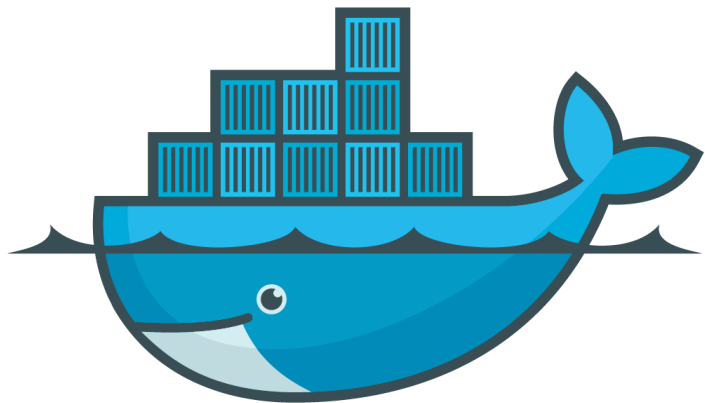


Docker & Kubernetes



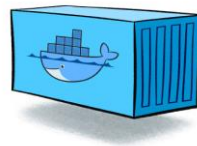


docker

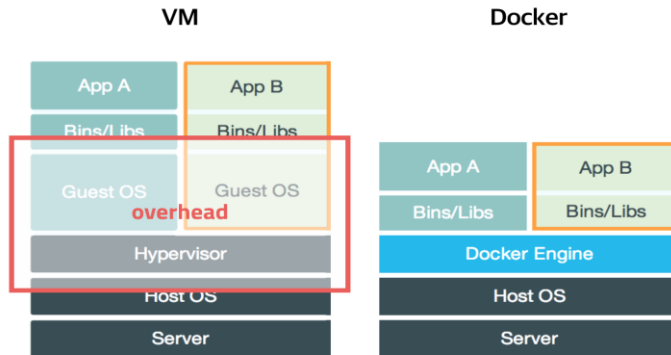
- 2013년 3월, dotCloud 창업자 Solomon Hykes가 Pycon Conference에서 발표
- Go 언어로 작성된 "The future of linux Containers"

- Container based

- 프로세스 격리 기술
- 오픈소스 가상화 플랫폼



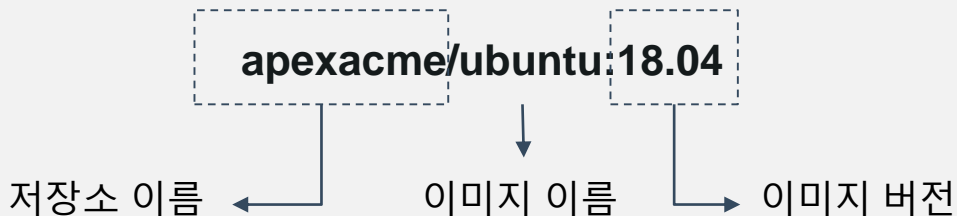
- 가상머신 .vs. Docker



Docker Image & Container

- Docker Image

- 가상머신 생성시 사용하는 ISO 와 유사한 개념의 이미지
- 여러 개의 층으로 된 바이너리 파일로 존재
- 컨테이너 생성시 읽기 전용으로 사용
- 도커 명령어로 레지스트리로부터 다운로드 가능

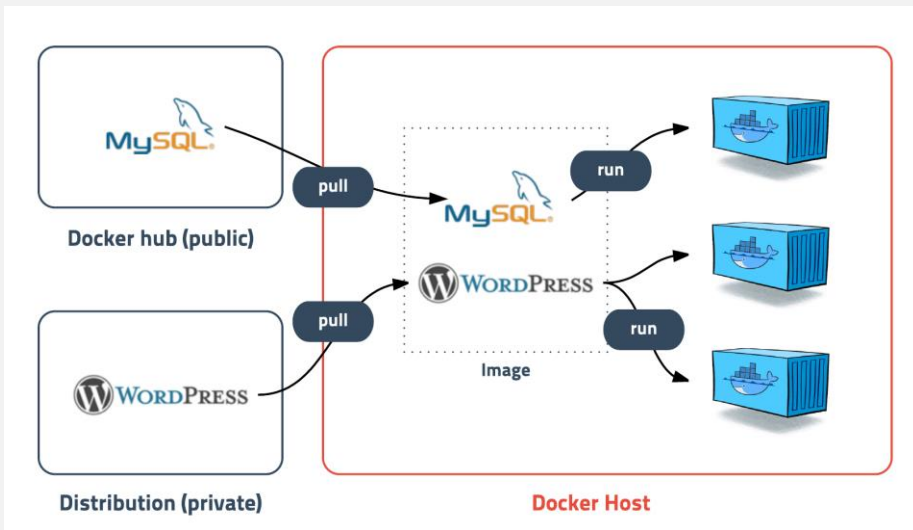


- 저장소 이름 : 이미지가 저장된 장소, 이름이 없으면 도커 허브(Docker Hub)로 인식
- 이미지 이름 : 이미지 이름, 생략 불가
- 이미지 버전 : 이미지 버전정보, 생략 시 latest 로 인식

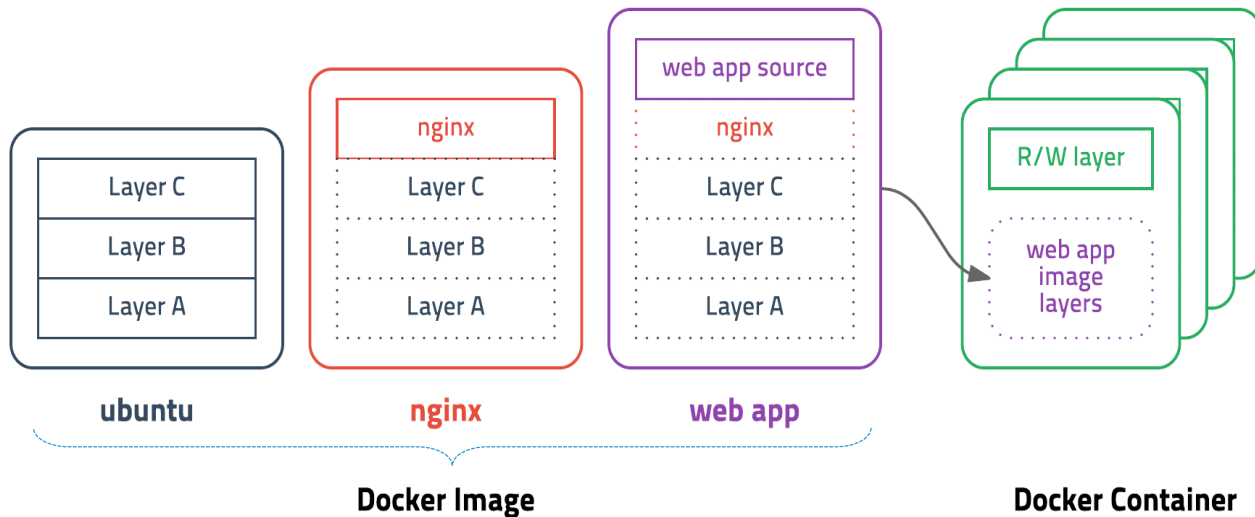
Docker Image & Container

- Docker Container

- 도커 이미지로 부터 생성
- 격리된 파일시스템, 시스템 자원, 네트워크를 사용할 수 있는 독립공간 생성
- 이미지를 읽기 전용으로 사용, 이미지 변경 데이터는 컨테이너 계층에 저장

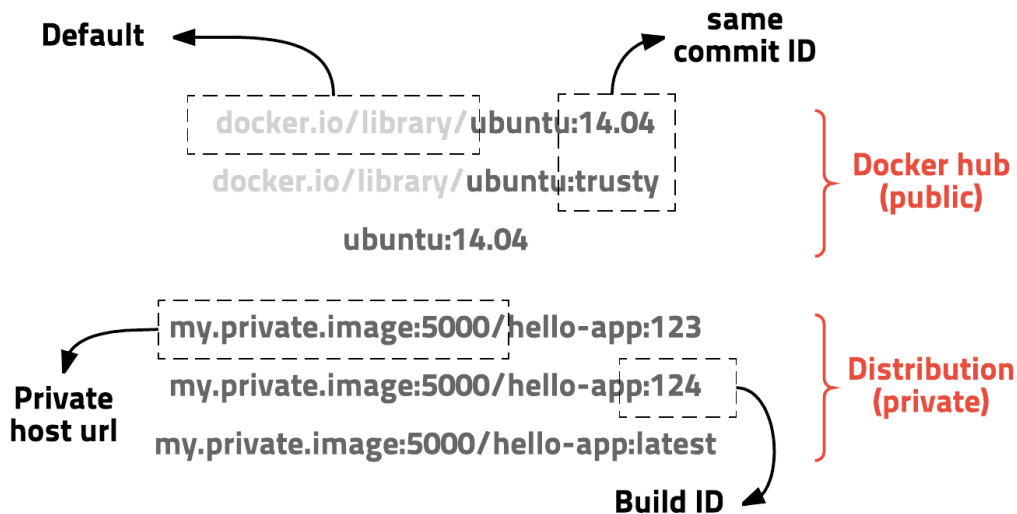


Layered Architecture



- Image : 여러 개의 읽기 전용(Read Only) 레이어로 구성
- Container : Image 위에 R/W 레이어를 두고, 실행 중 생성 또는 변경 내용 저장

Docker Image Path



- 이미지 Path는 <URL>/<namespace>/<Image_name>:<tag> 형식
- library는 도커허브 공식 이미지 Namespace로, 여기에 사용자 이름이 위치

Dockerfile sample

```
FROM openjdk:8-jdk-alpine
```

```
RUN apk --no-cache add tzdata && cp  
/usr/share/zoneinfo/Asia/Seoul /etc/localtime
```

```
WORKDIR /app
```

```
COPY hello.jar hello.jar
```

```
COPY entrypoint.sh run.sh
```

```
RUN chmod 774 run.sh
```

```
ENV PROFILE=local
```

```
ENTRYPOINT ["/run.sh"]
```

- **FROM** : 이미지를 생성할 때 사용할 베이스 이미지를 지정한다.
- **RUN** : 이미지를 생성할 때 실행할 코드 지정한다. 예제에서는 패키지를 설치하고 파일 권한을 변경하기 위해 RUN을 사용
- **WORKDIR** : 작업 디렉토리를 지정한다. 해당 디렉토리가 없으면 새로 생성한다. 작업 디렉토리를 지정하면 그 이후 명령어는 해당 디렉토리를 기준으로 동작
- **COPY** : 파일이나 폴더를 이미지에 복사한다. WORKDIR로 지정한 디렉토리를 기준으로 복사
- **ENV** : 이미지에서 사용할 환경 변수 값을 지정한다. 컨테이너를 생성할 때 PROFILE 환경 변수를 따로 지정하지 않으면 local을 기본 값으로 사용
- **ENTRYPOINT** : 컨테이너를 구동할 때 실행할 명령어를 지정한다.

Docker Image Commands

- 도커 이미지 목록 확인
 - `$ docker images`
- 도커 이미지 불러오기
 - 컨테이너 run 시에 이미지가 없으면 Docker Hub로부터 자동으로 Pull
 - `$ docker pull [ImageName:태그]`
- 도커 이미지 삭제
 - `docker image rm [이미지 ID]`
 - `docker image rm -f [이미지 ID]` # 컨테이너를 삭제하기 전에 이미지 삭제
- 도커 모든 이미지 한 번에 삭제
 - `$ docker image rm $(docker images -q)`

Docker Container Commands

- 컨테이너 실행
 - `$ docker run [Options] [Image] [Command]`
- 실행 중인 컨테이너 확인
 - `$ docker ps`
 - `$ docker ps -a` # 정지된 컨테이너 포함
- 컨테이너 시작, 재시작, 종료
 - `$ docker start / restart / stop [컨테이너 이름]`
- 컨테이너 삭제
 - `$ docker container rm [컨테이너 ID]`
- 모든 컨테이너 한번에 삭제 (중지 후 삭제)
 - `$ docker container rm $(docker ps -a -q)`

Docker Build & Push Commands

- Dockerfile로 이미지 생성
 - `docker build --tag [생성할 이미지 이름] : [태그 이름] .`
 - # 맨 마지막의 .(마침표)은 Dockerfile의 위치
 - 이미지 이름은 URL(Docker hub, Cloud Container registry, Private registry)로 시작
- 이미지 Push
 - `$ docker login` # 도커 로그인
 - `$ docker push [이미지 REPOSITORY] : [태그]`
- Docker Hub에서 확인
 - `http://hub.docker.com` (login)

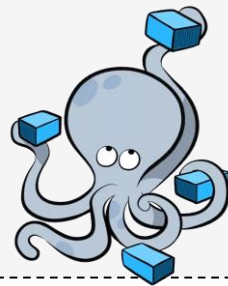
Table of content

Container Orchestration
(Docker & k8s)

1. MSA, Container, and Container Orchestration
2. Docker Hands-on Lab
3. Kubernetes Basic Object Model : Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret, Liveness/Readiness ✓
4. Kubernetes Advanced Lab : Ingress, Job, Cron Job, DaemonSet, StatefulSet
5. Real MSA Application Deployment
6. Kubernetes Architecture
7. Service Mesh: Istio for Advanced Services Control ✓
8. Course Test

Container Orchestration Features

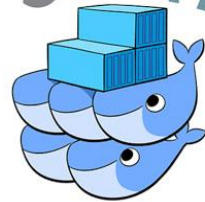
- 컨테이너 자동 배치 및 복제
- 컨테이너 그룹에 대한 로드 밸런싱
- 컨테이너 장애 복구
- 클러스터 외부에 서비스 노출
- 컨테이너 확장 및 축소
- 컨테이너 서비스간 인터페이스를 통한 연결



MESOS



kubernetes



Container Orchestrators

Kubernetes

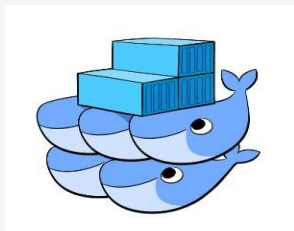


구글에서 개발, 가장 기능이 풍부하며 널리 사용되는 오케스트레이션 프레임워크

베어 메탈, VM환경, 퍼블릭 클라우드 등의 다양한 환경에서 작동하도록 설계

컨테이너의 롤링 업그레이드 지원

Docker Swarm



여러 개의 Docker 호스트를 함께 클러스터링 하여 단일 가상 Docker 호스트를 생성

호스트 OS에 Agent만 설치하면 간단하게 작동하고 설정이 용이

Docker 명령어와 Compose를 그대로 사용 가능

Apache Mesos



수만 대의 물리적 시스템으로 확장할 수 있도록 설계

Hadoop, MPI, Hypertable, Spark같은 응용 프로그램을 동적 클러스터 환경에서 리소스 공유와 분리를 통해 자원 최적화 가능

Docker 컨테이너를 적극적으로 지원

Kubernetes

*“ **Kubernetes** is an open-source system
for automating deployment, scaling, and management
of containerized applications.”*

“ **Kubernetes**는 컨테이너화된 어플리케이션을
자동으로 배포, 조정, 관리할 수 있는 오픈소스 플랫폼이다.

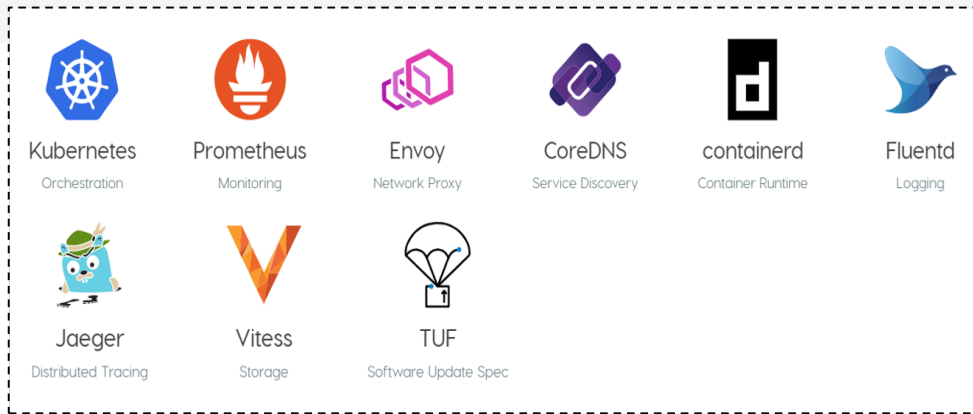
- From Kubernetes Website

Kubernetes Origin

- Borg System 영향을 받아, 2014년 구글의 의해 처음 발표
- 2015년 7월 21일, v1.0 출시
- 리눅스재단과 Cloud Native Computing Foundation(CNCF) 설립
- Kubernetes를 seed 테크놀로지(seed technology)로 제공



<https://www.cncf.io/>



Kubernetes key Features (1/2)

- **Automatic binpacking**

- 각 컨테이너가 필요로 하는 CPU와 메모리(RAM)를 쿠버네티스에게 요청하면, 쿠버네티스는 컨테이너를 노드에 맞추어서 자동으로 스케줄링

- **Self-healing**

- Kubernetes는 실패한 노드에 대해, 컨테이너를 자동으로 교체하고, 재 스케줄링하며, 또한 Health check에 반응하지 않는 컨테이너를 정해진 규칙에 따라 다시 시작

- **Horizontal scaling**

- Kubernetes는 CPU 및 메모리와 같은 리소스 사용량을 기반으로 애플리케이션을 자동으로 확장 할 수 있으며, 메트릭을 기반으로 하는 동적 스케일링도 지원

- **Service discovery and load balancing**

- Service Discovery 매커니즘을 위해 Application을 수정할 필요가 없으며, K8s는 내부적으로 Pod에 고유 IP, 단일 DNS를 제공하고 이를 이용해 load balancing

Kubernetes key Features (2/2)

- **Automated rollouts and rollbacks**

- Application, Configuration의 변경이 있을 경우 전체 인스턴스의 중단이 아닌 점진적으로 Container에 적용(rolling update) 가능
- Release revision이 관리되고 새로운 버전의 배포시점에 문제가 발생할 경우, 자동으로 이전의 상태로 Rollback 진행

- **Secret and configuration management**

- Kubernetes는 secrets와 Config 정보에 대해 이미지 재빌드없이 변경관리가 가능하고, Github 같은 저장소에 노출시키지 않고도 어플리케이션 내에서 보안정보 공유 가능

- **Storage Orchestration**

- Local storage를 비롯해서 Public Cloud(Azure, GCP, AWS), Network storage등을 구미에 맞게 자동 mount 가능

Kubernetes: 발음하기

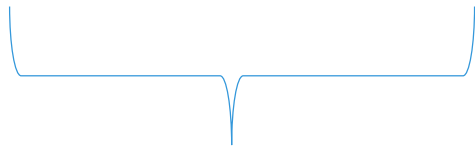
미국식: 큐브네리스

영국식: 쿠버네티스

인도식: 꾸-버네틱스

Kubernetes: 쓰기

Kubernetes



K8S \neq

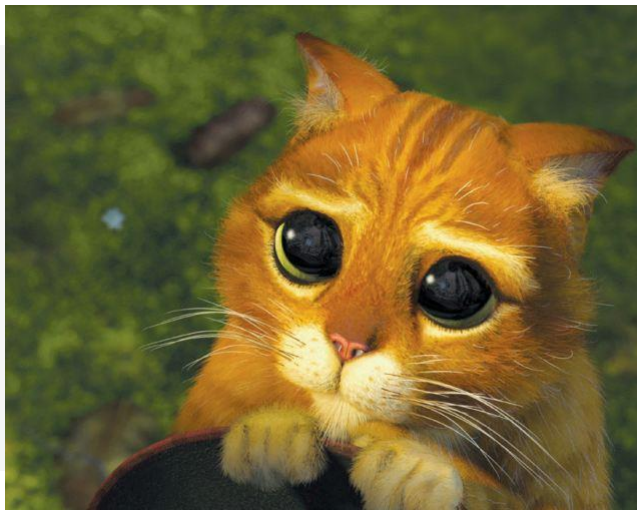




Kubernetes Basic Objects

- “Desired State” declaration
- Pod, Label, ReplicaSet, Deployment, Service, Volume, Configmap, Secret
- Liveness & Readiness

Kubernetes Core Concept : “Desired State”



Declarative Model & Desired States

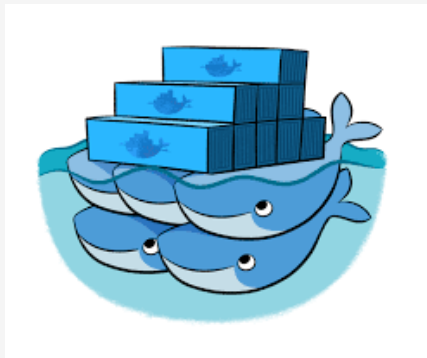
- **Kubernetes는 Current State을 모니터링하면서, Desired State를 유지하려는 습성**
- 직접적인 동작을 명령하지 않고, 원하는 상태를 선언(Not Imperative, But Declarative)
 - Imperative – “nginx 컨테이너를 3개 실행해줘, 그리고 80포트로 열어줘.”
 - Declarative – “80포트를 열어놓은 채로, nginx 컨테이너를 3개 유지해줘.”

Kubernetes Object, Controller and Kubectl

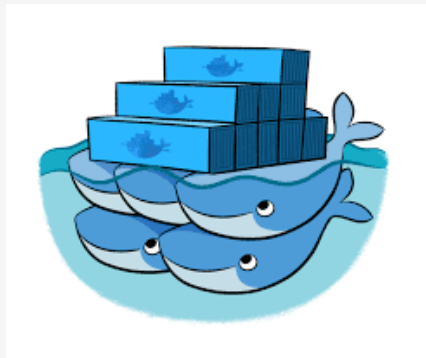
- Object : K8s의 상태를 나타내는 엔티티로 K8s API의 Endpoint
 - 유형 - Pod, Service, Volume, Namespace 등
 - Spec과 Status 필드를 가짐 - Spec(Desired State), Status(Current State)
- Controller : Object의 Status를 갱신하고, Object를 Spec에 정의된 상태로 지속적으로 변화시켜 주는 주체
 - 유형 - ReplicaSet, Deployment, StatefulSets, DaemonSet, Cronjob 등
- Kubectl : Command CLI에서 Object와 Controller를 제어하는 K8s Client
 - 발음하기 - “큐브시티엘”, “쿠베시티엘”

Pod ; Kubernetes 최소 배포 단위

- Pod : 미국식 [pa:d], 영국식 [pɒd]
 - “물고기, 고래” 작은 떼 (Docker의 심볼이 고래 모양에서 유래)
 - 발음하기 : “팟”, “파드”, “포드”



Pod

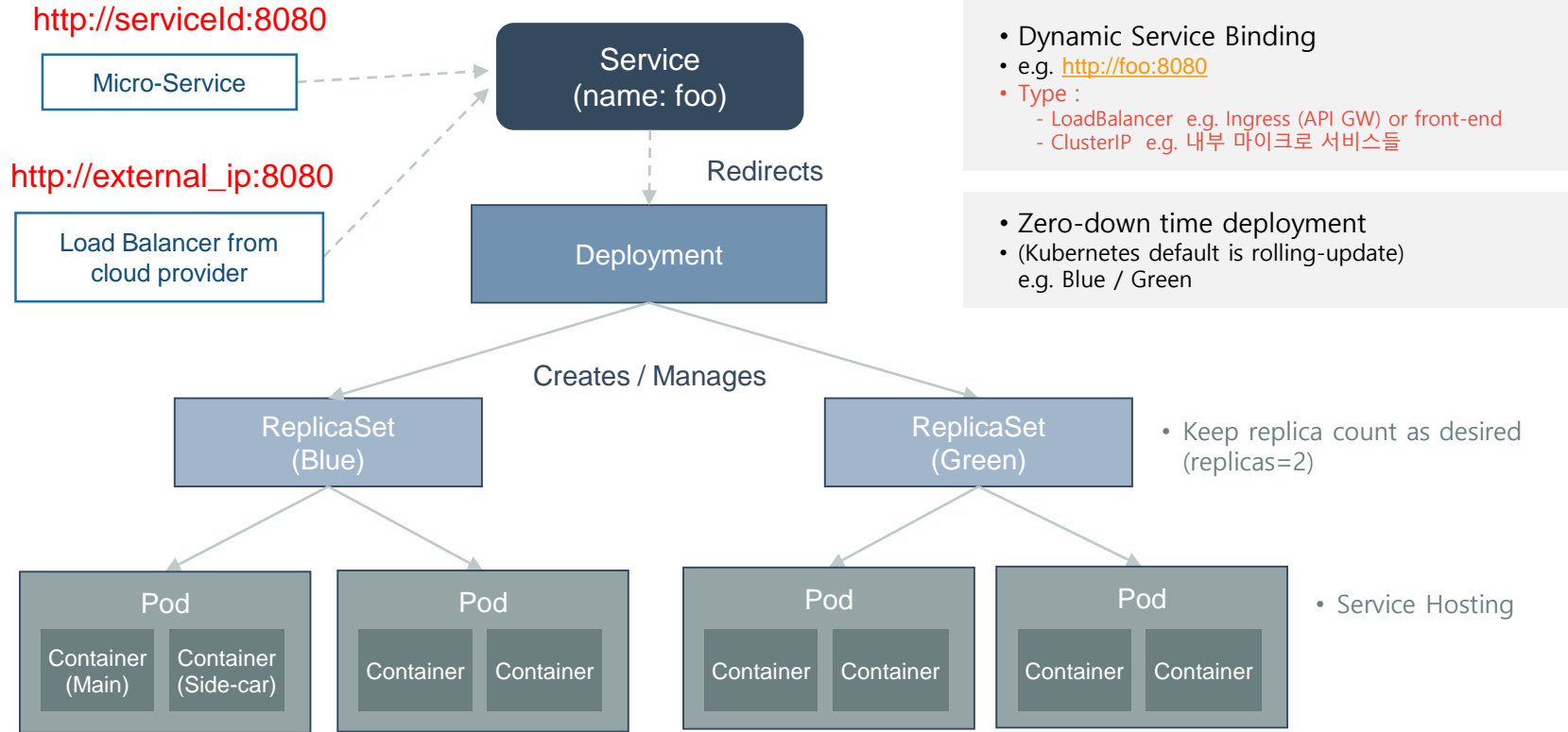


Pod

Pod ; Kubernetes 최소 배포 단위

- 한 Pod은 내부에 여러 컨테이너를 가질 수 있지만 대부분 1~2개의 컨테이너 가짐
 - 스케일링이 컨테이너가 아닌 Pod단위로 수행되기 때문에, Pod내부에 다수의 컨테이너들이 타이트하게 묶여 있으면 스케일링이 쉽지 않거나 비효율적으로 이뤄지는 문제
- 1개의 Pod은 1개의 물리서버(Node) 위에서 실행
- 동일 작업을 수행하는 Pod은 ReplicaSet Controller에 의해 룰에 따라 복제생성
 - 이때 복수의 Pod이 Master의 Scheduler에 의해 여러 개의 Node에 걸쳐 실행
- Pod의 외부에서는 이 'Service' 를 통해 Pod에 접근
 - Service를 Pod에 연결했을 때 Pod의 특정 포트가 외부로 expose

Kubernetes Object Model



Kubernetes Object Model

- **apiVersion** : 해당 Object description 을 해석할 수 있는 API server 의 버전
- **kind** : 오브젝트의 타입 – 예제는 **Deployment**
- **metadata** : 객체의 기본 정보. 예) 이름
- **spec (spec and spec.template.spec)** : 원하는 "Desired State" 의 세부 내역. 예제에서는 3개의 replica를 template 내의 pod 정의대로 찍어내어 유지하라는 desired state 설정임
- **spec.template.spec** : defines the desired state of the Pod. The example Pod would be created using **nginx:1.7.9**.

Once the object is created, the Kubernetes system attaches the **status** field to the object

```
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:1.7.9
          ports:
            - containerPort: 80
```

Deployment object Example

Declaration based configuration

> my-app.yml

> Desired state



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
```

```
...
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80
```

```
---
apiVersion: apps/v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: nginx
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
  labels:
    app: backend
spec:
  replicas: 3
```

```
...
template:
  metadata:
    labels:
      app: backend
  spec:
    containers:
      - name: backend
        image: backend:latest
        ports:
          - containerPort: 8080
```

```
---
apiVersion: apps/v1
kind: Service
metadata:
  name: backend-service
  labels:
    app: backend
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: db-deployment
  labels:
    app: db
spec:
  replicas: 2
```

```
...
template:
  metadata:
    labels:
      app: db
  spec:
    containers:
      - name: db
        image: mongo
        ports:
          - containerPort: 27017
```

```
---
apiVersion: apps/v1
kind: Service
metadata:
  name: mongo-service
  labels:
    app: mongo
```

Lab. K8s Sample App 생성 – Voting App



Lab Time

- Lab Script Location
 - Workflowy :

Lab. 기본적인 kubectl 명령어

- 객체 목록 불러오기
 - “`kubectl get [객체 타입]`“ Ex) `kubectl get pods` : pod 목록을 불러온다.
- 객체 삭제하기
 - “`kubectl delete [객체 타입] [객체 이름]`“
Ex) “`kubectl delete pods wordpress-5bb5dddcff-kwgf8`”
 - “`kubectl delete [객체타입,객체타입,...] --all`“
Ex) “`kubectl delete services,depeloyments,pods --all`”
 - 실습 중에 잘못 생성되었거나 초기화가 필요할 경우 사용
 - 콤마(,)로 구분하며 붙여 적기
- 객체 상세 설명 확인하기
 - “`kubectl describe [객체 타입] [객체 이름]`“
Ex) “`kubectl describe service wordpress`”

Lab. 이미지를 통한 어플리케이션 배포

- 현재 작동 중인 pod들이 있는지 확인
 - `kubectl get pods`
- Nginx 이미지 예제로 배포하기
 - `kubectl create deploy first-deployment --image=nginx`
- 실행된 Pods 확인
 - `kubectl get pods`
 - 각 Pod들은 'Pod명-{hash}' 형식의 고유한 이름을 가짐

Lab. Pod에 접속하기

- 로그 보기
 - `kubectl logs [복사된 pod 이름] -f`
- 복사된 pod이름으로 접속하기
 - `kubectl exec -it [복사된 pod 이름] -- /bin/bash`
- Pod 내에서 접속하기 (위 명령어로 진입후에는 리눅스 Shell 명령어 사용)
 - 해당 경로에 있는 html 파일을 호출하는 어플리케이션이다.
`echo Hello nginx`
 - Curl 명령어를 사용하기 위한 업데이트를 한다. (Curl 명령이 없으면 설치)
`apt-get update`
`apt-get install curl`
 - Curl 명령어로 호출하기
`curl localhost`

설정 파일을 통한 pod 배포 (1/2)

- 아래 내용으로 nano editor 를 이용하여 declarative-pod.yaml 파일을 생성
 - Nginx 이미지를 기반으로 pod를 배포하는 설정파일

```
apiVersion: v1
kind: Pod
metadata:
  name: declarative-pod
spec:
  containers:
    - name: memory-demo-ctr
      image: nginx
```


설정 파일을 통한 pod 배포 (2/2)

- nano declarative-pod.yaml 파일을 직접 작성 후 배포
 - `kubectl create -f declarative-pod.yaml`
(-f 는 파일 경로를 설정해서 배포할 수 있는 옵션이다.)
- 배포된 Pod를 검색 후 접속
 - 이름이 설정대로 declarative-pod 로 생성됨을 확인
`kubectl get pods`
 - 생성된 Pod에 접속
`kubectl exec -it declarative-pod -- /bin/bash`
`apt-get update`
`apt-get install curl`
`curl localhost`

원하는 노드 타입에 Pod 몰기 (1/2)

- Node를 확인하여 label 붙이기
 - `kubectl get nodes`
`kubectl label nodes [노드이름] disktype=ssd`
- Label과 함께 노드 목록 불러오기
 - `kubectl get nodes --show-labels`
- 설정 파일생성(오른쪽 내용)
 - `nano dev-pod.yaml`
- 설정파일을 기반으로 pod 배포
 - `kubectl create -f dev-pod.yaml`
- 생성된 pod의 상세 내용 확인
 - `kubectl get pods -o wide`

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

#여기가 중요

원하는 노드 타입에 Pod 몰기 (2/2)

- Pod가 Create 되지 않고, Pending 상태
- Node에 Label 추가
 - `kubectl get nodes`
 - `kubectl label nodes <your-node-name> disktype=ssd`
 - `kubectl get nodes --show-labels | grep ssd`
- Pod 확인
 - `kubectl get all`
- Node 정보와 함께 Pod 확인
 - `kubectl get po -o wide`

Pod Initialization

- Init.yaml 파일 생성
 - `nano pod-initialize.yaml`
- Pod 생성
 - `kubectyl create -f init.yaml`
- Pod 접속하기
 - `kubectyl exec -it init-demo -- /bin/bash`
 - `cd /usr/share/nginx/html`
 - `cat index.html`

```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: workdir
          mountPath: /usr/share/nginx/html
  initContainers:
    - name: install
      image: busybox
      command:
        - wget
        - "-O"
        - "/work-dir/index.html"
        - "http://kubernetes.io"
      volumeMounts:
        - name: workdir
          mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
    - name: workdir
      emptyDir: {}
```

← Pod 초기화 시점에
실행

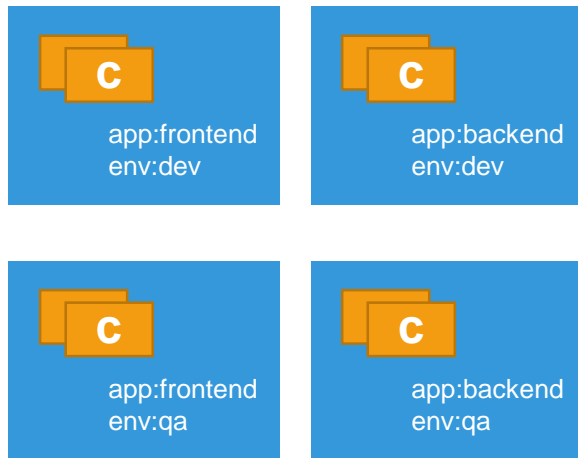
Lab. Pod



Lab Time

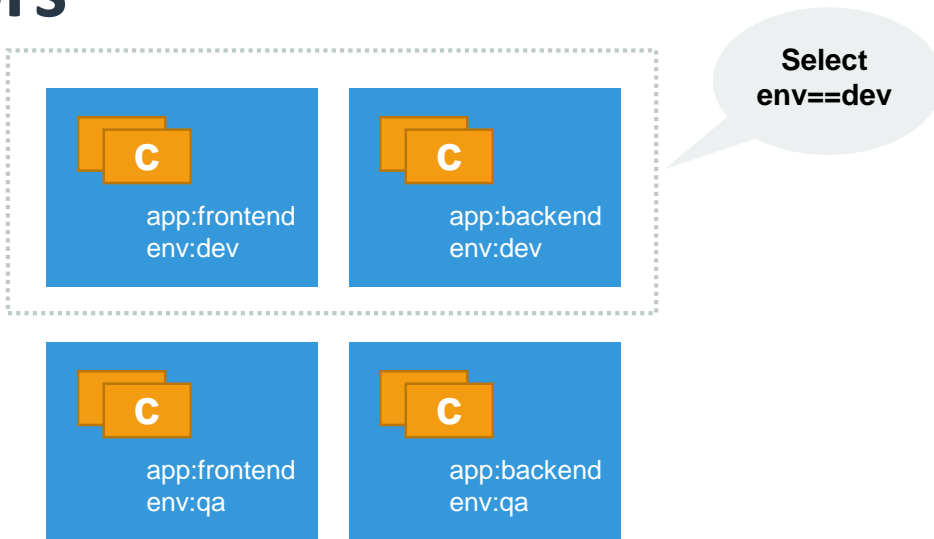
- Lab Script Location
 - Workflowy :

Labels



- Labels 은 객체 식별 정보로서 Kubernetes 객체라면 모두 붙일 수 있다.
- Label들은 요구사항에 맞춰 개체의 하위 집합을 구성하고 선택하는데 사용된다.
- Label들은 객체에 고유성을 제공하지 않아, 여러 객체들은 같은 label을 가질 수 있다.

Label Selectors



- Label Selectors들은 객체들의 집합을 선택하며, kubernetes는 2가지 종류를 지원한다.
 - **Equality-Based Selectors**
Uses the `=`, `==`, `!=` 연산자를 이용하여 Label key와 value 값을 기반으로 객체들을 필터링 할 수 있다.
 - **Set-Based Selectors**
`in`, `notin`, `exist` 연산자를 사용하며, value 값들을 기반으로 객체들을 선택할 수 있다.

Lab. Label

- `kubectl get po` # 실행 중, Pod list 확인
- `kubectl edit po <pod 명>` # Pod 인스턴스에 Label 추가

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-02-16T11:22:56Z"
  labels:
    env: test
  name: init-demo
  namespace: default
  resourceVersion: "588586"
```

vi 에디터로 편집

- `kubectl get pods -l env=test`
- `kubectl get pods --selector env=test`
- `kubectl get pods --selector 'env in (test, test1)'` # or 연산
- `kubectl get po --selector 'env in(test, test1), app in (nginx, nginx1)'` # and 연산
- `kubectl get po --selector 'env,app notin(nginx)'` # env가 있으면서, app이 nginx가 아닌 Pod

Annotations

- Label 처럼 식별 정보는 아닌 임의의 비 식별 메타데이터를 객체에 key-value 형태로 추가

```
"annotations": {  
  "key1" : "value1",  
  "key2" : "value2"  
}
```

- 주로, 히스토리, 스케줄 정책, 부가 정보 등을 기술
- 배포 주석을 추가해 서비스를 Deploy하고, 이전 서비스로 롤백 시, 해당 정보를 활용해 롤백

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
  labels:  
    app: nginx  
  annotations:  
    kubernetes.io/change-cause=nginx:1.7.9  
spec:  
  replicas: 3  
  selector:  
    .....
```



One-dash, Double-dash

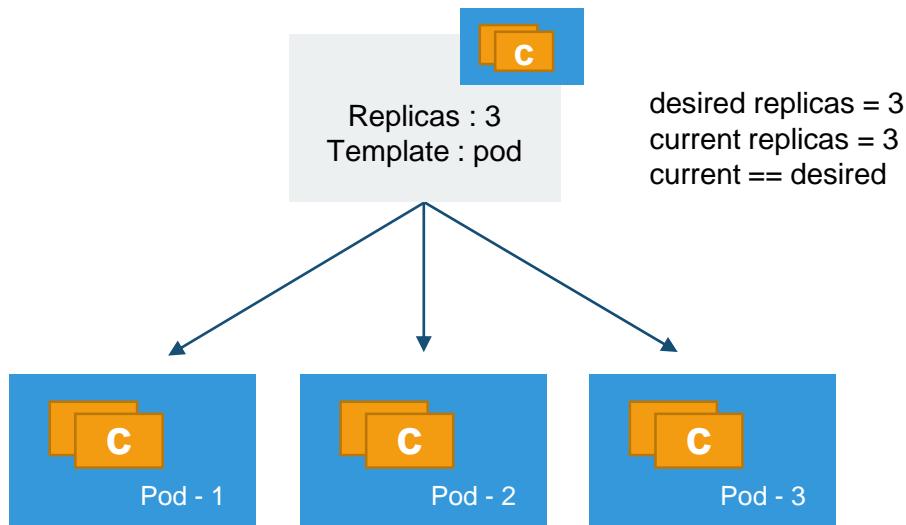
- Generally, in Command Options
 - Some have a long form without a short form (-- author, --block-size)
 - Some have a short form without a long form (-c, -f, -g)
 - Some have both a long form and a short form (-A / --almost-all, -b / --escape)
- In Pod Selector, short form (-l) equals long form(--selector)

Replication Controllers

- Master node의 Controller Manager 중 하나
- Pod의 복제품이 주어진 개수(Desired State)만큼 작동하고 있는지 확인하고 개수를 조절한다.
- Replication Controller는 Pod를 생성하고 관리한다.
 - 일반적으로 Pod는 자기 복구가 불가능 하기에 단독으로 배포를 하지 않는다.

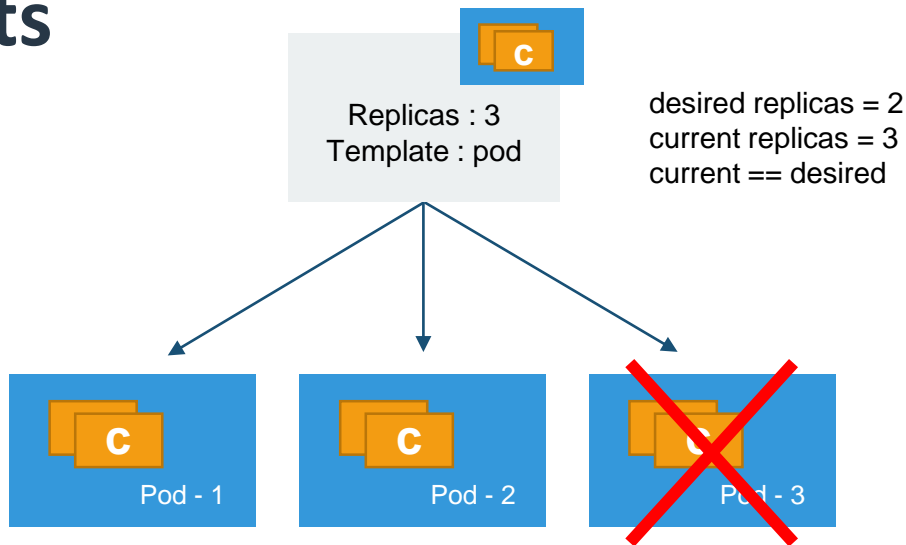


ReplicaSets



- ReplicaSet(rs)은 Replication Controller의 업그레이드 버전
- ReplicaSet은 equal 및 set 기반 Selector를 모두 지원하는 반면, Replication Controller는 equal기반 Selector만 지원

ReplicaSets



- 지정된 수의 Pod (Desired State)가 항상 실행되도록 보장
- ReplicaSets은 단독으로도 사용 가능하지만, 주로 Pod Orchestration에 사용(Pod creation, deletion, updates)
- Deployment가 ReplicaSets을 자동 생성하기 때문에 사용자는 관리에 신경 쓰지 않아도 됨

Lab. ReplicaSet

- ReplicaSet 파일 생성
 - `nano replicaset.yaml`
- 파일을 기반으로 ReplicaSet 배포
 - `kubectl create -f frontend.yaml`
- ReplicaSet을 확인
 - `kubectl get pods`
 - `kubectl describe rs/frontend`
- ReplicaSet을 삭제
 - `kubectl delete rs/frontend`
(관련된 pod 전부 제거한다.)
 - `kubectl delete rs/frontend --cascade=false`
(ReplicaSet 은 제거하지만 Pod 는 유지)

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
          ports:
            - containerPort: 80
```

Lab. ReplicaSet

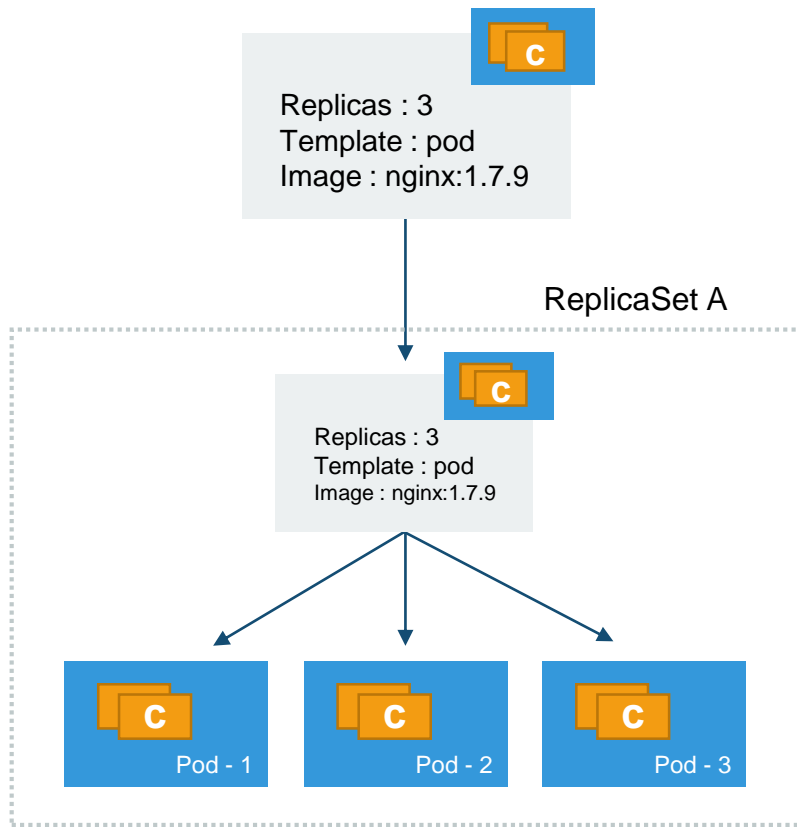


Lab Time

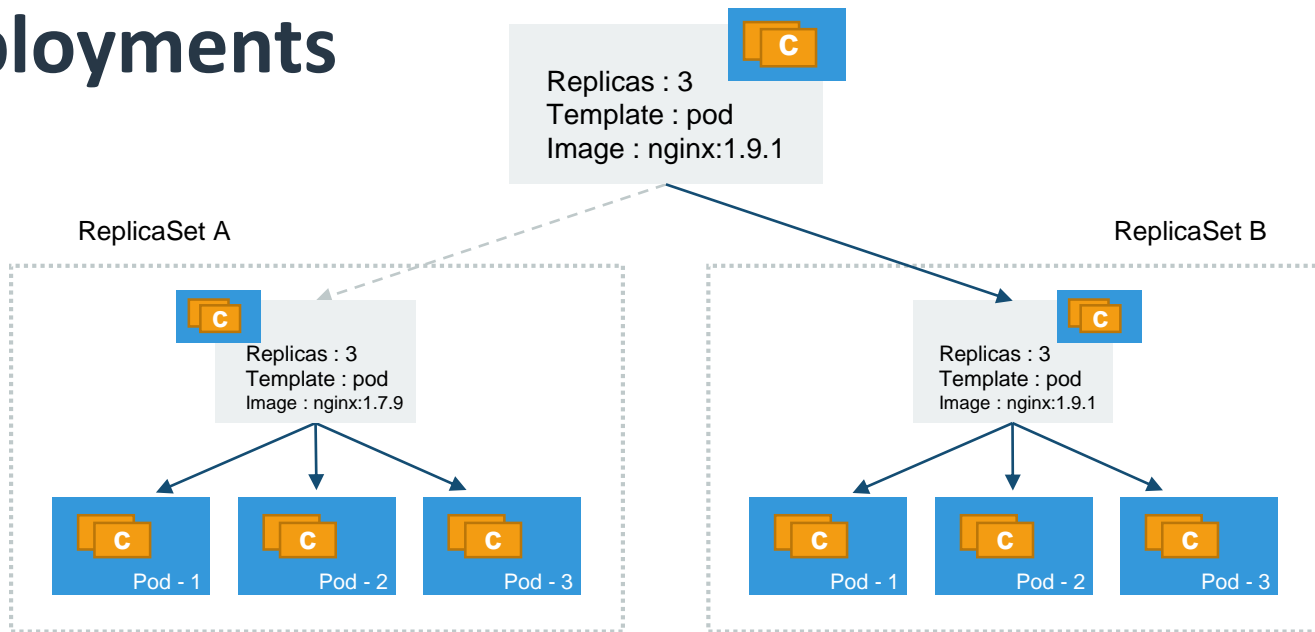
- Lab Script Location
 - Workflowy :

Deployments

- Deployment 객체는 Pods와 ReplicaSets에 대한 선언적 업데이트를 제공한다.
- Deployment Controller는 Master node 컨트롤러의 일부로 Desired state가 항상 만족이 되는지 확인한다.
- Deployment 가 ReplicaSet을 만들고 ReplicaSet은 그 뒤에 주어진 조건만큼의 Pod들을 생성한다.



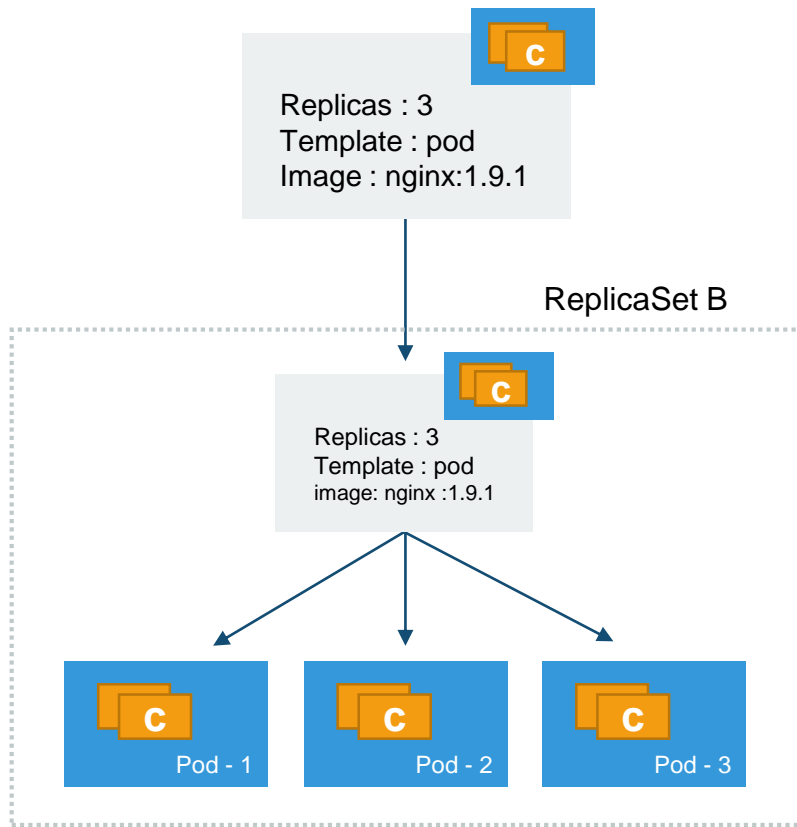
Deployments



- Deployment의 Pod template이 바뀌게 되면, 새로운 ReplicaSet이 생성되는데, 이를 **Deployment rollout**이라고 한다.
- Rollout은 Pod template에 변동이 생겼을 경우에만 동작하며, Scaling등의 작업은 ReplicaSet을 새로 생성하지 않는다.

Deployments

- 새로운 ReplicaSet이 준비되면 Deployment는 새로운 ReplicaSet을 바라본다.
- Deployment들은 Deployment recording 등의 rollback 기능을 제공하며, 문제가 발생했을 경우, 이전 단계로 돌릴 수 있다.



Deployment 생성

- 설정 파일을 생성
 - nano nginx-deployment.yaml
- 파일을 기반을 배포
 - `kubectl create -f nginx-deployment.yaml`
- 생성된 deployment 확인
 - `kubectl describe deployment nginx-deployment`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

← Pod 생성 시, 이 템플릿 참조
도커허브에서 nginx 1.7.9
이미지를 가져와 'nginx' 이름의
컨테이너 생성

Scaling Deployments

- Deployment의 replica의 개수를 확인한다.
 - `kubectl get pods`
- Deployment의 이름을 복사한다.
 - `kubectl get deployments`
- 해당 Deployment의 scale 조정
 - `kubectl scale deployments [deployment 이름] --replicas=3`
- 변경을 확인
 - `kubectl get pods`

Deployments 의 변경

- Deployment 파일을 변경
 - `kubectl get deployments`
 - `nano nginx-deployment.yaml`
(spec 아래 항목에 `replicas: 5` 속성 추가; 있으면 수정)
- 변경한 파일을 적용
 - `kubectl apply -f nginx-deployment.yaml`
- 변경 내용을 확인
 - `kubectl get pods`
- 설정파일에 추가된 `replicas` 속성을 삭제 후 다시 적용
 - `Nano nginx-deployment.yaml (replicas)`
 - `kubectl apply -f nginx-deployment.yaml`
 - `kubectl delete pods --all`
 - `kubectl get pods`

Rolling update

- 작동 중인 pod와 deployments 확인
 - `kubectl get pods`
`kubectl get deployments`
- 새로운 버전의 deployments 배포 및 배포 상태 확인
 - `kubectl apply -f nginx-deployment.yaml`
 - `kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1`
 - `kubectl rollout status deployment/nginx-deployment`
- 변경 확인
 - `kubectl get deployments`
`kubectl get pods`
`kubectl describe pods [pod 이름]`
- Pod에 접속하여 확인
 - `kubectl exec -it podname -- /bin/bash`
`apt-get update`
`apt-get install curl`
`curl localhost`

Rollback

- 현재 실행중인 객체들을 확인

- `kubectl get pods`

- `kubectl get deployments -o wide`

deployment에 적용된 Image:버전 추가 표시

- 객체를 롤백 처리

- `kubectl rollout undo deployment/nginx-deployment`

- 진행을 확인

- `kubectl get deployments -o wide`

Lab. Deployments



Lab Time

- Lab Script Location
 - Workflowy :

Namespaces

- Kubernetes는 동일 물리 클러스터를 기반으로 하는 복수의 가상 클러스터를 지원하는데 이들 가상 클러스터를 Namespace라고 한다.
- Namespace를 활용하면, 팀이나 프로젝트 단위로 클러스터 파티션을 나눌 수 있다.
- Namespace 내에 생성된 자원/객체는 고유하나, Namespace사이에는 중복이 발생할 수 있다.



Namespaces

- Namespaces Object 조회

```
$ kubectl get namespaces
```

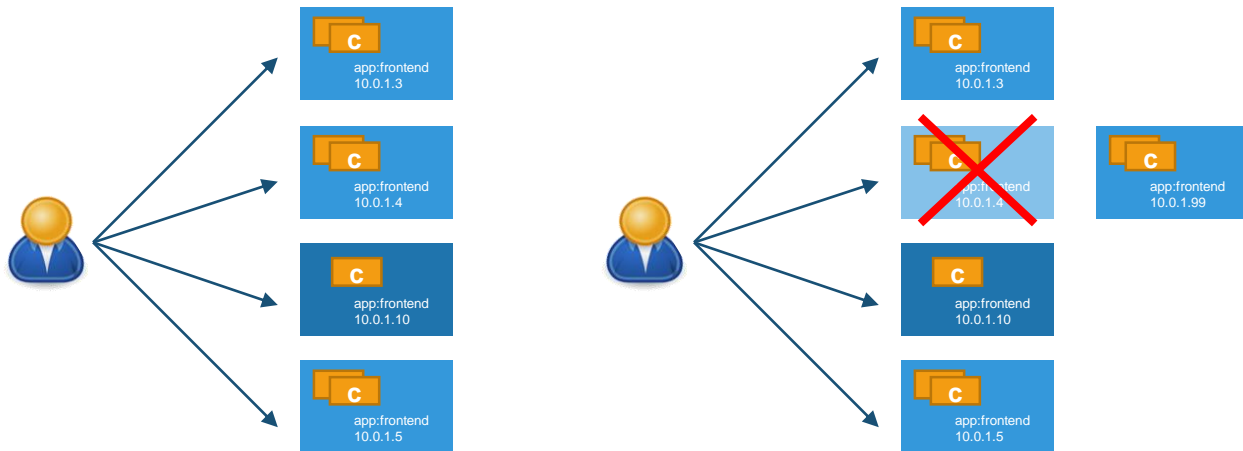
NAME	STATUS	AGE
default	Active	11h
kube-public	Active	11h
kube-system	Active	11h

- Kubernetes는 처음에 3개의 초기 네임스페이스를 가진다.
 - default** : 다른 namespace를 갖는 다른 객체들을 가지고 있다.
 - kube-public** 은 클러스터 bootstrapping 같은 모든 유저가 사용가능한 특별한 namespace 이다.
 - kube-system**: Kubernetes system에 의해서 생성된 객체를 가지고 있다.
- Resource Quotas를 사용하여 namespace 내에 존재하는 자원들을 나눌 수 있다.

Lab. Namespaces

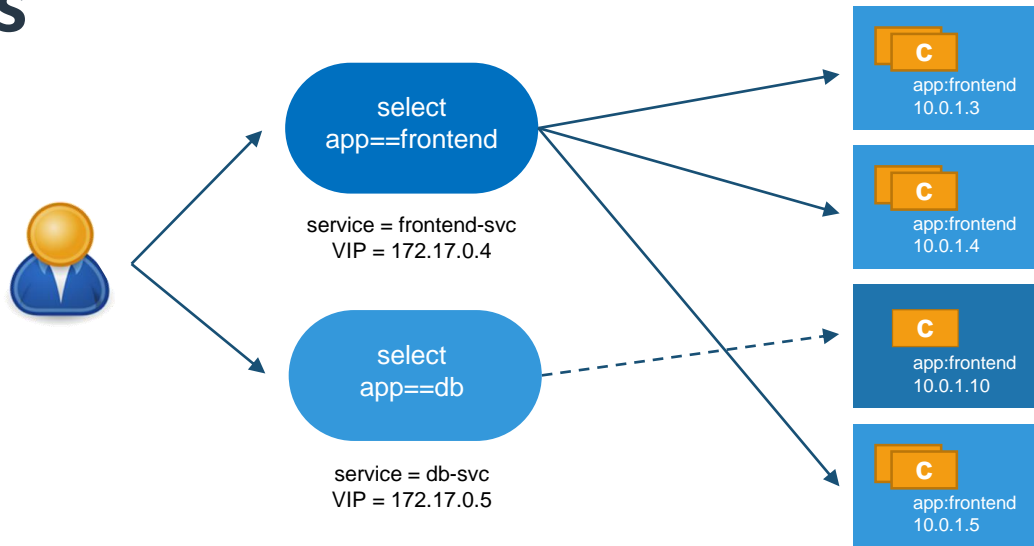
- 원하는 Namespace에 Pod 생성
 - `kubectl create deploy nginx --image=nginx --namespace=<insert-namespace-name-here>`
- 특정 Namespace상에 생성된 Pod 조회
 - `kubectl get po --namespace=<insert-namespace-name-here>`
- Namespace Option 생략 시, default Namespace

Pod Access Issues



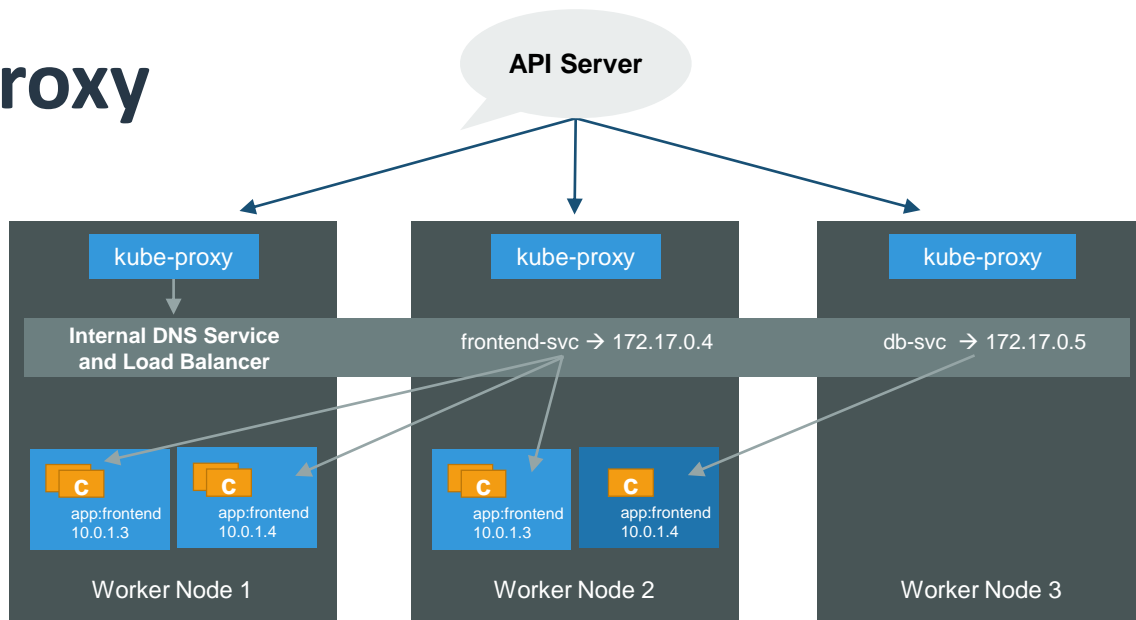
- 어플리케이션에 접근하기 위해서는, 사용자가 Pod에 접근해야 한다.
- Pod들은 언제든지 소멸 가능하기에 IP주소는 고정되어 있지 않다.
- 사용자가 직접 IP주소로 Pod에 연결되어 있을 때, Pod가 죽어서 새로 만들어지면 접속할 방법이 없다.
- 이 상황을 극복하기 위해서 추상화를 통해 Service라는 Pod들의 논리적 집단을 만들어 규칙을 설정하고 사용자들은 여기에 접속을 한다.

Services



- Selector를 사용하여 Pod를 논리적 그룹으로 나눌 수 있다.
- 각 논리적 그룹에 대해서 Service name이라는 이름을 부여할 수 있다.
- 사용자는 Service IP주소를 통해 Pod에 접속하게 된다.
 - 각 Service에 부여된 IP는 클러스터 IP 라고도 부른다.
- Service는 각 Pod에 대해 Load balancing을 자동으로 수행

kube-proxy



- 모든 Worker node들은 kube-proxy라는 데몬을 실행하며, 이 데몬은 Service 및 End-point 생성 /삭제를 위해 마스터 노드의 API 서버를 모니터링한다.
- 각 node에 있는 모든 새로운 Service에 대해서 kube-proxy는 Iptable 규칙을 구성하여, ClusterIP의 트래픽을 캡처하고 이를 End-point로 보낸다.
- Service가 제거되면, kube-proxy는 노드상의 Iptables 규칙도 지운다.

Service Discovery

- Service는 클라이언트가 애플리케이션에 접근하는 Kubernetes의 채널로 런타임시, 이를 검색할 수 있는 방법이 필요
- DNS를 이용하는 방법
 - 서비스는 생성되면, [서비스 명].[네임스페이스명].svc.cluster.local이라는 DNS명으로 쿠버네티스 내부 DNS에 등록되고, 쿠버네티스 내부 클러스터에서는 이 DNS명으로 접근 가능한데, 이 때 DNS에서 리턴해 주는 IP는 외부 IP(External IP)가 아니라 Cluster IP(내부 IP) 임
- External IP를 명시적으로 지정하는 방법
 - 외부 IP는 Service의 Spec 부분에서 externalIPs 항목의 Value로 기술

ServiceType

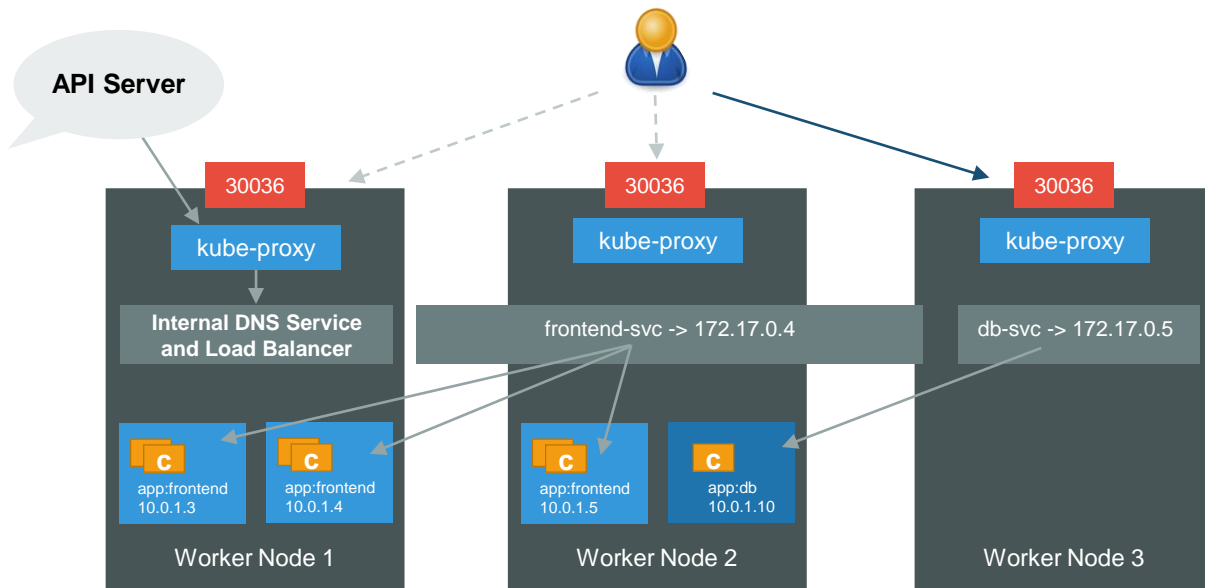
- Service를 정의할 때 Access Scope를 따로 정할 수 있다.
 - 클러스터 내에서만 접근이 가능한가?
 - 클러스터 내와 외부에서 접근이 가능한가?
 - 클러스터 밖의 리소스에 대한 Map을 가지는가?
- Service 생성 시, IP주소 할당 방식과 서비스 연계 등에 따라 4가지로 구분
 - ClusterIP
 - NodePort
 - LoadBalancer
 - ExternalName

ServiceType : ClusterIP

- 디폴트 설정으로 서비스에 클러스터 ip를 할당
- 쿠버네티스 클러스터 내에서만 이 서비스에 접근 가능
- 외부에서는 외부 IP를 할당 받지 못했기 때문에 접근이 불가능

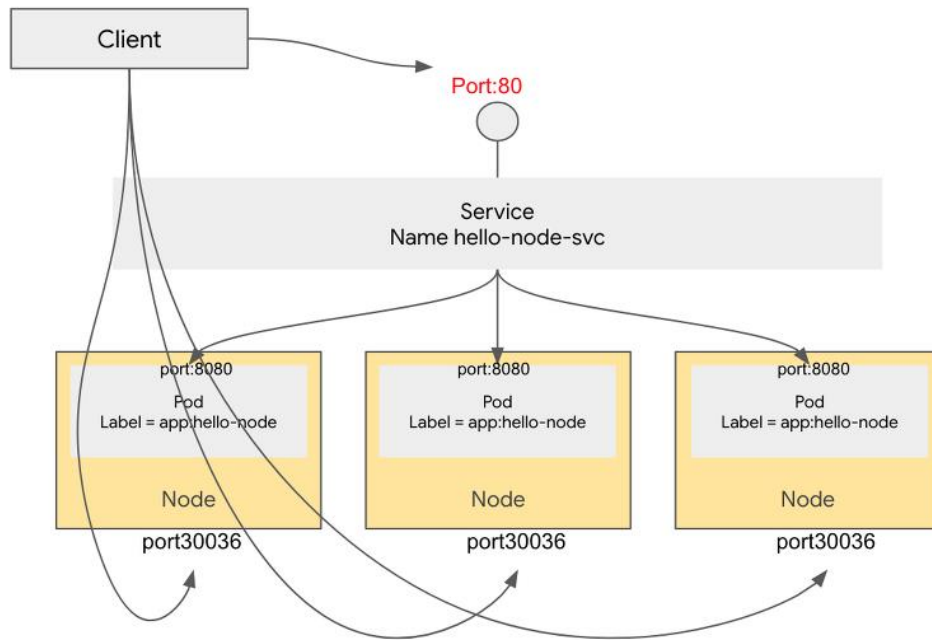
ServiceType : NodePort

- 고정 포트(NodePort)로 각 노드의 IP에 서비스를 노출
- Cluster IP 뿐만 아니라, 노드의 IP와 포트를 통해서도(<NodeIP>:<NodePort>) 접근 가능



ServiceType : NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: hello-node-svc
spec:
  selector:
    app: hello-node
  type: NodePort
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8080
      nodePort: 30036
```

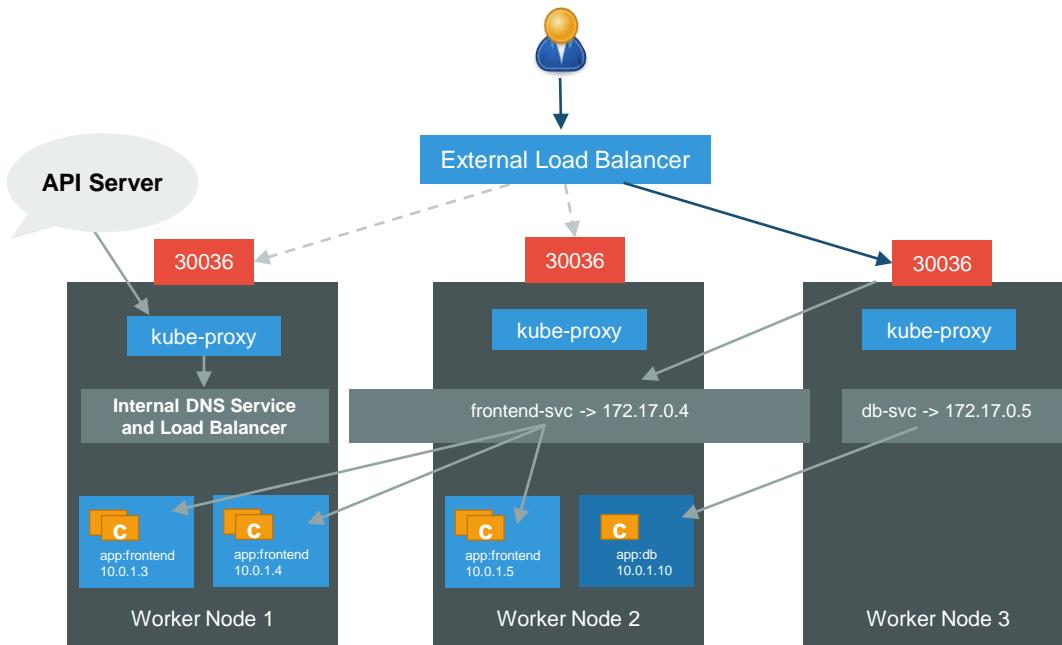


ServiceType : LoadBalancer

- 클라우드 밴더의 로드밸런싱 기능을 사용
 - NodePort와 ClusterIP Service들은 자동으로 생성되어 External Load Balancer가 해당 포트로 라우팅
 - Service들은 각 Worker node에서 Static port로 노출

LoadBalancer ServiceType은 기본 인프라가 Load balancer의 자동 생성을 제공하고, Kubernetes를 지원 할 경우에만 작동

Ex) Azure, Google Cloud Platform, AWS

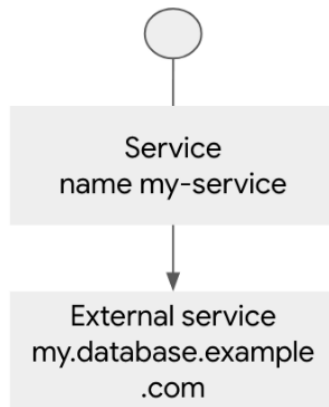


ServiceType: ExternalName

- 외부 서비스를 쿠버네티스 내부에서 호출할 때 사용
- 쿠버네티스 클러스터내의 Pod들은 클러스터 IP를 가지고 있기 때문에 클러스터 IP 대역 밖의 서비스를 호출하고자 하면, NAT 설정등 복잡한 설정이 필요
- Azure, AWS 나 GCP와 같은 클라우드 환경에서 DBMS나, 또는 클라우드에서 제공되는 메시징 서비스 (RDS, CloudSQL)등을 사용하고자 할 경우
 - 쿠버네티스 클러스터 외부이기에 호출이 어려운 경우가 있는데, 이를 쉽게 해결할 수 있는 방법이 ExternalName 타입

ServiceType: ExternalName

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



- 서비스를 ExternalName 타입으로 설정하고, 주소를 my.database.example.com으로 설정해주면 이 my-service는 들어오는 모든 요청을 my.database.example.com 으로 포워딩
- 일종의 프록시와 같은 역할

Lab. Service

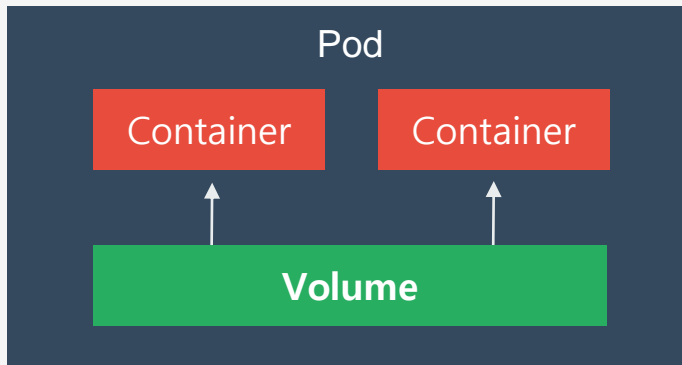


Lab Time

- Lab Script Location
 - Workflowy :

Volumes

- 쿠버네티스는 여러 호스트에 걸쳐 Stateless한 컨테이너를 마이크로서비스로 배포하는 것이 목표이기에 영속성 있는 저장장치(Persistent Volume)를 고려해야 함



- Volume은 Pod에 장착되어, 그 Pod에 있는 Container 간에 공유


Types of Volumes

- Pod에 마운트된 디스크는 Volume Type에 따라 사용 유형이 정의
- Volume Type 內 디스크의 크기, 내용 등의 속성 설정
- Types of Volumes

임시 볼륨	로컬 볼륨	네트워크 볼륨	네트워크 볼륨 (Cloud dependent)
emptyDir	hostPath	gitRepo, iSCSI, NFS cephFS, glusterFS	gcePersistentDisk, AzureDisk, Amazon EFS, Amazon EBS ...
✓ Pod내 컨테이너간 공유	✓ Host 디렉토리를 Pod와 공유 해 사용하는 방식	✓ 영구적으로 영속성 있는 데이터 관리 목적	
✓ Pod가 삭제되면, emptyDir도 지워지므로 휘발성 데이터 저장 용도	✓ 컨테이너에 nodeSelector를 지정안하면 매번 다른 호스 트에 할당 ✓ e.g. 호스트의 Metric 수집해 야 하는 경우	✓ 쿠버네티스는 PV와 PVC의 개념을 통해 Persistent 볼 륨을 Pod에 제공 ✓ gitRepo, iSCSI, NFS와 같은 표준 네트워크 볼륨과 Cloud Vendor가 제공하는 볼륨으로 구분	

Volumes : emptyDir

```
apiVersion: v1
kind: Pod
metadata:
  name: shared-volumes
spec:
  containers:
    - image: redis
      name: redis
      volumeMounts:
        - name: shared-storage
          mountPath: /data/shared
    - image: nginx
      name: nginx
      volumeMounts:
        - name: shared-storage
          mountPath: /data/shared
  volumes:
    - name: shared-storage
      emptyDir: {}
```

- emptyDir의 생명주기는 컨테이너 단위가 아닌 Pod 단위로 Container 재기동에도 계속 사용 가능
- 생성된 Pod 확인


NAME	READY	STATUS	RESTARTS	AGE
pod/shared-volumes	2/2	Running	0	10m
- 지정 컨테이너 접속 후, 파일 생성
 - `kubect exec -it shared-volumes --container redis -- /bin/bash`
 - `cd /data/shared`
 - `echo test... > test.txt`
- 다른 컨테이너로 접속 후, 파일 확인
 - `kubect exec -it shared-volumes --container ngx -- /bin/bash`
 - `cd /data/shared`
 - `ls`

Volumes : hostPath

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: somepath
          mountPath: /data/shared
  volumes:
    - name: somepath
      hostPath:
        path: /tmp
        type: Directory
```

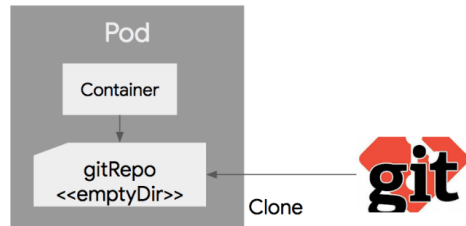
- Node의 Local 디스크 경로를 Pod에 마운트
- 같은 hostPath에 있는 볼륨은 여러 Pod사이에서 공유
- Pod가 삭제되어도 hostPath에 있는 파일은 유지
- Pod가 재기동 되어 다른 Node에서 기동될 경우, 새로운 Node의 hostPath를 사용
- Node의 로그 파일을 읽는 로그 에이전트 컨테이너 등에 사용가능
- Pod 생성 및 확인 (Pod 내, ls -al /data/shared)

```
drwxrwxrwt 8 redis root 4096 Feb 17 03:08 .
drwxr-xr-x 3 redis redis 4096 Feb 17 03:07 ..
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .ICE-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .Test-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .X11-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .XIM-unix
drwxrwxrwt 2 redis root 4096 Feb 11 05:39 .font-unix
drwx----- 3 redis root 4096 Feb 11 05:44 systemd-private-fe55104f60e34b2ea4
```

Volumes example : gitRepo

```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepo-volume-pod
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
      readOnly: true
    ports:
    - containerPort: 80
      protocol: TCP
  volumes:
  - name: html
    gitRepo:
      repository: https://github.com/luksa/kubia-website-example.git
      revision: master
      directory: .
```

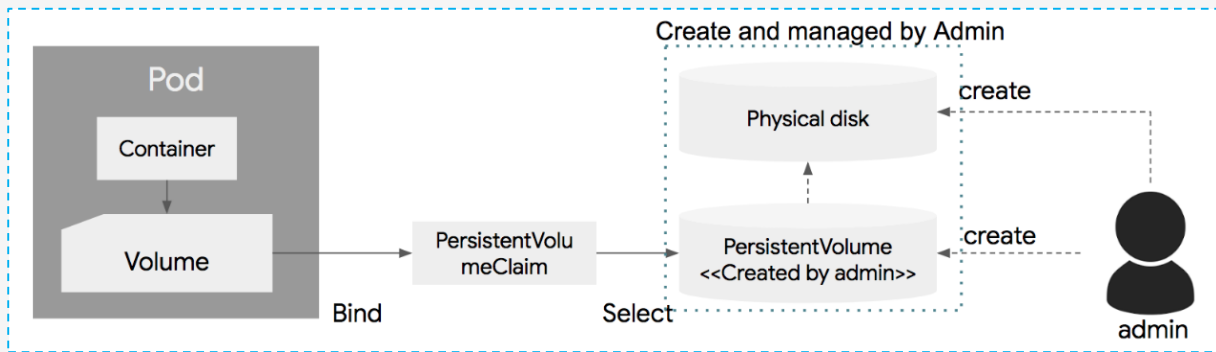
- Pod 생성시 지정된 Git 리파지토리의 특정 리비전을 Cloning하여 디스크 볼륨 생성
- 물리적으로는 emptyDir이 생성되고 Git Clone 수행



- HTML 같은 정적 파일 및 Nodejs 같은 스크립트 기반 코드 배포에 유용

PersistentVolume & PersistentVolumeClaim

- 특정 IT 환경에서는 영속성 있는 대용량 스토리지는 관리자에 의해 관리
 - 쿠버네티스 클러스터를 사용하는 개발자로부터 볼륨 프로비저닝 역할을 분리하는 사상
- 시스템 관리자가 실제 물리 디스크를 생성한 뒤, 이 디스크를 PersistentVolume 이라는 이름으로 Kubernetes에 등록
- 개발자는 Pod 생성 시, 볼륨을 정의하고, 해당 볼륨의 정의 부분에 PVC(PersistentVolumeClaim)를 지정하여 관리자가 생성한 PV와 연결



StorageClass - Dynamic PV Provisioning



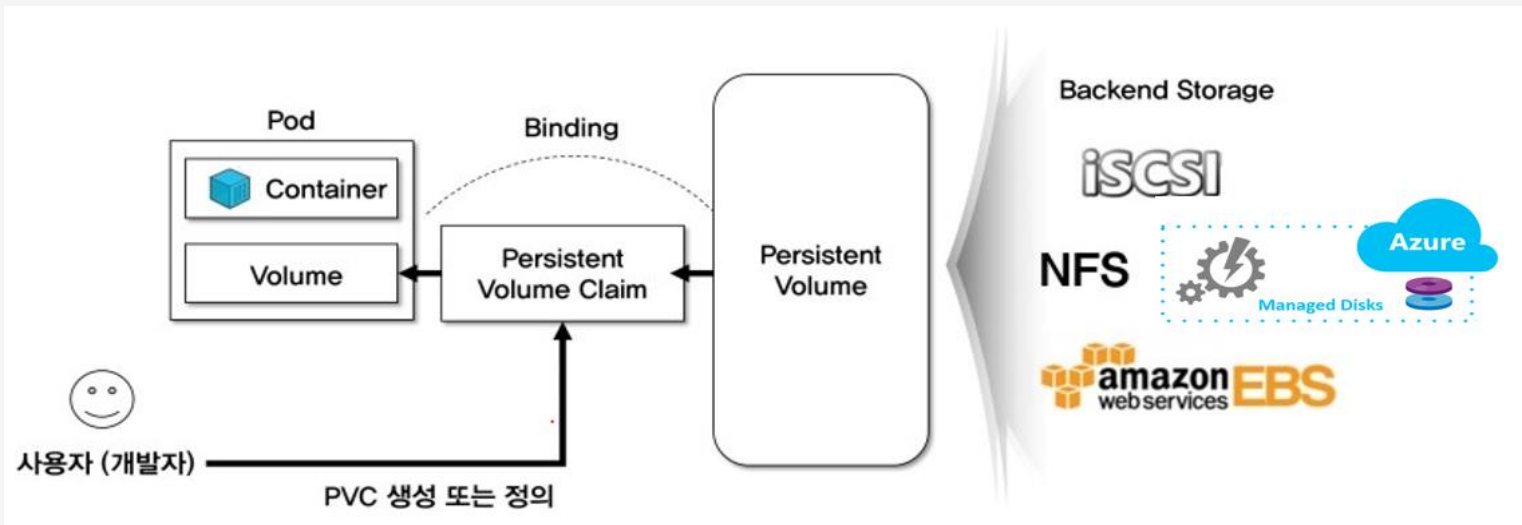
- PV는 관리자에 의해 수동으로 생성될 수 있지만, 자동 생성도 가능(Dynamic Provisioning)
- StorageClass(SC) Object
- StorageClass 객체에 의해 PersistentVolumes 동적 제공 가능
 - StorageClass는 PersistentVolume를 만들기 위해 Cloud Provider별 CSI 인터페이스를 구현하여 제공
- PersistentVolumes 스토리지 관리를 제공하는 Volume Types :
 - GCEPersistentDisk, AWSElasticBlockStore, AzureDisk, NFS, iSCSI

GCP StorageClass Object

```
apexacme@cloudshell:~ (public-education-uengine)$ kubectl get storageclass
NAME                                PROVISIONER          AGE
standard (default)  kubernetes.io/gce-pd  24h
apexacme@cloudshell:~ (public-education-uengine)$
```

GCP default Storage class Object

PersistentVolumeClaims



- Pod가 크기, 접근 모드에 따라 PVC를 요청, 적합한 PersistentVolume 발견시 PersistentVolume Claim에 바인딩
- PVC 조건을 만족하는 PV가 없을 경우, PV를 StorageClass가 자동으로 Provisioning하여 바인딩

Lab. Create PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard
  resources:
    requests:
      storage: 1Gi
```

- **accessMode:**
 - ReadWriteOnce : 하나의 Pod에만 마운트되고, 읽고 쓰기 가능
 - ReadOnlyMany : 여러 개의 Pod에서 마운트되고, 동시에 읽기만 가능 (쓰기는 불가능)
 - ReadWriteMany : 여러 개의 Pod에서 마운트되고, 동시에 읽고 쓰기 가능
- **kubectl apply -f volume-pvc.yaml**

- **kubectl get pvc**

```
apexacme@APEXACME: ~/yaml/volume$ kubectl get pvc
NAME                STATUS    VOLUME          CAPACITY   ACCESS MODES   STORAGECLASS   AGE
azure-managed-disk  Pending                                default       6s
apexacme@APEXACME: ~/yaml/volume$
```

- **kubectl describe pvc**

```
apexacme@APEXACME: ~/yaml/volume$ kubectl describe pvc
NAME                STATUS    VOLUME          CAPACITY   ACCESS MODES   STORAGECLASS   AGE
azure-managed-disk  Bound     pvc-817b4f22-5141-11ea-a89c-12c099b138d1  1Gi        RWO            default       67s
apexacme@APEXACME: ~/yaml/volume$
```

Lab. Create Pod with PersistentVolumeClaim

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:1.15.5
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 250m
        memory: 256Mi
    volumeMounts:
    - mountPath: "/mnt/gcp"
      name: volume
  volumes:
  - name: volume
    persistentVolumeClaim:
      claimName: azure-managed-disk
```

- `kubectl apply -f pod-with-pvc.yaml`
- `kubectl get pod`
- `kubectl describe pod mypod`
- `kubectl exec -it mypod -- /bin/bash`
- `cd /mnt/gcp`
- `df -k` 로 size 확인

```
root@mypod:/# df -k
Filesystem      1K-blocks    Used Available Use% Mounted on
overlay         101445900 19656992  81772524 20% /
tmpfs           65536        0    65536    0% /dev
tmpfs          3556916        0   3556916    0% /sys/fs/cgroup
/dev/sda1       101445900 19656992  81772524 20% /etc/hosts
/dev/sdc        999320      1284    981652    1% /mnt/azure
shm            65536        0    65536    0% /dev/shm
tmpfs          3556916      12   3556904    1% /run/secrets/kubernetes.io/serviceaccount
tmpfs          3556916        0   3556916    0% /proc/acpi
tmpfs          3556916        0   3556916    0% /proc/scsi
tmpfs          3556916        0   3556916    0% /sys/firmware
root@mypod:/#
```

Lab. Volumes

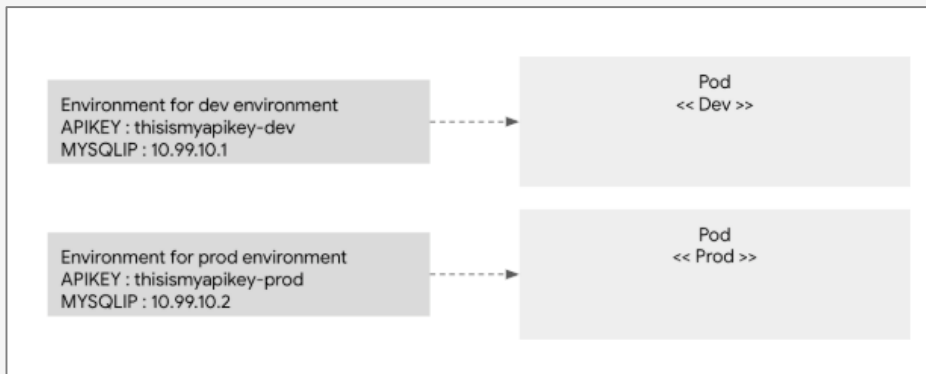


Lab Time

- Lab Script Location
 - Workflowy :

ConfigMaps

- ConfigMaps는 컨테이너 이미지로부터 설정 정보를 분리할 수 있게 해준다.
- 환경변수나 설정값 들을 환경변수로 관리해 Pod가 생성될 때 이 값을 주입



- ConfigMaps은 2가지 방법으로 생성
 - 리터럴 값
 - 파일
- ConfigMaps는 etcd에 저장

리터럴 값으로부터 ConfigMap 생성

- ConfigMap을 생성하는 명령어

```
$ kubectl create configmap my-config --from-literal=key1=value1 -  
-from-literal=key2=value2  
configmap "my-config" created
```

- 설정된 ConfigMap 정보 가져오기

```
$ kubectl get configmaps my-config -o yaml  
apiVersion: v1  
data:  
  key1: value1  
  key2: value2  
kind: ConfigMap  
metadata:  
  creationTimestamp: 2017-05-31T07:21:55Z  
  name: my-config  
  namespace: default  
  resourceVersion: "241345"  
  selfLink: /api/v1/namespaces/default/configmaps/my-config  
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```

- -o yaml 옵션은 해당 정보를 yaml형태로 출력하도록 요청한다.
- 해당 객체는 종류가 ConfigMap이며 key-value 값을 가지고 있다.
- ConfigMap의 이름 등의 정보는 metadata field에 들어 있다.

파일로부터 ConfigMap 생성 (1/2)

- 아래와 같은 설정 파일을 만든다.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: customer1
data:
  TEXT1: Customer1_Company
  TEXT2: Welcomes You
  COMPANY: Customer1 Company Technology Pct. Ltd.
```

- customer1-configmap.yaml**라는 이름으로 파일을 생성하였을 경우, 아래와 같이 ConfigMap를 생성한다.

```
$ kubectl create -f customer1-configmap.yaml
configmap "customer1" created
```

파일로부터 ConfigMap 생성 (2/2)

- Userinfo.properties 파일을 생성하고,

```
myname=apexacme  
email=apexacme@uengine.org  
Address=seoul
```

- 파일을 이용해 ConfigMap을 만들 때는 --from-file을 이용해 파일명을 넘긴다.
- `kubectl create configmap cm-file --from-file=./properties/profile.properties`
 - 이때, 키는 파일명이 되고, 값은 파일 내용이 됨

key : "profile.properties"

value :
myname=terry
email=myemail@mycompany.com
address=seoul

Containerizing with ConfigMap from Dockerizing

- Scenario
 - ConfigMap 생성
 - ConfigMap의 환경변수를 읽어 출력하는 NodeJS 어플리케이션 준비
 - Dockerfile 생성
 - Dockerizing & Azure Container Registry에 Push
 - Deployment yaml, Service yaml 준비
 - 배포 및 서비스 생성
 - 브라우저를 통해 서비스 확인
 - ConfigMap의 환경변수를 어플리케이션이 정상적으로 참조하여 출력하는지 여부

Containerizing with ConfigMap from Dockerizing

- ConfigMap 생성

```
$ kubectl create configmap hello-cm --from-literal=language=java
$ kubectl get cm
$ kubectl get cm hello-cm -o yaml
```

- ConfigMap의 환경변수를 읽어 출력하는 NodeJS 어플리케이션

```
var os = require('os');
var http = require('http');
var handleRequest = function(request, response) {
  response.writeHead(200);
  response.end(" my prefered language is "+process.env.LANGUAGE+ "\n");

  //log
  console.log("[ "+
    Date(Date.now()).toLocaleString()+
    " ] "+os.hostname());
}
var www = http.createServer(handleRequest);
www.listen(8080);
```

Containerizing with ConfigMap from Dockerizing

- Dockerfile 생성

```
FROM node:carbon
EXPOSE 8080
COPY server.js .
CMD node server.js > log.out
```

- Dockerizing & Azure Container Registry에 Push

```
$ docker build -t gcr.io/event-storming/cm-sandbox:v1 .
$ docker push gcr.io/event-storming/cm-sandbox:v1
```

Containerizing with ConfigMap from Dockerizing

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: cm-deployment
spec:
  replicas: 1
  minReadySeconds: 5
  selector:
    matchLabels:
      app: cm-literal
  template:
    metadata:
      name: cm-literal-pod
      labels:
        app: cm-literal
    spec:
      containers:
        - name: cm
          image: gcr.io/event-storming/cm-sandbox:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: LANGUAGE
              valueFrom:
                configMapKeyRef:
                  name: hello-cm
                  key: language
```

- Deployment(cm-deployment.yaml) 생성 / 실행
- `kubectl create -f cm-deployment.yaml`
- `$ kubectl get deploy`

Containerizing with ConfigMap from Dockerizing

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

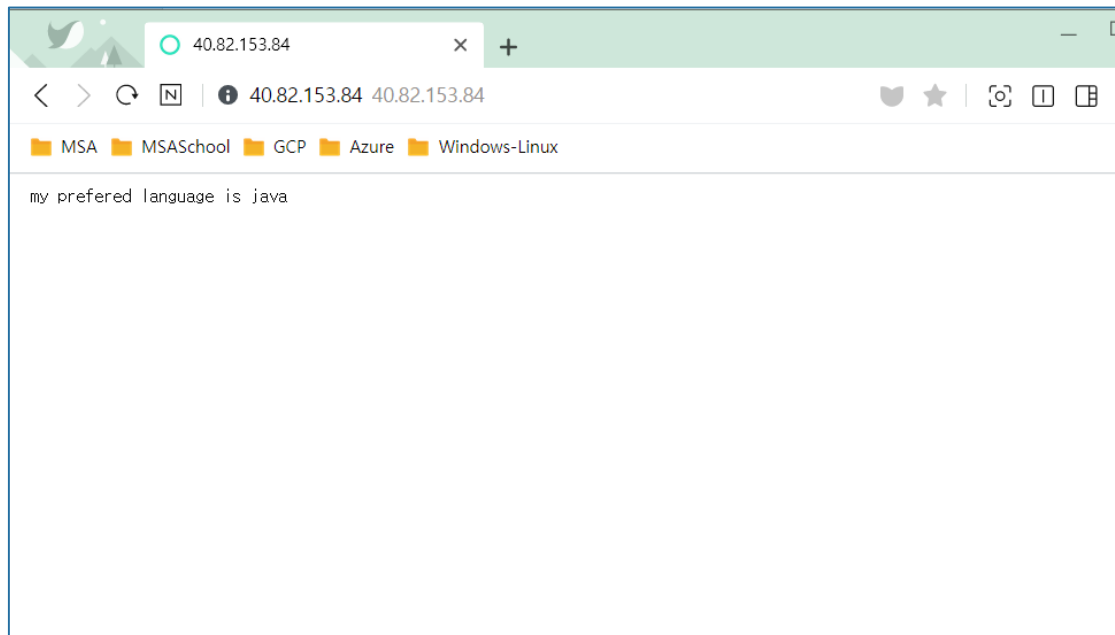
- Service(cm-service.yaml) 생성/ 실행
\$ kubectl create -f cm-service.yaml
- \$ kubectl get svc

```
apexacme@APEXACME:~/yaml/configmap$ kubectl get svc
NAME                TYPE                CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
cm-literal-svc      LoadBalancer        10.0.99.121     40.82.153.84     80:30181/TCP     30m
kubernetes           ClusterIP            10.0.0.1        <none>           443/TCP          28h
apexacme@APEXACME:~/yaml/configmap$
```

- 브라우저를 통해 서비스 확인
 - Service의 External-IP 접속

Containerizing with ConfigMap from Dockerizing

- 마이크로서비스 결과 확인



Pod에서 ConfigMap 추가 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: cm-file-deployment
spec:
  replicas: 3
  minReadySeconds: 5
  selector:
    matchLabels:
      app: cm-file
  template:
    metadata:
      name: cm-file-pod
      labels:
        app: cm-file
    spec:
      containers:
        - name: cm-file
          image: gcr.io/event-storming/cm-file:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: PROFILE
              valueFrom:
                configMapKeyRef:
                  name: cm-file
                  key: profile.properties
```

- 환경변수로 값 전달

- cm-file configMap에서 키가 "profile.properties" (파일명)인 값을 읽어와서 환경 변수 PROFILE에 저장
- 저장된 값은 파일의 내용인 아래 문자열이 됨
- myname=terry
email=myemail@mycompany.com
address=seoul

Pod에서 ConfigMap 추가 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: cm-file-deployment-vol
spec:
  replicas: 3
  minReadySeconds: 5
  selector:
    matchLabels:
      app: cm-file-vol
  template:
    metadata:
      name: cm-file-vol-pod
      labels:
        app: cm-file-vol
    spec:
      containers:
        - name: cm-file-vol
          image: gcr.io/event-storming/cm-file-volume:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: config-profile
              mountPath: /tmp/config
      volumes:
        - name: config-profile
          configMap:
            name: cm-file
```

- **디스크 볼륨으로 마운트 하기**
 - ConfigMap을 Volume으로 정의하고, 이 볼륨을 volumeMounts를 이용해 /tmp/config에 마운트 함
 - 이때 중요한점은 마운트 포인트에 마운트 될때, ConfigMap내의 키가 파일명이 됨

Lab. ConfigMap



Lab Time

- Lab Script Location
 - Workflowy :

Secrets

- ConfigMap이 일반적인 환경 설정 정보나 Config정보를 저장하도록 디자인 되었다면, 보안이 중요한 패스워드나 API 키, 인증서 파일들은 Secret에 저장
- Secret은 정보보안 차원에서 추가적인 보안 기능을 제공
 - 예를 들어, API서버나 Node의 파일에 저장되지 않고, 항상 메모리에 저장되므로 상대적 접근이 어려움
 - Secret의 최대 크기는 1MB (너무 커지면, apiserver나 Kubelet의 메모리에 부하 발생)
- ConfigMap과 기본적으로 유사하나, 값(value)에 해당하는 부분을 base64로 인코딩해야 함
 - SSL인증서와 같은 binary파일의 경우, 문자열 저장이 불가능하므로 인코딩 필요
 - 이를 환경변수로 넘길 때나 디스크볼륨으로 마운트해서 읽을 경우 디코딩 되어 적용

```
apiVersion: v1
kind: Secret
metadata:
  name: hello-secret
data:
  language: amF2YQo=
```

Kubectl 명령어로 Secret 생성 및 확인

- 명령어로 Secret 만들기
 - `$ kubectl create secret generic my-password --from-literal=password=mysqlpassword`
 - my-password라는 Secret을 생성하고, password 라는 key와 mysqlpassword라는 value 값을 가지게 된다.
 - Value는 base64로 자동 encoding
 - **generic** : create a secret from a local file, directory or literal value
- Secret 확인 : `kubectl get secret my-password -o yaml`
 - `echo [base64 value] | base64 --decode`

Secret을 직접 만들기

- base64 형태로 인코딩하여 YAML파일내에 직접 생성 가능

```
$ echo mysqlpassword | base64  
bXlzcWxwYXNzd29yZAo=
```

- 위 방식으로 인코딩 된 정보를 사용해 설정파일 생성

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: my-password  
type: Opaque  
data:  
  password: bXlzcWxwYXNzd29yZAo=
```

- **base64** 인코딩은 바로 디코딩 됨으로 주의!

```
$ echo "bXlzcWxwYXNzd29yZAo=" | base64 --decode
```

설정파일을 절대 소스코드에 넣지 않도록 주의한다!

Pod에서 Secret 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: hello-secret-deployment
spec:
  replicas: 1
  minReadySeconds: 5
  selector:
    matchLabels:
      app: hello-secret-literal
  template:
    metadata:
      name: hello-secret-literal-pod
      labels:
        app: hello-secret-literal
    spec:
      containers:
        - name: hello-secret
          image: gcr.io/event-storming/hello-secret:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: LANGUAGE
              valueFrom:
                secretKeyRef:
                  name: hello-secret
                  key: language
```

- Deployment.yaml 생성/ 실행
- `kubectl create -f hello-secret-deployment.yaml`
- `$ kubectl get deploy`

Pod에서 Secret 파일 마운트 사용하기

- 사용자 id를 저장한 user.property 파일과 비밀번호를 저장한 password.property 파일 생성
 - id file 내용 : `$ cat user.property $ admin`
 - password file 내용 : `$ cat password.property $ adminpassword`
- Secret 생성
 - `kubectl create secret generic db-password --from-file=./user.property --from-file=./password.property`
 - 생성된 secret은 user.property, password.property 파일명을 각각 key로 파일의 내용이 저장
- Secret을 읽어 출력할 어플리케이션 생성 : server.js

```
var os = require('os');
var fs = require('fs');
var http = require('http');
var handleRequest = function(request, response) {
  fs.readFile('/tmp/db-password/user.property', function(err, userid){
    response.writeHead(200);
    response.write("user id is "+userid+" \n");
    fs.readFile('/tmp/db-password/password.property', function(err, password){
      response.end(" password is "+password+ " \n");
    })
  })
  console.log("[ " +
    Date(Date.now()).toLocaleString()+ " ] " + os.hostname());
}
var www = http.createServer(handleRequest);
www.listen(8080);
```

Pod에서 Secret 파일 마운트 사용하기

- Dockerfile 생성

```
FROM node:carbon
EXPOSE 8080
COPY server.js .
CMD node server.js > log.out
```

- Dockerizing & Azure Container Registry에 Push

```
$ docker build -t gcr.io/event-storming/hello-secret-file:v1 .
$ docker push gcr.io/event-storming/hello-secret-file:v1
```

Pod에서 Secret 파일 마운트 사용하기

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: hello-serect-file-deployment
spec:
  replicas: 3
  minReadySeconds: 5
  selector:
    matchLabels:
      app: hello-secret-file
  template:
    metadata:
      name: hello-secret-file
      labels:
        app: hello-secret-file
    spec:
      containers:
        - name: hello-secret-file
          image: gcr.io/event-storming/hello-secret-file:v1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: db-password
              mountPath: "/tmp/db-password"
              readOnly: true
      volumes:
        - name: db-password
          secret:
            secretName: db-password
            defaultMode: 0600
```

- Deployment(hello-secret-file-deployment.yaml) 생성/ 실행
- `kubectl create -f hello-secret-file-deployment.yaml`
- `$ kubectl get deploy`

Pod에서 Secret 파일 마운트 사용하기

```
apiVersion: v1
kind: Service
metadata:
  name: hello-secret-file-service
spec:
  selector:
    app: hello-secret-file
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 8080
  type: LoadBalancer
```

- Service(hello-secret-file-service.yaml) 생성
\$ kubectl create -f hello-secret-file-service.yaml
- \$ kubectl get svc
- 브라우저를 통해 서비스 확인
 - Service의 External-IP 접속

Lab. Secret



Lab Time

- Lab Script Location
 - Workflowy :

여우야 여우야 뭐하니?
밥 먹는다
무슨 반찬?
개구리 반찬



살았니 죽었니?
살았다 야!

Liveness Probes & Readiness Probes

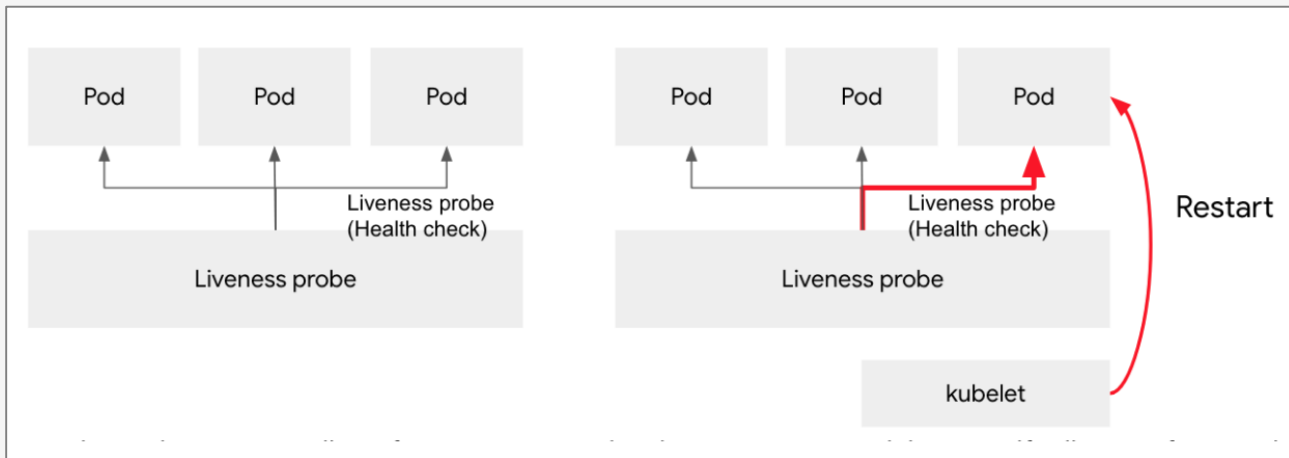
- 쿠버네티스는 각 컨테이너의 상태를 주기적으로 체크(Health Check)해서,
 - 문제가 있는 컨테이너를 자동으로 재시작하거나 또는 문제가 있는 컨테이너를 서비스에서 제외 한다.
- Liveness와 Readiness Probes은 kubelet이 pod내에서 실행되는 어플리케이션의 health를 조정하기 때문에 매우 중요하다.

Probe Types

- Liveness probe와 readiness probe는 컨테이너가 정상적인지 아닌지를 체크하는 방법으로 다음과 같이 3가지 방식을 제공한다.
 - Command probe
 - HTTP probe
 - TCP probe

Liveness Probes

- Pod는 정상적으로 작동하지만 내부의 어플리케이션이 반응이 없다면, 컨테이너는 의미가 없다.
 - 위와 같은 경우는 어플리케이션의 Deadlock 또는 메모리 과부화로 인해 발생할 수 있으며, 발생했을 경우 컨테이너를 다시 시작해야 한다.
- Liveness probe는 Pod의 상태를 체크하다가, Pod의 상태가 비정상인 경우 kubelet을 통해서 재시작한다.



Liveness Command probe

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
    name: liveness-exec
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
          initialDelaySeconds: 3
          periodSeconds: 5
```

- 왼쪽은 /tmp/healthy 파일이 존재하는지 확인하는 설정파일이다.
- periodSeconds 파라미터 값으로 5초마다 해당 파일이 있는지 조회한다.
- initialDelaySeconds 파라미터는 kubelet이 첫 체크하기 전에 기다리는 시간을 설정한다.
- 파일이 존재하지 않을 경우, 정상 작동에 문제가 있다고 판단되어 kubelet에 의해 자동으로 컨테이너가 재시작 된다.

Liveness HTTP probe

- Kubelet이 HTTP GET 요청을 /healthz 로 보낸다.
- 실패 했을 경우, kubelet이 자동으로 컨테이너를 재시작 한다.

```
livenessProbe:  
  httpGet:  
    path: /healthz  
    port: 8080  
    httpHeaders:  
      - name: X-Custom-Header  
        value: Awesome  
  initialDelaySeconds: 3  
  periodSeconds: 3
```

Liveness TCP Probe

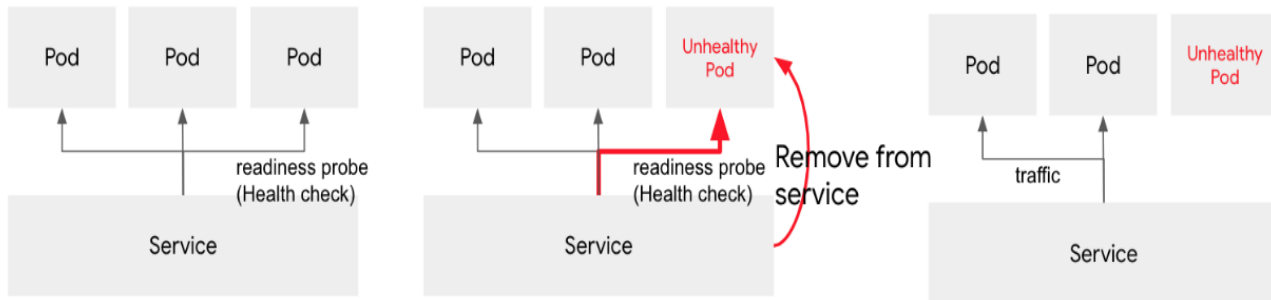
- kubelet은 TCP Liveness Probe를 통해, 지속적으로 어플리케이션이 실행중인 컨테이너의 TCP Socket을 열려고 한다.
- 정상인 아닌 경우 컨테이너를 재시작 한다.

```
livenessProbe:  
  tcpSocket:  
    port: 8080  
  initialDelaySeconds: 15  
  periodSeconds: 20
```


Readiness Probes

- Configuration을 로딩하거나, 많은 데이터를 로딩하거나, 외부 서비스를 호출하는 경우에는 일시적으로 서비스가 불가능한 상태가 될 수 있다.
- Readiness Probe를 사용하게 되면 주어진 조건이 만족할 경우, 서비스 라우팅하고, 응답이 없거나 실패한 경우, 서비스 목록에서 제외

```
readinessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/healthy  
  initialDelaySeconds: 5  
  periodSeconds: 5
```



Difference between Liveness and Readiness

- Liveness probe와 Readiness probe 차이점은
 - Liveness probe는 컨테이너의 상태가 비정상이라고 판단하면,
→ 해당 Pod를 재시작하는데 반해,
 - Readiness probe는 컨테이너가 비정상일 경우에는
→ 해당 Pod를 사용할 수 없음으로 표시하고, 서비스등에서 제외한다.
 - 주기적으로 체크하여, 정상일 경우 정상 서비스에 포함

Lab. Liveness 와 readiness probe

- exec-liveness 설정 파일 생성
 - `nano exec-liveness.yaml`
- 파일 설정으로 배포
 - `kubectl create -f exec-liveness.yaml`
- 결과 확인
 - `$ tmux #실행, split window on tmux`
 - `watch -n 1 kubectl get all`
 - `kubectl describe pod liveness-exec`

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy;
          sleep 600
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
          initialDelaySeconds: 5
          periodSeconds: 5
```

Lab. Liveness 와 readiness probe

- http-liveness 설정파일을 생성
 - `nano http-liveness.yaml`
- 파일 설정으로 배포
 - `kubectl create -f http-liveness.yaml`
- 내용 확인
 - `kubectl describe pod liveness-http`

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/liveness
      args:
        - /server
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
          httpHeaders:
            - name: X-Custom-Header
              value: Awesome
        initialDelaySeconds: 3
        periodSeconds: 3
```

Lab. Liveness 와 readiness probe

- tcp-liveness-readiness 파일을 생성
 - `nano tcp-liveness-readiness.yaml`
- 파일 설정으로 배포
 - `kubectl create -f tcp-liveness-readiness.yaml`
- 내용 확인
 - `kubectl describe pod goproxy`

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
    - name: goproxy
      image: k8s.gcr.io/goproxy:0.1
      ports:
        - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 20
```

Lab. Liveness, Readiness



Lab Time

- Lab Script Location
 - Workflowy :