You can access this page also inside the Remote Desktop by using the icons on the desktop

- Score
- Questions and Answers
- Preview Questions and Answers
- Exam Tips

CKA Simulator Kubernetes 1.26

https://killer.sh

Pre Setup

Once you've gained access to your terminal it might be wise to spend ~1 minute to setup your environment. You could set these:

```
alias k=kubectl                         # will already be pre-configured

export do="--dry-run=client -o yaml"    # k create deploy nginx --image=nginx $do

export now="--force --grace-period 0"   # k delete pod x $now
```

Vim

The following settings will already be configured in your real exam environment in `~/.vimrc`. But it can never hurt to be able to type these down:

```
set tabstop=2
set expandtab
set shiftwidth=2
```

More setup suggestions are in the **tips section**.

Question 1 | Contexts

Task weight: 1%

You have access to multiple clusters from your main terminal through `kubectl` contexts. Write all those context names into `/opt/course/1/contexts`.

Next write a command to display the current context into `/opt/course/1/context_default_kubectl.sh`, the command should use `kubectl`.

Finally write a second command doing the same thing into `/opt/course/1/context_default_no_kubectl.sh`, but without the use of `kubectl`.

Answer:

Maybe the fastest way is just to run:

```
k config get-contexts # copy manually

k config get-contexts -o name > /opt/course/1/contexts
```

Or using jsonpath:

```
k config view -o yaml # overview
k config view -o jsonpath="{.contexts[*].name}"
k config view -o jsonpath="{.contexts[*].name}" | tr " " "\n" # new lines
k config view -o jsonpath="{.contexts[*].name}" | tr " " "\n" > /opt/course/1/contexts
```

The content should then look like:

```
# /opt/course/1/contexts
k8s-c1-H
k8s-c2-AC
k8s-c3-CCC
```

Next create the first command:

```
# /opt/course/1/context_default_kubectl.sh
kubectl config current-context
```

```
➜ sh /opt/course/1/context_default_kubectl.sh
k8s-c1-H
```

And the second one:

```
# /opt/course/1/context_default_no_kubectl.sh
cat ~/.kube/config | grep current
```

```
➜ sh /opt/course/1/context_default_no_kubectl.sh
current-context: k8s-c1-H
```

In the real exam you might need to filter and find information from bigger lists of resources, hence knowing a little jsonpath and simple bash filtering will be helpful.

The second command could also be improved to:

```
# /opt/course/1/context_default_no_kubectl.sh
cat ~/.kube/config | grep current | sed -e "s/current-context: //"
```

Question 2 | Schedule Pod on Controlplane Node

Task weight: 3%

Use context: `kubectl config use-context k8s-c1-H`

Create a single *Pod* of image `httpd:2.4.41-alpine` in *Namespace* `default`. The *Pod* should be named `pod1` and the container should be named `pod1-container`. This *Pod* should **only** be scheduled on a controlplane *node*, do not add new labels any nodes.

Answer:

First we find the controlplane node(s) and their taints:

```
k get node # find controlplane node

k describe node cluster1-controlplane1 | grep Taint -A1 # get controlplane node taints

k get node cluster1-controlplane1 --show-labels # get controlplane node labels
```

Next we create the *Pod* template:

```
# check the export on the very top of this document so we can use $do
k run pod1 --image=httpd:2.4.41-alpine $do > 2.yaml

vim 2.yaml
```

Perform the necessary changes manually. Use the Kubernetes docs and search for example for tolerations and nodeSelector to find examples:

```
# 2.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: pod1
  name: pod1
spec:
  containers:
  - image: httpd:2.4.41-alpine
    name: pod1-container                       # change
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  tolerations:                                 # add
  - effect: NoSchedule                         # add
    key: node-role.kubernetes.io/control-plane # add
  nodeSelector:                                # add
    node-role.kubernetes.io/control-plane: ""  # add
status: {}
```

Important here to add the toleration for running on controlplane nodes, but also the nodeSelector to make sure it only runs on controlplane nodes. If we only specify a toleration the *Pod* can be scheduled on controlplane or worker nodes.

Now we create it:

```
k -f 2.yaml create
```

Let's check if the pod is scheduled:

```
➜ k get pod pod1 -o wide
NAME   READY   STATUS    RESTARTS   ...   NODE                     NOMINATED NODE
pod1   1/1     Running   0          ...   cluster1-controlplane1   <none>
```

Question 3 | Scale down StatefulSet

Task weight: 1%

Use context: `kubectl config use-context k8s-c1-H`

There are two *Pods* named `o3db-*` in *Namespace* `project-c13`. C13 management asked you to scale the *Pods* down to one replica to save resources.

Answer:

If we check the *Pods* we see two replicas:

```
➜ k -n project-c13 get pod | grep o3db
o3db-0                                  1/1     Running   0          52s
o3db-1                                  1/1     Running   0          42s
```

From their name it looks like these are managed by a *StatefulSet*. But if we're not sure we could also check for the most common resources which manage *Pods*:

```
➜ k -n project-c13 get deploy,ds,sts | grep o3db
statefulset.apps/o3db   2/2     2m56s
```

Confirmed, we have to work with a *StatefulSet*. To find this out we could also look at the *Pod* labels:

```
➜ k -n project-c13 get pod --show-labels | grep o3db
o3db-0                                  1/1     Running   0          3m29s   app=nginx,controller-revision-hash=o3db-
5fbd4bb9cc,statefulset.kubernetes.io/pod-name=o3db-0
o3db-1                                  1/1     Running   0          3m19s   app=nginx,controller-revision-hash=o3db-
5fbd4bb9cc,statefulset.kubernetes.io/pod-name=o3db-1
```

To fulfil the task we simply run:

```
➜ k -n project-c13 scale sts o3db --replicas 1
statefulset.apps/o3db scaled

➜ k -n project-c13 get sts o3db
NAME   READY   AGE
o3db   1/1     4m39s
```

C13 Mangement is happy again.

Question 4 | Pod Ready if Service is reachable

Task weight: 4%

Use context: `kubectl config use-context k8s-c1-H`

Do the following in *Namespace* `default`. Create a single *Pod* named `ready-if-service-ready` of image `nginx:1.16.1-alpine`. Configure a LivenessProbe which simply executes command `true`. Also configure a ReadinessProbe which does check if the url `http://service-am-i-ready:80` is reachable, you can use `wget -T2 -O- http://service-am-i-ready:80` for this. Start the *Pod* and confirm it isn't ready because of the ReadinessProbe.

Create a second *Pod* named `am-i-ready` of image `nginx:1.16.1-alpine` with label `id: cross-server-ready`. The already existing *Service* `service-am-i-ready` should now have that second *Pod* as endpoint.

Now the first *Pod* should be in ready state, confirm that.

Answer:

It's a bit of an anti-pattern for one *Pod* to check another *Pod* for being ready using probes, hence the normally available `readinessProbe.httpGet` doesn't work for absolute remote urls. Still the workaround requested in this task should show how probes and *Pod*<->*Service* communication works.

First we create the first *Pod*:

```
k run ready-if-service-ready --image=nginx:1.16.1-alpine $do > 4_pod1.yaml

vim 4_pod1.yaml
```

Next perform the necessary additions manually:

```
# 4_pod1.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: ready-if-service-ready
  name: ready-if-service-ready
spec:
  containers:
  - image: nginx:1.16.1-alpine
    name: ready-if-service-ready
    resources: {}
    livenessProbe:                                      # add from here
      exec:
        command:
        - 'true'
    readinessProbe:
      exec:
        command:
        - sh
        - -c
        - 'wget -T2 -O- http://service-am-i-ready:80'   # to here
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Then create the *Pod*:

```
k -f 4_pod1.yaml create
```

And confirm it's in a non-ready state:

```
➜ k get pod ready-if-service-ready
NAME                     READY   STATUS    RESTARTS   AGE
ready-if-service-ready   0/1     Running   0          7s
```

We can also check the reason for this using describe:

```
➜ k describe pod ready-if-service-ready
 ...
  Warning  Unhealthy  18s   kubelet, cluster1-node1  Readiness probe failed: Connecting to service-am-i-ready:80
(10.109.194.234:80)
wget: download timed out
```

Now we create the second *Pod*:

```
k run am-i-ready --image=nginx:1.16.1-alpine --labels="id=cross-server-ready"
```

The already existing *Service* `service-am-i-ready` should now have an *Endpoint*:

```
k describe svc service-am-i-ready
k get ep # also possible
```

Which will result in our first *Pod* being ready, just give it a minute for the Readiness probe to check again:

```
➜ k get pod ready-if-service-ready
NAME                     READY   STATUS    RESTARTS   AGE
ready-if-service-ready   1/1     Running   0          53s
```

Look at these *Pods* coworking together!

Question 5 | Kubectl sorting

Task weight: 1%

Use context: `kubectl config use-context k8s-c1-H`

There are various *Pods* in all namespaces. Write a command into `/opt/course/5/find_pods.sh` which lists all *Pods* sorted by their AGE (`metadata.creationTimestamp`).

Write a second command into `/opt/course/5/find_pods_uid.sh` which lists all *Pods* sorted by field `metadata.uid`. Use `kubectl` sorting for both commands.

Answer:

A good resources here (and for many other things) is the kubectl-cheat-sheet. You can reach it fast when searching for "cheat sheet" in the Kubernetes docs.

```
# /opt/course/5/find_pods.sh
kubectl get pod -A --sort-by=.metadata.creationTimestamp
```

And to execute:

```
➜ sh /opt/course/5/find_pods.sh
NAMESPACE         NAME                                          ...          AGE
kube-system       kube-scheduler-cluster1-controlplane1         ...          63m
kube-system       etcd-cluster1-controlplane1                   ...          63m
kube-system       kube-apiserver-cluster1-controlplane1         ...          63m
kube-system       kube-controller-manager-cluster1-controlplane1   ...          63m
...
```

For the second command:

```
# /opt/course/5/find_pods_uid.sh
kubectl get pod -A --sort-by=.metadata.uid
```

And to execute:

```
➜ sh /opt/course/5/find_pods_uid.sh
NAMESPACE         NAME                                      ...          AGE
kube-system       coredns-5644d7b6d9-vwm7g                  ...          68m
project-c13       c13-3cc-runner-heavy-5486d76dd4-ddvlt     ...          63m
project-hamster   web-hamster-shop-849966f479-278vp         ...          63m
project-c13       c13-3cc-web-646b6c8756-qsg4b              ...          63m
```

Question 6 | Storage, PV, PVC, Pod volume

Task weight: 8%

Use context: `kubectl config use-context k8s-c1-H`

Create a new *PersistentVolume* named `safari-pv`. It should have a capacity of *2Gi*, accessMode *ReadWriteOnce*, hostPath `/Volumes/Data` and no storageClassName defined.

Next create a new *PersistentVolumeClaim* in *Namespace* `project-tiger` named `safari-pvc` . It should request *2Gi* storage, accessMode *ReadWriteOnce* and should not define a storageClassName. The *PVC* should bound to the *PV* correctly.

Finally create a new *Deployment* `safari` in *Namespace* `project-tiger` which mounts that volume at `/tmp/safari-data`. The *Pods* of that *Deployment* should be of image `httpd:2.4.41-alpine`.

Answer

```
vim 6_pv.yaml
```

Find an example from https://kubernetes.io/docs and alter it:

```
# 6_pv.yaml
kind: PersistentVolume
apiVersion: v1
metadata:
 name: safari-pv
spec:
 capacity:
  storage: 2Gi
 accessModes:
  - ReadWriteOnce
 hostPath:
  path: "/Volumes/Data"
```

Then create it:

```
k -f 6_pv.yaml create
```

Next the *PersistentVolumeClaim*:

```
vim 6_pvc.yaml
```

Find an example from https://kubernetes.io/docs and alter it:

```
# 6_pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: safari-pvc
  namespace: project-tiger
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
     storage: 2Gi
```

Then create:

```
k -f 6_pvc.yaml create
```

And check that both have the status Bound:

```
➜ k -n project-tiger get pv,pvc
NAME                         CAPACITY  ... STATUS   CLAIM                    ...
persistentvolume/safari-pv   2Gi       ... Bound    project-tiger/safari-pvc ...

NAME                               STATUS   VOLUME      CAPACITY ...
persistentvolumeclaim/safari-pvc   Bound    safari-pv   2Gi      ...
```

Next we create a *Deployment* and mount that volume:

```
k -n project-tiger create deploy safari \
  --image=httpd:2.4.41-alpine $do > 6_dep.yaml

vim 6_dep.yaml
```

Alter the yaml to mount the volume:

```
# 6_dep.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: safari
  name: safari
  namespace: project-tiger
spec:
  replicas: 1
  selector:
    matchLabels:
      app: safari
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: safari
    spec:
      volumes:                                      # add
      - name: data                                  # add
        persistentVolumeClaim:                      # add
          claimName: safari-pvc                     # add
      containers:
      - image: httpd:2.4.41-alpine
```

```
        name: container
        volumeMounts:                               # add
        - name: data                                # add
          mountPath: /tmp/safari-data               # add
```

```
k -f 6_dep.yaml create
```

We can confirm it's mounting correctly:

```
➜ k -n project-tiger describe pod safari-5cbf46d6d-mjhsb  | grep -A2 Mounts:
    Mounts:
      /tmp/safari-data from data (rw) # there it is
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-n2sjj (ro)
```

Question 7 | Node and Pod Resource Usage

Task weight: 1%

Use context: `kubectl config use-context k8s-c1-H`

The metrics-server has been installed in the cluster. Your college would like to know the kubectl commands to:

1. show *Nodes* resource usage
2. show *Pods* and their containers resource usage

Please write the commands into `/opt/course/7/node.sh` and `/opt/course/7/pod.sh`.

Answer:

The command we need to use here is top:

```
➜ k top -h
Display Resource (CPU/Memory/Storage) usage.

 The top command allows you to see the resource consumption for nodes or pods.

 This command requires Metrics Server to be correctly configured and working on the server.

Available Commands:
  node        Display Resource (CPU/Memory/Storage) usage of nodes
  pod         Display Resource (CPU/Memory/Storage) usage of pods
```

We see that the metrics server provides information about resource usage:

```
➜ k top node
NAME               CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
cluster1-controlplane1   178m         8%     1091Mi          57%
cluster1-node1   66m          6%     834Mi           44%
cluster1-node2   91m          9%     791Mi           41%
```

We create the first file:

```
# /opt/course/7/node.sh
kubectl top node
```

For the second file we might need to check the docs again:

```
➜ k top pod -h
Display Resource (CPU/Memory/Storage) usage of pods.
...
Namespace in current context is ignored even if specified with --namespace.
      --containers=false: If present, print usage of containers within a pod.
      --no-headers=false: If present, print output without headers.
...
```

With this we can finish this task:

```
# /opt/course/7/pod.sh
kubectl top pod --containers=true
```

Question 8 | Get Controlplane Information

Task weight: 2%

Use context: `kubectl config use-context k8s-c1-H`

Ssh into the controlplane node with `ssh cluster1-controlplane1`. Check how the controlplane components kubelet, kube-apiserver, kube-scheduler, kube-controller-manager and etcd are started/installed on the controlplane node. Also find out the name of the DNS application and how it's started/installed on the controlplane node.

Write your findings into file `/opt/course/8/controlplane-components.txt`. The file should be structured like:

```
# /opt/course/8/controlplane-components.txt
kubelet: [TYPE]
kube-apiserver: [TYPE]
kube-scheduler: [TYPE]
kube-controller-manager: [TYPE]
etcd: [TYPE]
dns: [TYPE] [NAME]
```

Choices of `[TYPE]` are: `not-installed`, `process`, `static-pod`, `pod`

Answer:

We could start by finding processes of the requested components, especially the kubelet at first:

```
➜ ssh cluster1-controlplane1

root@cluster1-controlplane1:~# ps aux | grep kubelet # shows kubelet process
```

We can see which components are controlled via systemd looking at `/etc/systemd/system` directory:

```
➜ root@cluster1-controlplane1:~# find /etc/systemd/system/ | grep kube
/etc/systemd/system/kubelet.service.d
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf
/etc/systemd/system/multi-user.target.wants/kubelet.service

➜ root@cluster1-controlplane1:~# find /etc/systemd/system/ | grep etcd
```

This shows kubelet is controlled via systemd, but no other service named kube nor etcd. It seems that this cluster has been setup using kubeadm, so we check in the default manifests directory:

```
➜ root@cluster1-controlplane1:~# find /etc/kubernetes/manifests/
/etc/kubernetes/manifests/
/etc/kubernetes/manifests/kube-controller-manager.yaml
/etc/kubernetes/manifests/etcd.yaml
/etc/kubernetes/manifests/kube-apiserver.yaml
/etc/kubernetes/manifests/kube-scheduler.yaml
```

(The kubelet could also have a different manifests directory specified via parameter `--pod-manifest-path` in it's systemd startup config)

This means the main 4 controlplane services are setup as static *Pods*. Actually, let's check all *Pods* running on in the `kube-system` *Namespace* on the controlplane node:

```
➜ root@cluster1-controlplane1:~# kubectl -n kube-system get pod -o wide | grep controlplane1
coredns-5644d7b6d9-c4f68                            1/1     Running           ...   cluster1-controlplane1
coredns-5644d7b6d9-t84sc                            1/1     Running           ...   cluster1-controlplane1
etcd-cluster1-controlplane1                         1/1     Running           ...   cluster1-controlplane1
kube-apiserver-cluster1-controlplane1               1/1     Running           ...   cluster1-controlplane1
kube-controller-manager-cluster1-controlplane1      1/1     Running           ...   cluster1-controlplane1
kube-proxy-q955p                                    1/1     Running           ...   cluster1-controlplane1
kube-scheduler-cluster1-controlplane1               1/1     Running           ...   cluster1-controlplane1
weave-net-mwj47                                     2/2     Running           ...   cluster1-controlplane1
```

There we see the 5 static pods, with `-cluster1-controlplane1` as suffix.

We also see that the dns application seems to be coredns, but how is it controlled?

```
➜ root@cluster1-controlplane1$ kubectl -n kube-system get ds
NAME         DESIRED   CURRENT   ...   NODE SELECTOR            AGE
kube-proxy   3         3         ...   kubernetes.io/os=linux   155m
weave-net    3         3         ...   <none>                   155m

➜ root@cluster1-controlplane1$ kubectl -n kube-system get deploy
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
coredns   2/2     2            2           155m
```

Seems like coredns is controlled via a *Deployment*. We combine our findings in the requested file:

```
# /opt/course/8/controlplane-components.txt
kubelet: process
kube-apiserver: static-pod
kube-scheduler: static-pod
kube-controller-manager: static-pod
etcd: static-pod
dns: pod coredns
```

You should be comfortable investigating a running cluster, know different methods on how a cluster and its services can be setup and be able to troubleshoot and find error sources.

Question 9 | Kill Scheduler, Manual Scheduling

Task weight: 5%

Use context: `kubectl config use-context k8s-c2-AC`

Ssh into the controlplane node with `ssh cluster2-controlplane1`. **Temporarily** stop the kube-scheduler, this means in a way that you can start it again afterwards.

Create a single *Pod* named `manual-schedule` of image `httpd:2.4-alpine`, confirm it's created but not scheduled on any node.

Now you're the scheduler and have all its power, manually schedule that *Pod* on node cluster2-controlplane1. Make sure it's running.

Start the kube-scheduler again and confirm it's running correctly by creating a second *Pod* named `manual-schedule2` of image `httpd:2.4-alpine` and check if it's running on cluster2-node1.

Answer:

Stop the Scheduler

First we find the controlplane node:

```
➜ k get node
NAME                     STATUS   ROLES           AGE   VERSION
cluster2-controlplane1   Ready    control-plane   26h   v1.26.0
cluster2-node1           Ready    <none>          26h   v1.26.0
```

Then we connect and check if the scheduler is running:

```
➜ ssh cluster2-controlplane1

➜ root@cluster2-controlplane1:~# kubectl -n kube-system get pod | grep schedule
kube-scheduler-cluster2-controlplane1            1/1     Running   0          6s
```

Kill the Scheduler (temporarily):

```
➜ root@cluster2-controlplane1:~# cd /etc/kubernetes/manifests/

➜ root@cluster2-controlplane1:~# mv kube-scheduler.yaml ..
```

And it should be stopped:

```
➜ root@cluster2-controlplane1:~# kubectl -n kube-system get pod | grep schedule

➜ root@cluster2-controlplane1:~#
```

Create a *Pod*

Now we create the *Pod*:

```
k run manual-schedule --image=httpd:2.4-alpine
```

And confirm it has no node assigned:

```
➜ k get pod manual-schedule -o wide
NAME              READY   STATUS    ...   NODE     NOMINATED NODE
manual-schedule   0/1     Pending   ...   <none>   <none>
```

Manually schedule the *Pod*

Let's play the scheduler now:

```
k get pod manual-schedule -o yaml > 9.yaml
```

```
# 9.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2020-09-04T15:51:02Z"
  labels:
    run: manual-schedule
  managedFields:
...
    manager: kubectl-run
    operation: Update
    time: "2020-09-04T15:51:02Z"
  name: manual-schedule
  namespace: default
  resourceVersion: "3515"
  selfLink: /api/v1/namespaces/default/pods/manual-schedule
  uid: 8e9d2532-4779-4e63-b5af-feb82c74a935
spec:
  nodeName: cluster2-controlplane1        # add the controlplane node name
  containers:
  - image: httpd:2.4-alpine
    imagePullPolicy: IfNotPresent
    name: manual-schedule
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-nxnc7
      readOnly: true
  dnsPolicy: ClusterFirst
...
```

The only thing a scheduler does, is that it sets the nodeName for a *Pod* declaration. How it finds the correct node to schedule on, that's a very much complicated matter and takes many variables into account.

As we cannot `kubectl apply` or `kubectl edit` , in this case we need to delete and create or replace:

```
k -f 9.yaml replace --force
```

How does it look?

```
➜ k get pod manual-schedule -o wide
NAME              READY   STATUS    ...   NODE
manual-schedule   1/1     Running   ...   cluster2-controlplane1
```

It looks like our *Pod* is running on the controlplane now as requested, although no tolerations were specified. Only the scheduler takes tains/tolerations/affinity into account when finding the correct node name. That's why it's still possible to assign *Pods* manually directly to a controlplane node and skip the scheduler.

Start the scheduler again

```
➜ ssh cluster2-controlplane1

➜ root@cluster2-controlplane1:~# cd /etc/kubernetes/manifests/

➜ root@cluster2-controlplane1:~# mv ../kube-scheduler.yaml .
```

Checks it's running:

```
➜ root@cluster2-controlplane1:~# kubectl -n kube-system get pod | grep schedule
kube-scheduler-cluster2-controlplane1            1/1     Running   0          16s
```

Schedule a second test *Pod*:

```
k run manual-schedule2 --image=httpd:2.4-alpine
```

```
➜ k get pod -o wide | grep schedule
manual-schedule    1/1     Running   ...   cluster2-controlplane1
manual-schedule2   1/1     Running   ...   cluster2-node1
```

Back to normal.

Question 10 | RBAC ServiceAccount Role RoleBinding

Task weight: 6%

Use context: `kubectl config use-context k8s-c1-H`

Create a new *ServiceAccount* `processor` in *Namespace* `project-hamster`. Create a *Role* and *RoleBinding*, both named `processor` as well. These should allow the new *SA* to only create *Secrets* and *ConfigMaps* in that *Namespace*.

Answer:

Let's talk a little about RBAC resources

A *ClusterRole*|*Role* defines a set of permissions and **where it is available**, in the whole cluster or just a single *Namespace*.

A *ClusterRoleBinding*|*RoleBinding* connects a set of permissions with an account and defines **where it is applied**, in the whole cluster or just a single *Namespace*.

Because of this there are 4 different RBAC combinations and 3 valid ones:

1. *Role* + *RoleBinding* (available in single *Namespace*, applied in single *Namespace*)
2. *ClusterRole* + *ClusterRoleBinding* (available cluster-wide, applied cluster-wide)
3. *ClusterRole* + *RoleBinding* (available cluster-wide, applied in single *Namespace*)
4. *Role* + *ClusterRoleBinding* (**NOT POSSIBLE:** available in single *Namespace*, applied cluster-wide)

To the solution

We first create the *ServiceAccount*:

```
➜ k -n project-hamster create sa processor
serviceaccount/processor created
```

Then for the *Role*:

```
k -n project-hamster create role -h # examples
```

So we execute:

```
k -n project-hamster create role processor \
  --verb=create \
  --resource=secret \
  --resource=configmap
```

Which will create a *Role* like:

```
# kubectl -n project-hamster create role processor --verb=create --resource=secret --resource=configmap
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: processor
  namespace: project-hamster
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  - configmaps
  verbs:
  - create
```

Now we bind the *Role* to the *ServiceAccount*:

```
k -n project-hamster create rolebinding -h # examples
```

So we create it:

```
k -n project-hamster create rolebinding processor \
  --role processor \
  --serviceaccount project-hamster:processor
```

This will create a *RoleBinding* like:

```
# kubectl -n project-hamster create rolebinding processor --role processor --serviceaccount project-hamster:processor
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: processor
  namespace: project-hamster
```

```
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: processor
subjects:
- kind: ServiceAccount
  name: processor
  namespace: project-hamster
```

To test our RBAC setup we can use `kubectl auth can-i`:

```
k auth can-i -h # examples
```

Like this:

```
➜ k -n project-hamster auth can-i create secret \
  --as system:serviceaccount:project-hamster:processor
yes

➜ k -n project-hamster auth can-i create configmap \
  --as system:serviceaccount:project-hamster:processor
yes

➜ k -n project-hamster auth can-i create pod \
  --as system:serviceaccount:project-hamster:processor
no

➜ k -n project-hamster auth can-i delete secret \
  --as system:serviceaccount:project-hamster:processor
no

➜ k -n project-hamster auth can-i get configmap \
  --as system:serviceaccount:project-hamster:processor
no
```

Done.

Question 11 | DaemonSet on all Nodes

Task weight: 4%

Use context: `kubectl config use-context k8s-c1-H`

Use *Namespace* `project-tiger` for the following. Create a *DaemonSet* named `ds-important` with image `httpd:2.4-alpine` and labels `id=ds-important` and `uuid=18426a0b-5f59-4e10-923f-c0e078e82462`. The *Pods* it creates should request 10 millicore cpu and 10 mebibyte memory. The *Pods* of that *DaemonSet* should run on all nodes, also controlplanes.

Answer:

As of now we aren't able to create a *DaemonSet* directly using `kubectl`, so we create a *Deployment* and just change it up:

```
k -n project-tiger create deployment --image=httpd:2.4-alpine ds-important $do > 11.yaml

vim 11.yaml
```

(Sure you could also search for a *DaemonSet* example yaml in the Kubernetes docs and alter it.)

Then we adjust the yaml to:

```
# 11.yaml
apiVersion: apps/v1
kind: DaemonSet                                     # change from Deployment to Daemonset
metadata:
  creationTimestamp: null
  labels:                                           # add
    id: ds-important                                # add
    uuid: 18426a0b-5f59-4e10-923f-c0e078e82462      # add
  name: ds-important
  namespace: project-tiger                          # important
spec:
  #replicas: 1                                      # remove
  selector:
    matchLabels:
      id: ds-important                              # add
      uuid: 18426a0b-5f59-4e10-923f-c0e078e82462    # add
  #strategy: {}                                     # remove
  template:
    metadata:
      creationTimestamp: null
      labels:
        id: ds-important                            # add
        uuid: 18426a0b-5f59-4e10-923f-c0e078e82462  # add
    spec:
      containers:
      - image: httpd:2.4-alpine
        name: ds-important
        resources:
          requests:                                 # add
            cpu: 10m                                # add
            memory: 10Mi                            # add
      tolerations:                                  # add
      - effect: NoSchedule                          # add
        key: node-role.kubernetes.io/control-plane  # add
#status: {}                                         # remove
```

It was requested that the *DaemonSet* runs on all nodes, so we need to specify the toleration for this.

Let's confirm:

```
k -f 11.yaml create
```

```
➜ k -n project-tiger get ds
NAME           DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
ds-important   3         3         3       3            3           <none>          8s
```

```
➜ k -n project-tiger get pod -l id=ds-important -o wide
NAME                      READY   STATUS          NODE
ds-important-6pvgm        1/1     Running   ...   cluster1-node1
ds-important-lh5ts        1/1     Running   ...   cluster1-controlplane1
ds-important-qhjcq        1/1     Running   ...   cluster1-node2
```

Question 12 | Deployment on all Nodes

Task weight: 6%

Use context: `kubectl config use-context k8s-c1-H`

Use *Namespace* `project-tiger` for the following. Create a *Deployment* named `deploy-important` with label `id=very-important` (the `Pods` should also have this label) and 3 replicas. It should contain two containers, the first named `container1` with image `nginx:1.17.6-alpine` and the second one named container2 with image `kubernetes/pause`.

There should be only ever **one** *Pod* of that *Deployment* running on **one** worker node. We have two worker nodes: `cluster1-node1` and `cluster1-node2`. Because the *Deployment* has three replicas the result should be that on both nodes **one** *Pod* is running. The third *Pod* won't be scheduled, unless a new worker node will be added.

In a way we kind of simulate the behaviour of a *DaemonSet* here, but using a *Deployment* and a fixed number of replicas.

Answer:

There are two possible ways, one using `podAntiAffinity` and one using `topologySpreadConstraint`.

PodAntiAffinity

The idea here is that we create a "Inter-pod anti-affinity" which allows us to say a *Pod* should only be scheduled on a node where another *Pod* of a specific label (here the same label) is not already running.

Let's begin by creating the *Deployment* template:

```
k -n project-tiger create deployment \
  --image=nginx:1.17.6-alpine deploy-important $do > 12.yaml

vim 12.yaml
```

Then change the yaml to:

```
# 12.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    id: very-important                  # change
  name: deploy-important
  namespace: project-tiger              # important
spec:
  replicas: 3                           # change
  selector:
    matchLabels:
      id: very-important                # change
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        id: very-important              # change
    spec:
      containers:
      - image: nginx:1.17.6-alpine
        name: container1                # change
        resources: {}
      - image: kubernetes/pause         # add
        name: container2                # add
      affinity:                                             # add
        podAntiAffinity:                                    # add
          requiredDuringSchedulingIgnoredDuringExecution:   # add
          - labelSelector:                                  # add
              matchExpressions:                             # add
              - key: id                                     # add
                operator: In                                # add
                values:                                     # add
                - very-important                            # add
            topologyKey: kubernetes.io/hostname             # add
status: {}
```

Specify a topologyKey, which is a pre-populated Kubernetes label, you can find this by describing a node.

TopologySpreadConstraints

We can achieve the same with `topologySpreadConstraints`. Best to try out and play with both.

```
# 12.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    id: very-important                  # change
  name: deploy-important
  namespace: project-tiger              # important
spec:
  replicas: 3                           # change
  selector:
    matchLabels:
      id: very-important                # change
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        id: very-important              # change
    spec:
      containers:
      - image: nginx:1.17.6-alpine
        name: container1                # change
        resources: {}
      - image: kubernetes/pause         # add
        name: container2                # add
      topologySpreadConstraints:                 # add
      - maxSkew: 1                               # add
        topologyKey: kubernetes.io/hostname      # add
        whenUnsatisfiable: DoNotSchedule         # add
        labelSelector:                           # add
          matchLabels:                           # add
            id: very-important                   # add
status: {}
```

Apply and Run

Let's run it:

```
k -f 12.yaml create
```

Then we check the *Deployment* status where it shows 2/3 ready count:

```
➜ k -n project-tiger get deploy -l id=very-important
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
deploy-important   2/3     3            2           2m35s
```

And running the following we see one *Pod* on each worker node and one not scheduled.

```
➜ k -n project-tiger get pod -o wide -l id=very-important
NAME                                READY   STATUS    ...   NODE
deploy-important-58db9db6fc-9ljpw   2/2     Running   ...   cluster1-node1
deploy-important-58db9db6fc-lnxdb   0/2     Pending   ...   <none>
deploy-important-58db9db6fc-p2rz8   2/2     Running   ...   cluster1-node2
```

If we kubectl describe the *Pod* `deploy-important-58db9db6fc-lnxdb` it will show us the reason for not scheduling is our implemented podAntiAffinity ruling:

```
Warning  FailedScheduling  63s (x3 over 65s)  default-scheduler  0/3 nodes are available: 1 node(s) had taint {node-role.kubernetes.io/control-plane: }, that the pod didn't tolerate, 2 node(s) didn't match pod affinity/anti-affinity, 2 node(s) didn't satisfy existing pods anti-affinity rules.
```

Or our topologySpreadConstraints:

```
Warning  FailedScheduling  16s   default-scheduler  0/3 nodes are available: 1 node(s) had taint {node-role.kubernetes.io/control-plane: }, that the pod didn't tolerate, 2 node(s) didn't match pod topology spread constraints.
```

Question 13 | Multi Containers and Pod shared Volume

Task weight: 4%

Use context: `kubectl config use-context k8s-c1-H`

Create a *Pod* named `multi-container-playground` in *Namespace* `default` with three containers, named `c1`, `c2` and `c3`. There should be a volume attached to that *Pod* and mounted into every container, but the volume shouldn't be persisted or shared with other *Pods*.

Container `c1` should be of image `nginx:1.17.6-alpine` and have the name of the node where its *Pod* is running available as environment variable MY_NODE_NAME.

Container `c2` should be of image `busybox:1.31.1` and write the output of the `date` command every second in the shared volume into file `date.log`. You can use `while true; do date >> /your/vol/path/date.log; sleep 1; done` for this.

Container `c3` should be of image `busybox:1.31.1` and constantly send the content of file `date.log` from the shared volume to stdout. You can use `tail -f /your/vol/path/date.log` for this.

Check the logs of container `c3` to confirm correct setup.

Answer:

First we create the *Pod* template:

```
k run multi-container-playground --image=nginx:1.17.6-alpine $do > 13.yaml

vim 13.yaml
```

And add the other containers and the commands they should execute:

```
# 13.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: multi-container-playground
  name: multi-container-playground
spec:
  containers:
  - image: nginx:1.17.6-alpine
    name: c1                                                                     # change
    resources: {}
    env:                                                                         # add
    - name: MY_NODE_NAME                                                         # add
      valueFrom:                                                                 # add
        fieldRef:                                                                # add
          fieldPath: spec.nodeName                                               # add
    volumeMounts:                                                                # add
    - name: vol                                                                  # add
      mountPath: /vol                                                            # add
  - image: busybox:1.31.1                                                        # add
    name: c2                                                                     # add
    command: ["sh", "-c", "while true; do date >> /vol/date.log; sleep 1; done"] # add
    volumeMounts:                                                                # add
    - name: vol                                                                  # add
```

```
      mountPath: /vol                                                              # add
  - image: busybox:1.31.1                                                          # add
    name: c3                                                                       # add
    command: ["sh", "-c", "tail -f /vol/date.log"]                                 # add
    volumeMounts:                                                                  # add
    - name: vol                                                                    # add
      mountPath: /vol                                                              # add
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  volumes:                                                                         # add
    - name: vol                                                                    # add
      emptyDir: {}                                                                 # add
status: {}
```

```
k -f 13.yaml create
```

Oh boy, lot's of requested things. We check if everything is good with the *Pod*:

```
➜ k get pod multi-container-playground
NAME                         READY   STATUS    RESTARTS   AGE
multi-container-playground   3/3     Running   0          95s
```

Good, then we check if container c1 has the requested node name as env variable:

```
➜ k exec multi-container-playground -c c1 -- env | grep MY
MY_NODE_NAME=cluster1-node2
```

And finally we check the logging:

```
➜ k logs multi-container-playground -c c3
Sat Dec  7 16:05:10 UTC 2077
Sat Dec  7 16:05:11 UTC 2077
Sat Dec  7 16:05:12 UTC 2077
Sat Dec  7 16:05:13 UTC 2077
Sat Dec  7 16:05:14 UTC 2077
Sat Dec  7 16:05:15 UTC 2077
Sat Dec  7 16:05:16 UTC 2077
```

Question 14 | Find out Cluster Information

Task weight: 2%

Use context: `kubectl config use-context k8s-c1-H`

You're ask to find out following information about the cluster `k8s-c1-H`:

1. How many controlplane nodes are available?
2. How many worker nodes are available?
3. What is the Service CIDR?
4. Which Networking (or CNI Plugin) is configured and where is its config file?
5. Which suffix will static pods have that run on cluster1-node1?

Write your answers into file `/opt/course/14/cluster-info`, structured like this:

```
# /opt/course/14/cluster-info
1: [ANSWER]
2: [ANSWER]
3: [ANSWER]
4: [ANSWER]
5: [ANSWER]
```

Answer:

How many controlplane and worker nodes are available?

```
➜ k get node
NAME                    STATUS   ROLES           AGE   VERSION
cluster1-controlplane1  Ready    control-plane   27h   v1.26.0
cluster1-node1          Ready    <none>          27h   v1.26.0
cluster1-node2          Ready    <none>          27h   v1.26.0
```

We see one controlplane and two workers.

What is the Service CIDR?

```
➜ ssh cluster1-controlplane1

➜ root@cluster1-controlplane1:~# cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep range
    - --service-cluster-ip-range=10.96.0.0/12
```

Which Networking (or CNI Plugin) is configured and where is its config file?

```
➜ root@cluster1-controlplane1:~# find /etc/cni/net.d/
/etc/cni/net.d/
/etc/cni/net.d/10-weave.conflist

➜ root@cluster1-controlplane1:~# cat /etc/cni/net.d/10-weave.conflist
{
    "cniVersion": "0.3.0",
    "name": "weave",
...
```

By default the kubelet looks into `/etc/cni/net.d` to discover the CNI plugins. This will be the same on every controlplane and worker nodes.

Which suffix will static pods have that run on cluster1-node1?

The suffix is the node hostname with a leading hyphen. It used to be `-static` in earlier Kubernetes versions.

Result

The resulting `/opt/course/14/cluster-info` could look like:

```
# /opt/course/14/cluster-info

# How many controlplane nodes are available?
1: 1

# How many worker nodes are available?
2: 2

# What is the Service CIDR?
3: 10.96.0.0/12

# Which Networking (or CNI Plugin) is configured and where is its config file?
4: Weave, /etc/cni/net.d/10-weave.conflist

# Which suffix will static pods have that run on cluster1-node1?
5: -cluster1-node1
```

Question 15 | Cluster Event Logging

Task weight: 3%

Use context: `kubectl config use-context k8s-c2-AC`

Write a command into `/opt/course/15/cluster_events.sh` which shows the latest events in the whole cluster, ordered by time (`metadata.creationTimestamp`). Use `kubectl` for it.

Now kill the kube-proxy *Pod* running on node cluster2-node1 and write the events this caused into `/opt/course/15/pod_kill.log`.

Finally kill the containerd container of the kube-proxy *Pod* on node cluster2-node1 and write the events into `/opt/course/15/container_kill.log`.

Do you notice differences in the events both actions caused?

Answer:

```
# /opt/course/15/cluster_events.sh
kubectl get events -A --sort-by=.metadata.creationTimestamp
```

Now we kill the kube-proxy *Pod*:

```
k -n kube-system get pod -o wide | grep proxy # find pod running on cluster2-node1

k -n kube-system delete pod kube-proxy-z64cg
```

Now check the events:

```
sh /opt/course/15/cluster_events.sh
```

Write the events the killing caused into `/opt/course/15/pod_kill.log`:

```
# /opt/course/15/pod_kill.log
kube-system   9s          Normal    Killing           pod/kube-proxy-jsv7t   ...
kube-system   3s          Normal    SuccessfulCreate  daemonset/kube-proxy   ...
kube-system   <unknown>   Normal    Scheduled         pod/kube-proxy-m52sx   ...
default       2s          Normal    Starting          node/cluster2-node1  ...
kube-system   2s          Normal    Created           pod/kube-proxy-m52sx   ...
kube-system   2s          Normal    Pulled            pod/kube-proxy-m52sx   ...
kube-system   2s          Normal    Started           pod/kube-proxy-m52sx   ...
```

Finally we will try to provoke events by killing the container belonging to the container of the kube-proxy *Pod*:

```
➜ ssh cluster2-node1

➜ root@cluster2-node1:~# crictl ps | grep kube-proxy
1e020b43c4423   36c4ebbc9d979   About an hour ago   Running   kube-proxy     ...

➜ root@cluster2-node1:~# crictl rm 1e020b43c4423
1e020b43c4423

➜ root@cluster2-node1:~# crictl ps | grep kube-proxy
0ae4245707910   36c4ebbc9d979   17 seconds ago      Running   kube-proxy     ...
```

We killed the main container (1e020b43c4423), but also noticed that a new container (0ae4245707910) was directly created. Thanks Kubernetes!

Now we see if this caused events again and we write those into the second file:

```
sh /opt/course/15/cluster_events.sh
```

```
# /opt/course/15/container_kill.log
kube-system   13s         Normal    Created      pod/kube-proxy-m52sx    ...
kube-system   13s         Normal    Pulled       pod/kube-proxy-m52sx    ...
kube-system   13s         Normal    Started      pod/kube-proxy-m52sx    ...
```

Comparing the events we see that when we deleted the whole *Pod* there were more things to be done, hence more events. For example was the *DaemonSet* in the game to re-create the missing *Pod*. Where when we manually killed the main container of the *Pod*, the *Pod* would still exist but only its container needed to be re-created, hence less events.

Question 16 | Namespaces and Api Resources

Task weight: 2%

Use context: `kubectl config use-context k8s-c1-H`

Write the names of all namespaced Kubernetes resources (like *Pod*, *Secret*, *ConfigMap*...) into `/opt/course/16/resources.txt`.

Find the `project-*` *Namespace* with the highest number of `Roles` defined in it and write its name and amount of *Roles* into `/opt/course/16/crowded-namespace.txt`.

Answer:

Namespace and Namespaces Resources

Now we can get a list of all resources like:

```
k api-resources    # shows all

k api-resources -h # help always good

k api-resources --namespaced -o name > /opt/course/16/resources.txt
```

Which results in the file:

```
# /opt/course/16/resources.txt
bindings
configmaps
endpoints
events
limitranges
persistentvolumeclaims
pods
podtemplates
replicationcontrollers
resourcequotas
secrets
serviceaccounts
services
controllerrevisions.apps
daemonsets.apps
deployments.apps
replicasets.apps
statefulsets.apps
localsubjectaccessreviews.authorization.k8s.io
horizontalpodautoscalers.autoscaling
cronjobs.batch
jobs.batch
leases.coordination.k8s.io
events.events.k8s.io
ingresses.extensions
ingresses.networking.k8s.io
networkpolicies.networking.k8s.io
poddisruptionbudgets.policy
rolebindings.rbac.authorization.k8s.io
roles.rbac.authorization.k8s.io
```

Namespace with most Roles

```
➜ k -n project-c13 get role --no-headers | wc -l
No resources found in project-c13 namespace.
0

➜ k -n project-c14 get role --no-headers | wc -l
300

➜ k -n project-hamster get role --no-headers | wc -l
No resources found in project-hamster namespace.
0

➜ k -n project-snake get role --no-headers | wc -l
No resources found in project-snake namespace.
0

➜ k -n project-tiger get role --no-headers | wc -l
No resources found in project-tiger namespace.
0
```

Finally we write the name and amount into the file:

```
# /opt/course/16/crowded-namespace.txt
project-c14 with 300 resources
```

Question 17 | Find Container of Pod and check info

Task weight: 3%

Use context: `kubectl config use-context k8s-c1-H`

In *Namespace* `project-tiger` create a *Pod* named `tigers-reunite` of image `httpd:2.4.41-alpine` with labels `pod=container` and `container=pod`. Find out on which node the *Pod* is scheduled. Ssh into that node and find the containerd container belonging to that *Pod*.

Using command `crictl`:

1. Write the ID of the container and the `info.runtimeType` into `/opt/course/17/pod-container.txt`
2. Write the logs of the container into `/opt/course/17/pod-container.log`

Answer:

First we create the *Pod*:

```
k -n project-tiger run tigers-reunite \
  --image=httpd:2.4.41-alpine \
  --labels "pod=container,container=pod"
```

Next we find out the node it's scheduled on:

```
k -n project-tiger get pod -o wide

# or fancy:
k -n project-tiger get pod tigers-reunite -o jsonpath="{.spec.nodeName}"
```

Then we ssh into that node and and check the container info:

```
➜ ssh cluster1-node2

➜ root@cluster1-node2:~# crictl ps | grep tigers-reunite
b01edbe6f89ed    54b0995a63052    5 seconds ago    Running        tigers-reunite ...

➜ root@cluster1-node2:~# crictl inspect b01edbe6f89ed | grep runtimeType
    "runtimeType": "io.containerd.runc.v2",
```

Then we fill the requested file (on the main terminal):

```
# /opt/course/17/pod-container.txt
b01edbe6f89ed io.containerd.runc.v2
```

Finally we write the container logs in the second file:

```
ssh cluster1-node2 'crictl logs b01edbe6f89ed' &> /opt/course/17/pod-container.log
```

The `&>` in above's command redirects both the standard output and standard error.

You could also simply run `crictl logs` on the node and copy the content manually, if it's not a lot. The file should look like:

```
# /opt/course/17/pod-container.log
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.44.0.37. Set the
'ServerName' directive globally to suppress this message
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.44.0.37. Set the
'ServerName' directive globally to suppress this message
[Mon Sep 13 13:32:18.555280 2021] [mpm_event:notice] [pid 1:tid 139929534545224] AH00489: Apache/2.4.41 (Unix)
configured -- resuming normal operations
[Mon Sep 13 13:32:18.555610 2021] [core:notice] [pid 1:tid 139929534545224] AH00094: Command line: 'httpd -D
FOREGROUND'
```

Question 18 | Fix Kubelet

Task weight: 8%

Use context: `kubectl config use-context k8s-c3-CCC`

There seems to be an issue with the kubelet not running on `cluster3-node1`. Fix it and confirm that cluster has node `cluster3-node1` available in Ready state afterwards. You should be able to schedule a *Pod* on `cluster3-node1` afterwards.

Write the reason of the issue into `/opt/course/18/reason.txt`.

Answer:

The procedure on tasks like these should be to check if the kubelet is running, if not start it, then check its logs and correct errors if there are some.

Always helpful to check if other clusters already have some of the components defined and running, so you can copy and use existing config files. Though in this case it might not need to be necessary.

Check node status:

```
➜ k get node
NAME                     STATUS     ROLES           AGE   VERSION
cluster3-controlplane1   Ready      control-plane   14d   v1.26.0
cluster3-node1           NotReady   <none>          14d   v1.26.0
```

First we check if the kubelet is running:

```
➜ ssh cluster3-node1

➜ root@cluster3-node1:~# ps aux | grep kubelet
root     29294  0.0  0.2  14856  1016 pts/0    S+   11:30   0:00 grep --color=auto kubelet
```

Nope, so we check if it's configured using systemd as service:

```
➜ root@cluster3-node1:~# service kubelet status
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: inactive (dead) since Sun 2019-12-08 11:30:06 UTC; 50min 52s ago
...
```

Yes, it's configured as a service with config at `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`, but we see it's inactive. Let's try to start it:

```
➜ root@cluster3-node1:~# service kubelet start

➜ root@cluster3-node1:~# service kubelet status
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: activating (auto-restart) (Result: exit-code) since Thu 2020-04-30 22:03:10 UTC; 3s ago
     Docs: https://kubernetes.io/docs/home/
  Process: 5989 ExecStart=/usr/local/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS (code=exited, status=203/EXEC)
 Main PID: 5989 (code=exited, status=203/EXEC)

Apr 30 22:03:10 cluster3-node1 systemd[5989]: kubelet.service: Failed at step EXEC spawning /usr/local/bin/kubelet: No
such file or directory
Apr 30 22:03:10 cluster3-node1 systemd[1]: kubelet.service: Main process exited, code=exited, status=203/EXEC
Apr 30 22:03:10 cluster3-node1 systemd[1]: kubelet.service: Failed with result 'exit-code'.
```

We see it's trying to execute `/usr/local/bin/kubelet` with some parameters defined in its service config file. A good way to find errors and get more logs is to run the command manually (usually also with its parameters).

```
➜ root@cluster3-node1:~# /usr/local/bin/kubelet
-bash: /usr/local/bin/kubelet: No such file or directory

➜ root@cluster3-node1:~# whereis kubelet
kubelet: /usr/bin/kubelet
```

Another way would be to see the extended logging of a service like using `journalctl -u kubelet`.

Well, there we have it, wrong path specified. Correct the path in file `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf` and run:

```
vim /etc/systemd/system/kubelet.service.d/10-kubeadm.conf # fix

systemctl daemon-reload && systemctl restart kubelet

systemctl status kubelet  # should now show running
```

Also the node should be available for the api server, **give it a bit of time though**:

```
➜ k get node
NAME                     STATUS   ROLES           AGE   VERSION
cluster3-controlplane1   Ready    control-plane   14d   v1.26.0
cluster3-node1           Ready    <none>          14d   v1.26.0
```

Finally we write the reason into the file:

```
# /opt/course/18/reason.txt
wrong path to kubelet binary specified in service config
```

Question 19 | Create Secret and mount into Pod

Task weight: 3%

> **NOTE:** This task can only be solved if questions 18 or 20 have been successfully implemented and the k8s-c3-CCC cluster has a functioning worker node

Use context: `kubectl config use-context k8s-c3-CCC`

Do the following in a new *Namespace* `secret`. Create a *Pod* named `secret-pod` of image `busybox:1.31.1` which should keep running for some time.

There is an existing *Secret* located at `/opt/course/19/secret1.yaml`, create it in the *Namespace* `secret` and mount it readonly into the *Pod* at `/tmp/secret1`.

Create a new *Secret* in *Namespace* `secret` called `secret2` which should contain `user=user1` and `pass=1234`. These entries should be available inside the *Pod's* container as environment variables APP_USER and APP_PASS.

Confirm everything is working.

Answer

First we create the *Namespace* and the requested *Secrets* in it:

```
k create ns secret

cp /opt/course/19/secret1.yaml 19_secret1.yaml

vim 19_secret1.yaml
```

We need to adjust the *Namespace* for that *Secret*:

```
# 19_secret1.yaml
apiVersion: v1
data:
  halt: IyEgL2Jpbi9zaAo...
kind: Secret
metadata:
  creationTimestamp: null
  name: secret1
  namespace: secret           # change
```

```
k -f 19_secret1.yaml create
```

Next we create the second *Secret*:

```
k -n secret create secret generic secret2 --from-literal=user=user1 --from-literal=pass=1234
```

Now we create the *Pod* template:

```
k -n secret run secret-pod --image=busybox:1.31.1 $do -- sh -c "sleep 5d" > 19.yaml

vim 19.yaml
```

Then make the necessary changes:

```
# 19.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: secret-pod
  name: secret-pod
  namespace: secret                       # add
spec:
  containers:
  - args:
    - sh
    - -c
    - sleep 1d
    image: busybox:1.31.1
    name: secret-pod
    resources: {}
    env:                                  # add
    - name: APP_USER                      # add
      valueFrom:                          # add
        secretKeyRef:                     # add
          name: secret2                   # add
          key: user                       # add
    - name: APP_PASS                      # add
      valueFrom:                          # add
        secretKeyRef:                     # add
          name: secret2                   # add
          key: pass                       # add
    volumeMounts:                      # add
    - name: secret1                    # add
      mountPath: /tmp/secret1          # add
      readOnly: true                   # add
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  volumes:                             # add
  - name: secret1                      # add
    secret:                            # add
      secretName: secret1              # add
status: {}
```

It might not be necessary in current K8s versions to specify the `readOnly: true` because it's the default setting anyways.

And execute:

```
k -f 19.yaml create
```

Finally we check if all is correct:

```
➜ k -n secret exec secret-pod -- env | grep APP
APP_PASS=1234
APP_USER=user1
```

```
➜ k -n secret exec secret-pod -- find /tmp/secret1
/tmp/secret1
/tmp/secret1/..data
/tmp/secret1/halt
/tmp/secret1/..2019_12_08_12_15_39.463036797
/tmp/secret1/..2019_12_08_12_15_39.463036797/halt
```

```
➜ k -n secret exec secret-pod -- cat /tmp/secret1/halt
#! /bin/sh
### BEGIN INIT INFO
# Provides:          halt
# Required-Start:
# Required-Stop:
# Default-Start:
# Default-Stop:      0
# Short-Description: Execute the halt command.
# Description:
...
```

All is good.

Question 20 | Update Kubernetes Version and join cluster

Task weight: 10%

Use context: `kubectl config use-context k8s-c3-CCC`

Your coworker said node `cluster3-node2` is running an older Kubernetes version and is not even part of the cluster. Update Kubernetes on that node to the exact version that's running on `cluster3-controlplane1`. Then add this node to the cluster. Use kubeadm for this.

Answer:

Upgrade Kubernetes to cluster3-controlplane1 version

Search in the docs for kubeadm upgrade: https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade

```
➜ k get node
NAME                     STATUS   ROLES           AGE   VERSION
cluster3-controlplane1   Ready    control-plane   22h   v1.26.0
cluster3-node1           Ready    <none>          22h   v1.26.0
```

Controlplane node seems to be running Kubernetes 1.26.0 and `cluster3-node2` is not yet part of the cluster.

```
➜ ssh cluster3-node2

➜ root@cluster3-node2:~# kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"26", GitVersion:"v1.26.0",
GitCommit:"b46a3f887ca979b1a5d14fd39cb1af43e7e5d12d", GitTreeState:"clean", BuildDate:"2022-12-08T19:57:06Z",
GoVersion:"go1.19.4", Compiler:"gc", Platform:"linux/amd64"}

➜ root@cluster3-node2:~# kubectl version --short
Client Version: v1.25.5
Kustomize Version: v4.5.7

➜ root@cluster3-node2:~# kubelet --version
Kubernetes v1.25.5
```

Here kubeadm is already installed in the wanted version, so we don't need to install it. Hence we can run:

```
➜ root@cluster3-node2:~# kubeadm upgrade node
couldn't create a Kubernetes client from file "/etc/kubernetes/kubelet.conf": failed to load admin kubeconfig: open
/etc/kubernetes/kubelet.conf: no such file or directory
To see the stack trace of this error execute with --v=5 or higher
```

This is usually the proper command to upgrade a node. But this error means that this node was never even initialised, so nothing to update here. This will be done later using `kubeadm join`. For now we can continue with kubelet and kubectl:

```
➜ root@cluster3-node2:~# apt update
...
Fetched 5,775 kB in 2s (2,313 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
90 packages can be upgraded. Run 'apt list --upgradable' to see them.

➜ root@cluster3-node2:~# apt show kubectl -a | grep 1.26
Version: 1.26.0-00

➜ root@cluster3-node2:~# apt install kubectl=1.26.0-00 kubelet=1.26.0-00
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages will be upgraded:
  kubectl kubelet
2 upgraded, 0 newly installed, 0 to remove and 135 not upgraded.
Need to get 30.5 MB of archives.
After this operation, 9,996 kB of additional disk space will be used.
Get:1 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubectl amd64 1.26.0-00 [10.1 MB]
Get:2 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubelet amd64 1.26.0-00 [20.5 MB]
Fetched 30.5 MB in 1s (29.7 MB/s)
(Reading database ... 112508 files and directories currently installed.)
Preparing to unpack .../kubectl_1.26.0-00_amd64.deb ...
Unpacking kubectl (1.26.0-00) over (1.25.5-00) ...
Preparing to unpack .../kubelet_1.26.0-00_amd64.deb ...
Unpacking kubelet (1.26.0-00) over (1.25.5-00) ...
Setting up kubectl (1.26.0-00) ...
Setting up kubelet (1.26.0-00) ...

➜ root@cluster3-node2:~# kubelet --version
Kubernetes v1.26.0
```

Now we're up to date with kubeadm, kubectl and kubelet. Restart the kubelet:

```
➜ root@cluster3-node2:~# service kubelet restart

➜ root@cluster3-node2:~# service kubelet status
● kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
    Drop-In: /etc/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: activating (auto-restart) (Result: exit-code) since Wed 2022-12-21 16:29:26 UTC; 5s ago
       Docs: https://kubernetes.io/docs/home/
    Process: 32111 ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS
$KUBELET_EXTRA_ARGS (code=exited, status=1/FAILURE)
   Main PID: 32111 (code=exited, status=1/FAILURE)

Dec 21 16:29:26 cluster3-node2 systemd[1]: kubelet.service: Main process exited, code=exited, status=1/FAILURE
Dec 21 16:29:26 cluster3-node2 systemd[1]: kubelet.service: Failed with result 'exit-code'.
```

These errors occur because we still need to run `kubeadm join` to join the node into the cluster. Let's do this in the next step.

Add cluster3-node2 to cluster

First we log into the controlplane1 and generate a new TLS bootstrap token, also printing out the join command:

```
➜ ssh cluster3-controlplane1

➜ root@cluster3-controlplane1:~# kubeadm token create --print-join-command
kubeadm join 192.168.100.31:6443 --token rbhrjh.4o93r31o18an6dll --discovery-token-ca-cert-hash
sha256:d94524f9ab1eed84417414c7def5c1608f84dbf04437d9f5f73eb6255dafdb18

➜ root@cluster3-controlplane1:~# kubeadm token list
TOKEN                     TTL         EXPIRES                ...
44dz0t.2lgmone0i1o5z9fe   <forever>   <never>
4u477f.nmpq48xmpjt6weje   1h          2022-12-21T18:14:30Z
rbhrjh.4o93r31o18an6dll   23h         2022-12-22T16:29:58Z
```

We see the expiration of 23h for our token, we could adjust this by passing the ttl argument.

Next we connect again to `cluster3-node2` and simply execute the join command:

```
➜ ssh cluster3-node2

➜ root@cluster3-node2:~# kubeadm join 192.168.100.31:6443 --token rbhrjh.4o93r31o18an6dll --discovery-token-ca-cert-
hash sha256:d94524f9ab1eed84417414c7def5c1608f84dbf04437d9f5f73eb6255dafdb18
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.


➜ root@cluster3-node2:~# service kubelet status
● kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
    Drop-In: /etc/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: active (running) since Wed 2022-12-21 16:32:19 UTC; 1min 4s ago
       Docs: https://kubernetes.io/docs/home/
   Main PID: 32510 (kubelet)
      Tasks: 11 (limit: 462)
     Memory: 55.2M
     CGroup: /system.slice/kubelet.service
             └─32510 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --
kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --container-runti>
```

If you have troubles with `kubeadm join` you might need to run `kubeadm reset`.

This looks great though for us. Finally we head back to the main terminal and check the node status:

```
➜ k get node
NAME                     STATUS     ROLES           AGE   VERSION
cluster3-controlplane1   Ready      control-plane   22h   v1.26.0
cluster3-node1           Ready      <none>          22h   v1.26.0
cluster3-node2           NotReady   <none>          22h   v1.26.0
```

Give it a bit of time till the node is ready.

```
➜ k get node
NAME                     STATUS   ROLES           AGE   VERSION
cluster3-controlplane1   Ready    control-plane   22h   v1.26.0
cluster3-node1           Ready    <none>          22h   v1.26.0
cluster3-node2           Ready    <none>          22h   v1.26.0
```

We see `cluster3-node2` is now available and up to date.

Question 21 | Create a Static Pod and Service

Task weight: 2%

Use context: `kubectl config use-context k8s-c3-CCC`

Create a `Static Pod` named `my-static-pod` in *Namespace* `default` on cluster3-controlplane1. It should be of image `nginx:1.16-alpine` and have resource requests for `10m` CPU and `20Mi` memory.

Then create a NodePort *Service* named `static-pod-service` which exposes that static *Pod* on port 80 and check if it has *Endpoints* and if it's reachable through the `cluster3-controlplane1` internal IP address. You can connect to the internal node IPs from your main terminal.

Answer:

```
➜ ssh cluster3-controlplane1

➜ root@cluster1-controlplane1:~# cd /etc/kubernetes/manifests/

➜ root@cluster1-controlplane1:~# kubectl run my-static-pod \
    --image=nginx:1.16-alpine \
    -o yaml --dry-run=client > my-static-pod.yaml
```

Then edit the `my-static-pod.yaml` to add the requested resource requests:

```
# /etc/kubernetes/manifests/my-static-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: my-static-pod
  name: my-static-pod
spec:
  containers:
  - image: nginx:1.16-alpine
    name: my-static-pod
    resources:
      requests:
        cpu: 10m
        memory: 20Mi
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

And make sure it's running:

```
➜ k get pod -A | grep my-static
NAMESPACE     NAME                                    READY   STATUS    ...   AGE
default       my-static-pod-cluster3-controlplane1    1/1     Running   ...   22s
```

Now we expose that static *Pod*:

```
k expose pod my-static-pod-cluster3-controlplane1 \
  --name static-pod-service \
  --type=NodePort \
  --port 80
```

This would generate a *Service* like:

```
# kubectl expose pod my-static-pod-cluster3-controlplane1 --name static-pod-service --type=NodePort --port 80
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    run: my-static-pod
  name: static-pod-service
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    run: my-static-pod
  type: NodePort
status:
  loadBalancer: {}
```

Then run and test:

```
➜ k get svc,ep -l run=my-static-pod
NAME                          TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/static-pod-service    NodePort   10.99.168.252   <none>        80:30352/TCP   30s

NAME                            ENDPOINTS      AGE
endpoints/static-pod-service    10.32.0.4:80   30s
```

Looking good.

Question 22 | Check how long certificates are valid

Task weight: 2%

Use context: `kubectl config use-context k8s-c2-AC`

Check how long the kube-apiserver server certificate is valid on `cluster2-controlplane1`. Do this with openssl or cfssl. Write the exipiration date into `/opt/course/22/expiration`.

Also run the correct `kubeadm` command to list the expiration dates and confirm both methods show the same date.

Write the correct `kubeadm` command that would renew the apiserver server certificate into `/opt/course/22/kubeadm-renew-certs.sh`.

Answer:

First let's find that certificate:

```
➜ ssh cluster2-controlplane1

➜ root@cluster2-controlplane1:~# find /etc/kubernetes/pki | grep apiserver
/etc/kubernetes/pki/apiserver.crt
/etc/kubernetes/pki/apiserver-etcd-client.crt
/etc/kubernetes/pki/apiserver-etcd-client.key
/etc/kubernetes/pki/apiserver-kubelet-client.crt
/etc/kubernetes/pki/apiserver.key
/etc/kubernetes/pki/apiserver-kubelet-client.key
```

Next we use openssl to find out the expiration date:

```
➜ root@cluster2-controlplane1:~# openssl x509  -noout -text -in /etc/kubernetes/pki/apiserver.crt | grep Validity -A2
        Validity
            Not Before: Dec 20 18:05:20 2022 GMT
            Not After : Dec 20 18:05:20 2023 GMT
```

There we have it, so we write it in the required location on our main terminal:

```
# /opt/course/22/expiration
Dec 20 18:05:20 2023 GMT
```

And we use the feature from kubeadm to get the expiration too:

```
➜ root@cluster2-controlplane1:~# kubeadm certs check-expiration | grep apiserver
apiserver                Jan 14, 2022 18:49 UTC   363d        ca               no      
apiserver-etcd-client    Jan 14, 2022 18:49 UTC   363d        etcd-ca          no      
apiserver-kubelet-client Jan 14, 2022 18:49 UTC   363d        ca               no
```

Looking good. And finally we write the command that would renew all certificates into the requested location:

```
# /opt/course/22/kubeadm-renew-certs.sh
kubeadm certs renew apiserver
```

Question 23 | Kubelet client/server cert info

Task weight: 2%

Use context: `kubectl config use-context k8s-c2-AC`

Node cluster2-node1 has been added to the cluster using `kubeadm` and TLS bootstrapping.

Find the "Issuer" and "Extended Key Usage" values of the cluster2-node1:

1. kubelet **client** certificate, the one used for outgoing connections to the kube-apiserver.
2. kubelet **server** certificate, the one used for incoming connections from the kube-apiserver.

Write the information into file `/opt/course/23/certificate-info.txt`.

Compare the "Issuer" and "Extended Key Usage" fields of both certificates and make sense of these.

Answer:

To find the correct kubelet certificate directory, we can look for the default value of the `--cert-dir` parameter for the kubelet. For this search for "kubelet" in the Kubernetes docs which will lead to: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet. We can check if another certificate directory has been configured using `ps aux` or in `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`.

First we check the kubelet client certificate:

```
➜ ssh cluster2-node1

➜ root@cluster2-node1:~# openssl x509  -noout -text -in /var/lib/kubelet/pki/kubelet-client-current.pem | grep Issuer
        Issuer: CN = kubernetes

➜ root@cluster2-node1:~# openssl x509  -noout -text -in /var/lib/kubelet/pki/kubelet-client-current.pem | grep
"Extended Key Usage" -A1
            X509v3 Extended Key Usage:
                TLS Web Client Authentication
```

Next we check the kubelet server certificate:

```
➜ root@cluster2-node1:~# openssl x509  -noout -text -in /var/lib/kubelet/pki/kubelet.crt | grep Issuer
          Issuer: CN = cluster2-node1-ca@1588186506

➜ root@cluster2-node1:~# openssl x509  -noout -text -in /var/lib/kubelet/pki/kubelet.crt | grep "Extended Key Usage" -
A1
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
```

We see that the server certificate was generated on the worker node itself and the client certificate was issued by the Kubernetes api. The "Extended Key Usage" also shows if it's for client or server authentication.

More about this: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-tls-bootstrapping

Question 24 | NetworkPolicy

Task weight: 9%

Use context: `kubectl config use-context k8s-c1-H`

There was a security incident where an intruder was able to access the whole cluster from a single hacked backend *Pod*.

To prevent this create a *NetworkPolicy* called `np-backend` in *Namespace* `project-snake`. It should allow the `backend-*` *Pods* only to:

- connect to `db1-*` *Pods* on port 1111
- connect to `db2-*` *Pods* on port 2222

Use the `app` label of *Pods* in your policy.

After implementation, connections from `backend-*` *Pods* to `vault-*` *Pods* on port 3333 should for example no longer work.

Answer:

First we look at the existing *Pods* and their labels:

```
➜ k -n project-snake get pod
NAME        READY   STATUS    RESTARTS   AGE
backend-0   1/1     Running   0          8s
db1-0       1/1     Running   0          8s
db2-0       1/1     Running   0          10s
vault-0     1/1     Running   0          10s

➜ k -n project-snake get pod -L app
NAME        READY   STATUS    RESTARTS   AGE     APP
backend-0   1/1     Running   0          3m15s   backend
db1-0       1/1     Running   0          3m15s   db1
db2-0       1/1     Running   0          3m17s   db2
vault-0     1/1     Running   0          3m17s   vault
```

We test the current connection situation and see nothing is restricted:

```
➜ k -n project-snake get pod -o wide
NAME        READY   STATUS    RESTARTS   AGE     IP           ...
backend-0   1/1     Running   0          4m14s   10.44.0.24   ...
db1-0       1/1     Running   0          4m14s   10.44.0.25   ...
db2-0       1/1     Running   0          4m16s   10.44.0.23   ...
```

```
vault-0     1/1     Running   0          4m16s   10.44.0.22  ...

➜ k -n project-snake exec backend-0 -- curl -s 10.44.0.25:1111
database one

➜ k -n project-snake exec backend-0 -- curl -s 10.44.0.23:2222
database two

➜ k -n project-snake exec backend-0 -- curl -s 10.44.0.22:3333
vault secret storage
```

Now we create the *NP* by copying and chaning an example from the k8s docs:

```
vim 24_np.yaml
```

```
# 24_np.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-backend
  namespace: project-snake
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Egress                     # policy is only about Egress
  egress:
    -                            # first rule
      to:                            # first condition "to"
      - podSelector:
          matchLabels:
            app: db1
      ports:                         # second condition "port"
      - protocol: TCP
        port: 1111
    -                            # second rule
      to:                            # first condition "to"
      - podSelector:
          matchLabels:
            app: db2
      ports:                         # second condition "port"
      - protocol: TCP
        port: 2222
```

The *NP* above has two rules with two conditions each, it can be read as:

```
allow outgoing traffic if:
  (destination pod has label app=db1 AND port is 1111)
  OR
  (destination pod has label app=db2 AND port is 2222)
```

Wrong example

Now let's shortly look at a wrong example:

```
# WRONG
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-backend
  namespace: project-snake
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Egress
  egress:
    -                            # first rule
      to:                            # first condition "to"
      - podSelector:                     # first "to" possibility
          matchLabels:
            app: db1
      - podSelector:                     # second "to" possibility
          matchLabels:
            app: db2
      ports:                         # second condition "ports"
      - protocol: TCP                    # first "ports" possibility
        port: 1111
      - protocol: TCP                    # second "ports" possibility
        port: 2222
```

The *NP* above has one rule with two conditions and two condition-entries each, it can be read as:

```
allow outgoing traffic if:
  (destination pod has label app=db1 OR destination pod has label app=db2)
  AND
  (destination port is 1111 OR destination port is 2222)
```

Using this *NP* it would still be possible for `backend-*` *Pods* to connect to `db2-*` *Pods* on port 1111 for example which should be forbidden.

Create NetworkPolicy

We create the correct *NP*:

```
k -f 24_np.yaml create
```

And test again:

```
➜ k -n project-snake exec backend-0 -- curl -s 10.44.0.25:1111
database one

➜ k -n project-snake exec backend-0 -- curl -s 10.44.0.23:2222
database two

➜ k -n project-snake exec backend-0 -- curl -s 10.44.0.22:3333
^C
```

Also helpful to use `kubectl describe` on the *NP* to see how k8s has interpreted the policy.

Great, looking more secure. Task done.

Question 25 | Etcd Snapshot Save and Restore

Task weight: 8%

Use context: `kubectl config use-context k8s-c3-CCC`

Make a backup of etcd running on cluster3-controlplane1 and save it on the controlplane node at `/tmp/etcd-backup.db`.

Then create a *Pod* of your kind in the cluster.

Finally restore the backup, confirm the cluster is still working and that the created *Pod* is no longer with us.

Answer:

Etcd Backup

First we log into the controlplane and try to create a snapshop of etcd:

```
➜ ssh cluster3-controlplane1

➜ root@cluster3-controlplane1:~# ETCDCTL_API=3 etcdctl snapshot save /tmp/etcd-backup.db
Error:  rpc error: code = Unavailable desc = transport is closing
```

But it fails because we need to authenticate ourselves. For the necessary information we can check the etc manifest:

```
➜ root@cluster3-controlplane1:~# vim /etc/kubernetes/manifests/etcd.yaml
```

We only check the `etcd.yaml` for necessary information we don't change it.

```
# /etc/kubernetes/manifests/etcd.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
spec:
  containers:
  - command:
    - etcd
    - --advertise-client-urls=https://192.168.100.31:2379
    - --cert-file=/etc/kubernetes/pki/etcd/server.crt                           # use
    - --client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --initial-advertise-peer-urls=https://192.168.100.31:2380
    - --initial-cluster=cluster3-controlplane1=https://192.168.100.31:2380
    - --key-file=/etc/kubernetes/pki/etcd/server.key                            # use
    - --listen-client-urls=https://127.0.0.1:2379,https://192.168.100.31:2379   # use
    - --listen-metrics-urls=http://127.0.0.1:2381
    - --listen-peer-urls=https://192.168.100.31:2380
    - --name=cluster3-controlplane1
    - --peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt
    - --peer-client-cert-auth=true
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt                    # use
    - --snapshot-count=10000
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    image: k8s.gcr.io/etcd:3.3.15-0
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 8
      httpGet:
        host: 127.0.0.1
        path: /health
        port: 2381
        scheme: HTTP
      initialDelaySeconds: 15
      timeoutSeconds: 15
    name: etcd
    resources: {}
    volumeMounts:
    - mountPath: /var/lib/etcd
      name: etcd-data
    - mountPath: /etc/kubernetes/pki/etcd
      name: etcd-certs
  hostNetwork: true
  priorityClassName: system-cluster-critical
  volumes:
  - hostPath:
      path: /etc/kubernetes/pki/etcd
      type: DirectoryOrCreate
    name: etcd-certs
  - hostPath:
      path: /var/lib/etcd                                                     # important
      type: DirectoryOrCreate
    name: etcd-data
status: {}
```

But we also know that the api-server is connecting to etcd, so we can check how its manifest is configured:

```
➜ root@cluster3-controlplane1:~# cat /etc/kubernetes/manifests/kube-apiserver.yaml | grep etcd
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
```

We use the authentication information and pass it to etcdctl:

```
➜ root@cluster3-controlplane1:~# ETCDCTL_API=3 etcdctl snapshot save /tmp/etcd-backup.db \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/server.crt \
--key /etc/kubernetes/pki/etcd/server.key

Snapshot saved at /tmp/etcd-backup.db
```

> **NOTE:** Dont use `snapshot status` because it can alter the snapshot file and render it invalid

Etcd restore

Now create a *Pod* in the cluster and wait for it to be running:

```
➜ root@cluster3-controlplane1:~# kubectl run test --image=nginx
pod/test created

➜ root@cluster3-controlplane1:~# kubectl get pod -l run=test -w
NAME   READY   STATUS    RESTARTS   AGE
test   1/1     Running   0          60s
```

> **NOTE:** If you didn't solve questions 18 or 20 and cluster3 doesn't have a ready worker node then the created pod might stay in a Pending state. This is still ok for this task.

Next we stop all controlplane components:

```
root@cluster3-controlplane1:~# cd /etc/kubernetes/manifests/

root@cluster3-controlplane1:/etc/kubernetes/manifests# mv * ..

root@cluster3-controlplane1:/etc/kubernetes/manifests# watch crictl ps
```

Now we restore the snapshot into a specific directory:

```
➜ root@cluster3-controlplane1:~# ETCDCTL_API=3 etcdctl snapshot restore /tmp/etcd-backup.db \
--data-dir /var/lib/etcd-backup \
--cacert /etc/kubernetes/pki/etcd/ca.crt \
--cert /etc/kubernetes/pki/etcd/server.crt \
--key /etc/kubernetes/pki/etcd/server.key

2020-09-04 16:50:19.650804 I | mvcc: restore compact to 9935
2020-09-04 16:50:19.659095 I | etcdserver/membership: added member 8e9e05c52164694d [http://localhost:2380] to cluster
cdf818194e3a8c32
```

We could specify another host to make the backup from by using `etcdctl --endpoints http://IP`, but here we just use the default value which is: `http://127.0.0.1:2379,http://127.0.0.1:4001`.

The restored files are located at the new folder `/var/lib/etcd-backup`, now we have to tell etcd to use that directory:

```
➜ root@cluster3-controlplane1:~# vim /etc/kubernetes/etcd.yaml
```

```
# /etc/kubernetes/etcd.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
spec:
...
    - mountPath: /etc/kubernetes/pki/etcd
      name: etcd-certs
  hostNetwork: true
  priorityClassName: system-cluster-critical
  volumes:
  - hostPath:
      path: /etc/kubernetes/pki/etcd
      type: DirectoryOrCreate
    name: etcd-certs
  - hostPath:
      path: /var/lib/etcd-backup                # change
      type: DirectoryOrCreate
    name: etcd-data
status: {}
```

Now we move all controlplane yaml again into the manifest directory. Give it some time (up to several minutes) for etcd to restart and for the api-server to be reachable again:

```
root@cluster3-controlplane1:/etc/kubernetes/manifests# mv ../*.yaml .

root@cluster3-controlplane1:/etc/kubernetes/manifests# watch crictl ps
```

Then we check again for the *Pod*:

```
➜ root@cluster3-controlplane1:~# kubectl get pod -l run=test
No resources found in default namespace.
```

Awesome, backup and restore worked as our pod is gone.

Extra Question 1 | Find Pods first to be terminated

Use context: `kubectl config use-context k8s-c1-H`

Check all available *Pods* in the *Namespace* `project-c13` and find the names of those that would probably be terminated first if the *nodes* run out of resources (cpu or memory) to schedule all *Pods*. Write the *Pod* names into `/opt/course/e1/pods-not-stable.txt`.

Answer:

When available cpu or memory resources on the nodes reach their limit, Kubernetes will look for *Pods* that are using more resources than they requested. These will be the first candidates for termination. If some *Pods* containers have no resource requests/limits set, then by default those are considered to use more than requested.

Kubernetes assigns Quality of Service classes to *Pods* based on the defined resources and limits, read more here: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod

Hence we should look for *Pods* without resource requests defined, we can do this with a manual approach:

```
k -n project-c13 describe pod | less -p Requests # describe all pods and highlight Requests
```

Or we do:

```
k -n project-c13 describe pod | egrep "^(Name:|    Requests:)" -A1
```

We see that the *Pods* of *Deployment* `c13-3cc-runner-heavy` don't have any resources requests specified. Hence our answer would be:

```
# /opt/course/e1/pods-not-stable.txt
c13-3cc-runner-heavy-65588d7d6-djtv9map
c13-3cc-runner-heavy-65588d7d6-v8kf5map
c13-3cc-runner-heavy-65588d7d6-wwpb4map
o3db-0
o3db-1 # maybe not existing if already removed via previous scenario
```

To automate this process you could use jsonpath like this:

```
➜ k -n project-c13 get pod \
  -o jsonpath="{range .items[*]} {.metadata.name}{.spec.containers[*].resources}{'\n'}"

 c13-2x3-api-86784557bd-cgs8gmap[requests:map[cpu:50m memory:20Mi]]
 c13-2x3-api-86784557bd-lnxvjmap[requests:map[cpu:50m memory:20Mi]]
 c13-2x3-api-86784557bd-mnp77map[requests:map[cpu:50m memory:20Mi]]
 c13-2x3-web-769c989898-6hbgtmap[requests:map[cpu:50m memory:10Mi]]
 c13-2x3-web-769c989898-g57nqmap[requests:map[cpu:50m memory:10Mi]]
 c13-2x3-web-769c989898-hfd5vmap[requests:map[cpu:50m memory:10Mi]]
 c13-2x3-web-769c989898-jfx64map[requests:map[cpu:50m memory:10Mi]]
 c13-2x3-web-769c989898-r89mgmap[requests:map[cpu:50m memory:10Mi]]
 c13-2x3-web-769c989898-wtgxlmap[requests:map[cpu:50m memory:10Mi]]
 c13-3cc-runner-98c8b5469-dzqhrmap[requests:map[cpu:30m memory:10Mi]]
 c13-3cc-runner-98c8b5469-hbtdvmap[requests:map[cpu:30m memory:10Mi]]
 c13-3cc-runner-98c8b5469-n9lswmap[requests:map[cpu:30m memory:10Mi]]
 c13-3cc-runner-heavy-65588d7d6-djtv9map[]
 c13-3cc-runner-heavy-65588d7d6-v8kf5map[]
 c13-3cc-runner-heavy-65588d7d6-wwpb4map[]
 c13-3cc-web-675456bcd-glpq6map[requests:map[cpu:50m memory:10Mi]]
 c13-3cc-web-675456bcd-knlpxmap[requests:map[cpu:50m memory:10Mi]]
 c13-3cc-web-675456bcd-nfhp9map[requests:map[cpu:50m memory:10Mi]]
 c13-3cc-web-675456bcd-twn7mmap[requests:map[cpu:50m memory:10Mi]]
 o3db-0{}
 o3db-1{}
```

This lists all *Pod* names and their requests/limits, hence we see the three *Pods* without those defined.

Or we look for the Quality of Service classes:

```
➜ k get pods -n project-c13 \
  -o jsonpath="{range .items[*]}{.metadata.name} {.status.qosClass}{'\n'}"

c13-2x3-api-86784557bd-cgs8g Burstable
c13-2x3-api-86784557bd-lnxvj Burstable
c13-2x3-api-86784557bd-mnp77 Burstable
c13-2x3-web-769c989898-6hbgt Burstable
c13-2x3-web-769c989898-g57nq Burstable
c13-2x3-web-769c989898-hfd5v Burstable
c13-2x3-web-769c989898-jfx64 Burstable
c13-2x3-web-769c989898-r89mg Burstable
c13-2x3-web-769c989898-wtgxl Burstable
c13-3cc-runner-98c8b5469-dzqhr Burstable
c13-3cc-runner-98c8b5469-hbtdv Burstable
c13-3cc-runner-98c8b5469-n9lsw Burstable
c13-3cc-runner-heavy-65588d7d6-djtv9 BestEffort
c13-3cc-runner-heavy-65588d7d6-v8kf5 BestEffort
c13-3cc-runner-heavy-65588d7d6-wwpb4 BestEffort
c13-3cc-web-675456bcd-glpq6 Burstable
c13-3cc-web-675456bcd-knlpx Burstable
c13-3cc-web-675456bcd-nfhp9 Burstable
c13-3cc-web-675456bcd-twn7m Burstable
o3db-0 BestEffort
o3db-1 BestEffort
```

Here we see three with BestEffort, which *Pods* get that don't have any memory or cpu limits or requests defined.

A good practice is to always set resource requests and limits. If you don't know the values your containers should have you can find this out using metric tools like Prometheus. You can also use `kubectl top pod` or even `kubectl exec` into the container and use `top` and similar tools.

Extra Question 2 | Curl Manually Contact API

Use context: `kubectl config use-context k8s-c1-H`

There is an existing *ServiceAccount* `secret-reader` in *Namespace* `project-hamster`. Create a *Pod* of image `curlimages/curl:7.65.3` named `tmp-api-contact` which uses this *ServiceAccount*. Make sure the container keeps running.

Exec into the *Pod* and use `curl` to access the Kubernetes Api of that cluster manually, listing all available secrets. You can ignore insecure https connection. Write the command(s) for this into file `/opt/course/e4/list-secrets.sh`.

Answer:

https://kubernetes.io/docs/tasks/run-application/access-api-from-pod

It's important to understand how the Kubernetes API works. For this it helps connecting to the api manually, for example using curl. You can find information fast by search in the Kubernetes docs for "curl api" for example.

First we create our *Pod*:

```
k run tmp-api-contact \
  --image=curlimages/curl:7.65.3 $do \
  --command > e2.yaml -- sh -c 'sleep 1d'

vim e2.yaml
```

Add the service account name and *Namespace*:

```
# e2.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: tmp-api-contact
  name: tmp-api-contact
  namespace: project-hamster          # add
spec:
  serviceAccountName: secret-reader   # add
  containers:
  - command:
    - sh
    - -c
    - sleep 1d
    image: curlimages/curl:7.65.3
    name: tmp-api-contact
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Then run and exec into:

```
k -f 6.yaml create

k -n project-hamster exec tmp-api-contact -it -- sh
```

Once on the container we can try to connect to the api using `curl`, the api is usually available via the *Service* named `kubernetes` in *Namespace* `default` (You should know how dns resolution works across *Namespaces*.). Else we can find the endpoint IP via environment variables running `env`.

So now we can do:

```
curl https://kubernetes.default
curl -k https://kubernetes.default # ignore insecure as allowed in ticket description
curl -k https://kubernetes.default/api/v1/secrets # should show Forbidden 403
```

The last command shows 403 forbidden, this is because we are not passing any authorisation information with us. The Kubernetes Api Server thinks we are connecting as `system:anonymous`. We want to change this and connect using the *Pods ServiceAccount* named `secret-reader`.

We find the the token in the mounted folder at `/var/run/secrets/kubernetes.io/serviceaccount`, so we do:

```
➜ TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
➜ curl -k https://kubernetes.default/api/v1/secrets -H "Authorization: Bearer ${TOKEN}"
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0{
  "kind": "SecretList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/secrets",
    "resourceVersion": "10697"
  },
  "items": [
    {
      "metadata": {
        "name": "default-token-5zjbd",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/secrets/default-token-5zjbd",
        "uid": "315dbfd9-d235-482b-8bfc-c6167e7c1461",
        "resourceVersion": "342",
...
```

Now we're able to list all *Secrets*, registering as the *ServiceAccount* `secret-reader` under which our *Pod* is running.

To use encrypted https connection we can run:

```
CACERT=/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
curl --cacert ${CACERT} https://kubernetes.default/api/v1/secrets -H "Authorization: Bearer ${TOKEN}"
```

For troubleshooting we could also check if the *ServiceAccount* is actually able to list *Secrets* using:

```
➜ k auth can-i get secret --as system:serviceaccount:project-hamster:secret-reader
yes
```

Finally write the commands into the requested location:

```
# /opt/course/e4/list-secrets.sh
TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
curl -k https://kubernetes.default/api/v1/secrets -H "Authorization: Bearer ${TOKEN}"
```

CKA Simulator Preview Kubernetes 1.26

https://killer.sh

This is a preview of the full CKA Simulator course content.

The full course contains 25 scenarios from all the CKA areas. The course also provides a browser terminal which is a very close replica of the original one. This is great to get used and comfortable before the real exam. After the test session (120 minutes), or if you stop it early, you'll get access to all questions and their detailed solutions. You'll have 36 hours cluster access in total which means even after the session, once you have the solutions, you can still play around.

The following preview will give you an idea of what the full course will provide. These preview questions are in addition to the 25 of the full course. But the preview questions are part of the same CKA simulation environment which we setup for you, so with access to the full course you can solve these too.

The answers provided here assume that you did run the initial terminal setup suggestions as provided in the tips section, but especially:

```
alias k=kubectl

export do="-o yaml --dry-run=client"
```

These questions can be solved in the test environment provided through the CKA Simulator

Preview Question 1

Use context: `kubectl config use-context k8s-c2-AC`

The cluster admin asked you to find out the following information about etcd running on cluster2-controlplane1:

- Server private key location
- Server certificate expiration date
- Is client certificate authentication enabled

Write these information into `/opt/course/p1/etcd-info.txt`

Finally you're asked to save an etcd snapshot at `/etc/etcd-snapshot.db` on cluster2-controlplane1 and display its status.

Answer:

Find out etcd information

Let's check the nodes:

```
➜ k get node
NAME                     STATUS   ROLES           AGE   VERSION
cluster2-controlplane1   Ready    control-plane   89m   v1.23.1
cluster2-node1           Ready    <none>          87m   v1.23.1

➜ ssh cluster2-controlplane1
```

First we check how etcd is setup in this cluster:

```
➜ root@cluster2-controlplane1:~# kubectl -n kube-system get pod
NAME                                             READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-k8f48                         1/1     Running   0          26h
coredns-66bff467f8-rn8tr                         1/1     Running   0          26h
etcd-cluster2-controlplane1                      1/1     Running   0          26h
kube-apiserver-cluster2-controlplane1            1/1     Running   0          26h
kube-controller-manager-cluster2-controlplane1   1/1     Running   0          26h
kube-proxy-qthfg                                 1/1     Running   0          25h
kube-proxy-z55lp                                 1/1     Running   0          26h
kube-scheduler-cluster2-controlplane1            1/1     Running   1          26h
weave-net-cqdvt                                  2/2     Running   0          26h
weave-net-dxzgh                                  2/2     Running   1          25h
```

We see it's running as a *Pod*, more specific a static *Pod*. So we check for the default kubelet directory for static manifests:

```
➜ root@cluster2-controlplane1:~# find /etc/kubernetes/manifests/
/etc/kubernetes/manifests/
/etc/kubernetes/manifests/kube-controller-manager.yaml
/etc/kubernetes/manifests/kube-apiserver.yaml
/etc/kubernetes/manifests/etcd.yaml
/etc/kubernetes/manifests/kube-scheduler.yaml

➜ root@cluster2-controlplane1:~# vim /etc/kubernetes/manifests/etcd.yaml
```

So we look at the yaml and the parameters with which etcd is started:

```
# /etc/kubernetes/manifests/etcd.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
spec:
  containers:
  - command:
    - etcd
    - --advertise-client-urls=https://192.168.102.11:2379
    - --cert-file=/etc/kubernetes/pki/etcd/server.crt              # server certificate
    - --client-cert-auth=true                                      # enabled
    - --data-dir=/var/lib/etcd
    - --initial-advertise-peer-urls=https://192.168.102.11:2380
    - --initial-cluster=cluster2-controlplane1=https://192.168.102.11:2380
    - --key-file=/etc/kubernetes/pki/etcd/server.key               # server private key
    - --listen-client-urls=https://127.0.0.1:2379,https://192.168.102.11:2379
    - --listen-metrics-urls=http://127.0.0.1:2381
    - --listen-peer-urls=https://192.168.102.11:2380
    - --name=cluster2-controlplane1
    - --peer-cert-file=/etc/kubernetes/pki/etcd/peer.crt
    - --peer-client-cert-auth=true
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    - --snapshot-count=10000
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
...
```

We see that client authentication is enabled and also the requested path to the server private key, now let's find out the expiration of the server certificate:

```
➜ root@cluster2-controlplane1:~# openssl x509  -noout -text -in /etc/kubernetes/pki/etcd/server.crt | grep Validity -A2
        Validity
            Not Before: Sep 13 13:01:31 2021 GMT
            Not After : Sep 13 13:01:31 2022 GMT
```

There we have it. Let's write the information into the requested file:

```
# /opt/course/p1/etcd-info.txt
Server private key location: /etc/kubernetes/pki/etcd/server.key
Server certificate expiration date: Sep 13 13:01:31 2022 GMT
Is client certificate authentication enabled: yes
```

Create etcd snapshot

First we try:

```
ETCDCTL_API=3 etcdctl snapshot save /etc/etcd-snapshot.db
```

We get the endpoint also from the yaml. But we need to specify more parameters, all of which we can find the yaml declaration above:

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

This worked. Now we can output the status of the backup file:

```
➜ root@cluster2-controlplane1:~# ETCDCTL_API=3 etcdctl snapshot status /etc/etcd-snapshot.db
4d4e953, 7213, 1291, 2.7 MB
```

The status shows:

- Hash: 4d4e953
- Revision: 7213
- Total Keys: 1291
- Total Size: 2.7 MB

Preview Question 2

Use context: `kubectl config use-context k8s-c1-H`

You're asked to confirm that kube-proxy is running correctly on all nodes. For this perform the following in *Namespace* `project-hamster`:

Create a new *Pod* named `p2-pod` with two containers, one of image `nginx:1.21.3-alpine` and one of image `busybox:1.31`. Make sure the busybox container keeps running for some time.

Create a new *Service* named `p2-service` which exposes that *Pod* internally in the cluster on port 3000->80.

Find the kube-proxy container on all nodes `cluster1-controlplane1`, `cluster1-node1` and `cluster1-node2` and make sure that it's using iptables. Use command `crictl` for this.

Write the iptables rules of all nodes belonging the created *Service* `p2-service` into file `/opt/course/p2/iptables.txt`.

Finally delete the *Service* and confirm that the iptables rules are gone from all nodes.

Answer:

Create the *Pod*

First we create the *Pod*:

```
# check out export statement on top which allows us to use $do
k run p2-pod --image=nginx:1.21.3-alpine $do > p2.yaml

vim p2.yaml
```

Next we add the requested second container:

```
# p2.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: p2-pod
  name: p2-pod
  namespace: project-hamster             # add
spec:
  containers:
  - image: nginx:1.21.3-alpine
    name: p2-pod
  - image: busybox:1.31                  # add
    name: c2                             # add
    command: ["sh", "-c", "sleep 1d"]    # add
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

And we create the *Pod*:

```
k -f p2.yaml create
```

Create the *Service*

Next we create the *Service*:

```
k -n project-hamster expose pod p2-pod --name p2-service --port 3000 --target-port 80
```

This will create a yaml like:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2020-04-30T20:58:14Z"
  labels:
    run: p2-pod
  managedFields:
...
    operation: Update
    time: "2020-04-30T20:58:14Z"
  name: p2-service
  namespace: project-hamster
  resourceVersion: "11071"
  selfLink: /api/v1/namespaces/project-hamster/services/p2-service
  uid: 2a1c0842-7fb6-4e94-8cdb-1602a3b1e7d2
spec:
  clusterIP: 10.97.45.18
  ports:
  - port: 3000
    protocol: TCP
    targetPort: 80
  selector:
    run: p2-pod
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

We should confirm *Pods* and *Services* are connected, hence the *Service* should have *Endpoints*.

```
k -n project-hamster get pod,svc,ep
```

Confirm kube-proxy is running and is using iptables

First we get nodes in the cluster:

```
➜ k get node
NAME                     STATUS   ROLES           AGE   VERSION
cluster1-controlplane1   Ready    control-plane   98m   v1.23.1
cluster1-node1           Ready    <none>          96m   v1.23.1
cluster1-node2           Ready    <none>          95m   v1.23.1
```

The idea here is to log into every node, find the kube-proxy container and check its logs:

```
➜ ssh cluster1-controlplane1

➜ root@cluster1-controlplane1$ crictl ps | grep kube-proxy
27b6a18c0f89c       36c4ebbc9d979       3 hours ago         Running             kube-proxy

➜ root@cluster1-controlplane1~# crictl logs 27b6a18c0f89c
...
I0913 12:53:03.096620       1 server_others.go:212] Using iptables Proxier.
...
```

This should be repeated on every node and result in the same output `Using iptables Proxier`.

Check kube-proxy is creating iptables rules

Now we check the iptables rules on every node first manually:

```
➜ ssh cluster1-controlplane1 iptables-save | grep p2-service
-A KUBE-SEP-6U447UXLLQIKP7BB -s 10.44.0.20/32 -m comment --comment "project-hamster/p2-service:" -j KUBE-MARK-MASQ
```

```
-A KUBE-SEP-6U447UXLLQIKP7BB -p tcp -m comment --comment "project-hamster/p2-service:" -m tcp -j DNAT --to-destination 10.44.0.20:80
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.97.45.18/32 -p tcp -m comment --comment "project-hamster/p2-service: cluster IP" -m tcp --dport 3000 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.97.45.18/32 -p tcp -m comment --comment "project-hamster/p2-service: cluster IP" -m tcp --dport 3000 -j KUBE-SVC-2A6FNMCK6FDH7PJH
-A KUBE-SVC-2A6FNMCK6FDH7PJH -m comment --comment "project-hamster/p2-service:" -j KUBE-SEP-6U447UXLLQIKP7BB

➜ ssh cluster1-node1 iptables-save | grep p2-service
-A KUBE-SEP-6U447UXLLQIKP7BB -s 10.44.0.20/32 -m comment --comment "project-hamster/p2-service:" -j KUBE-MARK-MASQ
-A KUBE-SEP-6U447UXLLQIKP7BB -p tcp -m comment --comment "project-hamster/p2-service:" -m tcp -j DNAT --to-destination 10.44.0.20:80
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.97.45.18/32 -p tcp -m comment --comment "project-hamster/p2-service: cluster IP" -m tcp --dport 3000 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.97.45.18/32 -p tcp -m comment --comment "project-hamster/p2-service: cluster IP" -m tcp --dport 3000 -j KUBE-SVC-2A6FNMCK6FDH7PJH
-A KUBE-SVC-2A6FNMCK6FDH7PJH -m comment --comment "project-hamster/p2-service:" -j KUBE-SEP-6U447UXLLQIKP7BB

➜ ssh cluster1-node2 iptables-save | grep p2-service
-A KUBE-SEP-6U447UXLLQIKP7BB -s 10.44.0.20/32 -m comment --comment "project-hamster/p2-service:" -j KUBE-MARK-MASQ
-A KUBE-SEP-6U447UXLLQIKP7BB -p tcp -m comment --comment "project-hamster/p2-service:" -m tcp -j DNAT --to-destination 10.44.0.20:80
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.97.45.18/32 -p tcp -m comment --comment "project-hamster/p2-service: cluster IP" -m tcp --dport 3000 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.97.45.18/32 -p tcp -m comment --comment "project-hamster/p2-service: cluster IP" -m tcp --dport 3000 -j KUBE-SVC-2A6FNMCK6FDH7PJH
-A KUBE-SVC-2A6FNMCK6FDH7PJH -m comment --comment "project-hamster/p2-service:" -j KUBE-SEP-6U447UXLLQIKP7BB
```

Great. Now let's write these logs into the requested file:

```
➜ ssh cluster1-controlplane1 iptables-save | grep p2-service >> /opt/course/p2/iptables.txt
➜ ssh cluster1-node1 iptables-save | grep p2-service >> /opt/course/p2/iptables.txt
➜ ssh cluster1-node2 iptables-save | grep p2-service >> /opt/course/p2/iptables.txt
```

Delete the *Service* and confirm iptables rules are gone

Delete the *Service*:

```
k -n project-hamster delete svc p2-service
```

And confirm the iptables rules are gone:

```
➜ ssh cluster1-controlplane1 iptables-save | grep p2-service
➜ ssh cluster1-node1 iptables-save | grep p2-service
➜ ssh cluster1-node2 iptables-save | grep p2-service
```

Done.

Kubernetes *Services* are implemented using iptables rules (with default config) on all nodes. Every time a *Service* has been altered, created, deleted or *Endpoints* of a *Service* have changed, the kube-apiserver contacts every node's kube-proxy to update the iptables rules according to the current state.

Preview Question 3

Use context: `kubectl config use-context k8s-c2-AC`

Create a *Pod* named `check-ip` in *Namespace* `default` using image `httpd:2.4.41-alpine`. Expose it on port 80 as a ClusterIP *Service* named `check-ip-service`. Remember/output the IP of that *Service*.

Change the Service CIDR to `11.96.0.0/12` for the cluster.

Then create a second *Service* named `check-ip-service2` pointing to the same *Pod* to check if your settings did take effect. Finally check if the IP of the first *Service* has changed.

Answer:

Let's create the *Pod* and expose it:

```
k run check-ip --image=httpd:2.4.41-alpine

k expose pod check-ip --name check-ip-service --port 80
```

And check the *Pod* and *Service* ips:

```
➜ k get svc,ep -l run=check-ip
NAME                         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/check-ip-service     ClusterIP   10.104.3.45    <none>        80/TCP    8s

NAME                           ENDPOINTS      AGE
endpoints/check-ip-service     10.44.0.3:80   7s
```

Now we change the *Service* CIDR on the kube-apiserver:

```
➜ ssh cluster2-controlplane1

➜ root@cluster2-controlplane1:~# vim /etc/kubernetes/manifests/kube-apiserver.yaml
```

```
# /etc/kubernetes/manifests/kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.100.21
...
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-cluster-ip-range=11.96.0.0/12             # change
    - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
...
```

Give it a bit for the kube-apiserver and controller-manager to restart

Wait for the api to be up again:

```
➜ root@cluster2-controlplane1:~# kubectl -n kube-system get pod | grep api
kube-apiserver-cluster2-controlplane1            1/1     Running   0              49s
```

Now we do the same for the controller manager:

```
➜ root@cluster2-controlplane1:~# vim /etc/kubernetes/manifests/kube-controller-manager.yaml
```

```
# /etc/kubernetes/manifests/kube-controller-manager.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-controller-manager
    tier: control-plane
  name: kube-controller-manager
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-controller-manager
    - --allocate-node-cidrs=true
    - --authentication-kubeconfig=/etc/kubernetes/controller-manager.conf
    - --authorization-kubeconfig=/etc/kubernetes/controller-manager.conf
    - --bind-address=127.0.0.1
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --cluster-cidr=10.244.0.0/16
    - --cluster-name=kubernetes
    - --cluster-signing-cert-file=/etc/kubernetes/pki/ca.crt
    - --cluster-signing-key-file=/etc/kubernetes/pki/ca.key
    - --controllers=*,bootstrapsigner,tokencleaner
    - --kubeconfig=/etc/kubernetes/controller-manager.conf
    - --leader-elect=true
    - --node-cidr-mask-size=24
    - --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt
    - --root-ca-file=/etc/kubernetes/pki/ca.crt
    - --service-account-private-key-file=/etc/kubernetes/pki/sa.key
    - --service-cluster-ip-range=11.96.0.0/12         # change
    - --use-service-account-credentials=true
```

Give it a bit for the scheduler to restart.

We can check if it was restarted using `crictl`:

```
➜ root@cluster2-controlplane1:~# crictl ps | grep scheduler
3d258934b9fd6    aca5ededae9c8    About a minute ago   Running    kube-scheduler ...
```

Checking our existing *Pod* and *Service* again:

```
➜ k get pod,svc -l run=check-ip
NAME           READY   STATUS    RESTARTS   AGE
pod/check-ip   1/1     Running   0          21m

NAME                       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/check-ip-service   ClusterIP   10.99.32.177   <none>        80/TCP    21m
```

Nothing changed so far. Now we create another *Service* like before:

```
k expose pod check-ip --name check-ip-service2 --port 80
```

And check again:

```
➜ k get svc,ep -l run=check-ip
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
service/check-ip-service    ClusterIP   10.109.222.111   <none>        80/TCP    8m
service/check-ip-service2   ClusterIP   11.111.108.194   <none>        80/TCP    6m32s

NAME                          ENDPOINTS      AGE
endpoints/check-ip-service    10.44.0.1:80   8m
endpoints/check-ip-service2   10.44.0.1:80   6m13s
```

There we go, the new *Service* got an ip of the new specified range assigned. We also see that both *Services* have our *Pod* as endpoint.

CKA Tips Kubernetes 1.26

In this section we'll provide some tips on how to handle the CKA exam and browser terminal.

Knowledge

Study all topics as proposed in the curriculum till you feel comfortable with all.

General

- Study all topics as proposed in the curriculum till you feel comfortable with all
- Do 1 or 2 test session with this CKA Simulator. Understand the solutions and maybe try out other ways to achieve the same thing.
- Setup your aliases, be fast and breath `kubectl`
- The majority of tasks in the CKA will also be around creating Kubernetes resources, like it's tested in the CKAD. So preparing a bit for the CKAD can't hurt.
- Learn and Study the in-browser scenarios on https://killercoda.com/killer-shell-cka (and maybe for CKAD https://killercoda.com/killer-shell-ckad)
- Imagine and create your own scenarios to solve

Components

- Understanding Kubernetes components and being able to fix and investigate clusters: https://kubernetes.io/docs/tasks/debug-application-cluster/debug-cluster
- Know advanced scheduling: https://kubernetes.io/docs/concepts/scheduling/kube-scheduler
- When you have to fix a component (like kubelet) in one cluster, just check how it's setup on another node in the same or even another cluster. You can copy config files over etc
- If you like you can look at Kubernetes The Hard Way once. But it's NOT necessary to do, the CKA is not that complex. But KTHW helps understanding the concepts
- You should install your own cluster using kubeadm (one controlplane, one worker) in a VM or using a cloud provider and investigate the components
- Know how to use Kubeadm to for example add nodes to a cluster
- Know how to create an Ingress resources
- Know how to snapshot/restore ETCD from another machine

CKA Preparation

Read the Curriculum

https://github.com/cncf/curriculum

Read the Handbook

https://docs.linuxfoundation.org/tc-docs/certification/lf-candidate-handbook

Read the important tips

https://docs.linuxfoundation.org/tc-docs/certification/tips-cka-and-ckad

Read the FAQ

https://docs.linuxfoundation.org/tc-docs/certification/faq-cka-ckad

Kubernetes documentation

Get familiar with the Kubernetes documentation and be able to use the search. Allowed links are:

- https://kubernetes.io/docs
- https://kubernetes.io/blog
- https://helm.sh/docs

> ***NOTE:*** Verify the list here

The Test Environment / Browser Terminal

You'll be provided with a browser terminal which uses Ubuntu 20. The standard shells included with a minimal install of Ubuntu 20 will be available, including bash.

Laggin

There could be some lagging, definitely make sure you are using a good internet connection because your webcam and screen are uploading all the time.

Kubectl autocompletion and commands

Autocompletion is configured by default, as well as the `k` alias source and others:

`kubectl` with `k` alias and Bash autocompletion

`yq` and `jq` for YAML/JSON processing

`tmux` for terminal multiplexing

`curl` and `wget` for testing web services

`man` and man pages for further documentation

Copy & Paste

There could be issues copying text (like pod names) from the left task information into the terminal. Some suggested to "hard" hit or long hold `Cmd/Ctrl+C` a few times to take action. Apart from that copy and paste should just work like in normal terminals.

Percentages and Score

There are 15-20 questions in the exam and 100% of total percentage to reach. Each questions shows the % it gives if you solve it. Your results will be automatically checked according to the handbook. If you don't agree with the results you can request a review by contacting the Linux Foundation support.

Notepad & Skipping Questions

You have access to a simple notepad in the browser which can be used for storing any kind of plain text. It makes sense to use this for saving skipped question numbers and their percentages. This way it's possible to move some questions to the end. It might make sense to skip 2% or 3% questions and go directly to higher ones.

Contexts

You'll receive access to various different clusters and resources in each. They provide you the exact command you need to run to connect to another cluster/context. But you should be comfortable working in different namespaces with `kubectl`.

PSI Bridge

Starting with PSI Bridge:

- The exam will now be taken using the PSI Secure Browser, which can be downloaded using the newest versions of Microsoft Edge, Safari, Chrome, or Firefox
- Multiple monitors will no longer be permitted
- Use of personal bookmarks will no longer be permitted

The new ExamUI includes improved features such as:

- A remote desktop configured with the tools and software needed to complete the tasks
- A timer that displays the actual time remaining (in minutes) and provides an alert with 30, 15, or 5 minute remaining
- The content panel remains the same (presented on the Left Hand Side of the ExamUI)

Read more here.

Browser Terminal Setup

It should be considered to spend ~1 minute in the beginning to setup your terminal. In the real exam the vast majority of questions will be done from the main terminal. For few you might need to ssh into another machine. Just be aware that configurations to your shell will not be transferred in this case.

Minimal Setup

Alias

The alias `k` for `kubectl` will already be configured together with autocompletion. In case not you can configure it using this link.

Vim

The following settings will already be configured in your real exam environment in `~/.vimrc`. But it can never hurt to be able to type these down:

```
set tabstop=2
set expandtab
set shiftwidth=2
```

The `expandtab` make sure to use spaces for tabs. Memorize these and just type them down. You can't have any written notes with commands on your desktop etc.

Optional Setup

Fast dry-run output

```
export do="--dry-run=client -o yaml"
```

This way you can just run `k run pod1 --image=nginx $do`. Short for "dry output", but use whatever name you like.

Fast pod delete

```
export now="--force --grace-period 0"
```

This way you can run `k delete pod1 $now` and don't have to wait for ~30 seconds termination time.

Persist bash settings

You can store aliases and other setup in `~/.bashrc` if you're planning on using different shells or `tmux`.

Alias Namespace

In addition you could define an alias like:

```
alias kn='kubectl config set-context --current --namespace '
```

Which allows you to define the default namespace of the current context. Then once you switch a context or namespace you can just run:

```
kn default        # set default to default
kn my-namespace   # set default to my-namespace
```

But only do this if you used it before and are comfortable doing so. Else you need to specify the namespace for every call, which is also fine:

```
k -n my-namespace get all
k -n my-namespace get pod
...
```

Be fast

Use the `history` command to reuse already entered commands or use even faster history search through **Ctrl r** .

If a command takes some time to execute, like sometimes `kubectl delete pod x`. You can put a task in the background using **Ctrl z** and pull it back into foreground running command `fg`.

You can delete *pods* fast with:

```
k delete pod x --grace-period 0 --force

k delete pod x $now # if export from above is configured
```

Vim

Be great with vim.

toggle vim line numbers

When in `vim` you can press **Esc** and type `:set number` or `:set nonumber` followed by **Enter** to toggle line numbers. This can be useful when finding syntax errors based on line - but can be bad when wanting to mark&copy by mouse. You can also just jump to a line number with **Esc** `:22` + **Enter**.

copy&paste

Get used to copy/paste/cut with vim:

```
Mark lines: Esc+V (then arrow keys)
Copy marked lines: y
Cut marked lines: d
Past lines: p or P
```

Indent multiple lines

To indent multiple lines press **Esc** and type `:set shiftwidth=2`. First mark multiple lines using `Shift v` and the up/down keys. Then to indent the marked lines press `>` or `<`. You can then press `.` to repeat the action.

Split terminal screen

By default `tmux` is installed and can be used to split your one terminal into multiple. **But** just do this if you know your shit, because scrolling is different and copy&pasting might be weird.

https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux