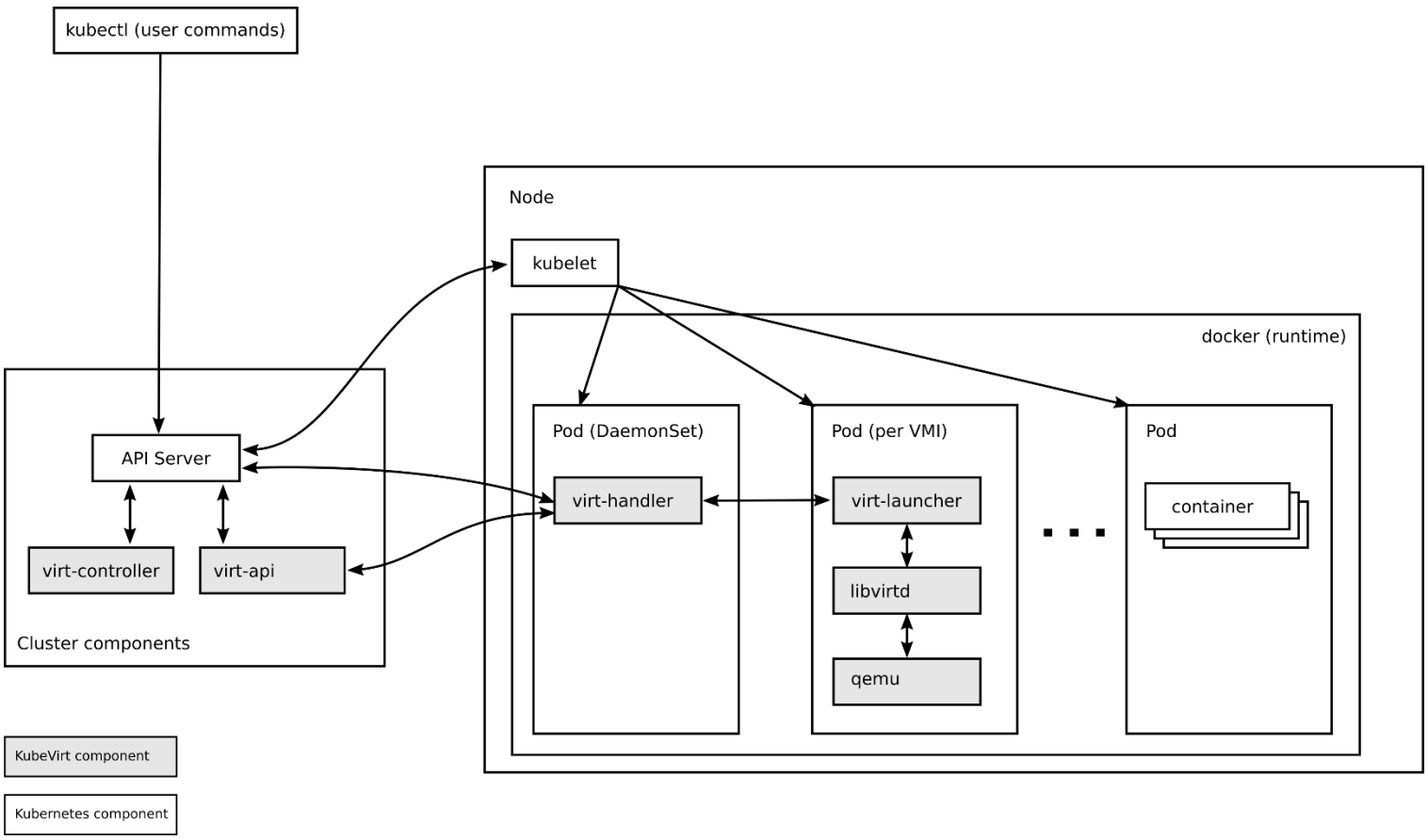
kube-virt소개

kube-virt는 쿠버네티스 기반으로 동작하는 가상머신 런타임이다. 자원 관리는 쿠버네티스에서 하지만, 실제로 가상머신은 libvirt에서 생성 및 관리를 하고 있다. 이러한 아키텍처를 통해서 빠르게 가상머신 생성 및 관리가 가능하다. 또한, 쿠버네티스 Pod와 함께 동작하기 때문에 컨테이너 플랫폼 및 가상머신 플랫폼을 하나로 통합 및 운영이 가능하다는 장점이 있다.

kube-virt 아키텍처

쿠버네티스에서 가상머신을 생성하기 위해서 추가적인 구성 요소가 필요하다. 먼저, 아래 그림을 확인한다.



kube-virt에서는 가상머신 생성 및 관리를 위해서 추가적 쿠버네티스에 컴포넌트를 추가한다. API요청 및 핸들링하기 위해서 virt-controller, virt-api서비스가 필요하다. 가상 머신을 Pod기반으로 생성하기 위해서 virt-launcher를 통해서 libvirt를 통해서 가상머신 생성을 한다. 그 이외 나머지 부분은 기존 쿠버네티스와 동일한 구조를 가지고 있다.

인프라 설치 및 구성

kubernetes/kube-virt 설치

kube-virt를 사용하기 위해서 쿠버네티스가 먼저 설치가 되어야 한다. 다만, kube-virt를 사용하기 위해서는 런타임을 둘 중 하나를 선택한다.

1. cri-o
2. containerd

현재 cri-docker는 kube-virt를 지원하지 않는다. 설치 시 위 두 가지 런타임으로 설치 및 구성한다. 수동으로 설치 시, 아래와 같이 진행한다.

```
node1/node2]# cat <<EOF> /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/v1.27/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.27/rpm/repodata/repomd.xml.key
# exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
EOF

node1/node2]# dnf install --disableexcludes=kubernetes kubectl kubeadm kubelet -y
node1/node2]# setenforce 0
node1/node2]# sed -i 's/SELINUX=enforcing/SELINUX=permissive/g' /etc/selinux/config> permissive
node1/node2]# systemctl stop firewalld && systemctl disable firewalld
node1/node2]# swapoff -a
node1/node2]# sed -i -e '/\dev\/mapper\/rl-swap/d' /etc/fstab
node1/node2]# cat <<EOF>> /etc/hosts
192.168.10.10 node1.example.com node1
192.168.10.10 node2.example.com node1
EOF
node1]# systemctl status kubelet
node1]# systemctl enable --now kubelet
node1/node2]# cat <<EOF | tee /etc/yum.repos.d/cri-o.repo
[cri-o]
name=CRI-O
baseurl=https://pkgs.k8s.io/addons:/cri-o:/prerelease:/main/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/addons:/cri-o:/prerelease:/main/rpm/repodata/repomd.xml.key
EOF
node1/node2]# dnf install conntrack container-selinux ebtables ethtool iptables socat -y
node1/node2]# dnf install cri-o -y
node1/node2]# systemctl enable --now crio
node1/node2]# modprobe br_netfilter
node1/node2]# modprobe overlay
node1/node2]# cat <<EOF> /etc/modules-load.d/k8s-modules.conf
```

```
br_netfilter
overlay
EOF
node1/node2]# cat <<EOF> /etc/sysctl.d/k8s-mod.conf
net.bridge.bridge-nf-call-iptables=1
net.ipv4.ip_forward=1
net.bridge.bridge-nf-call-ip6tables=1
EOF
node1/node2]# sysctl --system -p
node1]# kubeadm init --apiserver-advertise-address=192.168.10.10 --pod-network-cidr=192.168.0.0/16 --service-cidr=10.90.0.0/16
```

설치가 완료가 되면, 가상머신을 종료 후, nested기능을 활성화한다.

```
Set-VMProcessor -VMName <NODE_NAME> -ExposeVirtualizationExtensions $true
Start-VM <NODE_NAME>
```

문제없이, nested가 활성화가 되면, 가상머신을 다시 시작 후, kube-virt설치를 계속 진행한다.

```
node1/node2]# dnf install "Virtualization Host" -y
node1/node2]# virt-host-validate qemu
node1/node2]# virt-host-validate

node1]# export RELEASE=$(curl https://storage.googleapis.com/kubevirt-prow/release/kubevirt/kubevirt/stable.txt)
node1]# kubectl apply -f https://github.com/kubevirt/kubevirt/releases/download/${RELEASE}/kubevirt-operator.yaml
node1]# kubectl apply -f https://github.com/kubevirt/kubevirt/releases/download/${RELEASE}/kubevirt-cr.yaml
node1]# kubectl -n kubevirt wait kv kubevirt --for condition=Available
node1]# export VERSION=$(curl https://storage.googleapis.com/kubevirt-prow/release/kubevirt/kubevirt/stable.txt)
node1]# wget https://github.com/kubevirt/kubevirt/releases/download/${VERSION}/virtctl-${VERSION}-linux-amd64
```

설치가 완료가 되면, 다음과 같이 화면에 출력이 되는지 확인한다.

```
# kubectl get pod -n kubevirt --output=name
pod/virt-api-5bd5cb776d-4tkbg
pod/virt-controller-6f459b6bb4-hdc24
```

```
pod/virt-controller-6f459b6bb4-m5dps
pod/virt-handler-bzvzd
pod/virt-operator-5dcdbd8997-57ln7
pod/virt-operator-5dcdbd8997-cwc2g
```

NFS4 SERVER 구성

kubevirt에서는 CDI 및 스토리지를 사용하기 위해서는 NFSv4기반으로 저장소를 구성해야 한다. 그 이외 버전에 대해서는 제공하지 않는다.

```
# dnf install nfs-utils -y
# systemctl enable --now nfs-server
# mkdir -p /opt/nfs/
# cat <<EOF>> /etc/exportfs
/opt/nfs      *(rw,sync,no_root_squash,insecure,no_subtree_check,nohide)
EOF
# exportfs -avrs
# showmount -e node1.example.com
# mount -t nfs4 172.30.152.196:/opt/nfs /mnt/nfs4/
# mount | grep nfs4
172.30.152.196:/opt/nfs on /mnt/nfs4 type nfs4
(rw,relatime,vers=4.2,rsize=1048576,wsize=1048576,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=sys,cli
entaddr=172.30.152.196,local_lock=none,addr=172.30.152.196)
```

StorageClass구성

CDI를 사용하기 위해서 SC를 구성한다. 구성 시, 기본값으로 NFS을 사용하도록 한다. 먼저, NFS-CSI드라이버를 구성한다.

```
# curl -skSL https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/v4.9.0/deploy/install-driver.sh | bash
-s v4.9.0 --
```

설치가 완료가 되면, 다음 명령어 확인한다.

```
# kubectl -n kube-system get pod -o wide -l app=csi-nfs-controller
# kubectl -n kube-system get pod -o wide -l app=csi-nfs-node
```

SC POD가 생성이 완료가 되면, SC를 구성한다. 아래와 같이 YAML파일을 작성한다.

```
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
```

```
metadata:
   name: nfs-csi
   annotations:
      storageclass.kubernetes.io/is-default-class: "true"
allowVolumeExpansion: true
provisioner: nfs.csi.k8s.io
parameters:
   server: 192.168.10.1
   share: /opt/nfs
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
   - nfsvers=4.2
```

작성이 완료가 되면 적용 및 확인한다.

```
# kubectl get sc
# kubectl get sc -oname
storageclass.storage.k8s.io/nfs-csi
```

kubevirt-CDI 설치

가상머신에 영구적인 디스크를 제공하기 위해서 CDI(Containerized-Data-Importer)를 통해서 디스크를 제공한다.

```
node1]# export VERSION=$(curl -s https://api.github.com/repos/kubevirt/containerized-data-
importer/releases/latest | grep '"tag_name":' | sed -E 's/.*"([^"]+)".*/\1/')
node1]# kubectl create -f https://github.com/kubevirt/containerized-data-
importer/releases/download/$VERSION/cdi-operator.yaml
node1]# kubectl create -f https://github.com/kubevirt/containerized-data-
importer/releases/download/$VERSION/cdi-cr.yaml
node1]# kubectl get pod -n cdi -o name
pod/cdi-apiserver-7dd8f6657f-tgqpp
pod/cdi-deployment-54b55d677b-kzdc2
pod/cdi-operator-65665cd8bc-plgqr
pod/cdi-uploadproxy-8576ddc5cf-zsvp9
```

Prometheus 설치

Prometheus는 helmchart으로 설치하지 않고, kubectl로 설치를 진행한다. 아래와 같이 작업을 수행한다.

```
# git clone https://github.com/prometheus-operator/kube-prometheus.git
# cd /root/kube-prometheus
# kubectl apply -f manifests/
```

문제없이 설치가 진행이 되면, 다음과 같이 POD 및 자원 생성이 완료가 된다.

```
# kubectl get all -n monitoring
```

kubevirt-manager 설치

자원 관리를 위한 UI는 아래 명령어로 설치가 가능하다. 아래는 번들 패키지로 설치하는 방법이다.

```
# kubectl apply -f https://raw.githubusercontent.com/kubevirt-manager/kubevirt-manager/main/kubernetes/bundled.yaml
```

다만, 올바르게 사용하기 위해서 Prometheus 설치가 되어 있어야 한다.

번들이 아닌, 각 오퍼레이터 별로 설치를 원하는 경우 아래와 같이 설치를 진행한다.

```
# export VERSION=$(basename $(curl -s -w %{redirect_url} https://github.com/kubevirt/containerized-data-importer/releases/latest))
# kubectl create -f https://github.com/kubevirt/containerized-data-importer/releases/download/$VERSION/cdi-operator.yaml
# kubectl create -f https://github.com/kubevirt/containerized-data-importer/releases/download/$VERSION/cdi-cr.yaml
```

구성이 완료가 되면, 다음 명령어로 CDI확인이 가능하다.

```
# kubectl get cdi cdi -n cdi
NAME     AGE         PHASE
cdi      2d18h     Deployed
```

Prometheus와 연동을 하기 위해서 ConfigMap자원을 구성한다. 파일 이름은 kubevirt-prometheus.yaml으로 작성한다.

```
---
apiVersion: v1
kind: ConfigMap
metadata:
   name: prometheus-config
   namespace: kubevirt-manager
```

```
    labels:
      app: kubevirt-manager
      kubevirt-manager.io/version: 1.4.1
      kubevirt-manager.io/managed: "true"
data:
  prometheus.conf: |
      location /api {
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_set_header Authorization "";
            proxy_pass_request_body on;
            proxy_pass_request_headers on;
            client_max_body_size 5g;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
            proxy_pass http://prometheus-k8s.monitoring.svc:9090;
      }
```

kubevirt-manager는 Nginx기반으로 접근 관리를 하기 때문에, 아래와 같이 HTTP인증을 ConfigMap으로 구성 및 생성한다. 파일 이름은 kubevirt-nignx.yaml으로 작성한다.

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: auth-config
  namespace: kubevirt-manager
  labels:
    app: kubevirt-manager
    kubevirt-manager.io/version: 1.4.1
    kubevirt-manager.io/managed: "true"
data:
  basicauth.conf: |
    auth_basic "Restricted Content";
    auth_basic_user_file /etc/nginx/secret.d/.htpasswd;
```

구성이 완료가 되면, nginx에 접근할 사용자 계정을 Secret으로 생성한다. 기본 값은 admin/admin으로 되어 있다.

```
---
apiVersion: v1
kind: Secret
metadata:
   name: auth-secret
   namespace: kubevirt-manager
   labels:
      app: kubevirt-manager
      kubevirt-manager.io/version: 1.4.1
      kubevirt-manager.io/managed: "true"
data:
   .htpasswd: YWRtaW46JGFwcjEkMk53ckdFZVkkZmtPZkZldVR4Rm5pZzBsc2NaV3c1MQo=
```

위와 같이 구성이 되면 대시보드 사용이 가능하다.

준비

명령어는 기존의 kubectl과 관리용도 kubevirt 두 가지 명령어를 사용하여 생성 및 관리한다. 실질적인 자원 생성은 여전히 kubectl를 통해서 구성한다. 가상머신을 생성하기 위해서 다음과 같이 명령어를 실행한다.

```
# kubectl create -f kube-instance.yaml
```

생성이 완료가 되면, 기존과 동일하게 kubectl명령어로 확인 및 조회가 가능하다.

```
# kubectl get vm -oname
virtualmachine.kubevirt.io/kube-instance
```

여기까지 준비가 되었으면 가상머신 테스트 준비가 완료가 되었다. kubevirt명령어가 생각보다 타이핑 치기에 적절하지(?) 않는 위치에 배열되어 있어서 alias를 통해서 kv로 설정 후 사용하도록 한다.

```
# alias kv='kubevirt'
```

또한, 자동 완성 기능을 위해서 다음과 같이 명령어를 실행한다.

```
# kubevirt completion bash > /etc/profile.d/kubevirt.sh
# source /etc/profile
```

여기까지 하면, 사용할 준비가 완료가 되었다. 가상머신 구성 및 이미지 배포를 위해서 몇 가지 도구가 필요하다.

kubevirt YAML문법

기본 자원 생성 및 관리하기 위해서 아래와 같이 YAML작성하여 가상머신을 생성한다. 가상머신 이름 및 파일은 kube-instance.yaml파일로 작성한다.

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: kube-instance
spec:
  running: false
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: testvm
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
        interfaces:
        - name: default
          masquerade: {}
      resources:
        requests:
          memory: 64M
  networks:
  - name: default
    pod: {}
  volumes:
    - name: containerdisk
      containerDisk:
        image: quay.io/kubevirt/cirros-container-disk-demo
    - name: cloudinitdisk
      cloudInitNoCloud:
        userDataBase64: SGkuXG4=
```

가상머신을 생성하기 위해서 쿠버네티스와 동일하게 API를 호출해야 한다. kube-virt는 "VirtualMachine"으로 자원이 분류가 되어있으며, API는 kubevirt.io/v1으로 호출한다. 그 이외 나머지는 메타정보로 생성이 될 네임스페이스 위치 및 자원의 이름을 명시한다.

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: kube-instance
```

기존 쿠버네티스와 동일하게 가상머신 생성은 "**spec:"**통해서 구성 및 생성한다. 다만, 버전이 변경이 되면서 몇몇 사라지는 부분이 있다. 아래 설정 중, "size", "domain"은 레이블을 통해서 template에서 사용할 selector로 설정한다.

```
spec:
  running: false
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: testvm
```

"spec:"이후부터는 실제로 가상머신이 생성이 되는 영역이다. 컨테이너와 많이 다른 부분은 바로, 디스크 유형 및 버스 형식이다. 네트워크 인터페이스 또한, 컨테이너와 다르게 사용자가 직접 유형을 구성해야 한다.

```
spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            masquerade: {}
```

가상머신을 생성하기 위해서 CPU 및 MEMORY를 설정해야 한다. CPU는 기본값으로 vCPU 1를 가지고 있으며, 메모리 크기는 사용자가 명시를 해야한다.

```
resources:
        requests:
          memory: 64M
          cpu: 2
```

가상머신에서 사용할 디스크 이미지 및 유형을 아래처럼 명시한다. 기존에 사용하던 PV/PVC 및 StorageClass 도 그대로 활용 및 사용이 가능하다.

```
volumes:
      - name: containerdisk
        containerDisk:
          image: quay.io/kubevirt/cirros-container-disk-demo
      - name: cloudinitdisk
        cloudInitNoCloud:
          userDataBase64: SGkuXG4=
```

kube-virt자원

Kube-virt자원은 크게 다음과 같이 나누어진다.

1. POD
2. Network
3. Volumes
4. Console

POD는 기존 쿠버네티스에서 사용하던 POD이며, 가상머신을 생성 시, POD기반으로 가상머신을 구성한다. 쿠버네티스에서 제공하는 백엔드는 다음과 같이 구성이 가능하다.

이름	설명
POD	기존 쿠버네티스에서 사용하는 POD네트워크 기반으로 구성한다.
Multus	다중 네트워크를 구성하여 POD에 구성한다.

사용자가 가상머신에 접근하기 위한 프론트 엔드 기능은 다음과 같다.

이름	설명
bridge	리눅스 브릿지 기반으로 가상머신 네트워크를 구성한다.
slirp	slirp라이브러리 기반 네트워크로 가상머신을 구성한다.
sr-iov	SR-IOV기반으로 직접 인터페이스 가상머신을 연결한다.
masquerade	NAT네트워크 기반으로 구성한다.

가상머신을 구성하기 위해서 이미지는 기존과 동일하게 레지스트리 서버를 통해서 사용자가 구성한 이미지를 업로드 후, 가상머신 구성으로 사용이 가능하다. 가상머신에서 사용하는 디스크는 기존의 PV/PVC를 통해서 가상머신에 제공한다. 디스크 자원은 기존에 사용하던 쿠버네티스 자원과 거의 동일하다.

이름	설명
Ephemeral Disk	임시 디스크. 말 그대로 임시적으로 사용하는 디스크. 일시적으로 사용하며, 영구적인 데이터 저장은 권장하지 않는다.
Container Disk	컨테이너 디스크 이미지 형태로 저장한다. 컨테이너 이미지를 내려받기 후,
Empty Disk	비어 있는 디스크를 생성하여 연결한다.
Host Disk	호스트의 영역을 가상머신과 공유해서 사용한다.
Config Map	기존 쿠버네티스와 동일하게 설정 및 덜 민감한 정보를 공유한다.
Secret	기존 쿠버네티스와 동일하게 민감한 내용은 시크릿을 통해서 제공받는다.
Service Account	기존 쿠버네티스와 동일하게 대리인 계정을 통해서 권한을 할당 받는다.
Disk Sharing	공유 디스크를 생성하여 여러 가상머신들이 사용한다.
Disk Device Cache	QEMU에서 제공하는 DISK CACHE 모드를 직접 구성 및 설정한다.

가상머신에 디스크를 붙이기 위해서는 기존과 비슷하게 PV/PVC혹은 StorageClass기반으로 구성 후 가상머신에게 전달하면 된다.

kube-virt에서 콘솔에 접근하기 위해서 두 가지 방식을 지원하고 있다.

이름	설명
VNC	VNC기반으로 가상머신 디스플레이를 제공한다. 다만, 오디오 기능은 지원하지 않는다.
Serial	가상머신에 시리얼 인터페이스를 통해서 콘솔 접근 및 부팅 제어가 가능하다.

이미지 빌드

이미지 빌드 방식은 기존 가상머신과 다르게 생성 및 구성이 된다. 기본적으로 이미지는 컨테이너에서 사용하는 overlay2기반으로 구성된다. 그래서 kube-virt에서 사용하는 이미지는 좀 다르게 생성 및 관리가 된다. 일단, 기존의 이미지와 컨테이너 기반의 이미지로 가상머신 생성이 가능하다. 가상머신은 기존과 동일하게 QCOW, RAW형식으로 빌드한다. 이미지 빌드는 가급적이면 QCOW2기반으로 한다. 이미지 빌드 사용하는 도구는 이전과 동일하게 libguestfs도구를 사용하여 진행한다. 이미지 스크래치가 번거로우면 virt-builder를 통해서 진행하여도 된다. 혹은, 이전과 동일하게 oz 및 disk-imagebuilder를 통해서 생성 및 구성하여도 된다.

```
# dnf install libguestfs-tools guestfs-tools -y
# virt-builder --size 6G --format qcow2 --root-password password:centos -o /var/lib/libvirt/images/cent9s.qcow2 centosstream-9
```

이미 빌드 및 구성이 된 이미지는 다음과 같은 명령어로 패키지 관리가 가능하다.

```
# virt-customize -a /var/lib/libvirt/images/cent9s.qcow2 --install dpdk
```

생성 및 수정이 완료가 되면, 다음과 같이 이미지를 초기화 한다. 초기화는 선택적으로 혹은 전체적으로 적용이 가능하다.

```
# virt-sysprep -a /var/lib/libvirt/images/cent9s.qcow2 --operations machine-id,bash-history,logfiles,tmp-files,net-hostname,net-hwaddr
# virt-sysprep -a /var/lib/libvirt/images/cent9s.qcow2
```

위와 같은 단계로 이미지 빌드가 완료가 되면, 이미지를 컨테이너 서버에 업로드 해야 하는데, 가상머신 이미지는 컨테이너 이미지에 대한 메타정보를 가지고 있지 않기 때문에, 컨테이너 레이어로 한 번 더 랩핑(wrapping)를 해주어야 한다. 랩핑하는 방법은 매우 간단하다. 기존에 사용하던 데이터 파일을 컨테이너 이미지 안에 추가해주면 된다.

이미지 빌드를 위해서 가급적이면 buildah를 통해서 구성한다. 만약, docker-registry서버가 없는 경우, 레지스트리 서버를 실행 후 진행한다.

```
cp /var/lib/libvirt/images/cent9s.qcow2 .
cat << END > Containerfile
FROM scratch
ADD --chown=107:107 cent9s.qcow2 /disk/
END
dnf groupinstall "Virtualization Host" -y
dnf install buildah podman skopeo -y
podman run -d --name docker-registry -p 5000:5000 docker.io/library/registry:latest
buildah build -t localhost:5000/cent9s:latest .
buildah push --tls-verify=false localhost:5000/cent9s:latest
skopeo inspect --tls-verify=false docker://localhost:5000/cent9s:latest
"LayersData": [
        {
            "MIMEType": "application/vnd.oci.image.layer.v1.tar+gzip",
            "Digest": "sha256:c6fc61fe3568761e220e70c7416de4de92578e91fc1489ed8b44274ab75d51f8",
            "Size": 668179433,
            "Annotations": null
        }
    ]
```

위와 같이 화면에 데이터가 출력이 되면, 성공적으로 가상머신 이미지를 컨테이너로 변환이 완료가 되었다. 해당 이미지를 kubevirt에서 사용하기 위해서 아래와 같이 다시, 랩핑해서 확인을 해본다. 파일 이름은 centos9-stream.yaml으로 작성한다.

```
metadata:
  name: centos9-stream
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstance
spec:
  domain:
    resources:
      requests:
```

```
            memory: 1024M
        devices:
          disks:
          - name: containerdisk
            disk: {}
    volumes:
      - name: containerdisk
        containerDisk:
          image: localhost:5000/cent9s:latest
          path: /disk/cent9s.qcow2
```

완료가 되면 kubectl명령어로 가상머신을 생성한다.

```
# kubectl apply -f centos9-stream.yaml
```

구성이 완료가 되면 kubectl, virtctl명령어로 잘 동작하는지 확인한다.

```
# kubectl get pods
# virtctl start cent
```

Data Volume(디스크 이미지 및 디스크 PVC)

kubevirt에서는 Data Volume(DV)라는 개념을 PVC기반으로 제공하고 있다. 이는 CDI(Containerized Data Importer)기능도 사용하고 있으며, 이를 통해서 데이터를 import/upload/clone를 PVC를 통해서 제공한다.

기존에 사용하던 가상머신 이미지를 CDI에 저장에 가능하다. CDI에 저장하기 위해서 다음 명령어로 올바르게 구성이 되어 있는지 확인한다.

```
# kubectl get cdi cdi -n cdi
NAME     AGE       PHASE
cdi      3d1h      Deployed
# kubectl get pods -n cdi -oname
pod/cdi-apiserver-7dd8f6657f-tgqpp
pod/cdi-deployment-54b55d677b-kzdc2
pod/cdi-operator-65665cd8bc-plgqr
pod/cdi-uploadproxy-8576ddc5cf-zsvp9
```

CDI가 올바르게 구성이 되어 있으며, 외부에서 불러올 가상머신 이미지를 다음과 같이 작성하여 불러온다. 파일 이름은 fedora.yaml으로 작성한다.

```
cat <<EOF > fedora.yaml
```

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: "fedora"
spec:
  storage:
    resources:
      requests:
        storage: 5Gi
  source:
    http:
      url: "https://download.fedoraproject.org/pub/fedora/linux/releases/37/Cloud/x86_64/images/Fedora-Cloud-Base-37-1.7.x86_64.raw.xz"
EOF
# kubectl create -f fedora.yaml
```

만약, 블록(block)형태로 이미지를 저장하기 위해서 다음과 같이 YAML파일을 작성한다. 이름은 cirros.yaml이름으로 작성한다.

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: "cirros"
spec:
  source:
      http:
        url: "https://download.cirros-cloud.net/0.4.0/cirros-0.4.0-x86_64-disk.img"
        secretRef: ""
        certConfigMap: ""
  storage:
    volumeMode: Block
    resources:
      requests:
        storage: "64Mi"
```

만약, 위의 파일이 올바르게 내려받기가 되지 않으면, CoreDNS에서 forward내용을 다음처럼 수정한다.

```
# kubectedit cm -n kube-system coredns
forward . 8.8.8.8 8.8.4.4
```

```
# kubectl rollout restart -n kube-system deployment coredns
```

위의 YAML파일을 쿠버네티스에서 생성하면, 다음과 같이 작업이 수행이 된다.

```
# kubectl get pvc fedora
NAME      STATUS    VOLUME    CAPACITY    ACCESS MODES    STORAGECLASS    AGE
fedora    Pending                                         nfs-csi         2m16s
# kubectl get pod
importer-prime-d055bc95-b15d-40e3-9dcf-c6f2c7a44ca0
# kubectl logs importer-prime-d055bc95-b15d-40e3-9dcf-c6f2c7a44ca0
I0906 16:14:35.731280       1 prometheus.go:78] 42.61
I0906 16:14:36.731378       1 prometheus.go:78] 43.05
I0906 16:14:37.731864       1 prometheus.go:78] 43.64
I0906 16:14:38.732904       1 prometheus.go:78] 44.14
```

이미지를 PVC에 저장이 완료가 되면 다음 명령어로 가상머신 생성을 시도한다. 생성할 가상머신을 instance-fedora.yaml이름으로 생성한다.

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  creationTimestamp:
  generation: 1
  labels:
    kubevirt.io/os: linux
  name: fedora
spec:
  running: true
  template:
    metadata:
      creationTimestamp: null
      labels:
        kubevirt.io/domain: fedora
    spec:
      domain:
        cpu:
          cores: 2
        devices:
          disks:
```

```
        - disk:
            bus: virtio
          name: disk0
        - cdrom:
            bus: sata
            readonly: true
          name: cloudinitdisk
      machine:
        type: q35
      resources:
        requests:
          memory: 1024M
  volumes:
  - name: disk0
    persistentVolumeClaim:
      claimName: fedora
  - cloudInitNoCloud:
      userData: |
        #cloud-config
        hostname: fedora.example.com
        ssh_pwauth: True
        disable_root: false
        ssh_authorized_keys:
        - ssh-rsa YOUR_SSH_PUB_KEY_HERE
    name: cloudinitdisk
```

cloud-init에서 사용할 ssh-rsa키를 생성 후, YAML파일에 반영한다.

```
# ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
# PUBKEY=`cat ~/.ssh/id_rsa.pub`
# sed -i "s%ssh-rsa.*%$PUBKEY%" instance-fedora.yaml
```

위의 모든 작업이 완료가 되면, 아래 명령어로 가상머신을 생성한다.

```
# kubectl create -f instance-fedora.yaml
# kubectl get vmi,vm
```

POD와 가상머신

kubevirt에서 POD와 가상머신 차이점 및 조건은 다음과 같다.

이름	설명
Virtual Machine Custom Resource (VM CR)	This resource is created to define and configure a VM. It includes specifications like the VM's hardware resources, storage, network, and the desired image to run.
Virtual Machine Controller	The controller watches for changes to the VM CR and takes actions to ensure the VM's desired state is achieved. It interacts with the hypervisor or virtualization technology (such as KVM) to create and manage the actual VM.
Pods	While VMs are not directly encapsulated within pods, pods are used as a deployment mechanism for the underlying components required to run VMs. For example, each VM in Kubevirt typically has a corresponding pod that contains the required resources, such as the QEMU emulator and other necessary software.

기본적으로 VM혹은 VMI는 생성 시, POD네트워크를 통해서 생성되지 않는다. 가상머신 생성은 virt-handler를 통해서 생성 및 구성한다. 이 구성원은 기본적으로 가상머신에 대한 destroying, pausing, freezing, creating와 같은 라이프사이클을 담당한다. 이 기능은 오픈스택의 nova-compute와 흡사하며, 차이점은 각 VM를 kubernetes의 POD 내부에서 동작하며, 이 부분은 kubelet를 통해서 관리하게 된다. 위의 내용을 정리하면 다음과 같다.

1. virt-handler는 가상머신을 생성하며, 이는 kubelet에서 POD를 올바르게 생성 후 구성된다.
2. 가상머신 제거는 virt-handler를 통해서 제거가 먼저 되며, 그 이후 kubelet를 통해서 POD를 제거한다.

가상머신이 생성이 되면, 다음과 같이 프로그램 콜을 호출한다.

```
Run                                                    // cmd/virt-handler/virt-handler.go
  |-vmController := NewController()
  |-vmController.Run()
      |-Run()                                          // pkg/virt-handler/vm.go
          |-go c.deviceManagerController.Run()
          |
          |-for domain in c.domainInformer.GetStore().List() {
          |       d := domain.(*api.Domain)
          |       vmiRef := v1.NewVMIReferenceWithUUID(...)
          |       key := controller.VirtualMachineInstanceKey(vmiRef)
          |
          |       exists := c.vmiSourceInformer.GetStore().GetByKey(key)
          |       if !exists
          |             c.Queue.Add(key)
```

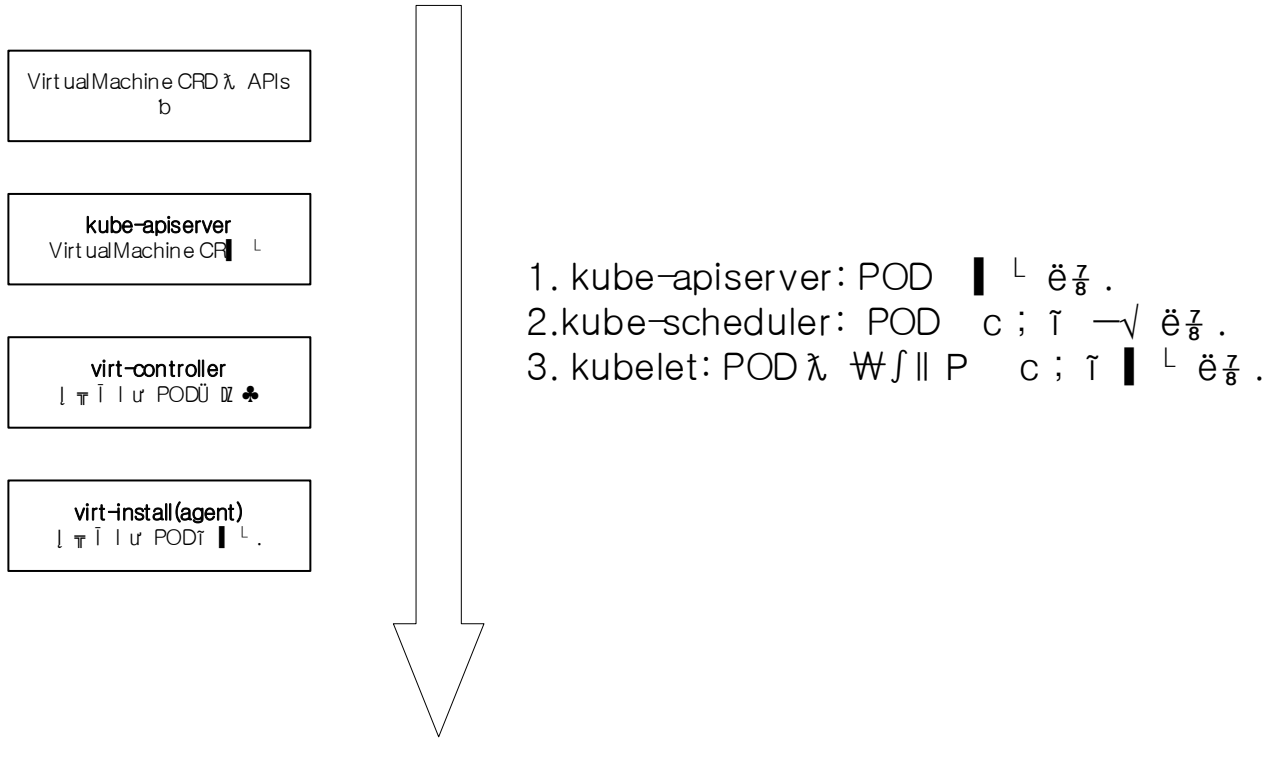
```
            |-}
            |
            |-for i := 0; i < threadiness; i++ // 10 goroutine by default
                    go c.runWorker
                        /
         /----------/
        /
runWorker
   |-for c.Execute() {
            |-key := c.Queue.Get()
            |-c.execute(key) // handle VM changes
                    |-vmi, vmiExists := d.getVMIFromCache(key)
                    |-domain, domainExists, domainCachedUID := d.getDomainFromCache(key)
                    |-if !vmiExists && string(domainCachedUID) != ""
                    |       vmi.UID = domainCachedUID
                    |-if string(vmi.UID) == "" {
                    |       uid := virtcache.LastKnownUIDFromGhostRecordCache(key)
                    |       if uid != "" {
                    |             vmi.UID = uid
                    |       } else { // legacy support, attempt to find UID from watchdog file it exists.
                    |             uid := watchdog.WatchdogFileGetUID(d.virtShareDir, vmi)
                    |             if uid != ""
                    |                   vmi.UID = types.UID(uid)
                    |       }
                    |-}
                    |-return d.defaultExecute(key, vmi, vmiExists, domain, domainExists)
    }
```

위의 내용을 정리하면, 가상머신 생성은 다음과 같이 진행이 된다.

1. 장치와 관련된 컨트롤러가 올바르게 동작하는지 확인한다.
2. 모든 노드에서 동작하는 가상머신을 확인하며, 필요에 따라서 자원을 정리한다.
3. 가상머신을 생성하기 위해서 kubevirt 루틴을 수행한다.

이러한 이유로 kube-controller, handler가 올바르게 동작 해야지 가상머신이 구성 및 생성이 된다. 또한, VMI에서 사용하는 가상머신은 Pod기반으로 생성이 되지만, 설정에 따라서 Pod 네트워크 혹은 VXLAN네트워크를 사용한다. 이러한, 기능을 사용하기 위해서 Multus-cni나 kubernetes-ovn이 필요하다.

위의 내용을 정리하여 그림으로 그리면 다음과 같이 진행이 된다.



kube-apiserver에서 VirtualMachine CR 생성

kubevirt는 VirtualMachine이라는 CR를 사용하며, 이를 통해서 가상머신 사양을 설정 및 구성한다. 예를 들어서 CPU/MEMORY/NETWORK 그리고 디스크와 같은 내용을 구성한다.

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: cirros
spec:
  running: true
  template:
    metadata:
      annotations:
        kubevirt.io/keep-launcher-alive-after-failure: "true"
    spec:
      nodeSelector:
        kubevirt.io/schedulable: "true"
      architecture: amd64
      domain:
        clock:
          timer:
            hpet:
              present: false
            hyperv: {}
```

kubevirt도 container와 동일하게 template필드가 필요하다. 없는 경우, vm자원에서 확인이 어렵고, vmi에서 확인이 가능하다.

running:은 생성 후 VM의 동작 상태이다.

가상머신 생성 시, 특정 노드의 레이블 혹은 스케줄링이 가능한 노드에만 생성.

"architecture:"는 서버에서 사용하는 CPU명령어. 일반적으로 64비트는 AMD를 사용한다. 아래 "domain: { clock: }"은 CPU관련된 Flags이다. 이 내용들은 필요에 따라서 API를 참조하여 호출한다.

```
      pit:
        tickPolicy: delay
      rtc:
        tickPolicy: catchup
    utc: {}
  cpu:
    cores: 1
  resources:
    requests:
      memory: 512M
  machine:
    type: q35
  devices:
    interfaces:
    - bridge: {}
      name: default
    disks:
    - disk:
        bus: virtio
      name: disk0
    - disk:
        bus: virtio
      name: emptydisk
    - disk:
        bus: cdrom
      name: cloudinitdisk
    features:
      acpi:
        enabled: true
    firmware:
      uuid: c3ecdb42-282e-44c3-8266-91b99ac91261
networks:
- name: default
  pod: {}
volumes:
- name: disk0
  persistentVolumeClaim:
    claimName: cirros
```

가상머신에서 사용할 네트워크 인터페이스 구성. 다양한 인터페이스를 사용하기 위해서는 Multus 및 OVS(Kubernetes-OVN)가 구성이 되어 있어야 한다.

추가 구성이 없으면 네트워크는 "default"로 구성이 된다. 우리가 사용하는 클러스터는 Multus가 구성이 안되어 있다.

디스크는 별 문제가 없으면 virtio 반-가상화 드라이버를 통해서 구성한다.

디스크는 별 문제가 없으면 virtio 반-가상화 드라이버를 통해서 구성한다.

오픈스택처럼 cloud-init를 통해서 가상머신 초기화 혹은 설정을 제공한다. 형식은 "cloudInitNoCloud:" 라는 지시자로 선언한다.

```
    - emptyDisk:
        capacity: 2Gi
      name: emptydisk
    - cloudInitNoCloud:
        userData: |-
          #cloud-config
          password: cirros
          chpasswd: { expire: False }
      name: cloudinitdisk
```

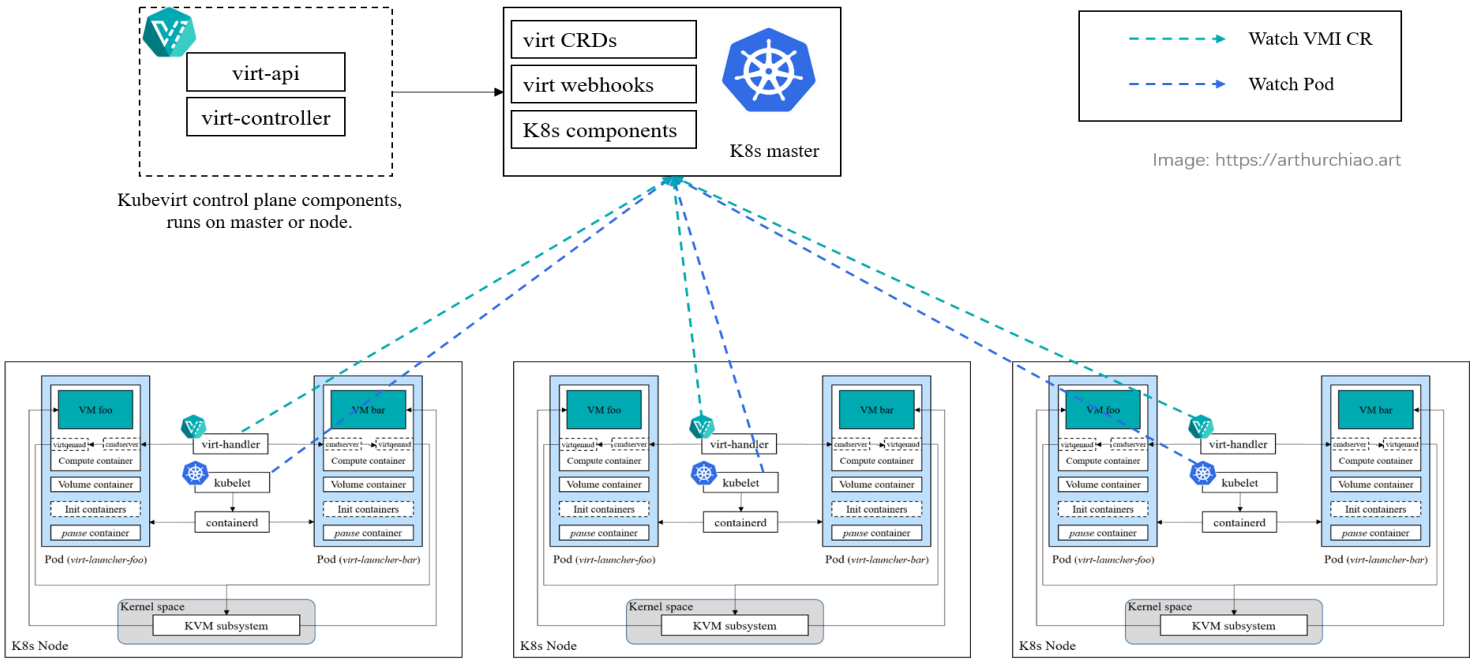
문제가 없으면, 위의 내용으로 가상머신이 구성이 되며, kubevirt console명령어로 POD에 생성된 가상머신에 접근이 가능하다.

```
# kubectl get vm
# virtctl console cirros
```

POD가 올바르게 구성이 되고 있으면, kube-scheduler에서 올바르게 관리가 잘 되고 있는지 VMI를 제거한다. 다음 명령어로 제거하여 POD에서 VM를 재구성하는지 확인한다. 앞서 이야기 하였지만, POD는 kube-scheduler가 담당하며, VM/VMI는 virt-controller가 담당한다.

```
# kubectl get vm,vmi
# kubectl delete vmi cirros
# kubectl get vmi
cirros   2s   Scheduled                        snk.example.com   False
# kubectl get vmi
cirros   19s   Running   192.168.10.67   snk.example.com   True
```

문제가 없으면, 가상머신은 다시 정상적으로 생성 및 실행이 된다. 마지막으로 POD 및 VM이 생성 시, 클러스터에 다음과 같이 이벤트가 발생한다. 아래는 쿠버네티스에서 가상머신을 생성 시, kubevirt-control-plane과 kubernetes control plane의 상호 관계를 나타낸다.



쿠버네티스에서 가상머신을 생성하는 경우, POD가 올바르게 생성이 되어야지, 그 다음 단계인 가상머신이 구성이 되기 때문에, 반드시 POD가 생성 및 구성이 되어야 하며, 그렇지 않는 경우 VM은 올바르게 생성되지 않는다.

kubelet-POD

가상머신이 생성이 되면, POD가 생성이 된다. 이 POD는 쿠버네티스에서 사용하는 기본 POD(Pause)와 동일하며, 이를 통해서 여러 컨테이너를 POD에 연결 및 구성한다. 이때 POD는 pause라는 애플리케이션을 통해서 네임스페이스(namespace)에서 생성되는 여러 컨테이너를 같이 묶어서 사용할 수 있도록 한다. 이러한 서비스 구성은 서비스 매시(Service Mesh)나 Side Car와 같은 구성을 쉽게 할 수 있도록 한다. 또한, 컨테이너에서 사용하는 네트워크는 POD를 통해서 제공이 된다. 조건을 정리하면 다음과 같이 된다.

- 컨테이너를 2개 이상을 생성하게 되면, 디렉터리 공유가 필요한 경우, POD를 통해서 디렉터리를 각 컨테이너에 공유한다.
- 특정 컨테이너에 볼륨(Volume) 제공하기 위해서 볼륨 컨테이너 역할.
- POD가 특정 노드에서 가상머신(VM)를 가지고 있음.

Pause Container(POD)

POD는 실제로는 pause라는 애플리케이션 기반으로 동작하는 컨테이너이다. 일반적인 컨테이너와 동일하게 가상머신은 pause에서 동작하게 된다. 이를 통해서 기존에 사용하던 kubernetes의 자원을 그대로 활용 및 공유가 가능하다.

```
# ps -fC virt-launcher
# ps -fC pause
```

인스턴스 구성

타입(instancetype)

가상머신을 생성 시, 인스턴스 형식을 지정할 수 있다. 이 기능은 오픈스택의 Flavor와 동일한 기능이라고 보면 된다. 생성 방법은 다음과 같다. API형식은 "instancetype.kubevirt.io"으로 구성이 된다. 간단하게 가상머신에서

사용할 인스턴스를 아래와 같이 생성한다. 파일 이름은 sample-instance-type.yaml으로 작성한다.

```
---
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachineInstancetype
metadata:
  name: sample-instance-type
spec:
  cpu:
    guest: 1
  memory:
    guest: 128Mi
```

작성이 완료가 되면, kubectl명령어로 자원을 생성한다. 생성 후 확인도 같이 한다.

```
# kubectl apply -f sample-instance-type.yaml
# kubectl get virtualmachineinstancetypes
sample-instance-type      25s
# kubectl describe virtualmachineinstancetypes
Spec:
Cpu:
Guest:  1
Memory:
Guest:  128Mi
```

형식에는 CPU 및 MEMORY의 값은 기본적으로 무조건 구성을 해야 되며, 나머지 부분은 선택적으로 구성이 가능하다.

선호(instance preference)

가상머신 생성 시, 어떠한 자원 형식을 사용할지 명시한다. 일반적으로 kubevirt가 지원하는 버스 드라이버는 다음과 같다.

이름	설명
lun	일반적인 LUN SCSI장치로 할당한다.
disk	PVC를 통해서 가상머신에 디스크를 할당한다. 버스 형식을 사용할 수 있으며 일반적으로 virtio으로 구성한다.
cdrom	시디롬 형식. 보통은 SATA형식으로 구성한다.
filesystem	virtiofs를 통해서 가상머신에 외부에서 생성한 파일 시스템을 사용할 수 있도록 한다. 다만, 이 형식은 마이그레이션은 지원하지 않는다.

이 교육에서 모든 부분에 대해서 다루지 않지만, 일반적으로 instance preference를 구성 시, disk를 virtio형식으로 많이 할당한다. 간단하게 가상머신 선호를 생성한다. 파일 이름은 sample-instance-prefer.yaml으로 작성한다.

```
---
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachinePreference
metadata:
   name: sample-instance-prefer
spec:
   devices:
      preferredDiskBus: virtio
      preferredInterfaceModel: virtio
```

위의 YAML으로 자원을 생성 확인 및 가상머신을 생성을 type, prefer를 통해서 생성을 시도한다.

```
# kubectl get VirtualMachinePreference
sample-instance-type
# kubectl describe VirtualMachinePreference sample-instance-prefer
Spec:
Devices:
Preferred Disk Bus: virtio
Preferred Interface Model: virtio
```

두 개의 자원 생성이 완료가 되면, 다음과 같은 명령어로 자원 생성이 가능하다. 파일 이름은 sample-preference-cirros.yaml으로 작성한다.

```
---
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
   name: example-preference-cirros
spec:
   instancetype:
      kind: VirtualMachineInstanceType
      name: sample-instance-type
   preference:
      kind: VirtualMachinePreference
      name: sample-instance-prefer
   running: false
   template:
```

```
spec:
  domain:
    devices:
      disks:
      - disk:
          bus: sata
        name: disk0
      - disk: {}
        name: cloudinitdisk
    resources: {}
  terminationGracePeriodSeconds: 0
  volumes:
  - name: disk0
    persistentVolumeClaim:
      claimName: cirros
  - cloudInitNoCloud:
      userData: |
        #!/bin/sh
        echo 'printed from cloud-init userdata'
    name: cloudinitdisk
```

가상머신이 생성이 되면, 이번에는 수동으로 가상머신을 시작해야 한다. 다음 명령어로 가상머신을 실행한다.

```
# kubectl apply -f sample-preference-cirros.yaml
# virtctl start example-preference-cirros
```

아마 가상머신이 올바르게 동작하지 않는다. 이유는 이전에 구성한 cirros가상머신이 사용하고 있기 때문에, 올바르게 동작하지 않는다. 이를 해결하기 위해서 기존에 사용하던 cirros가상머신을 제거 후 기다리면, 올바르게 가상머신 서비스가 시작이 된다.

```
# kubectl delete vm cirros
# kubectl get vm -w
```

가상머신 템플릿

아쉽게도 kubevirt에서는 템플릿 기능을 지원하지 않는다. 이 기능은 현재 오픈시프트에서 제공하고 있으며, 바닐라 버전에서는 현 시점에서는 아직 API가 존재하지 않는다. 자세한 내용은 아래 kubevirt문서를 참고한다.

https://kubevirt.io/user-guide/user_workloads/templates/#virtual-machine-templates

가상머신 풀(VirtualMachine Pool)

컨테이너 기반 POD에서는 ReplicaSet를 통해서 쉽게 구성이 가능하다. cirros기반으로 VM Pool를 생성 및 구성한다. 파일 이름은 cirros-pool.yaml이름으로 생성한다.

```
apiVersion: pool.kubevirt.io/v1alpha1
kind: VirtualMachinePool
metadata:
  name: cirros-pool
spec:
  replicas: 3
  selector:
    matchLabels:
      kubevirt.io/vmpool: cirros-pool
  virtualMachineTemplate:
    metadata:
      creationTimestamp: null
      labels:
        kubevirt.io/vmpool: cirros-pool
    spec:
      running: true
      template:
        metadata:
          creationTimestamp: null
          labels:
            kubevirt.io/vmpool: cirros-pool
        spec:
          domain:
            devices:
              disks:
              - disk:
                  bus: virtio
                name: containerdisk
            resources:
              requests:
                memory: 128Mi
          terminationGracePeriodSeconds: 0
          volumes:
          - containerDisk:
              image: kubevirt/cirros-container-disk-demo:latest
```

```
                name: containerdisk
```

YAML파일 작성 후, 가상머신을 kubectl명령어로 생성한다.

```
# kubectl get vm
# kubectl get VirtualMachinePool
NAME              DESIRED     CURRENT     READY     AGE
cirros-pool     3             3             3          84s
```

위와 같이 생성이 되었으면, 가상머신 Pool구성이 완료가 되었다. Pool자원은 쿠버네티스의 ReplicaSet, ReplicController와 동일하지 않기 때문에, 아래와 같이 조회하면 Pool자원은 위의 자원에서 조회가 되지 않는다.

```
# kubectl get rs
No resources found in default namespace.
# kubectl get rc
No resources found in default namespace.
```

구성이 된 Pool자원은 쿠버네티스의 HPA처럼, 스케일링이 가능하다.

```
# kubectl scale vmpool cirros-pool --replicas 5
# kubectl get vm
cirros-pool-0     5m32s     Running     True
cirros-pool-1     5m32s     Running     True
cirros-pool-2     5m32s     Running     True
cirros-pool-3     67s         Running     True
cirros-pool-4     67s         Running     True
```

만약, 특정 가상머신을 VM Pool에서 제거하기 위해서는 다음과 같이 patch명령어로 일시적으로 제거가 가능하다. 이 명령어, 불편하면 edit명령어로 수정이 가능하다.

```
# kubectl patch vm cirros-pool-0 --type merge --patch '{"metadata":{"ownerReferences":null}}'
```

HPA스케일링

쿠버네티스에서 지원하는 HPA기능을 kubevirt에서 똑같이 적용 및 사용이 가능하다. 사용 방법은 매우 간단하며, 다음과 같이 YAML파일 작성 및 구성한다. 파일 이름은 hpa-cirros-pool.yaml파일로 작성한다.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
   creationTimestamp: null
```

```
  name: cirros-pool
spec:
  maxReplicas: 10
  minReplicas: 3
  scaleTargetRef:
    apiVersion: pool.kubevirt.io/v1alpha1
    kind: VirtualMachinePool
    name: cirros-pool
  targetCPUUtilizationPercentage: 50
```

다만, 이 자원을 올바르게 사용하기 위해서 kubernetes metric 서비스 설치가 필요하다. 설치가 안되어 있는 경우, 아래와 같이 빠르게 설치한다.

```
# kubectl apply -f https://raw.githubusercontent.com/tangt64/training_memos/main/opensource-101/kubernetes-101/files/metrics.yaml
# kubectl get pod -n kube-system
# kubectl top node
# kubectl top pod
```

위의 서비스가 문제없이 동작하면 metrics서비스가 동작하면서 노드 및 Pod의 CPU 및 MEMORY정보 수집을 시작한다. 위의 서비스 적용 및 YAML 작성이 완료가 되면, 다음과 같이 적용 및 확인한다.

```
# kubectl apply -f hpa-cirros-pool.yaml
# kubectl get hpa -w
```

가상머신 서비스

가상머신은 기존 컨테이너와 동일하게 서비스를 통해서 외부에 노출한다. 방법은 두 가지가 있다.

1. kubectl명령어로 expose.
2. YAML기반으로 서비스 노출

어떠한 방법을 사용하여도 상관없지만, 일반적으로 YAML기반으로 작성 및 구성한다. 서비스 파일이름은 service-cirros-pool.yaml으로 작성한다. 첫번째 작성은 기본적으로 작성하는 ClusterIP로 구성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: cirros-pool-ssh-clusterip
spec:
  type: ClusterIP
  selector:
```

```
    kubevirt.io/vmpool: cirros-pool
  ports:
    - protocol: TCP
      port: 2222
      targetPort: 22
```

이번에는 NodePort로 작성한다. 서비스 형식만 변경하면 된다. 파일 이름은 service-cirros-pool-nodeport.yaml 으로 작성한다.

```
apiVersion: v1
kind: Service
metadata:
  name: cirros-pool-ssh-nodeport
spec:
  type: NodePort
  selector:
    kubevirt.io/vmpool: cirros-pool
  ports:
    - protocol: TCP
      port: 2222
      targetPort: 22
```

문제없이 NodePort가 구성이 되었으면 ssh명령어로 접근이 가능하다.

```
# kubectl get svc
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
cirros-pool-ssh-clusterip     ClusterIP   10.90.31.43     <none>        2222/TCP         2m9s
# ssh cirros@localhost -p30189
```