

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

CKAD Simulator Kubernetes 1.21

<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. Set these:

```
alias k=kubectl

export do="--dry-run=client -o yaml" # like short for dry output. use whatever you like
```

vim

To make vim use 2 spaces for a tab edit `~/.vimrc` to contain:

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

More setup suggestions are in the tips section of the CKAD Simulator.

Deprecated commands

You might not be able to use deprecated `kubectl` commands in the real exam environment. Make sure to avoid those.

Difficulty

This simulator is more difficult than the real certification. We think this gives you a greater learning effect and also confidence to score in the real exam. Most of the simulator scenarios require good amount of work and can be considered "hard". In the real exam you will also face these "hard" scenarios, just less often.

Question 0 | Instructions

You should avoid using deprecated `kubectl` commands as these might not work in the exam.

When the countdown reaches 0 you'll be able to see the proposed solutions for each task. It is just an indicator for yourself and won't revoke access to this terminal when finished.

Rules

You're only allowed to have one other browser tab open with the Kubernetes documentation

- <https://kubernetes.io/docs>
- <https://github.com/kubernetes>
- <https://kubernetes.io/blog>

Notes

You have a notepad (top right) where you can store plain text. This is useful to store questions you skipped and might try again at the end.

Difficulty

This simulator is more difficult than the real certification. We think this gives you a greater learning effect and also confidence to score in the real exam. Most of the simulator scenarios require good amount of work and can be considered "hard". In the real exam you will also face these "hard" scenarios, just less often.

[Make sure](#)

More Data

Only edit/modify/delete those resources you want to act on. In most scenarios there are existing resources which could be part of other questions. Just like in a real world cluster.

Permissions

You have root permissions using sudo should you encounter any permission issues.

Ctrl/Cmd-F Search

Do not use the browser search via Ctrl-F or Cmd-F because this will render the browser terminal unusable. If this happened you can simply reload your browser page.

Question 1 | Namespaces

Task weight: 1%

The DevOps team would like to get the list of all *Namespaces* in the cluster. Get the list and save it to `/opt/course/1/namespaces`.

Answer:

```
k get ns > /opt/course/1/namespaces
```

The content should then look like:

```
# /opt/course/1/namespaces
NAME      STATUS   AGE
default   Active   150m
earth     Active   76m
jupiter   Active   76m
kube-public Active  150m
kube-system Active  150m
mars      Active   76m
mercury   Active   76m
moon      Active   76m
neptune   Active   76m
pluto     Active   76m
saturn    Active   76m
shell-intern Active  76m
sun       Active   76m
venus     Active   76m
```

Question 2 | Pods

Task weight: 3%

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`.

Your manager would like to run a command manually on occasion to output the status of that exact *Pod*. Please write a command that does this into `/opt/course/2/pod1-status-command.sh`. The command should use `kubectl`.

Answer:

```
k run # help

# 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
```

Change the container name in `2.yaml` to `pod1-container`:

```
# 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
  status: {}
```

Then run:

```
→ k create -f 2.yaml
pod/pod1 created

→ k get pod
NAME     READY   STATUS            RESTARTS   AGE
pod1    0/1     ContainerCreating   0          6s

→ k get pod
NAME     READY   STATUS     RESTARTS   AGE
pod1    1/1     Running   0          30s
```

Next create the requested command:

```
vim /opt/course/2/pod1-status-command.sh
```

The content of the command file could look like:

```
# /opt/course/2/pod1-status-command.sh
kubectl -n default describe pod pod1 | grep -i status:
```

Another solution would be using jsonpath:

```
# /opt/course/2/pod1-status-command.sh
kubectl -n default get pod pod1 -o jsonpath=".status.phase"
```

To test the command:

```
→ sh /opt/course/2/pod1-status-command.sh
Running
```

Question 3 | Job

Task weight: 3%

Team Neptune needs a `Job` template located at `/opt/course/3/job.yaml`. This `Job` should run image `busybox:1.31.0` and execute `sleep 2 && echo done`. It should be in namespace `neptune`, run a total of 3 times and should execute 2 runs in parallel.

Start the `Job` and check its history. Each pod created by the `Job` should have the label `id: awesome-job`. The job should be named `neb-new-job` and the container `neb-new-job-container`.

Answer:

```
k -n neptun create job -h
```

```
# check the export on the very top of this document so we can use $do
k -n neptune create job neb-new-job --image=busybox:1.31.0 $do > /opt/course/3/job.yaml -
- sh -c "sleep 2 && echo done"

vim /opt/course/3/job.yaml
```

Before v1.18 it used to be possible to create a *Job* with `k -n neptune run neb-new-job --image=busybox:1.31.0 --restart=OnFailure`, but not any longer.

Make the required changes in the yaml:

```
# /opt/course/3/job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  creationTimestamp: null
  name: neb-new-job
  namespace: neptune      # add
spec:
  completions: 3          # add
  parallelism: 2          # add
  template:
    metadata:
      creationTimestamp: null
      labels:           # add
        id: awesome-job # add
    spec:
      containers:
        - command:
            - sh
            - -c
            - sleep 2 && echo done
          image: busybox:1.31.0
          name: neb-new-job-container # update
          resources: {}
        restartPolicy: Never
status: {}
```

Then to create it:

```
k -f /opt/course/3/job.yaml create # namespace already set in yaml
```

Check *Job* and *Pods*, you should see two running parallel at most but three in total:

```
→ k -n neptune get pod,job | grep neb-new-job
pod/neb-new-job-jhq2g          0/1     ContainerCreating  0      4s
pod/neb-new-job-vf6ts          0/1     ContainerCreating  0      4s

job.batch/neb-new-job  0/3      4s      5s

→ k -n neptune get pod,job | grep neb-new-job
pod/neb-new-job-gm8sz          0/1     ContainerCreating  0      0s
pod/neb-new-job-jhq2g          0/1     Completed       0      10s
pod/neb-new-job-vf6ts          1/1     Running        0      10s

job.batch/neb-new-job  1/3      10s     11s

→ k -n neptune get pod,job | grep neb-new-job
pod/neb-new-job-gm8sz          0/1     ContainerCreating  0      5s
pod/neb-new-job-jhq2g          0/1     Completed       0      15s
pod/neb-new-job-vf6ts          0/1     Completed       0      15s
job.batch/neb-new-job  2/3      15s     16s

→ k -n neptune get pod,job | grep neb-new-job
pod/neb-new-job-gm8sz          0/1     Completed       0      12s
pod/neb-new-job-jhq2g          0/1     Completed       0      22s
pod/neb-new-job-vf6ts          0/1     Completed       0      22s

job.batch/neb-new-job  3/3      21s     23s
```

Check history:

```

→ k -n neptune describe job neb-new-job
...
Events:
  Type    Reason        Age     From            Message
  ----  -----        ----   ----            -----
Normal  SuccessfulCreate  2m52s  job-controller  Created pod: neb-new-job-jhq2g
Normal  SuccessfulCreate  2m52s  job-controller  Created pod: neb-new-job-vf6ts
Normal  SuccessfulCreate  2m42s  job-controller  Created pod: neb-new-job-gm8sz

```

At the age column we can see that two `pods` run parallel and the third one after that. Just as it was required in the task.

Question 4 | Requests and Limits, ServiceAccount

Task weight: 4%

Team Neptune needs 3 *Pods* of image `httpd:2.4-alpine`, create a *Deployment* named `neptune-10ab` for this. The containers should be named `neptune-pod-10ab`. Each container should have a memory request of `20Mi` and a memory limit of `50Mi`.

Team Neptune has its own *ServiceAccount* `neptune-sa-v2` under which the *Pods* should run. The *Deployment* should be in *Namespace* `neptune`.

Answer:

```

k -n neptune create deployment -h # help
k -n neptune create deploy -h # deploy is short for deployment

# check the export on the very top of this document so we can use $do
k -n neptune create deploy neptune-10ab --image=httpd:2.4-alpine $do > 4.yaml

vim 4.yaml

```

NOTE: Before 1.18 it used to be possible to use the `kubectl run` to create a deployment with any attributes, but not any longer.

Now make the required changes using vim:

```

# 4.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: neptune-10ab
  name: neptune-10ab
  namespace: neptune
spec:
  replicas: 3          # change
  selector:
    matchLabels:
      app: neptune-10ab
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: neptune-10ab
    spec:
      serviceAccountName: neptune-sa-v2 # add
      containers:
        - image: httpd:2.4-alpine
          name: neptune-pod-10ab # change
          resources:           # add
            limits:             # add
              memory: 50Mi       # add
            requests:           # add
              memory: 20Mi       # add

```

```

memory: 20Mi          # add
requests: 20Mi        # add
status: {}

```

If we don't want to write the resources section manually we could run the following command and copy it manually into our yaml file:

```
k run tmp --image=busybox $do --requests=memory=20Mi --limits=memory=50Mi
```

Then create the yaml:

```
k create -f 4.yaml # namespace already set in yaml
```

To verify all *Pods* are running we do:

```

→ k -n neptune get pod | grep neptune-10ab
neptune-10ab-7d4b8d45b-4nzj5  1/1    Running      0      57s
neptune-10ab-7d4b8d45b-1zwrft 1/1    Running      0      17s
neptune-10ab-7d4b8d45b-z5hcc  1/1    Running      0      17s

```

Question 5 | ServiceAccount, Secret

Task weight: 3%

Team Neptune has its own *ServiceAccount* named `neptune-sa-v2` in *Namespace neptune*. A coworker needs the token from the *Secret* that belongs to that *ServiceAccount*. Write the base64 decoded token to file `/opt/course/5/token`.

Answer:

```

k -n neptune get sa # get overview
k -n neptune get secrets # shows all secrets of namespace
k -n neptune get sa neptune-sa-v2 -o yaml | grep secret -A 2 # shows the secret name
k -n neptune get secret neptune-sa-v2-token-lwhh1 -o yaml # shows the secret content

```

```

→ k -n neptune get secret neptune-sa-v2-token-lwhh1 -o yaml
apiVersion: v1
data:
...
token:
Zx1KaGJHY21PaUpTVxpJMU5psXNjbXRwWkNjNk1tNwFaRmRxWkRKMmFHTnZRM0JxV0haT1IxZzFiM3BjY201S1owa
EhOV3hUwmt3e1FuRmFhVeZhZDJNaWZRLmV5SnBjM01pT21KcmRSxmjb1tvSzEdWekwzTmxjb1pwwTJWaFkyTnZkvz
UwsWl3awEzVmLawEp1w1hsbGN5NbBiet16w1hKmxFtmxZv05qYjNwdwRDOXVZvFsYZNCaFkyVw1Paup1w1hCmGR
XNwxJaXdpYTNWaVpYSnVawFjsy3k1cGJ5oxpawEoyVvdobF1XTmpim1z1ZEM5e1pxTn1aWFF1Ym1GdFpTSTZjbTVs
Y0hSMWjtvxrjMkv0ZGpjdgRHOXJaVzR0wm5FNU1tb21mq0pyzfdkbgntNwxkr1z6TG1sdkwzTmxjb1pwwTJWaFkyT
nzKvzuwTDNoBGuNwBZM1V0WvdoamIzvnvkQzV1wvxbE1qb21ibvZ3ehwdvpTMxpuzEyTw1jco1tdfzbvz5Ym
1wMFpYTvxvhzh2yJewRtbgpav0zqwt15MwjuuxZjM1z5Zglsa1ptmwhzmk52Zfc1MexuvnBaQ0k2swpzm1ltuimp
oakoyTRKbf16txROREpowkMwne9HRTFMV0zowxpgbFpqwmxPVFpst1njc01utjFzauk2sw5ONWmzumxiVHB6w1hk
MmfXTmxZv05qYjNwdwRECHVawEfzFc1be9tnwxjsFIxy1vdlGMyRXRkak1pz1Ev11nym9NNENUZDbwZENKnz3a
1v3bxRhbGgtMnZzs2pBt1nc2gtNm1DxRdPdFdFcTVGYnc1WkhQdHZBZJMbfB6ce9IRwJBZTR1vu05nujsr1diwu
1kd2p1Tjk1SjBENFj0RmtwvXQOHR3b2FrU1Y3aC1huV3c1FYSghawnp5Nh1pbUzirz1Vzm1zazVzcjRSVmNHnm4
xMzd5LUZIMDhLOhpaaak1qqXNLRFQ01F0eGctbFp2d1zNaTz2au1ocnJ6QVFzME1CT1Y4Mk9KwUd5Mm8tV1FwYzBV
VwFuq2Y5NFkzz1qwVRpcVF2Y3pZTXM2bno5dxQtwd3axRyQ1k2Vgo5QmdQchjbowtfajVxRxhfTFVvw1vwUEFpr
U43T3pka0pzStbjdHROMTBseXBjMUF1RnI0M3Q2QuX5c1FvQk0z0wFirGzxM0Zrc1Itb2Nfv013
kind: Secret
...

```

This shows the base64 encoded token. To get the encoded one we could pipe it manually through `base64 -d` or we simply do:

```

→ k -n neptune describe secret neptune-sa-v2-token-lwhh1
...
Data
=====
token:
eyJhbGciOiJSUzI1NiISImtpZCI6Im5aZFdqZDJ2aGNvQ3BqWHZOR1g1b3pIcm5jZ0hHNwxTzkwQnFaaTFad2Mif

```

```

Q.eYJpc3MioiJrdwJlcM5ldGVzL3N1cnzpY2vHy2Nvdw50Iiwia3VizXJuZXr1cy5pbY9ZzXj2awN1YwNjb3Vudc9
uYw1lC3BhY2UiOijuzXB0dw5lIiwia3VizXJuZXr1cy5pbY9ZzXj2awN1YwNjb3Vudc9zzwNyZXQubmFtZSI6Im51
cHR1bmUtc2EtdjItdg9rzW4tznE5MmoiLCJrdwJlcM5ldGVzLmlvL3N1cnzpY2vHy2Nvdw50L3N1cnzpY2utYwNjb
3Vudc5uYw1lIjoibmVwdhvzs1zys12MiisImt1YmVbmv0Zxmuaw8vc2VydmljzwfjY291bnQvc2Vydmljzs1hy2
Nvdw50LnvpZCI6IjY2YmRjnJM2LTj1YzmtNDjhzc040GE1LWFhYzf1zjz1otz1nsisInN1yiI6InN5c3R1bTpzzXj
2awN1YwNjb3VudDpuZXb0dw5lom51cHR1bmUtc2EtdjIifQ.VYgbom4Ctd0pdCj78wjUwmthalh-2vsKjANyPsh-
6guEwotwEq5Fbw5ZHPtvAdrl1Pzp0HEBa4eUM95RGwbYIdwju95j0D4RNfkvt48twoakRV7h-
aPuwsQXHzzzy4imFHG9Ufmsk5Yr4Rvcg6n137y-FH08K8zzjIPASkDqNBQtxg-
1Zvvvmi6viIhrrzaQs0MBoV82ojYgy2o-wQvc0UUancf94Y3gt0YTiqQvczYMs6nz9ut-
XgwitrBY6Tj9BgPprA9k_j5qEx_LUUZUpPAiEN70zdkjsI8ctth10lyp1AeFr43t6ALyrQoBM39abDfq3FksR-
oc_wMw
ca.crt:      1066 bytes
namespace:   7 bytes

```

Copy the token (part under `token:`) and paste it using vim.

```
vim /opt/course/5/token
```

File `/opt/course/5/token` should contain the token:

```

# /opt/course/5/token
eyJhbGciOiJSUzI1NiIiSImtpZCI6Im5azFdqZDj2aGNVQ3BqWHZOR1g1b3pIcm5jZ0hHNwxTZkwzQnFaaTFad2Mif
Q.eYJpc3MioiJrdwJlcM5ldGVzL3N1cnzpY2vHy2Nvdw50Iiwia3VizXJuZXr1cy5pbY9ZzXj2awN1YwNjb3Vudc9
uYw1lC3BhY2UiOijuzXB0dw5lIiwia3VizXJuZXr1cy5pbY9ZzXj2awN1YwNjb3Vudc9zzwNyZXQubmFtZSI6Im51
cHR1bmUtc2EtdjItdg9rzW4tznE5MmoiLCJrdwJlcM5ldGVzLmlvL3N1cnzpY2vHy2Nvdw50L3N1cnzpY2utYwNjb
3Vudc5uYw1lIjoibmVwdhvzs1zys12MiisImt1YmVbmv0Zxmuaw8vc2VydmljzwfjY291bnQvc2Vydmljzs1hy2
Nvdw50LnvpZCI6IjY2YmRjnJM2LTj1YzmtNDjhzc040GE1LWFhYzf1zjz1otz1nsisInN1yiI6InN5c3R1bTpzzXj
2awN1YwNjb3VudDpuZXb0dw5lom51cHR1bmUtc2EtdjIifQ.VYgbom4Ctd0pdCj78wjUwmthalh-2vsKjANyPsh-
6guEwotwEq5Fbw5ZHPtvAdrl1Pzp0HEBa4eUM95RGwbYIdwju95j0D4RNfkvt48twoakRV7h-
aPuwsQXHzzzy4imFHG9Ufmsk5Yr4Rvcg6n137y-FH08K8zzjIPASkDqNBQtxg-
1Zvvvmi6viIhrrzaQs0MBoV82ojYgy2o-wQvc0UUancf94Y3gt0YTiqQvczYMs6nz9ut-
XgwitrBY6Tj9BgPprA9k_j5qEx_LUUZUpPAiEN70zdkjsI8ctth10lyp1AeFr43t6ALyrQoBM39abDfq3FksR-
oc_wMw

```

Question 6 | ReadinessProbe

Task weight: 7%

Create a single Pod named `pod6` in Namespace `default` of image `busybox:1.31.0`. The Pod should have a readiness-probe executing `cat /tmp/ready`. It should initially wait 5 and periodically wait 10 seconds. This will set the container ready only if the file `/tmp/ready` exists.

The Pod should run the command `touch /tmp/ready && sleep 1d`, which will create the necessary file to be ready and then idles. Create the Pod and confirm it starts.

Answer:

```
k run pod6 --image=busybox:1.31.0 $do --command -- sh -c "touch /tmp/ready && sleep 1d" > 6.yaml
```

```
vim 6.yaml
```

Search for a readiness-probe example on <https://kubernetes.io/docs>, then copy and alter the relevant section for the task:

```

# 6.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: pod6
  name: pod6
spec:
  containers:
    - args:
        - sh
        - -c

```

```

- touch /tmp/ready && sleep 1d
image: busybox:1.31.0
name: pod6
resources: {}
readinessProbe: # add
  exec: # add
    command: # add
      - sh # add
      - -c # add
      - cat /tmp/ready # add
    initialDelaySeconds: 5 # add
    periodSeconds: 10 # add
dnsPolicy: ClusterFirst
restartPolicy: Always
status: {}

```

Then:

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

Running `k get pod6` we should see the job being created and completed:

```

→ k get pod pod6
NAME READY STATUS RESTARTS AGE
pod6 0/1 ContainerCreating 0 2s

→ k get pod pod6
NAME READY STATUS RESTARTS AGE
pod6 0/1 Running 0 7s

→ k get pod pod6
NAME READY STATUS RESTARTS AGE
pod6 1/1 Running 0 15s

```

We see that the *Pod* is finally ready.

Question 7 | Pods, Namespaces

Task weight: 4%

The board of Team Neptune decided to take over control of one e-commerce webserver from Team Saturn. The administrator who once setup this webserver is not part of the organisation any longer. All information you could get was that the e-commerce system is called `my-happy-shop`.

Search for the correct *Pod* in *Namespace saturn* and move it to *Namespace neptune*. It doesn't matter if you shut it down and spin it up again, it probably hasn't any customers anyways.

Answer:

Let's see all those *Pods*:

```

→ k -n saturn get pod
NAME READY STATUS RESTARTS AGE
webserver-sat-001 1/1 Running 0 111m
webserver-sat-002 1/1 Running 0 111m
webserver-sat-003 1/1 Running 0 111m
webserver-sat-004 1/1 Running 0 111m
webserver-sat-005 1/1 Running 0 111m
webserver-sat-006 1/1 Running 0 111m

```

The *Pod* names don't reveal any information. We assume the *Pod* we are searching has a *label* or *annotation* with the name `my-happy-shop`, so we search for it:

```

k -n saturn describe pod # describe all pods, then manually look for it

# or do some filtering like this
k -n saturn get pod -o yaml | grep my-happy-shop -A10

```

We see the webserver we're looking for is `webserver-sat-003`

```
k -n saturn get pod webserver-sat-003 -o yaml > 7_webserver-sat-003.yaml # export  
vim 7_webserver-sat-003.yaml
```

Change the `Namespace` to `neptune`, also remove the `status:` section, the token `volume`, the token `volumeMount` and the `nodeName`, else the new `Pod` won't start. The final file could look as clean like this:

```
# 7_webserver-sat-003.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
  annotations:  
    description: this is the server for the E-Commerce System my-happy-shop  
  labels:  
    id: webserver-sat-003  
  name: webserver-sat-003  
  namespace: neptune # new namespace here  
spec:  
  containers:  
  - image: nginx:1.16.1-alpine  
    imagePullPolicy: IfNotPresent  
    name: webserver-sat  
    restartPolicy: Always
```

Then we execute:

```
k -n neptune create -f 7_webserver-sat-003.yaml
```

```
→ k -n neptune get pod | grep webserver  
webserver-sat-003           1/1     Running      0          22s
```

It seems the server is running in `Namespace neptune`, so we can do:

```
k -n saturn delete pod webserver-sat-003 --force --grace-period=0
```

Let's confirm only one is running:

```
→ k get pod -A | grep webserver-sat-003  
neptune      webserver-sat-003           1/1     Running      0          6s
```

This should list only one pod called `webserver-sat-003` in `Namespace neptune`, status running.

Question 8 | Deployment, Rollouts

Task weight: 4%

There is an existing `Deployment` named `api-new-c32` in `Namespace neptune`. A developer did make an update to the `Deployment` but the updated version never came online. Check the `Deployment` history and find a revision that works, then rollback to it. Could you tell Team Neptune what the error was so it doesn't happen again?

Answer:

```
k -n neptune get deploy # overview  
k -n neptune rollout -h  
k -n neptune rollout history -h
```

```
→ k -n neptune rollout history deploy api-new-c32  
deployment.extensions/api-new-c32  
REVISION  CHANGE-CAUSE  
1        <none>  
2        kubectl edit deployment api-new-c32 --namespace=neptune  
3        kubectl edit deployment api-new-c32 --namespace=neptune  
4        kubectl edit deployment api-new-c32 --namespace=neptune  
5        kubectl edit deployment api-new-c32 --namespace=neptune
```

We see 5 revisions. Let's check Pod and Deployment status:

Want to see all revisions, let's check out the deployment status:

```
→ k -n neptune get deploy,pod | grep api-new-c32
deployment.extensions/api-new-c32 3/3 1 3 141m
pod/api-new-c32-65d998785d-jtmqq 1/1 Running 0 141m
pod/api-new-c32-686d6f6b65-mj2fp 1/1 Running 0 141m
pod/api-new-c32-6dd45bdb68-2p462 1/1 Running 0 141m
pod/api-new-c32-7d64747c87-zh648 0/1 ImagePullBackoff 0 141m
```

Let's check the pod for errors:

```
→ k -n neptune describe pod api-new-c32-7d64747c87-zh648 | grep -i error
... Error: ImagePullBackoff
```

```
→ k -n neptune describe pod api-new-c32-7d64747c87-zh648 | grep -i image
Image: ngnix:1.16.3
Image ID:
Reason: ImagePullBackoff
Warning Failed 4m28s (x616 over 144m) kubelet, gke-s3ef67020-28c5-45f7--default-
pool-248abd4f-s010 Error: ImagePullBackoff
```

Someone seems to have added a new image with a spelling mistake in the name `ngnix:1.16.3`, that's the reason we can tell Team Neptune!

Now let's revert to the previous version:

```
k -n neptune rollout undo deploy api-new-c32
```

Does this one work?

```
→ k -n neptune get deploy api-new-c32
NAME READY UP-TO-DATE AVAILABLE AGE
api-new-c32 3/3 3 3 146m
```

Yes! All up-to-date and available.

Also a fast way to get an overview of the *ReplicaSets* of a *Deployment* and their images could be done with:

```
k -n neptune get rs -o wide | grep api-new-c32
```

Question 9 | Pod -> Deployment

Task weight: 8%

In *Namespace pluto* there is single *Pod* named `holy-api`. It has been working okay for a while now but Team Pluto needs it to be more reliable. Convert the *Pod* into a *Deployment* with 3 replicas and name `holy-api`. The raw *Pod* template file is available at `/opt/course/9/holy-api-pod.yaml`.

Please create the *Deployment* and save its yaml under `/opt/course/9/holy-api-deployment.yaml`.

Answer

There are multiple ways to do this, one is to copy an *Deployment* example from <https://kubernetes.io/docs> and then merge it with the existing *Pod* yaml. That's what we will do now:

```
cp /opt/course/9/holy-api-pod.yaml /opt/course/9/holy-api-deployment.yaml # make a copy!
vim /opt/course/9/holy-api-deployment.yaml
```

Now copy/use a *Deployment* example yaml and put the *Pod*'s **metadata:** and **spec:** into the *Deployment*'s **template:** section:

```
# /opt/course/9/holy-api-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: holy-api      # name stays the same
```

```

namespace: pluto      # important
spec:
  replicas: 3          # 3 replicas
  selector:
    matchLabels:
      id: holy-api     # set the correct selector
  template:
    # => from here down its the same as the pods metadata: and spec: sections
    metadata:
      labels:
        id: holy-api
        name: holy-api
    spec:
      containers:
        - env:
            - name: CACHE_KEY_1
              value: b&MTCi0=[T66Rxm!jO@
            - name: CACHE_KEY_2
              value: PCAILGej5Ld@Q%{Q1=#
            - name: CACHE_KEY_3
              value: 2qz-]2oJlWDStN_;RFQ
            image: nginx:1.17.3-alpine
            name: holy-api-container
      volumeMounts:
        - mountPath: /cache1
          name: cache-volume1
        - mountPath: /cache2
          name: cache-volume2
        - mountPath: /cache3
          name: cache-volume3
      volumes:
        - emptyDir: {}
          name: cache-volume1
        - emptyDir: {}
          name: cache-volume2
        - emptyDir: {}
          name: cache-volume3

```

To indent multiple lines using `vim` you should set the shiftwidth using `:set shiftwidth=2`. Then mark multiple lines using `Shift + V` and the up/down keys.

To then indent the marked lines press `>` or `<` and to repeat the action press `.`

Next create the new *Deployment*:

```
k -f /opt/course/9/holy-api-deployment.yaml create
```

and confirm its running:

```

→ k -n pluto get pod | grep holy
NAME                  READY   STATUS    RESTARTS   AGE
holy-api               1/1     Running   0          19m
holy-api-5dbfdb4569-8qr5x  1/1     Running   0          30s
holy-api-5dbfdb4569-b5c1h  1/1     Running   0          30s
holy-api-5dbfdb4569-rj2gz  1/1     Running   0          30s

```

Finally delete the single *Pod*:

```
k -n pluto delete pod holy-api --force --grace-period=0
```

```

→ k -n pluto get pod,deployment | grep holy
pod/holy-api-5dbfdb4569-8qr5x  1/1     Running   0          2m4s
pod/holy-api-5dbfdb4569-b5c1h  1/1     Running   0          2m4s
pod/holy-api-5dbfdb4569-rj2gz  1/1     Running   0          2m4s

deployment.extensions/holy-api   3/3     3          3          2m4s

```

Question 10 | Service, Logs

Task weight: 4%

Team Pluto needs a new cluster internal *Service*. Create a ClusterIP *Service* named `project-plt-6cc-svc` in

Namespace `pluto`. This Service should expose a single Pod named `project-plt-6cc-api` of image `nginx:1.17.3-alpine`, create that Pod as well. The Pod should be identified by label `project: plt-6cc-api`. The Service should use tcp port redirection of `3333:80`.

Finally use for example `curl` from a temporary `nginx:alpine` Pod to get the response from the Service. Write the response into `/opt/course/10/service_test.html`. Also check if the logs of Pod `project-plt-6cc-api` show the request and write those into `/opt/course/10/service_test.log`.

Answer

```
k -n pluto run project-plt-6cc-api --image=nginx:1.17.3-alpine --labels project=plt-6cc-api
```

This will create the requested Pod. In yaml it would look like this:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    project: plt-6cc-api
    name: project-plt-6cc-api
spec:
  containers:
  - image: nginx:1.17.3-alpine
    name: project-plt-6cc-api
    resources: {}
  dnsPolicy: clusterFirst
  restartPolicy: Always
status: {}
```

Next we create the service:

```
k -n pluto expose pod -h # help

k -n pluto expose pod project-plt-6cc-api --name project-plt-6cc-svc --port 3333 --
target-port 80
```

Expose will create a yaml where everything is already set for our case and no need to change anything:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    project: plt-6cc-api
    name: project-plt-6cc-svc  # good
    namespace: pluto          # great
spec:
  ports:
  - port: 3333                # awesome
    protocol: TCP
    targetPort: 80             # nice
  selector:
    project: plt-6cc-api     # beautiful
status:
  loadBalancer: {}
```

We could also use `create service` but then we would need to change the yaml afterwards:

```
k -n pluto create service -h # help
k -n pluto create service clusterip -h #help
k -n pluto create service clusterip project-plt-6cc-svc --tcp 3333:80 $do
# now we would need to set the correct selector labels
```

Check the Service is running:

```
→ k -n pluto get pod,svc | grep 6cc
pod/project-plt-6cc-api           1/1      Running   0          9m42s
service/project-plt-6cc-svc      clusterIP  10.31.241.234  <none>        3333/TCP  2m24s
```

Does the Service has one Endpoint?

```
→ k -n pluto describe svc project-plt-6cc-svc
Name:           project-plt-6cc-svc
Namespace:      pluto
Labels:         project=plt-6cc-api
Annotations:    <none>
Selector:       project=plt-6cc-api
Type:          ClusterIP
IP:            10.3.244.240
Port:          <unset>  3333/TCP
TargetPort:    80/TCP
Endpoints:     10.28.2.32:80
Session Affinity: None
Events:        <none>
```

Or even shorter:

```
→ k -n pluto get ep
NAME          ENDPOINTS      AGE
project-plt-6cc-svc  10.28.2.32:80  84m
```

Yes, endpoint there! Finally we check the connection using a temporary *Pod*:

```
→ k run tmp --restart=Never --rm --image=nginx:alpine -i -- curl http://project-plt-6cc-svc.pluto:3333
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload Total   Spent   Left  Speed
100  612  100  612    0     0  32210      0 --:--:-- --:--:-- --:--:-- 32210
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>welcome to nginx!</h1>
...

```

Great! Notice that we use the Kubernetes *Namespace dns* resolving (`project-plt-6cc-svc.pluto`) here. We could only use the *Service* name if we would also spin up the temporary *Pod* in *Namespace pluto*.

And now really finally copy or pipe the html content into `/opt/course/10/service_test.html`.

```
# /opt/course/10/service_test.html
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
}
</style>
</head>
<body>
<h1>welcome to nginx!</h1>
...

```

Also the requested logs:

```
k -n pluto logs project-plt-6cc-api > /opt/course/10/service_test.log
```

```
# /opt/course/10/service_test.log
10.44.0.0 - - [22/Jan/2021:23:19:55 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.69.1" "-"
```

Question 11 | Labels, Annotations

Task weight: 3%

Team Sunny needs to identify some of their *Pods* in namespace `sun`. They ask you to add a new label `protected: true` to all *Pods* with an existing label `type: worker` or `type: runner`. Also add an annotation `protected: do not delete this pod` to all *Pods* having the new label `protected: true`.

Answer

```
→ k -n sun get pod --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
0509649a	1/1	Running	0	25s	type=runner,type_old=messenger
0509649b	1/1	Running	0	24s	type=worker
1428721e	1/1	Running	0	23s	type=worker
1428721f	1/1	Running	0	22s	type=worker
43b9a	1/1	Running	0	22s	type=test
4c09	1/1	Running	0	21s	type=worker
4c35	1/1	Running	0	20s	type=worker
4fe4	1/1	Running	0	19s	type=worker
5555a	1/1	Running	0	19s	type=messenger
86cda	1/1	Running	0	18s	type=runner
8d1c	1/1	Running	0	17s	type=messenger
a004a	1/1	Running	0	16s	type=runner
a94128196	1/1	Running	0	15s	type=runner,type_old=messenger
afd79200c56a	1/1	Running	0	15s	type=worker
b667	1/1	Running	0	14s	type=worker
fdb2	1/1	Running	0	13s	type=worker

If we would only like to get pods with certain labels we can run:

```
k -n sun get pod -l type=runner # only pods with label runner
```

We can use this label filtering also when using other commands, like setting new labels:

```
k label -h # help
k -n sun label pod -l type=runner protected=true # run for label runner
k -n sun label pod -l type=worker protected=true # run for label worker
```

Or we could run:

```
k -n sun label pod -l "type in (worker,runner)" protected=true
```

Let's check the result:

```
→ k -n sun get pod --show-labels
```

NAME	... AGE	LABELS
0509649a	... 56s	protected=true,type=runner,type_old=messenger
0509649b	... 55s	protected=true,type=worker
1428721e	... 54s	protected=true,type=worker
1428721f	... 53s	protected=true,type=worker
43b9a	... 53s	type=test
4c09	... 52s	protected=true,type=worker
4c35	... 51s	protected=true,type=worker
4fe4	... 50s	protected=true,type=worker
5555a	... 50s	type=messenger
86cda	... 49s	protected=true,type=runner
8d1c	... 48s	type=messenger
a004a	... 47s	protected=true,type=runner
a94128196	... 46s	protected=true,type=runner,type_old=messenger
afd79200c56a	... 46s	protected=true,type=worker
b667	... 45s	protected=true,type=worker
fdb2	... 44s	protected=true,type=worker

Looking good. Finally we set the annotation using the newly assigned label `protected: true`:

```
k -n sun annotate pod -l protected=true protected="do not delete this pod"
```

Not requested in the task but for your own control you could run:

```
k -n sun get pod -l protected=true -o yaml | grep -A 8 metadata:
```

Question 12 | Storage, PV, PVC, Pod volume

Task weight: 10%

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

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

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

Answer

```
vim 12_pv.yaml
```

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

```
# 12_pv.yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: earth-project-earthflower-pv
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/volumes/Data"
```

Then create it:

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

Next the *PersistentVolumeClaim*:

```
vim 12_pvc.yaml
```

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

```
# 12_pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: earth-project-earthflower-pvc
  namespace: earth
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Then create:

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

And check that both have the status Bound:

```
→ k -n earth get pv,pvc
NAME                                     CAPACITY   ACCESS MODES   ...   STATUS   CLAIM
persistentvolume/...earthflower-pv       2Gi        RWO          ...   Bound    ...er-pvc

NAME           STATUS   VOLUME
CAPACITY
persistentvolumeclaim/...earthflower-pvc   Bound    ...earth-project-earthflower-pv   2Gi
```

```
persistencyvolumeclaim/...earth-project-earthflower-pvc Bound earth-project/earthflower-pv 2Gi
```

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

```
k -n earth create deploy project-earthflower --image=httpd:2.4.41-alpine > 12_dep.yaml  
vim 12_dep.yaml
```

Alter the yaml to mount the volume:

```
# 12_dep.yaml  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  creationTimestamp: null  
  labels:  
    app: project-earthflower  
    name: project-earthflower  
    namespace: earth  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: project-earthflower  
  strategy: {}  
  template:  
    metadata:  
      creationTimestamp: null  
      labels:  
        app: project-earthflower  
    spec:  
      volumes: # add  
      - name: data # add  
        persistentVolumeClaim: # add  
          claimName: earth-project-earthflower-pvc # add  
      containers:  
      - image: httpd:2.4.41-alpine  
        name: container  
        volumeMounts: # add  
        - name: data # add  
          mountPath: /tmp/project-data # add
```

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

We can confirm its mounting correctly:

```
→ k -n earth describe pod project-earthflower-d6887f7c5-pn5wv | grep -A2 Mounts:  
Mounts:  
  /tmp/project-data from data (rw) # there it is  
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-n2sjj (ro)
```

Question 13 | Storage, StorageClass, PVC

Task weight: 7%

Team Moonpie, which has the *Namespace* `moon`, needs more storage. Create a new *PersistentVolumeClaim* named `moon-pvc-126` in that namespace. This claim should use a new *StorageClass* `moon-retain` with the *provisioner* set to `moon-retainer` and the *reclaimPolicy* set to *Retain*. The claim should request storage of `3Gi`, an *accessMode* of `ReadWriteOnce` and should use the new *StorageClass*.

The provisioner `moon-retainer` will be created by another team, so it's expected that the *PVC* will not boot yet. Confirm this by writing the log message from the *PVC* into file `/opt/course/13/pvc-126-reason`.

Answer

```
vim 13_sc.yaml
```

Head to <https://kubernetes.io/docs>, search for "storageclass" and alter the example code to this:

```
# 13_sc.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: moon-retain
  provisioner: moon-retainer
  reclaimPolicy: Retain
```

```
kubectl create -f 13_sc.yaml
```

Now the same for the *PersistentVolumeClaim*, head to the docs, copy an example and transform it into:

```
vim 13_pvc.yaml
```

```
# 13_pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: moon-pvc-126          # name as requested
  namespace: moon              # important
spec:
  accessModes:
    - ReadWriteOnce           # RWO
  resources:
    requests:
      storage: 3Gi            # size
  storageClassName: moon-retain # uses our new storage class
```

```
kubectl create -f 13_pvc.yaml
```

Next we check the status of the *PVC*:

```
→ kubectl -n moon get pvc
NAME        STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
moon-pvc-126 Pending
```

```
→ kubectl -n moon describe pvc moon-pvc-126
Name:          moon-pvc-126
...
Status:        Pending
...
Events:
...
waiting for a volume to be created, either by external provisioner "moon-retainer" or
manually created by system administrator
```

This confirms that the *PVC* waits for the provisioner `moon-retainer` to be created. Finally we copy or write the event message into the requested location:

```
# /opt/course/13/pvc-126-reason
waiting for a volume to be created, either by external provisioner "moon-retainer" or
manually created by system administrator
```

Question 14 | Secret, Secret-Volume, Secret-Env

Task weight: 4%

You need to make changes on an existing *Pod* in *Namespace* `moon` called `secret-handler`. Create a new *Secret* `secret1` which contains `user=test` and `pass=pwd`. The *Secret*'s content should be available in *Pod* `secret-handler` as environment variables `SECRET1_USER` and `SECRET1_PASS`. The yaml for *Pod* `secret-handler` is available at `/opt/course/14/secret-handler.yaml`.

There is existing yaml for another *Secret* at `/opt/course/14/secret2.yaml`, create this *Secret* and mount it inside the same *Pod* at `/tmp/secret2`. Your changes should be saved under `/opt/course/14/secret-handler-new.yaml`. Both *Secrets* should only be available in *Namespace* `moon`.

Answer

```
k -n moon get pod # show pods
k -n moon create secret -h # help
k -n moon create secret generic -h # help
k -n moon create secret generic secret1 --from-literal user=test --from-literal pass=pwd
```

The last command would generate this yaml:

```
apiVersion: v1
data:
  pass: cHdk
  user: dGvzdA==
kind: Secret
metadata:
  creationTimestamp: null
  name: secret1
  namespace: moon
```

Next we create the second *Secret* from the given location, making sure it'll be created in *Namespace moon*:

```
k -n moon -f /opt/course/14/secret2.yaml create
```

```
→ k -n moon get secret
NAME          TYPE        DATA   AGE
default-token-rvzcf  kubernetes.io/service-account-token  3       66m
secret1        Opaque      2      4m3s
secret2        Opaque      1      8s
```

We will now edit the *Pod* yaml:

```
cp /opt/course/14/secret-handler.yaml /opt/course/14/secret-handler-new.yaml
vim /opt/course/14/secret-handler-new.yaml
```

Add the following to the yaml:

```
# /opt/course/14/secret-handler-new.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    id: secret-handler
    uid: 1428721e-8d1c-4c09-b5d6-af79200c56a
    red_ident: 9cf7a7c0-fdb2-4c35-9c13-c2a0bb52b4a9
    type: automatic
  name: secret-handler
  namespace: moon
spec:
  volumes:
    - name: cache-volume1
      emptyDir: {}
    - name: cache-volume2
      emptyDir: {}
    - name: cache-volume3
      emptyDir: {}
    - name: secret2-volume          # add
      secret:                  # add
        secretName: secret2      # add
  containers:
    - name: secret-handler
      image: bash:5.0.11
      args: ['bash', '-c', 'sleep 2d']
      volumeMounts:
        - mountPath: /cache1
          name: cache-volume1
        - mountPath: /cache2
          name: cache-volume2
        - mountPath: /cache3
          name: cache-volume3
        - name: secret2-volume      # add
          mountPath: /tmp/secret2      # add
  env:
    - name: SECRET_KEY_1
      value: ">8$KH#kj..i8}HIMQd"
    - name: SECRET_KEY_2
      value: "I0=a4L/XkRdvN8jM=Y+"
    - name: SECRET_KEY_3
      value: "-7PA0_Z]>{pwa43r)___"
```

```

- name: SECRET1_USER          # add
  valueFrom:
    secretKeyRef:
      name: secret1           # add
      key: user                # add
- name: SECRET1_PASS          # add
  valueFrom:
    secretKeyRef:
      name: secret1           # add
      key: pass                # add

```

There is also the possibility to import all keys from a *Secret* as env variables at once, though the env variable names will then be the same as in the *Secret*, which doesn't work for the requirements here:

```

containers:
- name: secret-handler
...
envFrom:
- secretRef:      # also works for configMapRef
  name: secret1

```

Then we apply the changes:

```
k -f /opt/course/14/secret-handler.yaml delete --force --grace-period=0
k -f /opt/course/14/secret-handler-new.yaml create
```

Instead of running delete and create we can also use recreate:

```
k -f /opt/course/14/secret-handler-new.yaml replace --force --grace-period=0
```

It was not requested directly, but you should always confirm its working:

```

→ k -n moon exec secret-handler -- env | grep SECRET1
SECRET1_USER=test
SECRET1_PASS=pwd

→ k -n moon exec secret-handler -- find /tmp/secret2
/tmp/secret2
/tmp/secret2/..data
/tmp/secret2/key
/tmp/secret2/..2019_09_11_09_03_08.147048594
/tmp/secret2/..2019_09_11_09_03_08.147048594/key

→ k -n moon exec secret-handler -- cat /tmp/secret2/key
12345678

```

Question 15 | ConfigMap, Configmap-Volume

Task weight: 5%

Team Moonpie has a nginx server *Deployment* called `web-moon` in *Namespace* `moon`. Someone started configuring it but it was never completed. To complete please create a *ConfigMap* called `configmap-web-moon-html` containing the content of file `/opt/course/15/web-moon.html` under the data key-name `index.html`.

The *Deployment* `web-moon` is already configured to work with this *ConfigMap* and serve its content. Test the nginx configuration for example using `curl` from a temporary `nginx:alpine` *Pod*.

Answer

Let's check the existing *Pods*:

```

→ k -n moon get pod
NAME        READY   STATUS      RESTARTS   AGE
secret-handler   1/1     Running      0          55m
web-moon-847496c686-2rzj4   0/1     ContainerCreating   0          33s
web-moon-847496c686-9nwwj   0/1     ContainerCreating   0          33s
web-moon-847496c686-cxdbx   0/1     ContainerCreating   0          33s

```

```
web-moon-847496c686-hvqlw 0/1 ContainerCreating 0 33s
web-moon-847496c686-tj7ct 0/1 ContainerCreating 0 33s
```

```
→ k -n moon describe pod web-moon-847496c686-2rzj4
...
Warning FailedMount 31s (x7 over 63s) kubelet, gke-test-default-pool-ce83a51a-p6s4
MountVolume.SetUp failed for volume "html-volume" : configmaps "configmap-web-moon-html"
not found
```

Good so far, now let's create the missing *ConfigMap*:

```
k -n moon create configmap -h # help
k -n moon create configmap configmap-web-moon-html --from-
file=index.html=/opt/course/15/web-moon.html # important to set the index.html key
```

This should create a *ConfigMap* with yaml like:

```
apiVersion: v1
data:
  index.html: |    # notice the key index.html, this will be the filename when mounted
    <!DOCTYPE html>
    <html lang="en">
      <head>
        <meta charset="UTF-8">
        <title>Web Moon Webpage</title>
      </head>
      <body>
        This is some great content.
      </body>
    </html>
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: configmap-web-moon-html
  namespace: moon
```

After waiting a bit or deleting/recreating (`k -n moon rollout restart deploy web-moon`) the *Pods* we should see:

```
→ k -n moon get pod
NAME          READY   STATUS    RESTARTS   AGE
secret-handler 1/1     Running   0          59m
web-moon-847496c686-2rzj4 1/1     Running   0          4m28s
web-moon-847496c686-9nwwj 1/1     Running   0          4m28s
web-moon-847496c686-cxdbx 1/1     Running   0          4m28s
web-moon-847496c686-hvqlw 1/1     Running   0          4m28s
web-moon-847496c686-tj7ct 1/1     Running   0          4m28s
```

Looking much better. Finally we check if the nginx returns the correct content:

```
k -n moon get pod -o wide # get pod cluster IPs
```

Then use one IP to test the configuration:

```
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl 10.44.0.78
% Total    % Received % Xferd  Average Speed   Time   Time   Current
          Dload  Upload Total   Spent    Left  Speed
100  161  100  161    0     0  80500      0  --:--:--  --:--:-- 157k
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Web Moon webpage</title>
  </head>
  <body>
    This is some great content.
  </body>
```

For debugging or further checks we could find out more about the *Pods* volume mounts:

```
→ k -n moon describe pod web-moon-c77655cc-dc8v4 | grep -A2 Mounts:
Mounts:
  /usr/share/nginx/html from html-volume (rw)
  /var/run/secrets/kubernetes.io/serviceaccount from default-token-rvzcf (ro)
```

And check the mounted folder content:

```
→ k -n moon exec web-moon-c77655cc-dc8v4 find /usr/share/nginx/html
/usr/share/nginx/html
/usr/share/nginx/html/..2019_09_11_10_05_56.336284411
/usr/share/nginx/html/..2019_09_11_10_05_56.336284411/index.html
/usr/share/nginx/html/..data
/usr/share/nginx/html/index.html
```

Here it was important that the file will have the name `index.html` and not the original one `web-moon.html` which is controlled through the `ConfigMap` data key.

Question 16 | Logging sidecar

Task weight: 6%

The Tech Lead of Mercury2D decided its time for more logging, to finally fight all these missing data incidents. There is an existing container named `cleaner-con` in Deployment `cleaner` in Namespace `mercury`. This container mounts a volume and writes logs into a file called `cleaner.log`.

The yaml for the existing Deployment is available at `/opt/course/16/cleaner.yaml`. Persist your changes at `/opt/course/16/cleaner-new.yaml` but also make sure the Deployment is running.

Create a sidecar container named `logger-con`, image `busybox:1.31.0`, which mounts the same volume and writes the content of `cleaner.log` to stdout, you can use the `tail -f` command for this. This way it can be picked up by `kubectl logs`.

Check if the logs of the new container reveal something about the missing data incidents.

Answer

```
cp /opt/course/16/cleaner.yaml /opt/course/16/cleaner-new.yaml
vim /opt/course/16/cleaner-new.yaml
```

Add a sidecar container which outputs the log file to stdout:

```
# /opt/course/16/cleaner-new.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  name: cleaner
  namespace: mercury
spec:
  replicas: 2
  selector:
    matchLabels:
      id: cleaner
  template:
    metadata:
      labels:
        id: cleaner
    spec:
      volumes:
        - name: logs
          emptyDir: {}
      initContainers:
        - name: init
          image: bash:5.0.11
          command: ['bash', '-c', 'echo init > /var/log/cleaner/cleaner.log']
          volumeMounts:
            - name: logs
              mountPath: /var/log/cleaner
      containers:
        - name: cleaner-con
          image: bash:5.0.11
          args: ['bash', '-c', 'while true; do echo `date` : "remove random file" >> /var/log/cleaner/cleaner.log; sleep 1; done']
          volumeMounts:
```

```

- name: logs
  mountPath: /var/log/cleaner
- name: logger-con
  image: busybox:1.31.0
  command: ["sh", "-c", "tail -f /var/log/cleaner/cleaner.log"]
  volumeMounts:
- name: logs
  mountPath: /var/log/cleaner

```

Then apply the changes and check the logs of the sidecar:

```
k -f /opt/course/16/cleaner-new.yaml apply
```

This will cause a deployment rollout of which we can get more details:

```

k -n mercury rollout history deploy cleaner
k -n mercury rollout history deploy cleaner --revision 1
k -n mercury rollout history deploy cleaner --revision 2

```

Check Pod statuses:

```

→ k -n mercury get pod
NAME          READY   STATUS    RESTARTS   AGE
cleaner-86b7758668-9pw6t  2/2     Running   0          6s
cleaner-86b7758668-qgh4v  0/2     Init:0/1  0          1s

→ k -n mercury get pod
NAME          READY   STATUS    RESTARTS   AGE
cleaner-86b7758668-9pw6t  2/2     Running   0          14s
cleaner-86b7758668-qgh4v  2/2     Running   0          9s

```

Finally check the logs of the logging sidecar container:

```

→ k -n mercury logs cleaner-576967576c-cqtgx -c logger-con
init
wed Sep 11 10:45:44 UTC 2099: remove random file
wed Sep 11 10:45:45 UTC 2099: remove random file
...

```

Mystery solved, something is removing files at random ;) It's important to understand how containers can communicate with each other using volumes.

Question 17 | InitContainer

Task weight: 4%

Last lunch you told your coworker from department Mars Inc how amazing *InitContainers* are. Now he would like to see one in action. There is a *Deployment* yaml at `/opt/course/17/test-init-container.yaml`. This *Deployment* spins up a single *Pod* of image `nginx:1.17.3-alpine` and serves files from a mounted volume, which is empty right now.

Create an *InitContainer* named `init-con` which also mounts that volume and creates a file `index.html` with content `check this out!` in the root of the mounted volume. For this test we ignore that it doesn't contain valid html.

The *InitContainer* should be using image `busybox:1.31.0`. Test your implementation for example using `curl` from a temporary `nginx:alpine` *Pod*.

Answer

```

cp /opt/course/17/test-init-container.yaml ~/17_test-init-container.yaml
vim 17_test-init-container.yaml

```

Add the *InitContainer*:

```

# 17_test-init-container.yaml
apiVersion: apps/v1

```

```

kind: Deployment
metadata:
  name: test-init-container
  namespace: mars
spec:
  replicas: 1
  selector:
    matchLabels:
      id: test-init-container
  template:
    metadata:
      labels:
        id: test-init-container
    spec:
      volumes:
        - name: web-content
          emptyDir: {}
      initContainers:           # initContainer start
        - name: init-con
          image: busybox:1.31.0
          command: ['sh', '-c', 'echo "check this out!" > /tmp/web-content/index.html']
          volumeMounts:
            - name: web-content
              mountPath: /tmp/web-content # initContainer end
      containers:
        - image: nginx:1.17.3-alpine
          name: nginx
          volumeMounts:
            - name: web-content
              mountPath: /usr/share/nginx/html
          ports:
            - containerPort: 80

```

Then we create the *Deployment*:

```
k -f 17_test-init-container.yaml create
```

Finally we test the configuration:

```

k -n mars get pod -o wide # to get the cluster IP

→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl 10.0.0.67
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload   Total   Spent    Left  Speed
check this out!

```

Beautiful.

Question 18 | Service misconfiguration

Task weight: 5%

There seems to be an issue in *Namespace mars* where the ClusterIP service `manager-api-svc` should make the *Pods* of *Deployment manager-api-deployment* available inside the cluster.

You can test this with `curl manager-api-svc.mars:4444` from a temporary `nginx:alpine` *Pod*. Check for the misconfiguration and apply a fix.

Answer

First let's get an overview:

```

→ k -n mars get all
NAME                               READY   STATUS    RESTARTS   AGE
pod/manager-api-deployment-dbcc6657d-bg2hh  1/1     Running   0          98m
pod/manager-api-deployment-dbcc6657d-f5fv4  1/1     Running   0          98m
pod/manager-api-deployment-dbcc6657d-httjv  1/1     Running   0          98m
pod/manager-api-deployment-dbcc6657d-k98xn  1/1     Running   0          98m
pod/test-init-container-5db7c99857-htx6b   1/1     Running   0          2m19s

```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/manager-api-svc	ClusterIP	10.15.241.159	<none>	4444/TCP	99m
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/manager-api-deployment	4/4	4	4	98m	
deployment.apps/test-init-container	1/1	1	1	2m19s	
...					

Everything seems to be running, but we can't seem to get a connection:

```
→ k -n mars run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 manager-api-svc:4444
If you don't see a command prompt, try pressing enter.
      0      0      0      0      0      0      0 --::--- 0:00:01 --::---      0
curl: (28) Connection timed out after 1000 milliseconds
pod "tmp" deleted
pod mars/tmp terminated (Error)
```

Ok, let's try to connect to one pod directly:

```
k -n mars get pod -o wide # get cluster IP
```

```
→ k -n mars run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 10.0.1.14
% Total    % Received % Xferd  Average Speed   Time     Time     Current
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

The *Pods* itself seem to work. Let's investigate the *Service* a bit:

```
→ k -n mars describe service manager-api-svc
Name:           manager-api-svc
Namespace:      mars
Labels:         app=manager-api-svc
...
Endpoints:     <none>
...
```

Endpoint inspection is also possible using:

```
k -n mars get ep
```

No endpoints - No good. We check the *Service* yaml:

```
k -n mars edit service manager-api-svc
```

```
# k -n mars edit service manager-api-svc
apiVersion: v1
kind: Service
metadata:
...
labels:
  app: manager-api-svc
  name: manager-api-svc
  namespace: mars
...
spec:
  clusterIP: 10.3.244.121
  ports:
  - name: 4444-80
    port: 4444
    protocol: TCP
    targetPort: 80
  selector:
    #id: manager-api-deployment # wrong selector, needs to point to pod!
    id: manager-api-pod
  sessionAffinity: None
  type: ClusterIP
```

Though *Pods* are usually never created without a *Deployment* or *ReplicaSet*, *Services* always select for *Pods* directly. This gives great flexibility because *Pods* could be created through various customized ways. After saving the new selector we check the *Service* again for endpoints:

```
→ k -n mars get ep
NAME           ENDPOINTS          AGE
manager-api-svc  10.0.0.30:80,10.0.1.30:80,10.0.1.31:80 + 1 more...  41m
```

Endpoints - Good! Now we try connecting again:

```
→ k -n mars run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 manager-api-svc:4444
% Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
          Dload  Upload   Total Spent  Left  Speed
100  612  100  612    0     0   99k      0  --:--:--  --:--:--  --:--:--   99k
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

And we fixed it. Good to know is how to be able to use Kubernetes DNS resolution from a different *Namespace*. Not necessary, but we could spin up the temporary *Pod* in default *Namespace*:

```
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 manager-api-svc:4444
% Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
          Dload  Upload   Total Spent  Left  Speed
0     0     0     0     0     0      0  --:--:--  --:--:--  --:--:--  0curl: (6)
could not resolve host: manager-api-svc
pod "tmp" deleted
pod default/tmp terminated (Error)

→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 manager-api-svc.mars:4444
% Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
          Dload  Upload   Total Spent  Left  Speed
100  612  100  612    0     0  68000      0  --:--:--  --:--:--  --:--:--  68000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Short `manager-api-svc.mars` or long `manager-api-svc.mars.svc.cluster.local` work.

Question 19 | Service ClusterIP->NodePort

Task weight: 3%

In *Namespace jupiter* you'll find an apache *Deployment* (with one replica) named `jupiter-crew-deploy` and a ClusterIP *Service* called `jupiter-crew-svc` which exposes it. Change this service to a NodePort one to make it available on all nodes on port 30100.

Test the NodePort *Service* using the internal IP of all available nodes and the port 30100 using `curl`, you can reach the internal node IPs directly from your main terminal. On which nodes is the *Service* reachable? On which node is the *Pod* running?

Answer

First we get an overview:

```
→ k -n jupiter get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/jupiter-crew-deploy-8cdf99bc9-k1wqt  1/1     Running   0          34m

NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/jupiter-crew-svc  ClusterIP  10.100.254.66  <none>       8080/TCP  34m
...
```

(Optional) Next we check if the ClusterIP *Service* actually works:

```
→ k -n jupiter run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 jupiter-crew-svc:8080
  % Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
     Dload  Upload Total   Spent   Left  Speed
100    45  100    45    0      0  5000      0 --:--:-- --:--:-- --:--:--  5000
<html><body><h1>It works!</h1></body></html>
```

The Service is working great. Next we change the *Service* type to *NodePort* and set the port:

```
k -n jupiter edit service jupiter-crew-svc
```

```
# k -n jupiter edit service jupiter-crew-svc
apiVersion: v1
kind: Service
metadata:
  name: jupiter-crew-svc
  namespace: jupiter
...
spec:
  clusterIP: 10.3.245.70
  ports:
    - name: 8080-80
      port: 8080
      protocol: TCP
      targetPort: 80
      nodePort: 30100 # add the nodePort
  selector:
    id: jupiter-crew
  sessionAffinity: None
  #type: ClusterIP
  type: NodePort      # change type
status:
  loadBalancer: []
```

We check if the *Service* type was updated:

```
→ k -n jupiter get svc
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)           AGE
jupiter-crew-svc  NodePort  10.3.245.70  <none>        8080:30100/TCP  3m52s
```

(Optional) And we confirm that the service is still reachable internally:

```
→ k -n jupiter run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 jupiter-crew-svc:8080
  % Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
     Dload  Upload Total   Spent   Left  Speed
<html><body><h1>It works!</h1></body></html>
```

Nice. A *NodePort Service* kind of lies on top of a *ClusterIP* one, making the *ClusterIP Service* reachable on the Node IPs (internal and external). Next we get the *internal* IPs of all nodes to check the connectivity:

```
→ k get nodes -o wide
NAME          STATUS  ROLES   AGE   VERSION  INTERNAL-IP      ...
cluster1-master1  Ready  master  18h   v1.21.0  192.168.100.11  ...
cluster1-worker1  Ready  <none>  18h   v1.21.0  192.168.100.12  ...
```

On which nodes is the *Service* reachable?

```
→ curl 192.168.100.11:30100
<html><body><h1>It works!</h1></body></html>

→ curl 192.168.100.12:30100
<html><body><h1>It works!</h1></body></html>
```

On both, even the master. On which node is the *Pod* running?

```
→ k -n jupiter get pod jupiter-crew-deploy-8cdf99bc9-klwqt -o yaml | grep nodeName
  nodeName: cluster1-worker1

→ k -n jupiter get pod -o wide # or even shorter
```

In our case on *cluster1-worker1*, but could be any other worker if more available. Here we hopefully gained some insight into how a *NodePort Service* works. Although the *Pod* is just running on one specific node, the *Service* makes it available through port 30100 on the internal and external IP addresses of all nodes. This is

at least the common/default behaviour but can depend on cluster configuration.

Question 20 | NetworkPolicy

Task weight: 12%

In Namespace `venus` you'll find two *Deployments* named `api` and `frontend`. Both *Deployments* are exposed inside the cluster using *Services*. Create a *NetworkPolicy* named `np1` which restricts outgoing tcp connections from *Deployment* `frontend` and only allows those going to *Deployment* `api`. Make sure the *NetworkPolicy* still allows outgoing traffic on UDP/TCP ports 53 for DNS resolution.

Test using: `wget www.google.com` and `wget api:2222` from a *Pod* of *Deployment* `frontend`.

Answer

First we get an overview:

```
→ k -n venus get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/api-5979b95578-gktxp     1/1     Running   0          57s
pod/api-5979b95578-lhc15     1/1     Running   0          57s
pod/frontend-789cbdc677-c9v8h 1/1     Running   0          57s
pod/frontend-789cbdc677-npk2m 1/1     Running   0          57s
pod/frontend-789cbdc677-p167g 1/1     Running   0          57s
pod/frontend-789cbdc677-rjt5r 1/1     Running   0          57s
pod/frontend-789cbdc677-xgf5n 1/1     Running   0          57s

NAME            TYPE        CLUSTER-IP      EXTERNAL-IP    PORT(S)    AGE
service/api     ClusterIP   10.3.255.137   <none>        2222/TCP   37s
service/frontend ClusterIP   10.3.255.135   <none>        80/TCP     57s
...
```

(Optional) This is not necessary but we could check if the *Services* are working inside the cluster:

```
→ k -n venus run tmp --restart=Never --rm -i --image=busybox -i -- wget -O- frontend:80
Connecting to frontend:80 (10.3.245.9:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
<body><h1>It works!</h1></body></html>

→ k -n venus run tmp --restart=Never --rm --image=busybox -i -- wget -O- api:2222
Connecting to api:2222 (10.3.250.233:2222)
<html><body><h1>It works!</h1></body></html>
```

Then we use any `frontend` *Pod* and check if it can reach external names and the `api` *Service*:

```
→ k -n venus exec frontend-789cbdc677-c9v8h -- wget -O- www.google.com
Connecting to www.google.com (216.58.205.227:80)
-                                     100% |*****| 12955 0:00:00 ETA
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head>
...
<body><h1>It works!</h1></body></html>

→ k -n venus exec frontend-789cbdc677-c9v8h -- wget -O- api:2222
<html><body><h1>It works!</h1></body></html>
Connecting to api:2222 (10.3.255.137:2222)
-                                     100% |*****| 45 0:00:00 ETA
...
```

We see *Pods* of `frontend` can reach the `api` and external names.

```
vim 20_np1.yaml
```

Now we head to <https://kubernetes.io/docs>, search for *NetworkPolicy*, copy the example code and adjust it to:

```
# 20_np1.yaml
```

```
# <0_0_ip1.yaml>
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np1
  namespace: venus
spec:
  podSelector:
    matchLabels:
      id: frontend          # label of the pods this policy should be applied on
  policyTypes:
  - Egress           # we only want to control egress
  egress:
  - to:
    - podSelector:
        matchLabels:
          id: api          # 1st egress rule
        # allow egress only to pods with api label
    - ports:
      - port: 53          # 2nd egress rule
        protocol: UDP   # allow DNS UDP
      - port: 53          # allow DNS TCP
        protocol: TCP
```

Notice that we specify two egress rules in the yaml above. If we specify multiple egress rules then these are connected using a logical OR. So in the example above we do:

```
allow outgoing traffic if  
  (destination pod has label id:api) OR ((port is 53 UDP) OR (port is 53 TCP))
```

Let's have a look at example code which wouldn't work in our case:

```
# this example does not work in our case
...
egress:
- to:                      # 1st AND ONLY egress rule
  - podSelector:           # allow egress only to pods with api label
    matchLabels:
      id: api
  ports:                   # STILL THE SAME RULE but just an additional selector
- port: 53                  # allow DNS UDP
  protocol: UDP
- port: 53                  # allow DNS TCP
  protocol: TCP
```

In the yaml above we only specify one egress rule with two selectors. It can be translated into:

```
allow outgoing traffic if  
  (destination pod has label id:api) AND ((port is 53 UDP) OR (port is 53 TCP))
```

Apply the correct policy:

```
k -f 20_np1.yaml create
```

And try again, external is not working any longer:

```
→ k -n venus exec frontend-789cbdc677-c9v8h -- wget -O- www.google.de
Connecting to www.google.de:2222 (216.58.207.67:80)
^C

→ k -n venus exec frontend-789cbdc677-c9v8h -- wget -O- -T 5 www.google.de:80
Connecting to www.google.com (172.217.203.104:80)
wget: download timed out
command terminated with exit code 1
```

Internal connection to `api` work as before:

CKAD Simulator Preview Kubernetes 1.21

<https://killer.sh>

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

The full course contains 20 questions and scenarios which cover all the CKAD 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 not part of the 20 in the full course but in addition to it. But the preview questions are part of the same CKAD 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="--dry-run=client -o yaml"
```

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

Preview Question 1

In Namespace `pluto` there is a *Deployment* named `project-23-api`. It has been working okay for a while but Team Pluto needs it to be more reliable. Implement a liveness-probe which checks the container to be reachable on port 80. Initially the probe should wait 10, periodically 15 seconds.

The original *Deployment* yaml is available at `/opt/course/p1/project-23-api.yaml`. Save your changes at `/opt/course/p1/project-23-api-new.yaml` and apply the changes.

Answer

First we get an overview:

```
→ k -n pluto get all -o wide  
NAME                               READY   STATUS    ... IP      ...  
pod/holy-api                      1/1     Running   ... 10.12.0.26  ...  
pod/project-23-api-784857f54c-dx6h6 1/1     Running   ... 10.12.2.15  ...  
pod/project-23-api-784857f54c-sj8df 1/1     Running   ... 10.12.1.18  ...  
pod/project-23-api-784857f54c-t4xmh 1/1     Running   ... 10.12.0.23  ...  
  
NAME                           READY   UP-TO-DATE   AVAILABLE   ...  
deployment.apps/project-23-api   3/3     3           3          ...
```

To note: we see another *Pod* here called `holy-api` which is part of another section. This is often the case in the provided scenarios, so be careful to only manipulate the resources you need to. Just like in the real world and in the exam.

Next we use `nginx:alpine` and `curl` to check if one *Pod* is accessible on port 80:

```
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 10.12.2.15  
% Total % Received % Xferd Average Speed Time Time Current  
          Dload Upload Total Spent Left Speed  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
...
```

We could also use `busybox` and `wget` for this:

```
→ k run tmp --restart=Never --rm --image=busybox -i -- wget -O- 10.12.2.15  
Connecting to 10.12.2.15 (10.12.2.15:80)  
writing to stdout  
100% [*****] 0.00-0.00 ETA
```

```

written to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Now that we're sure the *Deployment* works we can continue with altering the provided yaml:

```

cp /opt/course/p1/project-23-api.yaml /opt/course/p1/project-23-api-new.yaml
vim /opt/course/p1/project-23-api-new.yaml
```

Add the liveness-probe to the yaml:

```

# /opt/course/p1/project-23-api-new.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: project-23-api
  namespace: pluto
spec:
  replicas: 3
  selector:
    matchLabels:
      app: project-23-api
  template:
    metadata:
      labels:
        app: project-23-api
    spec:
      volumes:
        - name: cache-volume1
          emptyDir: {}
        - name: cache-volume2
          emptyDir: {}
        - name: cache-volume3
          emptyDir: {}
      containers:
        - image: httpd:2.4-alpine
          name: httpd
          volumeMounts:
            - mountPath: /cache1
              name: cache-volume1
            - mountPath: /cache2
              name: cache-volume2
            - mountPath: /cache3
              name: cache-volume3
          env:
            - name: APP_ENV
              value: "prod"
            - name: APP_SECRET_N1
              value: "IO=a4L/XkRdvN8jM=Y+"
            - name: APP_SECRET_P1
              value: "-7PA0_Z]>{pwa43r)___"
          livenessProbe: # add
            tcpSocket: # add
              port: 80 # add
            initialDelaySeconds: 10 # add
            periodSeconds: 15 # add
```

Then let's apply the changes:

```
k -f /opt/course/p1/project-23-api-new.yaml apply
```

Next we wait 10 seconds and confirm the *Pods* are still running:

```

→ k -n pluto get pod
NAME           READY   STATUS    RESTARTS   AGE
holy-api       1/1     Running   0          144m
project-23-api-5b4579fd49-8knh8  1/1     Running   0          90s
project-23-api-5b4579fd49-cbgph  1/1     Running   0          88s
project-23-api-5b4579fd49-tcfq5  1/1     Running   0          86s
```

We can also check the configured liveness-probe settings on a *Pod* or the *Deployment*:

```

→ k -n pluto describe pod project-23-api-5b4579fd49-8knh8 | grep Liveness
  Liveness:  tcp-socket :80 delay=10s timeout=1s period=15s #success=1 #failure=3
```

```
→ k -n pluto describe deploy project-23-api | grep Liveness
  Liveness:  tcp-socket :80 delay=10s timeout=1s period=15s #success=1 #failure=3
```

Preview Question 2

Team Sun needs a new *Deployment* named `sunny` with 4 replicas of image `nginx:1.17.3-alpine` in Namespace `sun`. The *Deployment* and its *Pods* should use the existing *ServiceAccount* `sa-sun-deploy`.

Expose the *Deployment* internally using a ClusterIP *Service* named `sun-srv` on port 9999. The nginx containers should run as default on port 80. The management of Team Sun would like to execute a command to check that all *Pods* are running on occasion. Write that command into file `/opt/course/p2/sunny_status_command.sh`. The command should use `kubectl`.

Answer

```
k -n sun create deployment -h #help

# check the export on the very top of this document so we can use $do
k -n sun create deployment sunny --image=nginx:1.17.3-alpine $do > p2_sunny.yaml

vim p2_sunny.yaml
```

Then alter its yaml to include the requirements:

```
# p2_sunny.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: sunny
    name: sunny
    namespace: sun
spec:
  replicas: 4          # change
  selector:
    matchLabels:
      app: sunny
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: sunny
    spec:
      serviceAccountName: sa-sun-deploy      # add
      containers:
        - image: nginx:1.17.3-alpine
          name: nginx
          resources: {}
  status: {}
```

Now create the yaml and confirm its running:

```
→ k create -f p2_sunny.yaml
deployment.apps/sunny created

→ k -n sun get pod
NAME           READY   STATUS    RESTARTS   AGE
0509649a       1/1     Running   0          149m
0509649b       1/1     Running   0          149m
1428721e       1/1     Running   0          149m
...
sunny-64df8dbdbb-9mxbw  1/1     Running   0          10s
sunny-64df8dbdbb-mp5cf  1/1     Running   0          10s
sunny-64df8dbdbb-pggdf  1/1     Running   0          6s
sunny-64df8dbdbb-zvqth  1/1     Running   0          7s
```

Confirmed, the AGE column is always an important information about if changes were applied. Next we expose the *Pods* by created the *Service*:

```
k -n sun expose -h # help
```

```
k -n sun expose deployment sunny --name sun-srv --port 9999 --target-port 80
```

Using expose instead of `kubectl create service clusterip` is faster because it already sets the correct selector-labels. The previous command would produce this yaml:

```
# k -n sun expose deployment sunny --name sun-srv --port 9999 --target-port 80
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: sunny
    name: sun-srv      # required by task
spec:
  ports:
  - port: 9999        # service port
    protocol: TCP
    targetPort: 80    # target port
  selector:
    app: sunny       # selector is important
status:
  loadBalancer: {}
```

Let's test the *Service* using `wget` from a temporary *Pod*:

```
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 sun-srv.sun:9999
Connecting to sun-srv.sun:9999 (10.23.253.120:9999)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Because the *Service* is in a different *Namespace* as our temporary *Pod*, it is reachable using the names `sun-srv.sun` or fully: `sun-srv.sun.svc.cluster.local`.

Finally we need a command which can be executed to check if all *Pods* are running, this can be done with:

```
vim /opt/course/p2/sunny_status_command.sh
```

```
# /opt/course/p2/sunny_status_command.sh
kubectl -n sun get deployment sunny
```

To run the command:

```
→ sh /opt/course/p2/sunny_status_command.sh
NAME     READY   UP-TO-DATE   AVAILABLE   AGE
sunny   4/4     4           4           13m
```

Preview Question 3

Management of EarthAG recorded that one of their *Services* stopped working. Dirk, the administrator, left already for the long weekend. All the information they could give you is that it was located in *Namespace earth* and that it stopped working after the latest rollout. All *Services* of EarthAG should be reachable from inside the cluster.

Find the *Service*, fix any issues and confirm its working again. Write the reason of the error into file `/opt/course/p3/ticket-654.txt` so Dirk knows what the issue was.

Answer

First we get an overview of the resources in *Namespace earth*:

```
→ k -n earth get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/earth-2x3-api-584df69757-ngnwp   1/1     Running   0          116m
pod/earth-2x3-api-584df69757-ps8cs   1/1     Running   0          116m
pod/earth-2x3-api-584df69757-ww9q8   1/1     Running   0          116m
pod/earth-2x3-web-85c5b7986c-48vjt   1/1     Running   0          116m
pod/earth-2x3-web-85c5b7986c-6mqmb   1/1     Running   0          116m
pod/earth-2x3-web-85c5b7986c-6vj11   1/1     Running   0          116m
```

pod/earth-2x3-web-85c5b7986c-fnkbp	1/1	Running	0	116m	
pod/earth-2x3-web-85c5b7986c-pjm5m	1/1	Running	0	116m	
pod/earth-2x3-web-85c5b7986c-pwfvj	1/1	Running	0	116m	
pod/earth-3cc-runner-6cb6cc6974-8wm5x	1/1	Running	0	116m	
pod/earth-3cc-runner-6cb6cc6974-9fx8b	1/1	Running	0	116m	
pod/earth-3cc-runner-6cb6cc6974-b9nrw	1/1	Running	0	116m	
pod/earth-3cc-runner-heavy-6bf876f46d-b47vq	1/1	Running	0	116m	
pod/earth-3cc-runner-heavy-6bf876f46d-mrzqd	1/1	Running	0	116m	
pod/earth-3cc-runner-heavy-6bf876f46d-qkd74	1/1	Running	0	116m	
pod/earth-3cc-web-6bfd8b848-f74cj	0/1	Running	0	116m	
pod/earth-3cc-web-6bfd8b848-n4z7z	0/1	Running	0	116m	
pod/earth-3cc-web-6bfd8b848-rcmxs	0/1	Running	0	116m	
pod/earth-3cc-web-6bfd8b848-xl467	0/1	Running	0	116m	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/earth-2x3-api-svc	ClusterIP	10.3.241.242	<none>	4546/TCP	116m
service/earth-2x3-web-svc	ClusterIP	10.3.250.247	<none>	4545/TCP	116m
service/earth-3cc-web	ClusterIP	10.3.243.24	<none>	6363/TCP	116m
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/earth-2x3-api	3/3	3	3	116m	
deployment.apps/earth-2x3-web	6/6	6	6	116m	
deployment.apps/earth-3cc-runner	3/3	3	3	116m	
deployment.apps/earth-3cc-runner-heavy	3/3	3	3	116m	
deployment.apps/earth-3cc-web	0/4	4	0	116m	
NAME	DESIRED	CURRENT	READY	AGE	
replicaset.apps/earth-2x3-api-584df69757	3	3	3	116m	
replicaset.apps/earth-2x3-web-85c5b7986c	6	6	6	116m	
replicaset.apps/earth-3cc-runner-6cb6cc6974	3	3	3	116m	
replicaset.apps/earth-3cc-runner-heavy-6bf876f46d	3	3	3	116m	
replicaset.apps/earth-3cc-web-6895587dc7	0	0	0	116m	
replicaset.apps/earth-3cc-web-6bfd8b848	4	4	0	116m	
replicaset.apps/earth-3cc-web-d49645966	0	0	0	116m	

First impression could be that all *Pods* are in status RUNNING. But looking closely we see that some of the *Pods* are not ready, which also confirms what we see about one *Deployment* and one *replicaset*. This could be our error to further investigate.

Another approach could be to check the *Services* for missing endpoints:

NAME	ENDPOINTS	AGE
earth-2x3-api-svc	10.0.0.10:80,10.0.1.5:80,10.0.2.4:80	116m
earth-2x3-web-svc	10.0.0.11:80,10.0.0.12:80,10.0.1.6:80 + 3 more...	116m
earth-3cc-web		

Service `earth-3cc-web` doesn't have endpoints. This could be a selector/label misconfiguration or the endpoints are actually not available/ready.

Checking all *Services* for connectivity should show the same (this step is optional and just for demonstration):

→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 earth-2x3-api-svc.earth:4546	...	
	<html><body><h1>It works!</h1></body></html>	
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 earth-2x3-web-svc.earth:4545	% Total % Received % Xferd Average Speed Time Time Time Current	
	Dload Upload Total Spent Left Speed	
100 45 100 45 0 0 5000 0 --:--:-- --:--:-- --:--:-- 5000		
<html><body><h1>It works!</h1></body></html>		
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 earth-3cc-web.earth:6363		
If you don't see a command prompt, try pressing enter.	0 0 0 0 0 0 0 --:--:-- 0:00:05 --:--:-- 0	
curl: (28) Connection timed out after 5000 milliseconds		
pod "tmp" deleted		
pod default/tmp terminated (Error)		

Notice that we use here for example `earth-2x3-api-svc.earth`. We could also spin up a temporary Pod in Namespace `earth` and connect directly to `earth-2x3-api-svc`.

We get no connection to `earth-3cc-web.earth:6363`. Let's look at the Deployment `earth-3cc-web`. Here we see that the requested amount of replicas is not available/ready:

```
→ k -n earth get deploy earth-3cc-web
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
earth-3cc-web   0/4       4           0          7m18s
```

To continue we check the *Deployment* yaml for some misconfiguration:

```
k -n earth edit deploy earth-3cc-web
```

```
# k -n earth edit deploy earth-3cc-web
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
...
  generation: 3                      # there have been rollouts
  name: earth-3cc-web
  namespace: earth
...
spec:
...
  template:
    metadata:
      creationTimestamp: null
      labels:
        id: earth-3cc-web
    spec:
      containers:
        - image: nginx:1.16.1-alpine
          imagePullPolicy: IfNotPresent
          name: nginx
          readinessProbe:
            failureThreshold: 3
            initialDelaySeconds: 10
            periodSeconds: 20
            successThreshold: 1
            tcpSocket:
              port: 82             # this port doesn't seem to be right, should be 80
            timeoutSeconds: 1
...

```

We change the readiness-probe port, save and check the *Pods*:

```
→ k -n earth get pod -l id=earth-3cc-web
NAME      READY   STATUS   RESTARTS   AGE
earth-3cc-web-d49645966-52vb9   0/1     Running   0          6s
earth-3cc-web-d49645966-5tts6   0/1     Running   0          6s
earth-3cc-web-d49645966-db5gp   0/1     Running   0          6s
earth-3cc-web-d49645966-mk7gr   0/1     Running   0          6s
```

Running, but still not in ready state. Wait 10 seconds (initialDelaySeconds of readinessProbe) and check again:

```
→ k -n earth get pod -l id=earth-3cc-web
NAME      READY   STATUS   RESTARTS   AGE
earth-3cc-web-d49645966-52vb9   1/1     Running   0          32s
earth-3cc-web-d49645966-5tts6   1/1     Running   0          32s
earth-3cc-web-d49645966-db5gp   1/1     Running   0          32s
earth-3cc-web-d49645966-mk7gr   1/1     Running   0          32s
```

Let's check the service again:

```
→ k run tmp --restart=Never --rm -i --image=nginx:alpine -- curl -m 5 earth-3cc-
web.earth:6363
  % Total   % Received % Xferd  Average Speed   Time   Time   Time  Current
               Dload  Upload Total   Spent    Left  Speed
100  612  100  612    0     0  55636      0 --:--:-- --:--:-- 55636
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

```
<style>
  body {
    width: 35em;
    margin: 0 auto;
    font-family: Tahoma, Verdana, Arial, sans-serif;
  }
</style>
</head>
<body>
<h1>welcome to nginx!</h1>
...

```

We did it! Finally we write the reason into the requested location:

```
vim /opt/course/p3/ticket-654.txt
```

```
# /opt/course/p3/ticket-654.txt
yo Dirk, wrong port for readinessProbe defined!
```

CKAD Tips Kubernetes 1.21

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

Knowledge

- study all topics as proposed in the curriculum till you feel comfortable with all
- do all of these, maybe 2-3 times (using LATEST kubectl) <https://github.com/dgkanatsios/CKAD-exercises>
- we have a [series with scenarios](#) on Medium, do all of these. Also imagine and create your own ones.
- read this and do all examples: <https://kubernetes.io/docs/concepts/cluster-administration/logging>
- understand Rolling Update Deployment including maxSurge and maxUnavailable
- do 1 or 2 test session with this CKAD Simulator. Understand the solutions and maybe try out other ways to achieve the same
- setup your aliases, be fast and breath `kubectl`

CKAD 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. You can have one browser tab open with one of the allowed links: <https://kubernetes.io/docs> <https://github.com/kubernetes> <https://kubernetes.io/blog>

NOTE: You can have the other tab open as a separate window, this is why a big screen is handy

Deprecated commands

Make sure to not depend on deprecated commands as they might stop working at any time. When you execute a deprecated `kubectl` command a message will be shown, so you know which ones to avoid.

With `kubectl` version 1.18 things have changed. Like its no longer possible to use `kubectl run` to create Jobs, CronJobs or Deployments, only Pods still work. This makes things a bit more verbose when you for example need to create a Deployment with resource limits or multiple replicas.

What if we need to create a Deployment which has, for example, a resources section? We could use both `kubectl run` and `kubectl create`, then do some vim magic. [Read more here.](#)

The Test Environment / Browser Terminal

You'll be provided with a browser terminal which uses Ubuntu 18. The standard shells included with a minimal install of Ubuntu 18 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 autocomplete

Autocompletion is not configured by default. Some wouldn't recommend setting this up because of the lagging internet connection but you can check if it works for you. See further down for how to set up.

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.

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 and namespaces

You'll receive access to 4 different clusters which multiple namespaces and various 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`.

Your Desktop

You are allowed to have multiple monitors connected and have to share every monitor with the proctor. Having one large screen definitely helps as you're only allowed **one** application open (Chrome Browser) with two tabs, one terminal and one k8s docs.

NOTE: You can have the other tab open as a separate window, this is why a big screen is handy

The questions will be on the left (default maybe ~30% space), the terminal on the right. You can adjust the size of the split though to your needs in the real exam.

If you use a laptop you could work with lid closed, external mouse+keyboard+monitor attached. Make sure you also have a webcam+microphone working.

You could also have both monitors, laptop screen and external, active. Though Chrome can only run on one screen. You might be asked that your webcam points straight into your face. So using an external screen and your laptop webcam could not be accepted. Just keep that in mind.

You have to be able to move your webcam around in the beginning to show your whole room and desktop. Have a clean desk with only the necessary on it. You can have a glass/cup with water without anything printed on.

In the end you should feel very comfortable with your setup.

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

We would suggest to minimally setup this alias:

```
alias k=kubectl
```

which means you always run `kubectl` just with `k`.

Vim

Also create the file `~/.vimrc` with the following content:

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

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

Variables

```
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 you like.

Vim

You can also add the following to `~/.vimrc` to be able to apply tab on multiple selected lines.

```
set shiftwidth=2
```

More further down in the `vim` section.

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 again, 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  
...
```

Kubectl autocompletion

Depending on how much is already pre-installed in the CKAD terminal it might be enough to do this:

```
source <(kubectl completion bash)  
complete -F __start_kubectl k # to make it work with the alias k
```

Else you need to run these lines before:

```
sudo apt-get install bash-completion  
source ~/.bashrc
```

Persist bash settings

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

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
```

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© 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.hamvoeke.com/blog/a-quick-and-easy-guide-to-tmux>

