МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«До захисту допущено»
В.о. завідувача кафедри
Оксана ШОВКОПЛЯС
_____
(підпис)
14 грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»
освітньо-професійної програми «Інформатика»
на тему: Інформаційна технологія автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління
здобувача групи ІН.м-33 Якименка Івана Олександровича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

_____ Іван ЯКИМЕНКО
(підпис)

Керівник Ігор ШЕЛЕХОВ
доцент,кандидат технічних наук
_____
(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Оксана ШОВКОПЛЯС
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 «Комп'ютерні науки», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-33 Якименка Івана Олександровича

1. Тема роботи: Інформаційна технологія ……
затверджую наказом по СумДУ від *«00» листопада 2024 року № 0000-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 14 грудня 2024 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Інформаційний огляд 3) Розробка інформаційної системи 4) Тестування та оцінка результатів.*
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
6. Консультанти до проекту (роботи), із зазначенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «____» _____ 20 ___ р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Інформаційний огляд*		
3	*Розробка інформаційної системи автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління*		
4	*Тестування та аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 62 стр., 68 рис., 3 додаток, 22 використаних джерел.

Обґрунтування актуальності теми роботи – Тема даної роботи є актуальною, оскільки вона присвячена роз’язанню важливої практичних задач автоматизації процесів підготовки комп’ютерної інфраструктури, інтеграції змін до програмного забезпечення, автоматичного розгортання контейнеризованого програмного забезпечення у розподілених середовищах.

Об’єкт дослідження — процес автоматичного розгортання контейнеризованих додатків у розподілених середовищах.

Предмет дослідження — комплексна інформаційна система для розгортання та підтримки контейнеризованих додатків у розподілених середовищах.

Мета роботи — розробка нформаційної системи автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління.

Методи дослідження — аналіз літератури і документації, інструменти побудови інформаційних моделей, емпіричне тестування, симуляції сценаріїв, порівняльний аналіз.

Результати — розроблено інформаційну систему для автоматичного розгортання контейнеризованих додатків у розподілених середовищах. Розроблено додаток для демонтрації роботи системи. Проведено тестування.

ІНФОРМАЦІЙНА СИСТЕМА, АВТОМАТИЗАЦІЯ РОЗГОРТАННЯ ДОДАТКІВ, КОНТЕЙНЕРИЗАЦІЯ

ЗМІСТ

ВСТУП

Сучасні інформаційні технології на сьогоднішній день продовжують тримати високий темп розвитку. Новації у підходах та методах розробки створюють потребу у активному розвитку систем, що супровожують програмне забезпечення, пропонуючи автоматизацію для управління життєвим циклом інформаційних продуктів. З появою підходів до управління процесами розробки програмного забезпечення, таких як Scrum та Agile, кількість методологій значно збільшилася. Поява методології DevOps внесла кардинальні зміни до поглядів на сучасні підходи до проєктування, розробки, тестування та підтримки програмного забезпечення.

DevOps методологія створила значну кількість дочірніх методологій та підходів, таких як:

- GitOps — використання системи контролю версій Git для управління інфраструктурою та додатками;
- ChatOps — використання чат-ботів для управління процесами при розробці програмного забезпечення;
- SecOps — використання проактивних методів моніторингу для забезпечення найвищого рівня безпеки на рівні інфраструктури та програмного забезпечення.

Водночас, DevOps методологія увібрала у себе існуючі принципи та підходи, такі як: використання розподілених систем та платформ для розгортання програмного забезпечення, використання контейнеризації, інтеграція декларативного підходу до управління інфраструктурою (infrastructure as a code), використання автоматизації за допомогою принципів безперервної інтеграції (continuous integration, CI) та безперервної доставки/розгортання (continuous deployment/delivery, CD). Правильно спроєктована та реалізована система автоматичної інтеграції та розгортання програмних продуктів — це гарант швидкої та ефективної розробки програмного забезпечення з мінімальними потребами у ручній роботі.

Тема даної роботи є актуальною, оскільки вона присвячена роз’язанню важливої практичних задач автоматизації процесів підготовки комп’ютерної інфраструктури, інтеграції змін до програмного забезпечення, автоматичного розгортання контейнеризованого програмного забезпечення у розподілених середовищах.

Об’єктом дослідження даної роботи є процес автоматичного розгортання контейнеризованих додатків у розподілених середовищах.

Предметом дослідження є комплексна інформаційна система для розгортання та підтримки контейнеризованих додатків у розподілених середовищах.

Тема даної роботи надає широкий простір для досліджень сучасних інформаційних технологій та демонстрації отриманих знань на прикладі конкретної реалізації інформаційної технології автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління.

Дана робота складається зі вступу, інформаційного огляду, постановки задачі, вибору методу розв’язання поставленої задачі, опису програмного забезпечення інформаційної системи, висновків, списку використаних джерел та додатків.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Контейнеризація

Контейнеризація — це технологія, яка дозволяє упакувати додаток та всі його залежності (бібліотеки, конфігураційні файли, середовище виконання) в єдиний стандартний блок, що називається контейнером. Контейнери забезпечують ізоляцію додатків від операційної системи та інших додатків, що працюють на одному вузлі.

Контейнеризація як технологія стала стандартом для розповсюдження програмного забезпечення через її основні переваги:

- ізоляція контейнерів від операційної системи вузла;
- гарантія відтворюваності контейнеризованих додатків у різних середовищах;
- швидкість запуску контейнерів, обумовлена використанням ядра операційної системи вузла;
- ефективність використання ресурсів.

Контейнери широко використовуються як при локальній розробці, так і в системах автоматизації при впровадженні процесів неперервної інтеграції змін (CI) та доставки коду (CD).

1.2 Розподілені середовища

Розподілене середовище — це система, яка складається із сукупності взаємопов’язаних вузлів (серверів, комп’ютерів або контейнерів), що працюють спільно для забезпечення виконання робочих завдань. Сукупність учасників розподіленого середовища називать кластером. Використання таких середовищ допомагає ефективно розподіляти обчислювальні ресурси, підвищувати надійність системи та масштабувати її.

У розподілених середовищах обчислення можуть виконуватися паралельно, а дані зберігатися у різних географічних точках. Це підходить для масштабованих додатків, які працюють у хмарних або гібридних інфраструктурах.

Основні характеристики розподілених середовищ:

- масштабованість (здатність середовища динамічно додавати чи зменшувати кількість ресурсів залежно від навантаження);
- висока доступність (забезпечення роботи системи при виході з ладу окремих компонентів без впливу на функціональність);
- ефективне використання комп’ютерних та мережевих ресурсів;
- абстрактність (відокремлення додатків від фізичної інфраструктури через віртуалізацію або контейнеризацію);
- автоматизація (використання інструментів для автоматичного розгортання програмного забезпечення та управління системою).

Розподілені середовища є основним типом цільової платформи для сучасної розробки програмного забезпечення.

1.2 Постановка задачі

Метою роботи є створення інформаційної системи автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління.

Для досягнення поставленої мети необхідно вирішити наступні задачі:

1) обрати платформу та програмне забезпечення для реалізації інформаційної технології;
2) розробити модель інформаційної системи, яка демонструє основні компоненти та процеси;
3) розробити простий веб-додаток для демонстрації роботи інформаційної технології, контейнеризувати його;
4) реалізувати інформаційну технологію автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління;
5) провести тестування інформаційної технології автоматичного розгортання контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління.

2 ІНФОРМАЦІЙНЕ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Хмарні провайдери

Хмарні провайдери є основою для сучасних інформаційних систем та програмних продуктів. Від вибору хмарного провайдера залежить обсяг доступних функцій та сервісів, який напряму впливає на простоту реалізації інформаційних систем.

Після огляду доступних рішень для розміщення комп'ютерної інфраструктури у хмарі, можна виділити такі провайдери:

- AWS (Amazon Web Services);
- Microsoft Azure;
- GCP (Google Cloud Platform);
- Digital Ocean;
- IBM Cloud;
- Oracle Cloud.

Основними та найпопулярнішими виборами з них є AWS, Microsoft Azure та GCP. Ці рішення пропонують найбільшу кількість сервісів, але є найдорожчими у використанні та комплексними.

Для спрощення та прискорення процесу ознайомлення з хмарним провайдером, зазвичай обирають більш прості рішення, такі, як Digital Ocean. Digital Ocean пропонує усі необхідні сервіси для забезпечення функціонування комп'ютерної інфраструктури у хмарі, серед яких:

- віртуальна мережа (VPC);
- балансувальник навантаження (load balancer);
- віртуальні машини (Droplets);
- готові рішення для серверів баз даних (manged databases);
- об'єктні сховища;
- дискові сховища ;
- Kubernetes.

Після проведення аналізу доступних рішень, у якості платформи було обрано хмарний провайдер Digital Ocean.

2.2 Платформи для зберігання коду в Git

Git є основною та найбільш використовуваною розподіленою системою контролю версій. Git використовується у великій кількості платформ для управління розробкою програмних продуктів, які забезпечують можливості для контролю версій і спільної роботи над програмним кодом.

Серед найпопулярніших платформ можна виділити:

- GitLab;
- GitHub;
- BitBucket;
- Microsoft Azure DevOps;
- Amazon AWS CodeCommit.

GitHub є сучасним стандартом для розробки програмного забезпечення з відкритим вихідним кодом, що пропонує сервіс GitHub Actions у якості реалізації системи автоматизації. Такі платформи як Microsoft Azure DevOps та Amazon AWS CodeCommit є популярними рішеннями у разі використання відповідного хмарного провайдера.

Платформа GitLab є більш пристосованою для автоматизації, оскільки сервіс GitLab CI/CD є доступним без додаткових налаштувань та конфігурації, а його функціонал — найбільш гнучким. Окремо можна виділити можливість розгортання GitLab на власних серверах.

GitLab пропонує такі важливі сервіси:

- сценарії для автоматизації (GitLab CI/CD);
- сховище артефактів (GitLab Artifact Registry);
- використання оточень для програмних продуктів (GitLab environments);
- агенти для виконання задач автоматизації (GitLab runners).

Для зберігання коду та створення сценаріїв автоматизації було обрано платформу GitLab.

2.3 Технології декларативного управління інфраструктурою

З появою можливості розміщення комп'ютерної інфраструктури у хмарних оточеннях, виникла потреба у стовренні підходу до ефективного управління комплексними системами, що включають у себе велику кількість компонентів та конфігурацій, одночасно маючи можливість відтворення окремих частин інфраструктури у декількох екземплярах. Підхід «Інфраструктура як код» (Infrasctructure as code) вирішує проблему управління хмарною інфраструктурою за допомогою декларативного методу опису.

Найпопулярнішим рішенням серед декларативних інструментів опису інфраструктури кодом є HashiCorp Terraform.

В Terraform закладено принцип «стану» (Terraform State), що представлений файлом, у якому зберігається поточний список інфраструктурних ресурсів та їх конфігурацій. Terraform State може зберігатися локально, разом з кодом Terraform, або у віддаленому сховищі (Terraform Remote Backend), що дозволяє вести розробку декільком розробникам.

Взаємодія Terraform з хмарними провайдерами у більшості випадків відбувається за допомогою API, інструкції для взаємодії з API конкретної системи представлені набором модулів мови програмування Go, які об'єднані в провайдер (Terraform Provider). Бібліотека доступних провайдерів (Terraform Registry) є загальнодоступною.

Для опису інфраструктури використовується мова HCL (HashiCorp Language) та файли з розширенням «.tf». Для управління інфраструктурою за допомогою Terraform, використовують такі основні команди:

- «terraform plan» - команда для планування змін до стану інфраструктури;
- «terraform apply» - команда для застосування змін до стану

інфраструктури;

- «terraform destroy» - команда для видалення інфраструктури.

У якості інструменту для декларативного опису інфраструктури було обрано Terraform.

2.4 Розподілені середовища оркестрації

Розподілені системи оркестрації (розподілені середовища оркестрації) є одним із основних елементів для сучасної розробки програмних продуктів, так як являють собою цільові платформи для розміщення додатків та супутніх служб і націлені на забезпечення стандартизації формату розгортання, розширюваності, декларативності конфігурацій та автоматизіції.

Серед сучасних доступних систем оркестрації можна виділити наступні:

- Kubernetes;
- Docker Swarm;
- HashiCorp Nomad;
- Amazon ECS;
- Google Cloud Run;
- OpenShift.

Kubernetes — це платформа з відкритим вихідним кодом для управління контейнеризованими сервісами та супутніми службами. Можливості Kubernetes є найширшими серед інших представників, що є причиною популярності платформи.

Основними об’єктами кластеру Kubernetes є:

- Control plane;
- Worker node.

Control plane — це керуючий вузол, яикий контролює кластер, керує навантаженням та комунікацією у системі. Кластер може мати один або більше керуючих вузлів. На керуючому вузлі розміщені декілька компонентів, кожен з яких є окремим процесом та може бути запущений на одному або декількох вузлах. Компоненти Control plane:

- kube-apiserver — основний серверний компонент який надає доступ до API кластера;

- etcd — високодоступне сховище, яке використовує структуру даних «ключ-значення» для зберігання об’єктів kube-apiserver;
- kube-scheduler — планувальник для аналізу доступних ресурсів та прийняття рішень про розміщення робочих навантажень на вузлах кластеру;
- kube-controller-manager — процес, який керує вбудованими контролерами об’єктів Kubernetes.

Worker node — це робочий вузол, на якому розміщуються контейнеризовані сервіси та служби. Кожен робочий вузол має такі компоненти:

- kubelet — сервіс, що відповідає за забезпечення контролю створених контейнеризованих компонентів;
- kube-proxy — сервіс, що відповідає за мережеві конфігурації для забезпечення функціонування мережевих об’єктів;
- Container runtime — програмне забезпечення, що відповідає за запуск контейнерів.

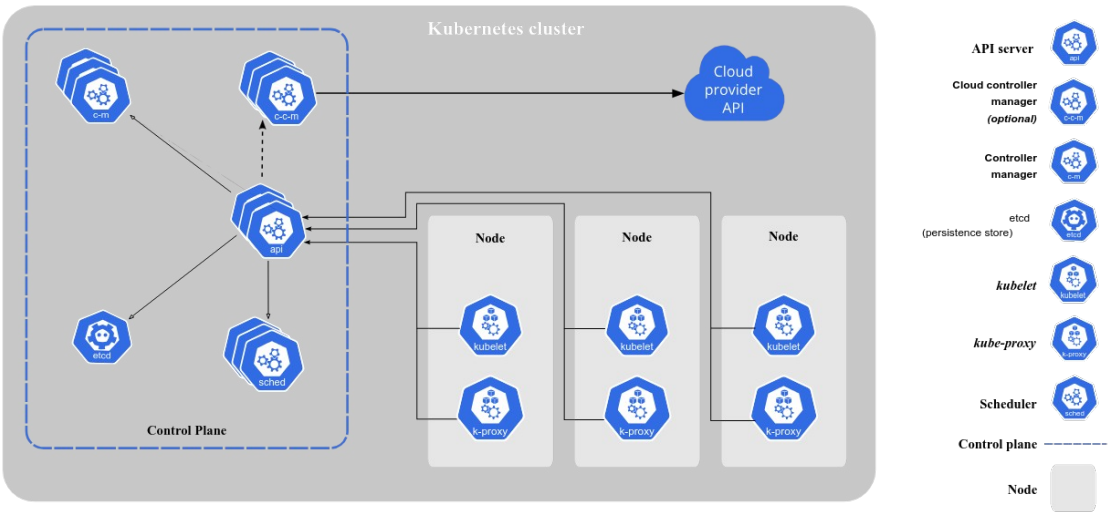


Рисунок 2.1 – Схема архітектурних компонентів кластеру Kubernetes

Об’єкти Kubernetes представлені абстракціями API, які є детальними інструкціями для Kubernetes щодо специфікацій додатків, необхідних

ресурсів, мережевих конфігурацій. Основними об’єктами є:

- Namespace — об’єкт логічного поділу для ізоляції ресурсів у кластері;
- Pod («под») — мінімальна одиниця виконання в Kubernetes, яка містить один або кілька контейнерів;
- Deployment — об’єкт для керування «подами» та оновленнями їх версій;
- ReplicaSet — об’єкт для керування реплікацією «подів»;
- PersistentVolume — зовнішнє сховище для постійного зберігання даних;
- Service — об’єкт, який забезпечує доступ до «подів» через стабільну IP-адресу або DNS;
- ConfigMap — об’єкт для збереження конфігураційних даних у форматі «ключ-значення»;
- Secret — об’єкт для зберігання конфіденційних даних.

Для опису об’єктів Kubernetes використовуються файли з розширенням «.yaml», які мають назву «маніфести». Для роботи з об’єктами Kubernetes, використовується утиліта kubectl.

Через комплексність Kubernetes, хмарні провайдери пропонують власні реалізації у вигляді готових сервісів:

- Elastic Kubernetes Service (EKS) для AWS;
- Google Kubernetes Engine (GKE) для GCP;
- Azure Kubernetes Service (AKS) для Microsoft Azure;
- Digital Ocean Kubernetes Service (DOKS) від Digital Ocean.

На основі вибору хмарного провайдера, було вирішено використати рішення Digital Ocean Kubernetes Service (DOKS) від Digital Ocean.

2.5 Декларативні системи неперервної доставки коду

Останнім етапом більшості сценаріїв неперервної інтеграції (CI) є створення артефакту. Артефактом є результат збірки програмного продукту для його подальшого тестування та розгортання у цільовому середовищі. Існує два основних варіанти реалізації наступного кроку — неперервної доставки: метод відправки (push method) та метод стягування (pull method).

Метод відправки реалізується підключенням до цільової платформи для розгортання додатків та оновленні версії програмного забезпечення. Цей підхід є простим, але його використання не забезпечує ясність та строгий контроль стану (версії) програмного забезпечення.

Для вирішення проблеми методу відправки, було створено новий підхід, декларативний метод стягування, який покладено в основу інструментів неперервної доставки (CD). Основним принципом підходу є «єдине джерело правди» (single source of truth), що найчастіше представлене репозиторієм Git та програмним забезпеченням, що відслідковує зміни в репозиторії та синхронізує стан (версію) додатку на цільовій платформі. Процес оновлення версії програмного забезпечення у такому випадку відбувається шляхом внесенням змін до «єдиного джерела правди».

Представниками декларативних систем неперервної доставки коду для Kubernetes є:

- ArgoCD;
- FluxCD.

Обидві системи є реалізацією підходу GitOps (використання Git для контролю стану об’єктів Kubernetes) та пропонують мажливості автоматичної синхронізації конфігурацій у форматі стандартних «маніфестів» Kubernetes або у форматі Helm та Kustomize.

Система FluxCD була створена раніше, незважаючи на це, ArgoCD на даний момент є найбільш популярним рішенням. Основною перевагою ArgoCD є зручність використання через інтуїтивний та функціональний

користувацький інтерфейс, що надає можливість не тільки бачити стан додатків у Kubernetes кластері, а й переглядати логи додатків, підключатися до окремих контейнерів та оновлювати конфігурацію «маніфестів».

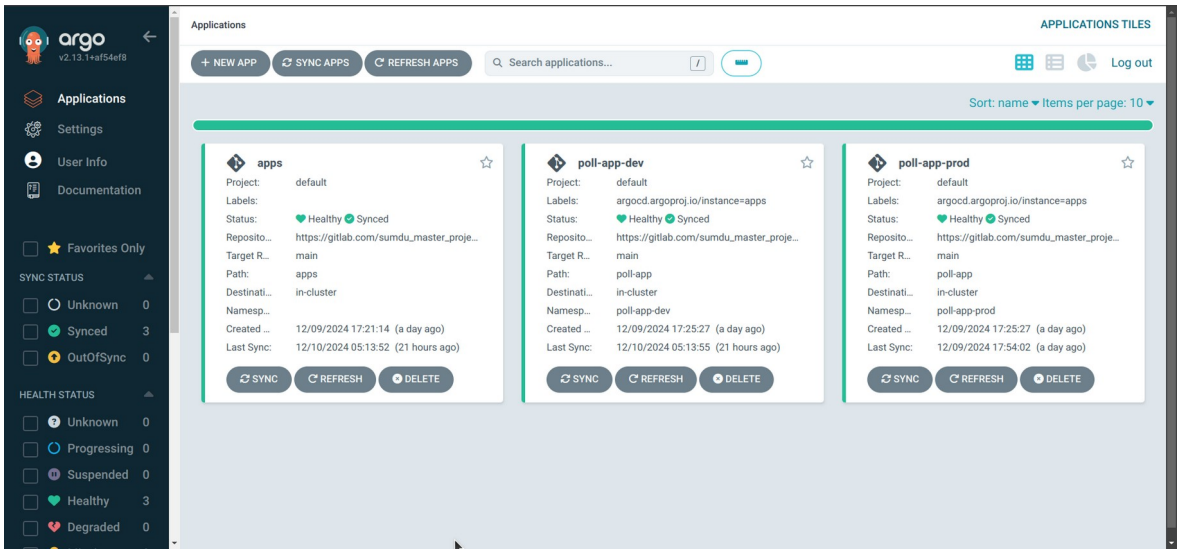


Рисунок 2.2 – Користувацький інтерфейс системи ArgoCD

У якості декларативні системи неперервної доставки коду для реалізації було обрано ArgoCD.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ДОДАТКУ

3.1 Розробка додатку

Для демонстрації роботи майбутньої інформаційної технології, необходно створити простий додаток з веб частиною, який використовує базу даних для зберігання даних.

Мова програмування python є одною із найбільш популярних мов програмування через свою простоту, велику кількість модулів та бібліотек для вирішення більшості сучасних задач. Фреймворк Django надає можливість швидко створювати веб-додатки і має широкий набір готових рішень, таких, як панель адміністратора.

Для демонстрації роботи інформаційної технології було вирішено створити веб-додаток для голосування з використанням фреймворку Django та бази даних PostgreSQL.

Додаток повинен мати панель адміністратора з можливістю додавати, оновлювати та видаляти голосування. Користувачі повинні мати можливість переглядати основну сторінку зі списком доступних голосувань, та можливість проголосувати.

Для програмної реалізації було створено основну директорію «poll_app» та ініціалізовано проєкт Django за допомогою команди «django-admin startproject pollProject». Для структуризації проєкту було вирішено відділити веб-частину від основної логіки, тому було підготовано два додатки: «pollApp» та «landingPage» за допомогою команд «python manage.py startapp pollApp» та «python manage.py startapp pollApp». Для зберігання спільних шаблонів HTML було використано окрему директорію «templates». Таким чином було створено базову структуру проєкту Django.

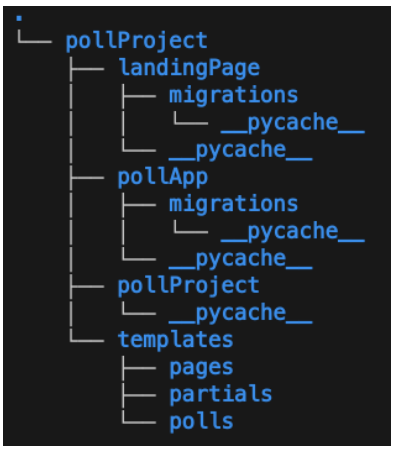


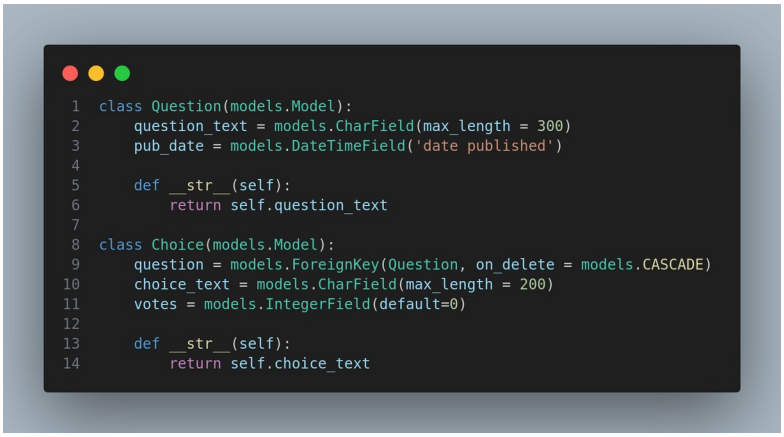
Рисунок 3.1 – Структура директорій проєкту Django

Для взаємодії з PosqreSQL, було додано відповідну конфігурацію «бекенду» до файлу налаштувань проєкту «settings.py».

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.environ.get('DATABASE_NAME'),
        "USER": os.environ.get('DATABASE_USER'),
        "PASSWORD": os.environ.get('DATABASE_PASSWORD'),
        "HOST": os.environ.get('DATABASE_HOST'),
        "PORT": os.environ.get('DATABASE_PORT'),
        "OPTIONS": {
            "sslmode": "require",
        },
    }
}
```

Рисунок 3.2 – Налаштування для роботи з базою даних PostgreSQL

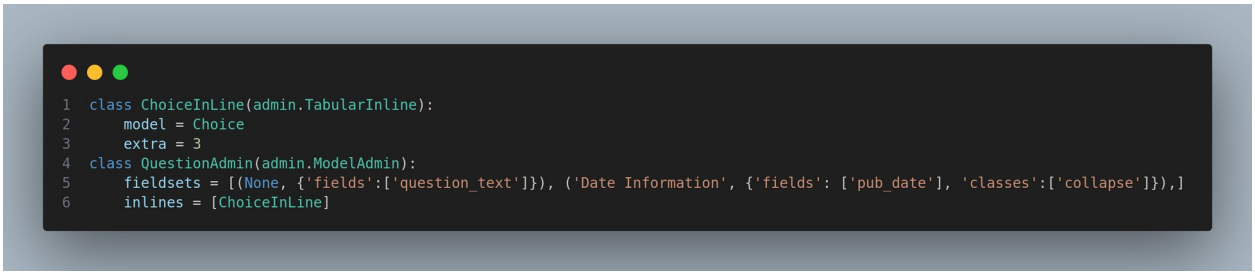
Для роботи з даними голосувань було створено два класи з використанням моделей Django: клас для запитань (Question) та клас для відповідей (Choice). Клас «запитань» містить два значення: текст питання та дата створення. Клас «відповідей» містить три значення: запитання (що представлене зовнішнім ключем класу «запитань»), текст відповіді та вибір при голосуванні (відповідь). Відповідний код класів було додано до файлу «models.py» додатку pollApp.



```python
class Question(models.Model):
    question_text = models.CharField(max_length = 300)
    pub_date = models.DateTimeField('date published')

    def __str__(self):
        return self.question_text

class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete = models.CASCADE)
    choice_text = models.CharField(max_length = 200)
    votes = models.IntegerField(default=0)

    def __str__(self):
        return self.choice_text
```

Рисунок 3.3 – Класи моделей бази даних

Для надання адміністратору можливості керувати голосуваннями, було зареєстровано базові класи Django для керування інтерфейсом адміністратора.



```python
class ChoiceInLine(admin.TabularInline):
    model = Choice
    extra = 3
class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [(None, {'fields':['question_text']}), ('Date Information', {'fields': ['pub_date'], 'classes':['collapse']}),]
    inlines = [ChoiceInLine]
```

Рисунок 3.3 – базові класи Django для керування інтерфейсом адміністратора

Для реалізації процесу голосування користувачами, було визначено відповідні функції:

- «index» — для отримання та відображення списку голосувань;
- «detail» — для відображення питання голосування та можливих відповідей;
- «results» — для отримання запитання та відображення результатів;
- «vote» — для голосування.

Код функцій було додано до файлу «views.py» додатку pollAPP. Для відображення вихідних даних було підготовано структуру HTML шаблонів. Django використовує шаблонізатор Jinja2, що дозволяє розробляти зручні шаблони для відображення даних.

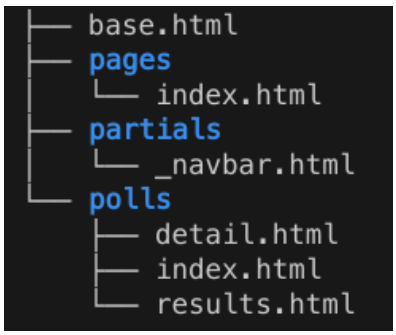
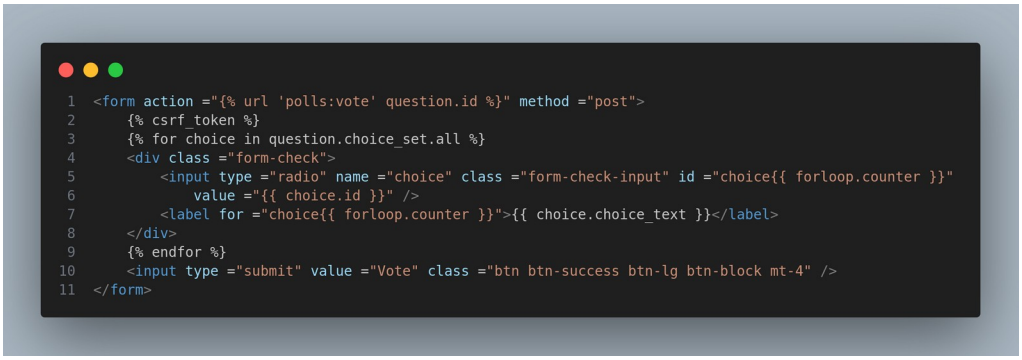


Рисунок 3.4 – Файлова структура шаблонів HTML



```
<form action ="{% url 'polls:vote' question.id %}" method ="post">
    {% csrf_token %}
    {% for choice in question.choice_set.all %}
    <div class ="form-check">
        <input type ="radio" name ="choice" class ="form-check-input" id ="choice{{ forloop.counter }}"
            value ="{{ choice.id }}" />
        <label for ="choice{{ forloop.counter }}">{{ choice.choice_text }}</label>
    </div>
    {% endfor %}
    <input type ="submit" value ="Vote" class ="btn btn-success btn-lg btn-block mt-4" />
</form>
```

Рисунок 3.5 – Частина шаблону HTML для відображення форми голосування

Django дозволяє автоматично створювати необхідні скрипти міграції за допомогою команди «python manage.py makemigrations» на основі визначених класів моделей для бази даних. Виконавши команду, було отримано відповідний файл міграцій. Для виконання міграцій використовується команда

«python manage.py migrate».

Запуск додатку локально було виконано за допомогою команди «python manage.py runserver 0.0.0.0:8080».

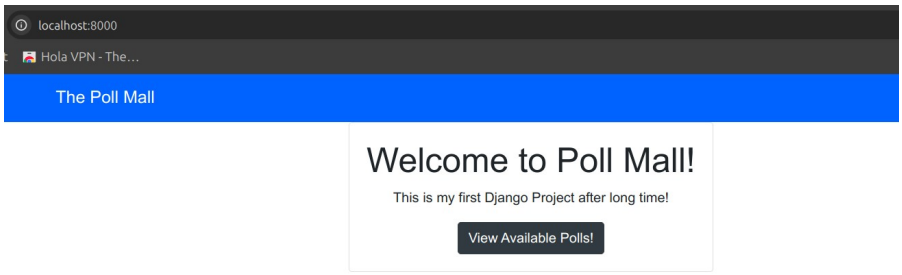


Рисунок 3.6 – Головна сторінка додатку

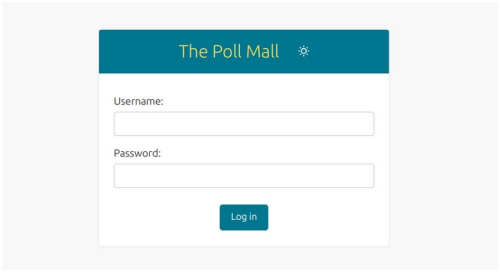


Рисунок 3.7 – Форма для входу адміністратора

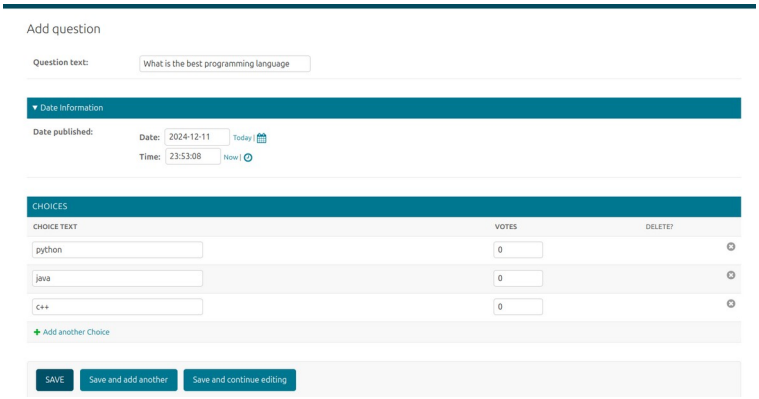


Рисунок 3.8 – Форма для додавання голосування

Рисунок 3.9 – Сторінка для голосування

3.2 Контейнеризація додатку

Для контейнеризації додатку було створено файл з назвою Dockerfile, що є стандартним іменем файлів-інструкцій для збірки образів додатків.

Для створення Docker образу додатка python, було використано базовий образ «python:3.12-slim» на базі Linux дистрибутиву Debian.

Основними етапами збірки образу для додатка python є:

- встановлення необхідних системних залежностей та пакетів;
- копіювання файлу залежностей «requirements.txt»;
- встановлення залежностей за допомогою пакетного менеджера, згідно версій, зазначених в «requirements.txt»;
- копіювання коду додатка.

Згідно з зазначеними етапами, було створено інструкцію файлу Dockerfile.

```
# Use official Python image as the base
FROM python:3.12-slim

# Set the working directory in the container
WORKDIR /app

# Install system dependencies for PostgreSQL client and other utilities
# hadolint ignore=DL3015,DL3008
RUN apt-get update && apt-get install -y \
    build-essential \
    libpq-dev \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements.txt into the container
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy the Django project into the container
COPY . .

# Expose the port the app runs on
EXPOSE 8000

# Set environment variables for Django
ENV PYTHONUNBUFFERED=1

WORKDIR /app/pollProject

# Run migrations and start the Django development server
CMD ["sh", "-c", "python manage.py runserver 0.0.0.0:8000"]
```

Рисунок 3.9 – Інструкція збірки образу Docker

4 РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ

4.1 Інфраструктурне рішення Digital Ocean

Першим кроком до створення хмарної інфраструктури є планування та визначення списку необхідних ресурсів.

Основним та центральним об’єктом хмарного проєкту є кластер Kubernetes. Для створення кластеру необхідно підготувати мережевий простір, для його створення буде використано ресурс VPC (Virtual Private Cloud). Основними конфігураціями VPC є фізичне розташування (регіон) та адресний простір. Адресний блок розміром «/16» забезпечить можливість створення усіх необхідних мережевих ресурсів для кластеру Kubernetes і надасть можливість масштабувати проєкт у майбутньому. Ресурси мережі будуть розташовані у регіоні «fra1», датацентр якого розташований у Німеччині.

DOKS (Digital Ocean Kubernetes Service) — це Kubernetes кластер, який використовує віртуальні машини (Droplets) у якості робочих вузлів. DOKS дозволяє використовувати автоматичне горизонтальне масштабування кластеру в залежності від навантаження, тому для конфігурації групи робочих вузлів було обрано такі характеристики:

- максимальна кількість вузлів у групі масштабування — 5;
- мінімальна кількість вузлів у групі масштабування — 1;
- тип віртуальної машини — s1-1vcp2-2gb.

Для зберігання даних додатку, необхідно створити сервер баз даних. Для цього було використано ресурс Managed Postgres Database, що є імплементацією PostgreSQL. Створення ресурсу передбачає створення кластеру серверів баз даних, для якого були визначені наступні характеристики:

- версія Postgres — 15;
- тип віртуальної машини — db-s-1vcpu-1gb;
- кількість вузлів у кластері — 1.

Додатковою необхідно гарантувати доступ до серверу баз даних з адресного простору кластеру Kubernetes для використання баз даних додатками. За замовченням сервер баз даних в DigitalOcean забороняє зовнішній доступ, тому необхідно додати відповідне правило firewall. Додаток буде розгорнуто в двох оточеннях, тому необхідно підготувати два сервери баз даних.

4.2 Опис інфраструктури за допомогою Terraform

Стандартна структура модулю Terraform являє собою набір файлів з розширенням «.tf». Існують стандартні назви файлів відповідно до коду, що в них додають:

- «versions.tf» - файл, у якому задається версія Terraform та версії необхідних провайдерів;
- «variables.tf» - файл для конфігурацій вхідних змінних;
- «backend.tf» - файл для розміщення конфігурації віддаленого сховища для «Terraform state»;
- «main.tf» - головний файл для опису ресурсних блоків;
- «outputs.tf» - файл, у якому зберігаються блоки вихідних даних, найчастіше це параметри та атрибути створених ресурсів.

Першим кроком у написанні коду Terraform є визначення набору необхідних провайдерів. Для управління інфраструктурою в Digital Ocean, було використано провайдер «digitalocean/digitalocean». Відповідний блок було додано до файлу «versions.tf».

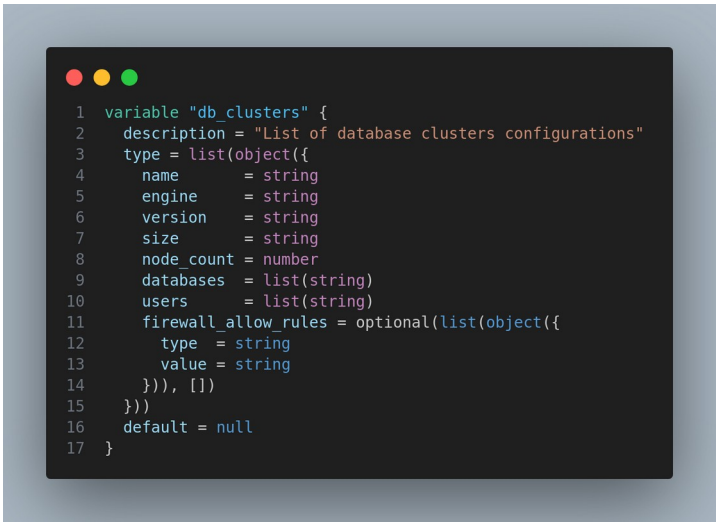
Рисунок 4.1 – Конфігурація провайдерів Terraform

Для розміщення вхідних змінних, було створено файл зі стандартним іменем «variables.tf». Для розміщення основних конфігурацій, було створено декілька файлів з інформативними назвами, що забезпечить простоту у

читанні коду . Значення для задекларованих змінних розміщуються у файлах зі специфічним розширенням «.tfvars».

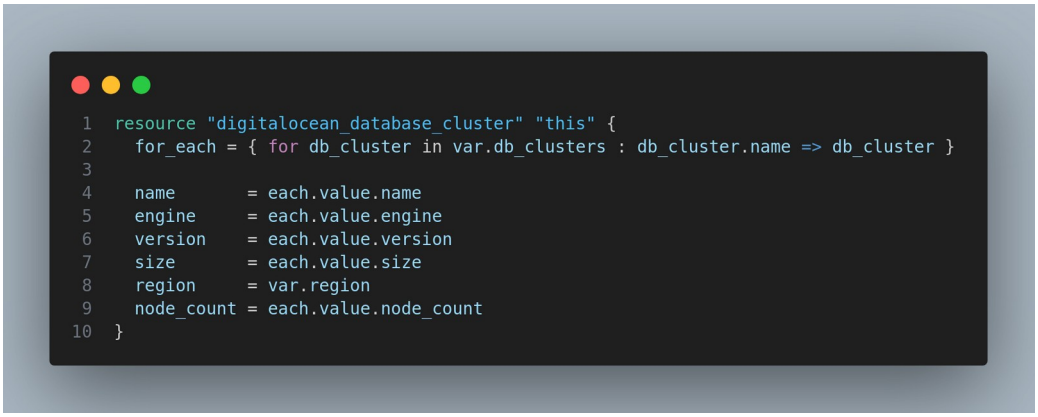
Рисунок 4.2 – Структура файлів Terraform

Для уникнення повторюваності коду, було використано мета-аргументи, динамічні блоки та трансформацію масивів даних, що дозволяє створювати декілька ресурсів за допомогою одного ресурсного блоку Terraform та комплексних вхідних змінних.



```
variable "db_clusters" {
  description = "List of database clusters configurations"
  type = list(object({
    name       = string
    engine     = string
    version    = string
    size       = string
    node_count = number
    databases  = list(string)
    users      = list(string)
    firewall_allow_rules = optional(list(object({
      type  = string
      value = string
    })), [])
  }))
  default = null
}
```

Рисунок 4.3 – Блок декларування змінної для створення кластерів баз даних та їх компонентів



```
resource "digitalocean_database_cluster" "this" {
  for_each = { for db_cluster in var.db_clusters : db_cluster.name => db_cluster }

  name       = each.value.name
  engine     = each.value.engine
  version    = each.value.version
  size       = each.value.size
  region     = var.region
  node_count = each.value.node_count
}
```

Рисунок 4.4 – Ресурсний блок для створення кластерів базданих

```
db_clusters = [
  {
    name       = "pg-poll-app-prod"
    engine     = "pg"
    version    = "15"
    size       = "db-s-1vcpu-1gb"
    node_count = 1
    databases  = ["poll-app"]
    users      = ["poll-app"]
    firewall_allow_rules = [
      {
        type  = "ip_addr"
        value = "95.47.151.190"
      }
    ]
  },
  {
    name       = "pg-poll-app-dev"
    engine     = "pg"
    version    = "15"
    size       = "db-s-1vcpu-1gb"
    node_count = 1
    databases  = ["poll-app"]
    users      = ["poll-app"]
    firewall_allow_rules = [
      {
        type  = "ip_addr"
        value = "95.47.151.190"
      }
    ]
  }
]
```

Рисунок 4.5 – Зачення змінної для створення кластерів баз даних та їх компонентів

При використанні мета-аргумету «for_each» для створення декількох ресурсів, Terraform індексує створювані об’єкти, індекси можуть бути представлені як числовими значеннями, так і словом.

Для опису необхідних інфраструктурних одиниць, було використано такі ресурси Terraform для Digital Ocean:

- «digitalocean_vpc» - ресурс для створення мережі;
- «digitalocean_kubernetes_cluster» - ресурс для створенні кластера Kubernetes;
- «digitalocean_kubernetes_node_pool» - ресурс для створення робочих вузлів кластера Kubernetes;
- «digitalocean_database_cluster» - ресурс для створення кластера серверів баз даних;
- «digitalocean_database_db» - ресурс для створення баз даних;
- «digitalocean_database_user» - ресурс для створення користувачів баз даних;
- «digitalocean_database_firewall» - ресурс для створення правил firewall для визначення доступу до серверів баз даних.

Проєкт було успішно ініціалізовано за допомогою команди «terraform init».

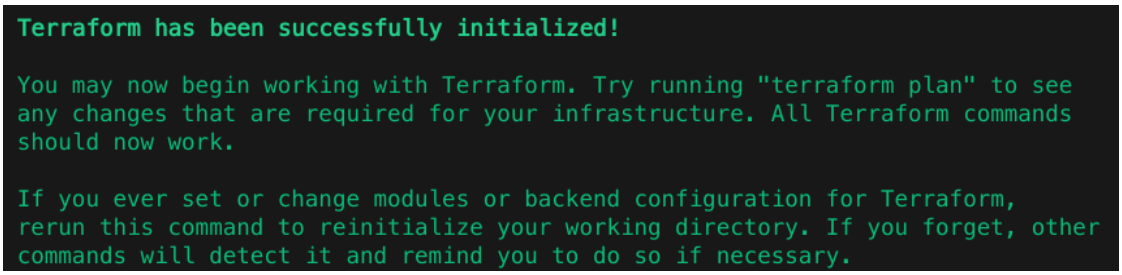


Рисунок 4.6 – Повідомлення про успішну ініціалізацію Terraform

Після застосування змін за допомогою команди «terraform apply», список створених ресурсів було переглянуто за допомогою команди «terraform state list».

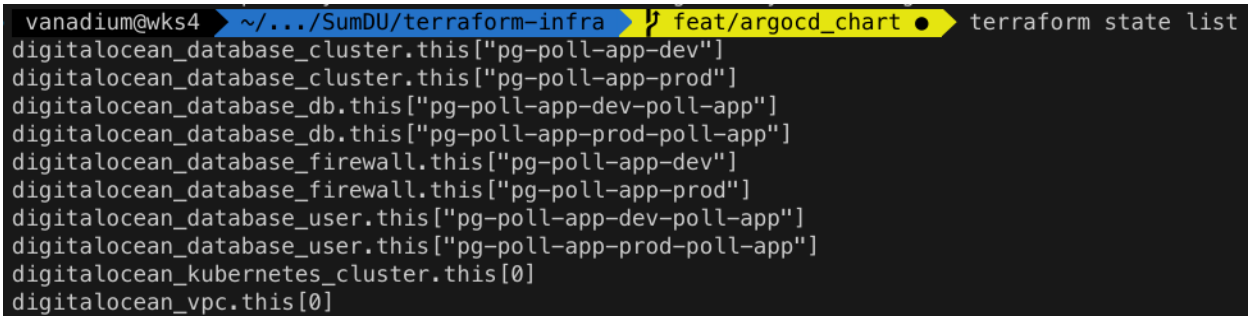


Рисунок 4.7 – Список створених ресурсів Terraform

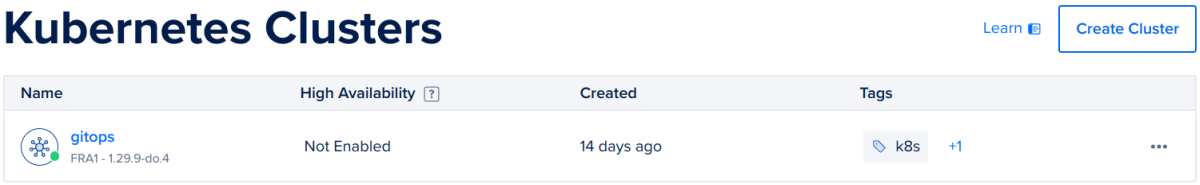


Рисунок 4.8 – Створений кластер Kubernetes

4.3 Підготовка служб та конфігурацій Kubernetes

Після того, як кластер Kubernetes було створено, необхідно перевірити його доступність. Для перевірки створених робочих вузлів та їх статусів, використано команду «kubectl get» та передано аргумент «nodes».

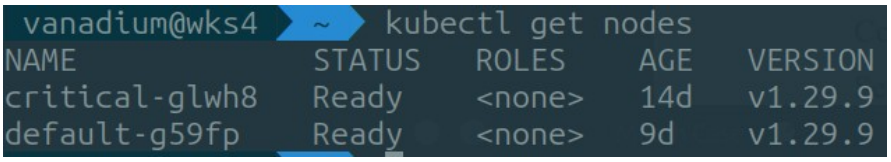


Рисунок 4.9 — Список робочих вузлів Kubernetes та їх статуси

Наступним кроком є створення моделі об’єктів Kubernetes , яка включає у себе усі необхідні об’єкти та служби.

Для реалізації розгортання додатку у кластері було обрано такі об’єкти:

- Deployment — для опису конфігурації «подів» та контролю їх версій;
- Service — для надання єдиної мережевої адреси до декількох копій «подів» у складі Deployment;
- ConfigMap та Secret — для окремого зберігання конфігурації додатку;
- HorizontalPodAutoscaler — для контролю автоматичного горизонтального масштабування «подів»;
- Ingress — для надання доступу до додатку з мережі Інтернет через інтеграцію з балансувальником навантаження провайдера Digital Ocean.

Ресурси Deployment, ConfigMap, Service та Secret є стандартними та можуть створюватися без потреби у додаткових службах.

HotizontalPodAutoscaler — це об’єкт, що надає можливість зміни кількості копій додатку в залежності від навантаження. Для оцінки навантаження, найчастіше використовуються показники використання CPU та RAM. За замовчуванням, новостворений кластер не надає можливості

отримувати ці показнки, тому для реалізації горизонтального масштабування додатку необхідно розгорнути Metrics-Server.

Ingress є ресурсом, що реалізує доступ (найчастіше HTTP) до додатків, розгорнутих у кластері. Ingress може використовувати різні реалізації балансувальників навантаження та керувати їх правилами.

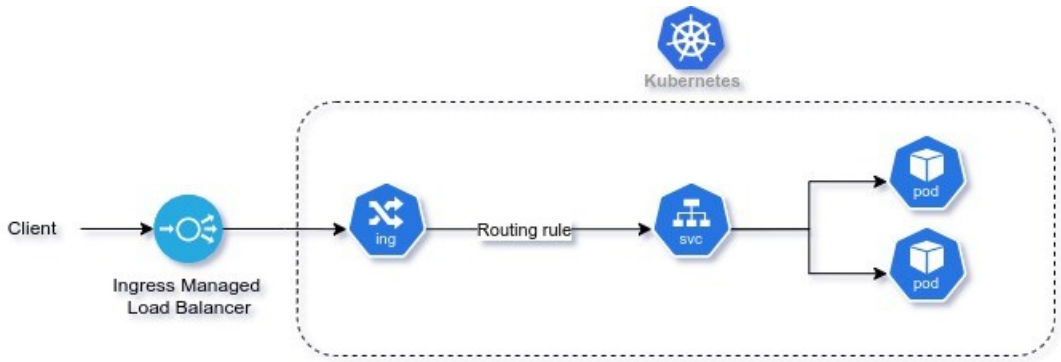


Рисунок 4.10 — Схема принципу роботи об’єкта Ingress

Для створення об’єктів Ingress, необхідно обрати контроллер Ingress, який надасть можливість створювати балансувальники навантаження в Digital Ocean та розгорнути його в Kubernetes. Класичним рішенням є NGINX Ingress Controller, який побудовано на базі проксі-сервера та вебсервера NGINX.

Невід’ємною частиною забезпечення безпеки для взаємодії з веб-додатками є використання протоколу HTTPS, що передбачає створення сертифікатів для доменних імен. Ручна робота з оновлення сертифікатів є складною та не забезпечує мінімальної вірогідності помилок, тому автоматичне управління сертифікатами є одним із важливих критеріїв для сучасної інформаційної інфраструктури. Для реалізації системи автоматичного створення та оновлення сертифікатів, необхідно розгорнути сервіс Cert-manager. У якості центру сертифікації та надання криптографічних сертифікатів для TLS-шифрування буде використано Let’s

Encrypt. Для створення доменних імен буде використано Cloud Flare DNS.

На основі отриманої інформації було створено ресурсну модель додатка в Kubernetes.

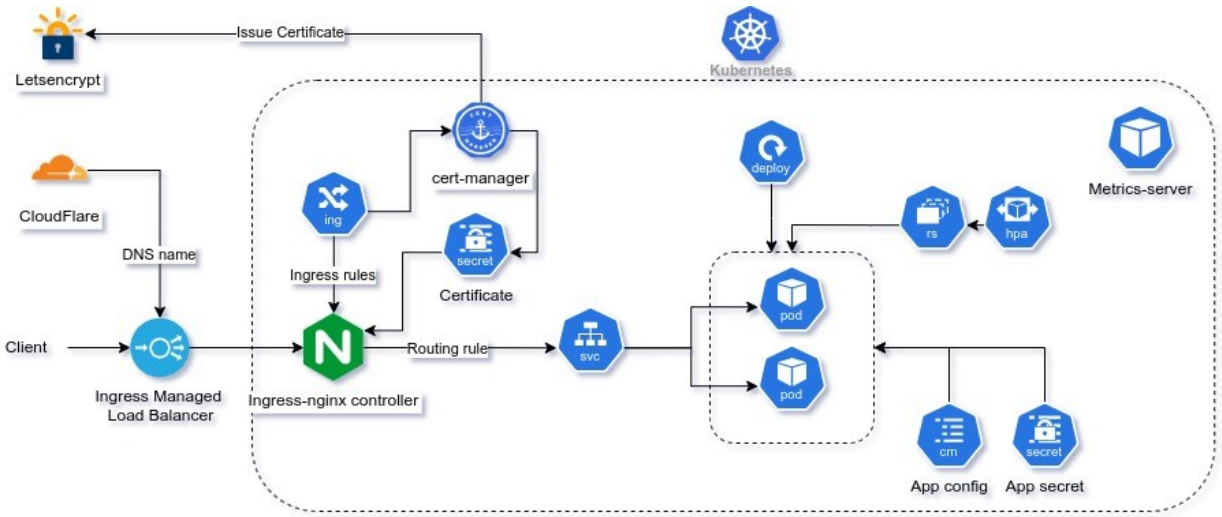


Рисунок 4.11 — Модель додатка в Kubernetes

4.4 Використання пакетного менеджера Helm

Для зручності використання та розповсюдження, програмне забезпечення в Kubernetes встановлюється за допомогою пакетного менеджера Helm. Helm дозволяє об'єднувати конфігурації об'єктів Kubernetes, які необхідні для розгортання програманого забезпечення, в сутність Helm Chart. Для гнучкості Helm пропонує використання набору шаблонів Go templates для маніфестів Kubernetes та вхідні значення для шаблонів.

Стандартна структура Helm Chart включає такі файли та дерикторії :

- файл метаданих «Chart.yaml»;
- файл значень «values.yaml»;
- директорія залежностей «charts»;
- директорія шаблонів «templates»;
- файл для виконання додаткових перетворень значень «_helpers.tpl»;
- файл для виведення важливої інформації при встановленні «NOTES.txt».

Для сервісів NGINX Ingress Controller, Cert-manager, Metrics-server існують готові пакунки Helm Chart, тому для їх розгортання було вирішено використати Helm. Сервіси є базовими для функціонування додатків кластеру, тому було прийнято рішення використати Terraform для їх автоматизованого розгортання, використовуючи провайдер «hashicorp/helm».

```
terraform {
  required_version = ">= 1.8"
  required_providers {
    digitalocean = {
      source  = "digitalocean/digitalocean"
      version = "~> 2.0"
    }
    helm = {
      source  = "hashicorp/helm"
      version = "2.16.1"
    }
  }
}
```

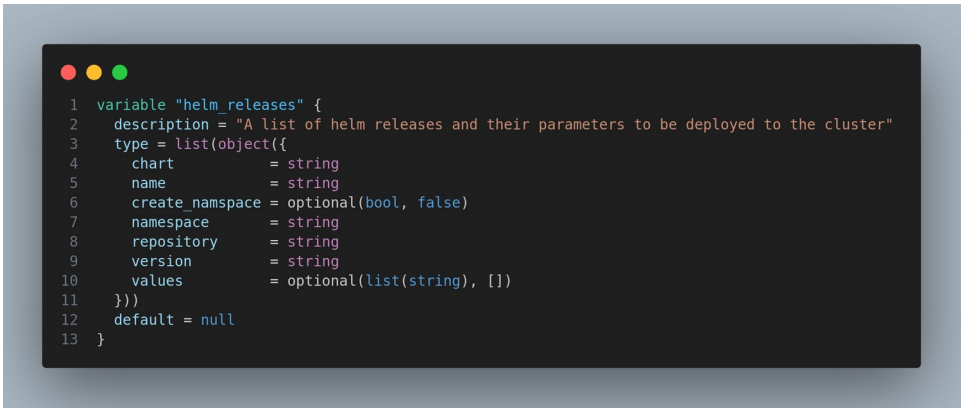
Рисунок 4.12 — Оновлена конфігурація провайдерів Terraform

Для розгортання пакунків Helm Chart за допомогою Terraform, було використано ресурс «helm_release» та реалізовано можливість перевикористання ресурсного блоку за допомогою мета-аргумента «for_each».

```
resource "helm_release" "this" {
  for_each         = var.helm_releases == null ? {} : { for helm_release in var.helm_releases : helm_release.name => helm_release }
  name             = each.value.name
  repository       = each.value.repository
  chart            = each.value.chart
  version          = each.value.version
  create_namespace = each.value.create_namspace
  namespace        = each.value.namespace
  values           = [for path in each.value.values : file(path)]
}
```

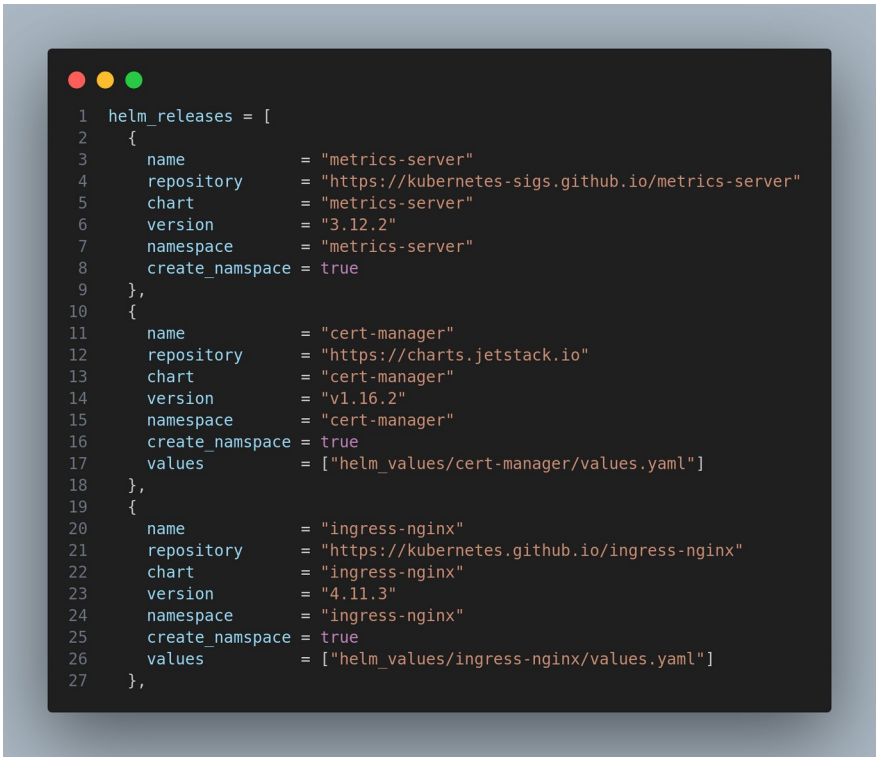
Рисунок 4.13 — Блок ресурсу Terraform для розгортання сервісів з використанням Helm

Для вхідних даних ресурсу «helm_release» було задекларовано комплексну додаткову змінну «helm_releases» та додано її значення.



```
variable "helm_releases" {
  description = "A list of helm releases and their parameters to be deployed to the cluster"
  type = list(object({
    chart           = string
    name            = string
    create_namspace = optional(bool, false)
    namespace       = string
    repository      = string
    version         = string
    values          = optional(list(string), [])
  }))
  default = null
}
```

Рисунок 4.14 — Блок змінної Terraform для вхідних даних ресурсу «helm_release»



```
helm_releases = [
  {
    name            = "metrics-server"
    repository      = "https://kubernetes-sigs.github.io/metrics-server"
    chart           = "metrics-server"
    version         = "3.12.2"
    namespace       = "metrics-server"
    create_namspace = true
  },
  {
    name            = "cert-manager"
    repository      = "https://charts.jetstack.io"
    chart           = "cert-manager"
    version         = "v1.16.2"
    namespace       = "cert-manager"
    create_namspace = true
    values          = ["helm_values/cert-manager/values.yaml"]
  },
  {
    name            = "ingress-nginx"
    repository      = "https://kubernetes.github.io/ingress-nginx"
    chart           = "ingress-nginx"
    version         = "4.11.3"
    namespace       = "ingress-nginx"
    create_namspace = true
    values          = ["helm_values/ingress-nginx/values.yaml"]
  },
```

Рисунок 4.15 — Значення змінної «helm_releases»

Для контейнеризованого додатка було розроблено та створено окремий пакет Helm Chart, який включає такі шаблони:

- «deployment.yaml»;
- «service.yaml»;
- «configmap.yaml»;
- «hpa.yaml»;
- «ingress.yaml»;
- «serviceaccount.yaml».

Конфігурації додатку буде реалізована за допомогою змінних оточення, набір яких підключаються до об’єкту Deployment у формі об’єкту ConfigMap. Для зручності додавання конфігурацій, у шаблон «configmap.yaml» було додано функцію для зчитування змінних у форматі «ключ-значення».

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "poll-app.fullname" . }}
data:
{{- range $key, $value := .Values.appConfig }}
  {{ $key }}: {{ $value | quote }}
{{- end }}
```

Рисунок 4.16 — Шаблон об’єкту ConfigMap

Відповідні змінні були додані до файлу значень «values.yaml»

```yaml
appConfig:
  DATABASE_NAME: poll-app
  DATABASE_HOST: pg-poll-app-dev-do-user-12295190-0.h.db.ondigitalocean.com
  DATABASE_PORT: 25060
  ALLOWED_HOSTS: localhost,127.0.0.1,pollapp-dev.opsforapps.org
  ALLOWED_CIDR_NETS: 10.244.0.0/16
  DEBUG: False
```

Рисунок 4.17 — Об’єкт для визначення змінних оточення для конфігурації додатку

4.5 Автоматизація за допомогою сценаріїв Gitlab CI/CD

Для реалізаці технології автоматизацій розгортання контейнеризованих додатків, було створена модель взаємодії компонентів автоматизації.

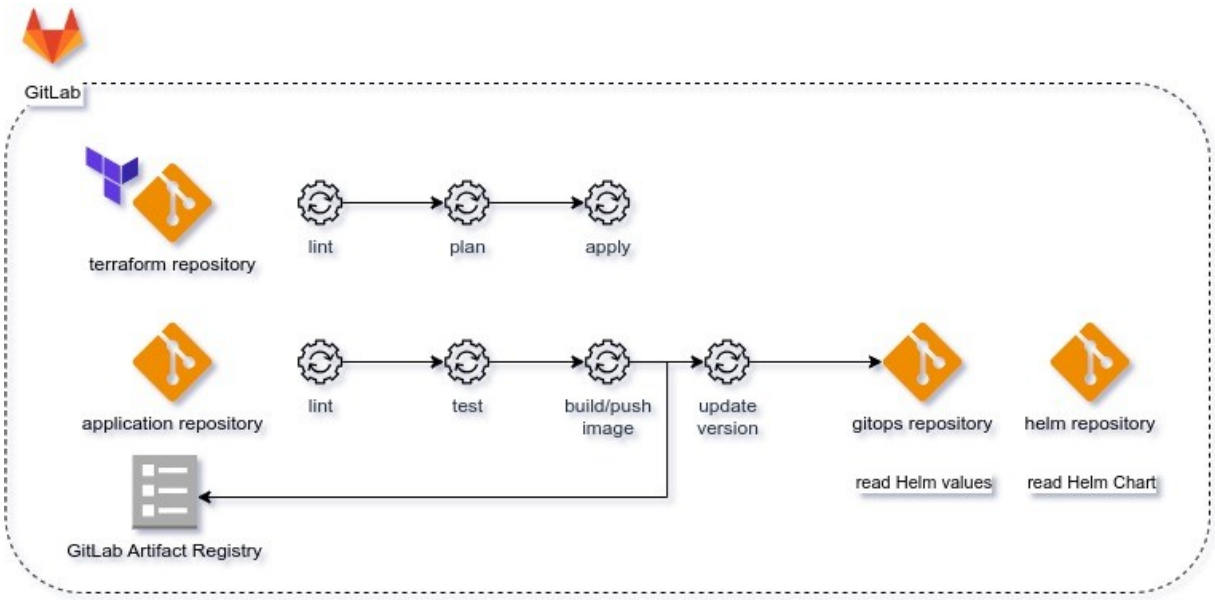


Рисунок 4.18 — Модель взаємодії компонентів автоматизації GitLab CI/CD

Відповідно моделі, було створено репозиторії в GitLab:

- Poll App — репозиторій для розробки коду додатку так автоматизації його збірки;
- Terraform Infra — репозиторій для розробки коду Terraform та автоматизації застосування змін до інфраструктури;
- Helm Charts — репозиторій для зберігання створеного покету Helm Chart для додатку;
- Gitops — репозиторій для майбутньої імплементації ArgoCD.

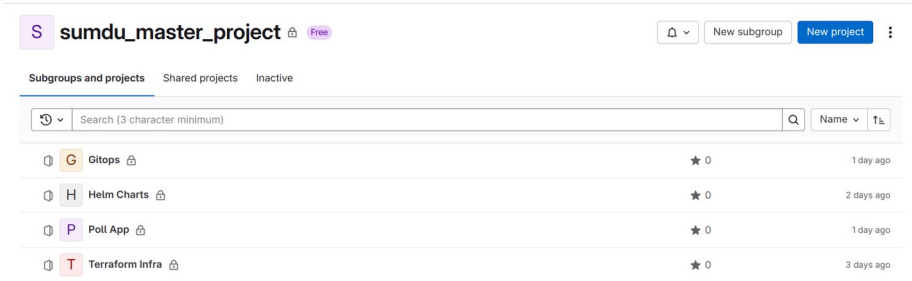


Рисунок 4.19 — Список створених репозиторіїв в GitLab

Для автоматизації збірки образів додатку було створено сценарій GitLab, для цього в репозиторій було додано файл «.gitlab-ci.yml». Було визначено етап сценарію «build» та додано задачу для збірки.

У якості оточення для виконання задач, використані агенти GitLab типу «docker», які представлені контейнерами, що виконують визначені скрипти автоматизації.

Для збірки образів Docker в контейнеризованому оточенні було використано інструмент Kaniko. В якості сховища для зберігання та розповсюдження артефактів збірки, було використано рішення від GitLab — Container Registry.

```
kaniko_build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:v1.23.2-debug
    entrypoint: [""]
  script:
    - mkdir -p /kaniko/.docker
    # Write credentials to access Gitlab Container Registry within the runner/ci
    - echo "{\"auths\":{\"$CI_REGISTRY\":{\"auth\":\"$(echo -n ${CI_REGISTRY_USER}:${CI_REGISTRY_PASSWORD} | base64 | tr -d '\n')\"}}}" > /kaniko/.docker/config.json
    - /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_SHORT_SHA}"
```

Рисунок 4.20 — Задача збірки для сценарію репозиторія додатку

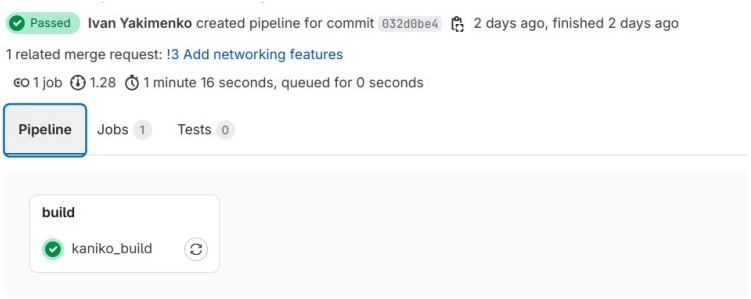


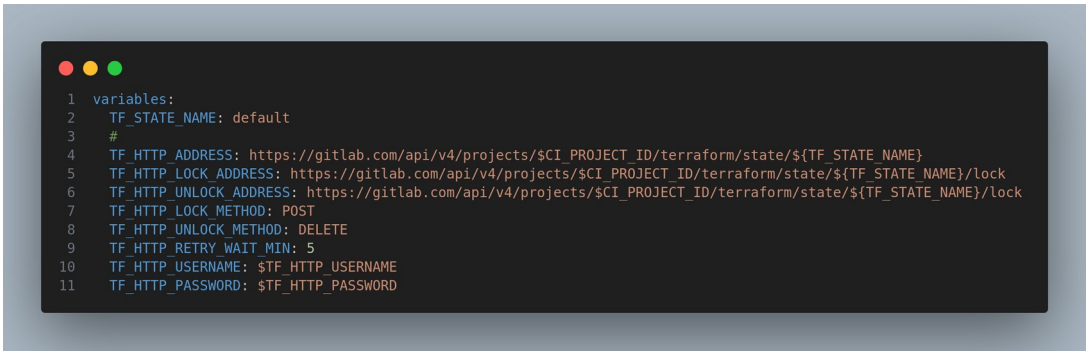
Рисунок 4.21 — Приклад успішного виконання задачі збірки в GitLab

Рисунок 4.22 — Docker образ додатку в сховищі Container Registry

Для автоматизації застосування змін до інфраструктури за допомогою Terraform, до репозиторію Teraform було додано сценарій автоматизації з відповідними задачами:

- перевірка форматування коду за допомогою «terraform fmt» та tflint;
- планування змін до інфраструктури та збереження плану до артефакту;
- застосування змін відповідно до створеного артефакту плану.

Для запуску Terraform команд в контейнеризованих оточеннях, «стан» Terraform, представлений файлом «terraform.tfstate» було перенесено до GitLab сервісу Terraform States. Для цього було додано конфігурацію для віддаленого сховища «стану». У сценарії автоматизації для Terraform ця конфігурація визначена набором спеціальних змінних.



```
variables:
  TF_STATE_NAME: default
  #
  TF_HTTP_ADDRESS: https://gitlab.com/api/v4/projects/$CI_PROJECT_ID/terraform/state/${TF_STATE_NAME}
  TF_HTTP_LOCK_ADDRESS: https://gitlab.com/api/v4/projects/$CI_PROJECT_ID/terraform/state/${TF_STATE_NAME}/lock
  TF_HTTP_UNLOCK_ADDRESS: https://gitlab.com/api/v4/projects/$CI_PROJECT_ID/terraform/state/${TF_STATE_NAME}/lock
  TF_HTTP_LOCK_METHOD: POST
  TF_HTTP_UNLOCK_METHOD: DELETE
  TF_HTTP_RETRY_WAIT_MIN: 5
  TF_HTTP_USERNAME: $TF_HTTP_USERNAME
  TF_HTTP_PASSWORD: $TF_HTTP_PASSWORD
```

Рисунок 4.23 — Набір спеціальних змінних для конфігураційдля віддаленого сховища «стану» Terraform

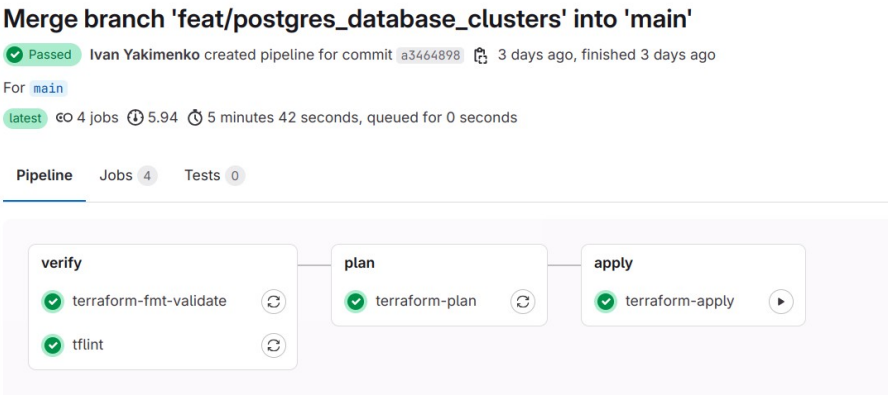


Рисунок 4.24 — Успішно виконаний сценарій автоматизації Terraform

В результаті реалізації сценаріїв автоматизації, було створено дві підсистеми:

- система автоматичної збірки образів Docker при внесенні змін до коду додатку;
- система автоматичного застосування змін до інфраструктури при оновленні коду Terraform.

4.5 Імплементація ArgoCD

Для розгортання системи ArgoCD використано Terraform та Helm. До змінної «helm_releases» в коді Terraform було додано відповідний блок для визначення параметрів розгортання нового Helm Chart.

Рисунок 4.25 — Новий блок значення змінної «helm_releases» для ArgoCD

Після розгортання ArgoCD за допомогою автоматизованого сценарію Terraform, було виконано перевірку статусів компонентів.

Рисунок 4.26 — Список компонентів ArgoCD та їх статуси

ArgoCD має власний додатковий набір типів об’єктів Kubernetes (custom resource definitions, CRDs). Їх наявність у кластері було перевірено за допомогою відповідної команди «kubectl», з використанням фільтру утиліти «grep».

```
vanadium@wks4  ~  kubectl get crds | grep argo
applications.argoproj.io                              2024-12-08T18:58:28Z
applicationsets.argoproj.io                           2024-12-08T18:58:28Z
appprojects.argoproj.io                               2024-12-08T18:58:28Z
```

Рисунок 4.27 — Список компонентів ArgoCD та їх статуси

Для реалізації автоматичної системи розгортання додатку, було створено модель взаємодії ArgoCD та наявних репозиторіїв GitLab.

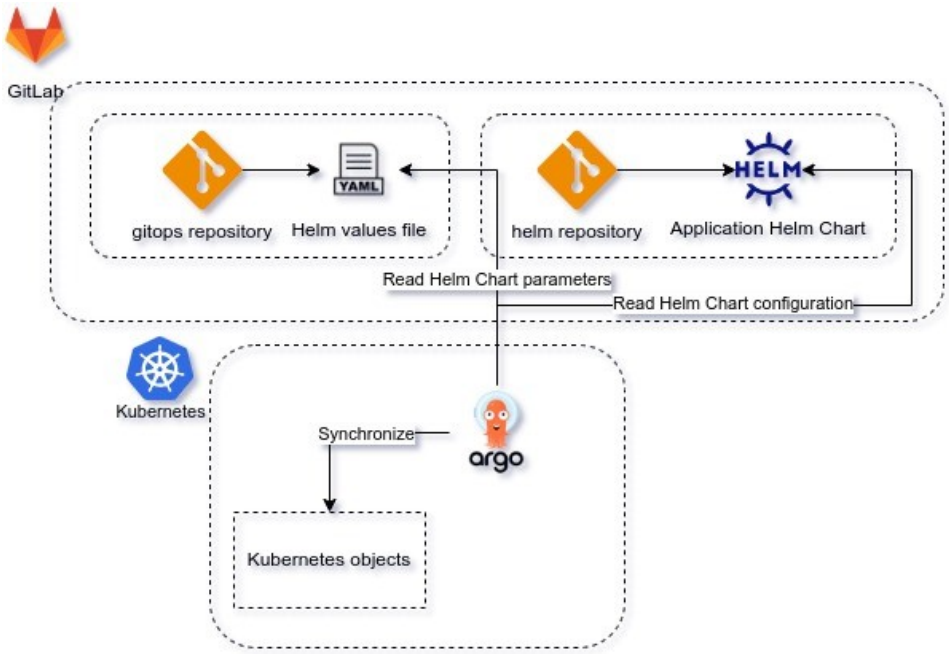


Рисунок 4.28 — Модель взаємодії ArgoCD та наявних репозиторіїв GitLab

Для надання доступу ArgoCD до репозиторіїв GitLab, було створено необхідні конфігурації у вигляді об’єктів Secret. Об’єкти містять у собі параметри підключення до репозиторію, такі як:

- посилання на репозиторій;
- ім’я користувача для підключення;
- секретний токен користувача.

```
vanadium@wks4 ~ k get -n argo-cd secrets | grep repo
repo-access-gitops              Opaque              5
repo-access-helmcharts          Opaque              5
```

Рисунок 4.29 — Створені об’єкти типу Secret для конфігурації підключення до репозиторіїв GitLab

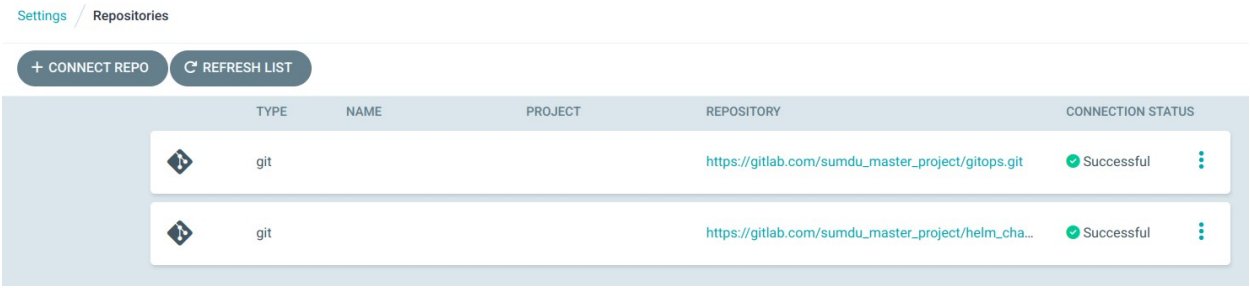


Рисунок 4.30 — Список підключених репозиторіїв в панелі управління ArgoCD

Для опису додатків, які будуть керуватися за допомогою декларативного підходу, було використано об’єкт Application.

Об’єкт Application має такі конфігураційні параметри:

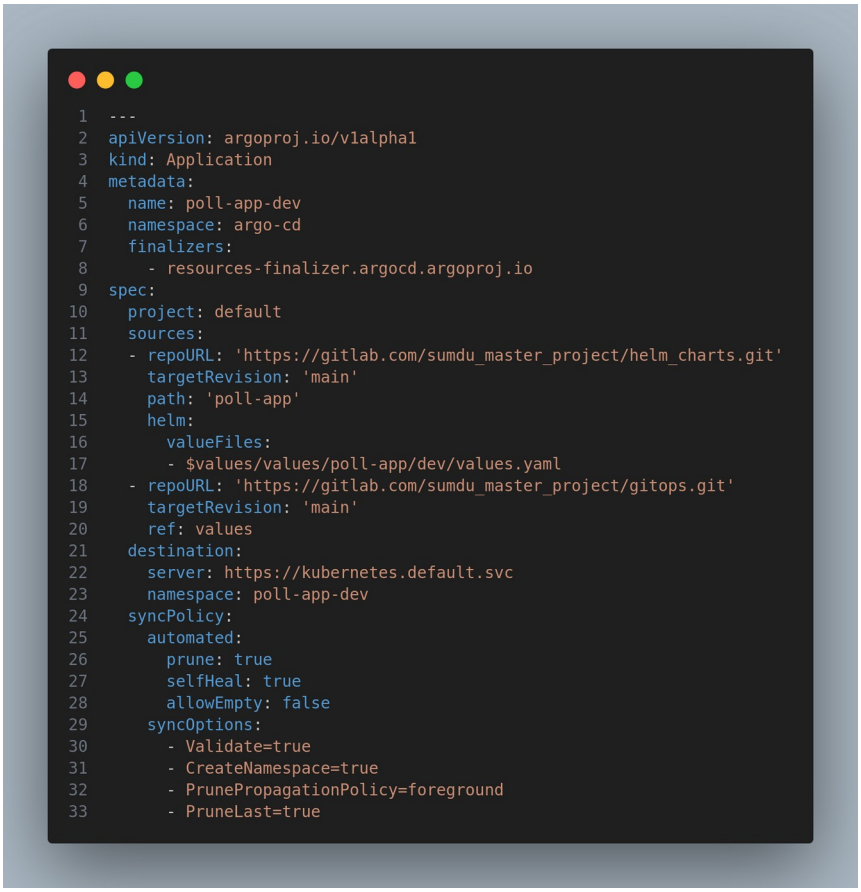
- джерело для зчитування конфігурації;
- цільовий кластер Kubernetes;
- політика синхронізації.

Джерелом у обраній моделі буде конфігурація, розміщена в двох окремих репозиторіях GitLab:

- Helm Charts (де розміщено пакунок Helm Chart для додатка);
- GitOps (де розміщено файл значень для пакунка Helm Chart).

Цільовим кластером є кластер, у якому розгорнута система ArgoCD.

Політика синхронізації визначає дозвіл на автоматичну синхронізацію стану об’єктів Kubernetes відповідно до конфігурацій джерела.



```yaml
---
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: poll-app-dev
  namespace: argo-cd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  sources:
  - repoURL: 'https://gitlab.com/sumdu_master_project/helm_charts.git'
    targetRevision: 'main'
    path: 'poll-app'
    helm:
      valueFiles:
      - $values/values/poll-app/dev/values.yaml
  - repoURL: 'https://gitlab.com/sumdu_master_project/gitops.git'
    targetRevision: 'main'
    ref: values
  destination:
    server: https://kubernetes.default.svc
    namespace: poll-app-dev
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
      allowEmpty: false
    syncOptions:
      - Validate=true
      - CreateNamespace=true
      - PrunePropagationPolicy=foreground
      - PruneLast=true
```

Рисунок 4.31 — Конфігурація об’єкту Application для автоматичного розгортання та синхронізації додатка

Проблемою використання окремих об’єктів Application для кожного додатка є потреба у їх ручному додаванні до кластеру. Патерн «App-of-Apps» - це підхід, що дозволяє створити один «батьківський» об’єкт Application, який має у якості джерела конфігурацій директорію репозиторію Git з «дочірніми» об’єктами Application, кожен з яких має унікальні параметри для усіх необхідних додатків. При використанні такого підходу, достатньо додати єдиний «батьківський» об’єкт Application до кластеру, зберігаючи «дочірні» об’єкти Application в репозиторії Git.

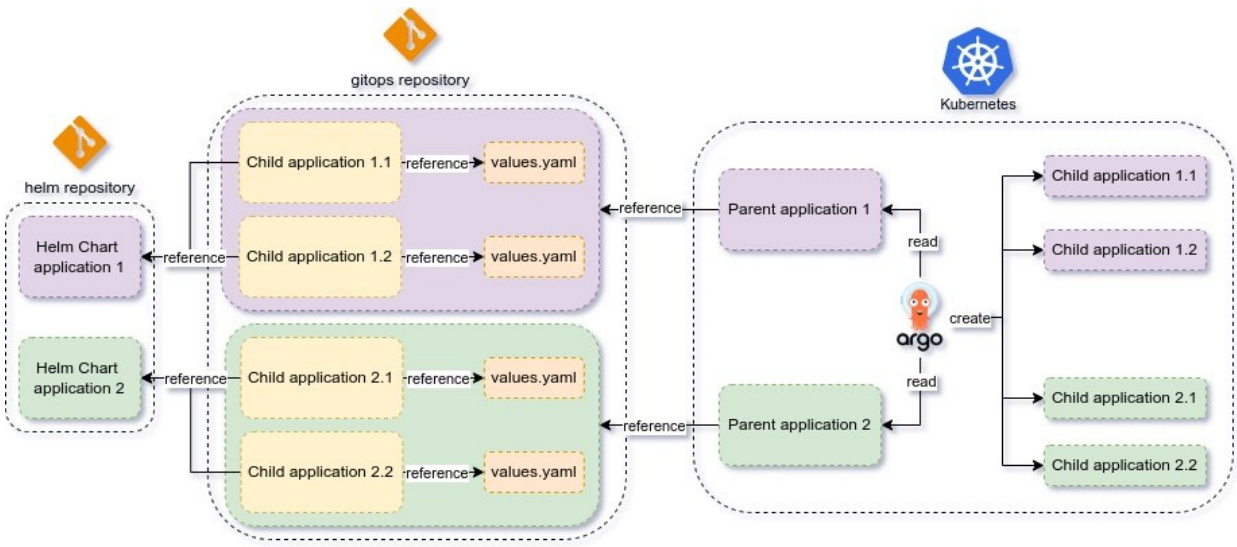


Рисунок 4.32 — Приклад моделі роботи AdrgoCD з використанням патерну «App-of-apps»

Для реалізації розгортання додатку у двох оточеннях («dev» та «prod»), в репозиторії GitOps було створено відповдіну структуру директорій.

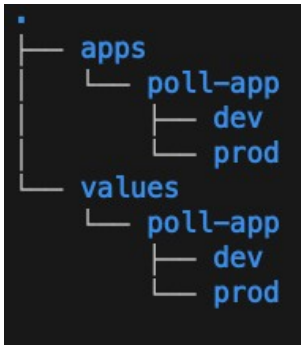


Рисунок 4.33 — Структура директорій репозиторію GitOps

До директорії «apps/poll-app/dev» та «apps/poll-app/prod» було додано конфігурації об’єктів Application для двох оточень, джерелом конфігурацій для яких є пакунок Helm Chart репозиторію Helm Charts та відповідні файли значень «values.yaml» в директоріяї «values/poll-app/dev» та «values/poll-app/prod». До кластеру Kubernetes додано «батьківський» об’єкт

Application.

```yaml
---
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: apps
  namespace: argo-cd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  source:
    repoURL: 'https://gitlab.com/sumdu_master_project/gitops.git'
    targetRevision: 'main'
    path: 'apps'
    directory:
      recurse: true
  destination:
    server: https://kubernetes.default.svc
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
      allowEmpty: false
    syncOptions:
      - Validate=true
      - CreateNamespace=false
      - PrunePropagationPolicy=foreground
      - PruneLast=true
```

Рисунок 4.34 — Конфігурація «батьківського» об’єкту Application

В результаті синхронізації стану, додаток було розгонуто у двох екземплярах: в оточенні «dev» та «prod».

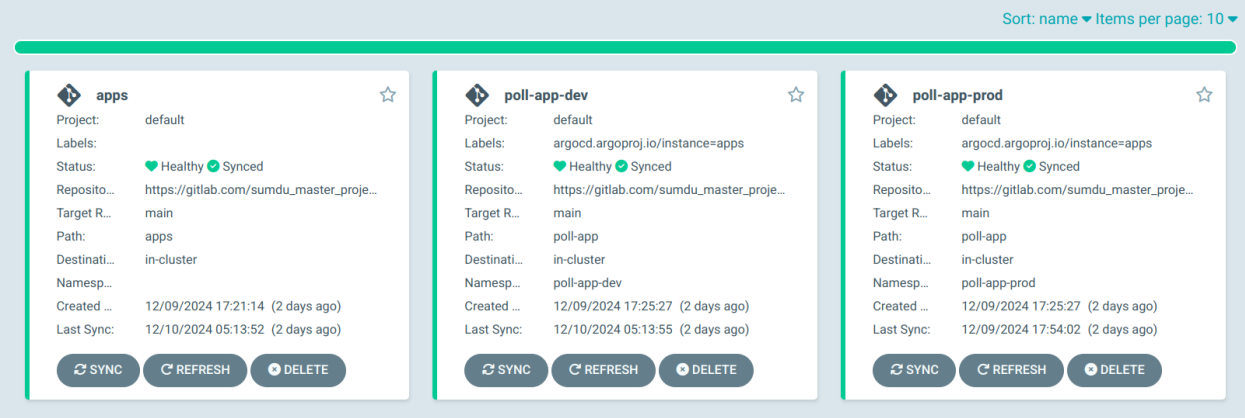


Рисунок 4.35 — Успішно синхронізовані «батьківський» та «дочірні» об’єкти Application в панелі управління ArgoCD

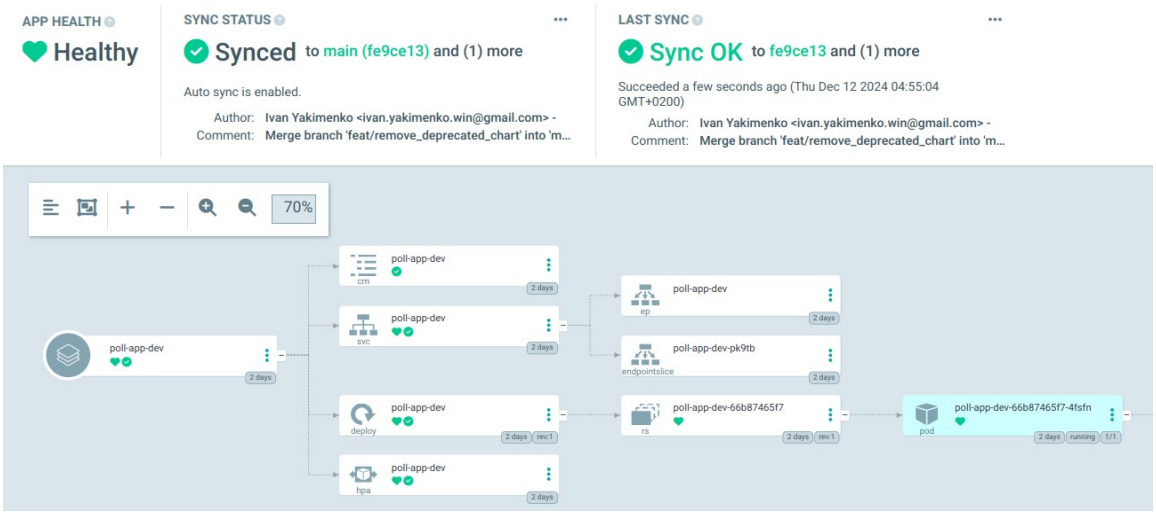


Рисунок 4.36 — Набір створених об’єктів Kubernetes в панелі управління ArgoCD

Для реалізація автоматизованого внесення змін до репозиторію GitOps при збірці нової версії образу Docker, сценарій репозиторію Poll App було розширено задачею по створенню Merge Request.

```
auto_mr:
  stage: trigger_deploy
  image:
    name: alpine/git:2.47.1
    entrypoint: [""]
  variables:
    IMAGE_TAG: ${DEPLOY_ENV}-${CI_COMMIT_SHORT_SHA}
    UPDATE_BRANCH_NAME: update/${DEPLOY_ENV}/${CI_PROJECT_NAME}/tag/${IMAGE_TAG}
  script:
    - apk add yq curl
    - git config --global user.email "ci-bot@example.com"
    - git config --global user.name "CI Bot"
    - git clone https://${GITLAB_USER}:${GITLAB_TOKEN}@gitlab.com/sumdu_master_project/gitops.git gitops
    - cd gitops
    - git checkout -b ${UPDATE_BRANCH_NAME}
    - yq eval '.image.tag = env(IMAGE_TAG)' -i values/poll-app/${DEPLOY_ENV}/values.yaml
    - git commit values/poll-app/${DEPLOY_ENV}/values.yaml -m "Update image tag for ${DEPLOY_ENV} environment of poll-app application"
    - git push origin ${UPDATE_BRANCH_NAME}
    - |
      curl -X POST -H "PRIVATE-TOKEN: ${GITLAB_TOKEN}" \
      -d "source_branch=${UPDATE_BRANCH_NAME}" \
      -d "target_branch=main" \
      -d "title=Update image tag for ${DEPLOY_ENV} environment of poll-app application" \
      "https://gitlab.com/api/v4/projects/65212880/merge_requests"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      variables:
        DEPLOY_ENV: "prod"
    - if: '$CI_COMMIT_BRANCH == "dev"'
      variables:
        DEPLOY_ENV: "dev"
    - if: '$CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "dev"'
```

Рисунок 4.37 — Скрипт задачі для автоматичного створення Merge Request в репозиторію GitOps

Реалізацією задачі є скрипт, який виконує такі дії:

- клонування репозиторію GitOps;
- вносення зміни до версії образу Docker одного з оточень («dev» або «prod») за допомогою утиліти для роботи за файлами YAML «yq»;
- створення Merge Request з відповідною зміною.

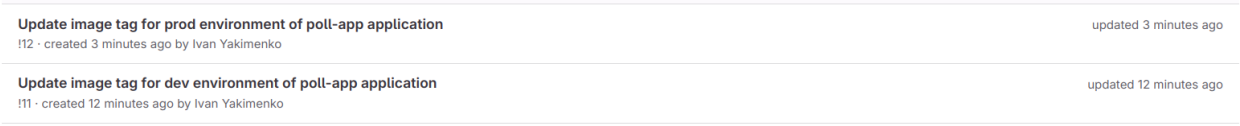


Рисунок 4.38 — Приклад створених Merge Request в репозиторії GitOps з гілок «dev» та «main»

4.6 Реалізація доступу до додатку через мережу Інтернет

Для надання доступу до додатку через мережу Інтернет, у DNS провайдері CloudFlare було створено доменну зону «opsforapps.org» та DNS записи типу A для двох оточень:

- pollapp-dev для оточення «dev»;
- pollap для оточення «prod».

IP-адресою є публічна адреса зовнішнього балансувальника навантаження Digital Ocean:

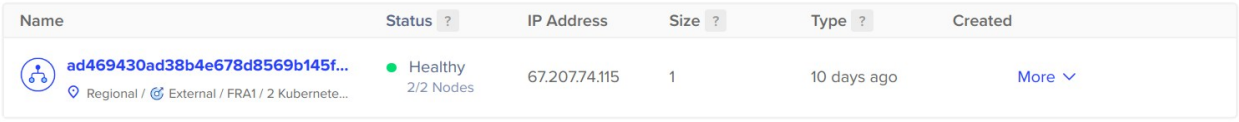


Рисунок 4.39 — Балансувальник навантаження в панелі управління Digital Ocean

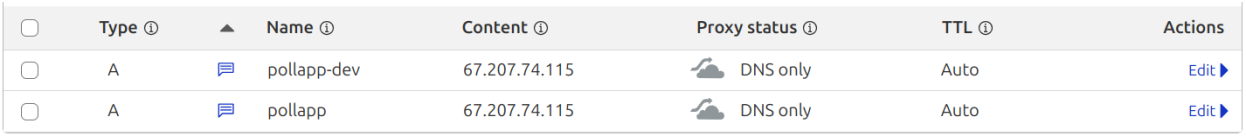


Рисунок 4.40 — A-записи DNS в панелі управління Cloud Flare

Для перевірки коректності конфігурацій DNS, було використано утиліту «dig».

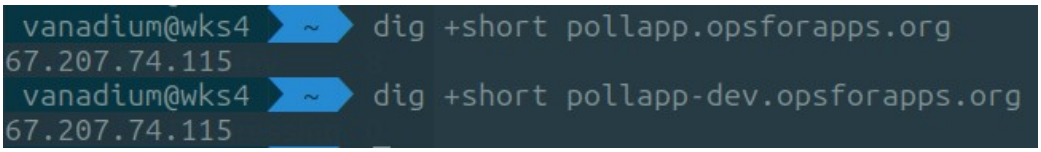


Рисунок 4.41 — Результат запитів утиліти «dig»

Для підключення DNS до додатка в Kubernetes, було сконфігуровано Ingress об’єкт за допомогою файлу значень для пакунка Helm Chart та додано конфігурацію для автоматичного створення замовлень сертифікатів TLS сервісу Cert-manager.

```yaml
ingress:
  enabled: true
  className: "nginx"
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod-cf
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
  hosts:
    - host: pollapp.opsforapps.org
      paths:
        - path: /
          pathType: Prefix
  tls:
    - secretName: pollapp.opsforapps.org-tls
      hosts:
        - pollapp.opsforapps.org
```

Рисунок 4.41 — Конфігурація файлу значень для пакунка Helm Chart для об’єкту Ingress

Після внесення змін до репозиторію GitOps, ArgoCD автоматично синхнонізував стан об’єктів Kubernetes, додавши новий ресурс Ingress.

Рисунок 4.42 — Створений об’єкт Ingrees в панелі управління ArgoCD

4.7 Тестування та оцінка результатів

Головним сценарієм для перевірки коректності роботи інформаційної системи є зміна кодової бази додатку.

З гілки «dev» в Git була створена нова гілка.

```
vanadium@wks4 ~/.../SumDU/poll_app  dev git checkout -b refactor/update_html_template
Switched to a new branch 'refactor/update_html_template'
```

Рисунок 4.43 — Команда створення нової гілки Git

Зміни були внесені до шаблона HTML головної сторінки додатку та збережені. Нова гілка зі змінами була відправлена до віддаленого репозиторію GitLab

```
vanadium@wks4 ~/.../SumDU/poll_app  refactor/update_html_template ● gcam "refactor: update HTML template for the main page"
[refactor/update_html_template 594a243] refactor: update HTML template for the main page
1 file changed, 1 insertion(+), 2 deletions(-)
```

Рисунок 4.44 — Команда збереження змін до нової гілки Git

```
diff --git a/pollProject/templates/pages/index.html b/pollProject/templates/pages/index.html
index 1ef01c0..d652028 100644
--- a/pollProject/templates/pages/index.html
+++ b/pollProject/templates/pages/index.html
@@ -4,8 +4,7 @@
 <div class="card text-center">
     <div class="card-body" >
        <h1> Welcome to Poll Mall!</h1>
-    <p>This is my first Django
-         Project after long time!</p>
+    <p>Vote!</p>
     <a class="btn btn-dark" href = "{% url 'polls:index' %}">
         View Available Polls! </a>
     </div>
```

Рисунок 4.45 — Зміни до кодової бази

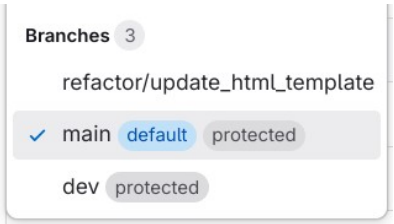


Рисунок 4.46 — Список гілок у віддаленому репозиторію GitLab

За допомогою Merge Request, зміни додані в гілку «dev». Подія внесення змін до гілки «dev» автоматично створює та запускає сценарій GitLab.

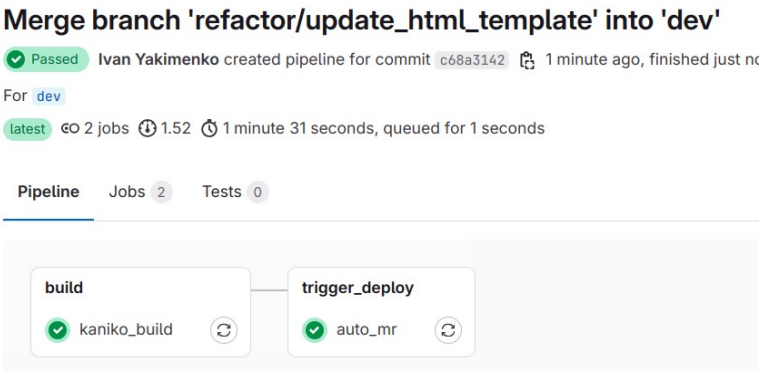


Рисунок 4.46 — Успішно виконаний сценарій GitLab після внесення змін

Останнім кроком сценарію є створення Merge Request в репозиторію Gitops зі змінами у версії додатку, новий образ якого було зібрано.

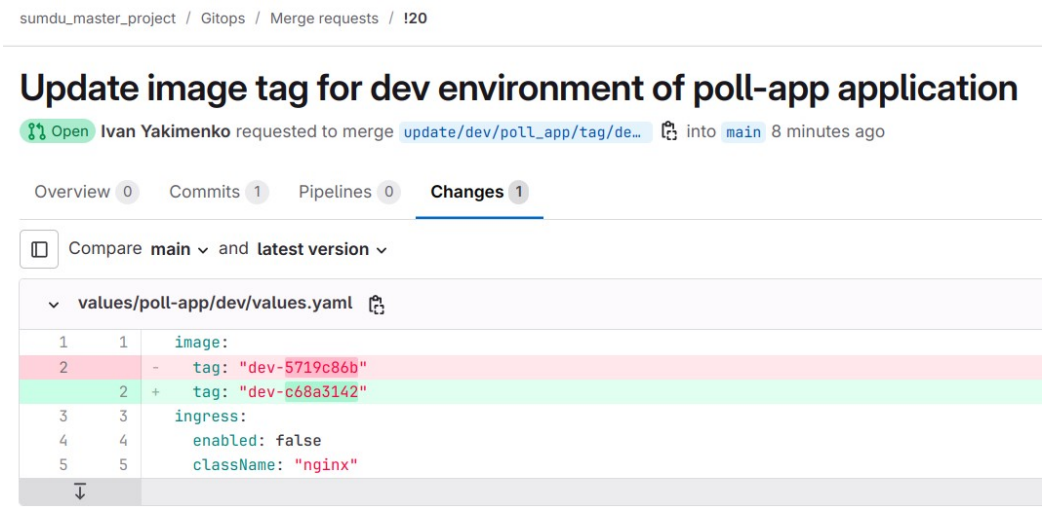


Рисунок 4.47 — Зміни в автоматично створеному Merge Request

Після внесення змін до версії образу для додатка до репозиторію Gitops, який відслідковує ArgoCD, відбулась синхронізація стану об’єктів

Kubernetes.

Рисунок 4.48 — Процес синхронізації в панелі управління ArgoCD

Відповідно до політики оновлення об’єкту Deployment «RollingUpdate» в Kubernetes, «под» з новою версією додатку було створено та запущено, «под» з попередньою версією додатку було видалено.

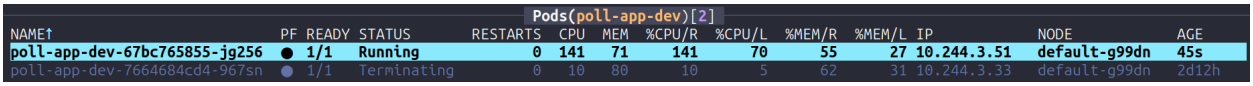


Рисунок 4.49 — Процес оновлення версії додаткув в панелі інструменту k9s для роботи з Kubernetes

Рисунок 4.50 — Результат виконання команди для перегляду поточної версії образу

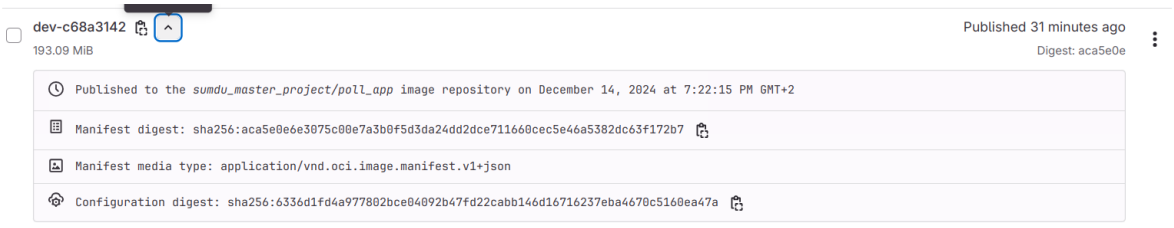


Рисунок 4.51 — Образ нової версії додатку у сховищі Container Registry

Додаток зі змінами, внесеними до шаблону HTML сторінки, було розгорнуто успішно.

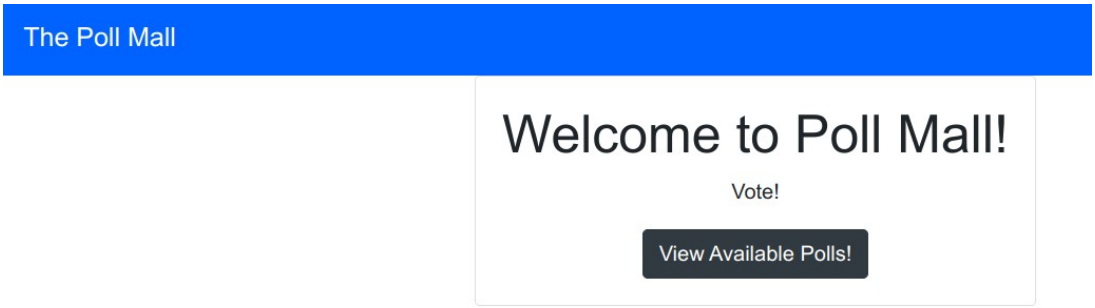


Рисунок 4.52 — Змінений блок головної сторінки додатка

Так як оточення «dev» не відкрите для доступу з мережі Інтернет, переверка сертифікату було виконано для оточення «prod».

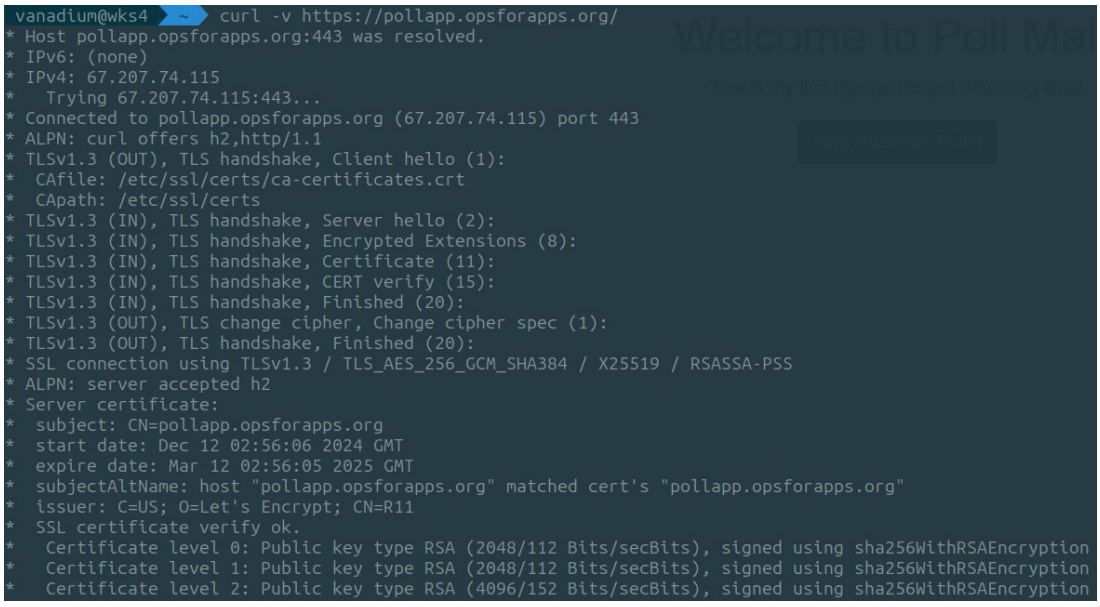


Рисунок 4.52 — Результат виконання команди «curl» до доменного імені додатку з використанням протоколу HTTPS

Отже, нформаційна технологія автоматичного розгортання

контейнеризованих додатків у розподілених середовищах із використанням принципів декларативного управління виконує поставлену задачу.

ВИСНОВКИ

Виконання даної кваліфікаційної роботи дало можливість розглянути та дослідити сучасні технології та програмні продукти для автоматизації процесів розробки, інтеграції та розгортання програмного забезпечення. Реалізована інформаційна система є ефективним інструментом для прискорення процесу розробки програмного забезпечення, оптимізації процесів запровадження змін до кодової бази, уніфікації підходу до розгортання контейнеризованих додатків у розподіленому середовищі Kubernetes будь-якої реалізації. Програмні та технологічні рішення, обрані під час виконання кваліфікаційної роботи, можуть стати основою для імплементації декларативних підходів управління як комп'ютерною інфраструктурою, так і життєвим циклом програмного забезпечення.

У ході виконання кваліфікаційної роботи було виконано такі завдання:

1. досліджено процес автоматичного розгортання контейнеризованих додатків у розподілених середовищах
2. проведено інформаційний огляд доступних програмних та технологічних рішень
3. обрано конкретні технологічні та програмні рішення для проєктування та реалізації інформаційної системи
4. підготовано та скофнігуровано необхідну комп'ютерну інфраструктуру
5. реалізовано інформаційну систему автоматичного розгортання контейнеризованих додатків у розподілених середовищах
6. проведено огляд результатів реалізації.

Надалі планується імплементація рішення щодо моніторингу та збору логів для компонентів інформаційної системи. Розглядається можливість створення «внутрішньої платформи для розробки» на основні результату реалізації інформаційної системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ArgoCD concepts [Електронний ресурс] // argo-cd.readthedocs.io. Режим доступу до ресурсу: https://argo-cd.readthedocs.io/en/stable/core_concepts/.

2. ArgoCD declarative setup [Електронний ресурс] // argo-cd.readthedocs.io. Режим доступу до ресурсу: https://argo-cd.readthedocs.io/en/stable/operator-manual/declarative-setup/.

3. Cert-manager Certificate Issuer [Електронний ресурс] // cert-manager.io. Режим доступу до ресурсу: https://cert-manager.io/docs/configuration/issuers/.

4. Cert-manager installation using Helm [Електронний ресурс] // cert-manager.io. Режим доступу до ресурсу: https://cert-manager.io/docs/installation/helm/.

5. Cloud Flare Domain records configuration [Електронний ресурс] // developers.cloudflare.com. Режим доступу до ресурсу: https://developers.cloudflare.com/learning-paths/get-started/domain-resolution/review-dns-records/.

6. Django Framework for Python [Електронний ресурс] // docs.djangoproject.com. Режим доступу до ресурсу: https://docs.djangoproject.com/en/5.1/.

7. Digital Ocean Kubernetes [Електронний ресурс] // docs.digitalocean.com. Режим доступу до ресурсу: https://docs.digitalocean.com/products/kubernetes/.

8. GitLab CI/CD YAML syntax [Електронний ресурс] // docs.gitlab.com. Режим доступу до ресурсу: https://docs.gitlab.com/ee/ci/yaml/.

9. GitLab Predefined variables [Електронний ресурс] // docs.gitlab.com. Режим доступу до ресурсу: https://docs.gitlab.com/ee/ci/variables/predefined_variables.html.

10. Helm documentation [Електронний ресурс] // helm.sh. Режим доступу до ресурсу: https://helm.sh/docs/.

11. HashiCorp Terraform language [Електронний ресурс] // developer.hashicorp.com. Режим доступу до ресурсу: https://developer.hashicorp.com/terraform/language.

12. Kubernetes: Up and Running / Joe Beda, Kelsey Hightower, Brendan Burns // O'Reilly Media, Inc. - 2017 September.

13. Kubernetes documentation [Електронний ресурс] // kubernetes.io. Режим доступу до ресурсу: https://kubernetes.io/docs/home/.

14. Kubernetes Ingress [Електронний ресурс] // kubernetes.io. Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/services-networking/ingress/.

15. Kubernetes metrics [Електронний ресурс] // kubernetes.io. Режим доступу до ресурсу: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/.

16. NGINX Ingress controller for Kubernetes [Електронний ресурс] // docs.nginx.com. Режим доступу до ресурсу: https://docs.nginx.com/nginx-ingress-controller/configuration/ingress-resources/advanced-configuration-with-annotations/.

17. NGINX Load Balancer [Електронний ресурс] // docs.nginx.com. Режим доступу до ресурсу: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

18. Terraform: Up and Running / Yevgeniy Brikman // O'Reilly Media, Inc. - 2022 September.

19. Terraform code structure best practices [Електронний ресурс] // www.terraform-best-practices.com. Режим доступу до ресурсу: https://www.terraform-best-practices.com/code-structure.

20. Terraform Digital Ocean Provider [Електронний ресурс] // registry.terraform.io. Режим доступу до ресурсу: https://registry.terraform.io/providers/digitalocean/digitalocean/latest/doc.

21. Terraform state in GitLab [Електронний ресурс] // spacelift.io. Режим доступу до ресурсу: https://spacelift.io/blog/gitlab-terraform-state.

22. Using Cert-manager with Ingress [Електронний ресурс] // cert-manager.io. Режим доступу до ресурсу: https://cert-manager.io/docs/usage/ingress/.

ДОДАТОК А

```
stages:
- test
- build
- trigger_deploy

default:
image: python:3.12-slim
tags:
- gitlab-org

lint:
stage: test
script:
- pycodestyle --statistics --count

test:
stage: test
script:
- pip install -q -r requirements.txt
- pytest --junitxml=report.xml
artifacts:
expire_in: 1 day
when: always
reports:
junit: report.xml

kaniko_build:
stage: build
image:
name: gcr.io/kaniko-project/executor:v1.23.2-debug
entrypoint: [""]
variables:
IMAGE_TAG: ${DEPLOY_ENV}-${CI_COMMIT_SHORT_SHA}
script:
- mkdir -p /kaniko/.docker
# Write credentials to access Gitlab Container Registry within the runner/ci
- echo "{\"auths\":{\"$CI_REGISTRY\":{\"auth\":\"$(echo -n $
{CI_REGISTRY_USER}:${CI_REGISTRY_PASSWORD} | base64 | tr -d '\n')\"}}}" >
/kaniko/.docker/config.json
- /kaniko/executor
--context "${CI_PROJECT_DIR}"
--dockerfile "${CI_PROJECT_DIR}/Dockerfile"
--destination "${CI_REGISTRY_IMAGE}:${IMAGE_TAG}"
rules:
- if: '$CI_COMMIT_BRANCH == "main"'
variables:
DEPLOY_ENV: "prod"
- if: '$CI_COMMIT_BRANCH == "dev"'
variables:
DEPLOY_ENV: "dev"
- if: '$CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "dev"'

auto_mr:
stage: trigger_deploy
image:
name: alpine/git:2.47.1
entrypoint: [""]
variables:
IMAGE_TAG: ${DEPLOY_ENV}-${CI_COMMIT_SHORT_SHA}
UPDATE_BRANCH_NAME: update/${DEPLOY_ENV}/${CI_PROJECT_NAME}/tag/${IMAGE_TAG}
script:
```

```
- apk add yq curl
- git config --global user.email "ci-bot@example.com"
- git config --global user.name "CI Bot"
- git clone
https://${GITLAB_USER}:${GITLAB_TOKEN}@gitlab.com/sumdu_master_project/
gitops.git gitops
- cd gitops
- git checkout -b ${UPDATE_BRANCH_NAME}
- yq eval '.image.tag = env(IMAGE_TAG)' -i
values/poll-app/${DEPLOY_ENV}/values.yaml
- git commit values/poll-app/${DEPLOY_ENV}/values.yaml -m "Update image tag
for ${DEPLOY_ENV} environment of poll-app application"
- git push origin ${UPDATE_BRANCH_NAME}
- |
curl -X POST -H "PRIVATE-TOKEN: ${GITLAB_TOKEN}" \
-d "source_branch=${UPDATE_BRANCH_NAME}" \
-d "target_branch=main" \
-d "title=Update image tag for ${DEPLOY_ENV} environment of poll-app
application" \
"https://gitlab.com/api/v4/projects/65212880/merge_requests"
rules:
- if: '$CI_COMMIT_BRANCH == "main"'
variables:
DEPLOY_ENV: "prod"
- if: '$CI_COMMIT_BRANCH == "dev"'
variables:
DEPLOY_ENV: "dev"
- if: '$CI_COMMIT_BRANCH == "main" || $CI_COMMIT_BRANCH == "dev"'
```

ДОДАТОК Б

```
terraform {
required_version = ">= 1.8"
required_providers {
digitalocean = {
source = "digitalocean/digitalocean"
version = "~> 2.0"
}
helm = {
source = "hashicorp/helm"
version = "2.16.1"
}
}
}

#####################
# General variables #
#####################

variable "digitalocean_token" {
description = "Digital Ocean API token"
type = string
default = ""
}

variable "region" {
description = "A region name for the resources"
type = string
default = "fra1"
}

#################
# VPC variables #
#################

variable "vpc_enabled" {
type = bool
default = true
description = "Whether to create the resources. Set to `false` to prevent the
module from creating any resources."
}

variable "vpc_name" {
type = string
default = ""
description = "Name (e.g. `app` or `cluster`)."
}

variable "vpc_description" {
type = string
default = "VPC"
description = "A free-form text field up to a limit of 255 characters to
describe the VPC."
}

variable "vpc_ip_range" {
type = string
default = ""
description = "The range of IP addresses for the VPC in CIDR notation.
Network ranges cannot overlap with other networks in the same account and
must be in range of private addresses as defined in RFC1918. It may not be
larger than /16 or smaller than /24."
```

```
}

########################
# Kubernetes variables #
########################

variable "kubernetes_enabled" {
type = bool
default = true
description = "Whether to create the resources. Set to `false` to prevent the
module from creating any resources."
}

variable "kubernetes_name" {
type = string
default = ""
description = "Name (e.g. `app` or `cluster`)."
}

variable "kubernetes_cluster_version" {
type = string
default = "1.27.2"
description = "K8s Cluster Version."
}

variable "kubernetes_auto_upgrade" {
type = bool
default = false
description = "Enable auto upgrade during maintenance window."
}

variable "kubernetes_surge_upgrade" {
type = bool
default = false
description = "Enable surge upgrade during maintenance window."
}

variable "kubernetes_ha" {
type = bool
default = true
description = "Enable high availability control plane."
}

variable "kubernetes_registry_integration" {
type = bool
default = false
description = "Enables or disables the DigitalOcean container registry
integration for the cluster. This requires that a container registry has
first been created for the account."
}

variable "kubernetes_default_node_pool" {
type = any
default = {}
description = "Cluster default node pool."
}

variable "kubernetes_node_pools" {
type = map(any)
default = {}
description = "Cluster additional node pools."
}
variable "kubernetes_maintenance_policy" {
```

```
type = object({
day = string
start_time = string
})
default = {
day = "any"
start_time = "5:00"
}
description = "Define the window updates are to be applied when auto upgrade
is set to true."
}

###############
# Helm Charts #
###############

variable "helm_releases" {
description = "A list of helm releases and their parameters to be deployed to
the cluster"
type = list(object({
chart = string
name = string
create_namspace = optional(bool, false)
namespace = string
repository = string
version = string
values = optional(list(string), [])
}))
default = null
}

#############
# Databases #
#############

variable "db_clusters" {
description = "List of database clusters configurations"
type = list(object({
name = string
engine = string
version = string
size = string
node_count = number
databases = list(string)
users = list(string)
firewall_allow_rules = optional(list(object({
type = string
value = string
})), [])
}))
default = null
}

provider "digitalocean" {
token = var.digitalocean_token
}

resource "digitalocean_vpc" "this" {
count = var.vpc_enabled ? 1 : 0

name = var.vpc_name
region = var.region
ip_range = var.vpc_ip_range
```

```
description = var.vpc_description
}

resource "digitalocean_kubernetes_cluster" "this" {
count = var.kubernetes_enabled ? 1 : 0

name = var.kubernetes_name
region = var.region
version = var.kubernetes_cluster_version
vpc_uuid = digitalocean_vpc.this[0].id
auto_upgrade = var.kubernetes_auto_upgrade
surge_upgrade = var.kubernetes_surge_upgrade
ha = var.kubernetes_ha
registry_integration = var.kubernetes_registry_integration
dynamic "node_pool" {
for_each = var.kubernetes_default_node_pool
content {
name = lookup(node_pool.value, "name", "critical")
size = lookup(node_pool.value, "size", "s-1vcpu-2gb")
node_count = lookup(node_pool.value, "node_count", null)
auto_scale = lookup(node_pool.value, "auto_scale", true)
min_nodes = lookup(node_pool.value, "min_nodes", 1)
max_nodes = lookup(node_pool.value, "max_nodes", 2)
tags = lookup(node_pool.value, "tags", null)
labels = lookup(node_pool.value, "labels", null)
dynamic "taint" {
for_each = lookup(node_pool.value, "taint", [])
content {
key = lookup(taint.value, "key", null)
value = lookup(taint.value, "value", null)
effect = lookup(taint.value, "effect", null)
}
}
}
}
dynamic "maintenance_policy" {
for_each = var.kubernetes_auto_upgrade ? [1] : []
content {
day = var.kubernetes_maintenance_policy.day
start_time = var.kubernetes_maintenance_policy.start_time
}
}
}

resource "digitalocean_kubernetes_node_pool" "this" {
for_each = var.kubernetes_enabled ? var.kubernetes_node_pools : {}
cluster_id = join("", digitalocean_kubernetes_cluster.this[*].id)
name = lookup(each.value, "name", "application")
size = lookup(each.value, "size", "s-1vcpu-2gb")
node_count = lookup(each.value, "node_count", null)
auto_scale = lookup(each.value, "auto_scale", true)
min_nodes = lookup(each.value, "min_nodes", 1)
max_nodes = lookup(each.value, "max_nodes", 2)
tags = lookup(each.value, "tags", null)
labels = lookup(each.value, "labels", null)

dynamic "taint" {
for_each = lookup(each.value, "taint", [])
content {
key = taint.value["key"]
value = taint.value["value"]
effect = taint.value["effect"]
}
```

```
}
}
provider "helm" {
kubernetes {
host = digitalocean_kubernetes_cluster.this[0].endpoint
token = digitalocean_kubernetes_cluster.this[0].kube_config[0].token
cluster_ca_certificate = base64decode(
digitalocean_kubernetes_cluster.this[0].kube_config[0].cluster_ca_certificate
)
}
}

resource "helm_release" "this" {
for_each = var.helm_releases == null ? {} : { for helm_release in
var.helm_releases : helm_release.name => helm_release }
name = each.value.name
repository = each.value.repository
chart = each.value.chart
version = each.value.version
create_namespace = each.value.create_namspace
namespace = each.value.namespace
values = [for path in each.value.values : file(path)]
}

locals {
db_cluster_db_mappings = flatten([
for db_cluster in var.db_clusters : [
for db_name in db_cluster.databases : {
cluster_name = db_cluster.name
cluster_id = digitalocean_database_cluster.this[db_cluster.name].id
database_name = db_name
}
]
])
db_cluster_user_mappings = flatten([
for db_cluster in var.db_clusters : [
for user_name in db_cluster.users : {
cluster_name = db_cluster.name
cluster_id = digitalocean_database_cluster.this[db_cluster.name].id
user_name = user_name
}
]
])
}

resource "digitalocean_database_cluster" "this" {
for_each = { for db_cluster in var.db_clusters : db_cluster.name =>
db_cluster }

name = each.value.name
engine = each.value.engine
version = each.value.version
size = each.value.size
region = var.region
node_count = each.value.node_count
}

resource "digitalocean_database_db" "this" {
for_each = { for db_map in local.db_cluster_db_mappings : "$
{db_map.cluster_name}-${db_map.database_name}" => db_map }

cluster_id = each.value.cluster_id
name = each.value.database_name
```

```
}

resource "digitalocean_database_user" "this" {
for_each = { for user_map in local.db_cluster_user_mappings : "$
{user_map.cluster_name}-${user_map.user_name}" => user_map }

cluster_id = each.value.cluster_id
name = each.value.user_name
}

resource "digitalocean_database_firewall" "this" {
for_each = { for db_cluster in var.db_clusters : db_cluster.name =>
db_cluster }

cluster_id = digitalocean_database_cluster.this[each.key].id

rule {
type = "k8s"
value = digitalocean_kubernetes_cluster.this[0].id
}

dynamic "rule" {
for_each = each.value.firewall_allow_rules
content {
type = rule.value.type
value = rule.value.value
}
}
}


# .gitlab-ci.yml

stages:
- verify
- plan
- apply

# Common variables
variables:
TF_STATE_NAME: default
#
TF_HTTP_ADDRESS:
https://gitlab.com/api/v4/projects/$CI_PROJECT_ID/terraform/state/$
{TF_STATE_NAME}
TF_HTTP_LOCK_ADDRESS:
https://gitlab.com/api/v4/projects/$CI_PROJECT_ID/terraform/state/$
{TF_STATE_NAME}/lock
TF_HTTP_UNLOCK_ADDRESS:
https://gitlab.com/api/v4/projects/$CI_PROJECT_ID/terraform/state/$
{TF_STATE_NAME}/lock
TF_HTTP_LOCK_METHOD: POST
TF_HTTP_UNLOCK_METHOD: DELETE
TF_HTTP_RETRY_WAIT_MIN: 5
TF_HTTP_USERNAME: $TF_HTTP_USERNAME
TF_HTTP_PASSWORD: $TF_HTTP_PASSWORD
#
PLAN: plan.cache # A filename for the Terraform plan file
PLAN_JSON: plan.json # A filename for the GitLab plan report attached to the
merge request
PLAN_TXT: plan.txt # A filename for the Terraform plan in readable form
#
TF_VAR_digitalocean_token: ${DIGITAL_OCEAN_TOKEN}
```

```
# Default job configuration
default:
image:
name: hashicorp/terraform:1.8.4
entrypoint: [""]
tags:
- $RUNNER_TAG
cache:
key: terraform-providers
paths:
- .terraform

# Check code validity and formatting
terraform-fmt-validate:
stage: verify
script:
- terraform init
- terraform fmt -recursive -check -diff
- terraform validate

# Run tflint for basic linting
tflint:
stage: verify
before_script:
- apk --no-cache add tflint
script:
- tflint --init
- tflint --recursive --config .tflint.hcl
cache: []

# Perform terraform plan, prepare artifacts and reports
terraform-plan:
stage: plan
before_script:
- apk --no-cache add jq
- alias convert_report="jq -r '([.resource_changes[]?.change.actions?]|
flatten)|{\"create\":(map(select(.==\"create\"))|length),\"update\":
(map(select(.==\"update\"))|length),\"delete\":(map(select(.==\"delete\"))|
length)}'"
script:
- terraform init
- terraform plan -parallelism=120 -out=${PLAN} -input=false
- terraform show --json ${PLAN} | convert_report > ${PLAN_JSON}
- terraform show -no-color ${PLAN} > ${PLAN_TXT}
artifacts:
reports:
terraform: ${PLAN_JSON}
paths:
- ${PLAN_TXT}
- ${PLAN}
expire_in: 7 days

# Perform terraform apply using the plan artifact from plan job
terraform-apply:
stage: apply
script:
- terraform init
- terraform apply -auto-approve ${PLAN}
only:
- main
when: manual
dependencies:
- terraform-plan
```

ДОДАТОК В

```
# _helpers.tpl
{{/*
Expand the name of the chart.
*/}}
{{- define "poll-app.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" }}
{{- end }}

{{/*
Create a default fully qualified app name.
We truncate at 63 chars because some Kubernetes name fields are limited to
this (by the DNS naming spec).
If release name contains chart name it will be used as a full name.
*/}}
{{- define "poll-app.fullname" -}}
{{- if .Values.fullnameOverride }}
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" }}
{{- else }}
{{- $name := default .Chart.Name .Values.nameOverride }}
{{- if contains $name .Release.Name }}
{{- .Release.Name | trunc 63 | trimSuffix "-" }}
{{- else }}
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" }}
{{- end }}
{{- end }}
{{- end }}

{{/*
Create chart name and version as used by the chart label.
*/}}
{{- define "poll-app.chart" -}}
{{- printf "%s-%s" .Chart.Name .Chart.Version | replace "+" "_" | trunc 63 |
trimSuffix "-" }}
{{- end }}

{{/*
Common labels
*/}}
{{- define "poll-app.labels" -}}
helm.sh/chart: {{ include "poll-app.chart" . }}
{{ include "poll-app.selectorLabels" . }}
{{- if .Chart.AppVersion }}
app.kubernetes.io/version: {{ .Chart.AppVersion | quote }}
{{- end }}
app.kubernetes.io/managed-by: {{ .Release.Service }}
{{- end }}

{{/*
Selector labels
*/}}
{{- define "poll-app.selectorLabels" -}}
app.kubernetes.io/name: {{ include "poll-app.name" . }}
app.kubernetes.io/instance: {{ .Release.Name }}
{{- end }}

{{/*
Create the name of the service account to use
*/}}
{{- define "poll-app.serviceAccountName" -}}
{{- if .Values.serviceAccount.create }}
{{- default (include "poll-app.fullname" .) .Values.serviceAccount.name }}
```

```
{{- else }}
{{- default "default" .Values.serviceAccount.name }}
{{- end }}
{{- end }}

# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
name: {{ include "poll-app.fullname" . }}
data:
{{- range $key, $value := .Values.appConfig }}
{{ $key }}: {{ $value | quote }}
{{- end }}

# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: {{ include "poll-app.fullname" . }}
labels:
{{- include "poll-app.labels" . | nindent 4 }}
spec:
{{- if not .Values.autoscaling.enabled }}
replicas: {{ .Values.replicaCount }}
{{- end }}
selector:
matchLabels:
{{- include "poll-app.selectorLabels" . | nindent 6 }}
template:
metadata:
{{- with .Values.podAnnotations }}
annotations:
{{- toYaml . | nindent 8 }}
{{- end }}
labels:
{{- include "poll-app.labels" . | nindent 8 }}
{{- with .Values.podLabels }}
{{- toYaml . | nindent 8 }}
{{- end }}
spec:
{{- with .Values.imagePullSecrets }}
imagePullSecrets:
{{- range . }}
- name: {{ . }}
{{- end }}
{{- end }}
serviceAccountName: {{ include "poll-app.serviceAccountName" . }}
securityContext:
{{- toYaml .Values.podSecurityContext | nindent 8 }}
initContainers:
# Init container for Django migrations
- name: {{ .Chart.Name }}-migration
image: "{{ .Values.image.repository }}:{{ .Values.image.tag |
default .Chart.AppVersion }}"
imagePullPolicy: {{ .Values.image.pullPolicy }}
command:
- "sh"
- "-c"
- >
python manage.py migrate --no-input;
envFrom:
- configMapRef:
```

```
name: {{ include "poll-app.fullname" . }}
- secretRef:
name: {{ include "poll-app.fullname" . }}
resources:
{{- toYaml .Values.initResources | nindent 12 }}
# Init container for creating Django superuser
- name: {{ .Chart.Name }}-create-superuser
image: "{{ .Values.image.repository }}:{{ .Values.image.tag |
default .Chart.AppVersion }}"
imagePullPolicy: {{ .Values.image.pullPolicy }}
command:
- "sh"
- "-c"
- >
python manage.py createsuperuser --no-input || true;
envFrom:
- configMapRef:
name: {{ include "poll-app.fullname" . }}
- secretRef:
name: {{ include "poll-app.fullname" . }}
resources:
{{- toYaml .Values.initResources | nindent 12 }}
containers:
- name: {{ .Chart.Name }}
securityContext:
{{- toYaml .Values.securityContext | nindent 12 }}
image: "{{ .Values.image.repository }}:{{ .Values.image.tag |
default .Chart.AppVersion }}"
imagePullPolicy: {{ .Values.image.pullPolicy }}
command:
- "sh"
- "-c"
- >
python manage.py runserver 0.0.0.0:{{ .Values.service.port }};
envFrom:
- configMapRef:
name: {{ include "poll-app.fullname" . }}
- secretRef:
name: {{ include "poll-app.fullname" . }}
ports:
- name: http
containerPort: {{ .Values.service.port }}
protocol: TCP
livenessProbe:
{{- toYaml .Values.livenessProbe | nindent 12 }}
readinessProbe:
{{- toYaml .Values.readinessProbe | nindent 12 }}
resources:
{{- toYaml .Values.resources | nindent 12 }}
{{- with .Values.volumeMounts }}
volumeMounts:
{{- toYaml . | nindent 12 }}
{{- end }}
{{- with .Values.volumes }}
volumes:
{{- toYaml . | nindent 8 }}
{{- end }}
{{- with .Values.nodeSelector }}
nodeSelector:
{{- toYaml . | nindent 8 }}
{{- end }}
{{- with .Values.affinity }}
affinity:
```

```
{{- toYaml . | nindent 8 }}
{{- end }}
{{- with .Values.tolerations }}
tolerations:
{{- toYaml . | nindent 8 }}
{{- end }}

# hpa.yaml
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
name: {{ include "poll-app.fullname" . }}
labels:
{{- include "poll-app.labels" . | nindent 4 }}
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: {{ include "poll-app.fullname" . }}
minReplicas: {{ .Values.autoscaling.minReplicas }}
maxReplicas: {{ .Values.autoscaling.maxReplicas }}
metrics:
{{- if .Values.autoscaling.targetCPUUtilizationPercentage }}
- type: Resource
resource:
name: cpu
target:
type: Utilization
averageUtilization: {{ .Values.autoscaling.targetCPUUtilizationPercentage }}
{{- end }}
{{- if .Values.autoscaling.targetMemoryUtilizationPercentage }}
- type: Resource
resource:
name: memory
target:
type: Utilization
averageUtilization:
{{ .Values.autoscaling.targetMemoryUtilizationPercentage }}
{{- end }}
{{- end }}

# ingress.yaml
{{- if .Values.ingress.enabled -}}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: {{ include "poll-app.fullname" . }}
labels:
{{- include "poll-app.labels" . | nindent 4 }}
{{- with .Values.ingress.annotations }}
annotations:
{{- toYaml . | nindent 4 }}
{{- end }}
spec:
{{- with .Values.ingress.className }}
ingressClassName: {{ . }}
{{- end }}
{{- if .Values.ingress.tls }}
tls:
{{- range .Values.ingress.tls }}
- hosts:
{{- range .hosts }}
```

```
- {{ . | quote }}
{{- end }}
secretName: {{ .secretName }}
{{- end }}
{{- end }}
rules:
{{- range .Values.ingress.hosts }}
- host: {{ .host | quote }}
http:
paths:
{{- range .paths }}
- path: {{ .path }}
{{- with .pathType }}
pathType: {{ . }}
{{- end }}
backend:
service:
name: {{ include "poll-app.fullname" $ }}
port:
number: {{ $.Values.service.port }}
{{- end }}
{{- end }}
{{- end }}

# service.yaml
apiVersion: v1
kind: Service
metadata:
name: {{ include "poll-app.fullname" . }}
labels:
{{- include "poll-app.labels" . | nindent 4 }}
spec:
type: {{ .Values.service.type }}
ports:
- port: {{ .Values.service.port }}
targetPort: http
protocol: TCP
name: http
selector:
{{- include "poll-app.selectorLabels" . | nindent 4 }}

#serviceaccount.yaml
{{- if .Values.serviceAccount.create -}}
apiVersion: v1
kind: ServiceAccount
metadata:
name: {{ include "poll-app.serviceAccountName" . }}
labels:
{{- include "poll-app.labels" . | nindent 4 }}
{{- with .Values.serviceAccount.annotations }}
annotations:
{{- toYaml . | nindent 4 }}
{{- end }}
automountServiceAccountToken: {{ .Values.serviceAccount.automount }}
{{- end }}
```