

Udemy - k8s Maximilian

		Kubernetes
개발사	구글	
발표일	2014년 9월 9일 (10년 전)	
프로그래밍 언어	Go	
라이센스	Apache License 2.0	
	 	

↓
"CKA 취득까지"

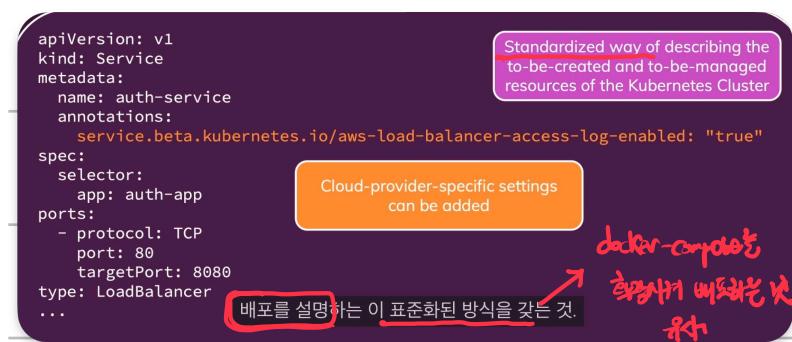
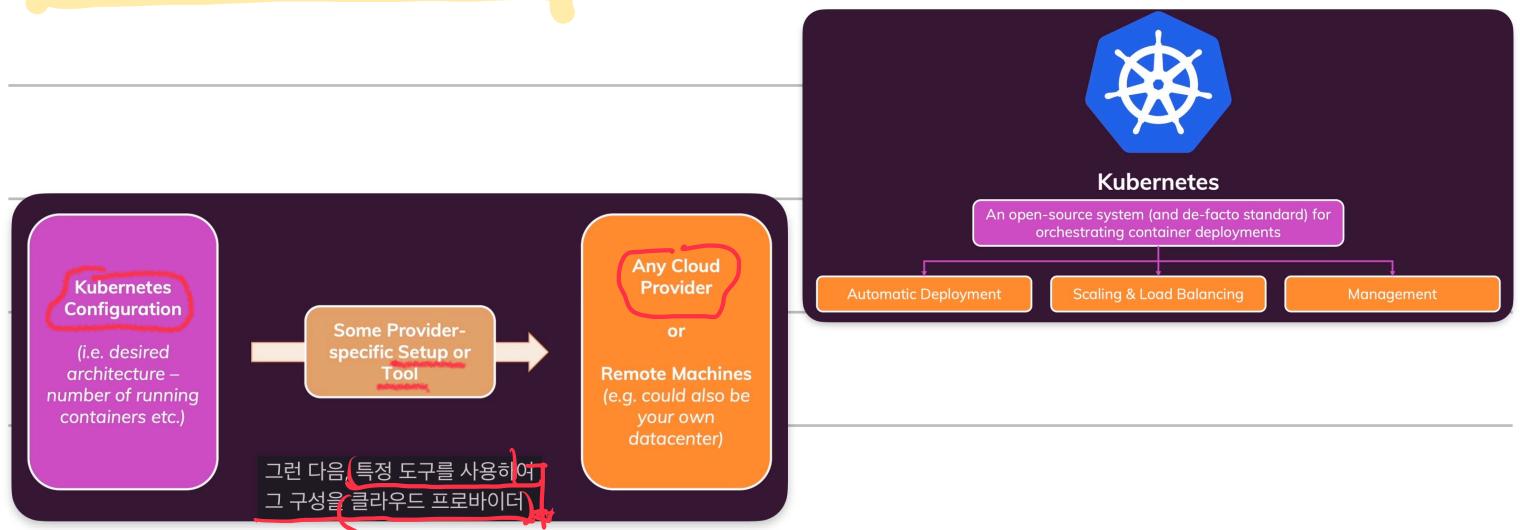
'24 12.29. ~ '25 1.8.

• KBS란?

컨테이너화된 App의 배포·확장 및 관리를 자동화하는 플랫폼

"간접적인 환경에서 간접적인 툴로 간접적인 투자"

→ SW(x), Tools(o)



"AWS의 표준"

인증을 보완

ECS → K8S

AWS only

AWS, Azure, GCP ...

Manual deployment of Containers is hard to maintain, error-prone and annoying

(even beyond security and configuration concerns!)

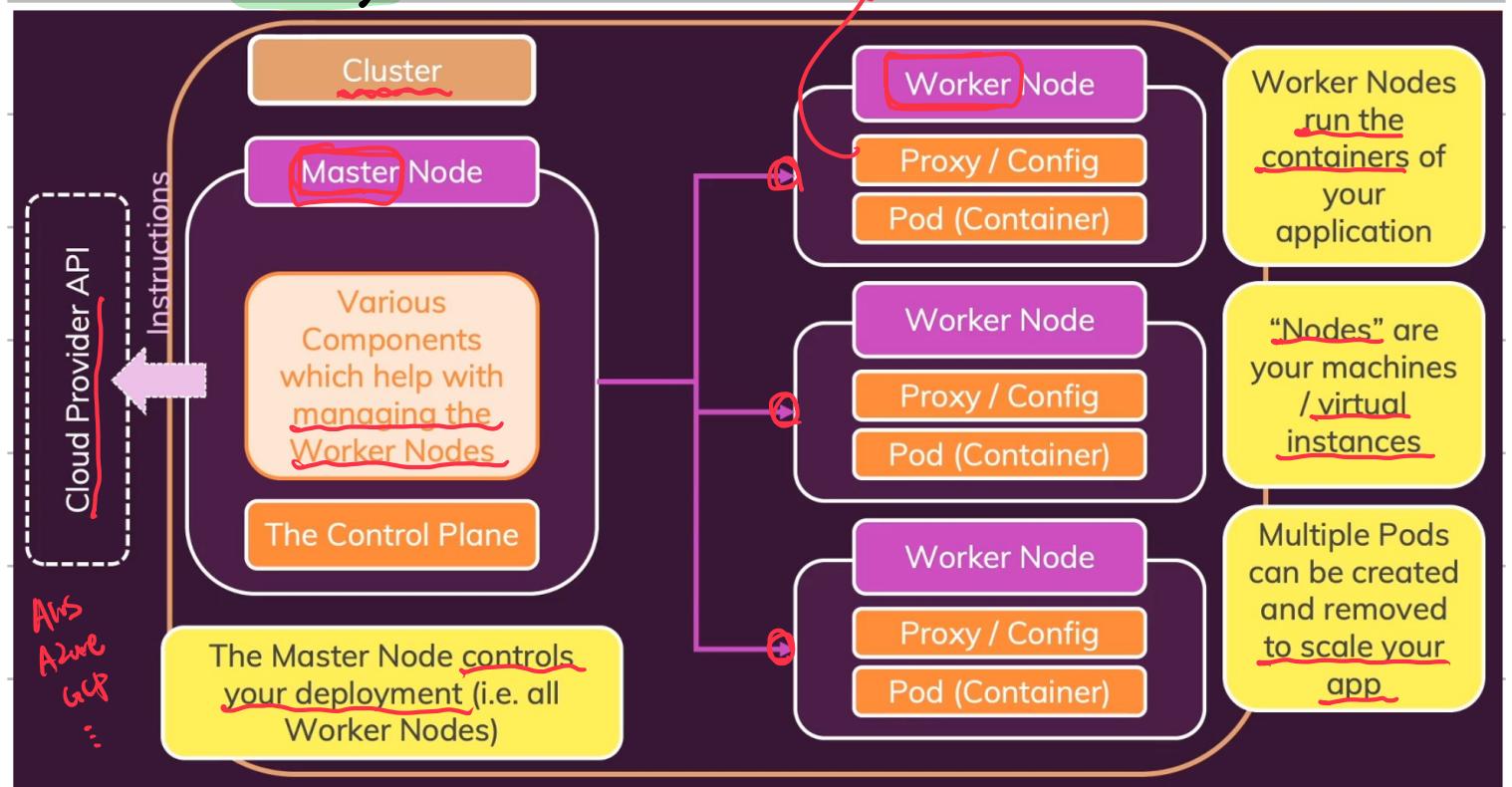


Container Health Check
+
Automatic Re-deployment

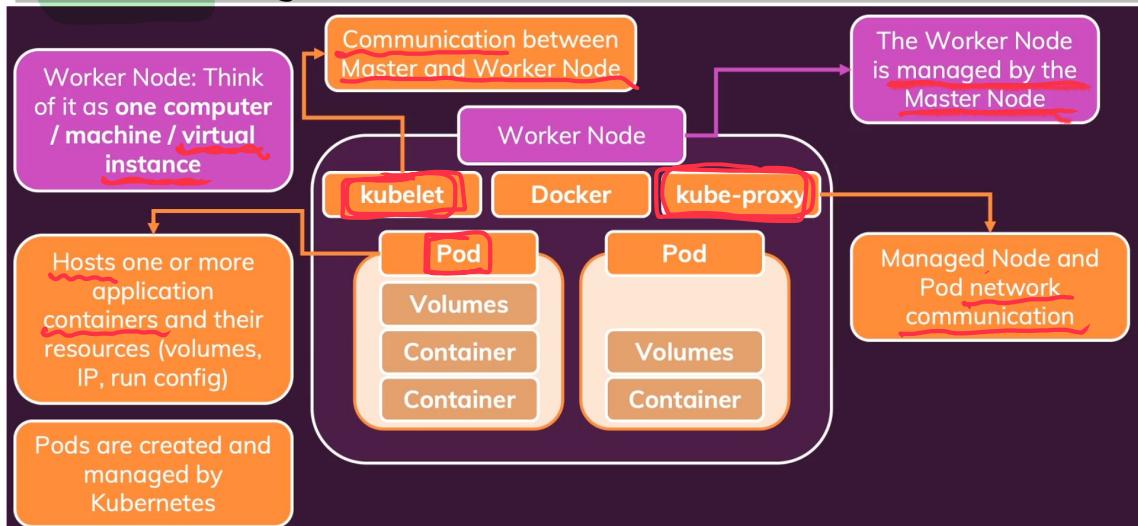
Auto Scaling

Load Balancer

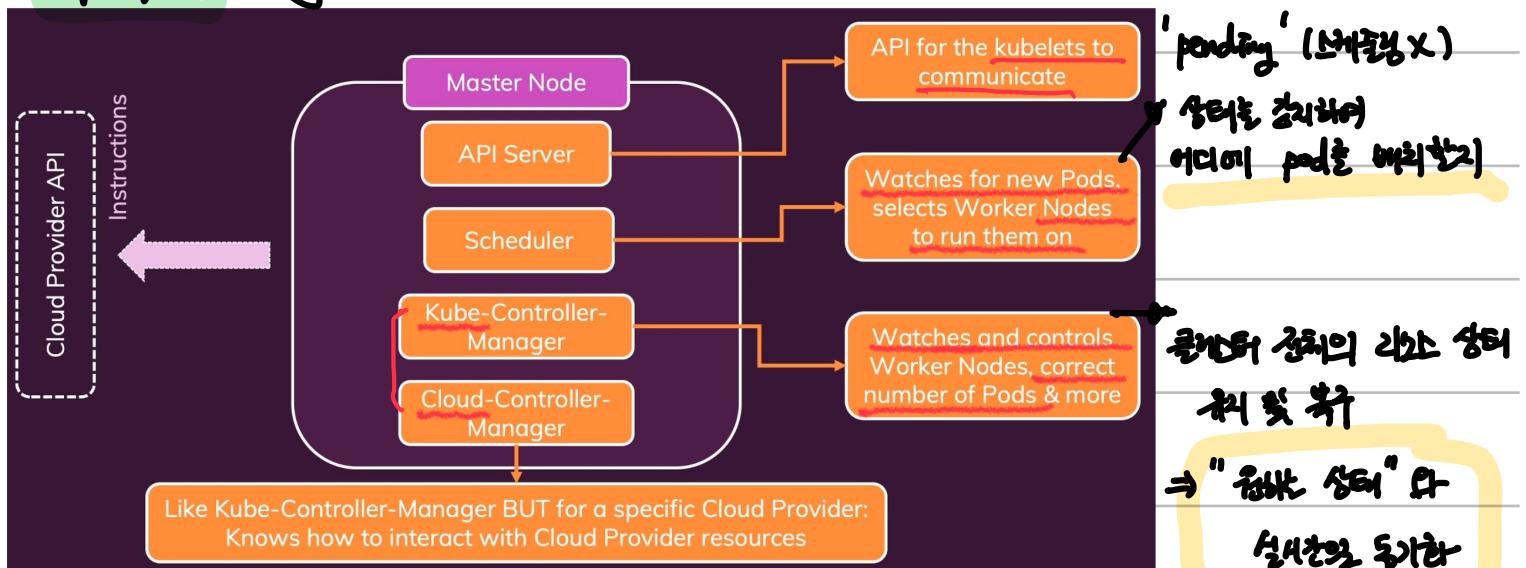
• 쿠버네티스 아키텍처



• 워커노드 구조



• 컨트롤러 매니저

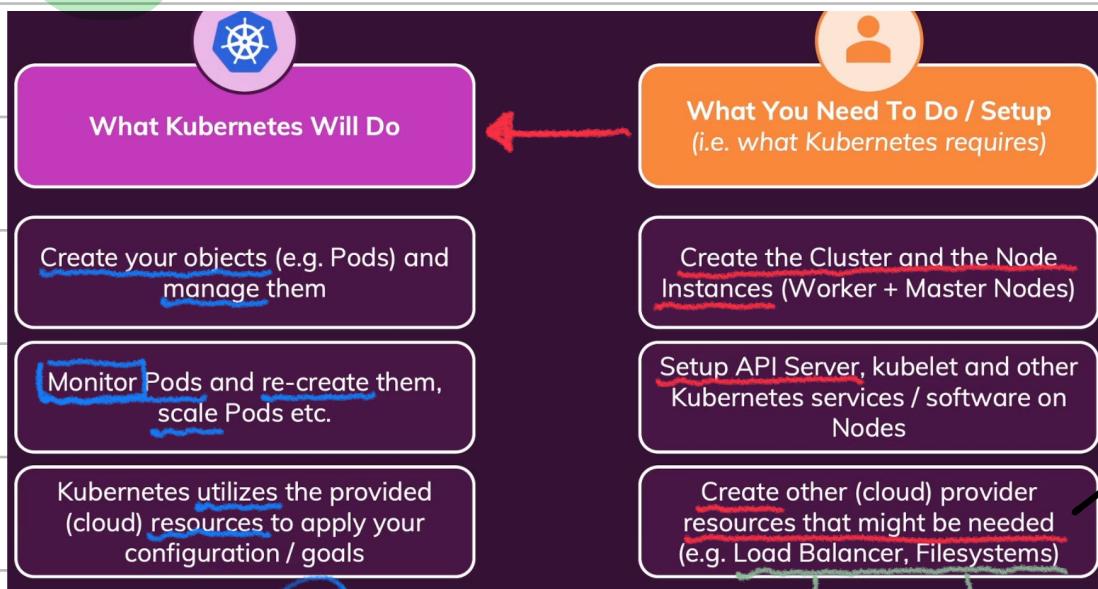


• Kubernetes

Cluster	A set of <u>Node</u> machines which are running the <u>Containerized Application (Worker Nodes)</u> or control other Nodes (Master Node)
Nodes	<u>Physical or virtual machine</u> with a certain hardware capacity which hosts <u>one or multiple Pods</u> and <u>communicates with the Cluster</u>
Master Node	Cluster Control Plane, <u>managing the Pods</u> across Worker Nodes
Worker Node	Hosts Pods, <u>running App Containers (+ resources)</u>
Pods	Pods <u>hold the actual running App Containers + their required resources</u> (e.g. volumes).
Containers	Normal (Docker) Containers
Services	A <u>logical set (group)</u> of Pods with a unique, Pod- and Container-independent IP address

서비스

• K&R



설정 + 구조화

해당부분

k8s는 외과를 치료하는

일반적이다.

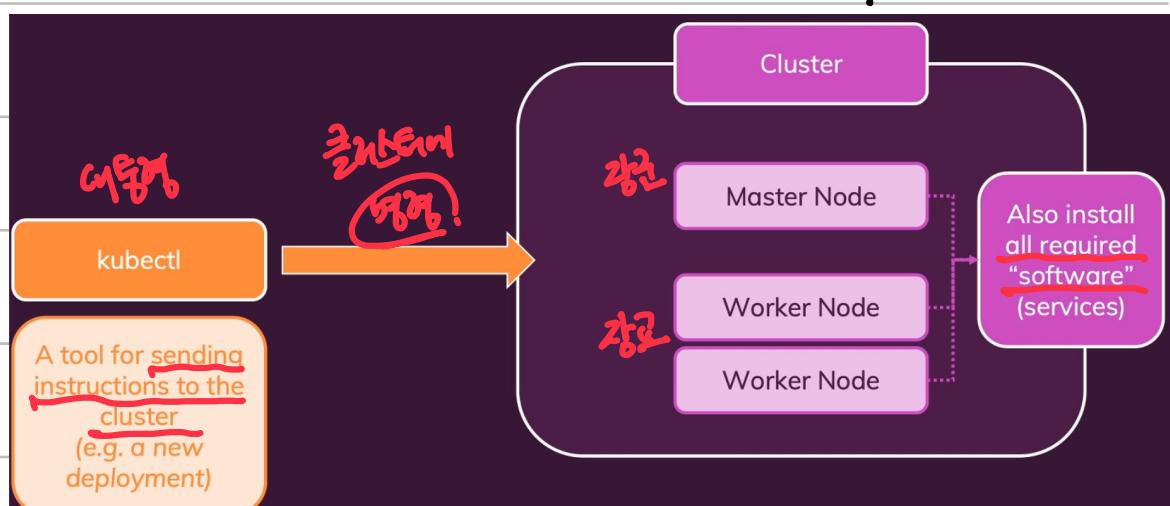
EKS, AKS, ... 이런

OR

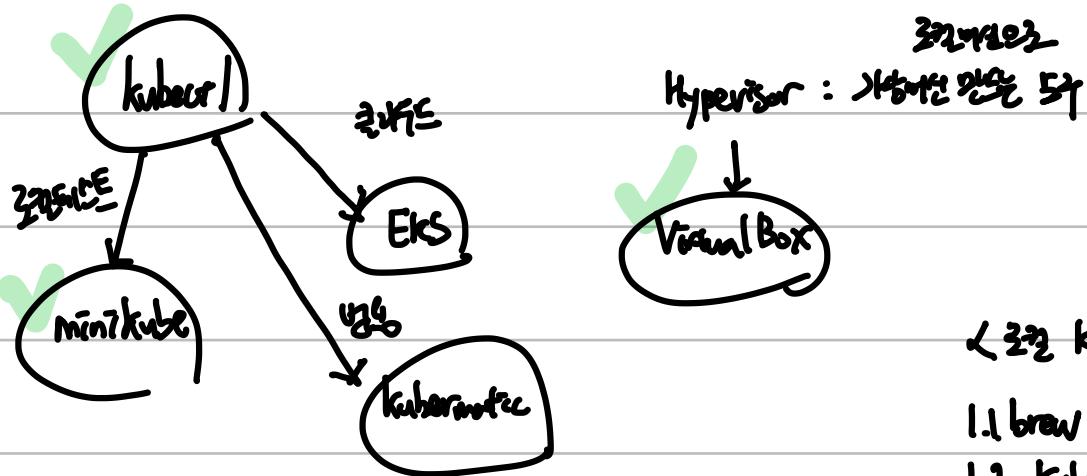
kubernetes

"k8s 자체는 어떤 안전성을 가질까"

• kubectl은 어떻게 k8s에 지시를 보내나 // k8s의 명령어 통제를 위한 명령어



3월 5주차



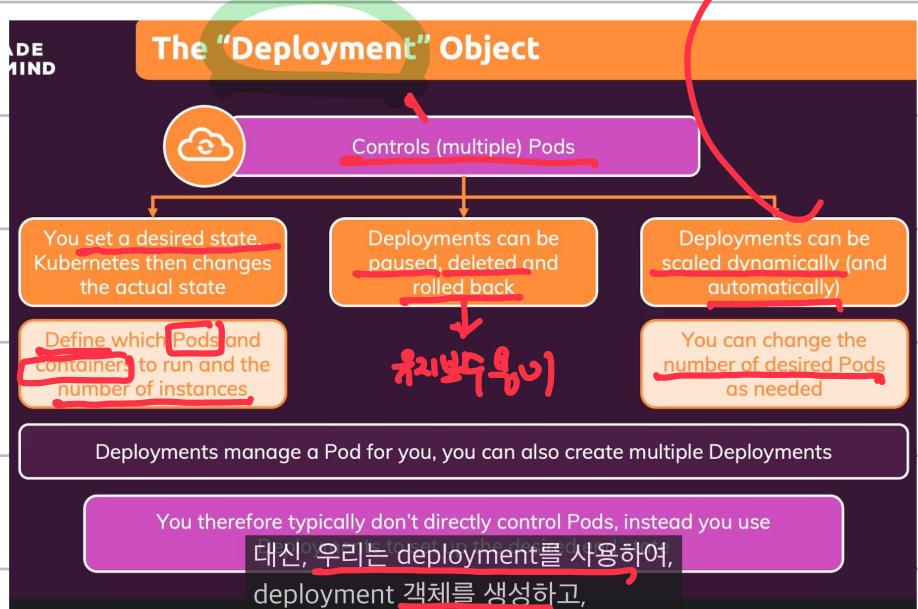
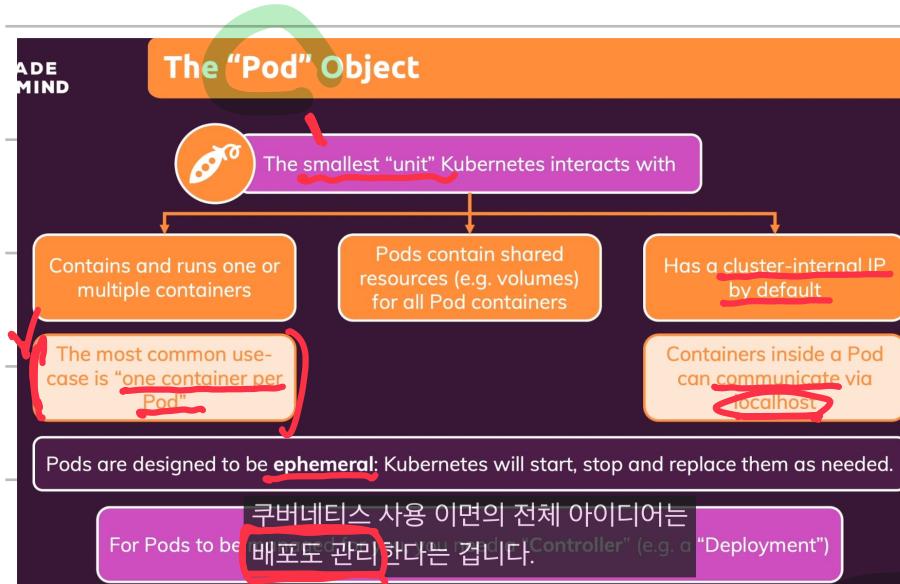
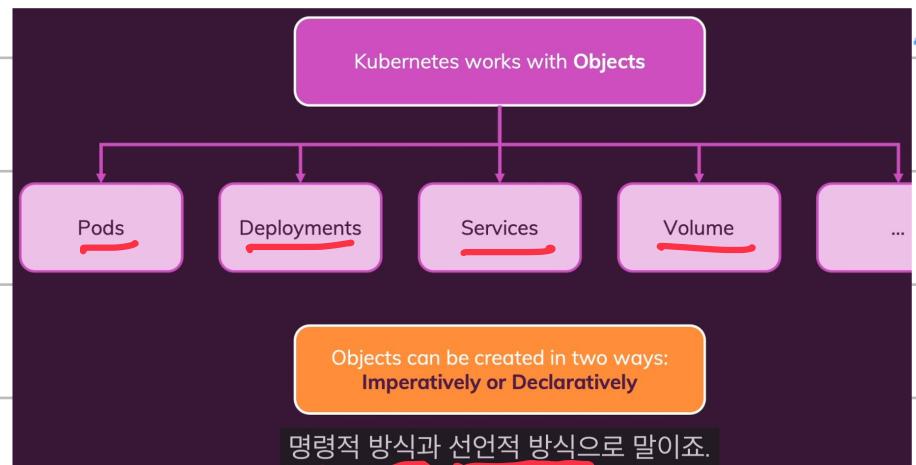
< 3주 k8s 퍼포먼스 - 준비 >

1. brew install kubectl
- 1.2. kubectl version --client
- 2.1 brew install minikube

2.2. minikube start
-- driver = docker

// 웹브라우저 docker 터미널로 실행
→ docker ps

2.3. minikube status
2.4. minikube dashboard
// k8s 웹API API



1. 3 디렉토리

여러개 파일 있으나

datamaster / k8s-first-app

kubectl create deployment <deployment名> --image=k8s-first-app // 실행

~~1. 페리~~ ⇒ docker hub에서 풀링된 이미지 선택 (\because 코딩언어 \neq 실행언어)

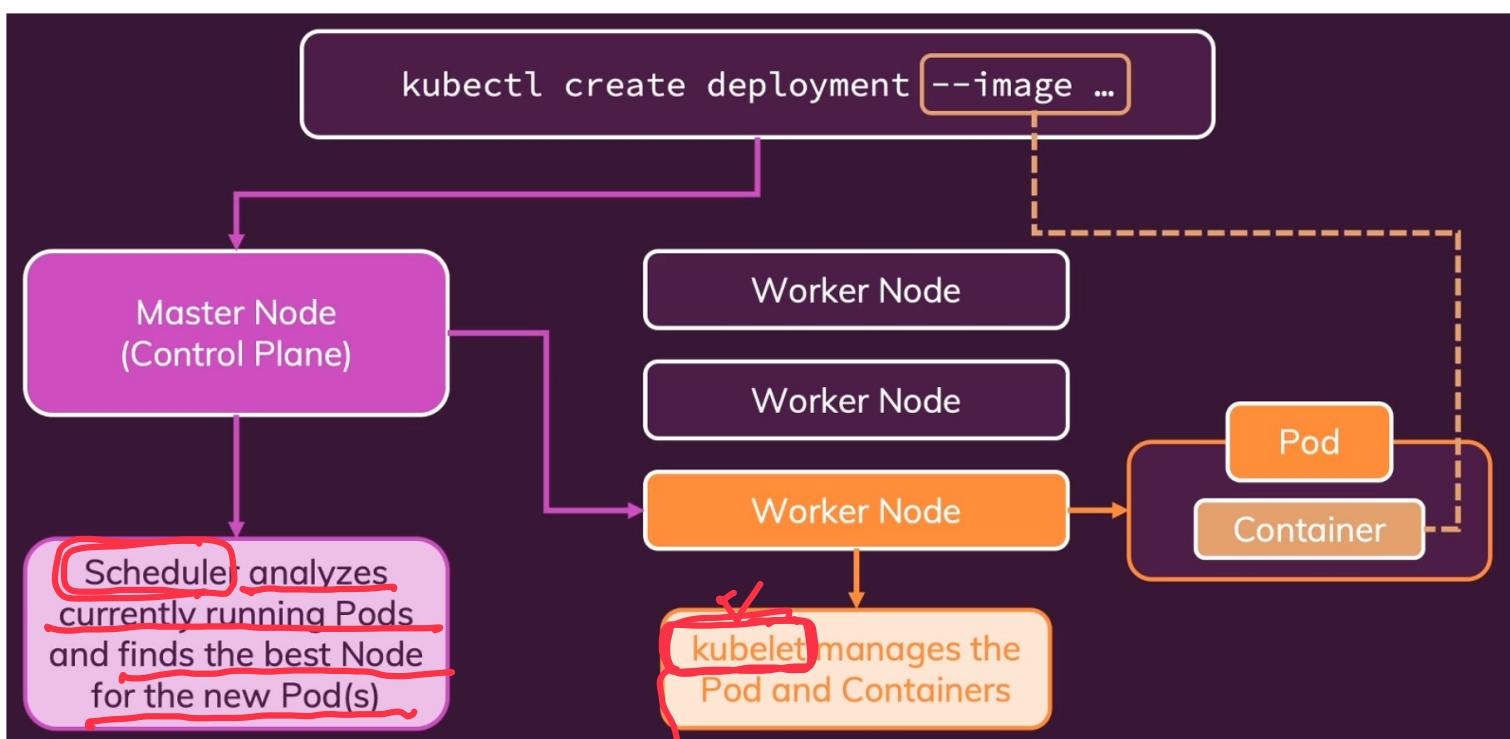
kubectl get deployments // deployment 목록 확인

kubectl get pods // deployment pod 목록 확인

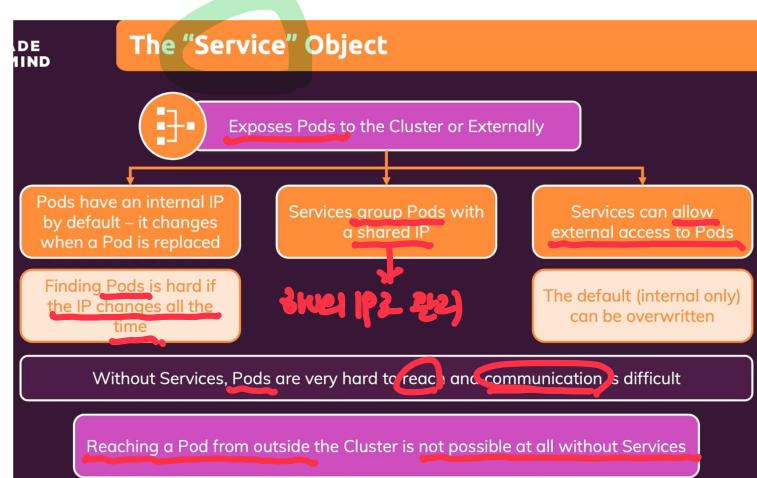
```
$ kubectl get pods
      READY   STATUS
59fb4-5zp2k  0/1   ErrImagePull
$ Current State   Target State
```

1/1이筹备!

기록이 아니었다



→ 외부로 Control Plane 접속,
내부로 Pod, Container 접속



Pod는 자체 IP를 갖고 있어

Reach. communication 가능하지 않음 //

- ① 대체 IP 주소
⇒ ② 외부에 있는 서비스

→ create service 및 퍼블릭 IP 설정

① hw@hws-MacBook-Pro-14 kub-action-01-starting-setup % kubectl expose deployment first-app --type=LoadBalancer --port=8080

service/first-app exposed
② hw@hws-MacBook-Pro-14 kub-action-01-starting-setup % kubectl get services
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
first-app LoadBalancer 10.98.198.51 <pending> 8080:31659/TCP 13s
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 2d4h

hw@hws-MacBook-Pro-14 kub-action-01-starting-setup % minikube service first-app

NAMESPACE	NAME	TARGET PORT	URL
default	first-app	8080	http://192.168.49.2:31659
first-app 서비스의 터널을 시작하는 중			
NAMESPACE	NAME	TARGET PORT	URL
default	first-app		http://127.0.0.1:60544

Opening service default/first-app in default browser...
darwin에서 Docker 드라이버를 사용하고 있기 때문에, 터미널을 열어야 실행할 수 있습니다

minikube의 IP

F4 IP 확인 가능

Autoscaling.

Scale

hw@hws-MacBook-Pro-14 kub-action-01-starting-setup % kubectl scale deployment/first-app

--replicas=3

deployment.apps/first-app scaled

hw@hws-MacBook-Pro-14 kub-action-01-starting-setup % kubectl get pods

NAME	READY	STATUS	RESTARTS	AGE
first-app-98d65897d-9zcp	1/1	Running	0	4s
first-app-98d65897d-csskz	0/1	ContainerCreating	0	4s
first-app-98d65897d-psjnw	1/1	Running	2 (3m32s ago)	79m

이미지 배포 과정

docker push datamaster/k8s-first-app:1 // tag은 빼놓지 말!

docker set image deployment/first-app

k8s-first-app = datamaster/k8s-first-app:1

이전 이미지

새로운 이미지

Deployment 툴박. 터미널

→ rollout undo

status

history

```
JS app.js > app.get('/') callback
4
5 app.get('/', (req, res) => {
6   res.send(`
7     <h1>Hello from this NodeJS app!!!!</h1>
8     <p>This is new!</p>
9     <p>Try sending a request to /error and see what happens</p>
10`);
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE 2: zsh

docker-complete \$ kubectl rollout undo deployment/first-app

deployment.apps/first-app rolled back

docker-complete \$ kubectl get pods

NAME READY STATUS RESTARTS AGE

first-app-5bf786b767-scx4r 1/1 Running 0 6m11s

docker-complete \$ kubectl rollout status deployment/first-app

deployment "first-app" successfully rolled out

→ rollback 되고 / 등록 --to-revision=(몇번) 그 번호로 이동

→ 모니터링 / 콘솔 dashboard 확인



```

JS app.js > app.get('/').callback
4
5   app.get('/', (req, res) => {
6     res.send(`
7       <h1>Hello from this NodeJS app!!!!</h1>
8       <p>This is new!</p>
9       <p>Try sending a request to /error and see what happens</p>
10      `);

```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE 2: zsh +

```

docker-complete $ kubectl rollout history deployment/first-app
deployment.apps/first-app
REVISION CHANGE-CAUSE
1 <none>
3 <none>
4 <none>

```

deployment의 버전은 4

```

docker-complete $ kubectl rollout undo --revision=3 deployment/first-app
리비전(revision) 식별자 중 하나를 사용하여

```

```

JS app.js > app.get('/').callback
4
5   app.get('/', (req, res) => {
6     res.send(`
7       <h1>Hello from this NodeJS app!!!!</h1>
8       <p>This is new!</p>
9       <p>Try sending a request to /error and see what happens</p>
10      `);

```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE 2: zsh +

```

docker-complete $ kubectl rollout undo deployment/first-app --to-revision=1
deployment.apps/first-app rolled back

```

• 리버스 캐시

1. 서비스 초기화

2. deployment 초기화 ...

코드의 실행이 되었어

X 예상치 못한 향상

실패

2. deployment 초기화

kubectl apply -f=deployment.yaml, service.yaml // 예상치 못한

delete

Config.yaml을 통한

인프라 정의

(1) yaml 파일을读后 apply를 하면 먼저 명령이 실행됨!

(2) git로 배포하려하면, 형상에 용이함. 추적 가능

```

JS app.js > app.get('/').callback
1  const express = require('express');
2
3  const app = express();
4
5  app.get('/', (req, res) => {
6    res.send(`
7      <h1>Hello from this NodeJS app!!!!</h1>

```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE 2: zsh +

```

docker-complete $ kubectl delete service first-app
service "first-app" deleted
docker-complete $ kubectl delete deployment first-app
deployment.apps "first-app" deleted

```

Imperative

Declarative

kubectl create deployment ...

kubectl apply -f config.yaml

Individual commands are executed to trigger certain Kubernetes actions

A config file is defined and applied to change the desired state

Comparable to using docker run only

Comparable to using Docker Compose with compose files

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: second-app
  #tier: backend pod 전체가 service로 관리되므로 생략 가능
  ports:
    - protocol: 'TCP'
      port: 80 # 외부로 노출하고자 하는 port
      targetPort: 8080 # 컨테이너 내부의 port. app.js가 수신대기중
    #- protocol: 'TCP'
      #port: 443
      #targetPort: 443
  type: LoadBalancer # ClusterIP 고정값 , NodePort
```

Service.yaml

--- 구현은 통해
file.yaml 이 아래
.yaml 형태가 됨

⇒ master-deployment.yaml

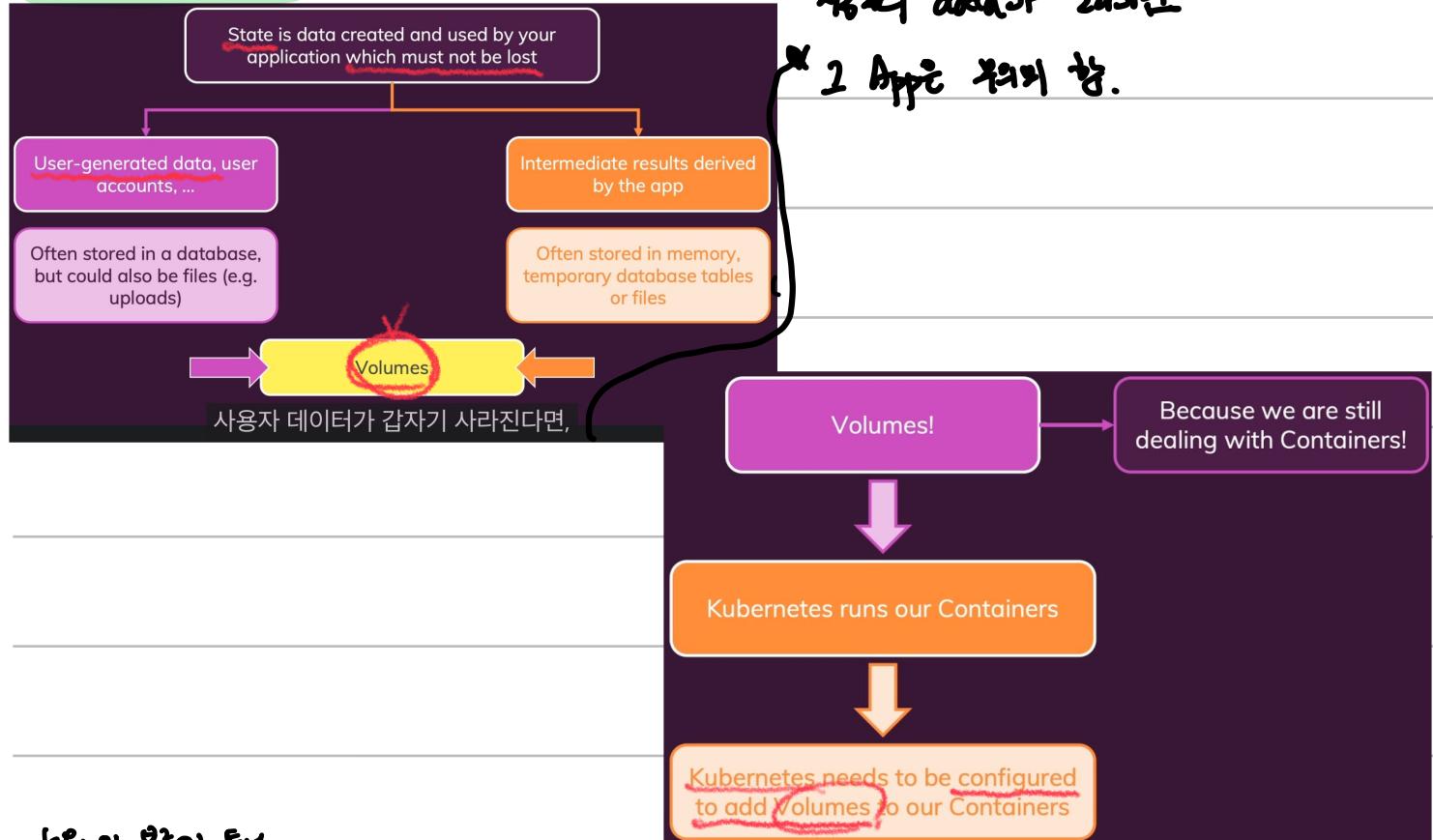
```
selector: # Deployment가 감시해야할 pod레이블의 키-값 쌍을 정의
# matchLabels:
#   app: second-app
#   tier: backend # tag느낌
matchExpressions:
  - {key: app, operator: In, values: [second-app]}
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: second-app-deployment
spec:
  selector: # Deployment가 감시해야할 pod레이블의 키-값 쌍을 정의
    matchLabels:
      app: second-app
      tier: backend # tag느낌
  replicas: 1 # pod를 scaling 개수
  template:
    metadata:
      labels:
        app: second-app
        tier: backend # tag느낌
    spec:
      containers: # -를 통해 pod 내 컨테이너를 구분
        - name: second-node
          image: datamaster/k8s-first-app
          resources:
            limits:
              memory: "128Mi"
              cpu: "500m"
            #ports:
            #- containerPort: <Port>
            #- name: ...
            # image: ...
```

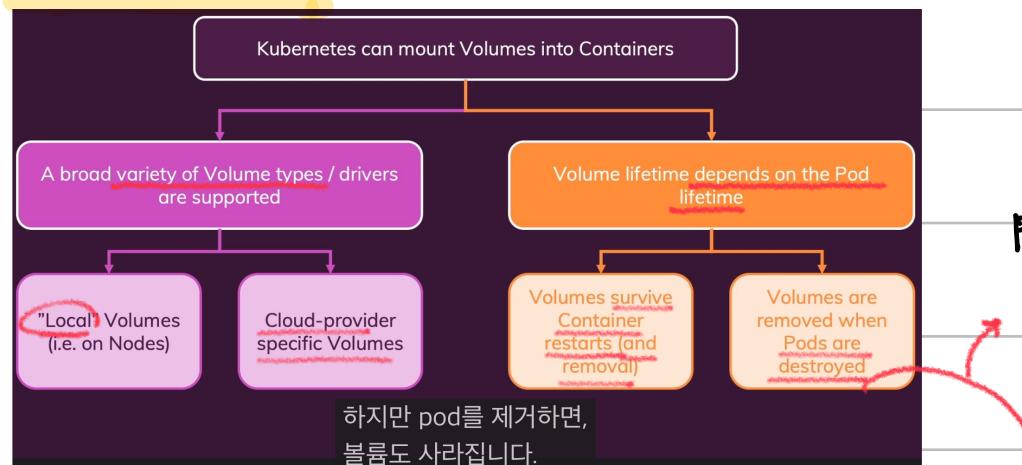
deployment.yaml

```
containers: # -를 통해 pod 내 컨테이너를 구분
- name: second-node
  image: datamaster/k8s-first-app
  livenessProbe: # pod 모니터링 설정
    httpGet:
      path: /
      port: 8080
    periodSeconds: 10 # 작업을 수행하는 빈도
    initialDelaySeconds: 5 # k8s가 처음으로 상태를 확인할 때까지의
```

• Dataft Volume



• k8s의 볼륨의 특성



pod가 삭제될 때 함께 삭제된다면,

볼륨이 삭제되는 경우가 많다.

Kubernetes Volumes	Docker Volumes
Supports many different Drivers and Types	Basically no Driver / Type Support
Volumes are not necessarily persistent	Volumes persist until manually cleared
Volumes survive Container restarts and removals	Volumes survive Container restarts and removals

• k8s의 볼륨 유형

① emptyDir

```

spec:
  containers:
    - name: story
      image: datamaster/k8s-data-demo:1
      resources:
        limits:
          memory: "128Mi"
          cpu: "500m"
      volumeMounts:
        - mountPath: /app/story
          name: story-volume
      volumes:
        - name: story-volume
          emptyDir: {}

```

Arbitrary 디렉토리가 pod에 마운트되는 경우

② hostPath

```

containers:
  - name: story
    image: datamaster/k8s-data-demo:1
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    volumeMounts:
      - mountPath: /app/story
        name: story-volume
    volumes:
      - name: story-volume
        hostPath:
          path: /data # pod의 볼륨이 워커노드에 있음 => pod가 달라도 상관x
          type: DirectoryOrCreate

```

pod의 볼륨은 워커노드에 두어
Arbitrary 디렉토리가 pod에 마운트되는 경우

③ CSI \Rightarrow AWS EBS는 k8s의 볼륨으로 사용이 용이

Container Storage Interface

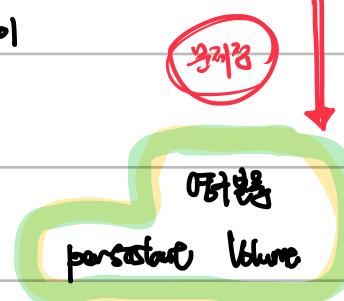
1 AWS Elastic Blockstore

AzureFile
AzureDisk

포맷팅 가능

minikube 및 외부
워커노드가 어떤 형태로
설정 cluster에서는? 볼륨의 위치?
 \rightarrow pod의 node에 속해있고 볼륨 동요성
ex. key, 파일 등 ...

\Rightarrow pod의 수명과 함께 동요됨.



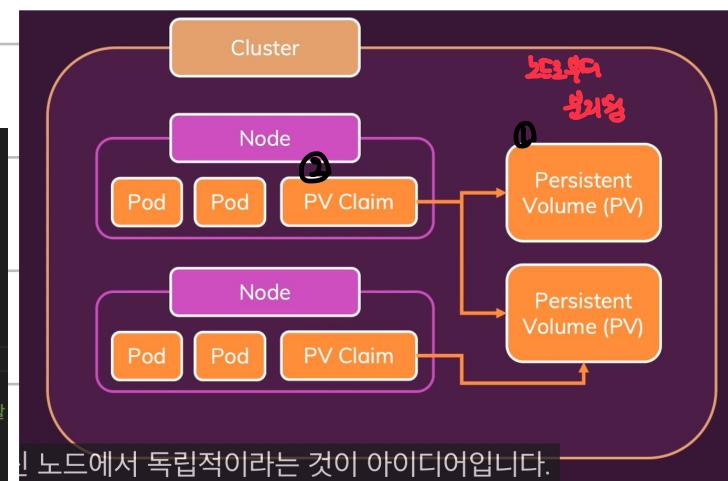
① 힘之城 : host-pv.yaml

```

### 단일 노드 환경 => hostPath 유형의 영구볼륨 ####
apiVersion: v1
kind: PersistentVolume
metadata:
  name: host-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem # or Block
  accessModes:
    - ReadWriteOnce # 파드가 '동일 노드'에서 구동되는 경우에는 복수의 파드에서 볼륨에 접근할 수 있다.
      #- ReadOnlyMany 볼륨이 '다수의 노드'에서 읽기 전용으로 마운트
      #- ReadWriteMany 볼륨이 '다수의 노드'에서 읽기-쓰기로 마운트
      #- ReadWriteOncePod 전체 클러스터에서 '단 하나의 파드'만 해당 PVC를 읽거나 쓸 수 있다.
  hostPath:
    path: /data
    type: DirectoryOrCreate # 아무것도 없으면 빈 디렉토리 생성

```

hostPath



kubectl get sc pv pvc

SC 별 노드의
PV 영구볼륨
위 호환

② 힘之城 claim : host-pvc.yaml

```

### 단일 노드 환경 => hostPath 유형의 영구볼륨 Claim ####
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: host-pvc
spec:
  volumeName: host-pv
  resources:
    requests:
      storage: 500mi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce

```

1Gi 미만

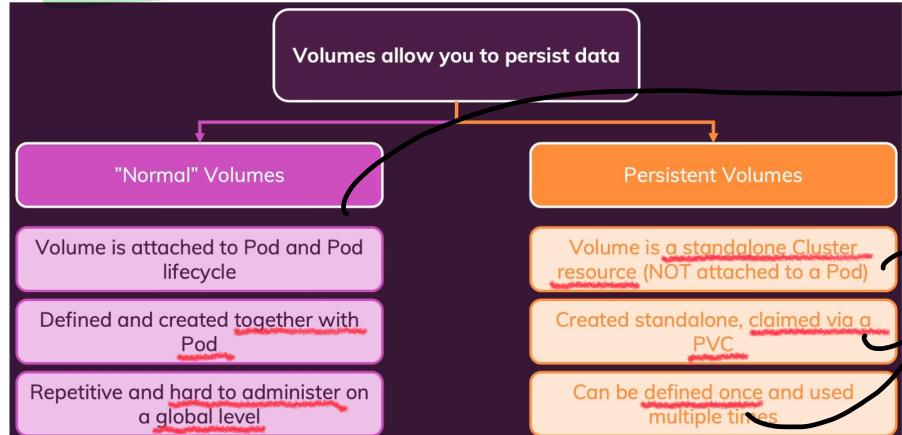
```

containers:
  - name: story
    image: datamaster/k8s-data-demo:1
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    volumeMounts:
      - mountPath: /app/story
        name: story-volume
    volumes:
      - name: story-volume
        persistentVolumeClaim: # 영구볼륨과 연결고리
          claimName: host-pvc

```

② deployment.yaml

• 일반 볼륨 vs 영구 볼륨



emptyDir: pod 종료 → 초기화
 hostPath: → 다른 저장소
 → Cloud 저장소
 기타:

캐시를 위한 예제

• 환경 변수 사용하기 → 외부값을 안전하고 유연하게 관리 가능 ex. API key, DB url

① value: 'story' ② valueFrom

```
containers:
- name: story
  image: datamasterr/k8s-data-demo:2
  env: # app.js에서 환경변수를 사용해서, .yaml에서만 유지보수 용이
    - name: STORY_FOLDER
      ① #value: 'story' => 직접 환경변수의 value 정의
      ② valueFrom: # environment.yaml에서 환경변수 value 가져옴
        configMapKeyRef:
          name: data-store-env
          key: folder
```

Environment.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: data-store-env
data:
  folder: 'story'
  # key: value ...
```

• Network

① pod 내 컨테이너끼리 통신 → localhost

외부 IP 설정
minikube service <서비스명>



users-deployment.yaml

```
spec:
  containers:
    - name: users
      image: datamasterr/k8s-demo-users
      env:
        - name: AUTH_ADDRESS
          value: localhost # pod 내 컨테이너간 통신
        - name: auth
          image: datamasterr/k8s-demo-auth
```

```
spec:
  selector:
    app: users
  ports:
    - protocol: 'TCP'
      port: 8080
      targetPort: 8080
    type: LoadBalancer
```

users-service.yaml

POST | http://127.0.0.1:56615/signup

Params Authorization Headers (8) Body

none form-data x-www-form-urlencoded

```

1 {
2   "email": "hyunwoo8504@gmail.com",
3   "password": "qwqw11"

```

Body tab selected. Response body: "message": "User created!"

POST | http://127.0.0.1:56615/login

Params Authorization Headers (8) Body

none form-data x-www-form-urlencoded

```

1 {
2   "email": "hyunwoo8504@gmail.com",
3   "password": "qwqw11"
4 }

```

Body tab selected. Response body: "token": "abc"

auth-app.js code:

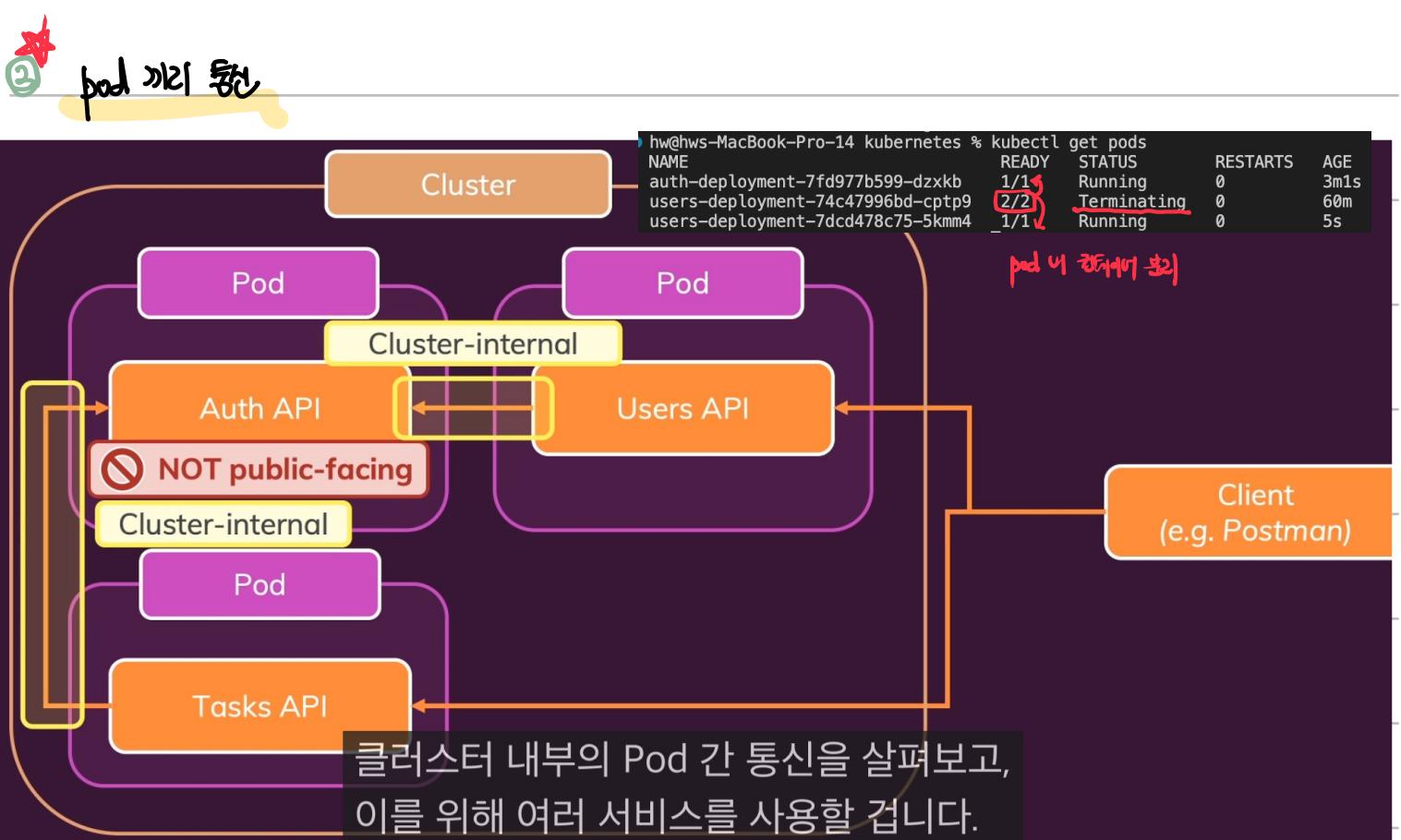
```

if(hashedPassword === enteredPassword + '_hash'){
  const token = 'abc';
  return res.status(200).json({ message: 'Token created.', token: token });
}
res.status(401).json({ message: 'Passwords do not match.' });

```

const hashedPW = await axios.get(`http://\${process.env.AUTH_ADDRESS}/hashed-password/` + password);

↳ part 4 팀원들 팀 localhost (로컬 호스팅)



• Users-API ft Auth-API가 pod끼리 통신

```
hw@hws-MacBook-Pro-14 kubernetes % kubectl get services
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
auth-service   ClusterIP  10.107.79.251  <none>       80/TCP
2m7s
kubernetes     ClusterIP  10.96.0.1      <none>       443/TCP
12d
users-service  LoadBalancer 10.108.25.187  <pending>    8080:30409/TCP
89m
```

```
kind: Service
metadata:
  name: auth-service
spec:
  selector:
    app: auth
  ports:
    - protocol: 'TCP'
      port: 80
      targetPort: 80
  type: ClusterIP # 외부에 노출하지 않음
```

```
spec:
  containers:
  - name: users
    image: datamaster/k8s-demo-users
    env:
      - name: AUTH_ADDRESS # users-app.js의 환경변수
        value: localhost
        # pod 내 컨테이너 간 통신 #
      - name: value: "localhost"
        # pod끼리 통신 #
        value: "10.107.79.251" # auth-service의 ClusterIP / ClusterIP는 서비스 내부에서만 사용 가능 OR
        value: "auth-service.default" # 서비스 이름을 이용하여 통신 / default는 namespace / docker-compose와 유사
```

①

< 서비스이름, 네임스페이스 >

이 방법을 가장 추천

```
hw@hws-MacBook-Pro-14 kubernetes % kubectl get namespaces
NAME          STATUS  AGE
default       Active  13d
kube-node-lease  Active  13d
kube-public   Active  13d
kube-system   Active  13d
kubernetes-dashboard  Active  13d
```

• Tasks-API

```
spec:
  containers:
  - name: tasks
    image: datamaster/k8s-demo-tasks:v2
    env:
      - name: AUTH_ADDRESS # users-app
        value: "10.107.79.251" # auth-service의 ClusterIP / ClusterIP는 서비스 내부에서만 사용 가능 OR
        value: "auth-service.default"
      - name: TASKS_FOLDER
        value: "tasks"
```

const filePath = path.join(__dirname, process.env.TASKS_FOLDER, 'tasks.txt');

app.use(bodyParser.json());

const extractAndVerifyToken = async (headers) => {
 if (!headers.authorization) {
 throw new Error('No token provided.');
 }
 const token = headers.authorization.split(' ')[1]; // expects Bearer TOKEN

const response = await axios.get(`http://\${process.env.AUTH_ADDRESS}/verify-token/\${token}`);
 return response.data.uid;

/app/tasks/tasks.txt 파일은.

POST http://127.0.0.1:56475/tasks

Params • Authorization Headers (9) Body • Scripts Settings

None form-data x-www-form-urlencoded raw binary

Body

```
1 { "text": "Hello_K8s",
2   "title": "K8s Basic"
3 }
```

GET http://127.0.0.1:56475/tasks

Params Authorization Headers (7) Body Scripts Settings

None

Headers

Key	Value
<input checked="" type="checkbox"/> Authorization	Bearer abc

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Tasks loaded.",
3   "tasks": [
4     {
5       "title": "K8s Basic",
6       "text": "Hello_K8s"
7     }
8   ]
9 }
```

Frontend app.js 편

```

function App() {
  const [tasks, setTasks] = useState([]);
  const fetchTasks = useCallback(function () {
    fetch('http://127.0.0.1:56475/tasks', {
      headers: {
        'Authorization': 'Bearer abc' // 키스로 배운 코드
      }
    })
    .then(function (response) {
      return response.json();
    })
    .then(function (jsonData) {
      setTasks(jsonData.tasks);
    });
  }, []);
}

```

키스로 배운 코드

headers의 역할

1. 요청에 필요한 메타데이터 제공

- headers는 클라이언트가 서버에 전달하려는 요청에 대한 부가적인 정보를 포함합니다.
- 예: 요청의 인증 정보, 데이터 형식, 사용자 애이전트 등.

2. 인증 및 권한 관리

- 코드에서 사용된 Authorization 헤더는 인증 정보를 서버에 전달합니다.
- 서버는 이 헤더를 사용해 클라이언트의 요청이 유효한지 검증합니다.

코드에서 Authorization 헤더의 역할

javascript

```

headers: {
  'Authorization': 'Bearer abc' // 키스로 배운 코드
}

```

Copy code

1. Authorization 헤더

- Authorization은 서버에 요청을 보낼 때 클라이언트의 인증 정보를 전달하는 데 사용됩니다.
- Bearer abc는 인증 방식과 인증 토큰을 포함한 값입니다.
- Bearer는 OAuth 2.0에서 사용하는 인증 방식 중 하나로, 토큰 기반 인증을 의미합니다.
- abc는 서버가 클라이언트를 식별하기 위해 사용하는 액세스 토큰입니다.
- 서버는 이 토큰을 검증하여 요청한 사용자가 권한이 있는지 확인합니다.

2. 인증된 요청

- 서버는 요청이 포함된 Authorization 헤더를 확인하고, 토큰이 유효한 경우 요청을 처리합니다.
- 토큰이 유효하지 않거나 누락된 경우 서버는 401 Unauthorized 응답을 반환합니다.

리버스 프록시의 주요 기능

1. 로드 밸런싱:

- 여러 서버로 요청을 분산시켜 부하를 균등하게 분배.
- 예: Nginx를 사용하여 트래픽을 여러 백엔드 서버로 나눔.

2. 보안 강화:

- 클라이언트는 리버스 프록시와만 통신하며 백엔드 서버는 외부에 노출되지 않음.
- DDoS 공격 방어 및 방화벽 역할.

3. SSL 종료(Termination):

- HTTPS 요청을 처리하고 내부 서버와는 HTTP로 통신하여 서버 부하를 줄임.

4. 캐싱:

- 자주 요청되는 데이터를 미리 저장하여 응답 속도를 높임.
- 예: 정적 파일(이미지, CSS) 캐싱.

5. 실제 서버 IP 숨김:

- 클라이언트는 리버스 프록시를 통해 통신하기 때문에 실제 서버의 IP가 숨겨짐.



백엔드 서버

• 리버스 프록시를 이용하여 URL 핸드코딩 얹하지!

① app.js는 백엔드에서 설정되었던 URL과 필요

② nginx.conf로 리버스 프록시로 접속된 주소로 백엔드 접속

⇒ K8S의 서비스의 puerto IP 사용 가능

app.js

```

function App() {
  const [tasks, setTasks] = useState([]);
  const fetchTasks = useCallback(function () {
    fetch('/api/tasks', { // nginx를 통해 리버스 프록시를 설정하면 => url을 /api로 설정
      headers: {
        'Authorization': 'Bearer abc' // http요청에 토큰을 넣어주는 방법
      }
    })
    .then(function (response) {
      return response.json();
    })
    .then(function (jsonData) {
      setTasks(jsonData.tasks);
    });
  }, []);
}

```

work > frontend > conf > nginx.conf

```

server {
  listen 80;

  location /api/ {
    # proxy_pass http://127.0.0.1:64942/; => 클러스터 내부에서 통신하므로 잘못된 방법 / JS는 브라우저에서 실행되므로 URL이 맞음.
    proxy_pass http://tasks-service.default:8000/; # tasks-service와 클러스터 내부에서 통신. 8000포트를 이용하여 통신
  }

  location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /index.html =404;
  }

  include /etc/nginx/extr

```

포트

AWS로 K8s 배포

① EKS 클러스터 생성 → AWS 콘솔 이용 // (2024년 1월 기준) IAM → EKS - AutoCluster 이용

↳ CloudFormation을 활용해 VPC 네트워크 생성 → EKS용 AWS 서비스

↳ S3 URL : EKS 콘솔에서 "Create an Amazon VPC for your Amazon EKS cluster"

② Access key 복구해야, VSCode에서 AWS CLI 사용

↳ VS code 터미널에서 aws configure 설정

③ AWS EKS와 VScode 환경설정

```
hw@hws-MacBook-Pro-14:4. AWS EKS % aws configure  
AWS Access Key ID [*****650D]:  
AWS Secret Access Key [*****DG07]:
```

```
Default region name [ap-northeast-2]: ap-northeast-2  
hw@hws-MacBook-Pro-14:4. AWS EKS % aws eks --region ap-northeast-2 update-kubeconfig --name k8s-dep-demo  
Added new context arn:aws:eks:ap-northeast-2:211125475530:cluster/k8s-dep-demo  
to /Users/hw/.kube/config  
hw@hws-MacBook-Pro-14:4. AWS EKS % kubectl get pods  
No resources found in default namespace.
```

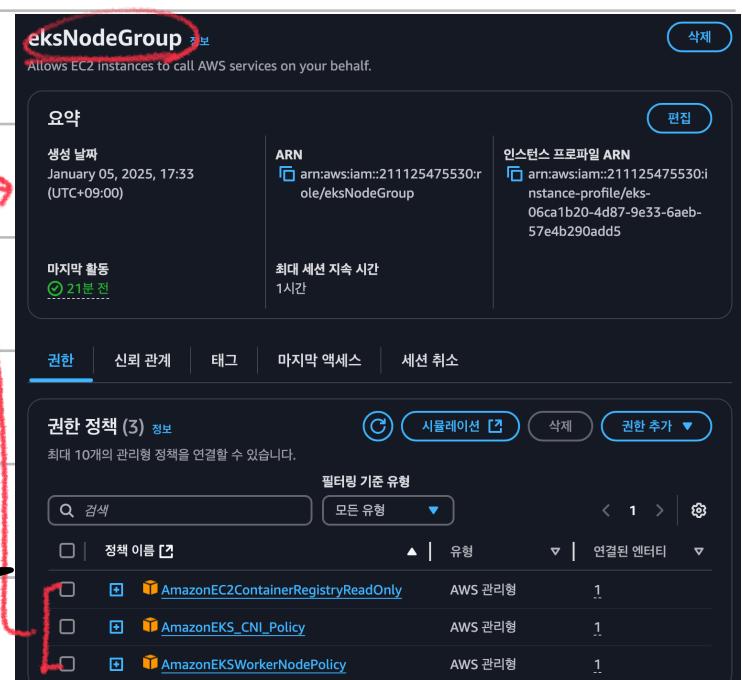
↳ EKS API 설정이 완료되었음을 알음

④ EKS → 컴퓨팅 → 노드 그룹 추가 : 워커노드추가

↳ Node의 업데이트 설정

↳ 최소 .small 모델로 워커노드 추가

최소 3개의 정책이
노드형에 표시로 표시



⑤ kubectl apply -f=<.yaml> 명령 ← minikube의 동작

⑥ kubectl get services → AWS의 외부 IP가 옴.

↳ postman으로 테스트

```
kubernetes > ! users.yaml > {} spec > {} template > {} spec > [ ] containers > {} 0 > [ ] env
```

```
18 spec:  
19   replicas: 1  
20   selector:  
21     matchLabels:
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

1: zsh

+ □ □ ^ X

```
users-api $ aws eks --region us-east-2 update-kubeconfig --name kub-dep-demo
```

```
Added new context arn:aws:eks:us-east-2:924744225256:cluster/kub-dep-dem
```

```
kubernetes $ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
auth-deployment-7458846c6f-5h942	1/1	Running	0	28s
users-deployment-867c65f96f-47f5c	1/1	Running	0	28s

```
kubernetes $ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
		PORT(S)	AGE
auth-service	ClusterIP	10.100.91.243 3000/TCP	<none> 36s
kubernetes	ClusterIP	10.100.0.1 443/TCP	<none> 22m
users-service	LoadBalancer	10.100.27.124 80:31337/TCP	a33415dfbbb7e452d8ee9b3ef26805f5-945247963.us-east-2.elb.amazonaws.com 35s

```
kubernetes $
```

POST a33415dfbbb7e452d8ee9b3ef26805f5-945247963.us-east-2.elb.amazonaws.com/signup

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "email": "test@test.com",  
3   "password": "testers"  
4 }
```

/signup

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "message": "User created.",  
3   "user": {  
4     "_id": "5f75ca176ff7c45d2a50326e",  
5     "email": "test@test.com",  
6     "password": "$2a$12$DKwfZhbAJrvem6/k/CYfw0uxdy7c00f0eAk5TI6mleJvkNMBpf2X.",  
7     "__v": 0  
8   }  
9 }
```

그러면 사용자가 생성되었다는 응답을 받습니다.

/login

POST a33415dfbbb7e452d8ee9b3ef26805f5-945247963.us-east-2.elb.amazonaws.com/login

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "email": "test@test.com",  
3   "password": "testers"  
4 }
```

POST a33415dfbbb7e452d8ee9b3ef26805f5-945247963.us-east-2.elb.amazonaws.com/login

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {  
2   "email": "test@test.com",  
3   "password": "testersssfasdfdas"  
4 }
```

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE2MDE1NTUwMTQsImV4cCI6MTYwMTU1ODYxNj0.xKBu",  
3   "userId": "5f75ca176ff7c45d2a50326e"  
4 }
```

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "message": "Request failed with status code 401"  
3 }
```

그리고 이 POST 요청을 보내면,
токن과 함께 응답을 받습니다.

무효한 비밀번호로 보내면,
오류가 반환됩니다.

⑦ 업로드 만들기 2. hostPath는 엔드포인트 아님이라 EKS의 부작용

→ AWS EFS 3. 파일 관리 by kubectl CSI

i) Installation 및 driver 설치

```

Just to make this really clear: We need this EFS driver since
AWS EFS is not supported as a volume type otherwise.

CONTAINERS:
  - name: users-api
    image: academind/kub-dep-users:latest
    env:
      - name: MONGODB_CONNECTION_URI
        value: 'mongodb+srv://maximilian:wk4nFupsbntPb3l@'
      - name: AUTH_API_ADDRESS
        value: 'auth-service.default:3000'

TERMINAL:
  kubectl apply -k "github.com/kubernetes-sigs/aws-efs-csi-driver/deploy/kubernetes/overlays/stable/?ref=release-1.0"
  daemonset.apps/efs-csi-node created
  csidriver.storage.k8s.io/efs.csi.aws.com created
  kubernetes $ 
  
```

이렇게 하고, 이제 EFS,
단역적 파일 시스템을 만들어야 합니다.

ii) 별도 경로 생성

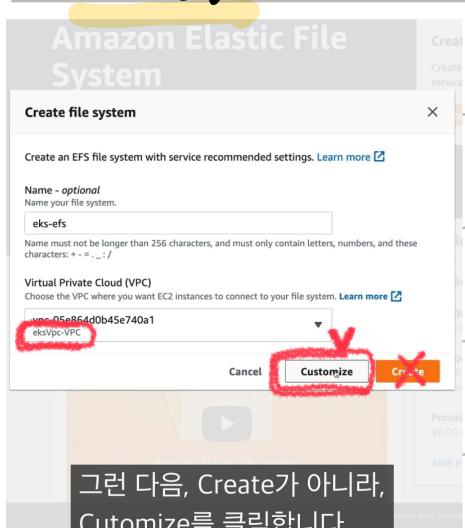
이동: eks-efs

VPC: eks VPC → EKS의 VPC

Amazon EFS → Type : NFS ... 2049

Source : Custom → eks VPC의 IPv4 CIDR 주소 ex) 192.168.0.0/16

iii) EFS 생성



Network

Virtual Private Cloud (VPC)
Choose the VPC where you want EC2 instances to connect to your file system. [Learn more](#)

vpc-05e864d0b45e740a1
eksVpc-VPN

Mount targets

A mount target provides an NFSv4 endpoint at which you can mount an Amazon EFS file system. We recommend creating one mount target per Availability Zone. [Learn more](#)

Availability zone	Subnet ID	IP address	Security groups
us-east-2a	subnet-04d...	Automatic	Choose secu... Remove sg-06eba42c0 af0a3ab4 eks-efs
us-east-2b	subnet-090f...	Automatic	Choose secu... Remove sg-06eba42c0 af0a3ab4 eks-efs

Add mount target

iv) volume yaml 생성

```

spec:
  containers:
    name: users-api
    image: datamaster/kub-dep-users
    env:
      - name: MONGODB_CONNECTION_URI
        value: 'mongodb+srv://<db_password>@hello-eks.pags.mongodb.net/?retryWrites=true&w=majority&appName=Hello-EKS'
      - name: AUTH_API_ADDRESS
        value: 'auth-service.default:3000'
    volumeMounts:
      - name: efs-vol
        mountPath: /app/users # Dockerfile에서 설정한 path에서 /users 추가
  volumes:
    - name: efs-vol
      persistentVolumeClaim: # EFS PVC yaml로 생성 필요함
        claimName: efs-pvc
  
```

deployment.yaml → deployment.yaml

```

apiVersion: v1
kind: StorageClass
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
---

apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem # EFS only supports Filesystem
  accessModes:
    - ReadWriteMany # 다른 pod에서 읽기 쓰기 가능 <-> ReadWriteOnce
  storageClassName: efs-sc # EFS StorageClass -> yaml로 StorageClass 생성 필요함
  csi: # EFS CSI Driver
    driver: efs.csi.aws.com
  volumeHandle: fs-0a4b3b3b # AWS EFS File System ID

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-pvc
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: efs-sc
  resources:
    requests:
      storage: 5Gi

```

Volume.yaml

CSI

↳ 제공자 지정 가능

ex. AWS EFS

Have filesystem

:

A Solid Docker + Kubernetes Foundation



You know what Docker is and why we use it



Docker can be used locally (development) and in production – you can do both or just one



Docker = Images + Containers



Docker-Compose helps with complex, multi-container projects – especially locally



Kubernetes helps with multi-machine container orchestration and deployment