POD, Storage and Service

K8s part 2

POD

Pod คือหน่วยเล็กที่สุดที่สามารถสร้างและจัดการได้ใน Kubernetes โดยทำหน้าที่คล้ายกับ Virtual Machine ในแง่ของการรันแอปพลิเคชันที่เรากำหนดไว้

องค์ประกอบภายใน Pod

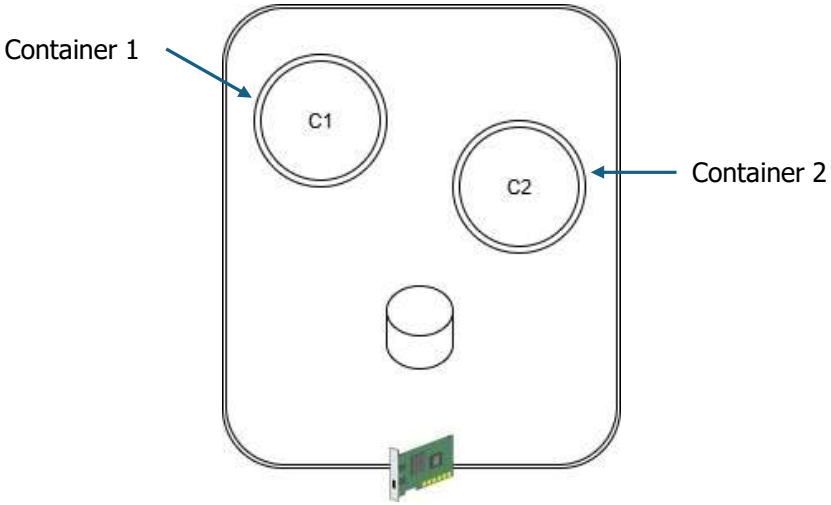
- ประกอบด้วย Container, Storage, และ Network
- เป็นพื้นที่ที่ Container สามารถทำงานร่วมกันได้อย่างใกล้ชิด

ความสัมพันธ์ระหว่าง Pod และ Container

- โดยทั่วไป 1 Pod จะมี 1 Container
- แต่สามารถรันหลาย Container ใน Pod เดียวกันได้ ซึ่งเรียกว่า Sidecar Pattern ใช้สำหรับกรณีที่ Container ต้องทำงานร่วมกัน เช่น logging, proxy, หรือ monitoring

การแชร์ Resource ภายใน Pod

- Storage Volume สำหรับการอ่าน/เขียนข้อมูลร่วมกัน
- Network Namespace ทำให้ Container ใช้ IP และพอร์ตร่วมกัน



Static POD

Pod ที่ถูกสร้างโดยตรงจากไฟล์ manifest ที่วางไว้ใน directory ที่กำหนด โดยไม่ผ่าน API Server หรือ Controller Manager โดย kubelet จะทำหน้าที่ monitor directory ที่กำหนดไว้ เมื่อพบไฟล์ manifest ใหม่หรือมีการเปลี่ยนแปลง kubelet จะสร้างหรืออัปเดต Pod โดยอัตโนมัติ Static Pod จะ รันตลอดเวลา ตราบใดที่ไฟล์ยังอยู่ใน directory นั้น

Path

Control Plan -> ไฟล์ manifest ของ Static Pod จะถูกวางไว้ที่ /etc/kubernetes/manifests/

Node -> การตั้งค่า directory ที่ kubelet ใช้ monitor จะอยู่ในไฟล์ /var/lib/kubelet/config.yaml -> staticPodPath

ใช้ใน environment ที่ต้องการความมั่นคงสูงและไม่พึ่งพา API

```
controlplane:~$ ls /etc/kubernetes/manifests/
etcd.yaml  kube-apiserver.yaml  kube-controller-manager.yaml  kube-scheduler.yaml
controlplane:~$
```

```
resolvConf: /run/systemd/resolve/resolv.conf
rotateCertificates: true
runtimeRequestTimeout: 0s
shutdownGracePeriod: 0s
shutdownGracePeriodCriticalPods: 0s
staticPodPath: /etc/kubernetes/manifests
streamingConnectionIdleTimeout: 0s
syncFrequency: 0s
volumeStatsAggPeriod: 0s
node01:~$
```

CMD

kubectl get po

kubectl run xxx --image=yyy

kubectl logs xxx

kubectl delete po xxx

```
controlplane:~$ kubectl get po -A
NAMESPACE           NAME                                       READY   STATUS    RESTARTS       AGE
kube-system         calico-kube-controllers-fdf5f5495-vxpw5    1/1     Running   2 (9m5s ago)   6d13h
kube-system         canal-4xxhw                                2/2     Running   2 (9m5s ago)   6d12h
```

```
controlplane:~$ k run nginx --image=nginx
pod/nginx created
```

```
controlplane:~$ kubectl logs nginx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2025/09/26 08:24:02 [notice] 1#1: using the "epoll" event method
2025/09/26 08:24:02 [notice] 1#1: nginx/1.29.1
2025/09/26 08:24:02 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14+deb12u1)
2025/09/26 08:24:02 [notice] 1#1: OS: Linux 6.8.0-51-generic
2025/09/26 08:24:02 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2025/09/26 08:24:02 [notice] 1#1: start worker processes
2025/09/26 08:24:02 [notice] 1#1: start worker process 30
```

```
controlplane:~$ kubectl delete po nginx
pod "nginx" deleted
controlplane:~$
```

CMD

kubectl describe po xxx

```
controlplane:~$ kubectl describe po nginx
Name:             nginx
Namespace:        default
Priority:         0
Service Account:  default
Node:             node01/172.30.2.2
Start Time:       Fri, 26 Sep 2025 08:32:15 +0000
Labels:           run=nginx
Annotations:      cni.projectcalico.org/containerID: aa0bed07071e6b0aa4edcdc773f1bc3f47aae2d368167303662dc01e2a3dbba6
                  cni.projectcalico.org/podIP: 192.168.1.5/32
                  cni.projectcalico.org/podIPs: 192.168.1.5/32
Status:           Running
IP:               192.168.1.5
IPs:
  IP:  192.168.1.5
Containers:
  nginx:
    Container ID:   containerd://6a2a0ba966799f8ced2187bbfd3e4781916a2f229dbd61c0a4227c1e47a8bd0e
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:d5f28ef21aabddd098f3dbc21fe5b7a7d7a184720bc07da0b6c9b9820e97f25e
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Fri, 26 Sep 2025 08:32:17 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-zr6bc (ro)
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
Volumes:
  kube-api-access-zr6bc:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    Optional:                false
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s

Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  11s   default-scheduler  Successfully assigned default/nginx to node01
  Normal  Pulling    10s   kubelet            Pulling image "nginx"
  Normal  Pulled     9s    kubelet            Successfully pulled image "nginx" in 791ms (791ms including waiting). Image size: 72319182 bytes.
  Normal  Created    9s    kubelet            Created container: nginx
  Normal  Started    9s    kubelet            Started container nginx
```

CMD

kubectl run xxx --image=yyy --dry-run=client –o yaml > xxx.yaml

ใช้สร้าง file template สำหรับสร้าง Pod โดยใช้ command

kubectl create -f xxx.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: xxx
  name: xxx
spec:
  containers:
  - image: yyy
    name: xxx
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

ReplicaSet

คือกลไกที่ใช้ควบคุมจำนวน Pod ที่ต้องการให้รันอยู่ในคลัสเตอร์ โดยสามารถกำหนดจำนวนสำเนาได้อย่างชัดเจนผ่าน

หากมีการลบ Pod ใดออกไป ระบบจะ สร้าง Pod ใหม่โดยอัตโนมัติ เพื่อคงจำนวนไว้ตามที่กำหนด

การสร้าง Pod ด้วย kubectl run หรือไฟล์ YAML จะสร้างเพียง 1 Pod เท่านั้น

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: demo-rs
  labels:
    app: web
    tier: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: web
        image: nginx:latest
```

CMD

kubectl get rs

kubectl describe rs xxx

kubectl delete rs xxx

kubectl scale rs xxx --replicas=y

DeamonSet

คือการสร้าง Pod ที่เหมือนกันไปทุก Node ใน Cluster ควบคุมโดย Control plan

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v5.0.1
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

Deployment

คือกลไกที่รวมเอา Pod และ ReplicaSet เข้าด้วยกัน เพื่อให้สามารถควบคุมการทำงานของแอปพลิเคชันได้อย่างมีประสิทธิภาพ

หน้าที่หลัก

- กำหนดจำนวน replica ที่ต้องการให้รันอยู่เสมอ
- ควบคุมการสร้างและอัปเดต Pod ผ่าน ReplicaSet
- มีการ monitor โดย Control Plane เพื่อให้สถานะของแอปเป็นไปตามที่กำหนด ไม่ว่าจะเกิดการลบ, crash หรือ node failure

รองรับการอัปเดตเวอร์ชันของแอปพลิเคชันผ่าน rolling update

- สามารถ rollback กลับไปยังเวอร์ชันก่อนหน้าได้อย่างปลอดภัย
- รองรับการ pause/resume การ deploy เพื่อควบคุมกระบวนการอัปเดต

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: web
  name: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: web
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

CMD

kubeclt get deploy

kubectl describe deploy xxx

kubectl delete deploy xxx

kubectl set image deploy/xxx containerName=image

kubeclt rollout history deploy xxx

ใช้ดูข้อมูลการ Deploy

kubectl rollout history deploy xxx --revision=

kubectl rollout undo deploy xxx

ใช้สั่ง restore การ Deploy

kubectl rollout undo deploy xxx --revision=

Storage

โดยปกติแล้ว Container ไม่สามารถเขียนไฟล์ลงในระบบไฟล์แบบถาวรได้ เพราะข้อมูลจะหายไปเมื่อ Container ถูกลบหรือรีสตาร์ต ดังนั้น จึงมี Volume เพื่อรองรับการอ่าน/เขียนข้อมูล

1. Volume Mount

คือการ ผูก Volume เข้ากับ Container

ทำให้ Container สามารถเข้าถึงข้อมูลที่อยู่ใน Volume ได้

ใช้สำหรับการอ่าน/เขียนไฟล์ภายใน หรือเชื่อมต่อกับข้อมูลจากภายนอก

2. Persistent Volume (PV) + Persistent Volume Claim (PVC)

เป็นกลไกที่ใช้สำหรับการจัดการ Volume แบบถาวร

PV: เป็น resource ที่จัดเตรียมพื้นที่เก็บข้อมูลไว้

PVC: เป็นการร้องขอใช้พื้นที่จาก PV ตามขนาดและ access mode ที่ต้องการ

สามารถใช้ร่วมกับ CSI (Container Storage Interface) เพื่อเชื่อมต่อกับ External Storage เช่น NFS, iSCSI, cloud disk

```yaml
volumeMounts:
  - name: mydata
    mountPath: /app/data
```

```yaml
apiVersion: v1
kind: PersistentVolume
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /mnt/data
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Volume Mount

1. emptyDir ใช้เพื่อต้องให้ Container สามารถเขียน file ภายในตัวมันเอง เมื่อ Pod หาย Data loss

2. hostPath ใช้เพื่อให้ Container สามารถอ่านเขียน file ร่วมกับ host เมื่อ Pod หาย Data ยังอยู่ที่ host

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox-ping
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["/bin/sh", "-c"]
    args:
      - ping 8.8.8.8 > /root/logs/ping.log
    volumeMounts:
    - name: log-volume
      mountPath: /root/logs
  volumes:
  - name: log-volume
    emptyDir: {}
  restartPolicy: Never
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox-ping
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["/bin/sh", "-c"]
    args:
      - ping 8.8.8.8 > /root/logs/ping.log
    volumeMounts:
    - name: log-volume
      mountPath: /root/logs
  volumes:
  - name: log-volume
    hostPath:
      path: /mnt/logs
      type: DirectoryOrCreate
  restartPolicy: Never
```

PV and PVC

- เป็น resource ระดับ cluster ที่ Kubernetes ใช้เพื่อจัดการ storage แบบถาวร
- PV คือ abstraction ที่แทน storage จริง เช่น NFS, iSCSI, cloud disk, local disk
- ถูกสร้างโดย admin หรือ provisioner เพื่อให้ pod ใช้งานได้ผ่าน PV

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath-logs
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /mnt/logs
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-hostpath-logs
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox-ping
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["/bin/sh", "-c"]
    args:
      - ping 8.8.8.8 > /root/logs/ping.log
    volumeMounts:
    - name: log-volume
      mountPath: /root/logs
  volumes:
  - name: log-volume
    persistentVolumeClaim:
      claimName: pvc-hostpath-logs
  restartPolicy: Never
```

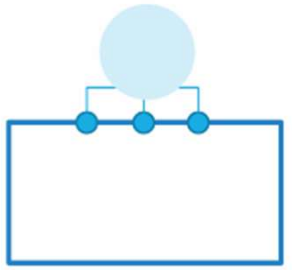
Storage Class

- เป็น resource ระดับ cluster ที่ Kubernetes ใช้เพื่อจัดการ storage แบบถาวร
- PV คือ abstraction ที่แทน storage จริง เช่น NFS, iSCSI, cloud disk, local disk
- ถูกสร้างโดย admin หรือ provisioner เพื่อให้ pod ใช้งานได้ผ่าน PV

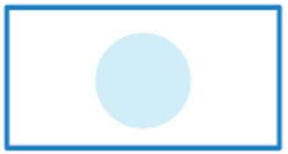
```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: delayed-volume-sc
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Service

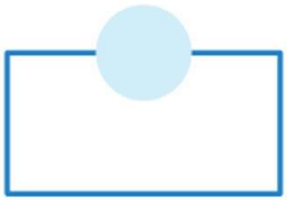
คือบริการ Network ที่ทำให้ Pod สื่อสารกันทั้งภายใน Cluster และนอก Cluster



LoadBalancer



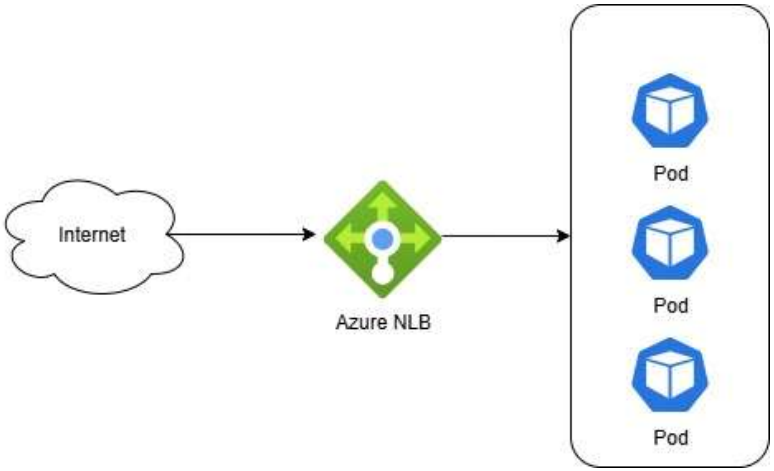
ClusterIP



NodePort

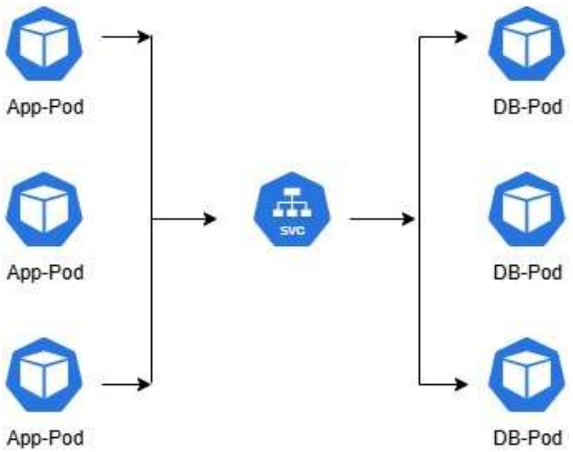
LoadBalance

- ทำหน้าที่เป็น gateway ระหว่างโลกภายนอกและคลัสเตอร์
- สร้าง External IP โดยอัตโนมัติผ่าน Cloud Provider เพื่อให้บริการสามารถเข้าถึงได้จากอินเทอร์เน็ต
- กระจาย traffic ไปยัง Pod ที่อยู่เบื้องหลังผ่าน internal load balancing
- ใช้ได้เฉพาะใน environment ที่มีการเชื่อมต่อกับ Cloud Provider เช่น AWS, Azure, GCP
- ต้องมีการตั้งค่า integration ระหว่าง Kubernetes กับ Cloud Provider เพื่อจัดการ provisioning ของ LoadBalancer



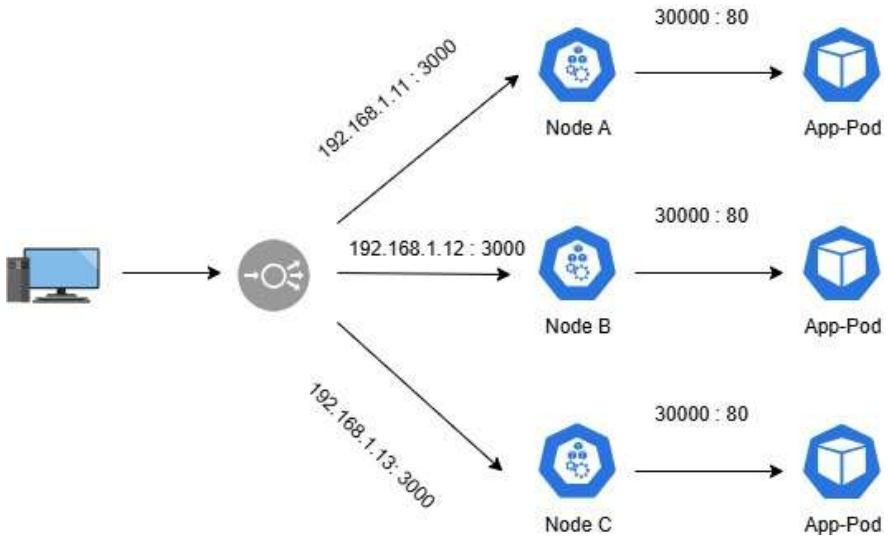
ClusterIP

- ทำหน้าที่เป็น Internal Load Balancer สำหรับการกระจาย traffic ไปยัง Pod ที่อยู่เบื้องหลัง
- ให้บริการผ่าน IP ภายในคลัสเตอร์ (ClusterIP) เท่านั้น
- ใช้สำหรับการสื่อสารระหว่าง Pod หรือระหว่าง Service ภายในคลัสเตอร์



NodePort

- ทำหน้าที่เป็น NAT device (PAT) ที่แปลงพอร์ตจาก Node ไปยังพอร์ตของ Pod
- ช่วยให้สามารถเข้าถึงบริการภายในคลัสเตอร์ผ่าน ได้จากภายนอกโดยใช้ NodeIP:NodePort
- พอร์ตที่สามารถกำหนดให้กับ NodePort อยู่ในช่วง 30000 ถึง 32767
- หากไม่กำหนดเอง Kubernetes จะสุ่มพอร์ตในช่วงนี้ให้อัตโนมัติ



External

- เชื่อมต่อกับ บริการภายนอก (เช่น database, API, หรือ SaaS) โดยใช้ชื่อ DNS
- ให้แอปในคลัสเตอร์เรียกใช้บริการภายนอกผ่านชื่อภายใน เช่น db.external.svc.cluster.local → db.home.local
- สร้าง abstraction layer เพื่อให้เปลี่ยนปลายทางได้ง่าย โดยไม่ต้องแก้โค้

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-external-db
spec:
  type: ExternalName
  externalName: db.example.com
```

CMD

kubeclt get svc

kubectl describe svc xxx

kubectl delete svc xxx

kubectl create nodeport xxx --tcp=port:target

kubectl create clusterip xxx --tcp=port:target

```
controlplane:~$ k get svc demo-svc
NAME       TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)           AGE
demo-svc   NodePort   10.97.52.19   <none>        30000:30659/TCP   13s
```

```
controlplane:~$ k describe svc demo-svc
Name:                     demo-svc
Namespace:                default
Labels:                   app=demo-svc
Annotations:              <none>
Selector:                 app=demo-app
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.97.52.19
IPs:                      10.97.52.19
Port:                     30000-80  30000/TCP
TargetPort:               80/TCP
NodePort:                 30000-80  30659/TCP
Endpoints:
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:                   <none>
```

```
controlplane:~$ k delete svc demo-svc
service "demo-svc" deleted
```

Demo services clusterip

```yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: demo-svc
  name: demo-svc
spec:
  ports:
  - name: 80-80
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: demo-app
  type: ClusterIP
status:
  loadBalancer: {}
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: demo-app
  name: demo-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo-app
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: demo-app
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

```
controlplane:~$ k describe svc demo-svc
Name:                     demo-svc
Namespace:                default
Labels:                   app=demo-svc
Annotations:              <none>
Selector:                 app=demo-app
Type:                     ClusterIP
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.101.204.114
IPs:                      10.101.204.114
Port:                     80-80  80/TCP
TargetPort:               80/TCP
Endpoints:                192.168.0.4:80,192.168.1.4:80
Session Affinity:         None
Internal Traffic Policy:  Cluster
Events:                   <none>
controlplane:~$ k get po -o wide
NAME                        READY   STATUS    RESTARTS   AGE     IP            NODE           NOMINATED NODE   READINESS GATES
demo-app-65bd44c89b-g88kb   1/1     Running   0          2m15s   192.168.1.4   node01         <none>           <none>
demo-app-65bd44c89b-mkt94   1/1     Running   0          2m15s   192.168.0.4   controlplane   <none>           <none>
```

Demo services nodeport

```yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: demo-svc
  name: demo-svc
spec:
  ports:
  - name: 30000-80
    port: 30000
    protocol: TCP
    targetPort: 80
  selector:
    app: demo-app
  type: NodePort
status:
  loadBalancer: {}
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: demo-app
  name: demo-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: demo-app
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: demo-app
    spec:
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

```
controlplane:~$ k describe svc demo-svc
Name:                     demo-svc
Namespace:                default
Labels:                   app=demo-svc
Annotations:              <none>
Selector:                 app=demo-app
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.103.36.205
IPs:                      10.103.36.205
Port:                     30000-80  30000/TCP
TargetPort:               80/TCP
NodePort:                 30000-80  30400/TCP
Endpoints:                192.168.1.4:80,192.168.0.4:80
Session Affinity:         None
External Traffic Policy:  Cluster
Internal Traffic Policy:  Cluster
Events:                   <none>
controlplane:~$ k get po -o wide
NAME                      READY   STATUS    RESTARTS   AGE   IP            NODE           NOMINATED NODE   READINESS GATES
demo-app-65bd44c89b-g88kb   1/1     Running   0          17m   192.168.1.4   node01         <none>           <none>
demo-app-65bd44c89b-mkt94   1/1     Running   0          17m   192.168.0.4   controlplane   <none>           <none>
```

Command and Arg

- สำหรับบาง Container อาจจำเป็นต้องกำหนดค่า Command และ Arguments (Args) เพื่อส่งพารามิเตอร์ที่ใช้ในการรันหรือเริ่มต้นการทำงาน เช่น การระบุ path ของ config file, การกำหนด mode การทำงาน, หรือการส่งค่าที่จำเป็นต่อการประมวลผลของแอปพลิเคชัน

kubectl run demo-cmd --image=busybox --command -- ping 8.8.8.8

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo-cmd
  name: demo-cmd
spec:
  containers:
  - command:
    - ping
    - 8.8.8.8
    image: busybox
    name: demo-cmd
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Example

รัน CMD ping 8.8.8.8 จำนวน 10 ครั้งแล้วสั่ง sleep 5 นาที

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: demo-ping
  name: demo-ping
spec:
  containers:
  - image: busybox
    name: demo-ping
    command: ["/bin/sh"]
    arg: ["-c", "ping -c 10 8.8.8.8 && sleep 300"]
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Multi Container-Pod

โดยทั่วไปแล้ว การใช้งานจะนิยมรันแบบ 1:1 ระหว่าง Pod และ Container เพื่อความเรียบง่ายในการจัดการและสเกลระบบ แต่ในบางกรณีอาจมีความจำเป็นต้องรันหลาย Container ภายใน Pod เดียวกัน เช่น การเพิ่ม Sidecar Container เพื่อจัดการกับการเก็บ log หรือการทำ validation ก่อนที่ Container หลักจะเริ่มทำงาน

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: multi-con
  name: multi-con
spec:
  replicas: 1
  selector:
    matchLabels:
      app: multi-con
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: multi-con
    spec:
      containers:
      - name: con-cmd
        image: busybox
        command: ["/bin/sh"]
        args: ["-c", "ping -c 5 8.8.8.8 && sleep 3600"]
      - name: con-web
        image: nginx
        resources: {}
status: {}
```

```
controlplane:~$ k get po
NAME                          READY   STATUS    RESTARTS   AGE
multi-con-76fb9bd649-wt8v5    2/2     Running   0          3s
```

Init-Container

โดยปกติแล้วในการใช้งานแบบ Multi-container ภายในPodจะมีการเริ่มต้น container หลักทั้งหมดพร้อมกัน ซึ่งอาจทำให้เกิดปัญหาเรื่องลำดับการทำงานหรือการพึ่งพาข้อมูลบางส่วนก่อนเริ่มต้น container หลัก ดังนั้นจึงมีการใช้ Init container เพื่อจัดการขั้นตอนเตรียมความพร้อม เช่น การโหลดไฟล์ การตั้งค่าระบบ หรือการตรวจสอบเงื่อนไขก่อนให้ container หลักเริ่มทำงานได้อย่างถูกต้องและเป็นระเบียบ

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: multi-con
  name: multi-con
spec:
  replicas: 1
  selector:
    matchLabels:
      app: multi-con
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: multi-con
    spec:
      initContainers:
      - name: init-myservice
        image: busybox:1.28
        command: ["/bin/sh"]
        args: ["-c", "ping -c 5 8.8.8.8 && sleep 60"]
      containers:
      - image: nginx
        name: nginx
        resources: {}
status: {}
```

```
Every 2.0s: kubectl get pods

NAME                          READY   STATUS     RESTARTS   AGE
multi-con-6b8f445c46-6wk4j    0/1     Init:0/1   0          29s
```

```
Every 2.0s: kubectl get pods

NAME                          READY   STATUS    RESTARTS   AGE
multi-con-6b8f445c46-6wk4j    1/1     Running   0          69s
```

Resource Magement

Resource Management คือการควบคุมการใช้ทรัพยากรของแต่ละ Pod โดยสามารถกำหนดได้ทั้ง CPU และ Memory ผ่านสองฟังก์ชันหลัก

- Request -> ระบุทรัพยากรขั้นต่ำที่ Pod ต้องการใช้งาน
- Scheduler จะใช้ค่านี้ในการเลือก Node ที่มีทรัพยากรเพียงพอ

ถ้าไม่มี Node ที่ตรงตาม request ระบบจะไม่ deploy Pod นั้น

Limit -> ระบุขีดจำกัดสูงสุดของทรัพยากรที่ Pod สามารถใช้ได้

- kubelet จะเป็นผู้บังคับใช้ limit นี้ในระดับ Node

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: multi-con
  name: multi-con
spec:
  replicas: 1
  selector:
    matchLabels:
      app: multi-con
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: multi-con
    spec:
      containers:
      - name: con-cmd
        image: busybox
        command: ["/bin/sh"]
        args: ["-c", "ping -c 5 8.8.8.8 && sleep 3600"]
        resources:
          requests:
            memory: "32Mi"
            cpu: ".25"
          limits:
            memory: "64Mi"
            cpu: ".5"
      - name: con-web
        image: nginx
        resources:
          requests:
            memory: "64Mi"
            cpu: ".25"
          limits:
            memory: "128Mi"
            cpu: ".5"
status: {}
```

```
NAME                          CPU(cores)   MEMORY(bytes)
multi-con-5f9f4d498c-d5rlf    0m           2Mi
controlplane:~$
```