

# 쿠버네티스- 실습가이드

최초 작성일 : 2024/02/27

최종 제출일 : 2024/02/28

김민경

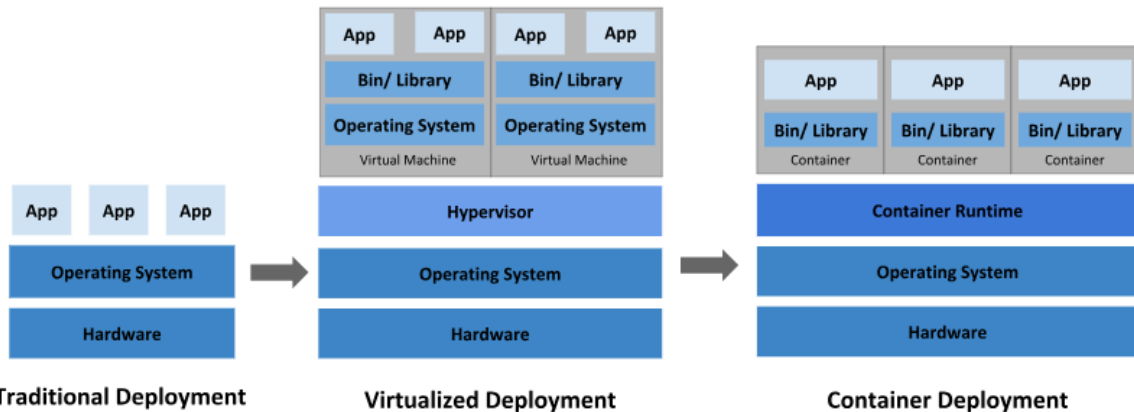
## 목차

- I. 개념정리-----2
  - 1. 쿠버네티스-----2
    - 1) 등장배경-----2
    - 2) 개념-----2
    - 3) 구성요소-----2
  - 2. Docker Compose -----5
    - 1) 개념-----6
    - 2) yaml 파일 작성법-----6
- IV. kubeadm으로 클러스터 구성하기 -----6
  - 1. 환경구성-----6
  - 2. 모든 서버에 공통으로 설정해야 할 부분-----7
  - 3. 마스터에 설정해야 할 부분-----13

# I. 개념정리

## 1. 쿠버네티스

### 1) 등장배경



- 다수의 물리서버를 계속 추가하기에는 **비용 문제**가 발생했음
- 이를 해결하기 위해 '**가상화**'라는 개념이 도입됨
  - 하나의 물리서버를 논리적으로 구분하여 자원을 논리적으로 늘려 사용할 수 있게 됨
- but 논리적인 구성인만큼 속도가 조금 밀리는 현상이 발생함
  - so 가볍고 빠르게, 변화하는 정보를 반영하기 위해 '**컨테이너**'가 등장하게 됨
- but 점차 서비스의 수가 증가하고 다양한 환경에서 실행되며 **수동으로 많은 컨테이너를 관리하고 배포하기에 어렵게** 됨
  - so 이에 따라 컨테이너의 효율적인 활용과 관리를 위해 **컨테이너 오케스트레이션**이 등장함

#### 컨테이너 오케스트레이션

복잡한 컨테이너 환경을 효과적으로 관리하기 위한 도구

ex) 쿠버네티스, Docker Swarm, Apache Mesos

### 2) 개념

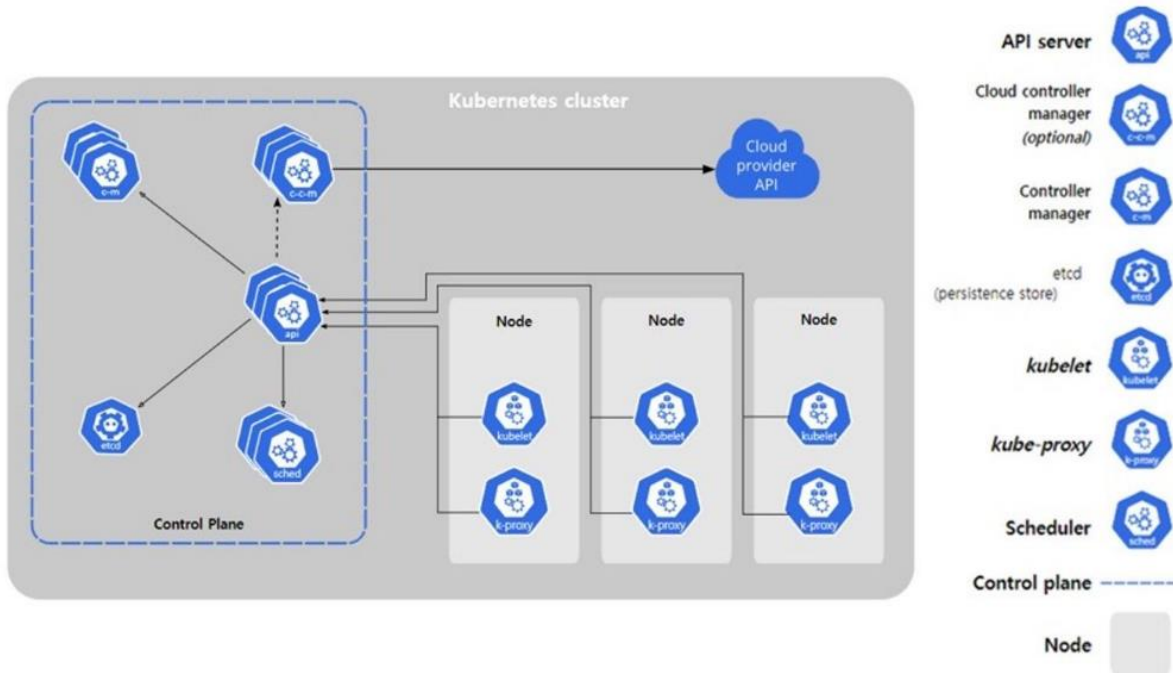
- 대규모 컨테이너를 관리, 조정 및 스케줄링할 수 있는 오픈 소스 컨테이너 오케스트레이션 소프트웨어

- 클라우드 네이티브를 가장 효과적으로 구현하기 위한 일환으로 사용되고 있음

#### 클라우드 네이티브

- 클라우드 컴퓨팅 환경에서 애플리케이션을 구축, 배포 및 관리할 때의 소프트웨어 접근 방식)

### 3) 구성 요소



- 쿠버네티스 컴포넌트를 크게 분류를 하자면, 쿠버네티스 기능 제어를 전체적으로 담당하는 **마스터 노드인 컨트롤 플레인(Control Plane) 컴포넌트**와 컨트롤 플레인 컴포넌트의 요청을 받아 각 노드에서 동작을 담당하는 **워커 노드(Node) 컴포넌트**로 나누어볼 수 있음

- 마스터 노드⇒ 컴포넌트회사 전반에 해당하는 '본사의 업무'
- 노드 컴포넌트⇒ 각 지역별 '지사의 업무'

마스터 노드	① kube-apiserver (REST API 서버)
	② etcd (정보 저장소)
	③ kube-scheduler (자리 배치)
	④ kube-controller-manager (상태 체크)
워커 노드	① kubelet
	② container runtime
	③ kube-proxy

## 1. 마스터 노드 (컨트롤 플레인 컴포넌트)

### ① kube-apiserver (REST API 서버)

- 마스터로 들어오는 모든 요청을 받아들이고, 응답하며, 요청을 보내는 REST API 서버

#### REST API

- 컴퓨터 시스템이 인터넷을 통해 정보를 안전하게 교환하기 위해 사용하는 인터페이스
- 쿠버네티스 커맨드 라인 도구인 kubectl 을 사용해 명령을 하면 kube-apiserver 로 전송됨. 이렇게 전달받은 요청은 kube-apiserver 가 적절한 컴포넌트로 전달함
- ex) 회사 첫 방문 시 안내데스크로 감. 방문 목적을 설명하면, 안내 데스크에서는 이 방문자가 적절한 방문자인지 시스템에서 검색하고 가야 할 곳을 안내해줌. 이런 역할을 하는 컴포넌트가 kube-apiserver 임

### ② etcd (정보 저장소)

- 정보를 보관해두는 곳 (어떤 노드에 어떤 컨테이너가 들어가 있는지에 대한 정보)
- 쿠버네티스 클러스터가 동작하기 위해서는 클러스터 및 리소스의 구성 정보, 상태 정보 및 명세 정보 등이 필요함
  - ⇒ etcd 가 이러한 정보를 키-값(key-value) 형태로 저장함
- 안정적인 동작을 위해 자료를 분산해서 저장하는 구조를 채택
- ex) 각종 중요 정보가 모두 모여 있는 금고와 같은 곳임. 매우 중요한 정보를 보관하기 때문에 동일한 자료를 여러 금고에 나눠서 보관하고 있다고 비유할 수 있음

### ③ kube-scheduler (자리 배치)

- 쿠버네티스 클러스터는 여러 노드로 구성되어 있고, 기본적인 작업 단위라고 할 수 있는 파드는 여러 노드 중 특정 노드에 배치되어 동작하게 됨.
  - 이때 새로 생성된 파드를 감지하여 어떤 노드로 배치할지 결정하는 스케줄링을 담당
  - ex) 회사로 비유하자면, 각 부서 인력 소요 계획과 신입사원 역량을 고려해 적절한 부서로 배치하는 인사 담당 부서라고 볼 수 있음

#### ④ kube-controller-manager (상태 체크)

- 지속적으로 control loop 를 돌면서 current state 와 desired state 를 비교함
  - current state 와 desired state 두 상태가 달라지면 current state → desired state 가 되도록 상태를 바꿔주는 역할을 함

다운된 노드가 없는지, 파드가 의도한 복제(Replicas) 숫자를 유지하고 있는지, 서비스와 파드는 적절하게 연결되어 있는지, 네임스페이스에 대한 기본 계정과 토큰이 생성되어 있는지를 확인하고 적절하지 않다면 적절한 수준을 유지하기 위해 조치하는 역할

ex) 회사에서는 목표 수준을 달성하기 위해 끊임없이 확인하고, 목표치에 미달하는 부분에 대해서는 대책을 수립함. 목표 달성한 후에도 이를 유지하려면 책임감 있는 관리자가 필요한데, 쿠버네티스에서 이 역할을 하는 컴포넌트가 kube-controller-manager 임

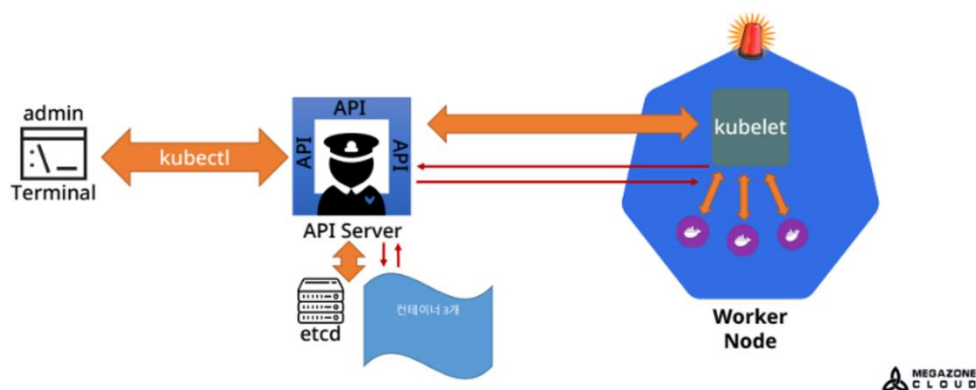
#### (2) 워커 노드 (노드 컴포넌트)

##### ① kubelet

노드에서 컨테이너가 동작하도록 관리해 주는 핵심 요소

각 노드에서 파드를 생성하고 정상적으로 동작하는지 관리하는 역할을 담당하고 있으며, 실제로 우리가 쿠버네티스의 워크로드를 관리하기 위해 내려지는 명령은 kubelet 을 통해 수행된다고 볼 수 있음

#### 쿠버네티스 동작 원리



우리가 쿠버네티스 파드를 관리하기 위해 작성하는 YAML 을 쿠버네티스 클러스터에 적용하기 위해 kubectl 명령어를 사용할 때, 이 YAML 이 kube-apiserver 로 전송된 후 kubelet 으로 전달됨

kubelet 은 이 YAML 을 통해 전달된 파드를 생성 혹은 변경하고, 이후 이 YAML 에 명시된 컨테이너가 정상적으로 실행되고 있는지 확인함.

ex) 지사에서 본사의 업무 요청을 받아 확인하는 막중한 역할을 수행하고 있다고 볼 수 있음

## ② container runtime

컨테이너 런타임은 파드에 포함된 컨테이너 실행을 실질적으로 담당하는 애플리케이션을 의미함

단, 쿠버네티스 구성 요소에 기본적으로 포함되어 있거나, 특정 소프트웨어를 지칭하는 것은 x

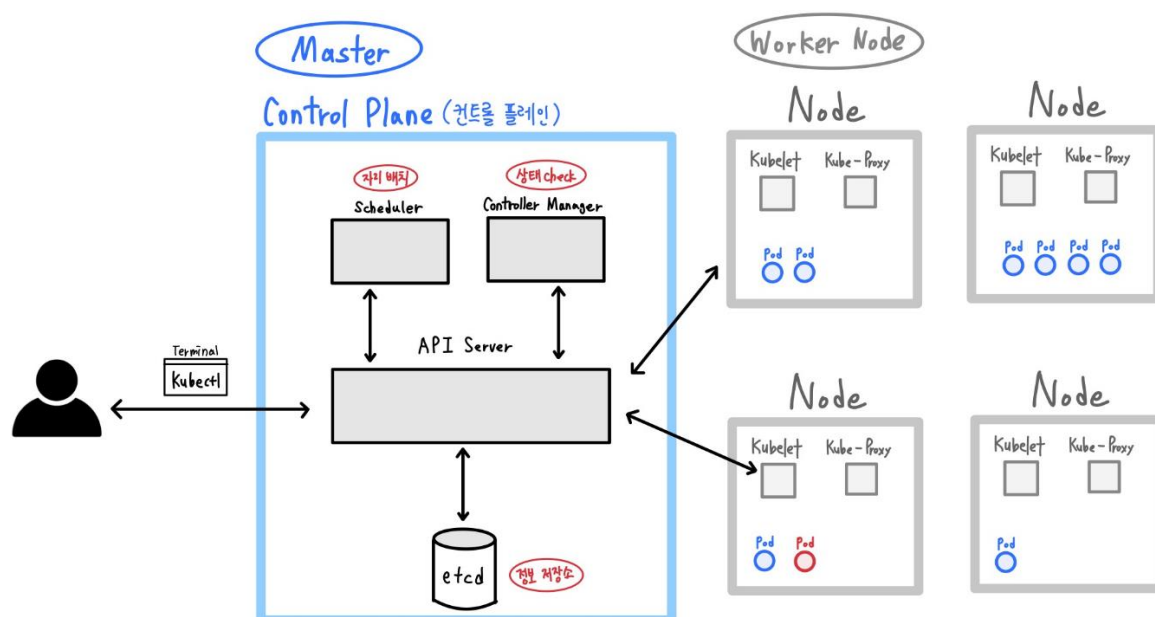
## ③ kube-proxy

쿠버네티스 클러스터 내부에서 네트워크 요청을 전달하는 역할

즉 파드의 IP 는 매번 변하지만 kube-proxy 가 이 파드에 접근할 수 있는 방법을 그때마다 관리하고 갱신하며, 서비스 오브젝트는 이 정보를 사용하여 파드가 외부에서 접근할 수 있는 경로를 제공함

<https://www.samsungsds.com/kr/insights/kubernetes-3.html>

## 4) 동작과정



- ① [사용자 → api server] 파드를 만들어달라고 요청
  - ② [api server → etcd] 요청한 상황을 저장
  - ③ [controller → api → etcd] current state 와 desired state 비교
  - ④ [controller → api] current state 와 desired state 가 동일하지 x 시 이 사실을 api 에게 알림
  - ⑤ [scheduler → api] 생성된 파드가 어떤 노드에 들어가면 좋을지 파악해서 api 에게 알림
  - ⑥ [api → kubelet] api 가 scheduler 가 지정해준 노드의 kubelet 에게 파드를 만들라고 지시함
  - ⑦ [kubelet] 파드를 만듦
- \* 모든 구성요소는 API Server 를 통해 유기적으로 상호작용함
- API Server 를 통하지 x 고서는 상호작용 불가함
- (ex\_Controller Manager – Scheduler 의 직접적 상호작용 불가)

## 2. Docker compose

### 1) 개념

단일 서버에서 여러개의 컨테이너를 하나의 서비스로 정의해 컨테이너의 묶음으로 관리할 수 있는 작업 환경을 제공하는 관리 도구

도커 컴포즈는 여러 개의 컨테이너의 옵션과 환경을 정의한 파일을 읽어 컨테이너를 순차적으로 생성하는 방식으로 동작함

컨테이너의 수가 많아지고 정의해야 할 옵션이 많아지고 정의해야 할 옵션이 많아진다면 도커 컴포즈를 사용하는 것이 좋음

### 2) yaml 파일 작성법

yaml 파일에서 들여쓰기를 할 때 탭(Tab)은 도커 컴포즈가 인식하지 못하므로 2 개의 공백(Space)을 사용해서 하위 항목을 구분해야 함

docker-compose.yml 파일을 읽어 로컬의 도커 엔진에게 컨테이너 생성을 요청합니다. docker compose up 명령어로 도커 컴포즈 프로젝트를 실행할 수 있음

## Ⅱ. kubeadm으로 클러스터 구성하기

### 1. 환경 구성

#### 1-1. D 드라이브에 Vagrantfile 생성하기

- D 드라이브에 'Test' 폴더 > 'T7' 폴더 생성하기
- 'T7' 폴더에 Vagrantfile 생성하기

- powershell에 관리자 권한으로 들어가기
- 'T7' 폴더에 Vagrantfile 생성하기

**D:** //D 드라이브에 접속하기

**cd /Test/T7** //해당 폴더에 접속하기

**vagrant init** //vagrantfile 생성하기

```
PS C:\> D:
PS D:\> cd /Test/T7
PS D:\Test\T7> vagrant init
A 'Vagrantfile' has been placed in this directory. You are now
ready to 'vagrant up' your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
'vagrantup.com' for more information on using Vagrant.
PS D:\Test\T7>
```

- 생성된 vagrantfile을 강사님이 제공해주신 스크립트로 변경 후 저장하기
- vagrantfile을 메모장으로 열어서 하기

#### 1-2. 작성된 vagrantfile을 바탕으로 프로비저닝 진행하기

**vagrant up**

**프로비저닝**

가상 머신에 소프트웨어를 설치하고 설정하는 프로세스

- 하지만 오류 났음

```
PS D:\Test\T7> vagrant up
PS D:\Test\T7> <internal:C:/Program Files/Vagrant/embedded/mingw64/lib/ruby/3.1.0/rubyg
ems/core_ext/kernel_require.rb>:85:in 'require': cannot load such file -- vagrant (Load
Error)
    from <internal:C:/Program Files/Vagrant/embedded/mingw64/lib/ruby/3.1.0/rubygem
s/core_ext/kernel_require.rb>:85:in 'require'
    from C:/Program Files/Vagrant/embedded/gems/gems/vagrant-2.4.1/bin/vagrant:111:
in '<main>'
```

⇒ vagrantfile의 띄워쓰기를 잘못해서 생기는 오류였음. 띄워쓰기 잘하기



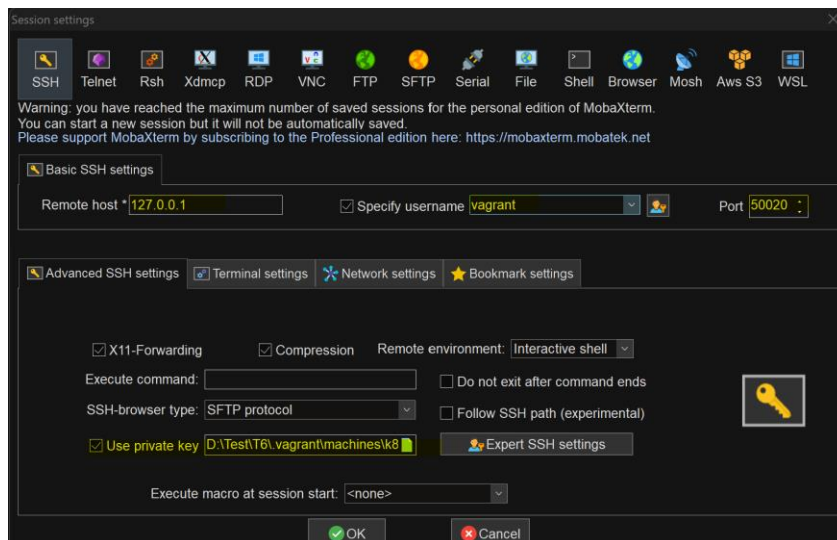
- **vagrant up** 다시 시도



## 2. 모든 서버에 공통으로 설정해야 할 부분

### 2-1. mobaxterm으로 접속하기

- private key를 입력해줘야 함



- 강사님이 주신 vagrantfile에 각 서버별로 포트번호 지정해줌. 참고해서 적기

### 2-2. 업데이트 및 재부팅(모든 서버)

```
sudo apt-get update && sudo apt-get -y full-upgrade
```

```
sudo reboot -f
```

### 2-3. 설치 전 필요한 패키지 설치(모든 서버)

```
sudo apt-get -y install curl gnupg2 software-properties-common W
```

```
apt-transport-https ca-certificates
```

// curl, gnupg2, software-properties-common, apt-transport-https, ca-certificates 패키지를 설치

// -y 옵션: 설치 과정에서 나타나는 모든 질문에 대해 "yes"로 대답하도록 하여 대화형 설치 프로세스를 건너뛰도록 함

## 2-4. 도커 리포지터리 등록(모든 서버)

Docker의 APT 저장소가 시스템에 추가되어 Docker 패키지를 APT 패키지 관리 시스템에서 설치할 수 있게 됨

```
sudo install -m 0755 -d /etc/apt/keyrings
```

// '/etc/apt/keyrings' 디렉토리를 만들고, 해당 디렉토리에 적절한 권한을 부여함. 이 디렉토리는 APT 패키지 관리 시스템에서 사용되는 키링 파일(보안 목적으로 사용되는 파일로, 주로 암호화나 디지털 서명과 관련된 작업에서 사용)을 저장하는 데 사용됩니다.

```
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o  
/etc/apt/keyrings/docker.asc
```

// Docker 공식 GPG 키를 다운로드하여 /etc/apt/keyrings/docker.asc 파일에 저장함. 이 키는 Docker 패키지를 APT 패키지 관리 시스템에서 안전하게 설치하기 위해 사용됨

```
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

// 다운로드된 Docker GPG 키 파일에 모든 사용자에게 대한 읽기 권한을 부여함. 이렇게 함으로써 해당 키 파일이 시스템에서 올바르게 사용될 수 있음

```
echo ₩
```

```
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]  
https://download.docker.com/linux/ubuntu ₩
```

```
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | ₩
```

```
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

// echo "...": 따옴표 내부에 있는 텍스트를 출력함. 이 텍스트는 Docker의 APT 저장소를 추가하는 데 필요한 정보를 포함함

// dev ~ stable: Docker의 APT 저장소를 정의함. 이 저장소는 특정 아키텍처에 대한 패키지를 지원하며, GPG 서명이 /etc/apt/keyrings/docker.asc 파일에 의해 확인됨. 저장소의 주소는 Docker의 공식 다운로드 서버에서 제공됨. 또한 Ubuntu의 버전에 따라 저장소의 구성이 달라짐.

// sudo ~ null: tee 명령을 사용하여 출력을 파일에 씴. 이를 통해 Docker의 APT 저장소 설정을 /etc/apt/sources.list.d/docker.list 파일에 저장함. > /dev/null은 출력을 표준 출력에서 제거하여 화면에 출력되지 않도록 함

## 2-5. 쿠버네티스 리포지터리 등록(모든 서버)

```
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.27/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
```

// -f : 오류가 발생한 경우 오류 메시지를 출력하지 않고 실패하기

// -s : 정적으로(무음으로) 출력

// -S : 오류 발생 시 에러 메시지를 표시, **-s** 옵션과 함께 사용할 경우에만 사용함

// -L : 리다이렉션을 따름. 이 옵션을 사용하면 curl이 HTTP 리다이렉션을 따르도록 함. 즉, 요청이 리다이렉션되는 경우에도 리다이렉션된 페이지를 요청함

// https://pkgs.k8s.io/core:/stable:/v1.27/deb/Release.key URL에서 Kubernetes의 공개 GPG 키를 다운로드하고, gpg --dearmor 명령을 사용하여 이 키를 디코딩(GPG 키를 이해 가능한 형식으로 변환하여 저장)하여 /etc/apt/keyrings/kubernetes-apt-keyring.gpg 파일에 저장함. 이 키는 Kubernetes 패키지를 APT 패키지 관리 시스템에서 안전하게 설치하기 위해 사용됨.

```
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.27/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

// 이 명령은 Kubernetes의 APT 저장소를 정의하여 APT 패키지 관리 시스템에 추가함. 저장소의 주소는 https://pkgs.k8s.io/core:/stable:/v1.27/deb/이며, GPG 서명이 /etc/apt/keyrings/kubernetes-apt-keyring.gpg 파일에 의해 확인됨. 이 저장소는 Kubernetes 패키지를 APT 패키지 관리 시스템에서 가져올 수 있도록 함

## 2-6. 리포지토리 업데이트 진행(모든 서버)

```
sudo apt update -y
```

## 2-7. 컨테이너 런타임 설치(모든 서버)

컨테이너 런타임(container runtime): 컨테이너화된 응용 프로그램을 실행하는 소프트웨어. 컨테이너는 격리된 환경에서 실행되는 가상화 된 프로세스로, 컨테이너 런타임은 이러한 컨테이너를 생성하고 관리함. 가장 널리 사용되는 컨테이너 런타임 중 하나는 Docker의 Containerd임

```
sudo apt install -y containerd.io
```

// containerd.io 패키지 : Containerd 런타임을 제공함

```
sudo mkdir -p /etc/containerd
```

// /etc/containerd 디렉토리: Containerd의 설정 파일인 config.toml을 저장하는 데 사용될 것임

```
sudo containerd config default | sudo tee /etc/containerd/config.toml > /dev/null
```

// containerd config default 명령을 실행하여 Containerd의 기본 설정을 생성함

// 파이프를 통해 sudo tee를 사용하여 /etc/containerd/config.toml 파일에 쓰여짐

// tee 명령의 출력은 /dev/null로 리디렉션되어 표시되지 않도록 함

## 2-8. 쿠버네티스 설치(모든 서버)

```
sudo apt -y install kubelet kubeadm kubectl
```

// kubelet, kubeadm, kubectl 패키지를 설치

// kubelet : Kubernetes 노드에서 실행되는 시스템 서비스로, 클러스터에 노드를 추가하고 컨테이너를 관리함

// kubeadm : Kubernetes 클러스터를 시작하는 데 사용되는 도구로, 클러스터를 초기화하고 노드를 추가하는 데 도움을 줌

// kubectl : Kubernetes 클러스터와 상호 작용하기 위한 커맨드 라인 인터페이스(CLI) 도구

- k8s\_w2에서만 **오류** 생성됨 (아직 해결하지 못함)

```
vagrant@k8s-w2:~$ sudo apt -y install kubelet kubeadm kubectl
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
Waiting for cache lock: Could not get lock /var/lib/dpkg/lock-frontend. It is held
^Zy process 2001 (apt)... 9s
```

⇒ 이는 **apt** 패키지 관리 시스템이 현재 다른 프로세스에 의해 사용되고 있어서 **lock** 파일을 얻을 수 없을 때 발생하는 것임

```
sudo apt-mark hold kubelet kubeadm kubectl
```

// kubelet, kubeadm, kubectl 패키지의 업그레이드를 방지함. 이렇게 함으로써 예기치 않은 업그레이드로 인한 시스템의 안정성을 유지할 수 있음

## 2-9. 스왑 비활성화(모든 서버)

스왑 : 컴퓨터의 물리적인 RAM(Random Access Memory) 메모리를 보완하는 가상 메모리 공간

- 메모리가 부족한 경우에 사용되며, RAM이 부족한 상황에서 프로그램이 추가 메모리를 요청할 때 스왑 공간이 사용됨

```
sudo sed -i '/swap/s/^/#/' /etc/fstab
```

// '/etc/fstab' 파일(시스템 부팅 시 마운트되는 파일 시스템을 정의하는 파일)에서 스왑 관련 항목을 주석 처리

// 스왑 관련 항목을 주석 처리함으로써 시스템이 부팅될 때 스왑이 자동으로 마운트되지 않도록 설정함.

```
sudo swapoff -a && sudo mount -a
```

// 스왑을 비활성화하고 변경된 /etc/fstab 파일을 다시 로드하여 스왑을 다시 마운트

## 2-10. 커널 기능 설정 변경(모든 서버)

```
sudo su - -c "echo 'net.bridge.bridge-nf-call-ip6tables = 1' >> /etc/sysctl.d/kubernetes.conf"
```

```
>> /etc/sysctl.d/kubernetes.conf"
```

// echo를 사용하여 특정 텍스트(net.bridge.bridge-nf-call-ip6tables = 1)를 kubernetes.conf라는 파일에 추가하는 것

// >> 연산자를 사용하여 파일에 내용을 추가함. 만약 kubernetes.conf 파일이 존재하지 않는 경우에는 새로운 파일이 생성됨

// 'net.bridge.bridge-nf-call-ip6tables = 1' : IPv6 패킷에 대한 iptables 처리를 활성화

```
sudo su - -c "echo 'net.bridge.bridge-nf-call-iptables = 1' >> /etc/sysctl.d/kubernetes.conf"
```

```
>> /etc/sysctl.d/kubernetes.conf"
```

// 커널이 iptables를 통해 브리지 트래픽을 처리하도록 하는 것

// net.bridge.bridge-nf-call-iptables = 1 : 커널에게 브리지에 도착하는 패킷을 iptables로 라우팅하도록 지시하는 것

```
sudo su - -c "echo 'net.ipv4.ip_forward = 1' >> /etc/sysctl.d/kubernetes.conf"
```

```
>> /etc/sysctl.d/kubernetes.conf"
```

// IPv4 패킷을 전달할 때 패킷을 다른 인터페이스로 전달할 수 있도록 허용하는 것으로, Kubernetes에서 노드 간 통신을 가능하게 함

## 2-11. 커널 모듈 로드 설정 (모든 서버)

```
sudo su - -c "echo 'overlay' >> /etc/modules-load.d/containerd.conf"
```

// '/etc/modules-load.d/containerd.conf' 파일에 overlay 모듈 추가

// Overlay 파일 시스템: 컨테이너의 파일 시스템을 관리하는 데 사용됨. 컨테이너의 루트 파일 시스템을 레이어로 구성하여 효율적인 디스크 공간 사용 및 빠른 파일 시스템 성능을 제공함

```
sudo su - -c "echo 'br_netfilter' >> /etc/modules-load.d/containerd.conf"
```

// '/etc/modules-load.d/containerd.conf' 파일에 br\_netfilter 모듈 추가

//br\_netfilter: 브리지 네트워킹과 관련된 네트워크 필터링 및 전달 기능을 제공합니다. 이 모듈은 컨테이너의 네트워크 트래픽을 관리하는 데 사용

## 2-11. 커널 모듈 및 설정 활성화 (모든 서버)

```
sudo modprobe overlay // overlay 모듈을 로드하기
```

```
sudo modprobe br_netfilter // br_netfilter 모듈을 로드
```

```
sudo sysctl --system // 시스템 설정을 적용
```

## 2-12. 서비스 활성화 (모든 서버)

```
sudo systemctl restart containerd kubelet // containerd와 kubelet 서비스를 다시 시작
```

```
sudo systemctl enable containerd kubelet // containerd와 kubelet 서비스를 부팅 시 자동으로 시작하도록 설정
```

## 2-13. 모든 서버의 /etc/hosts 파일 수정 (모든 서버)

모든 서버에 컨트롤러 및 워커 노드의 IP 주소와 hostname 정보를 등록

```
sudo echo "192.168.56.200 k8s-m" >> /etc/hosts
```

```
sudo echo "192.168.56.210 k8s-w1" >> /etc/hosts
```

```
sudo echo "192.168.56.220 k8s-w2" >> /etc/hosts
```

- 하지만 오류 뜸 ⇒ sudo -i로 관리자 권한모드에서 다시 실행하기

```
vagrant@k8s-w1:~$ sudo systemctl enable containerd kubelet
vagrant@k8s-w1:~$ sudo echo "192.168.56.200 k8s-m" >> /etc/hosts
-bash: /etc/hosts: Permission denied
vagrant@k8s-w1:~$ sudo echo "192.168.56.210 k8s-w1" >> /etc/hosts
-bash: /etc/hosts: Permission denied
vagrant@k8s-w1:~$ sudo echo "192.168.56.220 k8s-w2" >> /etc/hosts
-bash: /etc/hosts: Permission denied
```

```
vagrant@k8s-m:~$ sudo -i
root@k8s-m:~# sudo echo "192.168.56.200 k8s-m" >> /etc/hosts
root@k8s-m:~# sudo echo "192.168.56.210 k8s-w1" >> /etc/hosts
root@k8s-m:~# sudo echo "192.168.56.220 k8s-w2" >> /etc/hosts
root@k8s-m:~# sudo echo "192.168.56.200 k8s-m" >> /etc/hosts
root@k8s-m:~# sudo echo "192.168.56.210 k8s-w1" >> /etc/hosts
root@k8s-m:~# sudo echo "192.168.56.220 k8s-w2" >> /etc/hosts
```

## 2-14. 호스트 네임 변경 (마스터 / 워커1 / 워커2 노드에 각각 설정)

master에서 ⇒ `hostnamectl set-hostname k8s-m`

worker1에서 ⇒ `hostnamectl set-hostname k8s-w1`

worker2에서 ⇒ `hostnamectl set-hostname k8s-w2`

## 3. 마스터에 설정해야 할 부분

### 3-1. 컨트롤러 노드에 쿠버네티스 관련 서비스 이미지 설치 (마스터 노드에만 설치)

```
sudo kubeadm config images pull //필요한 이미지 다운
```

### 3-2. kubeadm으로 부트스트랩 진행

부트스트랩: 클러스터를 시작하고 초기화하는 과정

```
sudo kubeadm init --apiserver-advertise-address 192.168.56.200 ₩
```

```
--pod-network-cidr 172.30.0.0/16 ₩
```

```
--upload-certs
```

// `--apiserver-advertise-address 192.168.56.200` : Kubernetes API 서버가 사용할 IP 주소를 지정함.  
클러스터 내부 및 외부에서 API 서버에 접근할 수 있는 주소로 설정해야 함.

`--pod-network-cidr 172.30.0.0/16`: 파드 네트워크의 CIDR 범위를 지정함. 파드 네트워크는 클러스터 내부에서 파드 간 통신을 가능하게 하는데 사용됨

--upload-certs: 인증서를 업로드함. 이 옵션은 TLS 인증서를 Kubernetes API 서버에 업로드하여 보안 통신을 활성화합니다.

- 아래처럼 키가 생성 됨

```
kubeadm join 192.168.56.200:6443 --token rstzsf.ucu70xesq8uf705p \
--discovery-token-ca-cert-hash sha256:57fc265f32dc4f1cc920a3581ebd061371d2
```

### 3-3. 부트스트랩 완료 후 현재 계정으로 kubectl 명령을 사용할 수 있도록 설정

```
mkdir -p $HOME/.kube
```

// 현재 사용자의 홈 디렉토리에 .kube 디렉토리를 생성함. 이 디렉토리는 Kubernetes 클러스터에 대한 구성 파일인 config 파일을 저장하는 곳

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

// Kubernetes 클러스터의 구성 파일을 복사하여 사용자의 홈 디렉토리의 .kube 디렉토리에 있는 config 파일로 복사

// 이는 클러스터에 대한 접근을 위한 인증 및 구성 정보가 담긴 파일임

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

// config 파일의 소유자를 현재 사용자로 변경



### 3-4. 2개의 워커노드(1,2)에 이전에 생성된 키 넣어주기

```
root@k8s-w1:~# kubeadm join 192.168.56.200:6443 --token rstzsf.ucu70xesq8uf705p \
--discovery-token-ca-cert-hash sha256:57fc265f32dc4f1cc920a3581ebd061371d29ab9058461615c7664da17b96b00
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
```

- 마스터노드에서 아래처럼 명령어 치고 결과 보기

**kubectl get node -A** // Kubernetes 클러스터의 모든 노드에 대한 정보를 조회

```
root@k8s-m:~# kubectl get node -A
NAME      STATUS    ROLES    AGE   VERSION
k8s-m     NotReady  control-plane  6m25s  v1.27.11
k8s-w1    NotReady  <none>       100s   v1.27.11
k8s-w2    NotReady  <none>       22s    v1.27.11
root@k8s-m:~#
```

### 3-5. 쿠버네티스의 컨테이너 네트워크 인터페이스 설치 및 설치 파일 다운로드 (CNI)

**kubectl create -f**

**<https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/tigera-operator.yaml>**

// kubectl create -f: Kubernetes 클러스터에 새로운 리소스를 생성하기 위한 명령어

// <https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/tigera-operator.yaml>: Calico의 Tigera 연산자를 정의한 YAML 파일의 URL임. 이를 통해 해당 YAML 파일을 가져와서 Kubernetes 클러스터에 배포함

**wget <https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/custom-resources.yaml>**

```
root@k8s-m:~# wget https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/custom-resources.yaml
--2024-02-28 00:53:14-- https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/custom-resources.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 824 [text/plain]
Saving to: 'custom-resources.yaml'

custom-resources.ya 100%[=====>]      824  --.-KB/s   in 0s
2024-02-28 00:53:14 (29.0 MB/s) - 'custom-resources.yaml' saved [824/824]
```

// wget: 웹에서 파일을 다운로드하기 위한 명령어

### 3-6. 네트워크 설정 변경 후 설치

```
sed -i 's/cidr: 192.168.0.0/16/cidr: 172.30.0.0/16/g' custom-resources.yaml
```

// custom-resources.yaml 파일 내에서 cidr: 192.168.0.0/16을 찾아서 cidr: 172.30.0.0/16으로 바꿈

// sed를 사용하여 인라인으로 파일을 수정하며, 정규 표현식을 사용하여 대상 문자열을 검색하고 대체

```
sed -i '7a registry: quay.io/' custom-resources.yaml
```

// custom-resources.yaml 파일의 7번째 줄 바로 다음에 registry: quay.io/를 추가합니다.

이 명령어는 sed를 사용하여 파일을 수정하며, a 옵션을 사용하여 주어진 텍스트를 추가합니다.

```
kubectl create -f custom-resources.yaml
```

//

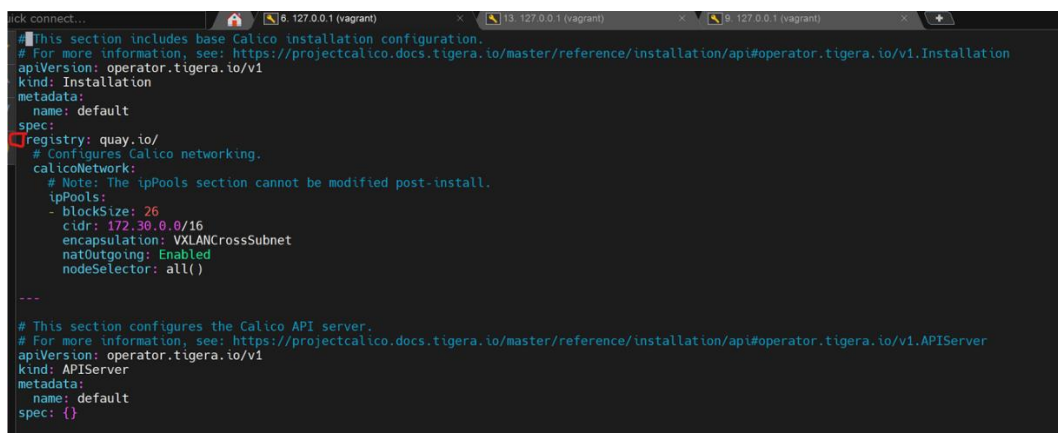
- 하지만 오류 뜸

```
root@k8s-m:~# kubectl create -f custom-resources.yaml
error: error parsing custom-resources.yaml: error converting YAML to JSON: yaml:
line 10: did not find expected key
root@k8s-m:~#
```

⇒ custom-resources.yaml 파일의 구문이 올바르지 않아 발생한 것

- yaml 파일 수정하기 (**vim custom-resources.yaml**)

- 두 칸 띄어쓰기



- **kubectl create -f custom-resources.yaml** 다시 시도하기 ⇒ 성공

```
root@k8s-m:~# kubectl create -f custom-resources.yaml
installation.operator.tigera.io/default created
apiserver.operator.tigera.io/default created
root@k8s-m:~#
```

```
kubectl create -f custom-resources.yaml
```

```
kubectl get nodes
```

```
kubectl get pod -A
```

```
root@k8s-m:~# kubectl get nodes
NAME      STATUS   ROLES    AGE   VERSION
k8s-m     Ready    control-plane  22m   v1.27.11
k8s-w1    Ready    <none>     17m   v1.27.11
k8s-w2    Ready    <none>     16m   v1.27.11
root@k8s-m:~# kubectl get pod -A
NAMESPACE   NAME                                                    READY   STATUS    RESTARTS   AGE
calico-system   calico-apiserver-f8c5d8455-7ld98                    1/1     Running   0           38s
calico-system   calico-apiserver-f8c5d8455-trvj2                    1/1     Running   0           38s
calico-system   calico-kube-controllers-5fbc6b5888-flgvm             1/1     Running   0           4m12s
calico-system   calico-node-49cls                                    1/1     Running   0           4m13s
calico-system   calico-node-4c4w5                                    1/1     Running   0           4m13s
calico-system   calico-node-hcv7r                                    1/1     Running   0           4m13s
calico-system   calico-typha-77bd7d686c-rbtvs                       1/1     Running   0           4m5s
calico-system   calico-typha-77bd7d686c-rrqgj                      1/1     Running   0           4m13s
calico-system   csi-node-driver-48td6                               2/2     Running   0           4m12s
calico-system   csi-node-driver-ck9xg                               2/2     Running   0           4m12s
calico-system   csi-node-driver-h55mq                               2/2     Running   0           4m12s
kube-system     coredns-5d78c9869d-8m66c                            1/1     Running   0           22m
kube-system     coredns-5d78c9869d-lz9ck                            1/1     Running   0           22m
kube-system     etcd-k8s-m                                            1/1     Running   0           22m
kube-system     kube-apiserver-k8s-m                                1/1     Running   0           22m
kube-system     kube-controller-manager-k8s-m                      1/1     Running   1 (20m ago)  22m
kube-system     kube-proxy-22dl7                                     1/1     Running   0           17m
kube-system     kube-proxy-6z5tq                                    1/1     Running   0           22m
kube-system     kube-proxy-lv89h                                    1/1     Running   0           16m
kube-system     kube-scheduler-k8s-m                                1/1     Running   0           22m
tigera-operator tigera-operator-5f4668786-fv98g                     1/1     Running   0           10m
root@k8s-m:~#
```

김서희님, 나지원님 실습가이드