МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Кафедра електроніки і комп'ютерної техніки

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ БАКАЛАВРА
НА ТЕМУ:

ПРОЕКТУВАННЯ KUBERNETES КЛАСТЕРА У ХМАРІ AWS

Завідувач кафедрою електроніки
і комп'ютерної техніки _____ А.С. Опанасюк

Керівник роботи _____ І.А. Кулик

Виконав студент гр. ТК-81 _____ М.Є. Петен

Суми 2022

РЕФЕРАТ

В кваліфікаційній роботі бакалавра був спроектований масштабований, відказостійкий Kubernetes кластер. Кластер розроблений за допомогою Terraform коду, це дає нам можливість перевикористовувати та підтримувати його дуже легко. Відказостійкість гарантуватиме нам працюючий кластер 99.9999% часу.

В кваліфікаційній роботі бакалавра проведений детальний опис ресурсів AWS для організації інфраструктури навколо кластера, helm чартів, Kubernetes об’єктів та способи взаємодії з ними.

Для виконання випускної роботи використано 7 літературних джерел. Область застосування даного пристрою – системи передачі даних, які застосовують проводові лінії зв'язку.

Кваліфікаційна робота бакалавра містить 32 сторінки тексту, 1 таблицю і 15 рисунків.

Ключові слова: AWS, Kubernetes, Helm, Terraform, Terragrunt, Docker.

ЗМІСТ

Введення

Kubernetes є проектом з відкритим вихідним кодом, який призначений для управління кластером контейнерів Linux як єдиною системою. Kubernetes управляє та запускає контейнери Docker на великій кількості хостів, а також забезпечує спільне розміщення та реплікацію великої кількості контейнерів. Проект розпочато Google і тепер підтримується багатьма компаніями, серед яких Microsoft, RedHat, IBM та Docker.

Компанія Google користується контейнерною технологією вже понад десять років. Вона розпочинала із запуску понад 2 млрд контейнерів протягом одного тижня. За допомогою проекту Kubernetes компанія ділиться своїм досвідом створення відкритої платформи, призначеної для запуску контейнерів, що масштабується.

Проект має дві мети. Якщо ви користуєтесь контейнерами Docker, виникає наступне питання про те, як масштабувати та запускати контейнери відразу на великій кількості хостів Docker, а також як виконувати їх балансування. У проекті пропонується високорівневий API, що визначає логічне групування контейнерів, що дозволяє визначати пули контейнерів, балансувати навантаження, а також задавати їх розміщення.

СПИСОК СКОРОЧЕНЬ

- AWS — Amazon Web Services;
- DNS - Domain Name System;
- RDS - Relational Database Service;
- EKS — Elastic Kubernetes Service;
- CSI - Container Storage Interface;
- EBS - Elastic Block Store;
- ELB — Elastic Load Balancer;
- IAM - Identity Access Management;
- ECR — Elastic Container Registry;
- VPC — Virtual Private Cloud;
- SSL - Secure Sockets Layer.

1. ОГЛЯД ЛІТЕРАТУРИ ТА ПОСТАНОВКА ЗАВДАННЯ ПРОЕКТУВАННЯ

1.1 Структура Kubernetes кластера

Kubernetes кластер складається з набору серверів, які називаються вузлами, що запускають контейнеризовані програми. Кожен кластер має принаймні один робочий вузол. Робочий вузол розміщує модулі, які є компонентами робочого навантаження програми. Control plane керує робочими вузлами та модулями в кластері. У продакшн оточеннях control plane зазвичай працює на декількох серверах і кластер зазвичай запускає декілька вузлів забезпечуючи відмовостійкість та високу доступність.

Controle Plane компоненти:

- kube-apiserver;
- etcd;
- kube-scheduler;
- kube-controller-manager;
- cloud-controller-manager.

Сервер API є компонентом Kubernetes панелі керування, який представляє API Kubernetes. API-сервер – це клієнтська частина панелі керування Kubernetes.

Основною реалізацією API-сервера Kubernetes є kube-apiserver. kube-apiserver призначений для горизонтального масштабування, тобто розгортання на кілька екземплярів. Ви можете запустити кілька екземплярів kube-apiserver та збалансувати трафік між цими екземплярами.

Etcd - розподілене та високонадійне сховище даних у форматі "ключ-значення", яке використовується як основне сховище всіх даних кластера в Kubernetes. Якщо ваш кластер Kubernetes використовує etcd як основне сховище, переконайтеся, що у вас настроєно резервне копіювання даних.

Kube-scheduler - компонент Control plane, який відстежує створені поди без прив'язаного вузла та вибирає вузол, на якому вони мають працювати.

Kube-controller-manager - компонент Control Plane запускає процеси контролера. Кожен контролер у свою чергу є окремим процесом, і для спрощення всі такі процеси скомпільовані в один двійковий файл і виконуються в одному процесі.

Ці контролери включають:

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

Cloud-controller-manager - запускає контролери, які взаємодіють із основними хмарними провайдерами. За допомогою cloud-controller-manager код як хмарних провайдерів, так і Kubernetes може розроблятися незалежно один від одного.

Наступні контролери залежать від хмарних провайдерів:

- Контролер вузла (Node Controller): перевіряє хмарний провайдер, щоб визначити, чи було видалено вузол у хмарі після того, як він перестав працювати;
- Контролер маршрутів (Route Controller): налаштовує маршрути в основній інфраструктурі хмари
- Контролер сервісів (Service Controller): створює, оновлює та видаляє балансувальники навантаження хмарного провайдера.
- Контролер тома (Volume Controller): створює, приєднує та монтує томи, а також взаємодіє з хмарним провайдером для оркестрації томів.

Компоненти вузла:

- kubelet;
- kube-proxy;

- container runtime.

Kubelet - агент, що працює на кожному вузлі у кластері. Він слідкує за тим, щоб контейнери були запущені в поді. Утилита kubelet принимает набор PodSpecs, и гарантирует работоспособность и исправность определённых в них контейнеров. Агент kubelet не отвечает за контейнеры, не созданные Kubernetes.

Kube-proxy - мережевий проксі, працюючий кожному вузлі в кластері, і реалізує частина концепції сервіс. kube-proxy конфігурує правила мережі на вузлах. За допомогою них дозволяються мережеві підключення до ваших подів зсередини та зовні кластера, використовує рівень фільтрації пакетів операційної системи, якщо він доступний. Інакше kube-proxy сам обробляє передачу мережевого трафіку.

Container-runtime - це програма, призначена для виконання контейнерів. Kubernetes підтримує кілька середовищ для запуску контейнерів: Docker, containerd, CRI-O, та будь-яка реалізація Kubernetes CRI (Container Runtime Interface).

Kubernetes компоненти:

- Deployments та ReplicaSet;
- StatefulSet
- DaemonSet
- Job і CronJob

Deployments та ReplicaSet. Deployments добре підходить для керування робочим навантаженням програми без збереження стану у вашому кластері, де будь-який Pod у розгортанні є взаємозамінним і може бути замінений, якщо це необхідно.

StatefulSet дозволяє запускати один або кілька пов'язаних модулів, які якимось чином відстежують стан. Наприклад, якщо ваше робоче навантаження постійно записує дані, ви можете запустити StatefulSet, який відповідає кожному Pod з PersistentVolume. Ваш код, що виконується в Pods для цього StatefulSet, може реплікувати дані на інші Pods в тому ж StatefulSet, щоб покращити загальну стійкість.

DaemonSet визначає модулі, які забезпечують локальні можливості вузла. Вони можуть мати основне значення для роботи вашого кластера, наприклад, допоміжний інструмент для роботи в мережі, або бути частиною надбудови.

Кожного разу, коли ви додаєте до свого кластера вузол, який відповідає специфікації в DaemonSet, Control plane планує Pod для цього DaemonSet на новий вузол.

Job і CronJob визначають завдання, які виконуються до завершення, а потім зупиняються. Jobs представляють собою одноразові завдання, тоді як CronJobs повторюються згідно з графіком.

1.2 Контейнеризація та Helm чарти

Контейнеризація - це метод віртуалізації, при якому єдиний неймспейс в ядрі операційної системи поділяється на кілька незалежних логічних розділів. Один екземпляр такого розділу називається контейнером. У кожному контейнері можна запускати одну програму, ізольовану від іншої системи. Кожна програма в контейнері отримує свою приватну мережу та віртуальну файлову систему, яка не використовується спільно з іншими контейнерами або хостом.

Docker – це платформа для розробки, доставки та запуску контейнерних програм (container runtime). Docker дозволяє створювати контейнери, автоматизувати їх запуск та розгортання, керує життєвим циклом. Він дозволяє запускати безліч контейнерів на одній машині.

Helm — це інструмент, який спрощує встановлення та керування додатками Kubernetes. Helm відтворює ваші шаблони та взаємодіє з Kubernetes API. Чарти — це пакети Helm, які містять принаймні дві речі:

- опис пакета (Chart.yaml);
- один або кілька шаблонів, які містять Kubernetes маніфест файли.

Структура Helm чарта:

- charts/ - у цій директорії розташовуються вкладені чарти. Цієї директорії не буде, якщо наша система не має залежностей від інших чартів;
- Chart.yaml — файл зі службовою інформацією про чарт і додатки, який він розгортає і залежність чарту;
- templates/*.yaml* — файли із шаблонами об'єктів Kuberenetes. Можна складати шаблони у підпапки, helm їх сканує рекурсивно.
- templates/NOTES.txt — містить шаблон із довідковою інформацією, яка буде виведена користувачеві під час встановлення чарту.
- values.yaml — файл конфігурації за замовчуванням для всіх оточень, містить властивості чарта з дефолтними значеннями. Також важливо використовувати значення файлів по оточенням для перекриття дефолтних значень.

Виклик helm install створює release object і release version secret.

Виклик helm upgrade вимагає наявності release object (який може змінювати) і створює новий release version secret, що містить нові значення і підготовлений маніфест. Відповідно якщо міняти кластер через kubectl, то Helm не побачить змін, і це загрожує помилками.

1.3 Огляд хмари AWS

Amazon Web Services – це набір хмарних сервісів від компанії Amazon. На єдиній платформі користувачі можуть замовити обчислювальні ресурси, сховище, інфраструктуру, послуги з готовими для використання інструментами. AWS налічує понад двухсот повнофункціональних сервісів.

Набір сервісів, що використовуються в роботі:

- EC2-Instances - є одним із сервісів Amazon Web Services, що дозволяє користувачеві орендувати віртуальний сервер, які називаються instance. Для запуску віртуальних серверів використовуються попередньо налаштовані образи;

- EC2-ELB - Балансувальник навантаження. Розподіляє мережевий трафік для покращення масштабованості додатків;
- Route53 - це високодоступний хмарний веб-сервіс системи доменних імен (DNS), що масштабується. Надсилає запити користувачів до інфраструктури AWS, наприклад до EC2 інстансів, балансувальників навантаження Elastic Load Balancing або кошиків S3;
- Relational Database Service (RDS) - це керований сервіс, який спрощує настроювання, використання та масштабування реляційної бази даних у хмарі. Цей сервіс надає економічні ресурси, що масштабуються, і одночасно керує трудомісткими завданнями адміністрування баз даних;
- Secrets Manager - дозволяє захищати конфіденційні дані, що використовуються для доступу до програм, сервісів та IT-ресурсів. Сервіс надає можливість простої ротації та отримання даних для доступу до БД, ключів API та інших конфіденційних даних, а також управління ними протягом усього життєвого циклу;
- S3 - протокол передачі, розроблений компанією Amazon. Об'єктне сховище;
- DynamoDB - повністю керована безсерверна база даних NoSQL на основі пар «ключ-значення», створена для запуску високопродуктивних програм у будь-якому масштабі. DynamoDB пропонує вбудований захист, безперервне резервне копіювання, автоматичну реплікацію у кількох регіонах, кешування в пам'яті та інструменти експорту даних;
- Identity Access Management (IAM) - забезпечує точний контроль доступу у всіх сервісах AWS. За допомогою IAM ми можемо вказати, хто може отримувати доступ до певних сервісів та ресурсів та за яких умов. Завдяки політикам IAM ми керуємо дозволами для співробітників та систем, надаючи дозволи з найменшими привілеями;
- Elastic Container Registry (ECR) - це повністю автоматизований реєстр контейнерів, що дозволяє розробникам легко розгортати образи контейнерів та артефакти, а також ділитися ними.

- Elastic Kubernetes Service (EKS) - це керований сервіс Kubernetes, який дозволяє запускати Kubernetes на AWS та в локальному середовищі. Amazon EKS має сертифікат сумісності з Kubernetes, тому наявні програми, що працюють на відкритій версії Kubernetes, сумісні з Amazon EKS.

1.4 Інфраструктура як код. Terraform

Terraform — це Infrastructure as a Code (IaaC) інстррумент з відкритим вихідним кодом, створений HashiCorp. Користувачі визначають та надають інфраструктуру за допомогою декларативної мови конфігурації, відомої як HashiCorp Configuration Language (HCL), або за бажанням JSON.

За допомогою Terraform ми опишемо всі необхідні ресурси для створення кластера та всієї інфраструктури як код, тож ми зможемо перевикористовувати його, наприклад, для інших проєктів, задач змінюючи лише змінні.

Terraform створює та керує ресурсами на хмарних платформах та інших сервісах за допомогою своїх інтерфейсів прикладного програмування (API). Провайдери дозволяють Terraform працювати практично з будь-якою платформою або сервісом із доступним API.

Основний робочий процес Terraform складається з трьох етапів:

Write: ви визначаєте ресурси, які можуть бути в кількох хмарних постачальників і служб. Наприклад, ви можете створити конфігурацію для розгортання програми на віртуальних машинах у мережі віртуальної приватної хмари (VPC) з групами безпеки та балансувальником навантаження.

Plan: Terraform створює план виконання, що описує інфраструктуру, яку вона створить, оновить або знищить на основі наявної інфраструктури та вашої конфігурації.

Apply: після затвердження Terraform виконує запропоновані операції в правильному порядку, дотримуючись будь-яких залежностей від ресурсів. Наприклад, якщо ви оновлюєте властивості VPC та змінюєте кількість

віртуальних машин у цьому VPC, Terraform відтворить VPC перед масштабуванням віртуальних машин.

Terraform підтримує багаторазові компоненти конфігурації, які називаються модулями, які визначають настроювані колекції інфраструктури, заощаджуючи час і заохочуючи до найкращих практик. Ви можете використовувати загальнодоступні модулі з реєстру Terraform або написати власні.

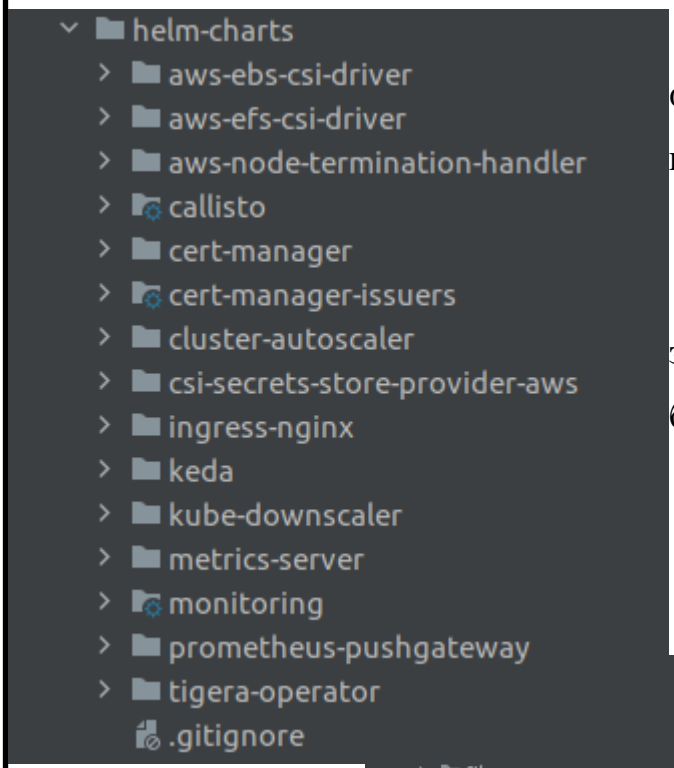
Terragrunt – це wrapper для Terraform, який надає додаткові інструменти для зберігання ваших конфігурацій Terraform, роботи з кількома модулями Terraform та керування віддаленим станом.

1.5 Постановка завдання проектування

КТП – кінцевий термінальний пристрій, що може бути джерелом інформації або її споживачем (одержувачем), або тим і іншим одночасно. КТП може являти собою персональний комп'ютер, супер ЕОМ, термінал, пристрій (систему) збирання даних (АЦІ), касовий апарат, навігаційний приймач або будь-яку апаратуру, що може приймати, зберігати, обробляти та передавати дані. Часто для визначення КТП застосовують міжнародний термін DTE (date terminal equipment).

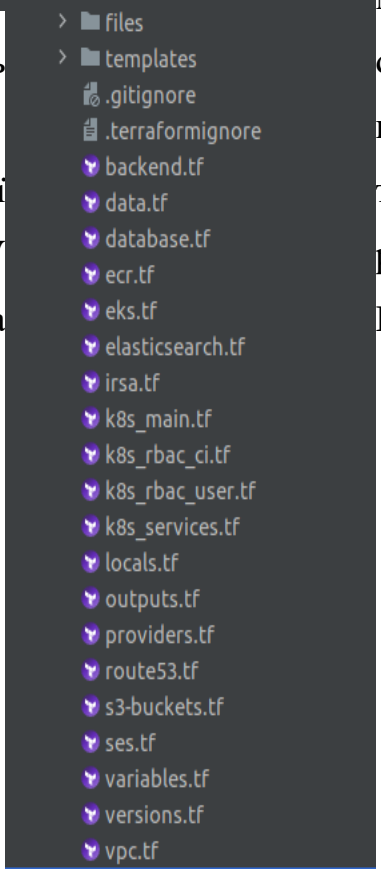
2. РОЗРОБКА TERRAFORM КОДУ, ОПИС НЕОБХІДНИХ РЕСУРСІВ

2.1 Архітектура інфраструктури



Спочатку створюємо структуру файлів, в яких ми будемо описувати наш код.

В директорії terraform/aws знаходяться файли з описом ресурсів, які будуть створені.



Файл backend.tf визначає місто, де ми будемо брати наш стейт файл, data.tf містить data об’єкти, які дозволяють Terraform використовувати зовнішні дані, вихідні дані функції та дані з інших конфігурацій, модулів. У providers.tf під’єднуємо провайдера AWS, Kubernetes, та Helm:

```
provider "aws" {
  region = var.region
  assume_role {
    role_arn = var.aws_role_arn
  }
  allowed_account_ids     = var.provider_allowed_account_ids
  skip_metadata_api_check = true
  default_tags {
    tags = merge({
      Prefix : var.prefix
      Environment : var.environment
      IaC = "Terraform"
      },
      var.tags
```

```
provider "kubernetes" {
  host                   = data.aws_eks_cluster.cluster.endpoint
  cluster_ca_certificate = base64decode(data.aws_eks_cluster.cluster.certificate_authority.0.data)
  token                  = data.aws_eks_cluster_auth.cluster.token
}

provider "helm" {
  kubernetes {
    host                   = data.aws_eks_cluster.cluster.endpoint
    cluster_ca_certificate = base64decode(data.aws_eks_cluster.cluster.certificate_authority.0.data)
    token                  = data.aws_eks_cluster_auth.cluster.token
  }
}
```

Файл eks.tf містить модуль для створення самого кластера — terraformaws-modules/eks/aws. Далі нам необхідно описати мережі для нашого кластера, конфігурація мереж знаходиться

у файлі vpc.tf де ми підключаємо модуль terraform-aws-modules/vpc/aws, створюємо публічні та приватні підмережі та elastic ip адрес для балансувальника навантаження (ELB). Після створення мереж переходимо до опису DNS у Route53. Ресурс за допомогою якого ми створюємо саму DNS зону — aws_route53_zone, за допомогою aws_route53_record ми створюємо необхідні записи у цій зоні. Далі переходимо до створення ECR, перш за все створюємо репозиторій через ресурс aws_ecr_repository та додаємо IAM полісі. Всі необхідні S3 кошики ми створюємо через модуль terraform-aws-modules/s3-bucket/aws і також додаємо IAM та lifecycle полісі. Для коректної роботи наших сервісів нам необхідна база даних (RDS), яку ми створюємо у файлі database.tf за допомогою модуля terraform-aws-modules/rds/aws та додаємо паролі у Secrets Manager.

Наступний крок — опис ресурсів всередені Kubernetes кластера, ресурси представлені у вигляді Helm чартів, робимо це у файлі k8s_services.tf. Необхідні ресурси:

- AWS cluster autoscaler — сервіс, який слідкує за навантаженням на вузлах і в залежності від цього додаємо чи прибираємо їх;
- AWS EBS CSI driver - драйвер Amazon Elastic Block Store Container Storage Interface (CSI) забезпечує інтерфейс CSI, який використовується Container Orchestrators для керування життєвим циклом томів Amazon EBS;
- Cert Manager — випускає SSL сертифікати для наших доменів та слідкує за тим, щоб вони вчасно оновлювались;
- Monitoring — допомагає нам слідкувати за навантаженням всередені кластера, дивитись історію логів контейнерів, робити алерти. Використовуємо для цього Prometheus оператора, Loki та Grafana;
- Ingress - відкриває маршрути HTTP і HTTPS поза кластером для служб всередині кластера. Маршрутизація трафіку контролюється правилами, визначеними на ресурсі;
- Kube downscaller — виключає необхідні ресурси за розкладом, потрібен для економії ресурсів та коштів, які витрачаються на утримання серверів.
- AWS Backup — сервіс для бекапа важливих даних всередені нашого кластера.

2.2 Використання змінних та динамічні ресурси

Змінні Terraform, роблять конфігурацію більш гнучкою і спрощують

```
variable "map_roles" {
  description = "Additional IAM roles to add to the aws-auth configmap."
  type = list(object({
    rolearn  = string
    username = string
    groups   = list(string)
  }))

  default = []
}
```

повторне використання даних. Правда, незважаючи на назву, вони не можуть змінюватися в процесі роботи Terraform. Але їх можна задавати динамічно перед виконанням. Визначення змінних може з'являтися у будь-якій частині конфігурації, оскільки це інструкція, а опис те, що хочемо отримати.

Типи змінних в Terraform:

- string;
- number;
- bool;
- map або object - асоціативний масив, що складається з пар ключ-значення;
- list.

Для звернення змінної використовується синтаксис var. Змінні можуть підставлятися у будь-яких місцях, де є привласнення

Значення вхідних змінних кореневого модуля можна зібрати у файлах визначення змінних і передати разом за допомогою -var-file=FILE .

Для всіх файлів, які відповідають terraform.tfvars або *.auto.tfvars, присутнім у поточному каталозі, Terraform автоматично завантажує їх для заповнення змінних. Якщо файл знаходиться в іншому місці, ви можете перейти шлях до файлу за допомогою прапорця -var-file. У директорії

terraform/aws/environments знаходяться файли, які містять саме значення усіх змінних, за допомогою цього ми можемо динамічно створювати різні оточення базуючись лише на змінних. Тут ми описуємо шлях до нашого стейт файлу:

```
remote_state {
  backend  = "s3"
  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }
  config = {
    bucket                  = "${local.common.prefix}-${local.environment}-tfstate"
    key                     = "infrastructure/terraform.tfstate"
    region                  = "${local.common.region}"
    encrypt                 = true
    dynamodb_table          = "${local.common.prefix}-${local.environment}-terraform-lock-table"
    acl                     = "bucket-owner-full-control"
    role_arn                = local.aws_role_arn
    skip_metadata_api_check = true
  }
}
```

Далі вказуємо значення для наших неймспейсів всередені кластера, production та staging, описуємо конфігурацію бази даних (RDS) та створюємо DNS зони.

У блочних конструкціях верхнього рівня, як ресурси, вирази зазвичай можна використовувати лише під час призначення значення аргументу за допомогою форми ім’я = вираз. Це охоплює багато застосувань, але деякі типи ресурсів включають повторювані вкладені блоки в свої аргументи, які зазвичай представляють окремі об’єкти, які пов’язані з (або вбудовані в) об’єкт. Ви можете динамічно створювати повторювані вкладені блоки, як налаштування, використовуючи спеціальний динамічний тип блоку, який підтримується всередині блоків ресурсів, даних та провайдерів. За замовчуванням блок ресурсів налаштовує один реальний об’єкт інфраструктури (і аналогічно блок модуля включає вміст дочірнього модуля в конфігурацію один раз). Однак іноді вам потрібно керувати кількома подібними об’єктами (наприклад, фіксованим 5пулом екземплярів обчислень), не записуючи окремий блок для кожного з них. Мета-аргумент for_each приймає вирази карти або набору. Однак, на відміну від більшості аргументів, значення for_each має бути відоме, перш ніж Terraform виконає будь-які дії з віддаленим ресурсом. Це означає, що for_each не може посилатися на атрибути ресурсу, які не відомі до моменту застосування

конфігурації (наприклад, унікальний ідентифікатор, створений віддаленим API під час створення об’єкта). Значення for_each має бути картою або набором з одним елементом на потрібний екземпляр ресурсу. Надаючи набір, ви повинні використовувати вираз, який явно повертає значення набору, як функція toset; щоб уникнути небажаних сюрпризів під час перетворення, аргумент for_each не перетворює списки або кортежі в набори неявно.

```
for_each = { for key, value in local.database_instances : key => value if value["enabled"] == true }

source  = "terraform-aws-modules/rds/aws"
version = "3.5.0"

identifier                = "${var.prefix}-${var.environment}-${each.key}"
engine                    = each.value["engine"]
engine_version            = each.value["engine_version"]
family                    = each.value["family"]
major_engine_version      = each.value["major_engine_version"]
auto_minor_version_upgrade = each.value["auto_minor_version_upgrade"]
instance_class            = each.value["instance_class"]
```

Якщо вам потрібно оголосити екземпляри ресурсу на основі вкладеної структури даних або комбінацій елементів із кількох структур даних, ви можете використовувати вирази та функції Terraform для отримання відповідного значення. Використання for_each для створення бази даних:

За допомогою якого ми можемо створити декілька баз даних, використовуючи лише один ресурс у вигляді модуля, опираючись на наші змінні.

3. РОЗРОБКА HELM ЧАРТІВ

3.1 Опис Kubernetes маніфестів, розробка чарту

Kubernetes маніфест - Специфікація об'єкта API Kubernetes у форматі JSON або YAML. У маніфесті вказується бажаний стан об'єкта, який буде підтримувати Kubernetes, коли ви застосуєте маніфест. Кожен конфігураційний файл може мати кілька маніфестів.

Щоб продумати структуру загального шаблону - подивимося, які блоки загальні, а які будуть відрізнятися, і як ми їх можемо максимально гнучко підключати.

Розглядатимемо лише файл deployment.yaml – інші шаблони потім зробимо за його подобою:

- metadata;
- name: загальний, значення будемо брати з файлу values.yaml конкретного проекту
- annotations: загальний, тут у нас тільки reloader.stakater.com/auto, і у всіх він буде один і завжди true (хоча можливі варіанти – див. Kubernetes: ConfigMap і Secrets - auto-reload даних у подах)
- labels: загальний, і буде використовуватися в кількох місцях, наприклад, у деплойменті нижче – spec.template.metadata.labels, і у файлі з Kubernetes
- spec;
- replicas: загальний, значення будемо брати з файлу values.yaml конкретного проекту
- strategy:
- type: загальний, значення будемо брати з файлу values.yaml конкретного проекту
- selector:
- matchLabels: загальний, і буде використовуватися в декількох місцях, наприклад у Cronjobs та Services – винесемо у _helpers.yaml name:

загальний, значення будемо брати з файлу values.yaml конкретного проекту

- image: загальний, значення будемо брати з файлу values.yaml конкретного проекту env: ось тут найцікавіше:

Можемо винести цілим блоком у values.yaml кожного проекту плюси: не треба буде окремо описувати самі значення - можемо їх вказати прямо тут, у полі value самих змінних мінуси: практично у всіх однакові змінні – дублюємо код між проектами роздутий файл values.yaml, т.к. змінних багато не всі змінні йдуть у вигляді ЗНАЧЕННЯ – деякі будуть брати значення через valueFrom.secretKeyRef.<KUBE_SECRET_NAME>, значить значення у values.yaml задати не вийде можемо включити до загального _helpers.yaml, і буде загальний набір змінних, кожна укладена у свій if – якщо у values.yaml значення знайдено, то змінна створюється у загальному шаблоні і на випадок, якщо у проекту зовсім окремий набір змінних - то в загальному шаблоні перед блоком env можна додати умову типу if {{ .Values.chartSettings.customEnvs == true }} - пропускатимемо включення змінних з _helpers.yaml або values.yaml , і підключати файл шаблону через tpl .File.Get project1/envs.yaml volumeMounts: теж може відрізнятися, подивимося потім ports - containerPort: буде завжди і у всіх, беремо з values.yaml (до речі – порти не обов'язкові взагалі) livenessProbe, readinessProbe: в цілому теж буде загальний, з httpGet.path і httpGet.port - можна винести в _helpers.yaml, але додати також умову customProbes, і при необхідності - підключати через .File.Get

Ресурси requests.cpu, requests.memory – буде у всіх чи – треба думати limits.cpu, limits.memory – пам'ять точно обмежуватимемо всім (або?) – треба думати

Volumes: - теж може відрізнятися, подивимося потім. ImagePullSecrets: однаковий у всіх:

- hpa.yaml – описуємо HPA
- network.yaml – Service, Ingress
- secrets.yaml – Kubernetes Secrets

- rbac.yaml – підключення власних груп до створюваних неймспейсів
- cronjobs.yaml – Kubernetes Cronjobs _helpers.tpl – і наш “помічник” – сюди ключимо Helm Named Templates

Власне, основна ідея така: загальний чарт із шаблонами, які лежать у templates у templates зберігаємо файли deployment.yaml, hpa.yaml, _helpers.tpl, etc поряд з templates створимо каталог projects і всередині нього – каталоги на ім'я проектів усередині кожного створимо каталог env всередині якого буде dev, stage, prod всередині яких будуть окремі values.yaml та secrets.yaml

Першим у нашому шаблоні нам зустрінеться мітки блоків, які ми хочемо винести в _helpers.yaml, а потім підключити в розгортанні та інші шаблони. Ідея названих шаблонів в тому, що ми один раз пишем який-то блок, який потім може включати в'їзд у нашому чарті. Плюс – вони дозволяють вибрати з основного шаблону лішний код, щоб зробити його більш красивим і лаконичним.

Файли названих шаблонів починаються з _ і закінчуються розширенням .tpl.

Самий відомий приклад – _helpers.tpl, який і будемо використовувати.

Опис кожного шаблону у файлі _helpers.tpl починається з define, закінчується end.

Ім'я названого шаблону, як правило, включає в себе ім'я чарта і блоку, який цей шаблон додає, але ніхто не змішує використовувати будь-яке інше, наприклад, у цьому випадку загальне ім'я в них буде помічниками.

З неудобства – тут в іменах неможливо явно використовувати значення типу .Chart.Name – можна створити змінну. Поглядим – будем ли це робити, поки просто захардкодим ім’я помічників.

Додамо в values.yaml параметр, який відключатиме include шаблону з _helpers.tpl, назвемо його customEnvs, і заразом параметр, який дозволить передавати шлях до файлу зі змінними – customEnvsFile

Описуємо всі необхідні типи ресурсів для повноцінного функціонування об’єкта всередині кластера. Deployment:

```
{{- if .Values.api.enabled }}
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "common.names.fullname" . }}-api
  labels:
    {{- include "common.labels.standart" . | nindent 4 }}
    app.kubernetes.io/name: {{ include "common.names.name" . }}-api
spec:
  replicas: {{ .Values.api.replicaCount }}
  selector:
    matchLabels:
      {{- include "common.labels.selector" . | nindent 6 }}
      app.kubernetes.io/name: {{ include "common.names.name" . }}-api
  template:
    metadata:
      labels:
        {{- include "common.labels.selector" . | nindent 8 }}
        app.kubernetes.io/name: {{ include "common.names.name" . }}-api
        {{- if .Values.api.podLabels }}
        {{- include "common.tplvalues.render" (dict "value" .Values.api.podLabels "context" $) | nindent 8 }}
        {{- end }}
      {{- if .Values.api.podAnnotations }}
      annotations: {{- include "common.tplvalues.render" ( dict "value" .Values.api.podAnnotations "context" $) | nindent 8 }}
      {{- end }}
```

Далі необхідно створити Service для взаємодії подів між собою та для можливості доєднатися до нього зовні через Ingress:

Наступним кроком створюємо безпосередньо Ingress:

```
{{- if .Values.api.enabled }}
---
apiVersion: v1
kind: Service
metadata:
  name: {{ include "common.names.fullname" . }}-api
  labels:
    {{- include "common.labels.standart" . | nindent 4 }}
    app.kubernetes.io/name: {{ include "common.names.name" . }}-api
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: service-port
      protocol: TCP
      name: service-port
  selector:
    {{- include "common.labels.selector" . | nindent 4 }}
    app.kubernetes.io/name: {{ include "common.names.name" . }}-api
{{- end }}
```

```
{{- if and .Values.api.enabled .Values.api.ingress.enabled }}
---
{{- $fullName := printf "%s-api" ( include "common.names.fullname" . ) -}}
{{- $svcPort := 80 -}}
{{- if semverCompare ">=1.14-0" .Capabilities.KubeVersion.GitVersion -}}
apiVersion: networking.k8s.io/v1beta1
{{- else -}}
apiVersion: extensions/v1beta1
{{- end }}
kind: Ingress
metadata:
  name: {{ $fullName }}
  labels:
    {{- include "common.labels.standart" . | nindent 4 }}
    app.kubernetes.io/name: {{ include "common.names.name" . }}-api
  {{- with .Values.api.ingress.annotations }}
  annotations:
    {{- toYaml . | nindent 4 }}
  {{- end }}
spec:
  {{- if .Values.api.ingress.tls }}
  tls:
    {{- range $i, $v := .Values.api.ingress.tls }}
    - hosts:
      {{- range $v.hosts }}
        - {{ . | quote }}
      {{- end }}
```

Значення для цих об’єктів вказуємо у values.yaml файлах, для динамічного застосування Helm чартів. Дані, розміщені всередині ключа global, будуть доступні як у поточному чарті, так і у всіх вкладених чартах (сабчарти, subcharts).

Дані, розміщені всередині довільного ключа SOMEKEY, будуть доступні в поточному чарті та у вкладеному чарті з ім'ям SOMEKEY.

Файл values.yaml — стандартний файл для зберігання даних. Дані можуть також передаватися такими способами:

За допомогою параметра --values=PATH_TO_FILE може бути вказаний окремий файл даних (може бути вказано кілька параметрів, по одному для кожного файлу даних).

За допомогою параметрів --set key1.key2.key3.array[0]=one, --set key1.key2.key3.array[1]=two можуть бути вказані безпосередньо пари ключ-значення (може бути вказано кілька параметрів, дивись також --set-string key = forced_string_value).

Спочатку об’являємо змінні для АПІ:

```
api-cmn:
  API_CMN_MAILER_API_KEY: ""
api-elastic:
  API_CONN_ELASTIC_SECRETKEY: ""
api-db:
  API_CONN_DB_PASSWORD: '{{ index .Values "postgresql-ha" "postgresql" "password" }}'
api-rabbitmq:
  API_CONN_RABBITMQ_PASSWORD: '{{ .Values.rabbitmq.auth.password }}'
api-sms:
  API_CONN_SMS_PASSWORD: ""
api-google:
  API_GOOGLE_CALENDAR_PRIVATE_KEY: ""
api-multichanel:
  MULTICHANNEL_FTP_PASSWORD: ""
mirror-api-token:
  MIRROR_API_TOKEN: ""
ocr:
  API_OCR_TEST_AUTOPOCKIUP_PRIVATE_KEY: ""
  API_OCR_NSF_AUTOPOCKIUP_PRIVATE_KEY: ""
  API_OCR_QN_AUTOPOCKIUP_PRIVATE_KEY: ""
public-api-redis:
  REDIS_PASSWORD: '{{ .Values.redis.password }}'
public-api-ujwt:
  UJWT_PRIVATE_PASSPHRASE: ""
```

Після чого описуємо сам ресурс:

```yaml
api:
  enabled: true
  replicaCount: 1
  image:
    repository: "087253572086.dkr.ecr.eu-west-1.amazonaws.com"
    tag: '{{ .Values.global.image.tag }}'
    pullPolicy: "IfNotPresent"
  imagePullSecrets: []
  podLabels:
    kubernetes.io/ingress.allow: "true"
  podAnnotations: {}
  serviceAccount:
    create: true
    name:
  podSecurityContext: {}
  securityContext: {}
  service:
    type: ClusterIP
    port: 80
    name: service-port
```

3.1 Застосування зовнішніх Helm чартів

Безпосередня передача даних відбувається за допомогою АПД -апаратури передачі даних, за міжнародним позначенням DCE (date communications equipment). Функція DCE полягає в тому, щоб забезпечити можливість обміну

За промовчанням Helm використовує офіційний репозиторій чартів Kubernetes. Він містить ретельно опрацьовані актуальні чарти на вирішення безлічі прикладних завдань. Цей репозиторій називається стабільний. Також ми можемо використовувати інші репозиторії хелм чартів, наприклад, bitnami

Для використання зовнішніх helm чартів у нашому сетапі, ми можемо підключити їх як залежність нашого чарту в Chart.yaml

```yaml
apiVersion: v2
name: cis
description: A Helm chart for Kubernetes charts do not define any templates and therefore cannot be deployed.
type: application
version: 0.1.0
appVersion: 0.1.0

dependencies:
  - name: common
    repository: file://../common
    version: x.x.x
  - name: rabbitmq
    version: "8.9.2"
    repository: "https://charts.bitnami.com/bitnami"
    condition: rabbitmq.enabled
  - name: elasticsearch
    version: "7.10.2"
    repository: "https://helm.elastic.co"
    condition: elasticsearch.enabled
```

Під'єднуємо чарт з RabbitMQ з репозиторія bitnami, він нам потрібен як платформа для обміну повідомленнями та побудови черг. Далі під'єднуємо Elasticsearch з його ж офіційного репозиторію - це пошукова система з відкритим вихідним кодом, яка дозволяє в режимі реального часу шукати та аналізувати дані в нереляційному сховищі. Налаштовуємо ці чарти за допомогою values.yaml, створюємо структуру як в вихідному варіанті чарта та додаємо свої значення:

```yaml
rabbitmq:
  enabled: true
  auth:
    username: user
    password: ""
    erlangCookie: ""
  memoryHighWatermark:
    enabled: false
  extraPlugins: ''
  loadDefinition:
    enabled: true
    existingSecret: '{{ .Release.Name }}-load-definition'
  extraSecrets:
    load-definition:
      load_definition.json: |
        {
          "vhost": "/",
          "name": "ha-two",
          "pattern": ".*",
          "apply-to": "all",
          "definition": {
            "ha-mode": "exactly",
            "ha-params": 2,
            "ha-sync-mode": "automatic"
          }
        }
```

4 ДЕПЛОЙ KUBERNETES КЛАСТЕРА ТА НЕОБХІДНИХ ДЛЯ ФУНКЦІОНУВАННЯ РЕСУРСІВ

Виконувати деплой ми будемо з ізольованого оточення, представленого докер контейнером у якому встановлені всі необхідні утиліти для роботи з Terraform кодом. Dockerfile:

```
ARG TERRAFORM_VERSION="1.0.11"
RUN curl -fsSLO https://releases.hashicorp.com/terraform/${TERRAFORM_VERSION}/terraform_${TERRAFORM_VERSION}_linux_amd64
    unzip ./terraform_${TERRAFORM_VERSION}_linux_amd64.zip -d /usr/local/bin && \
    rm ./terraform_*.zip

ARG TERRAGRUNT_VERSION="v0.35.13"
RUN curl -fsSL https://github.com/gruntwork-io/terragrunt/releases/download/${TERRAGRUNT_VERSION}/terragrunt_linux_amd64
    chmod +x /usr/local/bin/*

ARG HELM_VERSION="v3.7.2"
RUN curl -fsSLO https://get.helm.sh/helm-${HELM_VERSION}-linux-amd64.tar.gz && \
        tar -zxvf helm-${HELM_VERSION}-linux-amd64.tar.gz && \
        mv ./linux-amd64/helm /usr/local/bin/helm && \
        rm -rf ./helm-${HELM_VERSION}-linux-amd64.tar.gz

ARG KUBERNETES_VERSION="v1.21.2"
RUN curl -fsSL https://storage.googleapis.com/kubernetes-release/release/${KUBERNETES_VERSION}/bin/linux/amd64/kubectl -
        chmod +x /usr/local/bin/kubectl

ARG AWS_CLI_VERSION="2.4.6"
RUN curl -fsSL "https://awscli.amazonaws.com/awscli-exe-linux-x86_64-${AWS_CLI_VERSION}.zip" -o "awscliv2.zip" \
    && unzip awscliv2.zip \
    && aws/install \
```

Таким чином, ми можемо бути впевненими, що версії усіх інструментів коректні, та поведінка при деплої буде передбачуваною. Запускаємо цей контейнер за допомогою bash скрипта в якому маунтимо волум з нашим кодом, прокидуємо необхідні змінні та конфіг файли.

Всередені цього контейнера ініціалізуємо наш код. Ми використовуємо wrapper

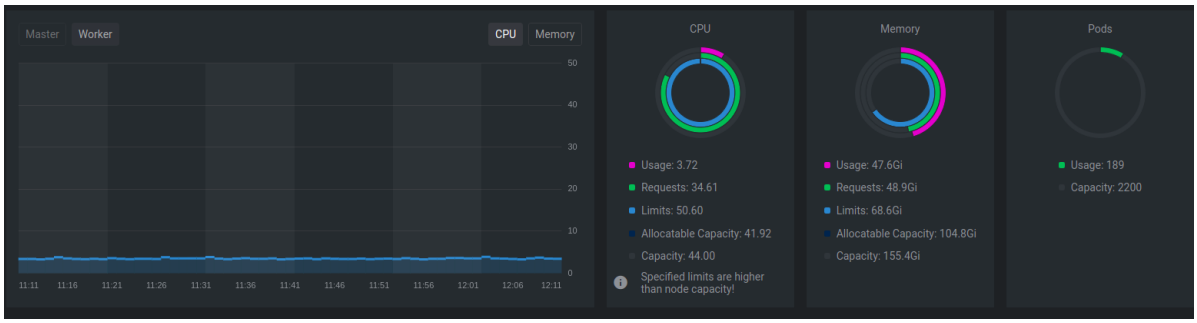
```
AUTH_WRAPPER=${AUTH_WRAPPER:-""}
IMAGE=$(docker build -f run_env.dockerfile -q --build-arg UID=$(id -u) --build-arg USER=${USER} .)

${AUTH_WRAPPER} docker run --rm -it \
-u "$(id -u)" \
-v "${PWD}/:/home/terraform/workspace" \
-v "${AWS_CONFIG_FILE:-~/.aws/config}:/home/terraform/.aws/config" \
-v "/etc/bash_completion.d:/etc/bash_completion.d:ro" \
-v "${HOME}/.bashrc:/home/terraform/.bashrc:ro" \
-w /home/terraform/workspace/aws/environments \
-e AWS_ACCESS_KEY_ID \
-e AWS_SECRET_ACCESS_KEY \
-e AWS_SESSION_TOKEN \
-e AWS_DEFAULT_REGION \
"${IMAGE}" \
bash
```

для terraform — terragrunt. Команда terragrunt init використовується для ініціалізації робочого каталогу, що містить файли конфігурації Terraform. Це перша команда, яку слід запустити після написання нової конфігурації Terraform або клонування наявної з контролю версій. Цю команду безпечно виконувати кілька разів.

Після ініціалізації робимо terragrunt plan. Він зчитує поточний стан будь-яких уже існуючих віддалених об'єктів, щоб переконатися, що стан Terraform оновлений. Порівнює поточну конфігурацію з попереднім станом і відзначає будь-які відмінності. Пропонує набір дій щодо зміни, які повинні, якщо їх застосувати, зробити так, щоб віддалені об'єкти відповідали конфігурації. Згідно кращим практикам terragrunt plan зберігаємо у файл, він нам знадобиться у наступному кроці. Після вивчення вихідних даних команди terragrunt plan, робимо terragrunt apply - terragrunt apply plan.out -auto-aprove, ключ auto-aprove потрібен для того, щоб terragrunt apply не запитував у нас ще раз, чи впевнені ми в тому, що будемо деплоїти зміни зазначені у плані, так як ми вже зробили план та вивчили його ми можемо сміливо виконувати terragrunt apply з ключем auto-aprove. Після закінчення виконання команди terragrunt apply йдемо у консоль AWS та переконуємося, що всі сервіси були задеплоєні корректно. Далі йдемо у Secrets Manager беремо kubeconfig та інші необхідні секретні значення. З цим kubeconfig ми будемо під'єднуватись до кластера. Для зручності будемо використовувати IDE для Kubernetes — Lens. Додаємо наш кубконфіг до Lens, та перевіряємо сервіси, які ми описували та задеплоїли через Terraform. За допомогою утиліти kubectl перевіряємо деплойменти в усіх неймспейсах — kubectl get deployments -n A, де -n ключ для визначення неймспейса, робимо теж саме для сервісів, конфігів, секретів, інгресів та волумів:

- kubectl get services -n A;
- kubectl get configmaps -n A;
- kuebctl get secrets -n A;
- kubectl get ingresses -n A;
- kubectl get pvc -n A.

На рисунку 4.* ми бачимо навантаження у нашому кластері. Prometheus оператор збирає метрики з кожного пода.



5 ДЕПЛОЙ HELM ЧАРТІВ

При роботі з Helm є одне важливе правило: всі дії, що модифікують кластер, повинні бути скоєні через Helm, а не безпосередньо за допомогою kubectl. Helm при кожному оновленні кластера створює секрет, де описує зміни. Якщо міняти кластер через kubectl, то Helm не побачить змін, і це може призвести до помилок.

Перш за все нам потрібно зібрати докер образ API. Для цього використовуємо наступну команду - docker build ${BUILD_PATH}/api -t "${REGISTRY}/api:${TAG}" -t "${REGISTRY}/api:${CACHE_TAG}" --cache-from "${REGISTRY}/api:${CACHE_TAG}". Після чого авторизуємось у AWS ECR - eval $(aws ecr get-login --region "${AWS_DEFAULT_REGION}" —no-include-email) та пушимо туди наш образ з нашими тегами - docker push "${REGISTRY}/api:${TAG}" && docker push "${REGISTRY}/api:${CACHE_TAG}"

Далі встановлюємо зовнішні чарти, які ми описали, для цього використовуємо команду helm dep update, після виконання ми побачимо нову директорію charts/ де буде лежати їх вихідний код. Далі перевіряємо валідність нашого коду і усих Kubernetes маніфестів за допомогою команди helm template -

f values.yaml. Продивляємось сгенеровані маніфести, після того, як ми впевнені в змінах можемо приступати до встановлення чарта у кластер.

Для простоти замість двох команд (helm install, helm ugrade) можна виконати команду upgrade з додатковим ключем --install. При першому виконанні Helm відправить команду на встановлення релізу, а надалі його оновлюватиме - helm upgrade "${RELEASE_NAME}" . -f "${ENV_VALUE_FILE}" --namespace "${NAMESPACE}" --install --render-subchart-notes --cleanup-on-fail --timeout 10m --set global.image.tag="${TAG}" --set envSecretMaps.api-db.API_DB_PASSWORD="${API_DB_PASSWORD}". Таким чином ми встановлюємо наш чарт прокидуючи змінні для нашої API, docker image tag та пароль до бази даних.

Параметри команди для деплоя:

- --install - якщо helm реліз з такою назвою ще не існує, запускає установку;
- --render-subchart-notes - якщо встановлено, відображати нотатки піддіаграми;
- --atomic - якщо встановлено, процес оновлення скасовує зміни, внесені в разі невдалого оновлення. Ключ --wait буде встановлено автоматично, якщо використовується — atomic;
- --timeout - час очікування будь-якої окремої операції Kubernetes (наприклад, Jobs для хуків) (за замовчуванням 5m0s);
- --set parameters - встановити значення в командному рядку (можна вказати кілька або окремі значення з комами: key1=val1,key2=val2);

```
helm upgrade "${RELEASE_NAME}" . \
  -f "${ENV_VALUE_FILE}" \
  --namespace "${NAMESPACE}" \
  --install \
  --render-subchart-notes \
  --atomic \
  --timeout 10m \
  ${EXTRA_HELM_ARGS}
```

Далі йдемо в Lens та перевіряємо наявність наших чартів. Для цього використовуємо команду helm list -n ${NAMEPSACE}.

Ця команда перераховує всі випуски для вказаного простору імен (використовує поточний контекст простору імен, якщо простір імен не вказано).

За замовчуванням у ньому перелічено лише релізи, які були розгорнуті або вийшли з ладу. Такі прапори, як «--uninstalled» і «--all», змінять цю поведінку. Такі прапорці можна комбінувати: '--uninstalled --failed'.

За замовчуванням елементи сортуються в алфавітному порядку. Використовуйте прапорець '-d' для сортування за датою випуску.

Якщо надано прапор --filter, він буде розглядатися як фільтр. Фільтри — це регулярні вирази, які застосовуються до списку випусків. Будуть повернені лише елементи, які відповідають фільтру.

Наступним кроком ми переконуємось, що усі компоненти з чарту успішно розгорнулися. Проходимось по усім компонентам використовуючи команду kubectl — kubectl get ${COMPONENT} -n ${NAMEPSACE}. Також дивимось, що всі SSL сертифікати були випущені без помилок, і базово перевіряємо доступність API за допомогою telnet - telnet ${API_HOST} ${API_PORT}

Коли ми впевнились, що порт відповідає коректно, ми можемо вважати, що кластер з нашим API працює справно. Для відладки можемо подивитись логи API за допомогою команди kubectl logs ${API} -n ${NAMESPACE}.

ВИСНОВОК

Таким чином можно зробити висновок про те, що було спроектовано та створено відмовостійкий, маштабований Kubernetes кластер у хмарі AWS за допомогою Terraform коду, який можна легко підтримувати, перевикористовувати, тому що все описано як код. Даний кластер має змогу динамічно використовувати свої ресурси, міняти їх в разі потреби.

СПИСОК ЛІТЕРАТУРИ

1. Офіційна документація Kubernetes - https://kubernetes.io/docs/home/

2. Офіційна документація Helm - https://helm.sh/docs/

3. O'Reilly, Sbastien Goasguen - Kubernetes Cookbook: Building Cloud Native Applications

4. Офіційна документація Amazon Web Services - https://docs.aws.amazon.com/

5. O'Reilly, Mike Zazon - Cookbook: Recipes for Success on AWS