

0. 개요

◦ 컨테이너

서버의 한 종류 → 소프트웨어(애플리케이션) 실행되는 곳

대부분의 서버는 사양이 매우 좋지만, 서비스 장애 대비를 위해 서버와 App은 1:1이 안전

→ 자원의 낭비가 발생함 → 가상화 기술 사용 : 하나의 서버에 여러 App 실행 가능

가상화 기술의 한 종류

(Host) OS 위에 여러 개의 격리된 환경 생성

컨테이너 안에서 애플리케이션 실행

Guest OS + Hypervisor = Container Engine
⇒ Docker Engine

* Container Engine

- 가상 머신을 생성·구동하는 SW

- 각각의 OS에 관련된 네트워크

↓ OS들의 환경을 분리 → 하드웨어에 진입

◦ 가상화 기술

서버에 가벽을 세우는 것처럼 하나의 서버에서 소프트웨어 실행환경을 분리하는 것

이전에는 VM(가상マシン)을 사용했으나, 프로세스 실행시간과 암고메모리에 많은 시간이 소요됨

→ 가상머신보다 빠르고 가벼운 컨테이너를 통한 가상화 기술이 주로 사용 → 효율적인 자원 사용, 운영의 유연성 확장

◦ 도커

- 도커 : 컨테이너를 관리 해주는 도구

애플리케이션 배포, 전달, 실행을 위한 개방형 플랫폼 (Open platform)

애플리케이션을 인프라에서 끌어내 → SW를 신속하게 제공

인프라를 애플리케이션 커널 관리 가능

코드 배포와 품질

'컨테이너'라는 적극적인 환경에서 애플리케이션을 재작성, 강제, 버그
필터링, 자체 품질 향상 (한번은 고친다)

(유익) 컨테이너 기본의 가상화 도구

애플리케이션을 짓거나 실행, 배포하는 단위

◦ 쿠버네티스

- k8s : 여러 컨테이너 관리 도구를 관리해주는 도구 = 컨테이너 오퍼레이션 도구

◦ Docker vs Containerd

컨테이너 관리 도구는 Docker 가 그 시작

→ k8s 는 처음엔 도커만 지원하다가 점점 지원 대상이 추가됨

* CRI : Container Runtime Interface

- k8s 와 Container Runtime Engine 을 사용할 수 있도록 해주는 인터페이스

* OCI : Open Container Initiative

- CRI 를 사용하는 공급자가 가져야 하는 표준

- imageSpec + runtimeSpec 으로 구성됨

↳ 이미지 빌드 방식에 대한 규칙

- Docker 는 OCI 탑재 이전부터 존재했으므로 표준과 거리가 떨어짐

→ 임시방편으로 dockershim 도입 → k8s 1.24 버전부터 dockershim 을 제거하고 Docker 지원 중단

- Containerd 는 Docker 와 달리 k8s 와 직접 상호작용 가능

Docker Image 는 OCI 표준規格을 지키고 있음

→ 이제는 계속 사용 가능

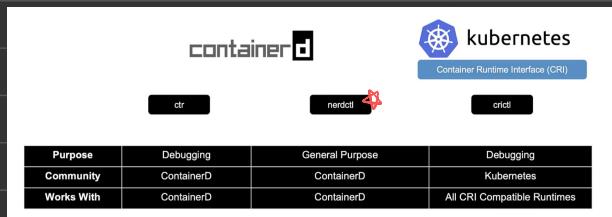
.∴ Containerd 가 더 강점화됨

◦ Containerd 컨테이너 실행 방법

① ctr

- 유저 친화적이지 않음

- 주로 디버깅 목적으로만 사용



② nerdctl

- 도커 명령어와 비슷하고 가장 많이 사용됨

- 다양한 기능 지원 ex) 암호화된 컨테이너 이미지, Lazy Pulling, P2P image Distribution 등

- nerdctl run --name redis redis:alpine

docker 2 ↗️ nerdctl run --name webserver -p 80:80 -d nginx
버켓도 통합

③ crictl

- 주로 디버깅 목적으로만 사용

◦ ETC

간단, 빠름, 분산, 신뢰 가능 키-밸류 저장소

표 형식의 전통적 저장소는 데이터가 추가되면 빈 공간 생길 수 있음
변경 사항이 다른 리스너에 영향을 줄 수 있음

Documents 또는 Pages 형식의 저장소

⇒ 모든 관련 정보가 Document에 저장됨

⊕ 어떤 형식의 파일이라도 저장 가능

K8S 클러스터의 다양한 정보를 저장하는 역할

↳ 현재 상태, 구성 정보, 시크릿, 설정 등

모든 노드 구성 요소는 ETC를 통해 데이터를 저장 간택

◦ Worker Node

클러스터에서 실제 애플리케이션의 실행되는 물리적 / 논리적 머신

애플리케이션 컨테이너 실행 관리 서비스 제공

◦ Master Node

클러스터의 제어 및 상태 관리를 담당

→ 전체적인 오케스트레이션을 수행

◦ kube-apiserver

K8S의 API 서버 : Restful API를 제공

클러스터의 중심 인터페이스 ⇒ 클라이언트와의 모든 상호작용이 이루어지는 곳

o Kube Controller Manager

컨트롤러 관리하는 역할

- └─ ① Watch Status : 상태 모니터링
- ② Remediate Situation : 원하는 상태로 설정

- 종류

- Node - Controller
 - └─ 아�플리케이션을 지속동작시키기 위한 행동을 취함 (through kube-api-server)
 - 노드의 상태 모니터링 (주기 : 5초)
 - 상태 체크가 안되면 unreachable 표기 (유예기간 : 40초)
 - 다시 실행할 시간을 줌 (Pod Eviction Timeout : 5분) → 5분 후에는 죽은 것으로 간주하고 node를 terminate
 - └─ 예플리케이션에서 프로비저닝을 통해 새로운 올림
- Replication - Controller
 - └─ Replicaset의 health check 담당
 - 원하는 수의 POD가 항상 사용가능하도록 준비
 - ⇒ POD가 죽으면 새로운 POD 생성
- etc...

이러한 컨트롤러들이 하나의 프로세스인 kube-controller-manager 이 위치

모든 컨트롤러 enable 상태가 디폴트 설정

- 융선 확인 방법 : 글러소더 설정 방법에 따라 다음과

- ① kubeadm 사용시 ~ 마스터 노드의 네임스페이스에 POD로서 배포
- ⇒ etc/kubernetes/manifests 에 위치한 POD 정의 파일에서 융선 확인 가능

- ② non-kubeadm

 ⇒ 서비스 설정 파일에서 확인 가능

- ③ ps -aux | grep kube-controller-manager 정령어로도 확인 가능

o Kube-scheduler

NODE에서 POD의 스케줄링 담당

직접 만드는 것이 아닌 어떤 POD가 어떤 NODE로 갈지 결정만 하는 역할

kubelet의 역할

특정한 기준을 토대로 결정을 진행

ex) 서로 다른 리소스 요구사항을 가진 PODs + 여러 NODE를 가진 클러스터

⇒ kube-scheduler는 POD를 이용해 Best Node를 찾아야 함

커스텀 마이닝
가능

① 요구사항을 바탕으로 Node 필터링
② POD 적합도 점수를 각 Node마다 산정

→ taints & tolerations
node selectors / Affinity

o kubelet

Ship : captain = worker Node : kubelet

컨테이너 올/내리기 등 모든 활동을 이끄는 역할 → 컨테이너 런타임 엔진에 image pull & run 요청을 보냄
(like Docker)

Master Node 와의 유일한 contact 지점

Master Node의 kube-apiserver에 노드 가치와 컨테이너들의 상태를 주기적으로 보고

kubeadmin은 차동으로 kubelet을 배포하지 않음 (다른 component 와의 차이점)

→ 예전 워커노드에 직접 설치해주어야 함

o Kube-proxy

각 Node에서 동작하는 프로세스 : 새 서비스를 찾고, 그에 맞는 접근규칙을 생성 (netfilter, iptables)

POD의 네트워킹 풀무원 담당 → 모든 POD가 다른 POD에 접근할 수 있도록 만듦

POD의 네트워크 = 모든 POD가 연결된 클러스터 전반에 걸쳐있는 내부 가상 네트워크

Kubeadmin이 각 Node에 POD 형태로 배포

시작은 DaemonSet 형태이기 때문에

노드에 항상 자동 배포됨

◦ POD

애플리케이션의 단일 인스턴스로 k8s에서 만들 수 있는 가장 작은 단위 Object

하나 이상의 컨테이너를 포함하는 논리적 흐르팅 단위

k8s는 워커노드에 컨테이너를 직접 배포하지 않음

↳ 컨테이너는 겹슬화됨

POD를 추가적으로 생성해야 할 때, 어떤지 생성(spin up)해야 하는가?

① 같은 node 내 같은 Pod 내 (x)

② 같은 node 내 새로운 Pod 내 (o)

↳ node에 충분한 용량이 없는 경우 : 새로운 node 생성 및 새로운 Pod 내 생성

* Pod et Container는 주로 1:1 관계

- Scale up : 새로운 Pod에 컨테이너 생성 (Pod 안에 컨테이너 더 만들지 않음)

- Scale down : Pod 삭제

* 앤 번드는 것이지, 뜬 번드는 것은 아님

⇒ 컨테이너 내 애플리케이션이 도움을 주기 위한 Supporting App 인 경우,

동일 Pod 내에 컨테이너를 생성하기도 함

⇒ 생성과 삭제가 동시에 이루어지게 됨

⇒ 두 컨테이너는 localhost로 직접 통신 · 동일 네트워크 사용 · 소프라저 공유 등이

◦ kubectl run

pod를 만들면서 그 안에 컨테이너를 배포

(① POD 만들기
 ② image pull
 ③ image로 인스턴스 생성)

◦ kubectl get pods

available Pod 목록 확인 명령어

◦ kubectl describe pod 파드명

Pod에 대한 상세정보

◦ YAML 기반 POD 생성

명령어: `kubectl create -f *.yaml`
 (`apply`) → pods, replicas, deployments, services 등
K8S에서의 yaml : K8S object 생성을 위한 입력값

* YAML의 4가지 top-level 필수 필드

① apiVersion

- Object 생성에 사용하는 K8S API 버전
- 만들려는 대상에 따라 달라짐

② kind

- object의 type

③ metadata

- object에 관한 정보
- name
- labels : 그룹핑을 도와줌 (ex. FE, BE, DB 등)

④ spec

- 명세서 : object 따라 다름
- 대시(-)는 list item 표기법

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: python-sample-deployment
  namespace: default
spec:
  selector:
    matchLabels:
      app: python-sample-app
  replicas: 1
  template:
    metadata:
      labels:
        app: python-sample-app
    spec:
      containers:
        name: python-sample
        image: dockerimage주소:태그
        imagePullPolicy: IfNotPresent
      resources:
        requests:
          cpu: 500m
          memory: 200Mi
      ports:
        - containerPort: 8000
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```

o Replication Controller

하나의 Pod에 대한 여러개의 인스턴스를 통제시킬 수 있도록 해줌 → 고가용성

④ pod가 하나만 존재할 때, fail이 되면 자동으로 새로운 pod를 퍼옴

.. 항상 특정 개수의 pod가 running하는 것을 보장해주는 도구
개수에 제한없이

그 외에도 Load Balancing & Scaling의 용도로 사용

vs. ReplicaSet

Replication Controller는 오래된 기술로서

거의 동일한 역할을 하는 ReplicaSet으로 대체되는 추세

- Replication Controller 설정 방법

① 정의 파일 만들기 → rc-definition.yaml

apiVersion : v1

kind : Replication Controller

metadata

Spec 중요 : (replica instance 생성을 위한 Pod template 이 필요)

⇒ kubectl create -f rc-definition.yaml

template :

apiVersion, kind
두 가지를 지정한
나머지 전부 각성

replicas : (replica를 얼마나 만들 것인가)

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-rc
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

◦ Replica Set

- 생성방법

apiVersion : apps/v1

kind : ReplicaSet

metadata :

name :

labels :

app:

+type:

spec :

template :

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      tier: backend
  template:
    metadata:
      labels:
        app: myapp
        tier: backend
    spec:
      containers:
        - name: myapp-container
          image: myapp:latest
          ports:
            - containerPort: 80
```

replicas :

→ Replication Controller 대신 사용가능한 필드이지만 생각가능 ⇒ 생략시 Pod 정의부분의 labels 를 기준

~~필수~~ selector : (Replication Controller 와의 차이점)

matchLabels : (ReplicaSet 이 관리할 Pod 를 선택하는 기준) → 없으면 replicaset의 생성물이 아닌 Pod 을 관리대상이 될 수 있음

↳ metadata 의 내용으로 써번

⇒ kubectl create -f rs-definition.yaml

kubectl get replicaset

- Scale 변경 ex) replica 개수 3 → 6개

① 정의 파일 작성 수정

) 변경 반영 과정 필요 ⇒ kubectl replace -f *.yaml

② kubectl scale 명령

⇒ kubectl scale --replicas=6 -f *.yaml

⇒ kubectl scale --replicas=6 replicaset myapp-replicaset
metadata.type metadata.name

o Deployments

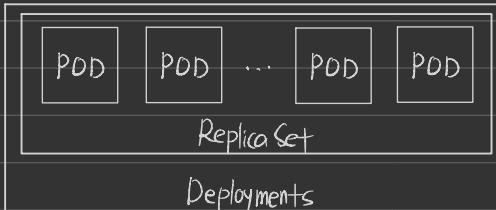
k8s에서 애플리케이션의 배포·관리 담당

- 배포 관리 : rolling update, 훌쩍 등

- 자동 스케일링 : Pod 수 조정

- 자가 복구 : 실패한 Pod 강제 및 교체

- 버전 관리 : 애플리케이션 버전 추적



- 생성 방법

replicaset 정의 파일과 비슷한 형식

Deployments는 ReplicaSet을 자동생성하고, ReplicaSet은 Pod를 자동생성

→ kubectl get deployments
(replicaset) all로 한번에 확인 가능
 pods

```
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:1.16
          ports:
            - containerPort: 80
```

o Service

애플리케이션 내외부 다양한 컴포넌트가 서로 통신 가능하도록 하는 역할 (k8s object의 한 종류)

(애플리케이션 ↔ 애플리케이션 or 애플리케이션 ↔ 사용자)

マイ크로 서비스 간 느슨한 결합을 도와줌 (Loose Coupling)

ex) 외부 애플리케이션이 돌아가는 Pod에 사용자는 어떻게 접근할까?

- Node는 IP 주소를 가짐
- Pod도 IP 주소를 가짐
) → 하지만, 둘은 다른 네트워크
 ↓

외부에서 Pod에 Ping / access 불가능

* Pod 접근 방법

① 노드에 SSH 접속 후, Pod에 curl 명령으로 접근

→ k8s 내부에서의 접근이므로 외부 접근이 아님

∴ 외부에서 접근하기 위해선 외부 요청을 Pod로 배포해주는 중간 계층이 필요 → Service

- Service 생성 방법

apiVersion : v1

kind : Service

metadata :

name :

labels :

spec :

type : NodePort / ClusterIP / LoadBalancer

ports :

- targetPort : 생략 시 'port' 설정값으로 자동설정

port : 필수

nodeport : 생략 시 범위 내에서 자동선택 할당

selector : Pod 식별 정보

app :) pod의 metadata.labels 필드 내용 기입

type :

⇒ 일단 설정해주면 어떤 상황에서도 알아서 서비스가 해줌 ⇒ labels의 중요성!

싱글파드 - 싱글 노드

멀티파드 - 싱글 노드

멀티 파드 - 멀티 노드

◦ NodePort Service

파드 Port 를 노드의 port 와 연결하는 역할의 Object

① target port : 실제 애플리케이션이 작동 중인 Pod 의 포트 ⇒ 서비스가 요청을 전달하는 포트

서비스 관점의
용어이므로
② (그냥) port : 서비스 자체에 있는 Port (* 서비스는 노드 내에 존재하는 가장 서버 같은 존재 → 자체 IP 주소를 가짐)

③ node port : 외부에서 애플리케이션에 접근할 때 쓰이는 포트 (기본 사용 범위 : 30000 ~ 32767)

◦ ClusterIP Service

다른 서비스들과 통신할 수 있게 클러스터 내부에 가상 IP 를 생성하는 Object

Pod 의 IP 는 사용자 입장에서 고정이 아님 (삭제 생성이 연체된다면 일어날 수 있음)

같은 역할을 하는 Pod 가 여러 개 존재할 수 있음

⇒ 요청을 어디로 보내야 할지 결정해주어야 함

⇒ k8s Service 가 Pod 들을 그룹화하여 Pod 에 접근하기 위한 단일 인터페이스를 제공

(요청을 보낼 파드는 무작위로 선정됨)

각 서비스는 클러스터 내에서 IP 와 이름을 부여받고, 통신하는데에는 이를 사용

Spec :

type : ClusterIP (기본값)

port :

- targetPort : 백엔드가 노출되는 포트

port : 서비스가 노출되는 포트

selector : Pod 의 metadata.labels 내용이 들어감

◦ Load Balancer Service

사용자와 서비스 중간에서 트래픽을 처리해주는 Object

사용자에게 하나의 URL 만 제공하여 여러 서비스에 접근할 수 있도록 함

◦ Namespace

지금까지의 object 들은 모두 namespace에서 작업하고 상호작용

파로 설정하지 않았다면 Default namespace : k8s가 클러스터를 처음 설정할 때 자동 생성

* kube-system

- 내부적 목적에 사용되는 서비스와 파드를 생성하는 namespace

- ex) 네트워킹 솔루션, DNS 서비스 등

- 실제 유저나 User로부터의 분리

* kube-public

- k8s 모든 사용자가 접근할 수 있는 차원들이 존재하는 namespace

작은 작업들은 Default namespace에서 충분히 가능

But! 엔터프라이즈 or 프로덕션 환경처럼 복잡한 경우 namespace 분리를 고려

⇒ 각 namespace는 각자의 고유정책을 설정하고, 권한을 설정할 수 있음

(각 namespace별 자원 할당량을 설정하고, 자원 최대치를 제한할 수 있음)

“namespace 내”에서는 각 서비스를 이름으로 구별·참조 가능

“서로 다른 namespace 간”에는 서비스 이름에 namespace를 포함시켜 참조해야 함

↳ 이름으로 참조가 가능한 이유 : 서비스 생성 시, DNS 항목에 자동으로 추가되기 때문

서비스의 서브메인

ex) db-service.dev.SVC.cluster.local
서비스 이름 namespace k8s 클러스터의 기본 설정 domain 이름

kubectl get pods : Default namespace에 존재하는 Pod 목록

④ --namespace 옵션 : 다른 namespace의 object 확인 가능

kubectl create --namespace=dev : dev namespace에 생성

또는

yml 파일의 metadata.namespace에 설정

- Namespace 생성

- 정의 파일 사용

```
apiVersion : v1
```

```
kind : Namespace
```

```
metadata :
```

```
name : 이중
```

- 명령어 사용 : kubectl create namespace dev

- Namespace 변경

```
kubectl config set-context $(kubectl config current-context) --namespace=dev
```

: 현재 namespace 상태에서 dev로 namespace switch

- Namespace 자원 할당량 설정 ⇒ ResourceQuota (object)

```
apiVersion : v1
```

```
kind : ResourceQuota
```

```
metadata :
```

```
name : compute-quota
```

```
namespace: dev
```

```
spec :
```

```
hard :
```

```
pods :
```

```
requests.cpu :
```

```
requests.memory :
```

```
limits.cpu :
```

```
limits.memory :
```