Linux Academy CKA Notes :

## Upgrading the Kubernetes Cluster :

----------------------------------------------------------------------------

Kubeadm allows us to upgrade our cluster components in the proper order, making sure to include important feature upgrades we might want to take advantage of in the latest stable version of Kubernertes. In this lesson, we will go through upgrading our cluster from version 1.13.5 to 1.14.1.

Get the version of the API server:

kubectl version --short

View the version of kubelet:

kubectl describe nodes

View the version of controller-manager pod:

kubectl get po [controller\_pod\_name] -o yaml -n kube-system

Release the hold on versions of kubeadm and kubelet:

sudo apt-mark unhold kubeadm kubelet

Install version 1.14.1 of kubeadm:

sudo apt install -y kubeadm=1.14.1-00

Hold the version of kubeadm at 1.14.1:

sudo apt-mark hold kubeadm

Verify the version of kubeadm:

kubeadm version

Plan the upgrade of all the controller components:

sudo kubeadm upgrade plan

Upgrade the controller components:

sudo kubeadm upgrade apply v1.14.1

Release the hold on the version of kubectl:

sudo apt-mark unhold kubectl

Upgrade kubectl:

sudo apt install -y kubectl=1.14.1-00

Hold the version of kubectl at 1.14.1:

sudo apt-mark hold kubectl

Upgrade the version of kubelet:

sudo apt install -y kubelet=1.14.1-00

Hold the version of kubelet at 1.14.1:

sudo apt-mark hold kubelet

**Helpful Links**

* [Upgrading Kubernetes](https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-upgrade/)
* [Changelog for v1.13](https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG-1.13.md)

Simple Steps :

On master:

Unhold kubeadm,kubelet version

Upgrade kubeadm on Master node , hold the version upgrade the control plain components.

Unhold the kubectl version , Upgrade the kubectl , hold the new version again , upgrade the kubelet hold the version.

On Workers:

Unhold kubeadm,kubelet version

Upgrade kubeadm on worker node , hold the version

Unhold the kubectl version , Upgrade the kubectl , hold the new version again , upgrade the kubelet hold the version.

**Operating System Upgrades within a Kubernetes Cluster**

When we need to take a node down for maintenance, Kubernetes makes it easy to evict the pods on that node, take it down, and then continue scheduling pods after the maintenance is complete. Furthermore, if the node needs to be decommissioned, you can just as easily remove the node and replace it with a new one, joining it to the cluster.

See which pods are running on which nodes:

kubectl get pods -o wide

Evict the pods on a node:

kubectl drain [node\_name] --ignore-daemonsets

Watch as the node changes status:

kubectl get nodes -w

Schedule pods to the node after maintenance is complete:

kubectl uncordon [node\_name]

Remove a node from the cluster:

kubectl delete node [node\_name]

Generate a new token:

sudo kubeadm token generate

List the tokens:

sudo kubeadm token list

Print the kubeadm join command to join a node to the cluster:

sudo kubeadm token create [token\_name] --ttl 2h --print-join-command

**Helpful Links**

* [Maintenance on a Node](https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/#maintenance-on-a-node)

## Backing Up and Restoring a Kubernetes Cluster :

Backing up your cluster can be a useful exercise, especially if you have a single etcd cluster, as all the cluster state is stored there. The etcdctl utility allows us to easily create a snapshot of our cluster state (etcd) and save this to an external location. In this lesson, we’ll go through creating the snapshot and talk about restoring in the event of failure.

Get the etcd binaries:

wget https://github.com/etcd-io/etcd/releases/download/v3.3.12/etcd-v3.3.12-linux-amd64.tar.gz

Unzip the compressed binaries:

tar xvf etcd-v3.3.12-linux-amd64.tar.gz

Move the files into /usr/local/bin:

sudo mv etcd-v3.3.12-linux-amd64/etcd\* /usr/local/bin

Take a snapshot of the etcd datastore using etcdctl:

sudo ETCDCTL\_API=3 etcdctl snapshot save snapshot.db --cacert /etc/kubernetes/pki/etcd/server.crt --cert /etc/kubernetes/pki/etcd/ca.crt --key /etc/kubernetes/pki/etcd/ca.key

View the help page for etcdctl:

ETCDCTL\_API=3 etcdctl --help

Browse to the folder that contains the certificate files:

cd /etc/kubernetes/pki/etcd/

View that the snapshot was successful:

ETCDCTL\_API=3 etcdctl --write-out=table snapshot status snapshot.db

Zip up the contents of the etcd directory:

sudo tar -zcvf etcd.tar.gz /etc/kubernetes/pki/etcd

Copy the etcd directory to another server:

scp etcd.tar.gz cloud\_user@18.219.235.42:~/

**Helpful Links**

* [Backing up the etcd Store](https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster)
* [etcd Disaster Recovery Examples](https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/recovery.md)

## Pod and Node Networking

Kubernetes keeps networking simple for effective communication between pods, even if they are located on a different node. In this lesson, we’ll talk about pod communication from within a node, including how to inspect the virtual interfaces, and then get into what happens when a pod wants to talk to another pod on a different node.

See which node our pod is on:

kubectl get pods -o wide

Log in to the node:

ssh [node\_name]

View the node's virtual network interfaces:

ifconfig

View the containers in the pod:

docker ps

Get the process ID for the container:

docker inspect --format '{{ .State.Pid }}' [container\_id]

Use nsenter to run a command in the process's network namespace:

nsenter -t [container\_pid] -n ip addr

**Helpful Links**

* [Cluster Networking](https://kubernetes.io/docs/concepts/cluster-administration/networking/)

## Container Network Interface (CNI)

A Container Network Interface (CNI) is an easy way to ease communication between containers in a cluster. The CNI has many responsibilities, including IP management, encapsulating packets, and mappings in userspace. In this lesson, we will cover the details of the Flannel CNI we used in our Linux Academy cluster and talk about the ways in which we simplified communication in our cluster.

Apply the Flannel CNI plugin:

kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml

**Helpful Links**

* [Flannel Documentation](https://github.com/coreos/flannel/blob/master/Documentation/kubernetes.md)
* [Installing Other CNI Plugins](https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/#pod-network)
* [Installing Addons in Kubernetes](https://kubernetes.io/docs/concepts/cluster-administration/addons/)

## Service Networking

Services allow our pods to move around, get deleted, and replicate, all without having to manually keep track of their IP addresses in the cluster. This is accomplished by creating one gateway to distribute packets evenly across all pods. In this lesson, we will see the differences between a NodePort service and a ClusterIP service and see how the iptables rules take effect when traffic is coming in.

YAML for the nginx NodePort service:

apiVersion: v1

kind: Service

metadata:

name: nginx-nodeport

spec:

type: NodePort

ports:

- protocol: TCP

port: 80

targetPort: 80

nodePort: 30080

selector:

app: nginx

Get the services YAML output for all the services in your cluster:

kubectl get services -o yaml

Try and ping the clusterIP service IP address:

ping 10.96.0.1

View the list of services in your cluster:

kubectl get services

View the list of endpoints in your cluster that get created with a service:

kubectl get endpoints

Look at the iptables rules for your services:

sudo iptables-save | grep KUBE | grep nginx

**Helpful Links**

* [Services in Kubernetes](https://kubernetes.io/docs/concepts/services-networking/service/)

## Ingress Rules and Load Balancers

When handling traffic from outside sources, there are two ways to direct that traffic to your pods: deploying a load balancer, and creating an ingress controller and an Ingress resource. In this lesson, we will talk about the benefits of each and how Kubernetes distributes traffic to the pods on a node to reduce latency and direct traffic to the appropriate services within your cluster.

View the list of services:

kubectl get services

The load balancer YAML spec:

apiVersion: v1

kind: Service

metadata:

name: nginx-loadbalancer

spec:

type: LoadBalancer

ports:

- port: 80

targetPort: 80

selector:

app: nginx

Create a new deployment:

kubectl run kubeserve2 --image=chadmcrowell/kubeserve2

View the list of deployments:

kubectl get deployments

Scale the deployments to 2 replicas:

kubectl scale deployment/kubeserve2 --replicas=2

View which pods are on which nodes:

kubectl get pods -o wide

Create a load balancer from a deployment:

kubectl expose deployment kubeserve2 --port 80 --target-port 8080 --type LoadBalancer

View the services in your cluster:

kubectl get services

Watch as an external port is created for a service:

kubectl get services -w

Look at the YAML for a service:

kubectl get services kubeserve2 -o yaml

Curl the external IP of the load balancer:

curl http://[external-ip]

View the annotation associated with a service:

kubectl describe services kubeserve

Set the annotation to route load balancer traffic local to the node:

kubectl annotate service kubeserve2 externalTrafficPolicy=Local

The YAML for an Ingress resource:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: service-ingress

spec:

rules:

- host: kubeserve.example.com

http:

paths:

- backend:

serviceName: kubeserve2

servicePort: 80

- host: app.example.com

http:

paths:

- backend:

serviceName: nginx

servicePort: 80

- http:

paths:

- backend:

serviceName: httpd

servicePort: 80

Edit the ingress rules:

kubectl edit ingress

View the existing ingress rules:

kubectl describe ingress

Curl the hostname of your Ingress resource:

curl http://kubeserve2.example.com

**Helpful Links**

* [Create an External Load Balancer](https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/)
* [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/)

## Cluster DNS

CoreDNS is now the new default DNS plugin for Kubernetes. In this lesson, we’ll go over the hostnames for pods and services. We will also discover how you can customize DNS to include your own nameservers.

View the CoreDNS pods in the kube-system namespace:

kubectl get pods -n kube-system

View the CoreDNS deployment in your Kubernetes cluster:

kubectl get deployments -n kube-system

View the service that performs load balancing for the DNS server:

kubectl get services -n kube-system

Spec for the busybox pod:

apiVersion: v1

kind: Pod

metadata:

name: busybox

namespace: default

spec:

containers:

- image: busybox:1.28.4

command:

- sleep

- "3600"

imagePullPolicy: IfNotPresent

name: busybox

restartPolicy: Always

View the resolv.conf file that contains the nameserver and search in DNS:

kubectl exec -it busybox -- cat /etc/resolv.conf

Look up the DNS name for the native Kubernetes service:

kubectl exec -it busybox -- nslookup kubernetes

Look up the DNS names of your pods:

kubectl exec -ti busybox -- nslookup [pod-ip-address].default.pod.cluster.local

Look up a service in your Kubernetes cluster:

kubectl exec -it busybox -- nslookup kube-dns.kube-system.svc.cluster.local

Get the logs of your CoreDNS pods:

kubectl logs [coredns-pod-name]

YAML spec for a headless service:

apiVersion: v1

kind: Service

metadata:

name: kube-headless

spec:

clusterIP: None

ports:

- port: 80

targetPort: 8080

selector:

app: kubserve2

YAML spec for a custom DNS pod:

apiVersion: v1

kind: Pod

metadata:

namespace: default

name: dns-example

spec:

containers:

- name: test

image: nginx

dnsPolicy: "None"

dnsConfig:

nameservers:

- 8.8.8.8

searches:

- ns1.svc.cluster.local

- my.dns.search.suffix

options:

- name: ndots

value: "2"

- name: edns0

**Helpful Links**

* [DNS for Services and Pods](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)
* [Debugging DNS Resolution](https://kubernetes.io/docs/tasks/administer-cluster/dns-debugging-resolution/)
* [Customizing DNS](https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/)
* [CoreDNS GitHub](https://github.com/coredns/deployment/tree/master/kubernetes)
* [Kubernetes DNS-Based Service Discovery](https://github.com/kubernetes/dns/blob/master/docs/specification.md)
* [Deploying CoreDNS using kubeadm](https://coredns.io/2018/01/29/deploying-kubernetes-with-coredns-using-kubeadm/)

## Configuring the Kubernetes Scheduler

The default scheduler in Kubernetes attempts to find the best node for your pod by going through a series of steps. In this lesson, we will cover the steps in detail in order to better understand the scheduler’s function when placing pods on nodes to maximize uptime for the applications running in your cluster. We will also go through how to create a deployment with node affinity.

Label your node as being located in availability zone 1:

kubectl label node chadcrowell1c.mylabserver.com availability-zone=zone1

Label your node as dedicated infrastructure:

kubectl label node chadcrowell2c.mylabserver.com share-type=dedicated

Here is the YAML for the deployment to include the node affinity rules:

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

name: pref

spec:

replicas: 5

template:

metadata:

labels:

app: pref

spec:

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 80

preference:

matchExpressions:

- key: availability-zone

operator: In

values:

- zone1

- weight: 20

preference:

matchExpressions:

- key: share-type

operator: In

values:

- dedicated

containers:

- args:

- sleep

- "99999"

image: busybox

name: main

Create the deployment:

kubectl create -f pref-deployment.yaml

View the deployment:

kubectl get deployments

View which pods landed on which nodes:

kubectl get pods -o wide

#### Helpful Links

* [Assigning a Pod to a Node](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/)
* [Pod and Node Affinity Rules](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity)

**Running Multiple Schedulers for Multiple Pods**

In Kubernetes, you can run multiple schedulers simultaneously. You can then use different schedulers to schedule different pods. You may, for example, want to set different rules for the scheduler to run all of your pods on one node. In this lesson, I will show you how to deploy a new scheduler alongside your default scheduler and then schedule three different pods using the two schedulers.

**ClusterRole.yaml**

apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRole

metadata:

name: csinodes-admin

rules:

- apiGroups: ["storage.k8s.io"]

resources: ["csinodes"]

verbs: ["get", "watch", "list"]

**ClusterRoleBinding.yaml**

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: read-csinodes-global

subjects:

- kind: ServiceAccount

name: my-scheduler

namespace: kube-system

roleRef:

kind: ClusterRole

name: csinodes-admin

apiGroup: rbac.authorization.k8s.io

**Role.yaml**

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

name: system:serviceaccount:kube-system:my-scheduler

namespace: kube-system

rules:

- apiGroups:

- storage.k8s.io

resources:

- csinodes

verbs:

- get

- list

- watch

**RoleBinding.yaml**

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-csinodes

namespace: kube-system

subjects:

- kind: User

name: kubernetes-admin

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: system:serviceaccount:kube-system:my-scheduler

apiGroup: rbac.authorization.k8s.io

**Edit the existing kube-scheduler cluster role with kubectl edit clusterrole system:kube-scheduler and add the following:**

- apiGroups:

- ""

resourceNames:

- kube-scheduler

- my-scheduler

resources:

- endpoints

verbs:

- delete

- get

- patch

- update

- apiGroups:

- storage.k8s.io

resources:

- storageclasses

verbs:

- watch

- list

- get

**My-scheduler.yaml**

apiVersion: v1

kind: ServiceAccount

metadata:

name: my-scheduler

namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: my-scheduler-as-kube-scheduler

subjects:

- kind: ServiceAccount

name: my-scheduler

namespace: kube-system

roleRef:

kind: ClusterRole

name: system:kube-scheduler

apiGroup: rbac.authorization.k8s.io

---

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

component: scheduler

tier: control-plane

name: my-scheduler

namespace: kube-system

spec:

selector:

matchLabels:

component: scheduler

tier: control-plane

replicas: 1

template:

metadata:

labels:

component: scheduler

tier: control-plane

version: second

spec:

serviceAccountName: my-scheduler

containers:

- command:

- /usr/local/bin/kube-scheduler

- --address=0.0.0.0

- --leader-elect=false

- --scheduler-name=my-scheduler

image: chadmcrowell/custom-scheduler

livenessProbe:

httpGet:

path: /healthz

port: 10251

initialDelaySeconds: 15

name: kube-second-scheduler

readinessProbe:

httpGet:

path: /healthz

port: 10251

resources:

requests:

cpu: '0.1'

securityContext:

privileged: false

volumeMounts: []

hostNetwork: false

hostPID: false

volumes: []

**Run the deployment for my-scheduler:**

kubectl create -f my-scheduler.yaml

**View your new scheduler in the kube-system namespace:**

kubectl get pods -n kube-system

**pod1.yaml**

apiVersion: v1

kind: Pod

metadata:

name: no-annotation

labels:

name: multischeduler-example

spec:

containers:

- name: pod-with-no-annotation-container

image: k8s.gcr.io/pause:2.0

**pod2.yaml**

apiVersion: v1

kind: Pod

metadata:

name: annotation-default-scheduler

labels:

name: multischeduler-example

spec:

schedulerName: default-scheduler

containers:

- name: pod-with-default-annotation-container

image: k8s.gcr.io/pause:2.0

**pod3.yaml**

apiVersion: v1

kind: Pod

metadata:

name: annotation-second-scheduler

labels:

name: multischeduler-example

spec:

schedulerName: my-scheduler

containers:

- name: pod-with-second-annotation-container

image: k8s.gcr.io/pause:2.0

View the pods as they are created:

kubectl get pods -o wide

**Helpful Links**

* [Configure Multiple Schedulers](https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/)

## Scheduling Pods with Resource Limits and Label Selectors

In order to share the resources of your node properly, you can set resource limits and requests in Kubernetes. This allows you to reserve enough CPU and memory for your application while still maintaining system health. In this lesson, we will create some requests and limits in our pod YAML to show how it’s used by the node.

View the capacity and the allocatable info from a node:

kubectl describe nodes

The pod YAML for a pod with requests:

apiVersion: v1

kind: Pod

metadata:

name: resource-pod1

spec:

nodeSelector:

kubernetes.io/hostname: "chadcrowell3c.mylabserver.com"

containers:

- image: busybox

command: ["dd", "if=/dev/zero", "of=/dev/null"]

name: pod1

resources:

requests:

cpu: 800m

memory: 20Mi

Create the requests pod:

kubectl create -f resource-pod1.yaml

View the pods and nodes they landed on:

kubectl get pods -o wide

The YAML for a pod that has a large request:

apiVersion: v1

kind: Pod

metadata:

name: resource-pod2

spec:

nodeSelector:

kubernetes.io/hostname: "chadcrowell3c.mylabserver.com"

containers:

- image: busybox

command: ["dd", "if=/dev/zero", "of=/dev/null"]

name: pod2

resources:

requests:

cpu: 1000m

memory: 20Mi

Create the pod with 1000 millicore request:

kubectl create -f resource-pod2.yaml

See why the pod with a large request didn’t get scheduled:

kubectl describe resource-pod2

Look at the total requests per node:

kubectl describe nodes chadcrowell3c.mylabserver.com

Delete the first pod to make room for the pod with a large request:

kubectl delete pods resource-pod1

Watch as the first pod is terminated and the second pod is started:

kubectl get pods -o wide -w

The YAML for a pod that has limits:

apiVersion: v1

kind: Pod

metadata:

name: limited-pod

spec:

containers:

- image: busybox

command: ["dd", "if=/dev/zero", "of=/dev/null"]

name: main

resources:

limits:

cpu: 1

memory: 20Mi

Create a pod with limits:

kubectl create -f limited-pod.yaml

Use the exec utility to use the top command:

kubectl exec -it limited-pod top

#### Helpful Links

* [Configure Default CPU Requests and Limits](https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace/)
* [Configure Default Memory Requests and Limits](https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/)

## DaemonSets and Manually Scheduled Pods

DaemonSets do not use a scheduler to deploy pods. In fact, there are currently DaemonSets in the Kubernetes cluster that we made. In this lesson, I will show you where to find those and how to create your own DaemonSet pods to deploy without the need for a scheduler.

Find the DaemonSet pods that exist in your kubeadm cluster:

kubectl get pods -n kube-system -o wide

Delete a DaemonSet pod and see what happens:

kubectl delete pods [pod\_name] -n kube-system

Give the node a label to signify it has SSD:

kubectl label node[node\_name] disk=ssd

The YAML for a DaemonSet:

apiVersion: apps/v1beta2

kind: DaemonSet

metadata:

name: ssd-monitor

spec:

selector:

matchLabels:

app: ssd-monitor

template:

metadata:

labels:

app: ssd-monitor

spec:

nodeSelector:

disk: ssd

containers:

- name: main

image: linuxacademycontent/ssd-monitor

Create a DaemonSet from a YAML spec:

kubectl create -f ssd-monitor.yaml

Label another node to specify it has SSD:

kubectl label node chadcrowell2c.mylabserver.com disk=ssd

View the DaemonSet pods that have been deployed:

kubectl get pods -o wide

Remove the label from a node and watch the DaemonSet pod terminate:

kubectl label node chadcrowell3c.mylabserver.com disk-

Change the label on a node to change it to spinning disk:

kubectl label node chadcrowell2c.mylabserver.com disk=hdd --overwrite

Pick the label to choose for your DaemonSet:

kubectl get nodes chadcrowell3c.mylabserver.com --show-labels

**Helpful Links**

* [DaemonSets](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/)

**Displaying Scheduler Events**

There are multiple ways to view the events related to the scheduler. In this lesson, we’ll look at ways in which you can troubleshoot any problems with your scheduler or just find out more information.

View the name of the scheduler pod:

kubectl get pods -n kube-system

Get the information about your scheduler pod events:

kubectl describe pods [scheduler\_pod\_name] -n kube-system

View the events in your default namespace:

kubectl get events

View the events in your kube-system namespace:

kubectl get events -n kube-system

Delete all the pods in your default namespace:

kubectl delete pods --all

Watch events as they are appearing in real time:

kubectl get events -w

View the logs from the scheduler pod:

kubectl logs [kube\_scheduler\_pod\_name] -n kube-system

The location of a systemd service scheduler pod:

/var/log/kube-scheduler.log

**Helpful Links**

* [Verify the Desired Scheduler](https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/#verifying-that-the-pods-were-scheduled-using-the-desired-schedulers)

## Deploying an Application, Rolling Updates, and Rollbacks

We already know Kubernetes will run pods and deployments, but what happens when you need to update or change the version of your application running inside of the Kubernetes cluster? That’s where rolling updates come in, allowing you to update the app image with zero downtime. In this lesson, we’ll go over a rolling update, how to roll back, and how to pause the update if things aren’t going well.

The YAML for a deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: kubeserve

spec:

replicas: 3

selector:

matchLabels:

app: kubeserve

template:

metadata:

name: kubeserve

labels:

app: kubeserve

spec:

containers:

- image: linuxacademycontent/kubeserve:v1

name: app

Create a deployment with a record (for rollbacks):

kubectl create -f kubeserve-deployment.yaml --record

Check the status of the rollout:

kubectl rollout status deployments kubeserve

View the ReplicaSets in your cluster:

kubectl get replicasets

Scale up your deployment by adding more replicas:

kubectl scale deployment kubeserve --replicas=5

Expose the deployment and provide it a service:

kubectl expose deployment kubeserve --port 80 --target-port 80 --type NodePort

Set the minReadySeconds attribute to your deployment:

kubectl patch deployment kubeserve -p '{"spec": {"minReadySeconds": 10}}'

Use kubectl apply to update a deployment:

kubectl apply -f kubeserve-deployment.yaml

Use kubectl replace to replace an existing deployment:

kubectl replace -f kubeserve-deployment.yaml

Run this curl look while the update happens:

while true; do curl http://10.105.31.119; done

Perform the rolling update:

kubectl set image deployments/kubeserve app=linuxacademycontent/kubeserve:v2 --v 6

Describe a certain ReplicaSet:

kubectl describe replicasets kubeserve-[hash]

Apply the rolling update to version 3 (buggy):

kubectl set image deployment kubeserve app=linuxacademycontent/kubeserve:v3

Undo the rollout and roll back to the previous version:

kubectl rollout undo deployments kubeserve

Look at the rollout history:

kubectl rollout history deployment kubeserve

Roll back to a certain revision:

kubectl rollout undo deployment kubeserve --to-revision=2

Pause the rollout in the middle of a rolling update (canary release):

kubectl rollout pause deployment kubeserve

Resume the rollout after the rolling update looks good:

kubectl rollout resume deployment kubeserve

### Helpful Links

* [Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/)
* [Creating a Deployment](https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/)
* [Performing a Rolling Update](https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/)

## Configuring an Application for High Availability and Scale

Continuing from the last lesson, we will go through how Kubernetes will save you from EVER releasing code with bugs. Then, we will talk about ConfigMaps and secrets as a way to pass configuration data to your apps.

The YAML for a readiness probe:

apiVersion: apps/v1

kind: Deployment

metadata:

name: kubeserve

spec:

replicas: 3

selector:

matchLabels:

app: kubeserve

minReadySeconds: 10

strategy:

rollingUpdate:

maxSurge: 1

maxUnavailable: 0

type: RollingUpdate

template:

metadata:

name: kubeserve

labels:

app: kubeserve

spec:

containers:

- image: linuxacademycontent/kubeserve:v3

name: app

readinessProbe:

periodSeconds: 1

httpGet:

path: /

port: 80

Apply the readiness probe:

kubectl apply -f kubeserve-deployment-readiness.yaml

View the rollout status:

kubectl rollout status deployment kubeserve

Describe deployment:

kubectl describe deployment

Create a ConfigMap with two keys:

kubectl create configmap appconfig --from-literal=key1=value1 --from-literal=key2=value2

Get the YAML back out from the ConfigMap:

kubectl get configmap appconfig -o yaml

The YAML for the ConfigMap pod:

apiVersion: v1

kind: Pod

metadata:

name: configmap-pod

spec:

containers:

- name: app-container

image: busybox:1.28

command: ['sh', '-c', "echo $(MY\_VAR) && sleep 3600"]

env:

- name: MY\_VAR

valueFrom:

configMapKeyRef:

name: appconfig

key: key1

Create the pod that is passing the ConfigMap data:

kubectl apply -f configmap-pod.yaml

Get the logs from the pod displaying the value:

kubectl logs configmap-pod

The YAML for a pod that has a ConfigMap volume attached:

apiVersion: v1

kind: Pod

metadata:

name: configmap-volume-pod

spec:

containers:

- name: app-container

image: busybox

command: ['sh', '-c', "echo $(MY\_VAR) && sleep 3600"]

volumeMounts:

- name: configmapvolume

mountPath: /etc/config

volumes:

- name: configmapvolume

configMap:

name: appconfig

Create the ConfigMap volume pod:

kubectl apply -f configmap-volume-pod.yaml

Get the keys from the volume on the container:

kubectl exec configmap-volume-pod -- ls /etc/config

Get the values from the volume on the pod:

kubectl exec configmap-volume-pod -- cat /etc/config/key1

The YAML for a secret:

apiVersion: v1

kind: Secret

metadata:

name: appsecret

stringData:

cert: value

key: value

Create the secret:

kubectl apply -f appsecret.yaml

The YAML for a pod that will use the secret:

apiVersion: v1

kind: Pod

metadata:

name: secret-pod

spec:

containers:

- name: app-container

image: busybox

command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]

env:

- name: MY\_CERT

valueFrom:

secretKeyRef:

name: appsecret

key: cert

Create the pod that has attached secret data:

kubectl apply -f secret-pod.yaml

Open a shell and echo the environment variable:

kubectl exec -it secret-pod -- sh

echo $MY\_CERT

The YAML for a pod that will access the secret from a volume:

apiVersion: v1

kind: Pod

metadata:

name: secret-volume-pod

spec:

containers:

- name: app-container

image: busybox

command: ['sh', '-c', "echo $(MY\_VAR) && sleep 3600"]

volumeMounts:

- name: secretvolume

mountPath: /etc/certs

volumes:

- name: secretvolume

secret:

secretName: appsecret

Create the pod with volume attached with secrets:

kubectl apply -f secret-volume-pod.yaml

Get the keys from the volume mounted to the container with the secrets:

kubectl exec secret-volume-pod -- ls /etc/certs

#### Helpful Links

* [Scaling Your Application](https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#scaling-your-application)
* [Configure Pod ConfigMaps](https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/)
* [Secrets](https://kubernetes.io/docs/concepts/configuration/secret/)

## Creating a Self-Healing Application

In this lesson, we’ll go through the power of ReplicaSets, which make your application self-healing by replicating pods and moving them around and spinning them up when nodes fail. We’ll also talk about StatefulSets and the benefit they provide.

The YAML for a ReplicaSet:

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: myreplicaset

labels:

app: app

tier: frontend

spec:

replicas: 3

selector:

matchLabels:

tier: frontend

template:

metadata:

labels:

tier: frontend

spec:

containers:

- name: main

image: linuxacademycontent/kubeserve

Create the ReplicaSet:

kubectl apply -f replicaset.yaml

The YAML for a pod with the same label as a ReplicaSet:

apiVersion: v1

kind: Pod

metadata:

name: pod1

labels:

tier: frontend

spec:

containers:

- name: main

image: linuxacademycontent/kubeserve

Create the pod with the same label:

kubectl apply -f pod-replica.yaml

Watch the pod get terminated:

kubectl get pods -w

The YAML for a StatefulSet:

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

serviceName: "nginx"

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi

Create the StatefulSet:

kubectl apply -f statefulset.yaml

View all StatefulSets in the cluster:

kubectl get statefulsets

Describe the StatefulSets:

kubectl describe statefulsets

#### Helpful Links

* [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/)
* [StatefulSets](https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/)

**Persistent Volumes**

In Kubernetes, pods are ephemeral. This creates a unique challenge with attaching storage directly to the filesystem of a container. Persistent Volumes are used to create an abstraction layer between the application and the underlying storage, making it easier for the storage to follow the pods as they are deleted, moved, and created within your Kubernetes cluster.

In the Google Cloud Engine, find the region your cluster is in:

gcloud container clusters list

Using Google Cloud, create a persistent disk in the same region as your cluster:

gcloud compute disks create --size=1GiB --zone=us-central1-a mongodb

The YAML for a pod that will use persistent disk:

apiVersion: v1

kind: Pod

metadata:

name: mongodb

spec:

volumes:

- name: mongodb-data

gcePersistentDisk:

pdName: mongodb

fsType: ext4

containers:

- image: mongo

name: mongodb

volumeMounts:

- name: mongodb-data

mountPath: /data/db

ports:

- containerPort: 27017

protocol: TCP

Create the pod with disk attached and mounted:

kubectl apply -f mongodb-pod.yaml

See which node the pod landed on:

kubectl get pods -o wide

Connect to the mongodb shell:

kubectl exec -it mongodb mongo

Switch to the mystore database in the mongodb shell:

use mystore

Create a JSON document to insert into the database:

db.foo.insert({name:'foo'})

View the document you just created:

db.foo.find()

Exit from the mongodb shell:

exit

Delete the pod:

kubectl delete pod mongodb

Create a new pod with the same attached disk:

kubectl apply -f mongodb-pod.yaml

Check to see which node the pod landed on:

kubectl get pods -o wide

Drain the node (if the pod is on the same node as before):

kubectl drain [node\_name] --ignore-daemonsets

Once the pod is on a different node, access the mongodb shell again:

kubectl exec -it mongodb mongo

Access the mystore database again:

use mystore

Find the document you created from before:

db.foo.find()

The YAML for a PersistentVolume object in Kubernetes:

apiVersion: v1

kind: PersistentVolume

metadata:

name: mongodb-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

- ReadOnlyMany

persistentVolumeReclaimPolicy: Retain

gcePersistentDisk:

pdName: mongodb

fsType: ext4

Create the Persistent Volume resource:

kubectl apply -f mongodb-persistentvolume.yaml

View our Persistent Volumes:

kubectl get persistentvolumes

**Helpful Links:**

* [Persistent Volumes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)
* [Configure Persistent Volume Storage](https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/)

## Volume Access Modes

Volume access modes are how you specify the access of a node to your Persistent Volume. There are three types of access modes: ReadWriteOnce, ReadOnlyMany, and ReadWriteMany. In this lesson, we will explain what each of these access modes means and two VERY IMPORTANT things to remember when using your Persistent Volumes with pods.

The YAML for a Persistent Volume:

apiVersion: v1

kind: PersistentVolume

metadata:

name: mongodb-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

- ReadOnlyMany

persistentVolumeReclaimPolicy: Retain

gcePersistentDisk:

pdName: mongodb

fsType: ext4

View the Persistent Volumes in your cluster:

kubectl get pv

#### Helpful Links:

* [Access Modes](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes)

## Persistent Volume Claims

Persistent Volume Claims (PVCs) are a way for an application developer to request storage for the application without having to know where the underlying storage is. The claim is then bound to the Persistent Volume (PV), and it will not be released until the PVC is deleted. In this lesson, we will go through creating a PVC and accessing storage within our persistent disk.

The YAML for a PVC:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mongodb-pvc

spec:

resources:

requests:

storage: 1Gi

accessModes:

- ReadWriteOnce

storageClassName: ""

Create a PVC:

kubectl apply -f mongodb-pvc.yaml

View the PVC in the cluster:

kubectl get pvc

View the PV to ensure it’s bound:

kubectl get pv

The YAML for a pod that uses a PVC:

apiVersion: v1

kind: Pod

metadata:

name: mongodb

spec:

containers:

- image: mongo

name: mongodb

volumeMounts:

- name: mongodb-data

mountPath: /data/db

ports:

- containerPort: 27017

protocol: TCP

volumes:

- name: mongodb-data

persistentVolumeClaim:

claimName: mongodb-pvc

Create the pod with the attached storage:

kubectl apply -f mongo-pvc-pod.yaml

Access the mogodb shell:

kubectl exec -it mongodb mongo

Find the JSON document created in previous lessons:

db.foo.find()

Delete the mongodb pod:

kubectl delete pod mogodb

Delete the mongodb-pvc PVC:

kubectl delete pvc mongodb-pvc

Check the status of the PV:

kubectl get pv

The YAML for the PV to show its reclaim policy:

apiVersion: v1

kind: PersistentVolume

metadata:

name: mongodb-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

- ReadOnlyMany

persistentVolumeReclaimPolicy: Retain

gcePersistentDisk:

pdName: mongodb

fsType: ext4

#### Helpful Links

* [PersistentVolumeClaims](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims)
* [Create a PersistentVolumeClaim](https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/#create-a-persistentvolumeclaim)

## Storage Objects

There’s an even easier way to provision storage in Kubernetes with StorageClass objects. Also, your storage is safe from data loss with the “Storage Object in Use Protection” feature, which ensures any pods using a Persistent Volume will not lose the data on the volume as long as it is actively mounted. We’ve been using Google Storage for this section, but there are many different volume types you can use in Kubernetes. In this lesson, we will talk about the hostPath volume and the empty directory volume type.

See the PV protection on your volume:

kubectl describe pv mongodb-pv

See the PVC protection for your claim:

kubectl describe pvc mongodb-pvc

Delete the PVC:

kubectl delete pvc mongodb-pvc

See that the PVC is terminated, but the volume is still attached to pod:

kubectl get pvc

Try to access the data, even though we just deleted the PVC:

kubectl exec -it mongodb mongo

use mystore

db.foo.find()

Delete the pod, which finally deletes the PVC:

kubectl delete pods mongodb

Show that the PVC is deleted:

kubectl get pvc

YAML for a StorageClass object:

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: fast

provisioner: kubernetes.io/gce-pd

parameters:

type: pd-ssd

Create the StorageClass type "fast":

kubectl apply -f sc-fast.yaml

Change the PVC to include the new StorageClass object:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mongodb-pvc

spec:

storageClassName: fast

resources:

requests:

storage: 100Mi

accessModes:

- ReadWriteOnce

Create the PVC with automatically provisioned storage:

kubectl apply -f mongodb-pvc.yaml

View the PVC with new StorageClass:

kubectl get pvc

View the newly provisioned storage:

kubectl get pv

The YAML for a hostPath PV:

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-hostpath

spec:

storageClassName: local-storage

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/mnt/data"

The YAML for a pod with an empty directory volume:

apiVersion: v1

kind: Pod

metadata:

name: emptydir-pod

spec:

containers:

- image: busybox

name: busybox

command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]

volumeMounts:

- mountPath: /tmp/storage

name: vol

volumes:

- name: vol

emptyDir: {}

#### Helpful Links:

* [Object in Use Protection](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#storage-object-in-use-protection)
* [Volumes](https://kubernetes.io/docs/concepts/storage/volumes/)

## Applications with Persistent Storage

In this lesson, we’ll wrap everything up in a nice little bow and create a deployment that will allow us to use our storage with our pods. This is to demonstrate how a real-world application would be deployed and used for storing data.

The YAML for our StorageClass object:

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: fast

provisioner: kubernetes.io/gce-pd

parameters:

type: pd-ssd

The YAML for our PVC:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: kubeserve-pvc

spec:

storageClassName: fast

resources:

requests:

storage: 100Mi

accessModes:

- ReadWriteOnce

Create our StorageClass object:

kubectl apply -f storageclass-fast.yaml

View the StorageClass objects in your cluster:

kubectl get sc

Create our PVC:

kubectl apply -f kubeserve-pvc.yaml

View the PVC created in our cluster:

kubectl get pvc

View our automatically provisioned PV:

kubectl get pv

The YAML for our deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: kubeserve

spec:

replicas: 1

selector:

matchLabels:

app: kubeserve

template:

metadata:

name: kubeserve

labels:

app: kubeserve

spec:

containers:

- env:

- name: app

value: "1"

image: linuxacademycontent/kubeserve:v1

name: app

volumeMounts:

- mountPath: /data

name: volume-data

volumes:

- name: volume-data

persistentVolumeClaim:

claimName: kubeserve-pvc

Create our deployment and attach the storage to the pods:

kubectl apply -f kubeserve-deployment.yaml

Check the status of the rollout:

kubectl rollout status deployments kubeserve

Check the pods have been created:

kubectl get pods

Connect to our pod and create a file on the PV:

kubectl exec -it [pod-name] -- touch /data/file1.txt

Connect to our pod and list the contents of the /data directory:

kubectl exec -it [pod-name] -- ls /data

## Kubernetes Security Primitives

Expanding on our discussion about securing the Kubernetes cluster, we’ll take a look at service accounts and user authentication. Also in this lesson, we will create a workstation for you to administer your cluster without logging in to the Kubernetes master server.

List the service accounts in your cluster:

kubectl get serviceaccounts

Create a new jenkins service account:

kubectl create serviceaccount jenkins

Use the abbreviated version of serviceAccount:

kubectl get sa

View the YAML for our service account:

kubectl get serviceaccounts jenkins -o yaml

View the secrets in your cluster:

kubectl get secret [secret\_name]

The YAML for a busybox pod using the jenkins service account:

apiVersion: v1

kind: Pod

metadata:

name: busybox

namespace: default

spec:

serviceAccountName: jenkins

containers:

- image: busybox:1.28.4

command:

- sleep

- "3600"

imagePullPolicy: IfNotPresent

name: busybox

restartPolicy: Always

Create a new pod with the service account:

kubectl apply -f busybox.yaml

View the cluster config that kubectl uses:

kubectl config view

View the config file:

cat ~/.kube/config

Set new credentials for your cluster:

kubectl config set-credentials chad --username=chad --password=password

Create a role binding for anonymous users (not recommended):

kubectl create clusterrolebinding cluster-system-anonymous --clusterrole=cluster-admin --user=system:anonymous

SCP the certificate authority to your workstation or server:

scp ca.crt cloud\_user@[pub-ip-of-remote-server]:~/

Set the cluster address and authentication:

kubectl config set-cluster kubernetes --server=https://172.31.41.61:6443 --certificate-authority=ca.crt --embed-certs=true

Set the credentials for Chad:

kubectl config set-credentials chad --username=chad --password=password

Set the context for the cluster:

kubectl config set-context kubernetes --cluster=kubernetes --user=chad --namespace=default

Use the context:

kubectl config use-context kubernetes

Run the same commands with kubectl:

kubectl get nodes

#### Helpful Links

* [Securing the Cluster](https://kubernetes.io/docs/concepts/cluster-administration/cluster-administration-overview/#securing-a-cluster)
* [Authentication](https://kubernetes.io/docs/reference/access-authn-authz/authentication/)
* [Administer the Cluster via kubectl](https://kubernetes.io/docs/reference/kubectl/overview/)

## Cluster Authentication and Authorization

Once the API server has determined who you are (whether a pod or a user), the authorization is handled by RBAC. In this lesson, we will talk about roles, cluster roles, role bindings, and cluster role bindings.

Create a new namespace:

kubectl create ns web

The YAML for a service role:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: web

name: service-reader

rules:

- apiGroups: [""]

verbs: ["get", "list"]

resources: ["services"]

Create a new role from that YAML file:

kubectl apply -f role.yaml

Create a RoleBinding:

kubectl create rolebinding test --role=service-reader --serviceaccount=web:default -n web

Run a proxy for inter-cluster communications:

kubectl proxy

Try to access the services in the web namespace:

curl localhost:8001/api/v1/namespaces/web/services

Create a ClusterRole to access PersistentVolumes:

kubectl create clusterrole pv-reader --verb=get,list --resource=persistentvolumes

Create a ClusterRoleBinding for the cluster role:

kubectl create clusterrolebinding pv-test --clusterrole=pv-reader --serviceaccount=web:default

The YAML for a pod that includes a curl and proxy container:

apiVersion: v1

kind: Pod

metadata:

name: curlpod

namespace: web

spec:

containers:

- image: tutum/curl

command: ["sleep", "9999999"]

name: main

- image: linuxacademycontent/kubectl-proxy

name: proxy

restartPolicy: Always

Create the pod that will allow you to curl directly from the container:

kubectl apply -f curl-pod.yaml

Get the pods in the web namespace:

kubectl get pods -n web

Open a shell to the container:

kubectl exec -it curlpod -n web -- sh

Access PersistentVolumes (cluster-level) from the pod:

curl localhost:8001/api/v1/persistentvolumes

#### Helpful Links:

* [Authorization](https://kubernetes.io/docs/reference/access-authn-authz/authorization/)
* [RBAC](https://kubernetes.io/docs/reference/access-authn-authz/rbac/)
* [RoleBinding and ClusterRoleBinding](https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding)

## Configuring Network Policies

Network policies allow you to specify which pods can talk to other pods. This helps when securing communication between pods, allowing you to identify ingress and egress rules. You can apply a network policy to a pod by using pod or namespace selectors. You can even choose a CIDR block range to apply the network policy. In this lesson, we’ll go through each of these options for network policies.

Download the canal plugin:

wget -O canal.yaml https://docs.projectcalico.org/v3.5/getting-started/kubernetes/installation/hosted/canal/canal.yaml

Apply the canal plugin:

kubectl apply -f canal.yaml

The YAML for a deny-all NetworkPolicy:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: deny-all

spec:

podSelector: {}

policyTypes:

- Ingress

Run a deployment to test the NetworkPolicy:

kubectl run nginx --image=nginx --replicas=2

Create a service for the deployment:

kubectl expose deployment nginx --port=80

Attempt to access the service by using a busybox interactive pod:

kubectl run busybox --rm -it --image=busybox /bin/sh

#wget --spider --timeout=1 nginx

The YAML for a pod selector NetworkPolicy:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: db-netpolicy

spec:

podSelector:

matchLabels:

app: db

ingress:

- from:

- podSelector:

matchLabels:

app: web

ports:

- port: 5432

Label a pod to get the NetworkPolicy:

kubectl label pods [pod\_name] app=db

The YAML for a namespace NetworkPolicy:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: ns-netpolicy

spec:

podSelector:

matchLabels:

app: db

ingress:

- from:

- namespaceSelector:

matchLabels:

tenant: web

ports:

- port: 5432

The YAML for an IP block NetworkPolicy:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: ipblock-netpolicy

spec:

podSelector:

matchLabels:

app: db

ingress:

- from:

- ipBlock:

cidr: 192.168.1.0/24

The YAML for an egress NetworkPolicy:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: egress-netpol

spec:

podSelector:

matchLabels:

app: web

egress:

- to:

- podSelector:

matchLabels:

app: db

ports:

- port: 5432

#### Helpful Links:

* [Network Policies](https://kubernetes.io/docs/concepts/services-networking/network-policies/)
* [Declare Network Policies](https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/)
* [Default Network Policies](https://kubernetes.io/docs/concepts/services-networking/network-policies/#default-policies)

## Creating TLS Certificates

A Certificate Authority (CA) is used to generate TLS certificates and authenticate to your API server. In this lesson, we’ll go through certificate requests and generating a new certificate.

Find the CA certificate on a pod in your cluster:

kubectl exec busybox -- ls /var/run/secrets/kubernetes.io/serviceaccount

Download the binaries for the cfssl tool:

wget -q --show-progress --https-only --timestamping \

https://pkg.cfssl.org/R1.2/cfssl\_linux-amd64 \

https://pkg.cfssl.org/R1.2/cfssljson\_linux-amd64

Make the binary files executable:

chmod +x cfssl\_linux-amd64 cfssljson\_linux-amd64

Move the files into your bin directory:

sudo mv cfssl\_linux-amd64 /usr/local/bin/cfssl

sudo mv cfssljson\_linux-amd64 /usr/local/bin/cfssljson

Check to see if you have cfssl installed correctly:

cfssl version

Create a CSR file:

cat <<EOF | cfssl genkey - | cfssljson -bare server

{

"hosts": [

"my-svc.my-namespace.svc.cluster.local",

"my-pod.my-namespace.pod.cluster.local",

"172.168.0.24",

"10.0.34.2"

],

"CN": "my-pod.my-namespace.pod.cluster.local",

"key": {

"algo": "ecdsa",

"size": 256

}

}

EOF

Create a CertificateSigningRequest API object:

cat <<EOF | kubectl create -f -

apiVersion: certificates.k8s.io/v1beta1

kind: CertificateSigningRequest

metadata:

name: pod-csr.web

spec:

groups:

- system:authenticated

request: $(cat server.csr | base64 | tr -d '\n')

usages:

- digital signature

- key encipherment

- server auth

EOF

View the CSRs in the cluster:

kubectl get csr

View additional details about the CSR:

kubectl describe csr pod-csr.web

Approve the CSR:

kubectl certificate approve pod-csr.web

View the certificate within your CSR:

kubectl get csr pod-csr.web -o yaml

Extract and decode your certificate to use in a file:

kubectl get csr pod-csr.web -o jsonpath='{.status.certificate}' \

| base64 --decode > server.crt

#### Helpful Links

* [Managing TLS in the Cluster](https://kubernetes.io/docs/tasks/tls/managing-tls-in-a-cluster/)
* [Bootstrapping TLS for Your kubelets](https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping/)
* [How kubeadm Manages Certificates](https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-certs/)

## Secure Images

Working with secure images is imperative in Kubernetes, as it ensures your applications are running efficiently and protecting you from vulnerabilities. In this lesson, we’ll go through how to set Kubernetes to use a private registry.

View where your Docker credentials are stored:

sudo vim /home/cloud\_user/.docker/config.json

Log in to the Docker Hub:

sudo docker login

View the images currently on your server:

sudo docker images

Pull a new image to use with a Kubernetes pod:

sudo docker pull busybox:1.28.4

Log in to a private registry using the docker login command:

sudo docker login -u podofminerva -p 'otj701c9OucKZOCx5qrRblofcNRf3W+e' podofminerva.azurecr.io

View your stored credentials:

sudo vim /home/cloud\_user/.docker/config.json

Tag an image in order to push it to a private registry:

sudo docker tag busybox:1.28.4 podofminerva.azurecr.io/busybox:latest

Push the image to your private registry:

docker push podofminerva.azurecr.io/busybox:latest

Create a new docker-registry secret:

kubectl create secret docker-registry acr --docker-server=https://podofminerva.azurecr.io --docker-username=podofminerva --docker-password='otj701c9OucKZOCx5qrRblofcNRf3W+e' --docker-email=user@example.com

Modify the default service account to use your new docker-registry secret:

kubectl patch sa default -p '{"imagePullSecrets": [{"name": "acr"}]}'

The YAML for a pod using an image from a private repository:

apiVersion: v1

kind: Pod

metadata:

name: acr-pod

labels:

app: busybox

spec:

containers:

- name: busybox

image: podofminerva.azurecr.io/busybox:latest

command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']

imagePullPolicy: Always

Create the pod from the private image:

kubectl apply -f acr-pod.yaml

View the running pod:

kubectl get pods

#### Helpful Links

* [Images](https://kubernetes.io/docs/concepts/containers/images/)
* [Pull Images from a Private Registry](https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/)
* [Configure Service Accounts](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/)
* [Add ImagePullSecrets](https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#add-imagepullsecrets-to-a-service-account)
* [11 Ways (Not) to Get Hacked](https://kubernetes.io/blog/2018/07/18/11-ways-not-to-get-hacked/)

## Defining Security Contexts

Defining security contexts allows you to lock down your containers, so that only certain processes can do certain things. This ensures the stability of your containers and allows you to give control or take it away. In this lesson, we’ll go through how to set the security context at the container level and the pod level.

Run an alpine container with default security:

kubectl run pod-with-defaults --image alpine --restart Never -- /bin/sleep 999999

Check the ID on the container:

kubectl exec pod-with-defaults id

The YAML for a container that runs as a user:

apiVersion: v1

kind: Pod

metadata:

name: alpine-user-context

spec:

containers:

- name: main

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

runAsUser: 405

Create a pod that runs the container as user:

kubectl apply -f alpine-user-context.yaml

View the IDs of the new pod created with container user permission:

kubectl exec alpine-user-context id

The YAML for a pod that runs the container as non-root:

apiVersion: v1

kind: Pod

metadata:

name: alpine-nonroot

spec:

containers:

- name: main

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

runAsNonRoot: true

Create a pod that runs the container as non-root:

kubectl apply -f alpine-nonroot.yaml

View more information about the pod error:

kubectl describe pod alpine-nonroot

The YAML for a privileged container pod:

apiVersion: v1

kind: Pod

metadata:

name: privileged-pod

spec:

containers:

- name: main

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

privileged: true

Create the privileged container pod:

kubectl apply -f privileged-pod.yaml

View the devices on the default container:

kubectl exec -it pod-with-defaults ls /dev

View the devices on the privileged pod container:

kubectl exec -it privileged-pod ls /dev

Try to change the time on a default container pod:

kubectl exec -it pod-with-defaults -- date +%T -s "12:00:00"

The YAML for a container that will allow you to change the time:

apiVersion: v1

kind: Pod

metadata:

name: kernelchange-pod

spec:

containers:

- name: main

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

capabilities:

add:

- SYS\_TIME

Create the pod that will allow you to change the container’s time:

kubectl run -f kernelchange-pod.yaml

Change the time on a container:

kubectl exec -it kernelchange-pod -- date +%T -s "12:00:00"

View the date on the container:

kubectl exec -it kernelchange-pod -- date

The YAML for a container that removes capabilities:

apiVersion: v1

kind: Pod

metadata:

name: remove-capabilities

spec:

containers:

- name: main

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

capabilities:

drop:

- CHOWN

Create a pod that’s container has capabilities removed:

kubectl apply -f remove-capabilities.yaml

Try to change the ownership of a container with removed capability:

kubectl exec remove-capabilities chown guest /tmp

The YAML for a pod container that can’t write to the local filesystem:

apiVersion: v1

kind: Pod

metadata:

name: readonly-pod

spec:

containers:

- name: main

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

readOnlyRootFilesystem: true

volumeMounts:

- name: my-volume

mountPath: /volume

readOnly: false

volumes:

- name: my-volume

emptyDir:

Create a pod that will not allow you to write to the local container filesystem:

kubectl apply -f readonly-pod.yaml

Try to write to the container filesystem:

kubectl exec -it readonly-pod touch /new-file

Create a file on the volume mounted to the container:

kubectl exec -it readonly-pod touch /volume/newfile

View the file on the volume that’s mounted:

kubectl exec -it readonly-pod -- ls -la /volume/newfile

The YAML for a pod that has different group permissions for different pods:

apiVersion: v1

kind: Pod

metadata:

name: group-context

spec:

securityContext:

fsGroup: 555

supplementalGroups: [666, 777]

containers:

- name: first

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

runAsUser: 1111

volumeMounts:

- name: shared-volume

mountPath: /volume

readOnly: false

- name: second

image: alpine

command: ["/bin/sleep", "999999"]

securityContext:

runAsUser: 2222

volumeMounts:

- name: shared-volume

mountPath: /volume

readOnly: false

volumes:

- name: shared-volume

emptyDir:

Create a pod with two containers and different group permissions:

kubectl apply -f group-context.yaml

Open a shell to the first container on that pod:

kubectl exec -it group-context -c first sh

#### Helpful Links

* [Security Contexts](https://kubernetes.io/docs/tasks/configure-pod-container/security-context/)

## Securing Persistent Key Value Store

Secrets are used to secure sensitive data you may access from your pod. The data never gets written to disk because it's stored in an in-memory filesystem (tmpfs). Because secrets can be created independently of pods, there is less risk of the secret being exposed during the pod lifecycle.

View the secrets in your cluster:

kubectl get secrets

View the default secret mounted to each pod:

kubectl describe pods pod-with-defaults

View the token, certificate, and namespace within the secret:

kubectl describe secret

Generate a key for your https server:

openssl genrsa -out https.key 2048

Generate a certificate for the https server:

openssl req -new -x509 -key https.key -out https.cert -days 3650 -subj /CN=www.example.com

Create an empty file to create the secret:

touch file

Create a secret from your key, cert, and file:

kubectl create secret generic example-https --from-file=https.key --from-file=https.cert --from-file=file

View the YAML from your new secret:

kubectl get secrets example-https -o yaml

Create the configMap that will mount to your pod:

apiVersion: v1

kind: ConfigMap

metadata:

name: config

data:

my-nginx-config.conf: |

server {

listen 80;

listen 443 ssl;

server\_name www.example.com;

ssl\_certificate certs/https.cert;

ssl\_certificate\_key certs/https.key;

ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;

ssl\_ciphers HIGH:!aNULL:!MD5;

location / {

root /usr/share/nginx/html;

index index.html index.htm;

}

}

sleep-interval: |

25

The YAML for a pod using the new secret:

apiVersion: v1

kind: Pod

metadata:

name: example-https

spec:

containers:

- image: linuxacademycontent/fortune

name: html-web

env:

- name: INTERVAL

valueFrom:

configMapKeyRef:

name: config

key: sleep-interval

volumeMounts:

- name: html

mountPath: /var/htdocs

- image: nginx:alpine

name: web-server

volumeMounts:

- name: html

mountPath: /usr/share/nginx/html

readOnly: true

- name: config

mountPath: /etc/nginx/conf.d

readOnly: true

- name: certs

mountPath: /etc/nginx/certs/

readOnly: true

ports:

- containerPort: 80

- containerPort: 443

volumes:

- name: html

emptyDir: {}

- name: config

configMap:

name: config

items:

- key: my-nginx-config.conf

path: https.conf

- name: certs

secret:

secretName: example-https

Describe the nginx conf via ConfigMap:

kubectl describe configmap

View the cert mounted on the container:

kubectl exec example-https -c web-server -- mount | grep certs

Use port forwarding on the pod to server traffic from 443:

kubectl port-forward example-https 8443:443 &

Curl the web server to get a response:

curl https://localhost:8443 -k

#### Helpful Links

* [Secrets](https://kubernetes.io/docs/concepts/configuration/secret/)

## Monitoring the Cluster Components

We are able to monitor the CPU and memory utilization of our pods and nodes by using the metrics server. In this lesson, we’ll install the metrics server and see how the kubectl top command works.

Clone the metrics server repository:

git clone https://github.com/linuxacademy/metrics-server

Install the metrics server in your cluster:

kubectl apply -f ~/metrics-server/deploy/1.8+/

Get a response from the metrics server API:

kubectl get --raw /apis/metrics.k8s.io/

Get the CPU and memory utilization of the nodes in your cluster:

kubectl top node

Get the CPU and memory utilization of the pods in your cluster:

kubectl top pods

Get the CPU and memory of pods in all namespaces:

kubectl top pods --all-namespaces

Get the CPU and memory of pods in only one namespace:

kubectl top pods -n kube-system

Get the CPU and memory of pods with a label selector:

kubectl top pod -l run=pod-with-defaults

Get the CPU and memory of a specific pod:

kubectl top pod pod-with-defaults

Get the CPU and memory of the containers inside the pod:

kubectl top pods group-context --containers

#### Helpful Links

* [Monitor Node Health](https://kubernetes.io/docs/tasks/debug-application-cluster/monitor-node-health/)
* [Resource Usage Monitoring](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/)

## Monitoring the Applications Running within a Cluster

There are ways Kubernetes can automatically monitor your apps for you and, furthermore, fix them by either restarting or preventing them from affecting the rest of your service. You can insert liveness probes and readiness probes to do just this for custom monitoring of your applications.

The pod YAML for a liveness probe:

apiVersion: v1

kind: Pod

metadata:

name: liveness

spec:

containers:

- image: linuxacademycontent/kubeserve

name: kubeserve

livenessProbe:

httpGet:

path: /

port: 80

The YAML for a service and two pods with readiness probes:

apiVersion: v1

kind: Service

metadata:

name: nginx

spec:

type: LoadBalancer

ports:

- port: 80

targetPort: 80

selector:

app: nginx

---

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

readinessProbe:

httpGet:

path: /

port: 80

initialDelaySeconds: 5

periodSeconds: 5

---

apiVersion: v1

kind: Pod

metadata:

name: nginxpd

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:191

readinessProbe:

httpGet:

path: /

port: 80

initialDelaySeconds: 5

periodSeconds: 5

Create the service and two pods with readiness probes:

kubectl apply -f readiness.yaml

Check if the readiness check passed or failed:

kubectl get pods

Check if the failed pod has been added to the list of endpoints:

kubectl get ep

Edit the pod to fix the problem and enter it back into the service:

kubectl edit pod [pod\_name]

Get the list of endpoints to see that the repaired pod is part of the service again:

kubectl get ep

#### Helpful Links

* [Container Probes](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes)

## Managing Cluster Component Logs

There are many ways to manage the logs that can accumulate from both applications and system components. In this lesson, we’ll go through a few different approaches to organizing your logs.

The directory where the container logs reside:

/var/log/containers

The directory where kubelet stores its logs:

/var/log

The YAML for a pod that has two different log streams:

apiVersion: v1

kind: Pod

metadata:

name: counter

spec:

containers:

- name: count

image: busybox

args:

- /bin/sh

- -c

- >

i=0;

while true;

do

echo "$i: $(date)" >> /var/log/1.log;

echo "$(date) INFO $i" >> /var/log/2.log;

i=$((i+1));

sleep 1;

done

volumeMounts:

- name: varlog

mountPath: /var/log

volumes:

- name: varlog

emptyDir: {}

Create a pod that has two different log streams to the same directory:

kubectl apply -f twolog.yaml

View the logs in the /var/log directory of the container:

kubectl exec counter -- ls /var/log

The YAML for a sidecar container that will tail the logs for each type:

apiVersion: v1

kind: Pod

metadata:

name: counter

spec:

containers:

- name: count

image: busybox

args:

- /bin/sh

- -c

- >

i=0;

while true;

do

echo "$i: $(date)" >> /var/log/1.log;

echo "$(date) INFO $i" >> /var/log/2.log;

i=$((i+1));

sleep 1;

done

volumeMounts:

- name: varlog

mountPath: /var/log

- name: count-log-1

image: busybox

args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']

volumeMounts:

- name: varlog

mountPath: /var/log

- name: count-log-2

image: busybox

args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']

volumeMounts:

- name: varlog

mountPath: /var/log

volumes:

- name: varlog

emptyDir: {}

View the first type of logs separately:

kubectl logs counter count-log-1

View the second type of logs separately:

kubectl logs counter count-log-2

#### Helpful Links

* [Logging](https://kubernetes.io/docs/concepts/cluster-administration/logging/)

## Managing Application Logs

Containerized applications usually write their logs to standard out and standard error instead of writing their logs to files. Docker then redirects those streams to files. You can retrieve those files with the kubectl logs command in Kubernetes. In this lesson, we’ll go over the many ways to manipulate the output of your logs and redirect them to a file.

Get the logs from a pod:

kubectl logs nginx

Get the logs from a specific container on a pod:

kubectl logs counter -c count-log-1

Get the logs from all containers on the pod:

kubectl logs counter --all-containers=true

Get the logs from containers with a certain label:

kubectl logs -lapp=nginx

Get the logs from a previously terminated container within a pod:

kubectl logs -p -c nginx nginx

Stream the logs from a container in a pod:

kubectl logs -f -c count-log-1 counter

Tail the logs to only view a certain number of lines:

kubectl logs --tail=20 nginx

View the logs from a previous time duration:

kubectl logs --since=1h nginx

View the logs from a container within a pod within a deployment:

kubectl logs deployment/nginx -c nginx

Redirect the output of the logs to a file:

kubectl logs counter -c count-log-1 > count.log

#### Helpful Links

* [Logs](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#logs)

## Troubleshooting Application Failure

Application failure can happen for many reasons, but there are ways within Kubernetes that make it a little easier to discover why. In this lesson, we’ll fix some broken pods and show common methods to troubleshoot.

The YAML for a pod with a termination reason:

apiVersion: v1

kind: Pod

metadata:

name: pod2

spec:

containers:

- image: busybox

name: main

command:

- sh

- -c

- 'echo "I''ve had enough" > /var/termination-reason ; exit 1'

terminationMessagePath: /var/termination-reason

One of the first steps in troubleshooting is usually to describe the pod:

kubectl describe po pod2

The YAML for a liveness probe that checks for pod health:

apiVersion: v1

kind: Pod

metadata:

name: liveness

spec:

containers:

- image: linuxacademycontent/candy-service:2

name: kubeserve

livenessProbe:

httpGet:

path: /healthz

port: 8081

View the logs for additional detail:

kubectl logs pod-with-defaults

Export the YAML of a running pod, in the case that you are unable to edit it directly:

kubectl get po pod-with-defaults -o yaml --export > defaults-pod.yaml

Edit a pod directly (i.e., changing the image):

kubectl edit po nginx

#### Helpful Links

* [Pod Liveness & Readiness Probes](https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/)
* [Pod Failure Reasons](https://kubernetes.io/docs/tasks/debug-application-cluster/determine-reason-pod-failure/)
* [Debug the Application](https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application-introspection/)
* [Troubleshoot Applications](https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/)
* [The Pod Lifecycle](https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/)

## Troubleshooting Control Plane Failure

The Kubernetes Control Plane is an important component to back up and protect against failure. There are certain best practices you can take to ensure you don’t have a single point of failure. If your Control Plane components are not effectively communicating, there are a few things you can check to ensure your cluster is operating efficiently.

Check the events in the kube-system namespace for errors:

kubectl get events -n kube-system

Get the logs from the individual pods in your kube-system namespace and check for errors:

kubectl logs [kube\_scheduler\_pod\_name] -n kube-system

Check the status of the Docker service:

sudo systemctl status docker

Start up and enable the Docker service, so it starts upon bootup:

sudo systemctl enable docker && systemctl start docker

Check the status of the kubelet service:

sudo systemctl status kubelet

Start up and enable the kubelet service, so it starts up when the machine is rebooted:

sudo systemctl enable kubelet && systemctl start kubelet

Turn off swap on your machine:

sudo su -

swapoff -a && sed -i '/ swap / s/^/#/' /etc/fstab

Check if you have a firewall running:

sudo systemctl status firewalld

Disable the firewall and stop the firewalld service:

sudo systemctl disable firewalld && systemctl stop firewalld

#### Helpful Links

* [Cluster Failure Overview](https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster/#a-general-overview-of-cluster-failure-modes)
* [Master HA](https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/)

## Troubleshooting Worker Node Failure

Troubleshooting worker node failure is a lot like troubleshooting a non-responsive server, in addition to the kubectl tools we have at our disposal. In this lesson, we’ll learn how to recover a node and add it back to the cluster and find out how to identify when the kublet service is down.

Listing the status of the nodes should be the first step:

kubectl get nodes

Find out more information about the nodes with kubectl describe:

kubectl describe nodes chadcrowell2c.mylabserver.com

You can try to log in to your server via SSH:

ssh chadcrowell2c.mylabserver.com

Get the IP address of your nodes:

kubectl get nodes -o wide

Use the IP address to further probe the server:

ssh cloud\_user@172.31.29.182

Generate a new token after spinning up a new server:

sudo kubeadm token generate

Create the kubeadm join command for your new worker node:

sudo kubeadm token create [token\_name] --ttl 2h --print-join-command

View the journalctl logs:

sudo journalctl -u kubelet

View the syslogs:

sudo more syslog | tail -120 | grep kubelet

* [Nodes](https://kubernetes.io/docs/concepts/architecture/nodes/)
* [Explore Nodes](https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/#nodes)

## Troubleshooting Networking

Network issues usually start to arise internally or when using a service. In this lesson, we’ll go through the many methods to see if your app is serving traffic by creating a service and testing the communication within the cluster.

Run a deployment using the container port 9376 and with three replicas:

kubectl run hostnames --image=k8s.gcr.io/serve\_hostname \

--labels=app=hostnames \

--port=9376 \

--replicas=3

List the services in your cluster:

kubectl get svc

Create a service by exposing a port on the deployment:

kubectl expose deployment hostnames --port=80 --target-port=9376

Run an interactive busybox pod:

kubectl run -it --rm --restart=Never busybox --image=busybox:1.28 sh

From the pod, check if DNS is resolving hostnames:

# nslookup hostnames

From the pod, cat out the /etc/resolv.conf file:

# cat /etc/resolv.conf

From the pod, look up the DNS name of the Kubernetes service:

# nslookup kubernetes.default

Get the JSON output of your service:

kubectl get svc hostnames -o json

View the endpoints for your service:

kubectl get ep

Communicate with the pod directly (without the service):

wget -qO- 10.244.1.6:9376

Check if kube-proxy is running on the nodes:

ps auxw | grep kube-proxy

Check if kube-proxy is writing iptables:

iptables-save | grep hostnames

View the list of kube-system pods:

kubectl get pods -n kube-system

Connect to your kube-proxy pod in the kube-system namespace:

kubectl exec -it kube-proxy-cqptg -n kube-system -- sh

Delete the flannel CNI plugin:

kubectl delete -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml

Apply the Weave Net CNI plugin:

kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

#### Helpful Links

* [Debugging Services in Kubernetes](https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/)
* [Port Forwarding to Access a Pod](https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/)
* [Troubleshooting](https://kubernetes.io/docs/tasks/debug-application-cluster/troubleshooting/)
* [Installing the CNI Plugin](https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/#pod-network)