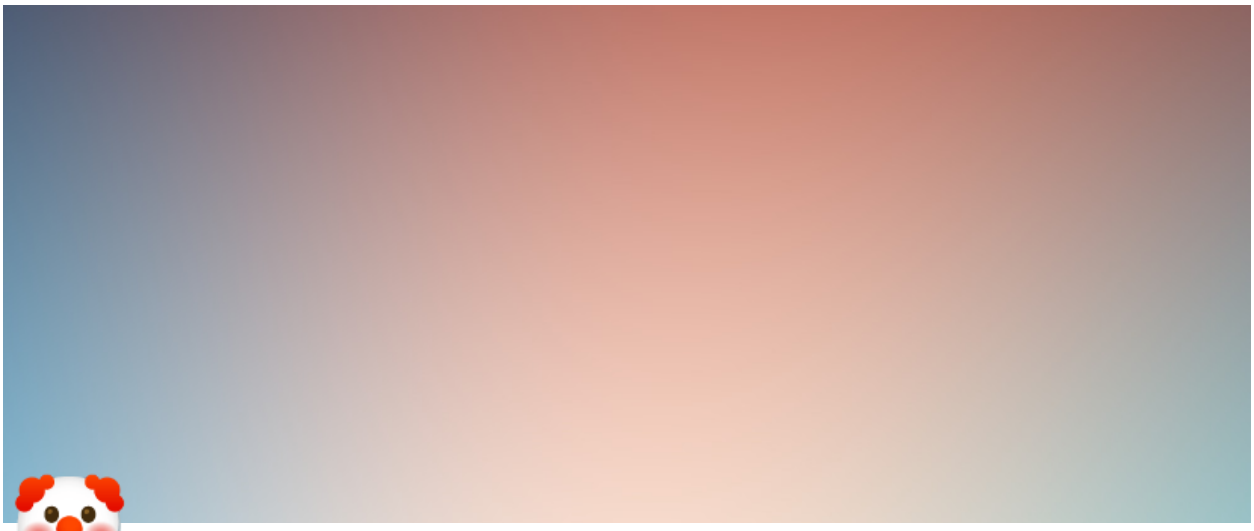

0. K8s for absolute beginers (Udemy)

💡 **Udemy 강의:** https://www.udemy.com/course/learn-kubernetes/learn/lecture/9723214?start=0#overview

실험실: https://kodekloud.com/topic/labs-familiarize-with-lab-environment-2/

Sec2-8

Containers Overview

Container?

Docker 같이 유명한 컨테이너 플랫폼이 있다.

왜 도커가 필요하며, 뭘 할 수 있는지 알아보자.

어플리케이션을 개발한다고 했을 때, Golang, Redis, MongoDB, Ansible 등 다양한 툴들을 조합하여야 한다. OS레벨에서 각 요소별로 호환되는 라이브러리 및 의존성들이 존재하는데 이들을 모두 만족시키는 것은 어려운 일이다.

개발/검증/운영계 환경이 다를 수 있으며 제각각 호환되는 각 요소들의 버전들이 존재하므로, 이를 만족시키는 환경을 구성하는 것은 어려운 일이다.

즉, compatibility 이슈가 발생할 수 있다.

이 이슈는 도커를 사용해서 각 구성요소들을 컨테이너화 하고, 각각을 독립적으로 라이브러리와 의존성을 구성할 수 있게끔 할 수 있는 기술로, 이러한 모듈간 호환성 이슈를 잠재울 수 있다.

컨테이너란 ?

컨테이너는 완전히 격리된 공간이다.

동일 OS 커널을 공유하는 도커 상에서 컨테이너는 각자 격리된 환경 내에서 각 요소들이 동작하도록 하는 기술이다.

OS kernel + software → 도커 아래에서 동작.

컨테이너들은 OS kernel 을 공유한다. 이 의미가 무엇일까?

도커는 OS 를 가상화하는 것이 아니다. 도커의 주요 목적은 어플리케이션을 컨테이너화 하고, 이를 실어서 구동시키는 것이다. 이것이 VM 과 차이다.

즉, 도커 컨테이너의 구성은

H/W → OS → Docker → Container (Libs/Deps → Application)

VM은

H/W → OS → Hypervisor → VM (OS → Libs/Deps → Application)

구성의 차이다.

VM 은 자원 사용량이 많고 사이즈가 크며 부팅 속도가 느리다.

도커 허브/스토어 상에 이미지들이 업로드 되어있고, 이 이미지들을 사용해서 이미지들을 받아 인스턴스들을 구성하고, 서비스를 구성할 수 있다.

컨테이너 vs 이미지

1) 도커 이미지 ? VM 템플릿처럼 도커 컨테이너들을 만드는 템플릿이라고 보면 된다.

2) 도커 컨테이너? 격리된 환경에서 실행중인 도커 이미지 인스턴스를 말한다. 일련의 프로세스들을 지닐 수 있다.

DevOps 관점에서 봤을 때,

개발자들이 개발한 형상이 동작하기 위한 제반 환경/인프라를 도커 이미지로 생성해서 운영팀에 전달하면 도커 컨테이너 상에서 실행될 수 있다는 보장이 된다. 이는 컨테이너 방식의 장점이다.

운영팀은 이 이미지를 갖고 배포하는데 사용할 수 있으며, 검증/운영계 동일 방식으로 배포할 수 있다.

챕터 9. 컨테이너 오케스트레이션

도커 컨테이너는 이제 이해했다. 도커 이미지가 동작하는 인스턴스.

도커 컨테이너 간 상호작용은 어떻게 할까? 스케일 아웃같은 것은?

이를 가능하게 하는것이 컨테이너 오케스트레이션이다.

이 컨테이너 오케스트레이션 기술은 대표적으로 쿠버네티스가 있다. 도커 스웜도 있고 MESOS 도 있다.

컨테이너 오케스트레이션 장점

1. 어플리케이션이 H/W failure 되어도 매우 높은 고가용성 때문에, 어플리케이션이 다운되는 일이 발생하지 않
   도록 해준다. 다른 노드에서 동작하는 컨테이너들이 존재하기 때문에.
   더 많은 컨테이너가 필요하다면 더 많은 인스턴스들을 수 초 내 배포할 수 있다.
   이를 서비스 레벨에서 수행할 수 있다.
   컨테이너 노드의 개수를 늘리거나 줄이는 것도 어플리케이션 다운 없이 가능하다.

즉, 컨테이너 오케스트레이션 기술의 장점은! 수백 수천개 컨테이너가 존재하는 클러스터 환경에서 배포와 관리를 용이하게 해준다는 것이다.

챕터 10. 쿠버네티스 구조 (’23.04.02)

개념 정리부터 하고 가자.

<1> 쿠버네티스 클러스터 구성요소

(1) 노드 (Node) : 노드는 쿠버네티스가 설치된 가상/물리적 장비를 의미한다.

노드는 워커 머신이며 쿠버네티스에 의해 도커 컨테이너가 실행되는 장소이다.

노드는 과거 ‘미니언’이라고도 불리었다 .

그런데, 만약 도커 컨테이너가 돌고있는, 즉 어플리케이션이 돌고있는 이 노드에서 어플리케이션이 죽으면 어떻게 될까? 이런 경우를 대비해서, 노드를 한개 보다 더 많이 동작시켜야 한다.

(2) 클러스터 : 함께 그룹화된 노드들을 클러스터라고 부른다.

클러스터를 구성하면 다른 노드로부터 요청이 처리될 수 있을 뿐 아니라, 부하분산의 효과도 있다.

(3) 마스터 : 클러스터를 통해 다중 노드가 구성된 상태이다. 이 상황에서 클러스터를 관리하는 주체가 누가 될까?

또, 클러스터를 구성하는 노드들에 대한 정보는 어디에 저장될까?

노드상태는 어떻게 모니터링되며, 만약 노드 하나가 다운될 경우 해당 노드로 흐르던 트래픽을 다른 워커노드로 어떻게 전달시킬 수 있을까?

'마스터' 노드는 이런 의문점에 대한 해답을 주는 노드이다.

워커노드와 동일하게 마스터노드도 쿠버네티스가 설치된 장비이며, 마스터로 지정된 노드이다.

이 마스터노드는 아래같은 임무가 있다.

- 클러스터에 존재하는 노드들을 관찰한다.
- 워커노드들을 오케스트레이션 하는 주체이다.

<2> 쿠버네티스를 이루는 '컴포넌트'

쿠버네티스를 설치한다는 건 아래 컴포넌트들을 설치한다는 것과 동일한 말이다.

- API server
  → 쿠버네티스에 있어서 프론트엔드와 동일한 역할을 한다.
  사용자, 관리장비, CLI 모든 것들은 쿠버네티스 클러스터와 통신하기 위해서 이 API Server 와 상호작용한다.

- etcd server (= etcd key store)
  → 쿠버네티스에 의해 사용되는 분산/신뢰가능한 키스토어로, 클러스터 관리를 위해 사용되는 모든 데이터들이 저장된다.
  → e.g., 클러스터에 다중 노드와 다중 마스터노드가 존재한다고 가정했을 때, etcd 는 클러스터에 있는 모든 노드들에 이런 정보들을 분산된 방법으로 저장한다.
  → etcd 는 클러스터 내 lock 구현에 대한 책임이 있는데, 이는 마스터들 간 자원접근에 대한 충돌을 발생시키지 않게하기 위한 목적이다.
- kubelet service
  → 클러스터 내 각 노드상에서 동작하는 에이전트이다.
  → 이 에이전트는 노드상의 컨테이너(들)이 예상대로 잘 동작하고 있는지 확인하는 것을 책임진다.
- Container runtime
  → 컨테이너들을 실행하기위해 사용되는 기저 소프트웨어다.
  → e.g., 도커
- Controller
  → 오케스트레이션에 있어서 '뇌'의 역할을 한다.
  → 노드/컨테이너/엔드포인트가 다운되었을 때 이를 감지하고 대처하는데 있어 책임이 있다.
  즉, 이런 장애상황에 있어서 새로운 컨테이너들을 띄우는 것에 대한 의사결정을 내린다.
- Scheduler
  → 복수의 노드들간에 있어서 작업 또는 컨테이너들을 분배하는 책임이 있다.
  → 새로 생성된 컨테이너들을 관찰하고 이를 노드들에 할당한다.

자, 여기까지 쿠버네티스에는 두 가지 종류의 서버가 존재한다는 것을 공부했다.

하나는 워커, 그리고 다른 하나는 마스터다.

또, 쿠버네티스를 이루는 여섯개의 컴포넌트들에 대해서도 공부를 했다.

그렇다면, 이런 컴포넌트들이 다른 타입의 서버들에 어떻게 분배되어 동작하는 것일까??

즉, 어떤 노드가 마스터가되고, 어떤 노드가 워커노드가 되는 것일까?

<3> 마스터 노드와 워커 노드 구성

워커 노드는 (도커) 컨테이너가 호스트되는 노드이다.

이 컨테이너를 실행시키기 위해서는 위에서 아래 컴포넌트들이 구성되어야 한다.

(1) container runtime : 이 강좌에서는 '도커'가 해당된다.

(2) </> kubelet agent : 마스터 노드와 상호작용하기위해 필요하다. 컨테이너 헬스 정보를 마스터로 보내거나, 마스터에의해 요청된 명령을 수행하기 위해 필요하다.

마스터 노드는 아래 컴포넌트들이 필요하다.

(1) </> kube API server : 이것이 바로 이 노드를 마스터 노드로 만들어주는 것이다.

(2) etcd : 모든 수집된 정보들은 이 키밸류 스토어에 저장한다.

(3) controller (control manager) :

(4) scheduler :

<4> Kubectl

CLU (Command Line Utility) 중 하나인 kubectl 을 공부해야한다.

(**kubectl** = kube command line tool = kube control)

이 kubectl 은 쿠버네티스 클러스터에 어플리케이션을 배포하고 관리하는데 사용된다.

클러스터 정보를 확인하거나, 클러스터 내 다른 노드들의 상태를 확인하거나, 다른 것들을 관리하기 위해 사용될 수 있다.

관련 명령어 몇 가지를 살펴보자.

(1) **kubectl run** 'applicationname' : applicationname 에 해당하는 어플리케이션을 클러스터에 배포하는 명령어다.

(2) kubectl cluster-inf : 클러스터에 대한 정보를 확인하는데 사용되는 명령어다.

(3) kubectl get nodes : 클러스터를 구성하는 노드들의 목록을 확인하는 명령어다.

Section 3. 쿠버네티스 컨셉

14강. Pods

가정. 도커 이미지와 쿠버네티스 클러스터가 구성된 상황이라고 가정하자.

즉, 어플리케이션이 이미 개발되어 도커 이미지 형태로 빌드되었고, 도커 허브같은 도커 리포지토리에서 이용가능하여 쿠버네티스가 이를 pull 할 수 있는 상태라고 가정하자.

이전 강의들에서 학습한 것처럼, 쿠버네티스를 사용하는 궁극적 목적은 우리 어플리케이션을 컨테이너 형태로 여러 머신들에 배포하기 위함인데, 이 머신들은 클러스터 내 워커 노드들로 구성된다.

하지만, 쿠버네티스는 컨테이너들을 직접적으로 워커노드들에 배포하지 않는다!

컨테이너들은 파드(pod) 라고 불리는, 쿠버네티스 객체 내에 캡슐화 된다!

Pod 는 하나의 어플리케이션에 대한 인스턴스이다.

Pod 는 쿠버네티스에서 생성할 수 있는 가장 작은 단위이다.

그럼, 또 다른 상황을 가정해서 웹 어플리케이션을 추가 투입하는 상황을 고려해보자.

단일 워커노드로 구성된 클러스터에서 Pod 한 개로 동작하던 우리 어플리케이션이 있다고 가정하자.

이 때, 어플리케이션을 하나 더 투입해야 한다면 Pod 내 동일 WAS 를 추가하는 것일까?

아니다. 동일 노드에 별개의 Pod 생성을 통해 WAS 를 추가하는 형태가 되어야 한다.

그럼에도 불구하고, 만약 유저가 더 늘어서 현재 단일 노드로 트래픽 감당이 되지 않을경우 어떻게 해야할까?

그러면 클러스터에 다른 워커노드 및 해당 노드안에 새로운 파드를 생성하여 클러스터의 물리적 용량 자체를 확장시켜야 한다.

즉, 이 예시의 핵심은, 파드는 컨테이너와 1:1 관계로 봐야 한다는 것이다.

그래서, 스케일 업/다운은 이 파드를 추가하거나 삭제하는 행위가 된다.

(파드 내에 동일 성격의 어플리케이션 컨테이너를 추가하여 어플리케이션을 스케일링 하지 않고 독립적으로 어플리케이션 컨테이너를 갖는 별개의 파드를 노드에 추가하여 스케일링을 한다.)

단일 파드는 복수개의 동일한 컨테이너를 포함하지 않는것이 일반적이며, 파드 내 다른 성격의 컨테이너들을 포함할 수 있다.

참고로, 파드가 죽으면 파드내 컨테이너들이 죽는다.

파드 내 두 컨테이너들은 서로 통신할 수 있으며, 같은 네트워크 스페이스를 공유하기 때문에, 로컬호스트 형태로 서로를 호출할 수 있다. 또한 스토리지도 공유한다.

그러나, 다중 컨테이너들이 파드에 존재하는 구성은 거의 채택하지 않는 방식이기에, 이 강좌에서는 단일 컨테이너로 구성된 파드를 다룬다.

이전 강좌에서 살펴본 쿠버네티스 명령어가 실제 동작하는 것을 살펴보자.

1) kubectl run nginx -- image=nginx :

- step1 : pod 생성
- step2 : nginx 도커 이미지를 인스턴스(도커 컨테이너) 형태로 배포.
  그런데, 이 때 도커 이미지를 어떻게 가져올까? → 'image' 파라미터를 사용해 이름을 특정 지어서 가져올 수 있다. nginx 도커 이미지는 *도커 허브 리파지토리에서 다운로드된다. (**초록색 nginx 는 파드의 이름**이 된다.) 파드 이름은 아무것이나 할 수 있지만, image 파라미터 뒤에오는 도커 이미지 이름은 저장소에 있는 이름으로 해야만 한다.

* 도커 허브는 다양한 어플리케이션들의 최신 도커 이미지들을 저장하고 있는 퍼블릭 저장소다.
* 쿠버네티스 설정에서 퍼블릭 도커 허브 또는 프라이빗 저장소 중 어디서 이미지 다운로드를 할 지 설정할 수도 있다.

자, 파드를 생성하고, 도커 이미지를 다운받아서, 이를 인스턴스화 한 도커 컨테이너를 파드에 배포하였다.

이제 파드들의 리스트를 확인하기 위해서는 어떻게 해야할까?

2) kubectl get pods :

이 명령어는 클러스터 내 파드들의 리스트를 확인할 수 있게 한다.

강의 예시에서는 이 명령어의 결과가
ContainerCreating (파드 내 컨테이너 생성 중인 상태) → Running (파드 내 컨테이너가 동작중인 상태)

로 바뀌어, 파드의 리스트 및 파드 내 컨테이너의 상태 정보를 확인할 수 있는 것을 알 수 있었다.

(하지만 현재까지 단계로서는 노드 내 파드 및 파드 내 컨테이너를 배포, 실행한 것이고, end-user 가 Nginx 컨테이너에 접근하는 것을 할 수는 없는 단계다. 이 과정은 나중 강의에서 다루게 된다.)

15강. Demo - Pods

kubectl run nginx --imgae=nginx : 도커 이미지 다운로드 및 파드 생성, 파드 내 컨테이너 배포

kubectl get pods : pod 이름, pod 내 컨테이너 개수, 재시작 횟수, 시작 후 경과시간 표시.

kubectl describe pod nginx : pod에 대한 상세 정보 출력. 파드 이름, 파드가 속한 노드의 이름 및 IP 주소, 시작시간, 파드 IP, 파드 내 컨테이너에 대한 ID, image, image ID, events (pod 생성 후 이벤트들)

kubectl get pods -o wide : pod의 내부 IP 정보 추가로 확인 가능.

Reference - Pods

A note about creating pods using **kubectl run**.

You can create pods from the command line using the below command:

Create an NGINX Pod

```
kubectl run nginx --image=nginx
```

Kubernetes Concepts - https://kubernetes.io/docs/concepts/

Pod Overview- https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

섹션 4. YAML 소개

17강. YAML 소개

YAML 이란 뭘까?

야믈이라고 읽는다.

XML, JSON 과 같이 사람이 쉽게 읽을 수 있는 데이터 직렬화 양식이다.

쿠버네티스에서는 주로 이 YAML 을 어떻게 활용하냐면 설정 값을 이 YAML 로 관리하기 때문에 YAML에 대해서 알 필요가 있다.

예시로 아래처럼 YAML을 통해 데이터를 표현할 수 있다.

<1> key-value pair

Vegetable: Carrot

Meat: Chicken

이처럼 '키:(스페이스)밸류' 형태로 YAML 데이터를 표현할 수 있다.

<2> Array/Lists

아래처럼 대쉬(-)를 활용해서 배열의 요소들을 표현할 수 있다.

Fruits:

- Orange
- Apple
- Banana

Vegetables:

- Carrot
- Tomato

<3> Dictionary/Map

Banana:

Calories: 105

Fat: 0.4g

Carbs: 27g

*YAML에서는 공백이 중요하다. 공백의 개수로 인해 다른 필드의 하위 항목이 될 수도 있는 만큼, 공백의 개수에 유의해야 한다.

아래 예시를 살펴보자

사전 자료형을 배열로 갖는 YAML

Fruits:

- Banana:
    - Calories: 105
    - Fat: 0.4g
    - Carbs: 27g
- Grape:
    - Calories: 62
    - Fat: 0.3g
    - Carbs: 16g

배열, 사전, 키-밸류 를 학습했다.

언제 배열을 쓸지, 사전을 쓸지, 사전으로 이루어진 배열을 쓸지 어떻게 결정할까?

결론은 표현하고자 하는 데이터에 따라 다르다 !

Answers to all coding exercises can be found here:

https://github.com/mmumshad/kubernetes-training-answers

YAML - 6

Update the YAML file to include Jacob's pay slips. Add a new property "**Payslips**" and create a list of pay slip details (**Use list of dictionaries**). Each payslip detail contains **Month** and **Wage**.

Month	Wage
June	4000
July	4500
August	4000

Employee:
Name: Jacob
Sex: Male
Age: 30
Title: Systems Engineer
Projects:
- Automation
- Support
Payslips:
- Month: June
** Wage: 4000**
- Month: July
** Wage: 4500**
- Month: August
** Wage: 4000**

섹션5. 쿠버네티스 컨셉 - 파드, 레플리카셋, 배포

20강. Pods with YAML

쿠버네티스는 YAML 을 입력 파일로 여러 목적으로 사용될 수 있는데, 파드, 레플리카, 서비스 배포같은 목적들이 있다.

이런 모든 종류의 목적은 비슷한 형태를 따르는데, 4개의 필드들로 구성된다.

apiVersion: 문자열
→ 쿠버네티스 API 버전 (POD: v1, Service: v1, ReplicaSet: apps/v1, Deployment: apps/v1)

kind: 문자열
→ 우리가 생성하고자 하는 객체의 타입을 의미한다. (e.g., POD, Service, ReplicaSet, Deployment)

metadata: 사전 자료형
→ 객체에 대한 데이터들을 의미한다. 예를들어, name, labels 같은 필드들이 존재할 수 있다.

중요한 점은, 쿠버네티스에서 이 metadata 의 필드로 사용 가능한 것들만 지정할 수 있다는 것이다. (name, labels, etc.) 그러나 labels 하위로는 원하는 키밸류 쌍을 원하는대로 설정할 수 있다.

spec: 사전 자료형
→ 생성하려는 오브젝트가 어떤 것이냐에 따라 달라지며, 쿠버네티스가 이를 참고하여 오브젝트 생성하기 때문에 중요하다.

→ 하위 항목으로 'containers' 가 있는데 이는 배열 자료형이다. 왜냐하면 파드 내에 여러개의 컨테이너들이 존재할 수 있기 때문이다. 'containers' 하위 요소 항목으로는 name, image 가 있다.

여기까지 모든 YAML 작성이 완료되면, 아래 명령어로 쿠버네티스를 사용해서 파드 생성이 가능하다.

```
kubectl create -f pod-definitinon.yml
```

파드를 생성하고 난 다음에 파드를 확인하기 위해서는 어떻게 할까?

```
kubectl get pods
```

파드 'myapp-pod'에 대한 상세한 정보를 확인하고 싶으면,

```
kubectl describe pod myapp-pod
```

예시로 Labs 에서 아래처럼 pod.yml 파일을 만든 뒤,

```
controlplane ~ ✦3 ✖ cat pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
    tier: frontend
spec:
  containers:
    - name: nginx
      image: nginx
```

kubectl create (또는 apply) -f pod.yml 하고나서 파드 정보를 출력시키면 아래처럼 나온다.

```
controlplane ~ ✦3 ✖ kubectl create -f pod.yml
pod/nginx created

controlplane ~ ✦3 ➜  kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          41s
```

좀더 상세한 정보를 보고 싶으면

```
controlplane ~ ✖3 ➜  kubectl describe pod nginx
Name:             nginx
Namespace:        default
Priority:         0
Service Account:  default
Node:             controlplane/172.25.0.18
Start Time:       Sun, 02 Apr 2023 08:39:14 +0000
Labels:           app=nginx
                  tier=frontend
Annotations:      <none>
Status:           Running
IP:               10.42.0.9
IPs:
  IP:  10.42.0.9
Containers:
  nginx:
    Container ID:   containerd://8196471354960b31e9937eda7a133b8cd96ddd0fa790f695da2f4c0fd40ed91f
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:2ab30d6ac53580a6db8b657abf0f68d75360ff5cc1670a85acb5bd85ba1b
0
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Sun, 02 Apr 2023 08:39:21 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-lljn4 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-lljn4:
    Type:                     Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:   3607
    ConfigMapName:            kube-root-ca.crt
    ConfigMapOptional:        <nil>
    DownwardAPI:              true
QoS Class:                    BestEffort
Node-Selectors:               <none>
```

```
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  2m37s  default-scheduler  Successfully assigned default/nginx to controlplane
  Normal  Pulling    2m36s  kubelet            Pulling image "nginx"
  Normal  Pulled     2m30s  kubelet            Successfully pulled image "nginx" in 5.892807831s (5.892831623s inc
luding waiting)
  Normal  Created    2m30s  kubelet            Created container nginx
  Normal  Started    2m30s  kubelet            Started container nginx

controlplane ~ ✖3 ➜
```

처럼 해서 볼 수도 있다.

https://www.udemy.com/course/learn-kubernetes/learn/quiz/4406758#overview

Pods 생성을 위한 YAML 설정을 할 수 있다.

apiVersion :

kind :

metadata :

spec:

예시 문제 1

Instruction: Create a Kubernetes Pod definition file using values below:

- **Name:** postgres
- **Labels:** tier => db-tier
- **Container name:** postgres
- **Image:** postgres

예시 문제 1 정답

```
apiVersion: v1
kind: Pod
metadata: #dict
   name: postgres
   labels:
      tier: db-tier
spec: #array
   containers:
      - name: postgres
        image: postgres
```

예시 문제 2

Pods - 9

Introduction: Postgres Docker image requires an environment variable to be set for password.

Instruction: Set an environment variable for the docker container. **POSTGRES_PASSWORD** with a value **mysecretpassword**. I know we haven't discussed this in the lecture, but it is easy. To pass in an environment variable add a new property '**env**' to the container object. It is a sibling of image and name. **env** is an array/list. So add a new item under it. The item will have properties **name** and **value**. **name** should be the name of the environment variable - **POSTGRES_PASSWORD.** And **value** should be the password - **mysecretpassword**

예시 문제 2 정답

```
apiVersion: v1
kind: Pod
metadata: #dict
    name: postgres
    labels:
        tier: db-tier
spec: #array
    containers:
        - name: postgres
          image: postgres
          env:
            - name: POSTGRES_PASSWORD
              value: mysecretpassword
```