

데이터 분산처리 끝내기

- Part6 Kubernetes

이 수업에서 다루고자 하는 것

이론

Chapter 1. Kubernetes란?

- Kubernetes 소개
- Kubernetes를 위한 Docker

Chapter 3. Kubernetes 기타 개념

- Helm Chart
- Yaml

Chapter 2. Kubernetes의 기본 개념과 구성

- 클러스터
- 워크로드 – Pod
- 워크로드 – Deployment
- 워크로드 – ReplicaSet, StatefulSet, DaemonSet
- Service
- Storage

Chapter 1. Kubernetes란?

Chapter 1. Kubernetes란?

01. Kubernetes 소개

01. Kubernetes 소개

Kubernetes

- Kubernetes는 컨테이너화된 애플리케이션을 어디서나 배포, 확장, 관리할 수 있는 오픈소스
- Kubernetes는 영어로 'captain(선장)'을 의미하는 그리스어
영문표기는 Helmsman(Kubernetes 클러스터 Package Manager의 이름이 Helm)
- 'K'와 'S' 사이의 문자 수를 나타내는 8을 사용하여 K8S로 줄여 쓰기도 함
- 비슷한 플랫폼으로 Docker Swarm, Hashicorp Nomad
Mesosphere Marathon, Amazon ECS 등이 있음
- 컨테이너를 도입하는 조직이 점점 많아지고 있는 상황
- 컨테이너 관리 소프트웨어인 Kubernetes를 통해 컨테이너화된 애플리케이션을
배포하고 운영하기 위한 표준



01. Kubernetes 소개

Kubernetes

- Kubernetes는 컨테이너 관리의 운영 작업을 자동화
- 컨테이너화된 애플리케이션을 쉽게 관리할 수 있음
- 서비스를 지속적으로 체크, 문제가 있는 컨테이너를 재시작
서비스가 정상일 때만 사용자가 서비스를 사용
- Google에서 개발되어 2014년에 오픈소스로 출시



01. Kubernetes 소개

Kubernetes 개발 이유

- Google은 전 세계적으로 수십만대의 서버를 운영 중
- 이 서버들을 효과적으로 운영 관리할 수 있는 방법?
- ‘Borg’가 탄생함
- 구글의 Borg 시스템은 2015년 논문으로 공개
"Large-scale cluster management at Google with Borg"
- 곧 쿠버네티스라는 이름으로 공개되었고 CNCF재단에 기증됨



01. Kubernetes 소개

Kubernetes 역사

- Borg가 1세대로 개발됨
- Borg – Omega – Kubernetes
- Docker가 사용되기 이전에 구글에서는 컨테이너 기술을 사용
- Borg System은 처음에 5 명 미만이 운영하는 소규모 프로젝트
- 2013년에는 대규모 컴퓨팅 클러스터를 위해 Omega 클러스터 관리 시스템 도입
- 2014년 Kubernetes를 Borg의 오픈 소스 버전으로 출시, 공개
- Docker, IBM, Microsoft 및 RedHat 등 기술 회사는 모두 Kubernetes 커뮤니티에 합류

01. Kubernetes 소개

Kubernetes 역사

- 2015년 Google과 Linux는 CNCF(Cloud Native Computing Foundation)를 구성
- 2017년 안정적인 버전이 출시됨
API 집계, 로컬 저장소, 확장성, 암호화 및 기타 타사 리소스를 지원
CNCF는 또한 시스템 표준화를 목표로 한 최초의 Kubernetes 인증 공급자를 공개함
- 2018년에는 Amazon EKS 및 AKS (Azure Kubernetes Service) 등 주요 클라우드 기반
공급자가 다양한 엔진을 출시

01. Kubernetes 소개

Kubernetes 용도

- 클라우드 기반의 마이크로서비스를 기반으로 하는 앱을 빌드하는 데 유용
- 기존 앱의 컨테이너화도 지원
- 애플리케이션을 현대화하여 앱을 더 빠르게 개발할 수 있음
- 쿠버네티스는 어디서나 사용할 수 있도록 구축
필요한 곳에서 애플리케이션을 실행
- Kubernetes는 서비스 실행에 필요한 클러스터의 크기를 자동으로 조정

Chapter 1. Kubernetes란?

02. Kubernetes를 위한 Docker

02. Kubernetes를 위한 Docker

Docker?

- Kubernetes를 사용하기 위해서는 컨테이너에 대한 이해가 필요
- 컨테이너 기술의 대표적인 예가 바로 Docker
- Kubernetes와 Docker는 둘 중 하나만 선택해야 한다는 오해가 있었음
- 컨테이너화된 애플리케이션을 실행하기 위한 서로 다르면서 보완적인 기술
- Docker를 사용하면 애플리케이션 실행을 위해 필요한 파일이나 설정 등을 컨테이너에 저장이 가능
- 애플리케이션을 컨테이너에 넣은 후 Kubernetes를 이용해 애플리케이션을 관리

02. Kubernetes를 위한 Docker

Why Container?

- “Local PC에서는 잘 됐는데… 서버에 올리니까 잘 안되네?”
- 소프트웨어의 실행은 OS와 라이브러리에 의존성을 갖게 됨
- 동일한 Library를 사용하지만 서로 다른 버전을 필요로 하는 경우
- software의 운영 체제가 다를 경우 (100개의 software일 경우 100개의 시스템이 필요)
- 컨테이너를 통해 소프트웨어 실행에 필요한 환경을 독립적으로 운용할 수 있도록 만듬
- 컨테이너에는 다른 실행환경과는 차단된 환경, OS수준의 격리 기술이 사용됨

02. Kubernetes를 위한 Docker

Docker?

- 도커는 컨테이너 기반의 오픈소스 가상화 플랫폼
- 다양한 프로그램, 실행환경을 컨테이너로 추상화
- 동일한 인터페이스를 제공하여 프로그램의 배포 및 관리를 간단하게 만듬
- 컨테이너는 격리된 공간에서 프로세스가 동작하는 기술
- 일종의 가상화 기술
- 기존의 가상화 기술과는 차이가 있음



02. Kubernetes를 위한 Docker

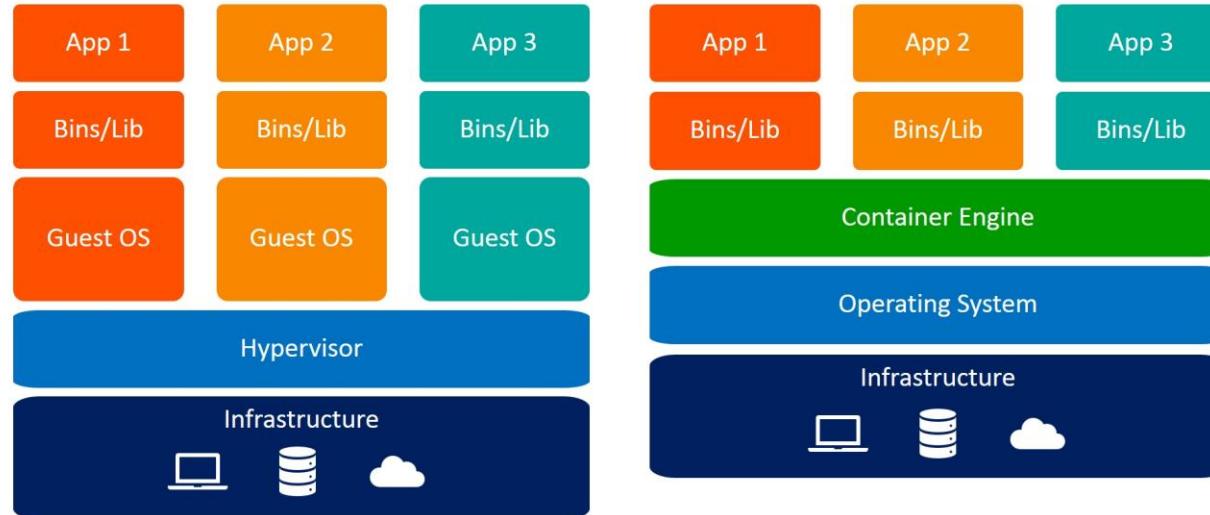
Docker?

- 기존의 가상화 기술은 OS를 가상화
- VMWare, VirtualBox 등은 호스트 OS위에 게스트 OS를 가상화하는 방식
너무 무겁고 느린 문제
- 도커 컨테이너는 프로세스 격리를 활용
- 도커 이전에 Linux(LXC, Linux Container)에서는
cgroups, namespace를 활용해 프로세스를 격리했음



02. Kubernetes를 위한 Docker

Docker?



Virtual Machines

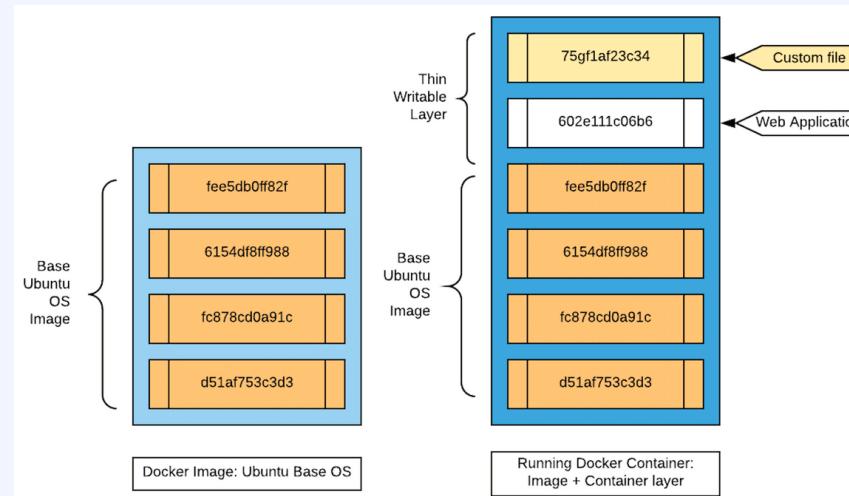
Containers

<https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms>

02. Kubernetes를 위한 Docker

Docker Image?

- 이미지에는 컨테이너 실행에 필요한 파일, 설정 값을 저장해 놓음
- 컨테이너는 이미지가 실행 된 상태, 이미지는 변하지 않음(Immutable)
- 레이어 개념을 활용해 여러 이미지들을 하나씩 쌓을 수 있음



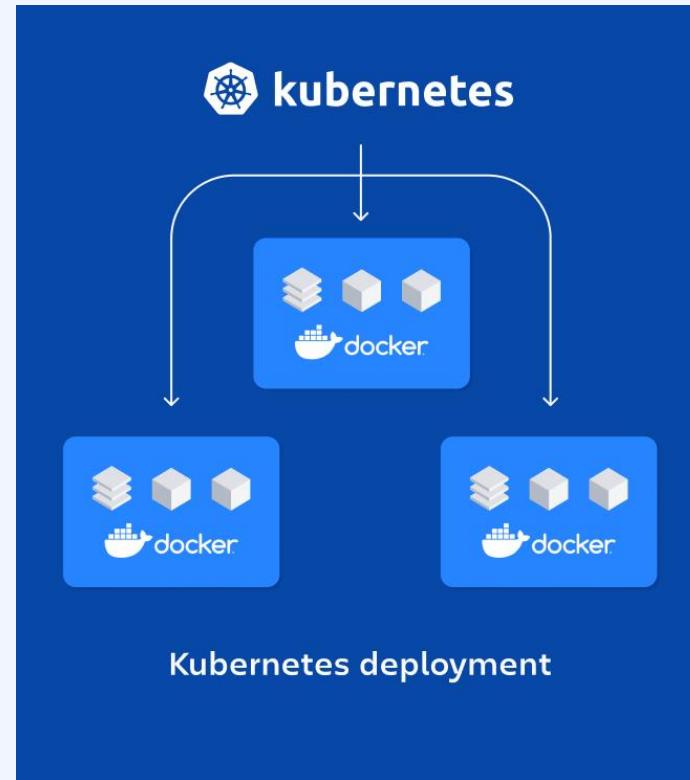
02. Kubernetes를 위한 Docker

Docker Image?

- 이미지 레이어를 구성해놓고 어딘가에(AWS ECR, Harbor, Docker Hub) 업로드
- 이미지를 받아 어디서든 원하는 컨테이너를 구성할 수 있음
- Docker를 사용하면 어디서나 안정적으로 실행할 수 있는 단일 객체를 확보
- 컨테이너와 이미지를 만들고 Kubernetes를 통해 애플리케이션을 관리
- 내부 컨테이너는 항상 같은 이미지를 활용해 동일한 환경이 구성됨
- 그 외의 health 체크, 부하분산, 상태 관리, 구동환경 관리 등은 Kubernetes에서
- Kubernetes를 통한 컨테이너 오케스트레이션

02. Kubernetes를 위한 Docker

Kubernetes, Docker



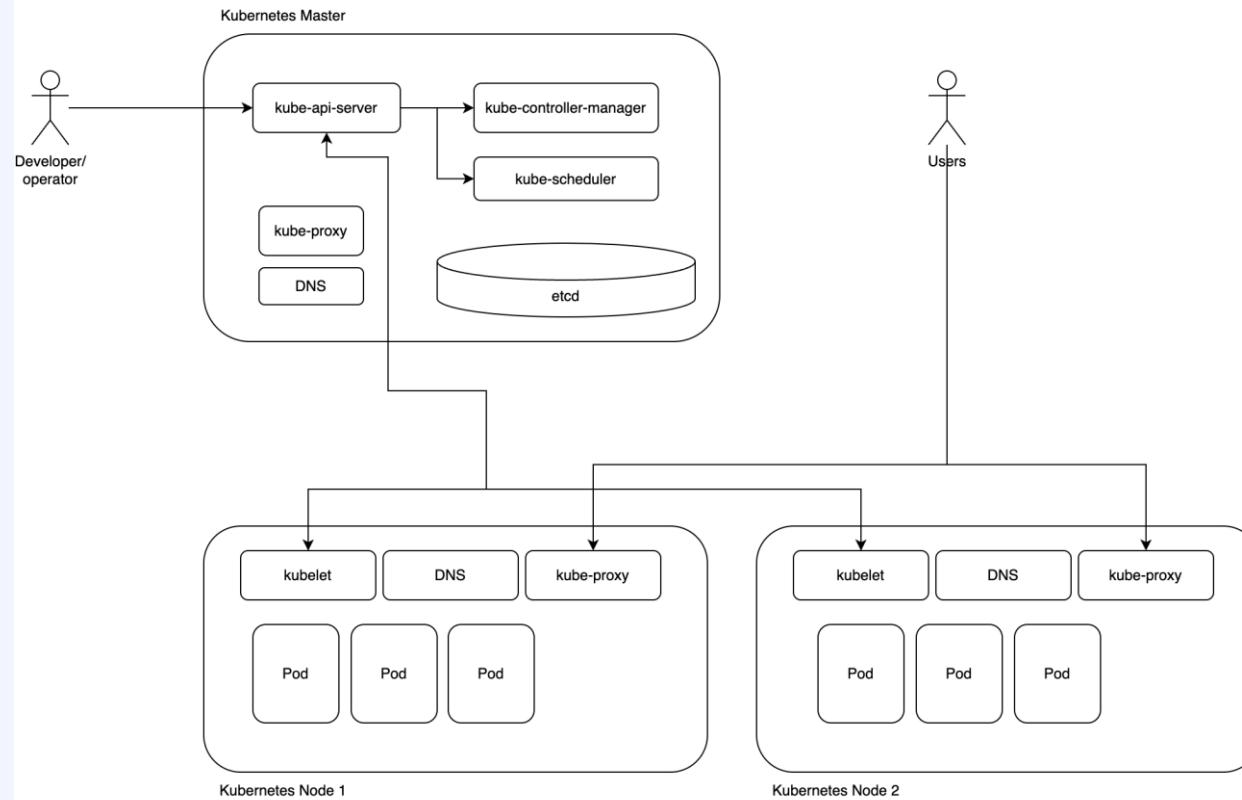
<https://www.atlassian.com/ko/microservices/microservices-architecture/kubernetes-vs-docker>

Chapter 2. Kubernetes의 기본 개념과 구성

Chapter 2. Kubernetes의 기본 개념과 구성

01. 클러스터

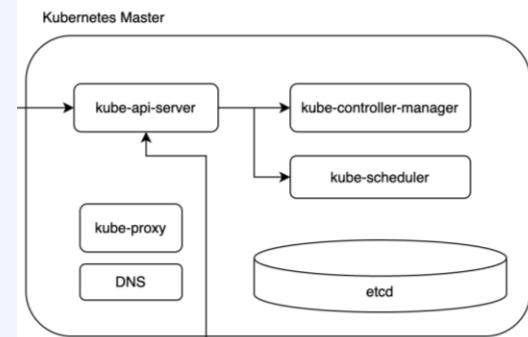
01. 클러스터



01. 클러스터

Kubernetes Master Node

- Node : 클러스터 내 가상 혹은 물리 서버를 의미, 컴퓨팅 엔진
- API Server : 쿠버네티스 API를 노출하는 REST 컴포넌트
k8s와 interact하기 위한 kubectl 같은 인터페이스는 API를 통하여 됨
 - 1.API 관리 - api를 노출하고 관리, 6443포트 오픈 16443(haproxy)
 - 2.요청 처리 - 클라이언트의 개별 api 요청 처리, http또는 json
 - 3.내부 제어 - api 실행에 필요한 백그라운드 작업 수행
- kube-scheduler : Pod를 어떤 노드에 배치할지 결정,
실제 배치는 Kubelet이 함
노드 선택 기준: 1. Pod가 요구하는 컴퓨팅 자원으로 필터링 하고
2. Pod가 배치된 이후 노드에 남는 컴퓨팅 자원의 양을 확인



01. 클러스터

ETCD

- kubectl get pod와 같은 command들은 모두 etcd로부터 받는 정보
- 노드를 추가하거나 pod를 배포하는 등의 부분들도 etcd에 업데이트 됨
- nodes, pods, configs, secrets, accounts, roles, bindings에 대한 정보가 저장됨
- 위 정보들로 개별 서버에 장애가 발생하더라도 작동 상태를 유지할 수 있게 됨
etcd에는 높은 신뢰성이 꼭 필요
- etcd는 고가용성을 위하여 클러스터로 설치 권고
- 여러 노드의 통신은 래프트(Raft) 알고리즘에 의해 처리
- Raft 알고리즘 (Distributed Consensus)
분산 시스템에서 특히 etcd에서 합의를 도출하는 프로토콜
<http://thesecretlivesofdata.com/raft/>

01. 클러스터

Kubernetes Master Node

- kube-controller-manager : 컨트롤러를 구동하고 관리, k8s의 뇌 역할을 맡음
 - Node Controller: 노드가 다운되었을 때 통지 / 대응
 - Replication Controller : 시스템의 모든 Replication Controller Object에 대하여 알맞은 수의 Pod들을 유지
 - Endpoint Controller: Service와 Pod을 연결 -> Endpoint object를 채워 서비스와 Pod를 연결
 - Service Account & Token Controller: 새로운 네임스페이스에 대한 기본 계정과 API 접근 토큰을 생성
- 마스터에는 일반 파드들이 스케줄링 되지 않음, 스케줄링 하지 않도록 설정이 되어 있음
- **api-server, etcd, scheduler, controller-managner**를 묶어 Control Plane이라고 부름

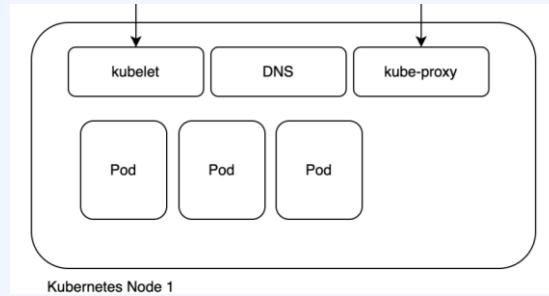
01. 클러스터

Kubernetes Worker Node

- 워커 노드에서 스케줄링 되는 파드들이 생성됨
- Control Plane이 이 스케줄링을 담당

노드 컴포넌트

- kubelet : 각 노드의 agent, 파드 안 컨테이너가 동작하도록 관리
API서버를 모니터링, 스케줄링 되는 파드를 감지, 파드 내 컨테이너 실행
- kube-proxy : 각 노드의 네트워크 프록시이며, 서비스의 구현체
노드의 네트워크 규칙을 관리함
설정된 네트워크 규칙을 통해 클러스터 내부/외부와 통신이 가능함
- kube-dns : k8s 클러스터 내 pod에서 특정 도메인을 찾고자 할 때 사용됨



01. 클러스터

Kubernetes Worker Node

노드 컴포넌트

- Container runtime engine : 컨테이너 이미지를 Pull하고 실행하는 엔진
쿠버네티스 1.23버전 기준으로 지원되는 런타임은 아래와 같음
containerd : 쿠버네티스 전용 컨테이너 런타임, 도커엔진의 내부 엔진은 containerd
- CRI-O : Redhat 개발, Kubernetes 용 Open Container Initiative (OCI) 컨테이너 런타임
- Docker Engine : 1.24버전 부터 제거됨
- Mirantis Container Runtime : Dockershim은 deprecated 되지만 Mirantis에서 지원됨

*K8S는 CRI(container runtime interface)를 이용, 컨테이너 런타임과 통신함

도커는 해당 인터페이스를 지원하지 않아 Dockershim이라는 추가 레이어를 통해 연동하고 있었음
이 Dockershim이 deprecated될 예정

Chapter 2. Kubernetes의 기본 개념과 구성

02. 워크로드 - Pod

02. 워크로드 - Pod

Kubernetes Workload

- 쿠버네티스 내에서 구동되는 애플리케이션
- 워크로드는 파드 집합 내에서 실행됨
- Pod는 클러스터 내에서 실행 중인 컨테이너의 집합을 나타냄
- Pod가 실행되고 난 다음 Node에 심각한 장애나 오류가 발생하면, 모든 Pod는 Fail
- Pod이 다시 생성될 때 직접 관리할 필요가 없도록 워크로드 리소스를 활용
- 컨트롤러를 통해 지정된 상태와 같이 올바른 수인지, 올바른 파드 유형이 실행되는지를 확인

02. 워크로드 - Pod

Kubernetes Workload

Pod

- Pod는 쿠버네티스에서 생성, 관리할 수 있는 가장 작은 컴퓨팅 단위
- pod of whale은 고래 떼를 의미, 이 의미와 마찬가지로 Pod는 하나 이상의 컨테이너의 그룹
- 한 Pod는 스토리지, 네트워크를 공유
- Pod안의 컨테이너를 구동하는 것에 대한 명세를 갖고 있음
- 이 명세를 통해 더 많은 인스턴스를 실행할 수도 있다(더 많은 리소스를 제공해야 하는 경우)
=> Replication(Replicas) 조정
- 명세를 갖기 때문에 Pod를 kill 해도 다시 명세대로 생성되어 복구된다

02. 워크로드 - Pod

Kubernetes Workload

Pod

- nginx:1.14.1 이미지를 받아 컨테이너로 구동하는 파드

```
# simple-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.1
    ports:
      - containerPort: 80
```

- **apiVersion** : api의 버전 명세를 나타냄
/를 통해 depth를 깊게 가져갈 수 있음
- **kind** : 리소스 타입
- **metadata** : 생성할 파드의 이름, label 등을 정함
- **spec** : 구체적인 사양을 작성
다양한 사양이 있지만 container에 대한 사양을 주로 작성
image명세를 통해 public, 또는 private 레포지토리에서
이미지를 가져올 수 있음

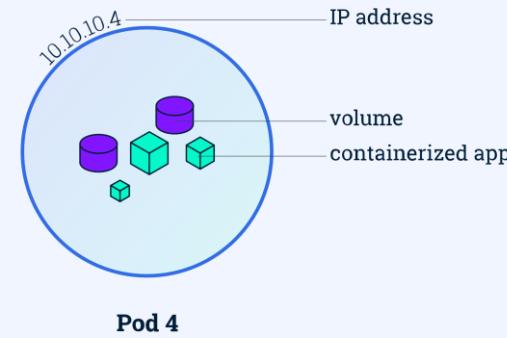
02. 워크로드 - Pod

Kubernetes Workload

Pod

- nginx:1.14.1 이미지를 받아 컨테이너로 구동하는 파드

```
# simple-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.1
      ports:
        - containerPort: 80
```



<https://kubernetes.io/ko/docs/tutorials/kubernetes-basics/explore/explore-intro/>

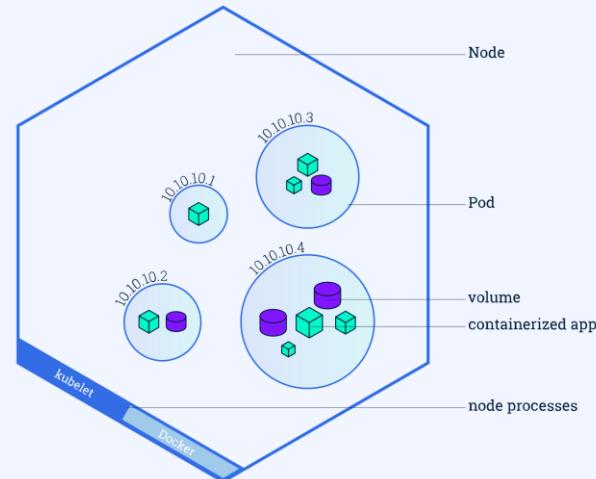
02. 워크로드 - Pod

Kubernetes Workload

Pod

- nginx:1.14.1 이미지를 받아 컨테이너로 구동하는 파드

```
# simple-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.1
      ports:
        - containerPort: 80
```



<https://kubernetes.io/ko/docs/tutorials/kubernetes-basics/explore/explore-intro/>

02. 워크로드 - Pod

Kubernetes Workload

Pod

- Pod는 대부분 직접 만드는 경우가 거의 없음
 - Deployment, Job 등의 워크로드 리소스를 통해서 생성되는게 대부분
 - 파드의 상태를 추적하고 관찰해야 한다면 StatefulSet을 사용
 - 파드는 하나의 컨테이너를 실행할 수도 있고 여러 컨테이너를 실행할 수도 있다
- *사이드카 패턴

Chapter 2. Kubernetes의 기본 개념과 구성

03. 워크로드 - Deployment

03. 워크로드 - Deployment

Kubernetes Workload Resources

Deployment

- Pod와 Replicaset에 대한 명세가 작성되어 있음
- ‘애플리케이션의 인스턴스를 어떻게 생성하고 업데이트할 것인가’에 대한 설명서
- 쿠버네티스의 초기에는 Replication Controller에서 앱이 배포됨
- 현재는 Deployment를 앱 배포시에 기본으로 사용
- 원하는 상태에 대해서 Deployment를 작성하고 Controller를 통해 조정함
컨트롤 플레인이 디플로이먼트로 정의된 애플리케이션이 노드에서 실행되도록 스케줄링

03. 워크로드 - Deployment

Kubernetes Workload Resources

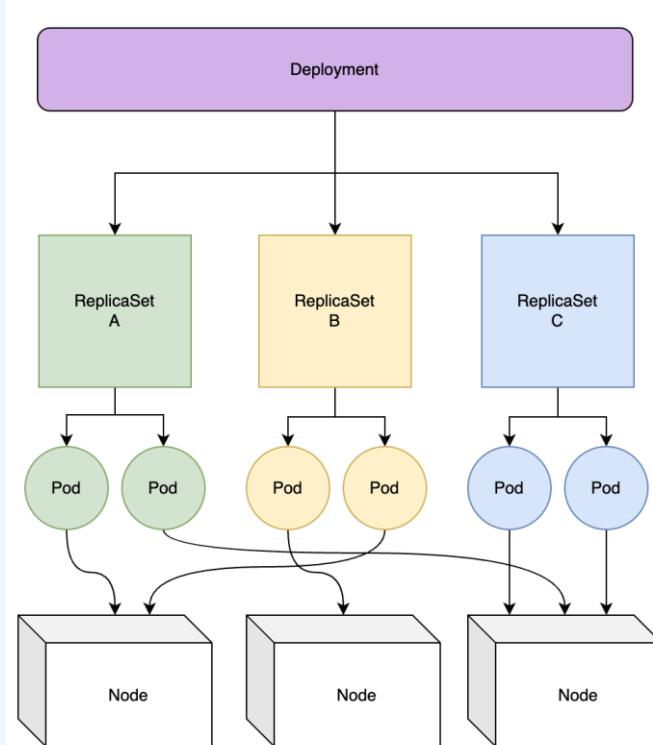
Deployment

- 인스턴스 생성 이후, 디플로이먼트 컨트롤러는 이들을 모니터링
self-healing 제공
- 배포 이후에 변경이 없는 애플리케이션 관리에 적절함
- 배포 기능을 세분화 한 것
- Pod, Replicaset에 버전 관리 기능을 추가한 것과 비슷함
파드 개수 유지, 롤링 업데이트, 배포 중 중지 후 배포, 롤백 가능

03. 워크로드 - Deployment

Kubernetes Workload Resources

Deployment



03. 워크로드 - Deployment

Kubernetes Workload Resources

Deployment 업데이트 방식

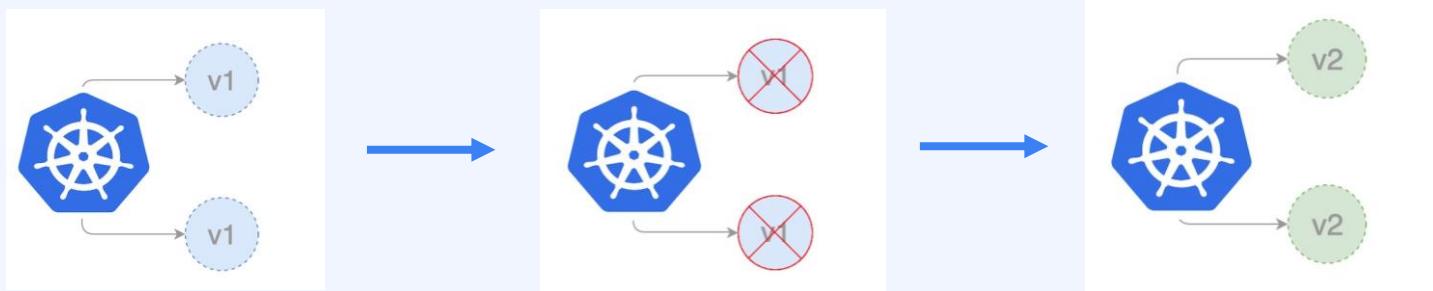
- ReCreate
- Rolling Update
- Blue/Green
- Canary

03. 워크로드 - Deployment

Kubernetes Workload Resources

Deployment 업데이트 방식

- ReCreate : 기존 파드 모두 삭제 후 새 파드를 생성하는 방식(다운타임 발생 필연적)



<https://auth0.com/blog/deployment-strategies-in-kubernetes/>

03. 워크로드 - Deployment

Kubernetes Workload Resources

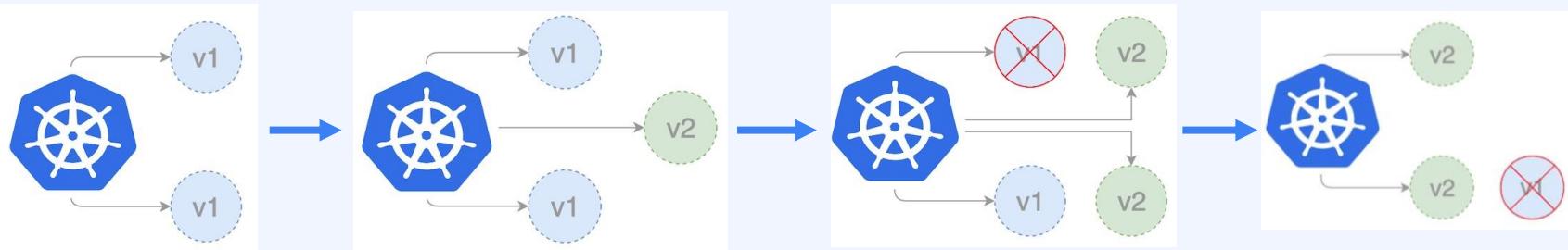
Deployment 업데이트 방식

- Rolling Update : 새 버전 배포와 동시에 기존 버전의 파드를 줄여나가는 방식
무중단 배포, 이전 버전과 새 버전이 공존하는 시간이 있다는 단점
- maxUnavailable : 롤링 업데이트 중, 사용할 수 없는 최대 파드의 개수(기본 25%)
롤링 중 동시에 삭제할 수 있는 파드의 최대 개수
ex) 2개라고 지정했다면 롤링 업데이트 시작하면서 2개가 동시에 삭제 됨
- maxSurge : 생성할 수 있는 최대 파드의 수(기본 25%)
롤링 중 동시에 생성하는 파드의 개수
ex) 2개라고 지정했다면 롤링 업데이트 시작할 때 새 파드가 2개 추가 됨
높게 설정할 수록 교체 시간이 줄어듬, 하지만 리소스가 많이 필요

03. 워크로드 - Deployment

Kubernetes Workload Resources

Rolling Update



<https://auth0.com/blog/deployment-strategies-in-kubernetes/>

03. 워크로드 - Deployment

Kubernetes Workload Resources

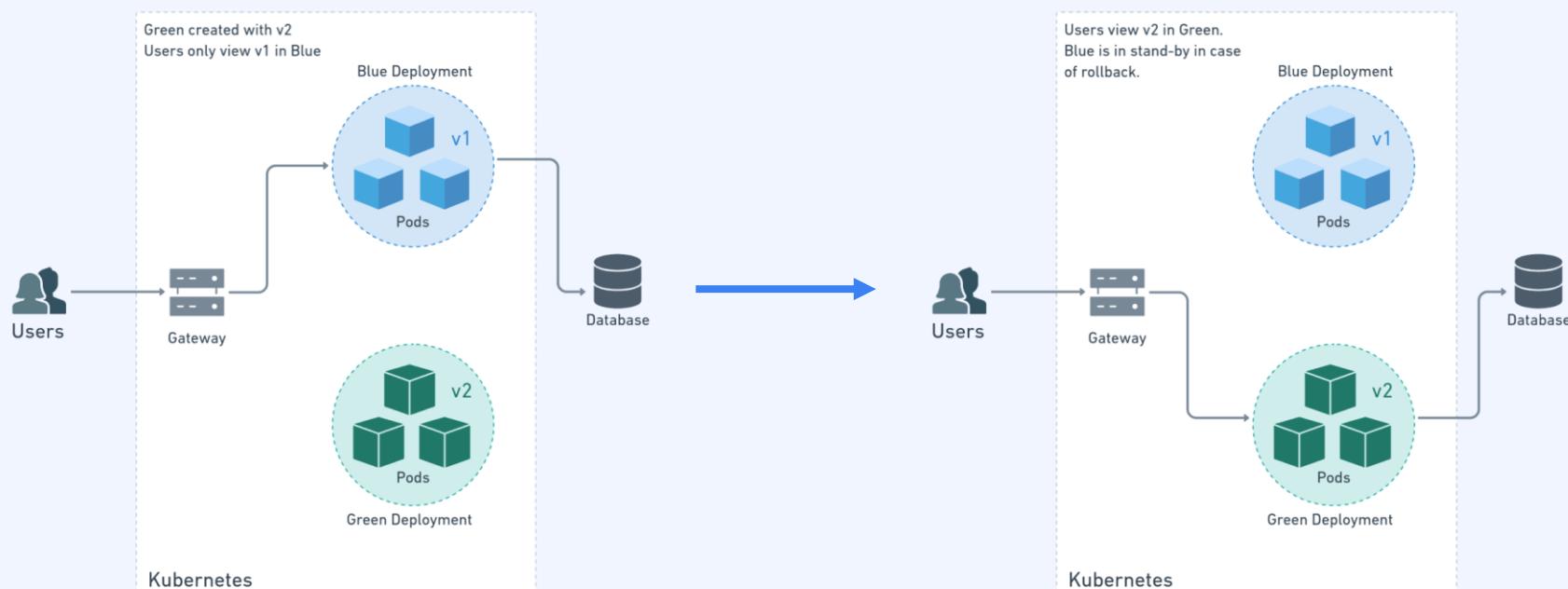
Deployment 업데이트 방식

- Blue/Green : Old, New 버전을 서버에 두고 한꺼번에 교체하는 방식
- 무중단 방식, 롤백이 용이함
- RollingUpdate의 단점을 해결
- 리소스를 2배로 사용한다는 단점이 존재

03. 워크로드 - Deployment

Kubernetes Workload Resources

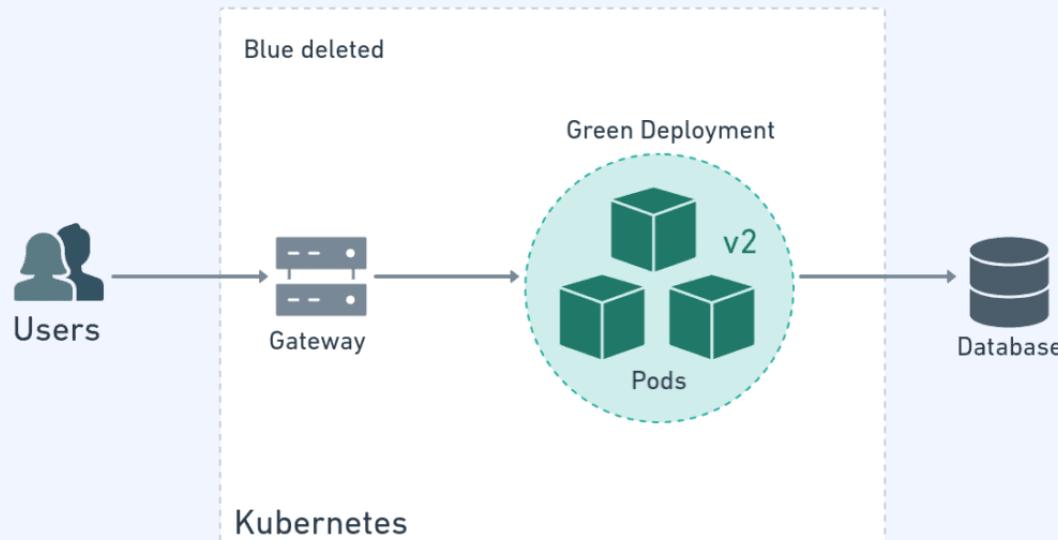
Blue/Green



03. 워크로드 - Deployment

Kubernetes Workload Resources

Blue/Green



03. 워크로드 - Deployment

Kubernetes Workload Resources

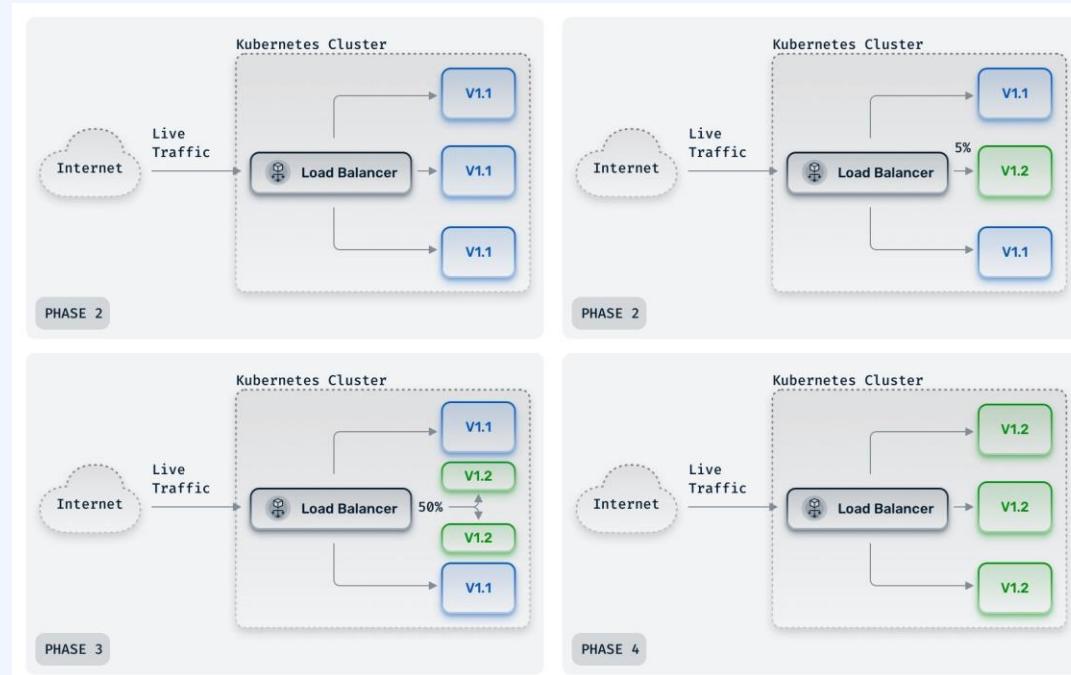
Deployment 업데이트 방식

- Canary : Old, New 버전을 구성하고 Old의 일부 트래픽을 New 버전으로 분산
일부에 대해 테스트를 진행 후 점점 New 버전으로 옮기는 방식
- 배포에 따른 위험감지를 사전에 하기 위한 전략
- 모니터링에 유용함

03. 워크로드 - Deployment

Kubernetes Workload Resources

Canary



<https://traefik.io/glossary/kubernetes-deployment-strategies-blue-green-canary/>

03. 워크로드 - Deployment

Kubernetes Workload Resources

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

- replicas : 클러스터 안에서 실행 할 파드 수
- selector : LabelSelector로 파드의 Template에 명시된 라벨과 일치해야 생성 “app:nginx”
- template : PodTemplateSpec, replicas에 명시된 파드 수와 일치하지 않으면 새로 작성되는 파드의 템플릿
- template/metadata : 템플릿의 이름, Label 등
- template/spec : 파드의 상세 정보

03. 워크로드 - Deployment

Kubernetes Workload Resources

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
  labels:
    app: myapp
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  minReadySeconds: 20
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - image: nginx:1.14.2
          name: myapp
          ports:
            - containerPort: 80
```

- strategy : Deployment 업데이트 방식
- maxUnavailable : 롤링 업데이트 중, 사용할 수 없는 최대 파드의 개수
- maxSurge : 생성할 수 있는 최대 파드의 수
- minReadySeconds : 새로 배포된 컨테이너가 준비되기까지 대기할 시간(기본 0초)

Chapter 2. Kubernetes의 기본 개념과 구성

04. 워크로드 - ReplicaSet, StatefulSet, DaemonSet

04. 워크로드 - ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

ReplicaSet

- 파드 집합의 실행을 안정적으로 유지하는 것이 목적
- 명시된 일정 파드의 수의 가용성을 보증하는데 사용됨
- 특정 사유로 노드에 파드를 실행할 수 없다면, 다른 노드에서 파드를 다시 생성하도록 한다
- 이 리소스를 감시하는 컨트롤러가 파드의 변화를 감지
파드가 죽었다면 다시 복구
- 보통 Deployment를 많이 사용

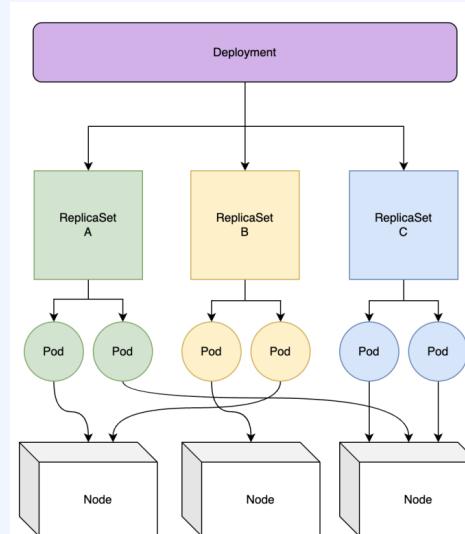
04. 워크로드 - ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

ReplicaSet

- 디플로이먼트와의 차이?

디플로이먼트는 레플리카셋을 관리하고 다른 기능을 갖고 있는 상위 개념



04. 워크로드 - ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp
  labels:
    app: nginx
spec:
  # 케이스에 따라 레플리카를 수정한다.
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
        - image: nginx:1.14.2
          name: myapp
          ports:
            - containerPort: 80
```

- replicas : 클러스터 안에서 실행 할 파드 수
- selector : LabelSelector로 파드의 Template에 명시된 라벨과 일치해야 생성 “app:nginx”
- template : PodTemplateSpec, replicas에 명시된 파드 수와 일치하지 않으면 새로 작성되는 파드의 템플릿
- template.metadata : 템플릿의 이름, Label 등
- template.spec : 파드의 상세 정보

04. 워크로드 - ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

StatefulSet

- 애플리케이션이 Stateful한지 관리하는데 사용됨
 - 파드의 디플로이먼트, 스케일링을 관리
 - 파드의 순서와 고유성 보장
 - 파드가 삭제되고 생성되면 새로운 가상환경이 실행됨
- 파드 상태를 유지하고 싶다면 StatefulSet을 사용

04. 워크로드 – ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

StatefulSet 사용목적

- 안정적인 네트워크 식별자
- 안정적, 지속적인 스토리지 (PV 유지)
- 순서가 보장된 파드의 배치와 확장
- 파드의 오토 롤링 업데이트

04. 워크로드 – ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

StatefulSet 주의할 점

- 관련 볼륨이 사라지지 않아 관리가 필요함
- 스토리지는 PV, StorageClass로 프로비저닝 필요
- 롤링업데이트 하는 경우 수동 복구가 필요할 수 있음(기존 스토리지와 충돌)
- Headless 서비스 필요

04. 워크로드 – ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

StatefulSet Deployment와 차이점

- Deployment는 Service를 통해서 외부에 노출되는 반면,
StatefulSet은 Headless Service를 사용, 서비스에 리퀘스트 불가
- Deployment는 내부에 ReplicaSet을 생성해 파드 관리, Rollback 가능
StatefulSet은 없음, Rollback 불가

04. 워크로드 - ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

StatefulSet

```
apiVersion: v1          apiVersion: apps/v1    serviceName: "nginx"
kind: Service           kind: StatefulSet   replicas: 3 # 기본값은 1
metadata:               metadata:          minReadySeconds: 10 # 기본값은 0
  name: nginx          name: web           template:
  labels:               spec:              metadata:
    app: nginx          selector:         labels:
                                matchLabels:   app: nginx
                                app:           spec:
spec:                   ...                terminationGracePeriodSeconds: 10
ports:                  ...                containers:
  - port: 80            ...                - name: nginx
    name: web          image: registry.k8s.io/nginx-slim:0.8
clusterIP: None          ports:             - containerPort: 80
selector:               ...                name: web
  app: nginx           volumeMounts:
                                - name: www
                                  mountPath: /usr/share/nginx/html
```

- kind : Service, StatefulSet
- replicas : 클러스터 안에서 실행 할 파드 수
- selector : LabelSelector로 파드의 Template에 명시된 라벨과 일치해야 생성 “app:nginx”
- template : PodTemplateSpec, replicas에 명시된 파드 수와 일치하지 않으면 새로 작성되는 파드의 템플릿
- template.metadata : 템플릿의 이름, Label 등
- template.spec : 파드의 상세 정보

04. 워크로드 – ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

DaemonSet

- 특정 노드 또는 모든 노드에 항상 실행되어야 하는 파드를 관리

사용 예

- 모든 노드에서 클러스터 스토리지 데몬 실행
- 모든 노드에서 로그 수집 데몬 실행
- 모든 노드에서 노드 모니터링 데몬 실행
- GPU 노드에 nvidia 플러그인 설치

04. 워크로드 – ReplicaSet, StatefulSet, DaemonSet

Kubernetes Workload Resources

DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-daemon
spec:
  selector:
    matchLabels:
      name: fluentd-daemon
  template:
    metadata:
      labels:
        name: fluentd-daemon
    spec:
      containers:
        - image: fluent/fluentd
          name: fluentd-daemon
```

- kind : DaemonSet
- metadata.name : DaemonSet 이름
- template.metadata : 템플릿의 이름, Label 등
- template.spec : 파드의 상세 정보
- fluentd-daemon : 로그 수집기의 역할

Chapter 2. Kubernetes의 기본 개념과 구성

05. Service

05. Service

Service 필요성

- 파드들은 반영속적(ephemeral)하기 때문에 서비스가 필요하게 됨
- 파드는 언제나 삭제될 수 있고 다른 노드로 옮겨질 수 있음
- 파드는 생성 시 내부 IP를 할당받게 되지만 클러스터 내/외부 통신이 자유롭진 않다
- 고정된 엔드포인트로 호출이 어려워짐
- 파드 간의 로드 밸런싱이 필요해지게 되는데 서비스가 이 역할을 해주는 것

05. Service

Service

- 서비스는 파드를 통해 실행되는 애플리케이션을 네트워크에 노출한다
- 파드, 애플리케이션 등이 외부 사용자나 앱들과 연결될 수 있게 만들어 줌
- 파드가 외부 통신을 하는데 주로 사용 됨
- 여러 파드들에 단일 네트워크 진입점을 제공
- 따라서 Port지정이 필요

05. Service

Service 유형

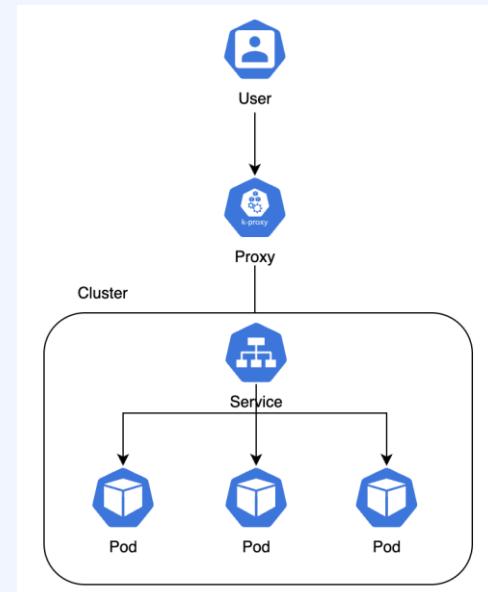
*default 는 ClusterIP

- ClusterIP
- NodePort
- LoadBalancer
- ExternalName

05. Service

Service 유형

- ClusterIP : 클러스터 내부의 다른 리소스들과 파드들이 통신할 수 있도록 해주는 가상의 IP
- ClusterIP로 들어온 트래픽을 해당 PodIP:targetPort로 전송
- IP를 알고있어도 외부에서 접근은 불가함
Proxy 또는 port forward 등으로 접근 가능
- 가장 기본적인 형태의 서비스



05. Service

Service 유형

- ClusterIP

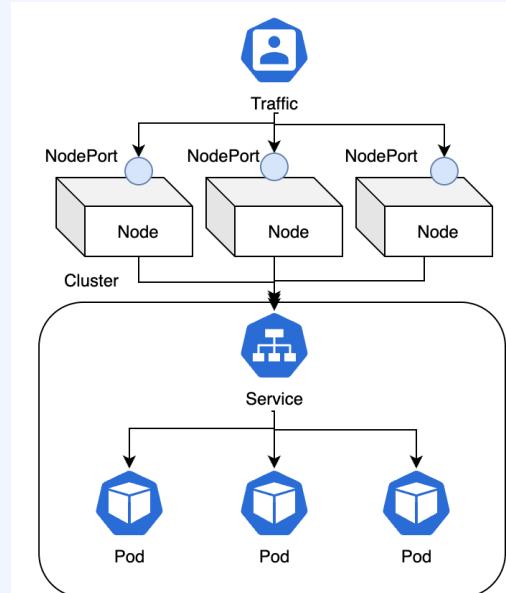
```
apiVersion: v1
kind: Service
metadata:
  name: clusterip-hostname-service
spec:
  type: ClusterIP
  selector:
    app: hostname-server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

- kind : Service
- spec.type : 서비스 타입 입력, ClusterIP
- port : 클러스터 내부에서 사용할 Service의 포트
- targetPort : 전달된 리퀘스트를 파드로 전달할 때 사용되는 포트

05. Service

Service 유형

- NodePort : 노드 포트는 외부에서 클러스터 내의 파드를 연결할 수 있음
- 모든 노드에서 특정 포트를 열고 이 포트로 전송되는 트래픽을 서비스로 전달
- NodeIP:NodePort로 전달되는 요청을 감지
NodePort는 30000 ~ 32767 사이의 포트
- 해당 파드의 포트로 트래픽을 전송
- 클러스터 내부로 들어온 트래픽을 파드로 연결하기 때문에
ClusterIP도 생성됨



05. Service

Service 유형

- NodePort

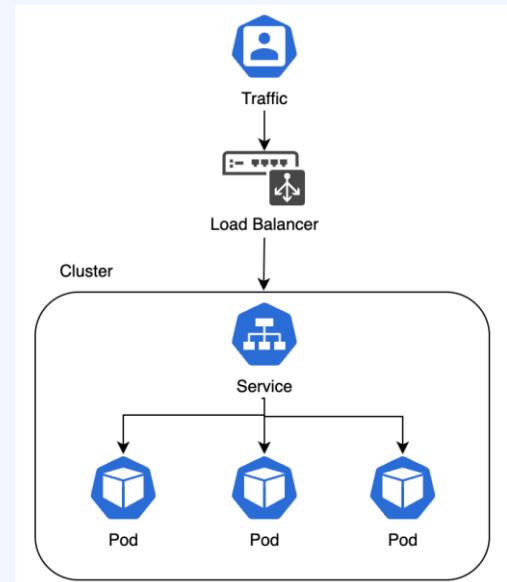
```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-hostname-service
spec:
  type: NodePort
  selector:
    app: hostname-server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
      nodePort: 30080
```

- kind : Service
- spec.type : 서비스 타입 입력, NodePort
- protocol : 사용할 프로토콜, TCP 사용
- port : 클러스터 내부에서 사용할 Service의 포트
- targetPort : 전달된 리퀘스트를 파드로 전달할 때 사용되는 포트
- nodePort : 외부에서 접속하기 위한 포트
- nodePort -> Port -> targetPort

05. Service

Service 유형

- LoadBalancer : 서비스를 노출하는 표준 방법
- AWS, GCP 등에서 사용하게 되면 클라우드 사의 자체 로드 밸런서로 노출됨
- 여기에 필요한 NodePort, ClusterIP도 자동 생성됨
따라서 비용이 들어가게 됨
- 로드밸런싱 방법은 클라우드 사의 설정에 따름
- 내부/외부 로드밸런서 사용 가능
이에 따라 적절한 Annotation 설정이 필요
- 클라우드 환경이 아니라면 NodePort와 동일하게 동작함



05. Service

Service 유형

- LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: hello-node-svc
spec:
  selector:
    app: hello-node
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  type: LoadBalancer
```

- kind : Service
- spec.type : 서비스 타입 입력, LoadBalancer
- protocol : 사용할 프로토콜, TCP 사용
- port : 클러스터 내부에서 사용할 Service의 포트
- targetPort : 전달된 리퀘스트를 파드로 전달할 때 사용되는 포트

05. Service

Service 유형

- ExternalName : DNS 이름에 대한 서비스에 매핑하는 방식
- 외부서비스를 쿠버네티스 내부에서 호출할 때 사용
- 파드가 Cname을 통해 특정 도메인과 통신하기 위해 생성
- 클러스터 내 파드는 내부 IP기 때문에 외부 통신이 매우 복잡
- externalName에 해당하는 도메인을 넣어주면 포워딩이 가능
- 일종의 프록시 역할을 함

05. Service

Service 유형

- ExternalName

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: www.google.com
```

- my-service를 검색했을 때 www.google.com이 리턴
- 클러스터 내부에서 외부로 접근이 가능

Chapter 2. Kubernetes의 기본 개념과 구성

06. Storage

06. Storage

Storage

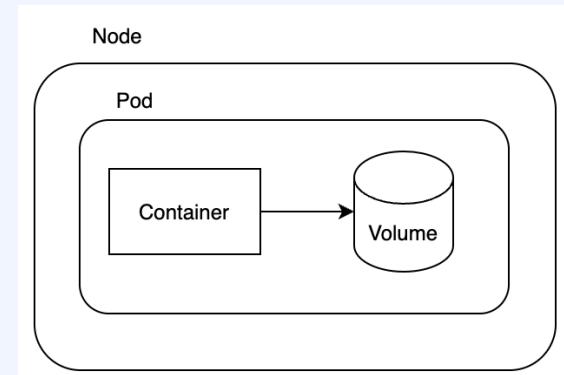
- 파드가 죽으면 내부에 있던 데이터는 모두 사라진다
- Storage는 파드에 long-term, 또는 임시의 스토리지를 제공할 수 있는 방법
- 대표적인 Storage
- Volume
- PersistentVolume
- PersistentVolumeClaim

06. Storage

Volume

- 컨테이너가 내려갈 때 컨테이너 내의 파일이 손실된다
- 이를 방지 위한 종류 중 하나가 볼륨
- persistentVolume 없이 파드에서 직접 볼륨을 지정할 수 있다
- emptyDir : 파드가 노드에 할당될 때 생성됨, 파드가 실행되는 동안만 존재

처음엔 비어있음, 파드 내 모든 컨테이너가 동일한 파일을 읽고 쓰는게 가능



06. Storage

Volume

- emptyDir

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: registry.k8s.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
    volumes:
      - name: cache-volume
        emptyDir: {}
```

- spec.containers.volumeMounts : 마운트할 경로와 이름을 정한다
- spec.containers.volumes : 이름을 정하고 어떤 종류의 볼륨인지 지정한다

06. Storage

Volume

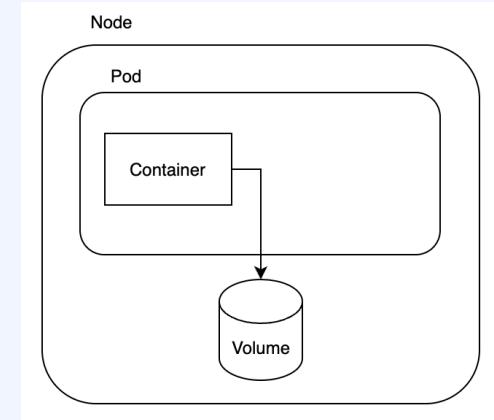
- **hostPath** : 파드들이 생성된 노드의 path를 볼륨으로 사용함
- path를 파드들이 공유함으로써 파드가 내려가도 노드의 볼륨은 그대로
- 파드가 내려가고 다른 노드에 생성된다면 기 생성된 노드의 볼륨은 사용할 수 없다

*HostPath 볼륨에는 많은 보안 위험이 있음

가능하면 HostPath를 사용하지 않는 것을 권장

HostPath 볼륨을 사용해야 하는 경우, 필요한 파일 또는

디렉터리로만 범위를 지정하고 ReadOnly로 마운트



06. Storage

Volume

- hostPath

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: registry.k8s.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
  volumes:
    - name: test-volume
      hostPath:
        path: /data
        type: Directory
```

- volumeMounts.mountPath : 볼륨을 마운트 하는 경로
- volumes.hostPath.path : 호스트의 디렉토리 경로
노드의 /data에 존재, 노드의 해당 경로에서 데이터를 사용함
- volumes.hostPath.type: 선택사항

06. Storage

Volume

- **configMap** : 데이터 또는 설정값을 파드에 넣는 방법을 제공
- 파드에서 컨피그 맵에 들어간 데이터를 참조하고 컨테이너의 애플리케이션이 소비
- API키, 환경 설정 파일들, cfg 등등
- 파드가 생성될 때 값을 key value형식으로 넣어주는 것
- 보안에 관련되거나 노출되면 안되는 정보는 제외

06. Storage

Volume

- **configMap**

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox:1.28
      volumeMounts:
        - name: config-vol
          mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        items:
          - key: log_level
            path: log_level
```

 - volumeMounts.mountPath : 볼륨을 마운트 하는 경로
 - volumes.configMap.name: 컨피그 맵의 이름
 - volumes.configMap.items : 들어갈 데이터, 키 밸류
 - data: data 아래에 example.yaml 과 같이 파일을 만들어서 마운트 시킬 수 있음

06. Storage

Volume

- **Secret**: configMap과 같이 데이터 또는 설정값을 파드에 넣는 방법을 제공
- 보안에 관련되거나 노출되면 안되는 정보에 대해서 Secret으로 관리함
- Key Value 형태
- 값에 해당하는 부분을 base64로 인코딩하는 작업이 필요
- 바이너리 파일 저장을 지원하기 위해 base64로 인코딩 하는 것

06. Storage

Volume

- **Secret**

```
apiVersion: v1
kind: Secret
metadata:
  name: hello-secret
data:
  language: cGFzc3dvcmQ=
  (password)
```

- kubectl secret 명령을 통해 값을 자동으로 base64화 할 수 있음
- 일일히 yaml에 넣는 것보다 kubectl secret으로 생성하는 것을 권장

06. Storage

Storage

- PersistentVolume(PV): 스토리지를 추상화하여 접근하는 방식
- 영구 스토리지 볼륨을 사용하기 위해 사용
- PV는 영구 스토리지 볼륨으로 설정하기 위한 객체
- PV는 클러스터 리소스
- 정적 프로비저닝 – Admin이 여러 PV를 만들어 놓는 것
- 동적 프로비저닝 – Admin이 StorageClass를 구성해 놓음,
StorageClass를 통해 동적으로 프로비저닝 한다

06. Storage

Storage

PersistentVolume 접근 모드의 종류

- **ReadWriteOnce** - 하나의 노드에서 볼륨을 읽기-쓰기로 마운트
- **ReadOnlyMany** - 여러 노드에서 볼륨을 읽기 전용으로 마운트
- **ReadWriteMany** - 여러 노드에서 볼륨을 읽기-쓰기로 마운트
- **ReadWriteOncePod** - 하나의 파드에서 볼륨을 읽기-쓰기로 마운트, 쿠버네티스 버전 1.22 이상인 경우에 CSI 볼륨에 대해서만 지원됨

06. Storage

Storage

- PersistentVolumeClaim(PVC): 유저의 스토리지에 대한 요청
- 파드가 노드 리소스를 사용하는 것처럼, PVC는 클러스터 리소스를 사용함
- PVC는 특정 크기의 스토리지 및 접근 모드를 요청할 수 있다
- 파드가 볼륨을 직접 할당하게 하지 않고 PVC를 두는 이유?

파드와 파드가 사용할 스토리지를 분리 - 각 상황에 맞게 다양한 스토리지 활용 가능한 구조

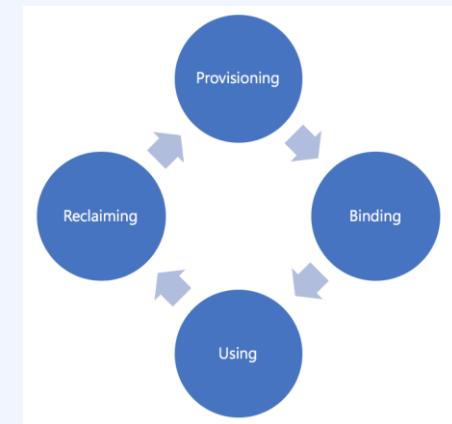
PVC를 사용하게 되면서 파드는 어떤 스토리지 사용할지 고민하지 않아도 됨(일일이 설정 X)

06. Storage

Storage

PV, PVC의 생명주기(Lifecycle)

- Provisioning
- Binding
- Using
- Reclaiming

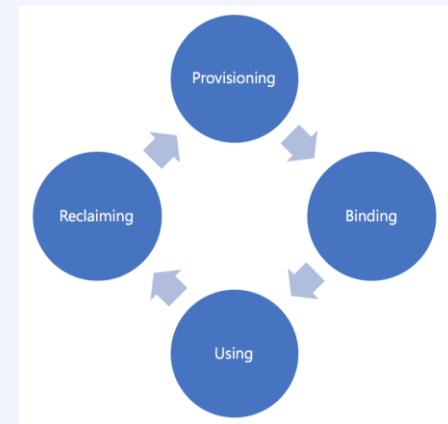


06. Storage

Storage

PV, PVC의 생명주기(Lifecycle)

- Provisioning : PV의 생성 – 프로비저닝
- 정상적으로 올라온다면 Status Available
- 정적 프로비저닝 : Manifest 파일을 통해 정해진 용량의 PV를 생성
요청 있을 때 미리 생성한 PV를 할당
- 동적 프로비저닝 : PV를 생성하여 할당하고 원하는 용량의 PV를
생성해서 자유롭게 사용 가능

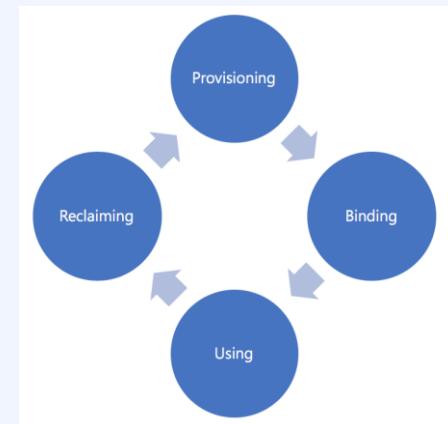


06. Storage

Storage

PV, PVC의 생명주기(Lifecycle)

- Binding : PV를 PVC에 연결하는 단계
- PVC는 요청된 볼륨을 PV에 요청하고 해당하는 볼륨이 있으면 할당
없다면 요청은 Pending
- PVC 대 PV 바인딩은 일대일 매팅
PV와 PVC사이의 양방향 바인딩인 ClaimRef를 사용

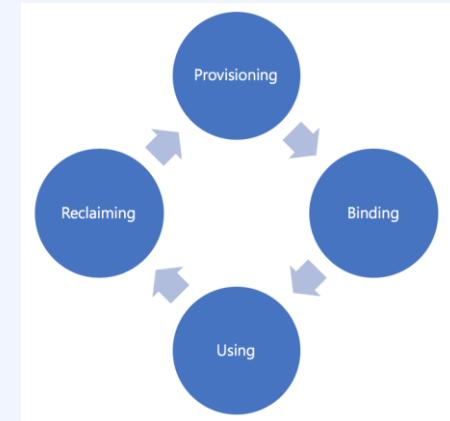


06. Storage

Storage

PV, PVC의 생명주기(Lifecycle)

- Using : 1:1 매핑 이후 파드가 PV를 볼륨으로 사용
 - 클러스터는 PVC를 확인 후 바인딩된 PV를 찾아 파드가 사용하도록 함
 - Storage Object in Use Protection에 의해 파드가 삭제 불가
- PVC 상태에 Finalizers
- 파드가 PVC를 사용하지 않아야 정상적으로 삭제가 됨

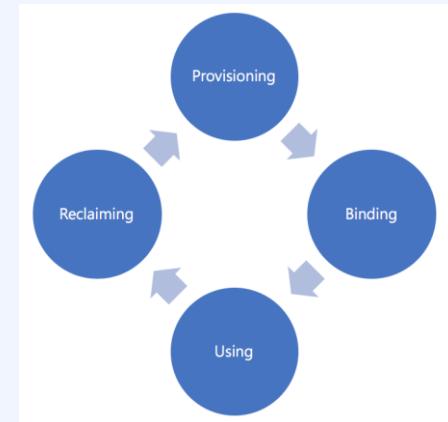


06. Storage

Storage

PV, PVC의 생명주기(Lifecycle)

- Reclaiming : PVC는 다른 PVC로도 재활용이 가능
- 파드가 사용하지 않는 PVC를 삭제할 때 PV 처리에 대한 설정
- Retain : 기존 PV 데이터 보존(수동 삭제 필요)
- Recycle : 기존 PV데이터 삭제 후 재사용
- Delete : 사용 종료 시 볼륨 삭제



06. Storage

Storage

StorageClass

- 관리자가 제공하는 스토리지의 Classes에 대한 명세
- 기존에는 관리자가 PersistentVolume을 개발자(사용자)로 부터 요청을 받을 때마다 생성
- 수동 프로비저닝의 불편함을 해결, 동적 프로비저닝을 위한 도구
- Container Storage Interface(CSI) : 컨테이너 오케스트레이션 시스템의 컨테이너화된 워크로드에 임의의 블록 및 파일 스토리지 시스템을 노출하기 위한 표준
- 볼륨 확장을 위해서는 Kubernetes와의 직접적인 연계 없이 구현가능
- **StorageClass는 CSI의 구현체를 가리키는 오브젝트**

*동적 프로비저닝은 모든 쿠버네티스 클러스터에서 범용적으로 사용할 수 있는 것은 아님
동적프로비저닝 기능이 지원되는 스토리지 Provisioner의 활성화 필요

06. Storage

Storage

StorageClass

- Provisioner : PV 프로비저닝에 사용되는 볼륨 플러그인을 결정, 반드시 지정
- 내부 프로비저너: AWSElasticBlockStore, GCEPersistentDisk, AzureFile, NFS, Glusterfs, Local ...
- CSI는 K8S와 Storage Provider간 인터페이스 담당, PV 자동 생성 가능
- Provisioner가 있다면 개별 정의된 Parameter를 통해 세부 연결 방식을 정의가 가능함

06. Storage

Storage

StorageClass 동적 프로비저닝 순서:

1. PVC는 StorageClass를 사용해 동적 프로비저닝 트리거링
2. VolumeProvisioning호출, Parameter type의 값, CreateVolume호출을 Provisioner에 전달
3. Provisioner는 새 볼륨을 생성, 이에 대한 PV를 자동으로 생성
4. K8S는 새 PV를 PVC에 바인딩
5. 파드는 PVC를 추가해 바인딩된 PV를 사용

*AWS EKS에서는 1.23 버전 이후 ebs-csi 플러그인 설치와 이에 해당하는 권한(AmazonEBSCSIDriverPolicy) 설정이 필요

*설치 이후 해당 드라이버에 맞는 StorageClass를 생성

Chapter 3. Kubernetes 기타 개념

Chapter 3. Kubernetes 기타 개념

01. Helm

01. Helm

Helm

- Helmsman에서 온 단어, 조타수, 키잡이를 뜻함(배를 조종하는 사람)



- 위 그림은 공식 페이지에서 Helm에 대해서 설명하는 것
- K8S를 위한 패키지 매니저(pip, apt와 비슷)

01. Helm

Helm

- 복잡한 Kubernetes 애플리케이션도 정의하고 설치 및 업그레이드하는데 도움
- Helm을 통해 애플리케이션을 관리 (yaml 복사 붙여넣기 X)
- CNCF의 graduated project

Helm Repository

- 차트는 저장소가 필요함
- 차트를 패키징에서 업로드 할 수 있는 곳
- Private Chart – Github Repository
- Public Chart – Artifact Hub(<https://artifacthub.io/>)

01. Helm

Helm Chart

- Helm 차트는 쿠버네티스 리소스와 관련된 셋을 설명하는 파일의 모음
- Public/Private 레포지토리에 올려놓고 받아 사용
- yaml을 차트로 패키징한 뒤 K8S에 애플리케이션을 쉽게 배포할 수 있음
- 애플리케이션에 필요한 Deployment, Storage, Service등이 정의되어 있음
- 차트는 특정한 디렉터리 구조를 가진 파일들로 생성
- helm pull 명령어를 통해 공개된 차트를 받을 수 있다
- 차트를 구성한 이후에 helm install로 설치

01. Helm

Helm Chart 구조

- FCApp이라는 애플리케이션이 있다면,
- FCApp이라는 디렉토리 안에 파일들을 구성해놓는다
- (헬름은 charts/, crds/, templates/ 디렉터리와 나열된 파일명을 예약어로 사용)

Chart.yaml 차트에 대한 정보를 가진 YAML 파일

LICENSE 옵션: 차트의 라이센스 정보를 가진 텍스트 파일

README.md 옵션: README 파일

values.yaml 차트에 대한 기본 환경설정 값들

values.schema.json 옵션: values.yaml 파일의 구조를 제약하는 JSON 파일

charts/ 이 차트에 종속된 차트들을 포함하는 디렉터리

crds/ 커스텀 자원에 대한 정의

templates/ values와 결합될 때, 유효한 쿠버네티스 manifest 파일들이 생성될 템플릿들의 디렉터리

templates/NOTES.txt 옵션: 간단한 사용법을 포함하는 텍스트 파일

01. Helm

Helm Chart 구조

`values.yaml` : 템플릿에 사용될 변수들을 모아놓은 파일

- Chart의 리소스 템플릿 파일들이 구성되어 있음
- `values.yaml` 파일만 수정해 적용하면 원하는 차트 배포가 가능
- `values.yaml` 수정된 이후에는 파드들이 내려갔다가 다시 올라오게됨
- 만약 Volume, Storage 설정을 안했다면, 파드 안의 데이터는 사라짐을 주의

01. Helm

Helm Chart 구조

*Helm 템플릿 언어는 Go lang으로 되어 있다

*Go의 자료형을 사용

- string: 텍스트 문자열
- bool: true 또는 false
- int: 정수 값 (8, 16, 32, 64 비트의 부호가 있거나(signed) 없는(unsigned) 다양한 자료형이 있다)
- float64: 64비트 부동 소수점 값 (8, 16, 32 비트의 다양한 자료형이 있다)
- byte slice([]byte), 흔히 (잠재적으로) 바이너리 데이터를 담기 위해 사용된다.
- struct(구조체): 프로퍼티와 메소드를 가지는 객체
- 위의 자료형 중 하나에 대한 슬라이스(인덱스 있는 리스트)
- 위의 자료형 중 하나에 대한 문자열-키 맵(map[string]interface{})

01. Helm

Helm Release

- K8S에서 구동되는 차트 인스턴스
- 클러스터 내부에 차트를 Repository에서 검색해서 설치하면
각 설치에 맞는 새 Release가 생성됨
- Docker와 같이 Helm도 배포된 서비스에 대한 버전관리를 위해 Release가 존재
- 여러 yaml파일로 관리한다면, 변경사항에 대한 롤백과 업데이트가 어려워짐
- Release로 관리하게 되면 업데이트와 롤백이 용이해짐
날짜 확인도 가능

01. Helm

Ex) Airflow 차트 설치

차트 설치

- helm repo add apache-airflow <https://airflow.apache.org>
- helm upgrade --install airflow apache-airflow/airflow --namespace airflow --create-namespace

차트 업그레이드

- helm upgrade airflow apache-airflow/airflow --namespace airflow

차트 삭제

- helm delete airflow --namespace airflow

Chapter 3. Kubernetes 기타 개념

02. Yaml

02. Yaml

Yaml

- YAML은 XML, C, 파이썬, 펄, RFC2822에서 정의된 e-mail 양식에서 개념을 얻어 만들어진 '사람이 쉽게 읽을 수 있는' 데이터 직렬화 양식이다. 2001년에 클라크 에반스가 고안했고, Ingy dot Net 및 Oren Ben-Kiki와 함께 디자인(<https://ko.wikipedia.org/wiki/YAML>)
- YAML의 이름은 YAML Ain't Markup Language라는 이름에서 유래
- 원래의 뜻은 Yet Another Markup Language
- yaml과 yml, yaml은 공식 확장자이며 그 외의 확장자로 yml도 사용
과거에는 파일의 확장자 길이가 3자로 제한되었기 때문

Yaml 사용 이유

- 한 파일에서 모든 configuration을 관리하기 위해 Yaml을 사용
- 설정 파일 관리하는데 굉장히 용이

02. Yaml

Yaml 문법

- YAML은 기본적으로 사람 눈으로 보기 가 편함
- 설정 파일 관리하는데 굉장히 용이
- yaml은 python과 마찬가지로 #을 주석을 다는 문자로 사용
- indentation에 민감
- yaml 문법 확인

<https://www.yamllint.com/>