільки MySQL, а й MariaDB та інші форки, як Percona.

Окремо треба помітити, що у наявності є оператори для MySQL від Oracle, та Percona Server від PressLabs з більш багатим функціоналом. Проте, у випадку з MariaDB я знайшов лише малофункціональні оператори.

При виборі теми для цієї дипломної роботи, я намагався не вказувати який саме додаток буде розроблений, адже основна мета цієї роботи – навчитися розробці будь-якого оператору, незалежно від додатку.

Проте, я маю бажання розширити функціонал цього оператору та при написанні магістерської роботи зробити акцент вже саме на MySQL операторі, який при наявності стабільної версії можна зробити Open Source.

3 Розробка Kubernetes оператору

3.1 Структура компонентів Kube-MySQL оператору

На теперішний час використовуються наступні компоненти:

1. Конфіг додатку – конфігурація версія або реалізації MySQL.
2. Кластер – контейнери MySQL серверу.
3. Термінал – контейнер, що містить MySQL клієнт для зв'язку з сервером. Він автоматично законфігурован на підключення до створеного кластеру.

3.1.1 MysqlConfig

Ресурс, що містить інформацію щодо додатку MySQL. Він створюється окремо під різні версії MySQL, а також різні реалізації (Oracle MySQL, MariaDB).

```
41  // MysqlConfig is the Schema for the mysqlconfigs API
42  type MysqlConfig struct {
43          metav1.TypeMeta   `json:",inline"`
44          metav1.ObjectMeta `json:"metadata,omitempty"`
45
46          Spec   MysqlConfigSpec   `json:"spec,omitempty"`
47          Status MysqlConfigStatus `json:"status,omitempty"`
48  }
```

Заданий ресурс містить об’єкти специфікації та статусу MysqlConfig, а також метаінформацію.

```
26  // MysqlConfigSpec defines the desired state of MysqlConfig
27  type MysqlConfigSpec struct {
28          Images map[string]string `json:"images,omitempty"`
29  }
30
```

У цієї версії є лише поле Images з образами, що повинні використовуватися для певних подів. Після додавання специфічних для кожної версії або реалізації функцій, він також буде включати інші поля.

Приклади:

```yaml
apiVersion: kubesql.vellanci.gh/v1alpha1
kind: MysqlConfig
metadata:
  name: mysqlconfig-sample
spec:
  images:
    mysql: "mysql:5.6"
```

Цей приклад вказує версію Oracle MySQL образу – 5.6

```yaml
apiVersion: kubesql.vellanci.gh/v1alpha1
kind: MysqlConfig
metadata:
  name: mysqlconfig-sample
spec:
  images:
    mysql: "mariadb:10.6"
```

Цей приклад вже використовує версію MariaDB 10.6

MysqlConfig є шаблоном для використання у MysqlCluster, тому не потребує окремого контролеру.

3.1.2 MysqlCluster

Ресурс, що містить бажаний стан MySQL кластеру. За його специфікацією буде створені об’єкти, необхідні для роботи з MySQL: StatefulSet, Service та PersistentVolumeClaim для зберігання інформації.

```
62  // MysqlCluster is the Schema for the mysqlclusters API
63  type MysqlCluster struct {
64          metav1.TypeMeta   `json:",inline"`
65          metav1.ObjectMeta `json:"metadata,omitempty"`
66
67          Spec   MysqlClusterSpec   `json:"spec,omitempty"`
68          Status MysqlClusterStatus `json:"status,omitempty"`
69  }
```

Заданий ресурс містить об’єкти специфікації та статусу MysqlCluster, а також метаінформацію.

```
41  type MysqlClusterSpec struct {
42          Config   MysqlClusterSpecConfig `json:"config,omitempty"`
43          Replicas *int32                 `json:"replicas,omitempty"`
44          Storage  MysqlClusterStorage    `json:"storage,omitempty"`
45          Terminal *MysqlClusterTerminal  `json:"terminal,omitempty"`
46  }
```

Специфікація MysqlCluster містить декілька полей:

```
27  type MysqlClusterSpecConfig struct {
28          Name string           `json:"name,omitempty"`
29          Spec *MysqlConfigSpec `json:"spec,omitempty"`
30  }
```

Config – об’єкт, що може містити ім'я MysqlConfig та\або специфікацію MysqlConfig. Якщо вказане лише ім’я, буде використаний створений заздалегідь

об’єкт MysqlConfig. Якщо вказана лише специфікація, буде використана вона. Якщо обидва вказані, то вони будуть об'єднані, проте специфікація має вищий пріоритет.

Replicas – кількість реплік для створення под.

```
32  type MysqlClusterStorage struct {
33          VolumeSource *v1.VolumeSource         `json:"volumeSource,omitempty"`
34          Resources    *v1.ResourceRequirements `json:"resources,omitempty"`
35  }
```

Storage – об’єкт, що може містити або VolumeSource або Resources. У першому випадку, користувач сам помічає яке сховище повинно використовуватися. У другому, треба лише вказати кількість storage ресурсів, а контроллер забезпечує сховище.

Terminal - об'єкт, при якого наявності створюється MysqlTerminal об’єкт, що автоматично прив'язується до цього кластеру. Таким чином можна без додаткових зусиль створити термінал MySQL.

Статус MysqlCluster містить декілька полей:

```
48  // MysqlClusterStatus defines the observed state of MysqlCluster
49  type MysqlClusterStatus struct {
50          ConfigSpec          MysqlConfigSpec `json:"configSpec,omitempty"`
51          StatefulSet         string          `json:"statefulSet,omitempty"`
52          Service             string          `json:"service,omitempty"`
53          StorageVolumeSource v1.VolumeSource `json:"StorageVolumeSource,omitempty"`
54
55          // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
56          // Important: Run "make" to regenerate code after modifying this file
57  }
```

ConfigSpec містить використану специфікацію MysqlConfig після об'єднання по необхідності.

StatefulSet містить ім'я створеного StatefulSet, що містить MySQL контейнери.

Service містить ім'я сервісу, що відкриває доступ до MySQL через порт за замовчуванням (3306).

StorageVolumeSource містить інформацію щодо використованого сховища.

Приклад:

```
apiVersion: kubesql.vellanci.gh/v1alpha1
kind: MysqlCluster
metadata:
  name: mysqlcluster-sample
spec:
  config:
    name: "mysqlconfig-sample"
  terminal: {}
```

Цей приклад створює Mysql кластер за заданим конфігом, а також автоматично створеним терміналом.

3.1.2 MysqlTerminal

Ресурс, що містить інформацію щодо бажаного стану терміналу, завдяки якому можно використовувати створений кластер. Перевагою цього об'єкту у порівнянні зі звичайним подом є автоматичне використання прив'язаного кластеру, а також правильних образів контейнеру.

```
45  // MysqlTerminal is the Schema for the mysqlterminals API
46  type MysqlTerminal struct {
47          metav1.TypeMeta   `json:",inline"`
48          metav1.ObjectMeta `json:"metadata,omitempty"`
49
50          Spec   MysqlTerminalSpec   `json:"spec,omitempty"`
51          Status MysqlTerminalStatus `json:"status,omitempty"`
52  }
```

Заданий ресурс містить об’єкти специфікації та статусу MysqlTerminal, а також метаінформацію.

```
26  type MysqlTerminalCluster struct {
27          Name      string `json:"name"`
28          Namespace string `json:"namespace,omitempty"`
29  }
30
31  // MysqlTerminalSpec defines the desired state of MysqlTerminal
32  type MysqlTerminalSpec struct {
33          Cluster MysqlTerminalCluster `json:"cluster"`
34  }
```

Специфікація терміналу містить ім’я та опціонально неймспейс кластеру. Завдяки цьому можливо підключатися до кластеру іншого неймспейсу. Якщо неймспейс не вказаний, термінал локальний.

При використанні поля Terminal ресурсу MysqlCluster, створюється локальний MysqlTerminal з ім’ям MysqlCluster.

Приклади:

```yaml
apiVersion: kubesql.vellanci.gh/v1alpha1
kind: MysqlTerminal
metadata:
  name: mysqlterminal-sample
spec:
  cluster:
    name: mysqlcluster-sample
```

Цей приклад створює локальний термінал до кластера mysqlcluster-sample.

```yaml
apiVersion: kubesql.vellanci.gh/v1alpha1
kind: MysqlTerminal
metadata:
  name: mysqlterminal-sample-diff-ns
spec:
  cluster:
    name: mysqlcluster-sample
    namespace: kubesql-sample
```

Цей приклад створює термінал до кластера mysqlcluster-sample, що розташований у неймспейсі kubesql-sample, при цьому цей термінал може бути розташований у будь-якому неймспейсі.

3.2 Пояснення коду Kubernetes оператору

3.2.1 Файли контролеру

Протягом написання цього кода, я намагався використовувати схоже найменування для функцій\структур, що схожі за метою:

build<Object> - функція, що створює певний об'єкт.

update<Object> - функція, що використовується ControllerUtil допоміжним файлом бібліотеки для вказання змін, які залежать від специфікації кастом ресурсу. Вона викликається під час створення або зміни об'єкту, на відміну від build<Object>, що лише створює об’єкт.

CreateOrUpdate<Object> - функція, що створює або оновлює об'єкт, використовуючи наведені функції. Вона звичайно використовує SetControllerReference функцію – яка зв’язує об’єкт з кастом ресурсом, щоб після видалення кастом ресурсу, видалявся цей залежний об’єкт.

deploy<Object> - функція, що об'єднує build<Object>, update<Object> та CreateOrUpdate<Object> за необхідністю відокремити певну частину

undeploy<Object> - функція, що видаляє зайві об'єкти у випадку де Garbage Collection не відпрацює (наприклад, якщо кастом ресурс не видалений).

reconcileWithoutResult – функція, що є альтернативою Reconcile, тобто це ітерація циклу, що викликується кожен раз, як кастом ресурс або залежні ресурси змінюються. Вона отримує запит з іменем та неймспейсом кастом ресурсу.

controller_tools.go

loggerForObject – створює логер, вказуючи ім’я та тип об’єкту в контексті.

CreateOrUpdate – створює або оновлює об’єкт

SetControllerReference – модіфікує об’єкт, для додавання controller reference, за допомогою якого здійснюється Kubernetes Garbage Collection, що видаляє залежні об’єкти від кастом ресурсу.

config_tools.go

GetMysqlConfig – отримує MysqlConfig об'єкт з ресурсу.

GetClusterConfig – отримує кінцевий конфіг.

MergeConfigs – об’єднує конфіги.

pvc_builder.go

buildPVC – створює об'єкт PersistentVolumeClaim з модом доступу ReadWriteOnce, та дефолтним розміром сховища 1 ГБ.

updatePVC – оновлює PVC у разі зміни ресурсів.

service_builder.go

buildService – створює об'єкт Service з відкритим портом 3306, а також типом ClusterIP, що відкриває доступ до подів з терміналів або інших подів.

set_builder.go

*build*StatefulSet та *buildPodSpec* – створюють об'єкт StatefulSet з єдиним контейнером, а також монтують сховище, що дозволяє MySQL кластеру зберігатися при перестворенні подів.

*update*StatefulSet – оновлює об’єкт у разі зміни специфікації подів.

cluster_terminal_builder.go

buildTerminal– створює локальний об'єкт MysqlTerminal з іменем кластеру для зв’язування.

undeployTerminal – видаляє термінал, якщо поле Terminal було видалено.

terminal_builder.go

buildPod – створює под з конфігурацією клієнту.

buildConfSecret – створює конфігурацію клієнту.

mysqlcluster_controller.go

reconcileWithoutResult – розгортає компоненти кластеру, а також локальний термінал при необхідності.

deployMysqlCluster – створює PersistentVolumeClaim, Service та StatefulSet.

SetupWithManager – зв'язує менеджер подій контроллера з кастом ресурсом, а також залежними ресурсами.

mysqlterminal_controller.go

reconcileWithoutResult – розгортає компоненти терміналу.

SetupWithManager – зв'язує менеджер подій контроллера з кастом ресурсом, а також залежними ресурсами.

4 Використання оператору

Для демонстрації результатів, треба встановити оператор:

git clone git@github.com:aleshchynskyi/kube-mysql.git

make deploy

Після чого, впевнитися, що оператор у здоровому стані:

```
NAME                                        READY   STATUS    RESTARTS   AGE
kube-mysql-controller-manager-5d77c8c7db-qpp6r   2/2     Running   0          5m29s
```

Після чого, можна створювати ресурси MysqlConfig, MysqlCluster, MysqlTerminal:

kubectl apply -f config/samples/kubesql_v1alpha1_mysqlconfig.yaml

kubectl apply -f config/samples/kubesql_v1alpha1_mysqlcluster.yaml

kubectl apply -f config/samples/kubesql_v1alpha1_mysqlterminal.yaml

Після перевірки, можна побачити декілка нових под:

```
default       mysqlcluster-sample-mysql-0                1/1     Running   0          23s
default       mysqlcluster-sample-terminal               1/1     Running   0          23s
default       mysqlterminal-sample-terminal              1/1     Running   0          22s
```

mysqlcluster-sample-mysql-0 – под Mysql кластеру

mysqlcluster-sample-terminal – под локального терміналу, що був зазначений у специфікації MysqlCluster

mysqlterminal-sample-terminal – под локального терміналу, щ обув створений окремо.

Для отримання доступу до MySQL Shell, виконуємо команду:

kubectl exec -it mysqlterminal-sample-terminal -- mysql -p

mysql> create database test;

Query OK, 1 row affected (0.01 sec)

Отже, маємо працюючий екземпляр MySQL.

Висновки

В сучасний час хмарні обчислення – величезна галузь.

Технології контейнеризації – приклад технологій, що змінюють поглядь на звичайну архітектуру додатків.

Завдяки контейнерам, збільшується безпека додатків за допомогою відділення середовища виконання.

Завдяки інструментам оркестрації як Kubernetes, ми отримуємо більш надійний засіб деплойменту контейнеризованих додатків, ніж традиційний. При цьому зменшується кількість зусиль для налаштування кластеру за допомогою оренди вже готового кластеру.

Технології Kubernetes виконують також роль розподільника навантаження, а також підвищення загальної доступності серед світу.

Офіційні доповнення, у тому числі Деплоймент та StatefulSet надають широкий спектр можливостей.

Окрім того, Kubernetes надає велику кількість можливостей для розширення функціоналу, що дозволяє створювати та використовувати неофіційні оператори.

У цій дипломний роботі було розроблено Kubernetes оператор для деплойменту MySQL кластеру.

Хоча акцент був спрямован саме на аналіз Kubernetes та розробку оператору, потенціал цього продукту може бути розширений. Наступною задачою після закінчення цієї дипломної роботи для мене буде використання більш продвинутих технологій роботи з MySQL, у тому числі Orchestrator для більш продуктивного та безпечного масштабування MySQL кластерів. Таким чином, я вважаю, що на

наступну дипломну роботу в мене буде можливість обрати тему MySQL оператору з акцентом вже на богатому функціоналі самого оператору.

Перелік посилань

1. https://github.com/aleshchynskyi/kube-mysql
2. https://kubernetes.io/
3. https://www.docker.com/resources/what-container
4. https://www.redhat.com/en/topics/containers
5. https://en.wikipedia.org/wiki/Cgroups
6. https://sdk.operatorframework.io/

ДОДАТОК А

Програмний код

api/v1alpha1/mysqlcluster_types.go

```
package v1alpha1

import (
	v1 "k8s.io/api/core/v1"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// EDIT THIS FILE!  THIS IS SCAFFOLDING FOR YOU TO OWN!
// NOTE: json tags are required.  Any new fields you add must have json tags for the fields to be serialized.

type MysqlClusterSpecConfig struct {
	Name string          `json:"name,omitempty"`
	Spec *MysqlConfigSpec `json:"spec,omitempty"`
}

type MysqlClusterStorage struct {
	VolumeSource *v1.VolumeSource         `json:"volumeSource,omitempty"`
	Resources    *v1.ResourceRequirements `json:"resources,omitempty"`
}

type MysqlClusterTerminal struct {
}

// MysqlClusterSpec defines the desired state of MysqlCluster
type MysqlClusterSpec struct {
	Config   MysqlClusterSpecConfig `json:"config,omitempty"`
	Replicas *int32                 `json:"replicas,omitempty"`
	Storage  MysqlClusterStorage    `json:"storage,omitempty"`
	Terminal *MysqlClusterTerminal  `json:"terminal,omitempty"`
}

// MysqlClusterStatus defines the observed state of MysqlCluster
type MysqlClusterStatus struct {
	ConfigSpec         MysqlConfigSpec `json:"configSpec,omitempty"`
```

```
	StatefulSet		string		`json:"statefulSet,omitempty"`
	Service		string		`json:"service,omitempty"`
	StorageVolumeSource v1.VolumeSource `json:"StorageVolumeSource,omitempty"`

	// INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
	// Important: Run "make" to regenerate code after modifying this file
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status

// MysqlCluster is the Schema for the mysqlclusters API
type MysqlCluster struct {
	metav1.TypeMeta   `json:",inline"`
	metav1.ObjectMeta `json:"metadata,omitempty"`

	Spec   MysqlClusterSpec   `json:"spec,omitempty"`
	Status MysqlClusterStatus `json:"status,omitempty"`
}

//+kubebuilder:object:root=true

// MysqlClusterList contains a list of MysqlCluster
type MysqlClusterList struct {
	metav1.TypeMeta `json:",inline"`
	metav1.ListMeta `json:"metadata,omitempty"`
	Items		[]MysqlCluster `json:"items"`
}

func init() {
	SchemeBuilder.Register(&MysqlCluster{}, &MysqlClusterList{})
}
```

api/v1alpha1/mysqlconfig_types.go

```
package v1alpha1

import (
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
```

```
)

// EDIT THIS FILE!  THIS IS SCAFFOLDING FOR YOU TO OWN!
// NOTE: json tags are required.  Any new fields you add must have json tags for the fields to be serialized.

// MysqlConfigSpec defines the desired state of MysqlConfig
type MysqlConfigSpec struct {
	Images map[string]string `json:"images,omitempty"`
}

// MysqlConfigStatus defines the observed state of MysqlConfig
type MysqlConfigStatus struct {
	// INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
	// Important: Run "make" to regenerate code after modifying this file
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status
//+kubebuilder:resource:scope=Cluster

// MysqlConfig is the Schema for the mysqlconfigs API
type MysqlConfig struct {
	metav1.TypeMeta   `json:",inline"`
	metav1.ObjectMeta `json:"metadata,omitempty"`

	Spec   MysqlConfigSpec   `json:"spec,omitempty"`
	Status MysqlConfigStatus `json:"status,omitempty"`
}

//+kubebuilder:object:root=true

// MysqlConfigList contains a list of MysqlConfig
type MysqlConfigList struct {
	metav1.TypeMeta `json:",inline"`
	metav1.ListMeta `json:"metadata,omitempty"`
	Items           []MysqlConfig `json:"items"`
}

func init() {
```

```
        SchemeBuilder.Register(&MysqlConfig{}, &MysqlConfigList{})
}
```

api/v1alpha1/mysqlterminal_types.go

```
package v1alpha1

import (
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

// EDIT THIS FILE!  THIS IS SCAFFOLDING FOR YOU TO OWN!
// NOTE: json tags are required.  Any new fields you add must have json tags for the fields to be serialized.

type MysqlTerminalCluster struct {
        Name      string `json:"name"`
        Namespace string `json:"namespace,omitempty"`
}

// MysqlTerminalSpec defines the desired state of MysqlTerminal
type MysqlTerminalSpec struct {
        Cluster MysqlTerminalCluster `json:"cluster"`
}

// MysqlTerminalStatus defines the observed state of MysqlTerminal
type MysqlTerminalStatus struct {
        // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
        // Important: Run "make" to regenerate code after modifying this file
}

//+kubebuilder:object:root=true
//+kubebuilder:subresource:status

// MysqlTerminal is the Schema for the mysqlterminals API
type MysqlTerminal struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   MysqlTerminalSpec   `json:"spec,omitempty"`
```

```
	Status MysqlTerminalStatus `json:"status,omitempty"`
}

//+kubebuilder:object:root=true

// MysqlTerminalList contains a list of MysqlTerminal
type MysqlTerminalList struct {
	metav1.TypeMeta `json:",inline"`
	metav1.ListMeta `json:"metadata,omitempty"`
	Items           []MysqlTerminal `json:"items"`
}

func init() {
	SchemeBuilder.Register(&MysqlTerminal{}, &MysqlTerminalList{})
}
```

controllers/cluster_terminal_builder.go

```
package controllers

import (
	"context"
	kubesqlv1alpha1 "github.com/vellanci/kube-mysql.git/api/v1alpha1"
	"k8s.io/apimachinery/pkg/api/equality"
	"k8s.io/apimachinery/pkg/api/errors"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
	"sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"
)

func (r *MysqlClusterReconciler) deployTerminal(ctx context.Context, cluster *kubesqlv1alpha1.MysqlCluster) error {
	terminal := buildTerminal(cluster)
	if err := r.SetControllerReference(ctx, cluster, terminal); err != nil {
		return err
	}
	_, err := r.CreateOrUpdate(ctx, terminal, func() error {
		return nil
	})
	if err != nil {
```

```
                return err
        }
        return nil
}

func buildTerminal(cluster *kubesqlv1alpha1.MysqlCluster) *kubesqlv1alpha1.MysqlTerminal {
        return &kubesqlv1alpha1.MysqlTerminal{
                ObjectMeta: metav1.ObjectMeta{
                        Name:      cluster.Name,
                        Namespace: cluster.Namespace,
                },
                Spec: kubesqlv1alpha1.MysqlTerminalSpec{
                        Cluster: kubesqlv1alpha1.MysqlTerminalCluster{
                                Name: cluster.Name,
                        },
                },
        }
}

func (r *MysqlClusterReconciler) undeployTerminal(ctx context.Context, cluster *kubesqlv1alpha1.MysqlCluster) error {
        template := buildTerminal(cluster)
        if err := r.SetControllerReference(ctx, cluster, template); err != nil {
                return err
        }
        logger := controllerruntime.LoggerFrom(ctx, "terminal", template.Name)
        found := &kubesqlv1alpha1.MysqlTerminal{}
        key := client.ObjectKey{Namespace: template.Namespace, Name: template.Name}
        if err := r.Get(ctx, key, found); err != nil {
                if errors.IsNotFound(err) {
                        return nil
                }
                logger.Error(err, "Failed to get terminal")
                return err
        }
        if !equality.Semantic.DeepEqual(found.OwnerReferences, template.OwnerReferences) {
                // If terminal doesn't belong to cluster ignore
                return nil
        }
        if found.ObjectMeta.DeletionTimestamp != nil {
```

```
                        // If terminal is being terminated ignore
                        return nil
                }
                logger.Info("Terminal per cluster is disabled but terminal object exists, removing")
                if err := r.Delete(ctx, found); err != nil {
                        logger.Error(err, "Failed to delete terminal")
                        return err
                }
                return nil
}
```

controllers/config.go

```
package controllers

import (
                "context"
                mysqlalpha1 "github.com/vellanci/kube-mysql.git/api/v1alpha1"
                "k8s.io/apimachinery/pkg/api/errors"
                "k8s.io/apimachinery/pkg/types"
                ctrl "sigs.k8s.io/controller-runtime"
)

func (r *MysqlClusterReconciler) GetClusterConfig(ctx context.Context, cluster *mysqlalpha1.MysqlCluster)
(*mysqlalpha1.MysqlConfigSpec, error) {
                referredMysqlConfig, err := r.GetMysqlConfig(ctx, cluster.Spec.Config.Name)
                if err != nil {
                        return nil, err
                }
                currentConfigSpec := MergeConfigs(referredMysqlConfig, cluster.Spec.Config.Spec)
                return currentConfigSpec, nil
}

func (r *MysqlClusterReconciler) GetMysqlConfig(ctx context.Context, configName string) (*mysqlalpha1.MysqlConfigSpec,
error) {
                config := &mysqlalpha1.MysqlConfig{}
                if configName == "" {
                        return &config.Spec, nil
                }
```

```
	logger := ctrl.LoggerFrom(ctx)
	if err := r.Get(ctx, types.NamespacedName{Name: configName}, config); err != nil {
		if errors.IsNotFound(err) {
			logger.Error(err, "Cannot find config", "name", configName)
			return &config.Spec, err
		}
		return &config.Spec, err
	}
	return &config.Spec, nil
}

func MergeConfigs(configs ...*mysqlalpha1.MysqlConfigSpec) *mysqlalpha1.MysqlConfigSpec {
	var result = &mysqlalpha1.MysqlConfigSpec{
		Images: map[string]string{},
	}
	for _, next := range configs {
		if next == nil {
			continue
		}
		for name, image := range next.Images {
			result.Images[name] = image
		}
	}
	return result
}
```

controllers/controller_tools.go

```
package controllers

import (
	"context"
	"github.com/go-logr/logr"
	"k8s.io/apimachinery/pkg/runtime"
	ctrl "sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"
	"sigs.k8s.io/controller-runtime/pkg/controller/controllerutil"
)
```

```
type CommonReconciler struct {
	client.Client
	Log    logr.Logger
	Scheme *runtime.Scheme
}

func (r *CommonReconciler) CreateOrUpdate(ctx context.Context, object client.Object, fn func() error)
(controllerutil.OperationResult, error) {
	logger := r.loggerForObject(ctx, object)
	result, err := ctrl.CreateOrUpdate(ctx, r.Client, object, fn)
	if err != nil {
		logger.Error(err, "Operation failed", "result", result)
		return result, err
	}
	if result != controllerutil.OperationResultNone {
		logger.Info("Operation succeeded", "result", result)
	}
	return result, nil
}

func (r *CommonReconciler) loggerForObject(ctx context.Context, object client.Object) logr.Logger {
	kinds, _, err := r.Scheme.ObjectKinds(object)
	if len(kinds) < 1 {
		logger := ctrl.LoggerFrom(ctx, "name", object.GetName())
		logger.Error(err, "Cannot find a kind of an object")
		return logger
	}
	return ctrl.LoggerFrom(ctx,
		"kind", kinds[0].Kind,
		"name", object.GetName(),
	)
}

func (r CommonReconciler) SetControllerReference(ctx context.Context, owner client.Object, object client.Object) error {
	if err := ctrl.SetControllerReference(owner, object, r.Scheme); err != nil {
		ctrl.LoggerFrom(ctx).Error(err, "Cannot set controller reference")
		return err
	}
	return nil
```

```
}
```

controllers/mysqlcluster_controller.go

```
package controllers

import (
	"context"
	kubesqlv1alpha1 "github.com/vellanci/kube-mysql.git/api/v1alpha1"
	appsv1 "k8s.io/api/apps/v1"
	corev1 "k8s.io/api/core/v1"
	"k8s.io/apimachinery/pkg/api/equality"
	"k8s.io/apimachinery/pkg/api/errors"
	ctrl "sigs.k8s.io/controller-runtime"
)

// MysqlClusterReconciler reconciles a MysqlCluster object
type MysqlClusterReconciler struct {
	CommonReconciler
}

//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlclusters,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlclusters/status,verbs=get;update;patch
//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlclusters/finalizers,verbs=update
//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlconfigs,verbs=get;list;watch
//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlterminals,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=apps,resources=statefulsets,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=core,resources=services,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=core,resources=persistentvolumeclaims,verbs=get;list;watch;create;update;patch;delete

func (r *MysqlClusterReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
	return ctrl.Result{}, r.reconcileWithoutResult(ctx, req)
}
func (r *MysqlClusterReconciler) reconcileWithoutResult(ctx context.Context, req ctrl.Request) error {
	logger := ctrl.LoggerFrom(ctx)
	objectKey := req.NamespacedName
	currentInstance := &kubesqlv1alpha1.MysqlCluster{}
	if err := r.Get(ctx, objectKey, currentInstance); err != nil {
		if errors.IsNotFound(err) {
```

```
                        logger.Info("Cluster is getting deleted")
                        return nil
                }
                logger.Error(err, "Cannot get cluster CR")
                return err
}
cluster := currentInstance.DeepCopy()

currentConfigSpec, err := r.GetClusterConfig(ctx, cluster)
if err != nil {
                return err
}
cluster.Status.ConfigSpec = *currentConfigSpec

err = r.deployMysqlCluster(ctx, cluster)
if err != nil {
                return err
}

if cluster.Spec.Terminal != nil {
                if err := r.deployTerminal(ctx, cluster); err != nil {
                        return err
                }
} else {
                if err := r.undeployTerminal(ctx, cluster); err != nil {
                        return err
                }
}

if !equality.Semantic.DeepEqual(currentInstance.Status, cluster.Status) {
                if err := r.Status().Update(ctx, cluster); err != nil {
                        logger.Error(err, "Cannot update cluster status")
                        return err
                }
                logger.Info("Updated cluster status")
                currentInstance.Status = cluster.Status
}

return nil
```

```
}

func (r *MysqlClusterReconciler) deployMysqlCluster(ctx context.Context, cluster *kubesqlv1alpha1.MysqlCluster) error {
	_, err := r.CreateOrUpdatePVC(ctx, cluster)
	if err != nil {
		return err
	}

	_, err = r.createOrUpdateService(ctx, cluster)
	if err != nil {
		return err
	}

	_, err = r.createOrUpdateSet(ctx, cluster)
	if err != nil {
		return err
	}
	return nil
}

func buildLabels(cluster *kubesqlv1alpha1.MysqlCluster) map[string]string {
	return map[string]string{
		"vellanci.gh/mysql-cluster": cluster.Name,
	}
}

// SetupWithManager sets up the controller with the Manager.
func (r *MysqlClusterReconciler) SetupWithManager(mgr ctrl.Manager) error {
	return ctrl.NewControllerManagedBy(mgr).
		For(&kubesqlv1alpha1.MysqlCluster{}).
		Owns(&corev1.PersistentVolumeClaim{}).
		Owns(&corev1.Service{}).
		Owns(&appsv1.StatefulSet{}).
		Owns(&corev1.Pod{}).
		Complete(r)
}
```

controllers/mysqlterminal_controller.go

```
package controllers

import (
	"context"
	corev1 "k8s.io/api/core/v1"
	"k8s.io/apimachinery/pkg/api/equality"
	"k8s.io/apimachinery/pkg/api/errors"

	ctrl "sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"

	kubesqlv1alpha1 "github.com/vellanci/kube-mysql.git/api/v1alpha1"
)

// MysqlTerminalReconciler reconciles a MysqlTerminal object
type MysqlTerminalReconciler struct {
	CommonReconciler
}

//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlterminals,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlterminals/status,verbs=get;update;patch
//+kubebuilder:rbac:groups=kubesql.vellanci.gh,resources=mysqlterminals/finalizers,verbs=update
//+kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=core,resources=secrets,verbs=get;list;watch;create;update;patch;delete

// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the MysqlTerminal object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
//
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.7.2/pkg/reconcile
func (r *MysqlTerminalReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
	return ctrl.Result{}, r.reconcileWithoutResult(ctx, req)
}

// SetupWithManager sets up the controller with the Manager.
```

```
func (r *MysqlTerminalReconciler) SetupWithManager(mgr ctrl.Manager) error {
	return ctrl.NewControllerManagedBy(mgr).
		For(&kubesqlv1alpha1.MysqlTerminal{}).
		Owns(&corev1.Secret{}).
		Owns(&corev1.Pod{}).
		Complete(r)
}

func (r *MysqlTerminalReconciler) reconcileWithoutResult(ctx context.Context, req ctrl.Request) error {
	logger := ctrl.LoggerFrom(ctx)
	objectKey := req.NamespacedName
	currentInstance := &kubesqlv1alpha1.MysqlTerminal{}
	if err := r.Get(ctx, objectKey, currentInstance); err != nil {
		if errors.IsNotFound(err) {
			logger.Info("Terminal is getting deleted")
			return nil
		}
		logger.Error(err, "Failed to get terminal CR")
		return err
	}
	terminal := currentInstance.DeepCopy()

	cluster := &kubesqlv1alpha1.MysqlCluster{}
	clusterId := client.ObjectKey{
		Namespace: terminal.Spec.Cluster.Namespace,
		Name:      terminal.Spec.Cluster.Name,
	}
	if clusterId.Namespace == "" {
		clusterId.Namespace = terminal.Namespace
	}
	if err := r.Get(ctx, clusterId, cluster); err != nil {
		if errors.IsNotFound(err) {
			logger.Error(err, "Cluster not found", "id", clusterId)
			return err
		}
		logger.Error(err, "Failed to get MySQL cluster")
		return err
	}
```

```
		secret, err := r.deployTerminalConf(ctx, cluster, terminal)
		if err != nil {
			return err
		}

		if err := r.deployTerminalPod(ctx, cluster, secret, terminal); err != nil {
			return err
		}

		if !equality.Semantic.DeepEqual(currentInstance.Status, terminal.Status) {
			if err := r.Status().Update(ctx, terminal); err != nil {
				logger.Error(err, "Cannot update terminal status")
				return err
			}
			logger.Info("Updated terminal status")
			currentInstance.Status = terminal.Status
		}

		return nil
}
```

controllers/pvc_builder.go

```
package controllers

import (
		"context"
		"github.com/vellanci/kube-mysql.git/api/v1alpha1"
		corev1 "k8s.io/api/core/v1"
		"k8s.io/apimachinery/pkg/api/resource"
		metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

func buildPVC(cluster *v1alpha1.MysqlCluster) *corev1.PersistentVolumeClaim {
		return &corev1.PersistentVolumeClaim{
			ObjectMeta: metav1.ObjectMeta{
				Name:      cluster.Name + "-mysql",
				Namespace: cluster.Namespace,
				Labels:    buildLabels(cluster),
```

```
                },
                Spec: corev1.PersistentVolumeClaimSpec{
                        AccessModes: []corev1.PersistentVolumeAccessMode{
                                corev1.ReadWriteOnce,
                        },
                        Resources: corev1.ResourceRequirements{
                                Requests: map[corev1.ResourceName]resource.Quantity{
                                        corev1.ResourceStorage: resource.MustParse("1Gi"),
                                },
                        },
                },
        }
}

func updatePVC(cluster *v1alpha1.MysqlCluster, pvc *corev1.PersistentVolumeClaim) error {
        if cluster.Spec.Storage.Resources != nil {
                pvc.Spec.Resources = *cluster.Spec.Storage.Resources
        }
        return nil
}

func (r *MysqlClusterReconciler) CreateOrUpdatePVC(ctx context.Context, cluster *v1alpha1.MysqlCluster)
(*corev1.PersistentVolumeClaim, error) {
        if cluster.Spec.Storage.VolumeSource != nil {
                return nil, nil
        }

        pvc := buildPVC(cluster)
        if err := r.SetControllerReference(ctx, cluster, pvc); err != nil {
                return nil, err
        }
        _, err := r.CreateOrUpdate(ctx, pvc, func() error {
                return updatePVC(cluster, pvc)
        })
        cluster.Status.StorageVolumeSource = corev1.VolumeSource{
                PersistentVolumeClaim: &corev1.PersistentVolumeClaimVolumeSource{
                        ClaimName: pvc.Name,
                },
        }
```

```
	return pvc, err
}
```

controllers/service_builder.go

```
package controllers

import (
	"context"
	"github.com/vellanci/kube-mysql.git/api/v1alpha1"
	"k8s.io/api/core/v1"
	"sigs.k8s.io/controller-runtime"
)

func buildService(cluster *v1alpha1.MysqlCluster) *v1.Service {
	return &v1.Service{
		ObjectMeta: controllerruntime.ObjectMeta{
			Name:      cluster.Name + "-mysql",
			Namespace: cluster.Namespace,
			Labels:    buildLabels(cluster),
		},
		Spec: v1.ServiceSpec{
			Ports: []v1.ServicePort{
				{
					Name: "mysql",
					Port: 3306,
				},
			},
			Selector:  buildLabels(cluster),
			ClusterIP: v1.ClusterIPNone,
			Type:      v1.ServiceTypeClusterIP,
		},
	}
}

func updateService(_ *v1alpha1.MysqlCluster, service *v1.Service) error {
	service.Spec.Ports = []v1.ServicePort{
		{
			Name: "mysql",
```

```
                                Port: 3306,
                        },
                }
                return nil
}

func (r *MysqlClusterReconciler) createOrUpdateService(ctx context.Context, cluster *v1alpha1.MysqlCluster) (*v1.Service,
error) {
                service := buildService(cluster)
                if err := r.SetControllerReference(ctx, cluster, service); err != nil {
                        return nil, err
                }
                _, err := r.CreateOrUpdate(ctx, service, func() error {
                        return updateService(cluster, service)
                })
                if err != nil {
                        return nil, err
                }
                cluster.Status.Service = service.Name
                return service, err
}
```

controllers/set_builder.go

```
package controllers

import (
                "context"
                "github.com/vellanci/kube-mysql.git/api/v1alpha1"
                appsv1 "k8s.io/api/apps/v1"
                corev1 "k8s.io/api/core/v1"
                metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
                ctrl "sigs.k8s.io/controller-runtime"
)

func buildStatefulSet(cluster *v1alpha1.MysqlCluster) *appsv1.StatefulSet {
                return &appsv1.StatefulSet{
                        ObjectMeta: ctrl.ObjectMeta{
                                Name:      cluster.Name + "-mysql",
```

```
                Namespace: cluster.Namespace,
                Labels:    buildLabels(cluster),
            },
            Spec: appsv1.StatefulSetSpec{
                Template: corev1.PodTemplateSpec{
                    ObjectMeta: ctrl.ObjectMeta{
                        Labels: buildLabels(cluster),
                    },
                },
                Selector: &metav1.LabelSelector{
                    MatchLabels: buildLabels(cluster),
                },
                ServiceName: cluster.Status.Service,
            },
        }
}

func updateStatefulSet(cluster *v1alpha1.MysqlCluster, set *appsv1.StatefulSet) error {
        if cluster.Spec.Replicas != nil {
            set.Spec.Replicas = cluster.Spec.Replicas
        }
        set.Spec.Template.Spec = buildPodSpec(cluster)
        return nil
}

func buildPodSpec(cluster *v1alpha1.MysqlCluster) corev1.PodSpec {
        return corev1.PodSpec{
            Containers: []corev1.Container{
                {
                    Name:  "mysql",
                    Image: cluster.Status.ConfigSpec.Images["mysql"],
                    VolumeMounts: []corev1.VolumeMount{
                        {Name: "storage", MountPath: "/var/lib/mysql"},
                    },
                    Ports: []corev1.ContainerPort{
                        {Name: "mysql", ContainerPort: 3306},
                    },
                    Env: []corev1.EnvVar{
                        {
```

```
                                                Name:  "MYSQL_ROOT_PASSWORD",
                                                Value: "root",
                                        },
                                },
                        },
                },
                Volumes: []corev1.Volume{
                        {
                                Name:         "storage",
                                VolumeSource: cluster.Status.StorageVolumeSource,
                        },
                },
        }
}

func (r *MysqlClusterReconciler) createOrUpdateSet(ctx context.Context, cluster *v1alpha1.MysqlCluster)
(*appsv1.StatefulSet, error) {
        statefulSet := buildStatefulSet(cluster)
        if err := r.SetControllerReference(ctx, cluster, statefulSet); err != nil {
                return nil, err
        }
        _, err := r.CreateOrUpdate(ctx, statefulSet, func() error {
                return updateStatefulSet(cluster, statefulSet)
        })
        if err != nil {
                return nil, err
        }
        cluster.Status.StatefulSet = statefulSet.Name
        return statefulSet, nil
}
```

controllers/terminal_builder.go

```
package controllers

import (
        "context"
        kubesqlv1alpha1 "github.com/vellanci/kube-mysql.git/api/v1alpha1"
        corev1 "k8s.io/api/core/v1"
```

```
	ctrl "sigs.k8s.io/controller-runtime"
)

const MysqlClientConfigName = ".mylogin.cnf"

func (r *MysqlTerminalReconciler) deployTerminalConf(ctx context.Context, cluster *kubesqlv1alpha1.MysqlCluster,
terminal *kubesqlv1alpha1.MysqlTerminal) (*corev1.Secret, error) {
	secret := &corev1.Secret{
		ObjectMeta: ctrl.ObjectMeta{
			Name:      terminal.Name + "-conf",
			Namespace: terminal.Namespace,
		},
	}
	if err := r.SetControllerReference(ctx, terminal, secret); err != nil {
		return nil, err
	}
	_, err := r.CreateOrUpdate(ctx, secret, func() error {
		secret.StringData = map[string]string{
			MysqlClientConfigName: "[client]" +
				"\nuser = " + "root" +
				"\npassword = " + "root" +
				"\nhost = " + cluster.Status.Service,
		}
		return nil
	})
	if err != nil {
		return nil, err
	}
	return secret, nil
}

func (r *MysqlTerminalReconciler) deployTerminalPod(ctx context.Context, cluster *kubesqlv1alpha1.MysqlCluster, secret
*corev1.Secret, terminal *kubesqlv1alpha1.MysqlTerminal) error {
	pod := buildPod(cluster, secret, terminal)
	if err := r.SetControllerReference(ctx, terminal, pod); err != nil {
		return err
	}
	_, err := r.CreateOrUpdate(ctx, pod, func() error {
		return nil
```

```
		})
		if err != nil {
			return err
		}
		return nil
}

func (r *MysqlTerminalReconciler) buildConfSecret(cluster *kubesqlv1alpha1.MysqlCluster) (*corev1.Secret, error) {
		return &corev1.Secret{StringData: map[string]string{
			MysqlClientConfigName: "[client]" +
				"\nuser = " + "root" +
				"\npassword = " + "root" +
				"\nhost = " + cluster.Status.Service + "." + cluster.Namespace,
		}}, nil
}

func buildPod(cluster *kubesqlv1alpha1.MysqlCluster, secret *corev1.Secret, terminal *kubesqlv1alpha1.MysqlTerminal)
*corev1.Pod {
		return &corev1.Pod{
			ObjectMeta: ctrl.ObjectMeta{
				Name:      terminal.Name + "-terminal",
				Namespace: terminal.Namespace,
			},
			Spec: corev1.PodSpec{
				Containers: []corev1.Container{
					{
						Name:  "mysql",
						Image: cluster.Status.ConfigSpec.Images["mysql"],
						Env: []corev1.EnvVar{
							{
								Name:  "MYSQL_HOST",
								Value: cluster.Status.Service + "." +
cluster.Namespace,
							},
						},
						Args: []string{"sleep", "infinity"},
						VolumeMounts: []corev1.VolumeMount{
							{
								Name:      "config",
```

```
                                                MountPath: "/root",
                                            },
                                        },
                                    },
                                },
                                Volumes: []corev1.Volume{
                                    {
                                        Name: "config",
                                        VolumeSource: corev1.VolumeSource{
                                            Secret: &corev1.SecretVolumeSource{
                                                SecretName: secret.Name,
                                            },
                                        },
                                    },
                                },
                            },
                        }
                }
```