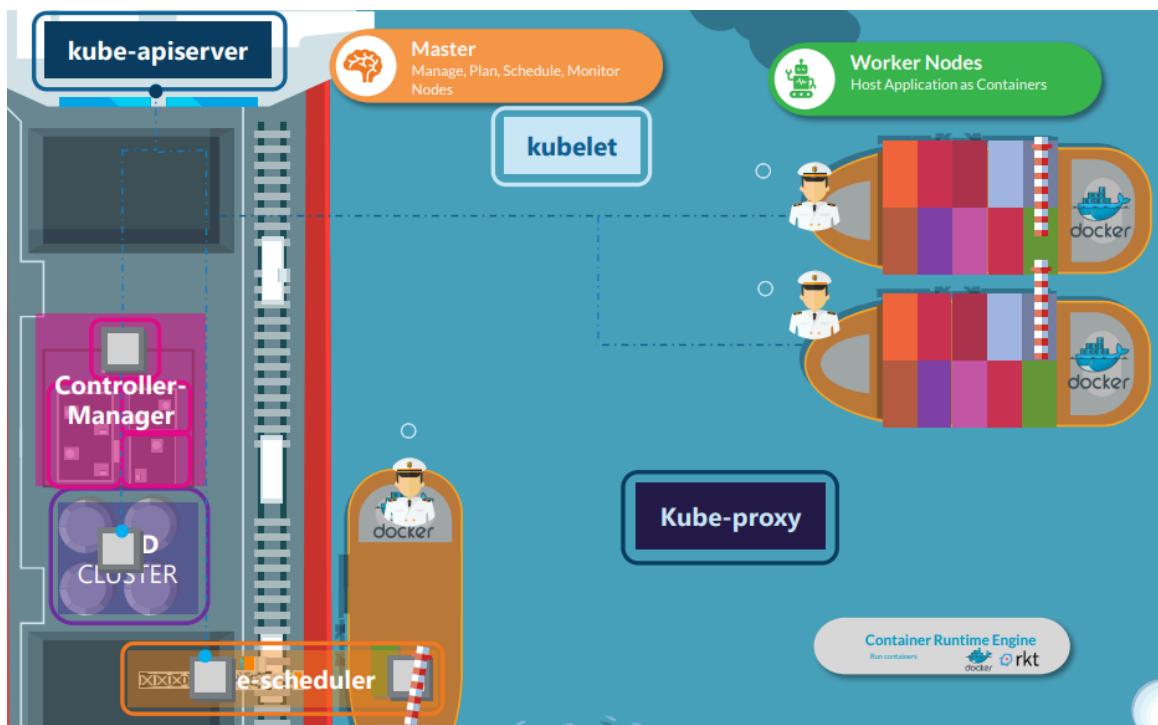
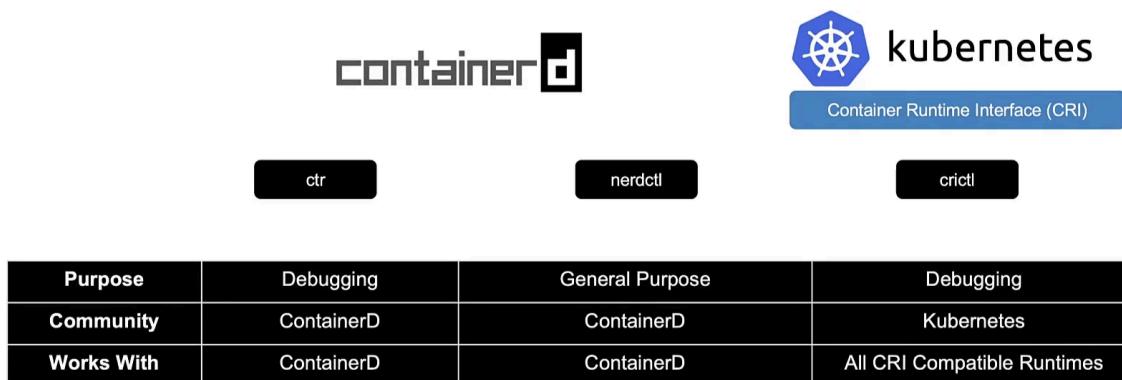


Section 2: Core Concepts

▼ 10. Cluster Architecture



▼ 11. Docker-vs-ContainerD



- cri = container runtime interface
 - 1. `ctr`
 - `ctr` 이 붙은 명령어는 컨테이너 관련 작업 수행하는데 사용

- ex1) `ctr images pull docker.io/library/redis:alpine` : redis 이미지 끌어오기
- ex2) `ctr run dockr.io/library/redis:alpine redis` : 컨테이너 실행하기
- 하지만 `ctr` 명령어 유ти리티는 오직 **ContainerD**를 디버깅하기 위해서 만들어졌고, 사용자 친화적이지도 않음. **기능이 매우 제한적임**. 따라서 제품 환경에서 컨테이너를 실행하거나 관리하는 용도로 사용되지 않음. 그래서 무시할 수 있음
- **ContainerD community**에서 온거

2. 그래서 나온 대안이 `nerdctl`

- **ContainerD를 위한 CLI 같은 것이다. docker와 같은 느낌**
- 나머지 둘과 다르게 디버깅이 아닌, 컨테이너를 생성하는 것과 같은 **일반적인 목적으로 사용됨**
- docker가 지원하는 대부분의 옵션을 지원하며 **그 이상의 기능을 지원**. docker와 명령어가 거의 유사함!
- 컨테이너에 구현된 최신 기능에 접근할 수 있음

```
$ docker → $ nerdctl
$ docker run --name redis redis:alpine → $ nerdctl run --name redis redis:alpine
$ docker run --name webserver -p 80:80 -d nginx → $ nerdctl run --name webserver -p 80:80 -d nginx
```

- 얘도 **ContainerD community**에서 온거
- 앞으로 많이 사용될것임!

3. `criclt`

- 디버깅 툴이며, **쿠버네티스 community**에서 온 것임
- 주로 cli 호환 가능한 모든 런타임과 상호작용하는 데 이용
- 컨테이너뿐만아니라 cri를 지원하는 모든 런타임 전체에 사용될 수 있음
- 이걸로 컨테이너를 만들 순 있지만 쉽진 않다. 온리 디버깅 목적으로 사용
- 큐블렛과 잘 어울린다는 것을 기억하기!
 - 큐블렛이 한번에 특정 컨테이너나 포드를 특정 개수씩 노드에서 사용할 수 있게 보장한다.

▼ 12. ETCD For Beginners

etcd

- `etcd` : 분산되고 신뢰할 수 있으며 단순, 안전, 신속한 key-value store
- etcd 설치하기



- etcd 를 run하면 포트 2379의 신호를 자동으로 듣는 서비스가 시작된다. 그러면 클라이언트를 붙여 정보를 저장하고 검색할 수 있다.
- etcd 와 함께 오는 기본 클라이언트는 `etcdctl` (엣시디 컨트롤 클라이언트)이다.
 - etcdctl은 etcd의 command-line client이다. key-value 쌍을 저장하고 검색(retrieve)할 수 있음.
 - `./etcdctl set key1 value` : 저장
 - `./etcdctl get key1` : 검색
 - 기타 많은 옵션을 보려면 `./etcdctl` 명령어를 통해서 가이드를 볼 수 있음
- etcd의 버전에 따른 변화
 - v2.0이 많이 쓰였지만 v2.0에서 v3 사이에 command 등 많은 변화가 있었다.
 - etcd 명령어는 v2와 v3 를 동시에 지원한다(?)
 - etcdctl이 어떤 버전인지 확인하는법: `etcdctl --version`
 - 아래 캡쳐에서 첫째줄은 etcdctl 그 자체의 버전이고, 두번째는 api 버전이다. 아래와 같이 etcdctl version이 3인데 api version이 2인 상

태라면 v2로 작동하도록 설정된 것이다. 그래서 `./etcdctl` 명령어를 다른 옵션 없이 실행하면 나오는 안내들은 version 2에 관한 것이다.

```
▶ ./etcdctl --version
etcdctl version: 3.3.11
API version: 2
```

- api version을 3으로 설정하고 싶다면, 명령 실행 직전에 아래와 같이 환경변수를 각 명령어에 설정한다.

```
▶ ETCCTL_API=3 ./etcdctl version
etcdctl version: 3.3.11
API version: 3.3
```

- 혹은 전체 세션에 대한 환경변수로 export 할 수도 있다. 이렇게 하면 명령 실행할때마다 위처럼 해줄 필요 없음

```
▶ export ETCCTL_API=3
./etcdctl version
etcdctl version: 3.3.11
API version: 3.3
```

- v3는 v2와 다르게 저장할 때 set 대신 put 사용함.. 이런식으로 많이 다르다!

ETCDCTL v3

```
▶ export ETCDCTL_API=3  
./etcdctl version
```

```
etcdctl version: 3.3.11  
API version: 3.3
```

```
▶ ./etcdctl put key1 value1
```

```
OK
```

```
▶ ./etcdctl get key1
```

```
key1  
value1
```

▼ 13. ETCD in Kubernetes

- 쿠버네티스 배포에는 두가지 유형이 있다.
 - kubeadm (큐베디움) tool을 이용한 배포
 - 연습 테스트 환경은 큐베디움 이용함. 나중에 클러스터를 setup하는 강의에서는 처음부터(from scratch) 배포한다. 그래서 두 방법의 차이를 아는 게 중요!

▼ 처음부터 다 배포

- etcd 등의 binary를 직접 다운로드해서 배포. 마스터 노드에 직접 바이너리 설치하고, 서비스로서 구성하는 것.
- 서비스에는 많은 선택지가 있는데, 그 중 상당수가 인증서와 관련있음. 나중에 이 강의에서는 인증서를 어떻게 생성하고 구성하는지 등 자세히 배울것임!

Setup - Manual

```
▶ wget -q --https-only \
  "https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"

[etcd.service]
ExecStart=/usr/local/bin/etcd \\
--name ${ETCD_NAME} \\
--cert-file=/etc/etcd/kubernetes.pem \\
--key-file=/etc/etcd/kubernetes-key.pem \\
--peer-cert-file=/etc/etcd/kubernetes.pem \\
--peer-key-file=/etc/etcd/kubernetes-key.pem \\
--trusted-ca-file=/etc/etcd/ca.pem \\
--peer-trusted-ca-file=/etc/etcd/ca.pem \\
--peer-client-cert-auth \\
--client-cert-auth \\
--initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
--listen-peer-urls https://${INTERNAL_IP}:2380 \\
--listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
--advertise-client-urls https://${INTERNAL_IP}:2379 \\
--initial-cluster-token etcd-cluster-0 \\
--initial-cluster controller-0=https://${CONTROLLER0_IP}:2380,controller-1=https://${CONTROLLER1_IP}:2380 \\
--initial-cluster-state new \\
--data-dir=/var/lib/etcd
```

▼ 15. Kube-API Server

- **kubectl** : 쿠버네티스 클러스터에 명령을 내리는 역할. api server와 주로 통신
- **kube-apiserver** : 통신 담당, 클러스터 내의 모든 작업을 오케스트레이션 함. 주로 상태값이 모두 저장되는 etcd와 주로 통신함. 또한 worker node가 server와 통신하게 함. 쿠버네티스 클러스터에서 변경을 위해 수행해야 하는 모든 작업의 중심에 있는 통로이다.
 - etcd data store과 직접 상호작용하는 유일한 구성요소이다. 스케줄러, kube-controller-manager, kubelet 같은 다른 구성 요소는 API 서버를 이용해 각 영역의 클러스터에서 업데이트를 수행한다.
 - kube-apiserver는 아래 초록색 부분과 같이 많은 매개변수로 실행된다.

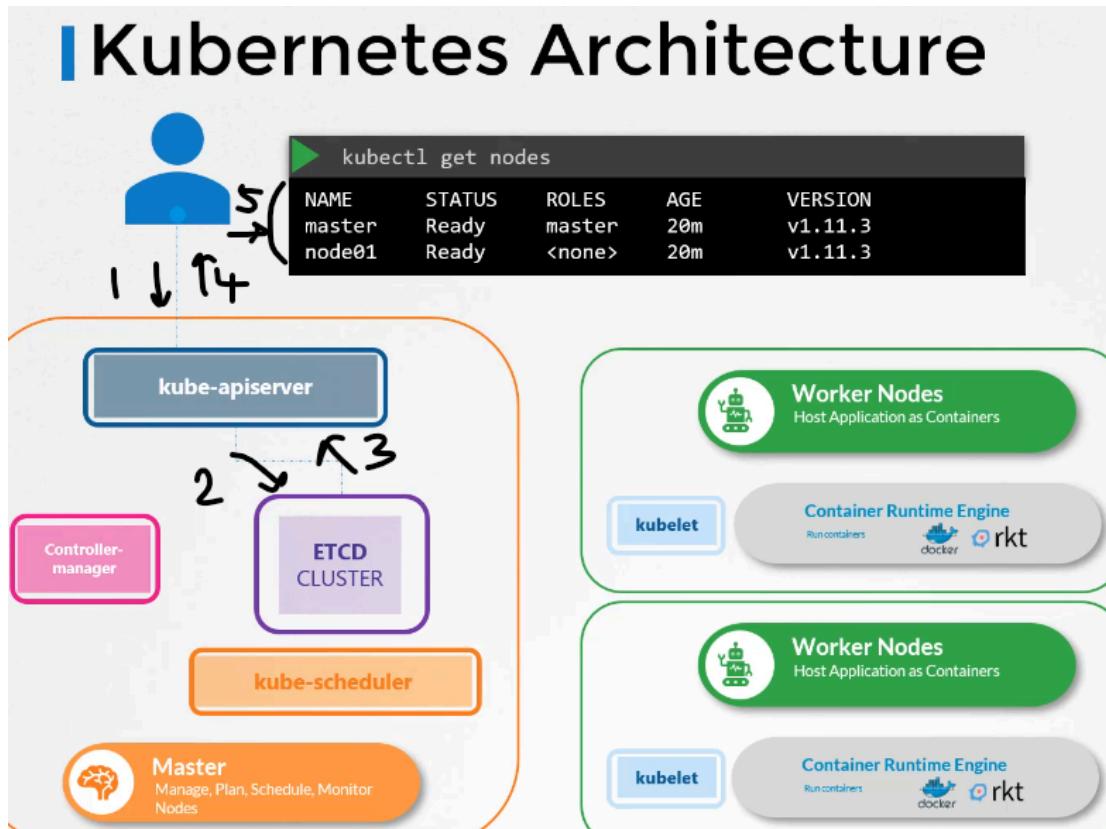
I Installing kube-api server

```
▶ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-apiserver

[kube-apiserver.service]
ExecStart=/usr/local/bin/kube-apiserver \\
--advertise-address=${INTERNAL_IP} \\
--allow-privileged=true \\
--apiserver-count=3 \\
--authorization-mode=Node,RBAC \\
--bind-address=0.0.0.0 \\
--client-ca-file=/var/lib/kubernetes/ca.pem \\
--enable-admission-plugins=Initializers,NamespaceLifecycle,NodeRestriction,LimitRanger,ServiceAccount,DefaultStorageClass,ResourceQuota \\
--enable-swagger-ui=true \\
--etcd-cafile=/var/lib/kubernetes/ca.pem \\
--etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
--etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
--etcd-servers=https://127.0.0.1:2379 \\
--event-ttl=1h \\
--experimental-encryption-provider-config=/var/lib/kubernetes/encryption-config.yaml \\
--kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
--kubelet-client-certificate=/var/lib/kubernetes/kubernetes.pem \\
--kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\
--kubelet-https=true \\
--runtime-config=api/all \\
--service-account-key-file=/var/lib/kubernetes/service-account.pem
```

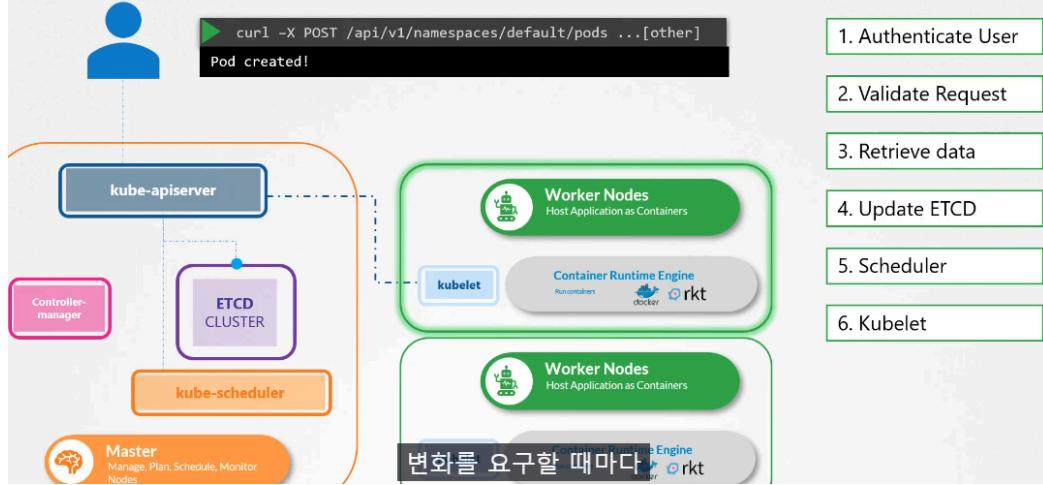
여기에서 볼 수 있듯이 말이죠

- `kubectl` 명령을 실행하면 일어나는 일



1. `kubectl` 유틸리티가 실제로 `kube-apiserver`에 도달한다. `kubectl get nodes`
 2. ~5. `kube-apiserver`는 먼저 요청을 인증하고 검증, `etcd` 클러스터에서 데이터를 검색하고 요청된 정보로 응답
- `kubectl` 명령어를 사용하는 대신에 이렇게 POST 요청을 보내서 API를 직접 호출할 수도 있다.

Kubernetes Architecture



pod를 생성하는 예시

- 이 경우 API 서버는 노드에 할당하지 않고 포드 자체를 생성함
- etcd 서버에 있는 정보를 업데이트하고 '포드가 생성된' 사용자를 업데이트 함
- kube-scheduler는 지속적으로 API 서버를 모니터하기 때문에, 노드가 할당되지 않은 새로운 포드가 있다는 걸 인지하게 됨
- scheduler가 올바른 노드를 식별해 새 pod를 켜고 다시 kube-apiserver 와 통신을 한다.
- API 서버는 etcd cluster 정보를 업데이트한다.
- API 서버는 해당 정보를 적절한 Worker Node에서 kubelet에게 전달한다.
- 그럼 kubelet은 노드 위에 포드를 생성하고 컨테이너 런타임 엔진에 지시해 앱 이미지를 배포한다.
- 완료되면 kubelete은 state를 API 서버로 다시 업데이트한다.
- API 서버는 기타 등의 클러스터에서 데이터를 업데이트한다.
- 변화를 요구할 때마다 비슷한 패턴이 반복됨

▼ 16. Kube Controller Manager

- **kube-controller-manager:** 시스템 내 다양한 구성 요소의 상태를 지속적으로 모니터링하고, 시스템 전체를 원하는 상태로 만든다. 다양한 상태 값을 관리하는 주체들이 컨트롤러 매니저에 소속돼 각자의 역할을 수행한다.

- **Node Controller** : node의 상태를 모니터링하고, application이 계속 실행되도록 하기 위해 kube-api server를 통해서 필요한 action을 취한다.
 - 노드 관리함, 새 노드를 클러스터에 온보딩함, 사용할 수 없는 상황 처리. 예를 들어, 워커 노드에서 통신이 되지 않는 경우, 상태 체크와 복구는 쿠버네티스 클러스터에 속한 노드 컨트롤러에서 이루어진다.
- **Replication Controller** : replica set의 상태를 모니터링하고, 레플리카셋에 요청받은 파드 개수대로 파드를 생성한다.
 - ex) pod가 죽으면 다른 pod를 만든다.
- **end-point Controller**: 서비스와 파드를 연결하는 역할

▼ 17. Kube Scheduler

파드를 어느 노드에 배치할지 결정하는 프로세스다. 결정만 한다는 점에 주의할 것. 실제로 해당 노드에 파드를 배치하는 작업은 아래에서 다룬 **kubelet**이 수행한다.

스케줄러가 파드를 할당할 때 노드에 우선순위를 부여하는 단계는 다음과 같다. 물론 여기에 적용되는 기준은 필요에 따라 우회하거나 변경할 수 있다.

1. 파드가 요구하는 컴퓨팅 자원(CPU, 메모리 등) 기준으로 필터링
2. 파드가 배치된 이후 해당 노드에 남게 될 잔여 컴퓨팅 자원의 양을 기준으로 우선순위 책정

스케줄러는 기본적으로 `kube-system` 네임스페이스에 `kube-controller-scheduler-master` 파드로 존재한다. 이 파드의 정의 파일(yaml) 위치는 클러스터 구축 방법에 따라 다르다.

- `kubeadm` 으로 구축했다면 `/etc/kubernetes/manifests/kube-scheduler.yaml` 에 존재한다.
- 그 외의 방법으로 구축했다면 `/etc/systemd/system/kube-scheduler.service` 에 존재한다.

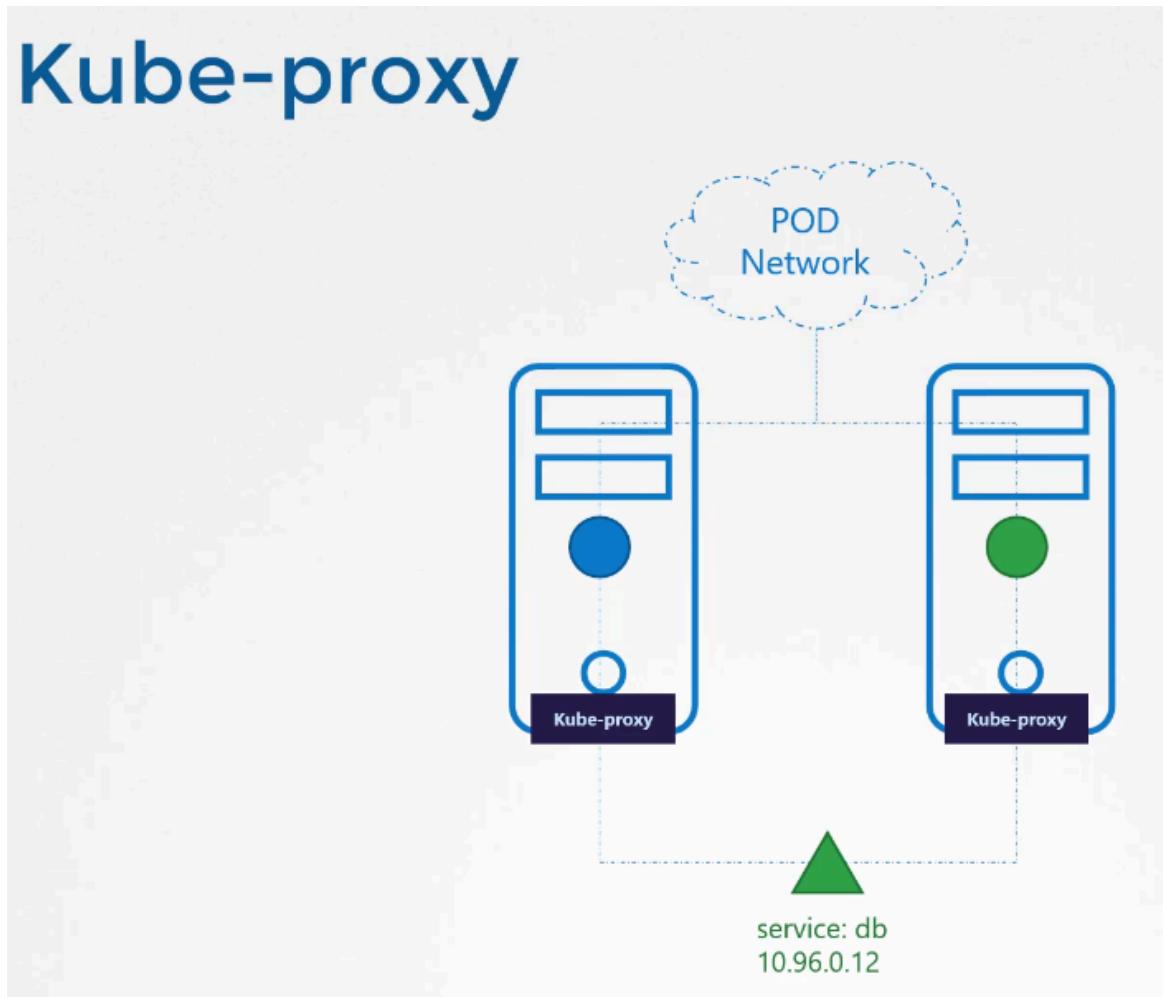
▼ 18. Kubelet

- **배의 선장. 클러스터의 각 노드에서 파드 안 컨테이너들이 정상 구동되도록 조율**하는 에이전트
- worker node에서 master node로의 유일한 연락망 역할
- kube scheduler가 파드를 어느 노드에 배치할지 결정해주면, **kubelet** 은 해당 노드에 파드를 배치하는 작업을 수행한다.
- 또한 파드와 컨테이너의 상태를 주기적으로 체크하여 그 결과를 API 서버에 전송하는 역할도 맡는다.
- `kubeadm` 으로 클러스터를 구축할 경우 **kubelet**이 포함되어 있지 않음에 유의해야 한다. 따라서 `kubeadm`, `kubectl` 과 함께 별도 설치를 진행해줘야 하며, 이때 상호 호

환성 문제가 일어나지 않도록 셋의 버전을 반드시 일치시켜야 한다. 이에 대한 내용은 아래 링크에서 확인할 수 있다.

- [kubernetes.io: Installing _kubeadm, kubelet and kubectl](#)

▼ 19. Kube Proxy



- cf) 파드: 다음 장에서 자세히..
- kube proxy: 쿠버네티스는 클러스터 안에 별도의 가상 네트워크를 설정하고 관리한다. `kube-proxy`는 이런 가상 네트워크의 동작을 관리하는 컴포넌트이다.
- 쿠버네티스 클러스터의 각 노드에서 실행되는 프로세스이다.
- 서비스가 생성될 때마다 각 노드에 적절한 규칙을 만들어 그 서비스로, 백엔드 pod로 트래픽을 전달한다. `iptables` 규칙을 사용하는 방법

▼ 20. Recap - Pods, 21. Pods with YAML, 22. Demo - Pods with YAML

<https://seongjin.me/kubernetes-pods/>

- pod: 1개 이상의 컨테이너가 캡슐화되어 클러스터 안에서 배포되는 가장 작은 단위의 객체
 - 이 course 내에서는 포드 당 컨테이너 하나씩만 쓸 것임! 다중 컨테이너 포드를 사용하는 사례는 드물기 때문.

▼ 파드의 특징

1. 기본적으로 하나의 파드에는 **하나 이상의 컨테이너**가 포함된다. 필요에 따라 하나의 파드에 여러 컨테이너를 포함시킬 수 있다.
2. 파드는 노드 IP와 별개로 고유 IP를 할당 받으며, 파드 안의 컨테이너들은 그 IP를 공유한다.
3. 파드 자체는 **일반적으로 1개의 IP**만 가진다. (단, [Multus CNI](#) 이용 등 특정 조건에 한해 2개의 IP를 가질 수도 있다.)
4. 파드 안의 컨테이너들은 동일한 볼륨과 연결이 가능하다.
5. 파드는 클러스터에서 배포의 최소 단위이고, 특정 네임스페이스(Namespace) 안에서 실행된다.
6. 파드는 기본적으로 **반영속적(ephemeral)**이다.

위에서 6번 항목에 주목하자. 동일한 역할을 하는 파드라도 환경에 따라 다른 노드들에 각각 배치될 수 있고, 특정 노드가 죽으면 해당 노드의 파드들이 건강한 상태의 다른 노드로 옮겨지기도 한다. 또는 필요에 따라 파드 숫자의 구성도 수시로 달라질 수 있다. **쿠버네티스에서의 파드는 무언가가 구동 중인 상태를 유지하기 위해 동원되는 일회성 자원이며, 필요에 따라 언제든 삭제될 수 있는 것임**을 유념해야 한다.

▼ 단일 컨테이너 파드

단일 컨테이너 파드(Single-Container Pod)는 오직 하나의 컨테이너만을 포함하고 있는 파드를 의미한다. 쿠버네티스 클러스터에서 운용되는 대다수의 파드가 이에 해당한다.

단일 컨테이너 파드의 경우 CLI 화면에서 `kubectl` 명령으로 간편하게 직접 배포할 수 있다. 혹은 명세(`spec`)를 포함한 YAML 파일을 이용해서도 배포할 수 있다. 예를 들어 `nginx` 컨테이너가 포함된 `nginx`라는 이름의 파드를 구동한다고 가정해보자.

▼ 1. CLI 명령으로 배포하기

마스터 노드에 터미널로 접속한 상태에서 다음과 같은 명령을 실행한다.

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

- 위 명령은 `kubectl run [파드명] --image=<이미지명>` 형태로 구성된다.
- `kubectl run` 은 `-image` 플래그로 지정된 이미지를 가져와 `[파드명]` 으로 지정된 이름의 파드를 생성하고 구동시키는 명령이다.
- `-image` 플래그는 필수 항목이다. 지정되지 않으면 파드 생성이 불가능하다.

▼ 2. YAML 파일로 배포하기

▼ 쿠버네티스 yaml 파일 기본 구조

4개의 루트 요소가 필요함! `apiVersion`, `kind`, `metadata`, `spec` 을 항상 기억 할 것

YAML in Kubernetes

The terminal shows the following content:

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
    type: front-end
spec:
  containers:
    - name: nginx-container
      image: nginx
```

Below the terminal, a blue bracket highlights the first item in the list: "1st Item in List". To the right is a table:

Kind	Version
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

- `spec.containers` 하위 요소에 `-`는 `containers`가 `list`이고 `-`는 첫번째 항목을 가르키는 것
- 탭은 권장되지 않으니 `space` 2개를 사용할 것

마스터 노드에 접속한 상태에서 원하는 경로에 아래 내용의 YAML 파일을 작성한 뒤, 터미널 화면에서 `kubectl apply -f <파일명.yaml>` 을 실행한다.

```
apiVersion: v1
kind: Pod # 대소문자 구별을 하므로 주의할 것
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
```

```
  containers:  
    - name: nginx-container  
      image: nginx
```

- `.apiVersion` : 객체 생성에 쓰일 쿠버네티스 API 버전을 의미한다. 객체 종류에 따라 버전이 조금씩 다름에 유의할 것. 파드와 서비스는 `v1`, 레플리카셋과 디플로이먼트 등은 `apps/v1` 으로 쓰인다.
- `.metadata.labels` : 파드 구동에 영향을 주지 않는 순수 키-값 메타데이터 영역이다. 원하는 조합으로 키-값을 부여할 수 있다. 수천 개의 파드가 돌아가는 환경에서 원하는 파드를 찾아 필요한 작업을 수행하려면 이러한 라벨링 작업에 신경써주는 것이 좋다.
- `.spec.containers` : 실제 파드에 담길 컨테이너의 속성을 정의하는 부분이다. 도커의 `docker-compose` 에 들어가는 내용과 상당 부분 유사하다.- YAML 파일에서는 상위 항목에 대해 리스트 형태의 하위 항목을 정의할 때 쓰인다.

▼ 3. 파드 관리에 쓰이는 주요 명령어

```
# 클러스터 내 파드 목록 조회(default 네임스페이스)  
kubectl get pods  
  
# 클러스터 내 특정 네임스페이스의 파드 목록 조회  
kubectl get pods -n <namespace>  
  
# 파드 목록의 상세 조회(사설IP, 노드정보 포함)  
kubectl get pods -o wide  
  
# nginx 파드를 수정  
kubectl edit pod nginx  
  
# nginx 파드의 상세 정보 확인  
kubectl describe pod nginx  
  
# nginx 파드 내 구동 중인 컨테이너의 로그 확인  
kubectl logs nginx  
  
# nginx 파드의 컨테이너에 접속하여 sh를 대화형으로 실행  
kubectl exec -it nginx -- /bin/sh
```

```
# nginx 파드 삭제  
kubectl delete pod nginx
```

- 위에서 두번째 명령어에 삽입된 "네임스페이스(Namespace)"는 하나의 클러스터 안에서 다양한 워크로드 리소스들을 필요에 따라 격리하는 데에 쓰이는 요소다. 이에 대해서는 이후 워크로드 리소스를 다루는 글에서 다시 소개할 예정이다.

▼ 4. 파드 배포시 유용한 팁

`kubectl` 을 이용한 객체 생성 및 배포는 CLI 커맨드로 간편히 진행할 수도 있지만, 상세 설정 내용을 검토하고 적용하여 진행하려면 YAML 파일 작성자를 통해 진행하는 것이 좋다.

파드를 비롯한 객체를 다룰 때 `kubectl` 커맨드에 `--dry-run=client` 와 `-o yaml` 옵션을 추가하면, 해당 명령으로 적용될 YAML 명령의 기본 골격을 파일 형태로 저장 할 수 있다. 이 경우 오직 명령문에 따라 작업이 정상적으로 완료될 수 있을 때에만 YAML 파일이 생성된다. 또한 파일만 생성될 뿐 실제 명령이 클러스터에 바로 적용되는 것은 아니므로, 앞으로 진행하려는 작업을 사전 검토할 때에도 유용한 옵션이다.

```
# Redis 파드 명세를 yaml 파일로 생성  
kubectl run redis --image=redis --dry-run=client -o yaml > redis.yaml
```

```
# 생성된 yaml 파일로 파드 구동(선언형 방식)  
kubectl apply -f redis.yaml
```

```
# 생성된 yaml 파일로 파드 구동(명령형 방식)  
kubectl create -f redis.yaml
```

▼ `kubectl run redis --image=redis --dry-run=client -o yaml > redis.yaml`

- `kubectl run nginx` : "nginx"라는 이름의 파드(Pod)를 생성하려고 합니다.
- `-image=nginx` : 이 파드는 Docker Hub의 공식 NGINX 이미지를 사용 합니다.
- `-dry-run=client` : 실제로 파드를 생성하지 않고 어떤 결과가 나올지 미리 확인합니다. 즉, 서버에 아무런 변화도 주지 않습니다.

- `-o yaml`: 결과를 YAML 형식으로 출력합니다. (`-o` : output)
- `> nginx.yaml`: 출력된 YAML을 "nginx.yaml"이라는 파일로 저장합니다.

▼ 멀티 컨테이너 파드

멀티 컨테이너 파드(Multi-Container Pod)는 2개 이상의 서로 다른 컨테이너를 포함하고 있는 파드를 의미한다. 본래 하나의 파드는 하나의 컨테이너를 포함하는 것이 일반적이지만, 반드시 함께 붙어있어야 하며 같은 생명주기를 공유하는 컨테이너들이 존재한다면 이들을 묶어 하나의 파드로 운영할 수 있다.

이렇게 하나의 파드에 묶여 배포된 컨테이너들은 아래와 같은 특징을 가진다.

1. 파드의 상태에 따라 함께 구동되고 함께 정지된다. 즉, 같은 생명주기를 공유한다.
2. 하나의 파드 안에서 서로 간편하게 통신이 가능하다.
3. 파드가 가진 스토리지(볼륨)를 함께 공유한다.

멀티 컨테이너 파드의 경우 실용성을 고려한 여러 **디자인 패턴**들이 정착되어 있다. 이에 대해서는 다음 글에서 보다 자세히 알아보기로 하자.

▼ 멀티 컨테이너 파드 YAML 파일로 배포하기

멀티 컨테이너 파드는 YAML 파일을 이용하여 배포해야 한다. 일반 파드를 배포할 때와 마찬가지로 아래 내용의 YAML 파일을 작성한 뒤, 터미널 화면에서 `kubectl apply -f <파일명.yaml>` 을 실행한다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-redis
  labels:
    app: nginx-redis
spec:
  containers:
    - name: nginx-container
      image: nginx
    - name: redis-container
      image: redis
```

- 기본적인 작성법은 단일 컨테이너 파드와 동일하다.

- `spec.containers` 아래에 리스트 형태로 삽입하고자 하는 컨테이너 정보를 입력한다.

▼ 컨테이너 로그 확인 및 접속

멀티 컨테이너 파드의 경우, 컨테이너 로그 확인 또는 접속시 컨테이너명을 함께 명시해줘야 한다.

```
# nginx-redis 파드의 redis-container 컨테이너의 로그 확인
kubectl logs nginx-redis redis-container
```

```
# nginx-redis 파드의 nginx-container 컨테이너에 접속하여 대화형으로 sh 실행
kubectl exec -it nginx-redis nginx-container -- /bin/sh
```

▼ 28. Recap - ReplicaSets, 31. Deployments

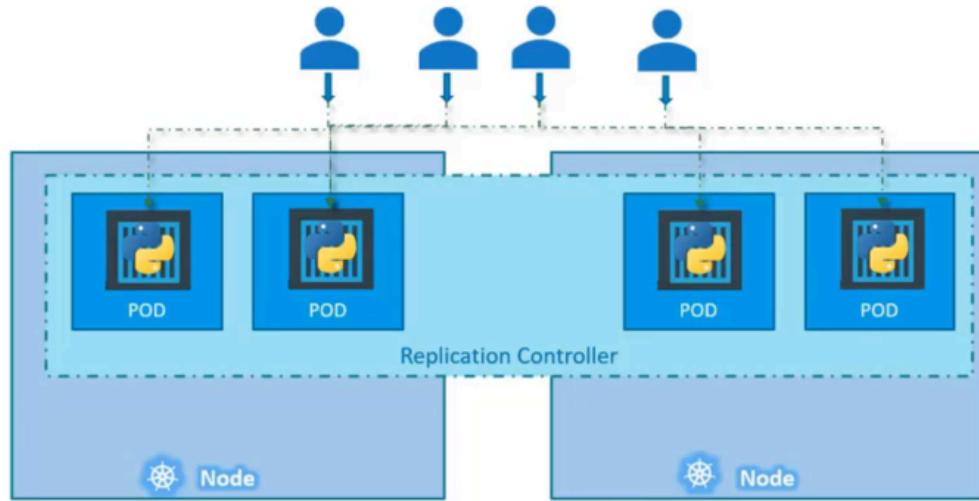
<https://seongjin.me/kubernetes-workloads/>

ReplicaSets (레플리카셋)

- **Replica**: 같은 application을 위해 동시에 구동되는 파드들의 집합
 - 상용 환경에서 애플리케이션을 오직 하나의 파드로만 운영하는 경우는 거의 없다. 다운 타임 없는 안정적인 서비스를 위해서, 대부분의 경우는 여러 개의 파드를 동시에 운영하게 된다.
- **레플리카셋(ReplicaSet)**은 레플리카 구성의 기준을 갖춘 파드들의 배포 규격을 정의하고 이를 관리하면서, 규격에 정의된 수 만큼의 파드가 언제나 정상 구동될 수 있도록 보장하는 역할을 한다. 만약 어떤 파드가 오류로 인해 정지되면, 동일한 스펙의 새 파드를 즉시 다시 배포시킨다.
- 쿠버네티스에서 사용되는 중요한 개념 중 하나가 바로 선언적 구성이다. 특정한 동작을 지시하는 것(**파드를 3개 만들어라**)이 아니라, 특정한 상태의 유지를 선언(**3개의 파드를 유지시켜라**)하는 것으로 시스템을 구성하는 개념이다. 이 개념의 대표적인 구현체 중 하나가 레플리카셋이다. 파드를 3개 갖고 싶다고 선언했는데 원치 않는 상황으로 인해 파드가 삭제되거나 노드가 고장났다면, 해당 파드를 복제하여 가용한 워커 노드에 다시 채워넣는 방식으로 처음에 선언했던 상태를 계속 유지시킨다.

과거에는 레플리케이션 컨트롤러(Replication Controller)가 이를 담당했으나, 쿠버네티스 1.2 기준으로는 레플리카셋으로 이 역할이 대체되었다.

Load Balancing & Scaling



- 복제 컨트롤러는 클러스터 내 여러 노드로 뻗어 있음. 서로 다른 노드의 여러 pod에 걸쳐 부하를 분산하는 데 도움이 되고, 수요가 증가하면 앱 scale도 조정 할 수 있다.

▼ 레플리카셋 생성하기

레플리카셋은 YAML 파일을 통해 생성 가능하다. 쿠버네티스 공식 메뉴얼에 게재된 예시를 살펴보자.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
```

```

spec:
  containers:
    - name: php-redis
      image: gcr.io/google_samples/gb-frontend:v3

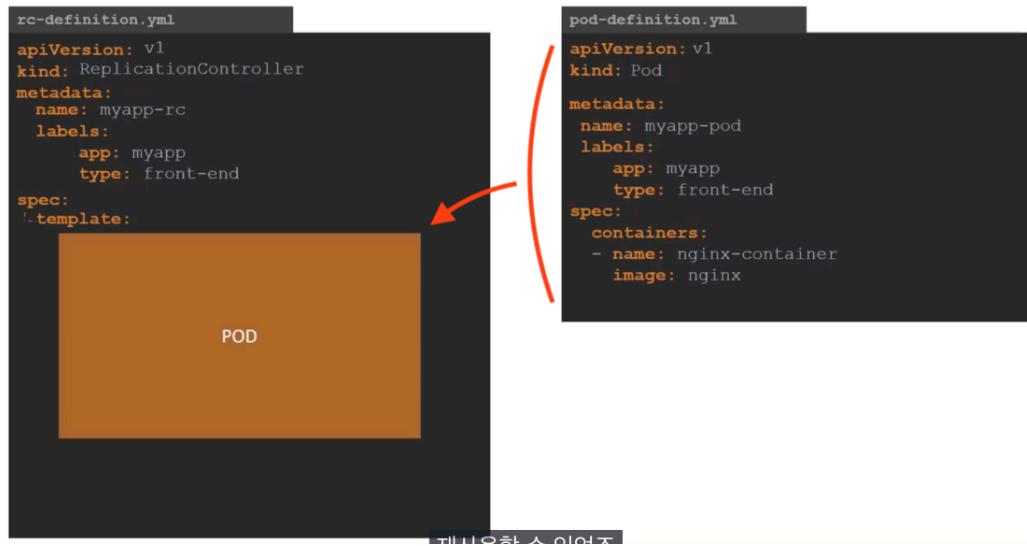
```

- ReplicaSet yaml의 spec 섹션에는 template, replicas, selector 세개의 섹션이 있다.

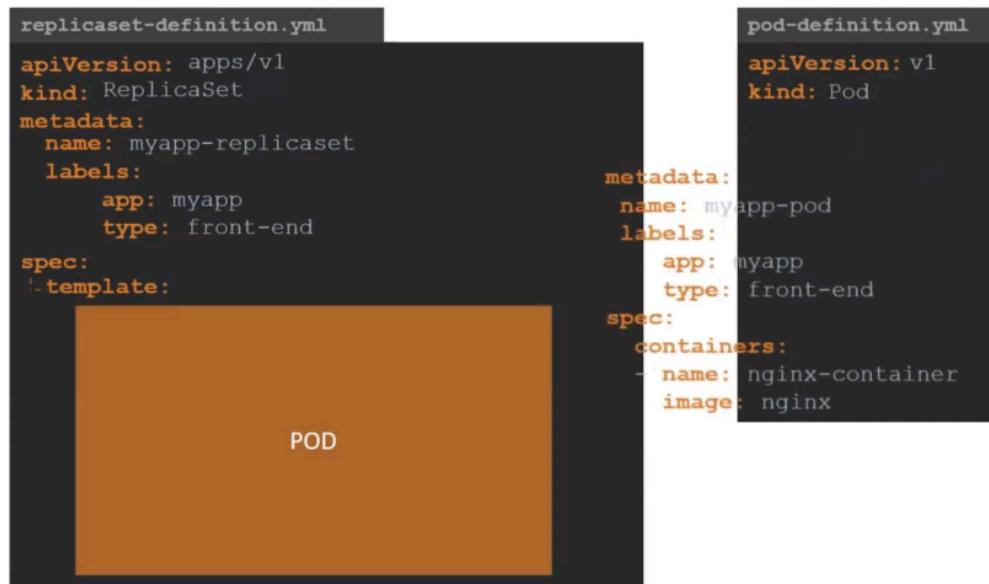
여기서 주의해야 할 점이 몇 가지 있다. 아래 사항들이 지켜지지 않으면 배포시 에러가 발생한다.

- `apiVersion` 이 `apps/v1` 인 점에 주의할 것. 파드, 서비스와 달리 복수의 객체를 다루는 레플리카셋, 디플로이먼트 등에서는 `apps/v1` 을 명시해야 한다.
- `spec.selector.matchLabels`, `spec.template.metadata.labels` 의 키-값 쌍은 반드시 동일해야 한다.

▼ yaml 작성 시 참고



- `spec.template` 섹션에는 replica controller가 ReplicaSet을 만들 때 사용할 pod template을 정의한다. pod 생성 yaml 작성할 때 처럼 했던 부분을 그대로 `spec.template` 하위에 두면 된다. 단, `apiVersion`, `kind` 부분만 빼고!
- ReplicaSet yaml 정의 시에도 마찬가지 ~



▼ 레플리카셋 관련 주요 명령어

대부분의 다른 쿠버네티스 객체들과 마찬가지로, `get` `describe` `delete` 등의 기본 커맨드를 활용 가능하다.

```

kubectl get rs
kubectl get replicaset
kubectl get all # pods, replicaset, deployment 전부 확인하고 싶을 때

kubectl describe rs/frontend
kubectl edit rs/frontend

# 레플리카셋 신규 배포
kubectl create -f replicaset.yaml

# 레플리카셋 신규/수정 배포
kubectl apply -f replicaset.yaml

# 레플리카셋 명세가 포함된 yaml 파일을 새로 적용
kubectl replace -f replicaset.yaml

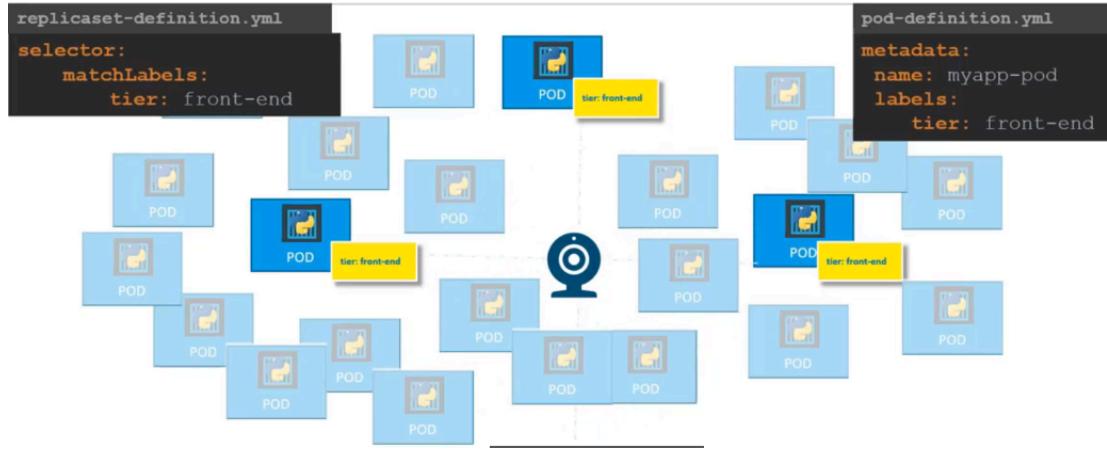
# yaml 파일 바탕으로 배포된 레플리카셋의 레플리카 수를 6으로 조정
kubectl scale --replicas=6 -f replicaset.yaml

```

```
# myapp 이름의 레플리카셋이 가진 레플리카 수를 6으로 조정
kubectl scale --replicas=6 replicaset myapp
```

▼ labels & selectors 부연설명

Labels and Selectors



- selector 부분은 ReplicaSets 하위에 놓인 포드를 식별할 수 있게 해준다.
 - 클러스터에 다른 application을 실행하는 포드가 수백개는 될 것인데, 이 때 selector를 통해 ReplicaSets이 어느 포드를 모니터하는지 빠르게 찾을 수 있는 것!

▼ 레플리카셋을 3개에서 6개로 업데이트하는 방법

1. yaml 파일의 replicas 설정 : `replicas: 6`, 그리고 `kubectl replace -f replicaset-definition.yml` 대체 명령어 실행
2. kubectl scale 명령어 실행 :
 - `kubectl scale --replicas=6 -f replicaset-definition.yml`
 - `kubectl scale --replicas=6 replicaset myapp-replicaset`

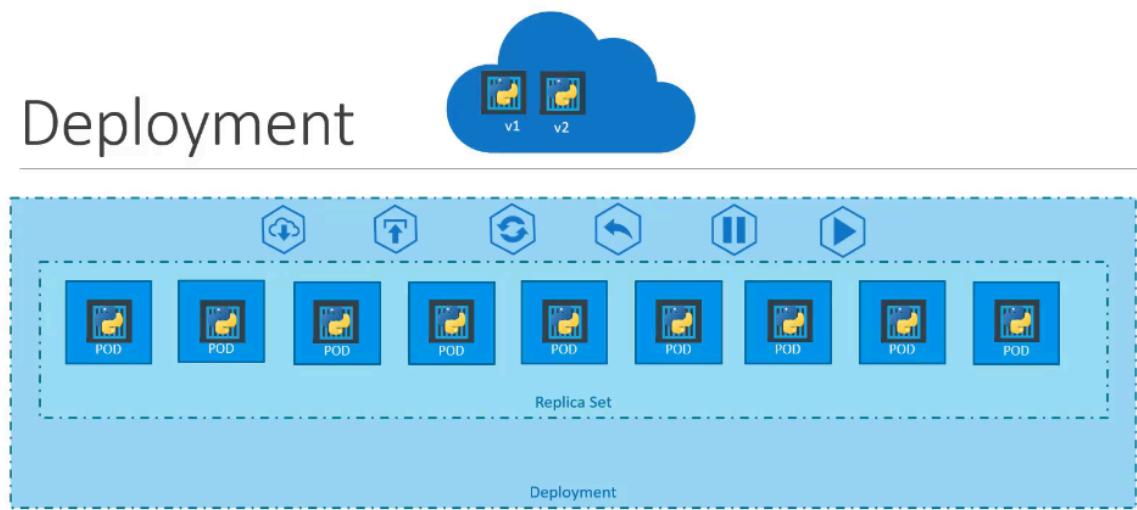
b 방식은 yml 파일의 복제본 수를 자동으로 업데이트 시켜주지는 않음!

```
> kubectl scale --replicas=6 replicaset myapp-replicaset
```



Deployment

파드와 레플리카셋에 대한 선언적인 업데이트를 제공하는 리소스다. 롤링 업데이트 (Rolling Update)를 비롯하여 애플리케이션 버전 및 배포 관리에 주로 쓰인다. 레플리카셋이 파드들의 상태 체크와 개수 유지를 담당한다면, 디플로이먼트는 **레플리카셋을 포함하는 상위 객체**로서 애플리케이션 또는 서비스 단위의 관리를 위해 쓰인다고 보면 된다.



디플로이먼트 리소스로 배포된 pod들은 모두 <디플로이먼트명>-<레플리카셋 고유번호>-<랜덤해시값> 형태의 식별자를 가진다. (아래 캡쳐 마지막 참고) 만약 레플리카 설정이 변경되거나, 파드가 삭제 후 재배포된 상황이라면 식별자도 함께 달라진다. 즉, 디플로이먼트에서 파드들의 식별자는 모두 상황에 따라 유동적으로 변화한다.

```
> kubectl create -f deployment-definition.yml
deployment "myapp-deployment" created
```

```
> kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
myapp-deployment   3         3         3             3           21s
```

```
> kubectl get replicaset
NAME          DESIRED   CURRENT   READY   AGE
myapp-deployment-6795844b58   3         3         3       2m
```

```
> kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
myapp-deployment-6795844b58-5rbjl   1/1     Running   0          2m
myapp-deployment-6795844b58-h4w55   1/1     Running   0          2m
myapp-deployment-6795844b58-1fjhv   1/1     Running   0          2m
```

▼ Deployment 배포하기

CLI 커맨드로 생성하기

```
kubectl create deployment nginx --image=nginx:1.14.2 --replicas=3
```

- 위 명령은 `kubectl create deployment <디플로이먼트명> --image=<이미지명> --replicas=<레플리카 수>` 형태로 구성된다.
- `-image`로 지정된 이미지의 파드를 `-replicas`에 지정된 숫자 만큼 생성하고 유지하는 `[디플로이먼트명]`의 디플로이먼트를 생성한다.
- `-image` 플래그는 필수 항목이다. `-replicas`는 지정하지 않을 경우 기본값인 1로 취급된다.
- deployment는 `deploy`로 줄여 쓸 수 있다.

▼ YAML 파일로 생성하기

마스터 노드에 접속하여 아래 YAML 파일을 작성, 저장한 뒤 터미널 화면에서 `kubectl apply -f <파일명.yaml>`을 실행한다.

```
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:1.14.2
```

ports:

- containerPort: 80

- `kind` 의 값은 반드시 `Deployment` 여야 한다. (대소문자 구분 필수)
- 레플리카의 수는 `spec.replicas` 에 입력한다.
- 배포할 파드에 대한 정보는 `spec.template` 아래에 위치하게 된다. 파드에 들어갈 컨테이너 이미지 정보는 `spec.template.spec.containers` 아래에 입력한다.
- 이렇게 배포된 파드들을 일괄 관리하기 위해 `spec.selector.matchLabels` 항목을 통해 키-값 쌍을 부여한다.
- `spec.selector.matchLabels`, `spec.template.metadata.labels`에 명시된 키-값 쌍은 반드시 동일해야 한다.
- replicaset yaml 설정에서 kind를 Deployment로 바꾸는 것 제외하고 모두 동일하게 하면 된다

```
deployment-definition.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
        - name: nginx-container
          image: nginx
      replicas: 3
      selector:
        matchLabels:
          type: front-end
```

▼ 배포된 Deployment의 설정 변경

디플로이먼트로 배포되어 이미 구동 중인 파드의 이미지를 업데이트하거나, 레플리카의 수를 조정해야 할 때가 있다. 이런 경우엔 아래와 같이 조정해주자. 디플로이먼트에 대한 업데이트 및 롤백에 대한 자세한 내용은 별도의 포스팅으로 다시 소개할 예정이다.

이미지 업데이트

우선 CLI 환경에서는 `kubectl set` 명령을 이용할 수 있다. 아래 명령은 위에서 배포한 `nginx-deployment` 의 `nginx` 앱에 대해 컨테이너 이미지를 `nginx:1.14.2`에서 `nginx:1.16.1`로 업데이트하는 내용이다.

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
```

다음으로는, `kubectl edit` 명령으로 배포된 디플로이먼트의 YAML 정보를 불러와 컨테이너 이미지 항목을 수정하는 방법이 있다. 이때 수정해야 할 항목

은 `.spec.template.spec.containers[0].image`에 위치해 있다.

```
kubectl edit deployment/nginx-deployment
```

만약 YAML 파일로 직접 생성한 경우라면, 해당 YAML 파일을 열어 `spec.template.spec.containers`에 `image` 항목을 `nginx:1.16.1`로 변경한 뒤 `kubectl apply` 명령을 사용해도 된다.

Scaling

CLI 환경에서 간단히 레플리카 수를 조정하고 싶다면 `kubectl scale` 명령을 이용할 수 있다. 아래 명령을 실행하면 `nginx-deployment`의 레플리카 수가 5개로 즉시 늘어난다.

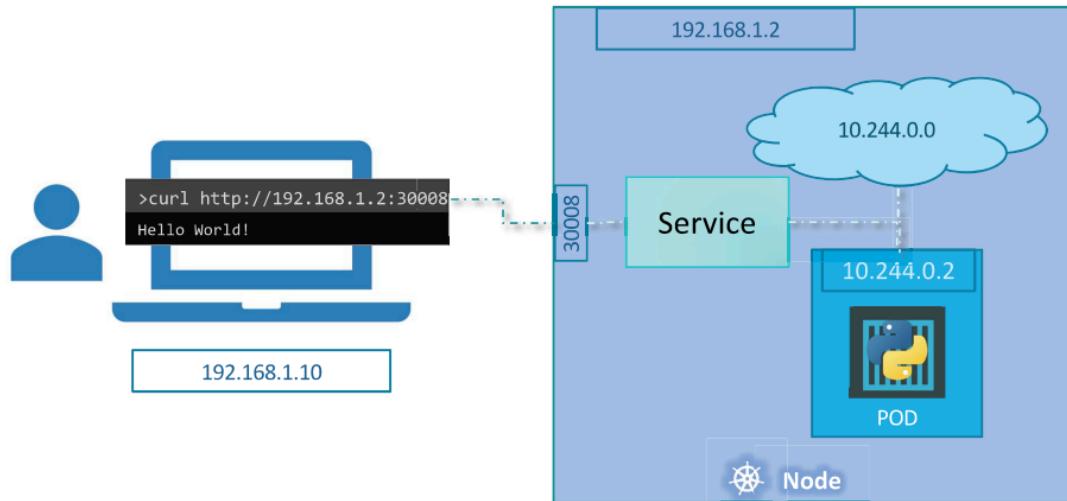
```
kubectl scale deployment/nginx-deployment --replicas=5
```

단, 위의 명령은 실제 배포에 쓰인 YAML 파일 내용과 무관하게 명시적으로만 적용되는 변화다. `kubectl edit`이나 기존 배포에 쓰인 YAML 파일을 직접 수정 후 적용하는 방법도 있으므로 용도에 따라 방법을 선택하면 좋다.

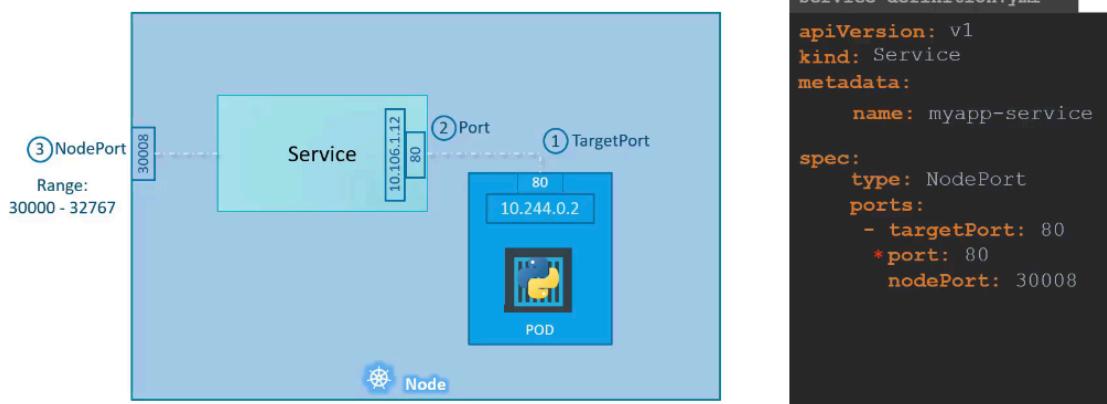
▼ 35. Services, 36. Services Cluster IP, 37. Services - Loadbalancer

▼ NodePort

Service - NodePort

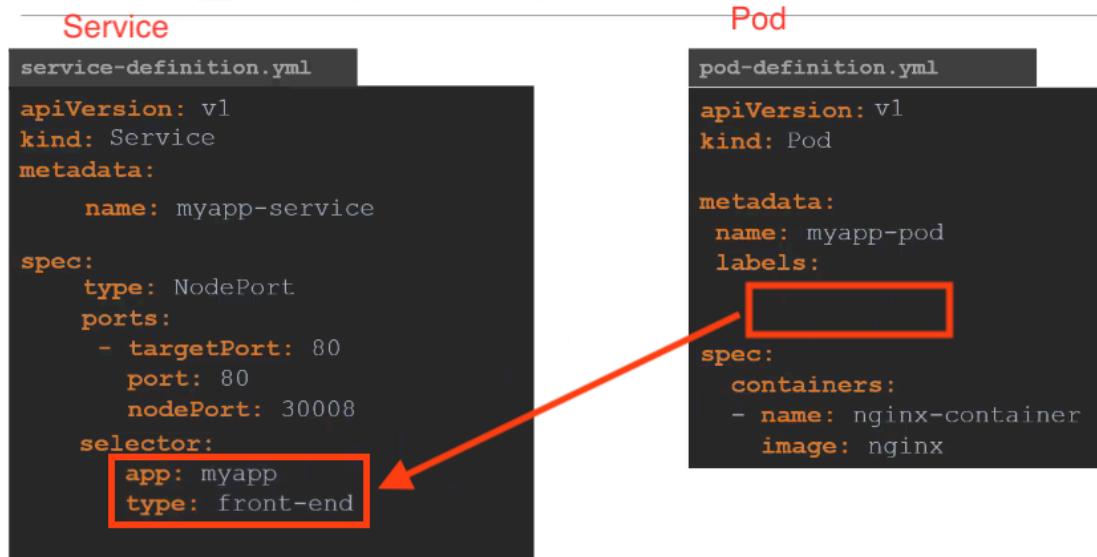


Service - NodePort



- spec.ports에서 port만 필수, targetPort는 적지 않으면 port와 같게 설정되고 nodePort는 적지 않으면 30000~32767 중에 랜덤한 포트로 설정된다.

Service - NodePort



- selector 하위에 pod의 labels 하위 항목들을 기입하여 서비스와 pod를 연결한다.

Service - NodePort

```
service-definition.yml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  type: NodePort
  ports:
    - targetPort: 80
      port: 80
      nodePort: 30008
  selector:
    app: myapp
    type: front-end

> kubectl create -f service-definition.yml
service "myapp-service" created

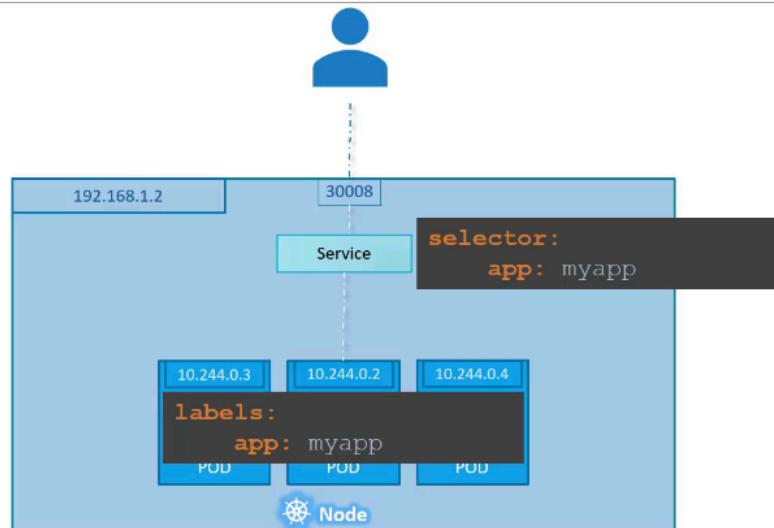
> kubectl get services
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)       AGE
kubernetes     ClusterIP  10.96.0.1   <none>        443/TCP      16d
myapp-service  NodePort   10.106.127.123 <none>        80:30008/TCP  5m

> curl http://192.168.1.2:30008
<html>
<head>
<title>Welcome to nginx!</title>
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
```

- 위와 같이 yaml 적고 service를 create하면, `curl http://192.168.1.2:30008` 을 통해 웹 서버에 접근할 수 있다! `http:{Node IP}:{Node Port}`

여기까진 한 노드에 파드가 하나일 때의 예시이다. 만약 노드 안에 고가용성(high availability)과 로드밸런싱을 위해 파드가 여러개라면?

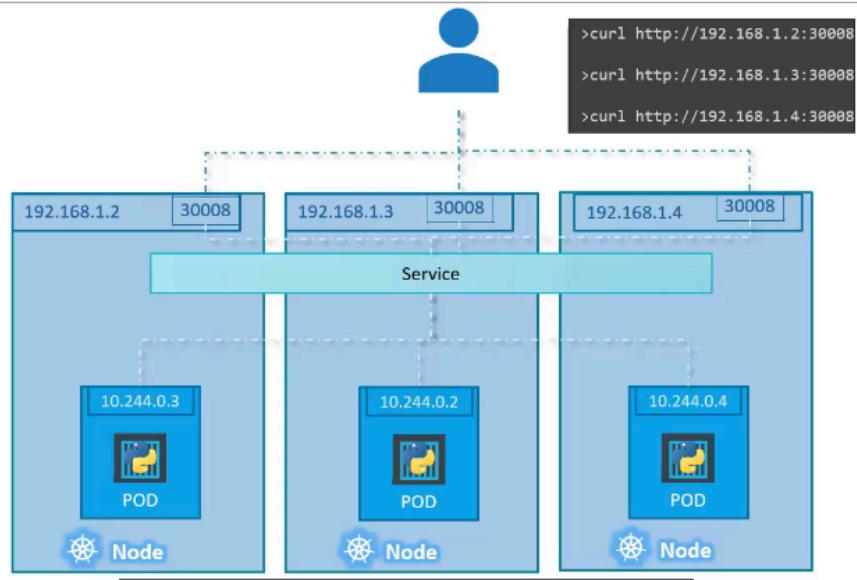
Service - NodePort



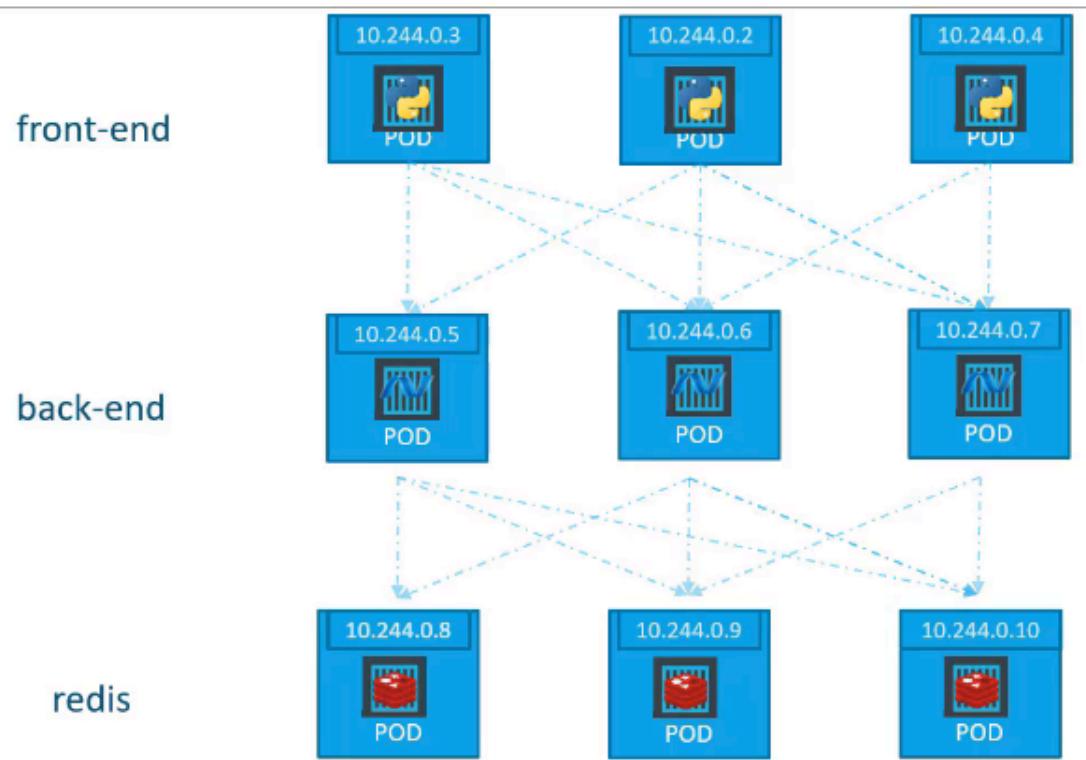
- Service에 selector를 지정하면, 매칭되는 label을 가진 pod를 찾는다. Service는 자동으로 end-point인 세 개의 pod를 찾아서 사용자로부터 받은

외부 요청을 forwarding한다.

Service - NodePort

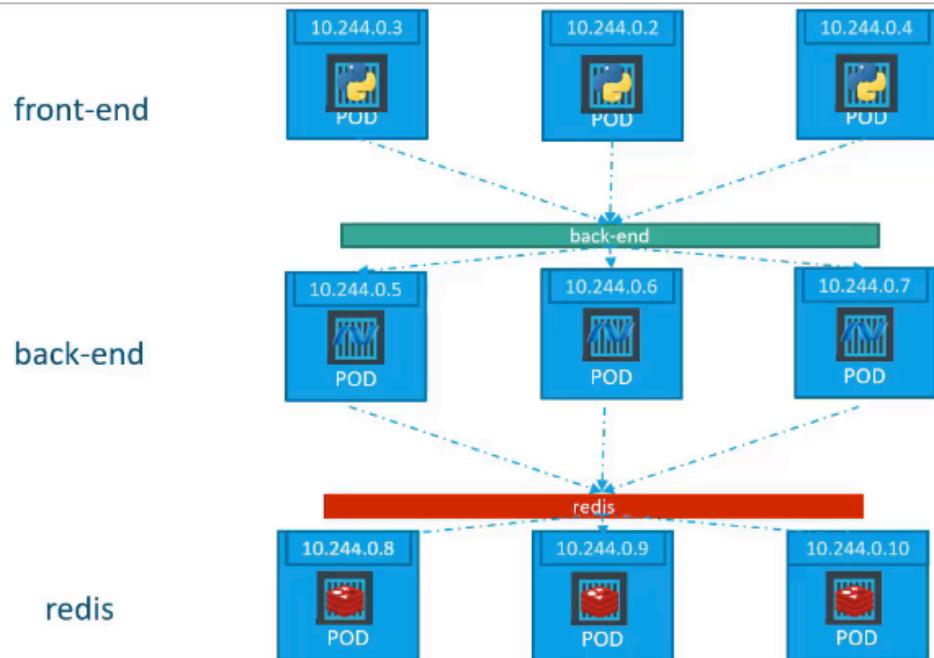


▼ ClusterIP



- 예를 들어 프론트엔드 pod들, 백엔드 pod들, redis pod들이 있을 때 프론트엔드 서버는 백엔드 서버와 통신하길 원하고, 백엔드 서비스는 Redis 서비스들과 통신하길 원한다. 이런 서비스 혹은 계층 간의 연결을 확립하는 올바른 방법은 무엇일까?
- 위 캡쳐와 같이, 포드들은 모두 IP 주소를 할당받지만, 이 정적인 IP가 아니다. 포드들은 언제든 삭제될 수 있고, 새 포드는 계속 만들어지므로 앱 간의 내부 통신은 이 IP 주소에 의존할 수 없다.
- 프론트엔드 pod 중 10.244.0.3은 백엔드와 통신할 때 백엔드의 몇번째 파드와 통신해야 할까?

ClusterIP



- 쿠버네티스 Service를 통해, 각 계층의 포드들을 하나로 묶고, 하나의 인터페이스를 통해 그룹화된 포드에 접근할 수 있다. 요청은 random한 pod로 전달된다.
- 이를 통해, 쿠버네티스 클러스터에 마이크로서비스 기반의 application을 쉽고 효과적으로 배포할 수 있다.
- 각 계층은 서비스 간의 통신에 전혀 영향을 주지 않으면서 필요한 만큼 확장 또는 이동할 수 있다.
- 각각의 service는 클러스터 내부에서 IP와 name을 갖고 있다. 다른 포드가 서비스에 접근하려면 name을 사용해야 한다. 그리고 이런 서비스 유형을 바로 클

러스터 IP라고 한다.

```
service-definition.yml
apiVersion: v1
kind: Service
metadata:
  name: back-end
spec:
  type: ClusterIP
  ports:
    - targetPort: 80
      port: 80
  selector:
    app: myapp
    type: back-end
```

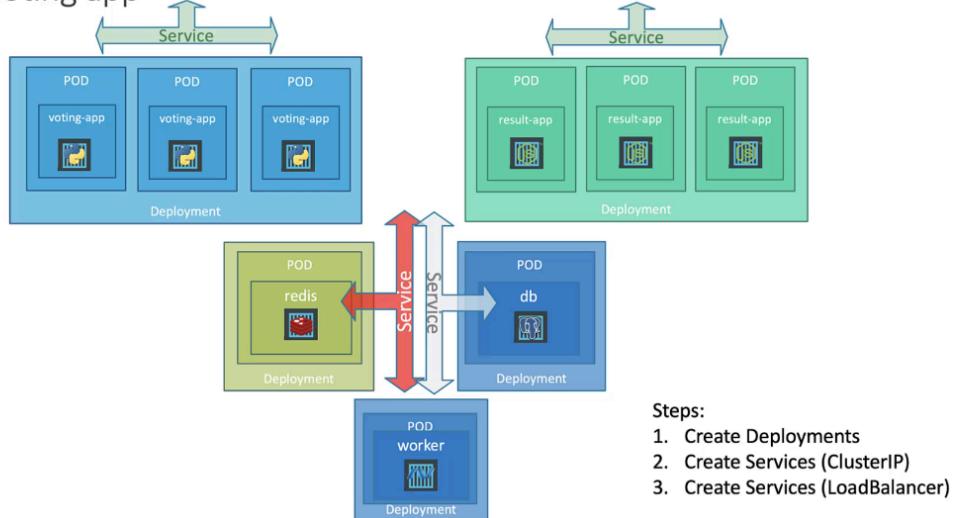
```
> kubectl create -f service-definition.yml
service "back-end" created
```

```
> kubectl get services
NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes     ClusterIP  10.96.0.1   <none>       443/TCP   16d
back-end       ClusterIP  10.106.127.123  <none>       80/TCP    2m
```

▼ Load Balancer

frontend 앱 두 개 (투표 앱, 투표 결과 앱) application 예시

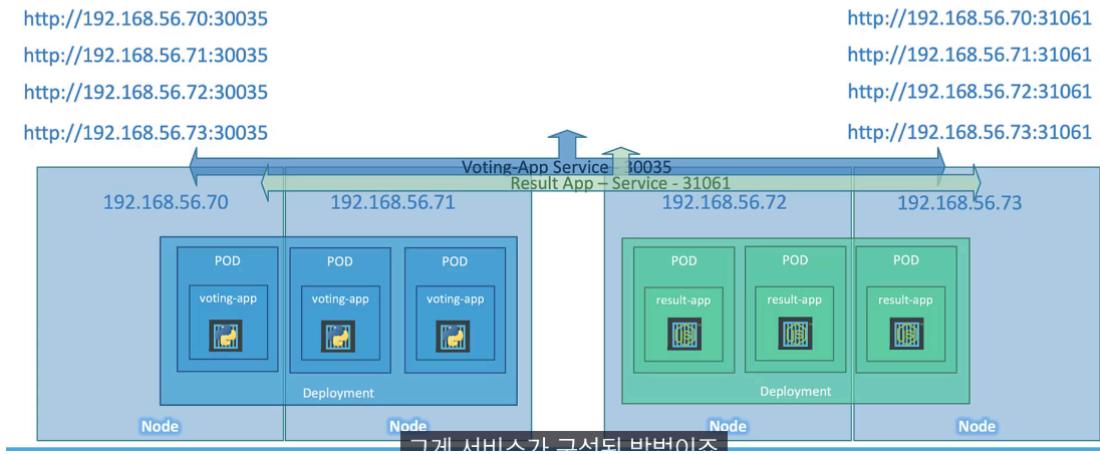
Example voting app



- pod들은 cluster 안의 worker node에서 호스트된다.
- 노드 클러스터가 있다고 가정해보자. 외부 사용자에게 application을 접근 가능하게 하기 위해 NodePort 타입의 Service를 만든다. NodePort Service를 이용하면 NodePort에서 트래픽을 수신하고, 각 포트로 트래픽을 routing 할 수 있다.
- end-user에게 application에 접근하도록 하기 위해 어떤 url을 주어야 할까?
 - 사용자는 이 중 아무 노드와 서비스가 있는 high port를 이용해서 이 두 application 중 어떤 것이든 접근할 수 있다.

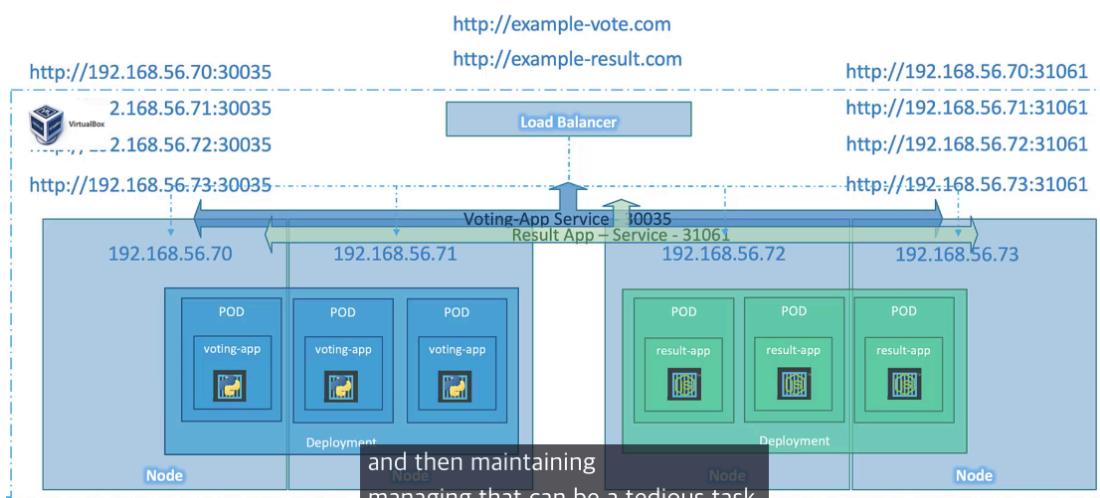
- 포트가 두 노드에서만 호스트되더라도, 여전히 클러스터의 모든 노드의 IP에서 access할 수 있다.

Example voting app



- 투표 앱 포트들이 첫번째(IP 마지막자리 70)와 두번째(IP 마지막자리 71) 노드에만 호스트된다고 해도, 투표 앱 포트들은 여전히 클러스터 내 모든 노드들의 IP에 접근할 수 있다.

Example voting app



- 하지만, end-user는 이런걸 원하는게 아니다. <http://example-vote.com> 과 같은 단일 url이 필요하다. 이를 위한 한가지 방법은, load balancer를 위한 새로운

VM을 생성하고 적합한 load balancer를 설치, 구성하는 것이다. (ex. HA proxy, nginx 등) 그 다음, 트래픽을 아래의 노드로 라우팅하도록 load balancer를 구성한다.

- 모든 외부 부하 분산을 설정하고 관리를 유지하는 것은 지루한 작업이 될 수 있다. 하지만 Google Cloud, AWS, Azure와 같은 지원되는 클라우드 플랫폼에 있다면, 우리는 그 클라우드 플랫폼의 기본 로드 밸런서를 활용할 수 있다.

Kubernetes는 특정 클라우드 제공업체의 네이티브 로드 밸런서와 통합하여 이를 구성할 수 있도록 지원한다.

프론트엔드 서비스의 유형을 NodePort 대신 LoadBalancer로 설정하기만 하면 된다.

이는 지원되는 클라우드 플랫폼에서만 작동한다는 점을 기억하자. (GCP, AWS, Azure는 확실히 지원됨)

서비스 유형을 지원되지 않는 환경(ex. VirtualBox)이나 다른 환경에서 로드 밸런서로 설정하면, 서비스가 노드의 고급 포트에 노출되는 NodePort로 설정하는 것과 동일한 효과를 갖게 된다. 거기서는 외부 로드 밸런서 구성을 전혀 수행하지 않는다.

▼ NameSpace

NameSpace: 하나의 동일한 물리 클러스터를 공유하는 가상 클러스터

- 클러스터의 자원을 여러 사용자나 용도에 따라 나누는 기능이 필요할 때 쓴다.
- 네임스페이스는 워크로드 리소스는 아니지만, 하나의 클러스터 안에서 다양한 워크로드 리소스들을 필요에 따라 격리하는 데에 쓰인다.

왜 필요한가?

실제 프로젝트에서는 개발 단계에 따라 여러 환경(Dev, Staging, Production, etc...)을 동시 운영하게 될 수 있다. 네임스페이스는 별도로 물리적인 호스트 환경의 격리 없이, 이처럼 상호 다른 환경의 특성에 맞게 정책과 리소스량을 정하여 격리된 환경을 만들 수 있는 것이다.

예를 들어, 같은 네임스페이스 안에서 리소스의 명칭은 반드시 고유해야 하지만, 다른 네임스페이스들을 함께 통틀어서 고유할 필요는 없다. 정책(Policy), 리소스 호출 등도 네임스페이스 범위를 기준으로 적용된다.

`kubectl get` 명령어로 특정 유형의 리소스 목록을 조회할 때, 이 명령은 현재 유효한 네임스페이스의 범위 안에서만 적용된다. 별도로 특정 네임스페이스를 현재 작업 환경으로 지정한 상태가 아니라면, 기본값인 `default` 네임스페이스에 속한 리소스만 조회된다.

기본으로 주어지는 Namespace

- `default` : 다른 네임스페이스가 없는 오브젝트를 위한 기본 네임스페이스
- `kube-system` : 쿠버네티스 시스템에서 생성한 오브젝트를 위한 네임스페이스
- `kube-public` : 모든 사용자(인증되지 않은 사용자 포함)가 읽기 권한으로 접근할 수 있는 네임스페이스다. 주로 전체 클러스터 중에 공개적으로 드러나서 읽을 수 있는 리소스를 위해 예약되어 있다. 다만 이 특성은 단지 관례로서 적용되어 있을 뿐, 필수적인 사항은 아니다.
- `kube-node-lease` : 클러스터가 스케일링될 때 노드 하트비트의 성능을 향상시키는 각 노드와 관련된 리스(lease) 오브젝트에 대한 네임스페이스

네임스페이스 관련 명령어 정리

```
# 네임스페이스 목록 조회하기
kubectl get namespace -A

# 현재 네임스페이스 확인하기
kubectl config view | grep namespace

# 네임스페이스 "test" 추가하기
kubectl create namespace test

# 현재 활성상태인 네임스페이스의 이름 변경하기
kubectl config set-context --current --namespace=<새로-바꿀-이름>

# 네임스페이스 "test" 삭제하기
kubectl delete namespace test
```

- 주의할 점이 있다. 네임스페이스를 삭제할 경우, 해당 네임스페이스에 속한 모든 객체, 리소스가 함께 삭제된다. 만약 `kube-system` 네임스페이스를 삭제한다면 클러스터가 동작하지 않게 된다.

객체 배포 조회시 네임스페이스 설정하기

- `-namespace` 또는 `n` 플래그를 사용한다. 예시는 다음과 같다.

```
# "test" 네임스페이스에 nginx 파드 배포하기
kubectl run nginx --image=nginx --namespace=test

# "test" 네임스페이스에 속한 파드 목록 조회하기
```

```
kubectl get pods -n test
```

```
# 모든 네임스페이스에 속한 파드 목록 조회하기  
kubectl get pods --all-namespaces  
kubectl get pods -A
```

CLI 환경에서 사용할 네임스페이스 설정하기

`kubectl config set-context` 명령을 통해 이후 CLI 환경에서 사용하는 네임스페이스를 영구 지정할 수 있다. 기본적으로는 `default`로 설정되어 있다.

```
# 현재 CLI 환경에서 사용할 기본 네임스페이스 설정하기  
kubectl config set-context --current --namespace=test
```

```
# 현재 기본값으로 설정된 네임스페이스 확인하기  
kubectl config view --minify | grep namespace:
```

서로 다른 네임스페이스 간의 네트워크 통신

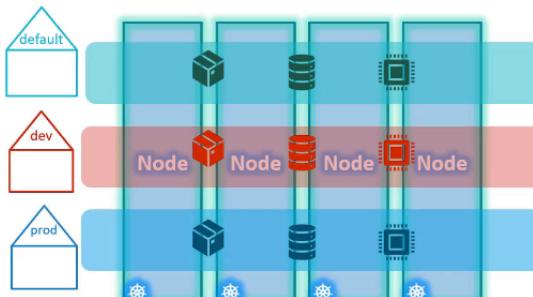
같은 네임스페이스 안의 파드들은 이름(service name)을 기준으로 서로를 인식할 수 있다. 만약 다른 네임스페이스의 파드와 연결하고자 한다면, `<service-name>. <namespace>.svc.cluster.local`의 형식으로 원하는 대상을 찾아 식별할 수 있게 되어있다.

예를 들어 `dev` 네임스페이스에 속한 `db-service` 란 이름의 파드를 호출하려면, `db-service.dev.svc.cluster.local` 형식으로 호출하면 된다.

Resource Quota (리소스 할당량)

- namespace에서 리소스를 제한하려면 리소스 할당량을 생성한다.
- 아래 yaml 참고 - 10개의 pod, 4개의 cpu, 10GB 메모리 제한 등 설정

Resource Quota



```
Compute-quota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: dev
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```

```
> kubectl create -f compute-quota.yaml
```

▼ 43. Imperative vs Declarative

프로그래밍의 주요 패러다임으로 명령형 프로그래밍과 선언형 프로그래밍이 있듯이, 코드로서의 인프라(IaC; Infrastructure as a Code) 영역에서도 **명령형 접근법**과 **선언형 접근법**이 존재한다.

쿠버네티스에서는 두 가지 접근법이 모두 쓰인다. CLI 명령어를 통한 객체 관리는 **명령형 접근법**에, `kubectl apply` 명령을 통한 YAML 파일 기반 객체 관리는 **선언형 접근법**에 해당한다. 이 두 가지 접근법은 각각의 장단점이 있으므로, 상황과 필요에 따라 구분하여 활용해야 한다. 많은 개발 및 상용 환경에서는 프로젝트의 성격에 따라 클러스터 및 하부 요소들의 세밀한 관리가 요구되므로, YAML 기반의 선언형 접근법이 보다 주요하게 쓰일 것이다.

명령형(Imperative) 접근법

명령형 접근법은, 원하는 상태를 만들기 위해 필요한 동작을 지시하는 방식이다. 쿠버네티스에서는 아래와 같이 CLI 환경에서 `kubectl`을 통한 구성 요소 생성/수정/삭제 명령어를 수행하는 방식이 이에 해당한다. 이러한 접근법은 결국 "**요구되는 환경을 어떻게(how) 만들 것인가**"에 초점을 두고 있다.

```
kubectl run nginx --image=nginx
kubectl create deployment nginx --image=nginx
kubectl expose deployment nginx --port=80
kubectl edit deployment nginx
kubectl scale deployment nginx --replicas=5
kubectl set image deployment nginx nginx=nginx:1.18
kubectl <create|replace|delete> -f nginx.yaml
```

YAML 파일을 통해 '구성 요소의 상태'를 정의해 놓은 경우라도, `create` 나 `replace`에 해당하는 작업을 실행한다면 이는 명령형 접근법에 속한다. 해당 파일의 내용을 토대로 어떤 작업을 수행할 것인지를(생성/대체/삭제) 명시적으로 지시하는 방식이기 때문이다. 예를 들어 `nginx` 파드가 이미 구동 중인 상태에서 같은 이름의 파드가 정의된 YAML 파일로 `create` 명령을 실행하면 중복 오류가 발생한다. 반대로 `nginx` 파드가 존재하지 않는데 같은 파일로 `replace` 명령을 실행한다면 마찬가지로 오류가 발생한다.

정리하자면, 명령형 접근법은 아래와 같은 한계점을 가진다.

1. **명령어 만으로 수행 가능한 작업이 제한적이다.** 특히 멀티 컨테이너 파드처럼 복잡도가 있는 쿠버네티스의 구성 요소를 명령어 만으로 하나하나 설정하기는 어렵다.
2. **작업 내역을 추적하기 어렵다.** 여러 사람이 함께 하는 협업 환경에서는 누군가가 명령어로 단순 생성시킨 특정 요소의 히스토리를 파악하기 힘들다.
3. **현재 작업 환경의 설정사항을 직접 파악해야 한다.** 앞서 YAML 파일로 `create`, `replace`, `delete` 실행 시 오류가 발생하는 경우들을 살펴봤었다. 이런 오류의 가능성으로 인해, 명령 수행 전에 현재 작업 환경을 수동으로 체크해야 하는 과정이 추가로 요구된다.

다만 필요한 요소를 명령어 한 줄로 즉시 생성하여 다룰 수 있게 해주는 것은 명확한 장점이므로, 빠르고 간결한 작업 수행이 요구되는 환경에서 명령형 접근법은 여전히 유용할 수 있다.

선언형(Declarative) 접근법

선언형 접근법은, 원하는 상태 그 자체를 선언하는 방식이다. 쿠버네티스에서는 YAML 파일을 통해 원하는 구성 요소의 원하는 상태를 기술한 뒤, `kubectl apply -f <파일명.yaml>` 형태의 명령어로 이를 적용하는 방식이 해당된다. 이렇게 선언된 상태를 실제로 적용하기 위해 필요한 작업은 쿠버네티스 시스템이 알아서 판단하고 수행한다. 이 접근법은 결국 "요구되는 환경이 무엇인가(what)"에 초점을 둔다.

앞서 파드 설명 부분에서 다루었던 `nginx` 파드의 YAML 정의 파일(`nginx.yaml`)을 다시 참고하면 다음과 같다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
```

```
- name: nginx-container
  image: nginx
```

위의 파일을 통해 `nginx` 파드를 배포했는데, 이 파드에 쓰이고 있는 이미지를 특정 버전 (`nginx:1.20.2`)으로 수정하여 재배포해야 한다고 가정해보자. 이 경우엔 우선 YAML 파일을 아래와 같이 수정할 수 있을 것이다.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
    - name: nginx-container
      image: nginx:1.20.2
```

이렇게 수정된 내용을 기존의 파드에 적용하고 싶다면, 다음의 명령어를 실행하면 된다.

```
kubectl apply -f nginx.yaml
```

여기서 수행하는 `kubectl apply` 명령이 바로 쿠버네티스의 선언적 접근법에 해당하는 부분이다. 이 명령은 적용하고자 하는 YAML 파일이 요구되는 문법에 맞게 작성되었는지를 검토한 뒤, 해당되는 요소가 기존의 클러스터에 이미 배포되어 있는지를 체크하고 있다면 업데이트를, 없다면 신규 생성을 알아서 진행한다. 관리자가 선언한 특정한 상태를 시스템이 스스로 파악하여 반영하는 이 프로세스가 바로 쿠버네티스에서의 선언형 접근법이다.

선언형 관리 환경에서의 주의점: 명령형 커맨드를 혼용하지 말 것!

YAML 등 구성 파일을 통해 쿠버네티스 요소들을 선언형으로 관리하는 환경라면, `kubectl`에서 제공하는 `create`, `edit` 또는 `scale` 같은 명령을 `kubectl apply` 와 절대 혼용해서는 안 된다. 만약 혼용할 경우, 이미 생성된 요소들의 추후 업데이트 과정에서 의도하지 않은 결과가 나오거나 업데이트 이력에 대한 추적 관리가 어려워진다. 왜일까?

이 문제를 이해하려면, 우선 **활성 객체 설정(Live Object Configuration)**이라는 개념과 이에 연관된 `kubectl apply`의 동작 방식을 이해해야 한다.

활성 객체 설정(Live Object Configuration)

쿠버네티스에서는 클러스터 안에서 구동 중인 객체에 대한 정보를 담고 있는 일종의 데이터로, 쿠버네티스 클러스터 스토리지(대개의 경우 `etcd`)에 저장되어 있다. 이 내용은 `kubectl get <객체종류> <객체명> -o yaml` 명령으로 확인 가능하다.

만약 YAML 형식의 구성 파일로 만든 객체라면, 이 "활성 객체 설정(**live object configuration**)"에 한 가지 내용이 더 추가된다. 앞서 쓰인 YAML 파일의 내용이 JSON 포맷으로 변환되어 어노테이션(`metadata.annotation`) 안에 `kubectl.kubernetes.io/last-applied-configuration` 항목으로 함께 삽입되는 것이다. 여기에는 `kubectl apply` 가 실행된 가장 최근의 시점을 기준으로 원본 객체 구성 파일(**Object Configuration File**)에 있던 내용이 그대로 반영되어 있다. 이것을 **최신 적용 설정**(`last-applied-configuration`)이라 부른다.

즉, 쿠버네티스의 선언형 관리 환경에서는 `kubectl apply` 명령으로 현재 활성 상태인 요소에 변화를 줄 때 다음의 3가지 항목이 함께 활용된다.

1. YAML 포맷의 원본 객체 구성 파일(**Object Configuration File**) : yaml을 직접 수정할 때에만 바뀜
2. 1번의 내용이 JSON 포맷으로 변환된 최신 적용 설정(`last-applied-configuration`) : yaml을 직접 수정할 때에만 바뀜
3. 2번의 내용을 어노테이션으로 포함하고 있는 활성 객체 설정(**Live Object Configuration**) : 현재 실행중인 느낌인듯..?? yaml만 바꾸고 replace를 다시 안 하면 현재 실행중인건 그대로이니까..

`kubectl apply` 의 동작 방식

`kubectl apply` 는 원본 객체 구성 파일(1번)의 내용을 최신 적용 설정(2번)으로 옮겨 업데이트 시키는 명령이다. 이때 활성 객체 설정(3번)이 업데이트 되는 과정은 경우에 따라 약간의 차이가 있다.

경우 1. 항목(필드)이 추가되거나 수정될 경우

1. 쿠버네티스는 활성 객체 설정(3번)을 원본 객체 구성 파일(1번)과 대조시킨다.
2. 만약 서로 일치하지 않는 항목이 있다면 해당 항목이 새로 추가되거나 수정된 것으로 간주하고, 이를 원본 객체 구성 파일(1번) 기준으로 업데이트 시킨다.
3. 최신 적용 설정(2번)은 원본 객체 구성 파일(1번) 내용을 따라 업데이트 된다.

경우 2. 항목(필드)를 삭제할 경우

1. 쿠버네티스는 최신 적용 설정(2번)에는 있지만 원본 객체 구성 파일(1번)에는 없는 항목을 탐색한다.
2. 이렇게 발견된 항목은 삭제 대상으로 간주하고, 해당 항목을 활성 객체 설정(3번)에서 제거시킨다.

3. 최신 적용 설정(2번)은 원본 객체 구성 파일(1번) 내용을 따라 업데이트 된다.

그렇다면 왜 **최신 적용 설정(2번)**과 **활성 객체 설정(3번)**을 굳이 별개로 구분하여 생각해야 하는 걸까? 만약 객체를 수정할 때 `kubectl apply` 대신 '명령형 접근법' (`create`, `edit`, `scale` 등)'을 이용할 경우, **최신 적용 설정(2번)**은 업데이트 되지 않고 **활성 객체 설정(3번)** 부분만 바뀌게 된다. 선언형 관리 환경에서 명령형 커맨드를 잘못 혼용할 경우 **원본 객체 구성 파일(1번)**과 **최신 적용 설정(2번)**엔 누락되어 있는 내용이 **활성 객체 설정(3번)**에는 버젓이 포함된 채로 돌아갈 수도 있는 것이다.

예시로 살펴보는 `kubectl apply`의 동작 방식

위의 내용이 실제 환경에서 어떻게 적용되는지 [쿠버네티스 공식 문서](#)의 예시를 통해 살펴보자. 아래는 `nginx-deployment` 디플로이먼트의 **원본 구성 파일(1번)**이다.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

위의 디플로이먼트가 배포되었을 때 쿠버네티스 시스템 안에 생성되는 **활성 객체 설정(3번)**은 아래와 같다. `metadata.annotations` 아래에 **최신 적용 설정(2번)**이 JSON 포맷으로 함께 삽입되어 있다.

```
apiVersion: v1
kind: Deployment
```

```

metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
       "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
       "spec":{"minReadySeconds":5,"selector":{"matchLabels":{"app":nginx}}, "template":{"metadata":{"labels":{"app":"nginx"}}, "spec":{"containers":[{"image":"nginx:1.14.2","name":"nginx", "ports":[{"containerPort":80}]}]}}}
      ...
spec:
  minReadySeconds: 5
  selector:
    matchLabels:
      ...
      app: nginx
  template:
    metadata:
      ...
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:1.14.2
          ...
          name: nginx
      ports:
        - containerPort: 80
      ...
...

```

여기서 아래와 같은 작업을 수행한다고 가정해보자.

1. `kubectl scale deployment/nginx-deployment --replicas=2` 명령으로 레플리카 설정을 직접 추가한다.
2. 원본 구성 파일(1번)에 다음의 수정 사항을 반영하여 `kubectl apply`를 실행한다.
 1. `spec.containers`에 포함된 `image` 버전을 `nginx:1.20.2`로 변경

2. `spec.minReadySeconds` 항목을 삭제3. 1번에서 적용한 `replicas` 항목은 파일에 추가하지 않음

두 가지 변경 사항을 모두 적용한 뒤 쿠버네티스 메모리 상에 적용된 **활성 객체 설정(3번)**의 내용은 다음과 같다.

```
apiVersion: v1
kind: Deployment
metadata:
  annotations:
    # 2-1, 2-2 항목이 아래의 JSON 내용에도 반영되었음을 알 수 있다.
    # 반면 kubectl scale로 추가시킨 replicas 설정은 누락되어 있다.
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment",
       "metadata":{"annotations":{},"name":"nginx-deployment","namespace":"default"},
       "spec":{"selector":{"matchLabels":{"app":nginx}),"template":{"meta
       data":{"labels":{"app":nginx}}},
       "spec":{"containers":[{"image":nginx:1.20.2,"name":nginx,
       "ports":[{"containerPort":80}]}]}}}
    ...
spec:
  replicas: 2    # kubectl scale로 추가된 부분(1)
  # minReadySeconds 항목 삭제됨(2-2)
  selector:
    matchLabels:
      ...
      app: nginx
  template:
    metadata:
      ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.20.2    # nginx 이미지 버전 변경(2-1)
      ...
      name: nginx
    ports:
```

```
- containerPort: 80  
...  
...
```

- 원본 구성 파일(1번)에서 변경된 `nginx:1.20.2` 부분과 `minReadySeconds` 항목은 활성 객체 설정(3번)과 `last-applied-configuration`에 모두 올바르게 적용되었다.
- 그러나 `kubectl scale`로 추가된 레플리카 설정 항목은 활성 객체 설정(3번)의 `spec` 내부에만 명시될 뿐 `last-applied-configuration`에는 반영되지 않았다. 원본 구성 파일(1번)에는 기록된 적 없는 수정사항이기 때문이다. 이 경우, 관리자는 원본 YAML에 존재하지 않은 채로 활성 상태의 객체에 적용되어 있는 항목을 직접 찾아서 YAML 파일에 다시 반영시켜야 하는 번거로움을 감수해야 할 것이다.
- `kubectl edit`로 수정된 내용 역시 마찬가지의 난점을 가진다. 만약 YAML 파일로 생성한 요소를 이 방식으로 수정할 경우, 그 내용은 해당 요소가 동작하고 있는 쿠버네티스 시스템 메모리에만 반영될 뿐 기존의 YAML 파일에는 반영되지 않는다. 이는 추후 배포용 파일을 이용한 해당 요소의 버전 관리에 문제를 일으킬 수 있다.

선언형 관리 환경에서 유의할 점

1. `kubectl`에서 YAML 파일로 객체를 관리할 때에는 `create`나 `replace` 대신 `apply`를 사용한다.
2. `kubectl`에서 `create`, `edit` 또는 `scale` 같은 명령을 `apply`와 절대 혼용하지 않는다.

쿠버네티스는 "원하는 상태를 선언한다"는 선언적 구성을 설계 사상으로 가지고 있다. 클러스터와 하부 요소들의 세밀한 관리를 위해서는 선언형 접근법에 보다 익숙해져야 할 필요가 있으며, 이러한 환경에서 자칫 클러스터 구성 요소들의 변경 이력 추적 등에 문제의 소지를 만드는 일이 없도록 유의점을 반드시 숙지해야 할 것이다.

▼ CKA 시험을 위한 tip

- 시험적 관점에서 보면, 명령형 (Imperative) 접근법을 써서 최대한 시간을 절약 할 수 있다.
 - ex) 주어진 이미지로 포드나 배포를 생성해야 하는 경우
 - ex) 존재하는 object의 속성을 수정해야 할 때 : `kubectl edit` 명령 사용
- 복잡한 요구사항 (ex. 다중 컨테이너, 환경 변수 명령, 컨테이너 init 등)이 있다면, object configuration file을 통해서 object를 생성하는 것이 낫다. 이 경우에는, 내가 실수했을 때 file을 쉽게 update하고 다시 apply할 수 있다. 그리고 이 경우에는 `kubectl apply` 명령을 쓰는게 낫다.

- 아래 2개의 옵션을 숙지할 것!
 - `--dry-run` : 기본적으로 명령이 실행되자마자 리소스가 생성된다. 단순히 명령어를 테스트하려면 `--dry-run=client` 옵션을 사용한다. 이렇게 하면 리소스가 생성되지 않으면서, 리소스를 생성할 수 있는지 여부와 명령어가 맞는지 여부를 알 수 있다!
 - `-o yaml` : YAML 형식의 리소스 정의가 화면에 출력된다.

▼ 위 옵션을 활용한 예시

POD

NGINX Pod 생성

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

POD Manifest YAML file 을 생성 (-o yaml). Don't create it(--dry-run)

```
kubectl run nginx --image=nginx --dry-run=client -o yaml
```

Deployment

deployment 생성

```
kubectl create deployment --image=nginx nginx
```

Deployment YAML file 생성 (-o yaml). Don't create it(--dry-run)

```
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml
```

Generate Deployment with 4 Replicas

```
kubectl create deployment nginx --image=nginx --replicas=4
```

You can also scale a deployment using the `kubectl scale` command.

```
kubectl scale deployment nginx --replicas=4
```

Another way to do this is to save the YAML definition to a file and modify

```
kubectl create deployment nginx --image=nginx --dry-run=client -o yaml > nginx-deployment.yaml
```

이렇게 한 다음, 배포를 생성하기 전에 복제본 또는 다른 필드와 함께 YAML 파일을 업데이트할 수 있다.

Service

Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379

```
kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml
```

(이것은 자동으로 pod의 labels를 selectors로 사용한다)

Or

```
kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml
```

(이것은 pods의 labels를 selectors로 사용하지 않고, 대신 selectors를 app=redis로 가정한다. 우리는 selectors를 옵션으로 전달할 수 없다. 따라서 pod에 다른 labels set이 있는 경우 잘 작동하지 않는다. 따라서 service를 만들기 전에 파일을 생성하고 selectors를 수정하도록!)

Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:

```
kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml
```

(이것은 자동으로 pod의 labels를 selectors로 사용하지만 노드의 포트를 지정 할 수는 없다. definition file을 생성한 다음 pod로 service를 만들기 전에 node port를 수동으로 추가해야 한다.)

Or

```
kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml
```

(이것은 자동으로 pod의 labels를 selectors로 사용하지 않는다)

위의 두 명령어 모두 단점이 있다. 하나는 selector를 허용할 수 없지만, 다른 하나는 노드 포트를 허용할 수 없다. 강사는 첫번째 명령어, 즉 `kubectl expose` 명령어를 사용하는 것을 추천한다. 노드 포트를 지정해야 하는 경우, 동일한 명령어를 사용하여 정의 파일을 생성하고, 서비스를 생성하기 전에 노드 포트를 수동으로 입력하자.