Kubernetes Certified Application Developer (CKAD)

Guide v1.0beta

Author: Sujay Bhowmick
Email: sujaybhowmick at gmail dot com

References:

- from and others dgkanatsios

Kubernetes Objects

Deployment Object

Creating a Deployment

```yaml
# deployment-definition.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

```
$ kubectl create -f deployment-definition.yaml
```

List deployments

```
$ kubectl get deployments # default namespace
$ kubectl get deployments --namespace=<namespace-name>
$ kubectl get deployment <deployment-name> # default namespace
$ kubectl get deployment <deployment-name> --namespace=<namespace-name>
```

Deployment details

```
$ kubectl describe deployment <deployment-name> # default namespace
$ kubectl get deployment <deployment-name> --namespace=<namespace-name>
```

Delete deployment

```
$ kubectl delete deployment <deployment-name>
$ kubectl delete deployment <deployment-name> --namespace=<namespace-name>
```

Update a deployment

```
$ kubectl apply -f deployment-definition.yaml
```

Pod Object

Creating a Pod

> In general, users shouldn't need to create pods directly. They should almost always use controllers even for singletons, for example, Deployments. Controllers provide self-healing with a cluster scope, as well as replication and rollout management. Controllers use a Pod Template that you provide to create the Pods for which it is responsible.

```
# pod-definition.yaml - POD Template
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  namespace: default
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

```
$ kubectl create -f pod-definition.yaml
```

List Pods

```
$ kubectl get pods # default namespace
$ kubectl get pods --namespace=<namespace-name>
```

Pod details

```
$ kubectl describe pod <pod-name> # default namespace
$ kubectl describe pod <pod-name> --namespace=<namespace-name>
```

Delete Pod

```
$ kubectl delete pod <pod-name> # default namespace
$ kubectl delete pod <pod-name> --namespace=<namespace-name>
```

ReplicaSets Object

Create a ReplicaSet

> It is recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.
>
> we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

```
# replicaset-definition.yaml - ReplicaSet Template
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  namespace: default
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

```
$ kubectl create -f replicaset-definition.yaml
```

List ReplicaSets

```
$ kubectl get replicaset # default namespace
$ kubectl get replicaset --namespace=<namespace-name>
```

ReplicaSet details

```
$ kubectl describe replicaset <replicaset-name> # default namespace
$ kubectl describe replicaset <replicaset-name> --namespace=<namespace-name>
```

Delete ReplicaSet

```
$ kubectl delete replicaset <replicaset-name> # default namespace
$ kubectl delete replicaset <replicaset-name> --namespace=<namespace-name>
```

Namespace Object

Create a namespace

```
# namespace-definition.yaml - Namespace template
apiVersion: v1
kind: Namespace
metadata:
  name: development-namespace
```

```
$ kubectl create -f namespace-definition.yaml
```

List Namespace

```
$ kubectl get namespaces
```

Namespace details

```
$ kubectl describe namespace <namespace-name>
```

Delete Namespace

> **Warning:** This deletes *everything* under the namespace!

```
$ kubectl delete namespace <namespace-name>
```

ConfigMap Object

Create a ConfigMap

```
# creating configmap from properties files in a directory (key=value)
$ kubectl create configmap <configmap-name> --from-file=configMap/
# creating configmap from env files in a directory (VAR=VALUE)
$ kubectl create configmap <configmap-name> --from-env-file=configMap/
# creating configmap from literal value
$ kubectl create configmap <configmap-name> --from-literal=VAR=VALUE
```

List ConfigMaps

```
$ kubectl get configmaps # default namespace
$ kubectl get configmaps --namespace=<namespace-name>
```

ConfigMap details

```
$ kubectl describe configmap <config-map-name> # default namespace
$ kubectl describe configmap <config-map-name> --namespace=<namespace-name>
```

Delete ConfigMap

```
$ kubectl delete configmap <config-map-name> # default namespace
$ kubectl delete configmap <config-map-name> --namespace=<namespace-name>
```

Using ConfigMaps

```
# example usage of configmaps in Deployments
```

Secrets Object

Create a Secret

```
# secrets-definition.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

```
$ kubectl create -f secrets-definition.yaml
# creating secret from files
$ kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./password.txt
# creating secret using literal
$ kubectl create secret generic prod-db-secret --from-literal=username=produser --from-literal=password=Y4nys7f11
```

List Secrets

```
$ kubectl get secrets # default namespace
$ kubectl get secrets --namespace=<namespace-name>
```

Secret details

```
$ kubectl describe secret <secret-name> # default namespace
$ kubectl describe secret <secret-name> --namespace=<namespace-name>
```

Delete a Secret

```
$ kubectl delete secret <secret-name> # default namespace
$ kubectl delete secret <secret-name> --namespace=namespace
```

ServiceAccount Object

Create a ServiceAccount

```
# service-account-definition.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: secret-access-sa
```

```
$ kubectl create -f service-account-definition.yaml
```

List ServiceAccounts

```
$ kubectl get serviceaccounts # default namespace
$ kubectl get serviceaccounts --namespace=<namespace-name>
```

ServiceAccount details

```
$ kubectl describe serviceaccount <service-account-name> # default namespace
$ kubectl describe serviceaccount <service-account-name> --namespace=<namespace-name>
```

Delete a ServiceAccount

```
$ kubectl delete serviceaccount <service-account-name> # default namespace
$ kubectl delete serviceaccount <service-account-name> --namespace=<namespace-name>
```

Service Object

Create a Service

```yaml
# service-definition.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: default
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

```
$ kubectl create -f service-definition.yaml
```

List Services

```
$ kubectl get services # default namespace
$ kubectl get services --namespace=<namespace-name>
```

Service details

```
$ kubectl describe service <service-name> # default namespace
$ kubectl describe service <service-name> --namespace=<namespace-name>
```

Delete a Service

```
$ kubectl delete service <service-name> # default namespace
$ kubectl delete service <service-name> --namespace=<namespace-name>
```

ResourceQuota Object

Create a Quota

```yaml
# resource-quota-definition.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: myresourcequota
  namespace: default
spec:
  hard:
    cpu: "1"
    memory: 1G
    pods: "2"
```

```
$ kubectl create -f resource-quota-definition.yaml
# or
$ kubectl create quota myrq --hard=cpu=1,memory=1G,pods=2 --dry-run -o yaml
```

List ResourceQuota

```
$ kubectl get quota <resource-quota-name> # default namespace
$ kubectl get quota <resource-quota-name> --namespace=<namspace-name>
```

ResourceQuota details

```
$ kubectl describe quota <resource-quota-name> # default namespace
$ kubectl describe quota <resource-quota-name> --namespace=<namspace-name>
```

Delete a ResourceQuota

```
$ kubectl delete quota <resource-quota-name> # default namespace
$ kubectl delete quota <resource-quota-name> --namespace=<namspace-name>
```

Configuring Volumes in Kubernetes

Configure a Volume

```yaml
# pod-volume-aws-definition.yaml for AWS block store
apiVersion: v1
kind: Pod
metadata:
  name: rand-num-gen
  namespace: default
spec:
  containers:
    - name: alpine
      image: alpine
      command: ["/bin/sh" "-c"]
      args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
      volumeMounts:
        - mountPath: /opt
          name: data-volume

  volumes:
    - name: data-volume
      awsElasticBlockStore:
        volumeID: <volume-id>
        fsType: ext4
```

> Do not use the node directory in a cluster setup, only for single node.

```yaml
# pod-volume-definition.yaml for node directory
apiVersion: v1
kind: Pod
metadata:
  name: rand-num-gen
  namespace: default
spec:
  containers:
    - name: alpine
      image: alpine
      command: ["/bin/sh", "-c"]
      args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
      volumeMounts:
        - mountPath: /opt
          name: data-volume
  volumes:
    - name: data-volume
      hostPath:
        path: /data
        type: Directory
```

Exercise

Core Concepts (13%)

1. Create a namespace called *'mynamespace'* and a pod with image *nginx* called *nginx* on this namespace

```
# Answer - Namespace
apiVersion: v1
kind: Namespace
metadata:
  name: mynamespace
```

```
# Answer - Pod
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: mynamespace
spec:
  containers:
    - name: ngnix
      image: nginx
```

2. Create a busybox pod (using kubectl command) that runs the command "env". Run it and see the output

```
# $ kubectl run busybox --image=busybox --restart=Never --dry-run -o yaml --
command  -- env > envpod.yaml
# $ kubectl logs busybox
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
    - name: busybox
      image: busybox
      command:
        - "env"
```

3. Get the YAML for a new namespace called 'myns' without creating it

```
# Get the YAML for a new namespace called 'myns' without creating it
# $ kubectl create namespace myns --dry-run -o yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: myns
spec: {}
status: {}
```

4. Get pods on all namespaces

```
$ kubectl get pods --all-namespaces
```

5. Create a pod with image nginx called nginx and allow traffic on port 80

```
# Create a pod with image nginx called nginx and allow traffic on port 80
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

6. Get the pod's ip, use a temp busybox image to wget its '/'

```
$ kubectl get po -o wide # get the IP, will be something like '10.1.1.131'
# create a temp busybox pod
$ kubectl run busybox --image=busybox --rm -it --restart=Never -- sh
# run wget on specified IP:Port
wget -O- 10.1.1.131:80
exit

# Or alternatively
# Get IP of the nginx pod
NGINX_IP=$(kubectl get pod nginx -o jsonpath='{.status.podIP}')
# create a temp busybox pod
```

```
$ kubectl run busybox --image=busybox --env="NGINX_IP=$NGINX_IP" --rm -it --
restart=Never -- sh
# run wget on specified IP:Port
wget -O- $NGINX_IP:80
exit
```

7. Get this pod's YAML without cluster specific information

```
$ kubectl get pod nginx -o yaml --export
```

8. Get information about the pod, including details about potential issues (e.g. pod hasn't started)

```
$ kubectl describe pod <pod-name>
```

9. Get pod logs

```
$ kubectl logs <pod-name>
```

10. If pod crashed and restarted, get logs about the previous instance

```
$ kubectl logs <pod-name> -p
```

11. Connect to the nginx pod

```
$ kubectl exec -it <pod-name> -- /bin/sh
```

12. Create a busybox pod that echoes 'hello world' and then exits

```
# Create a busybox pod that echoes 'hello world' and then exits
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
    - name: busybox
      image: busybox
      command:
        - "echo"
      args:
        - "hello world"
```

14. Do the same, but have the pod deleted automatically when it's completed

    ```
    $ kubectl run busybox --image=busybox -it --rm --restart=Never -- /bin/sh -c 'echo hello world'
    $ kubectl get po
    ```

15. Create an nginx pod and set an env value as 'var1=val1'. Check the env value existence within the pod

    ```
    $ kubectl run nginx --image=nginx --env=var1=val1
    $ kubectl exec -it nginx -- env
    ```

16. Get the YAML for a new ResourceQuota called 'myrq' without creating it

    ```
    $ kubectl create quota myrq --hard=cpu=1,memory=1G,pods=2 --dry-run -o yaml
    ```

Pod Design (20%)

Labels and Annotations

1. Create 3 pods with names nginx1,nginx2,nginx3. All of them should have the label app=v1

   ```
   $ kubectl run nginx1 --image=nginx --labels=app=v1
   $ kubectl run nginx2 --image=nginx --labels=app=v1
   $ kubectl run nginx3 --image=nginx --labels=app=v1
   ```

2. Show all labels of the pods

   ```
   $ kubectl get pods --show-labels
   ```

3. Change the labels of pod 'nginx2' to be app=v2

   ```
   $ kubectl label pod nginx2 app=v2 --overwrite
   ```

4. Get the label 'app' for the pods

   ```
   $ kubectl get pod -L app
   ```

5. Get only the 'app=v2' pods

```
$ kubectl get pod -l app=v2
# or
$ kubectl get pod -l 'app in (v2)'
```

6. Remove the 'app' label from the pods we created before

```
$ kubectl label pod -lapp app-
# or
$ kubectl label pod nginx{1..3} app-
```

7. Create a pod that will be deployed to a Node that has the label 'accelerator=nvidia-tesla-p100'

```
# Create a pod that will be deployed to a Node that has the label
'accelerator=nvidia-tesla-p100'
apiVersion: v1
kind: Pod
metadata:
  name: fastpod
  namespace: default
spec:
  containers:
    - name: fastpod
      image: nginx
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

8. Annotate pods nginx1, nginx2, ningx3 with "description='my description'" value

```
$ kubectl annotate nginx1, nginx2, nginx3 description='my description'
```

9. Check the annotations for pod nginx1

```
$ kubectl describe pod nginx1 | grep -i annotations
```

10. Remove the annotations for these three pods

```
$ kubectl annotate pod nginx{1..3} description-
```

Deployments

1. Create a deployment with image nginx:1.7.8, called nginx, having 2 replicas, defining port 80 as the port that this container exposes (don't create a service for this deployment)

```
# Create a deployment with image nginx:1.7.8, called nginx, having 2 replicas,
# defining port 80 as the port that this container exposes
# (don't create a service for this deployment)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: default
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      namespace: default
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.8
          ports:
            - containerPort: 80
  replicas: 2
```

2. View the YAML of this deployment

```
$ kubectl get deployment nginx --export -o yaml
```

3. View the YAML of the ReplicaSet created

```
$ kubectl get replicaset -l app=nginx
# or
$kubectl get rs -l app=nginx
```

4. Get the YAML for one of the pods

```
$ kubectl get po # list of all pods
$ kubectl get po nginx-5746858fb4-l6qfb -o yaml --export
```

5. Check how the deployment rollout is going

```
$ kubectl rollout status deployment nginx
```

6. Update the nginx image to nginx:1.7.9

```
$ kubectl edit deployment nginx
# or
$ kubectl set image deploy nginx nginx=nginx:1.7.9
```

7. Undo the latest rollout and verify that new pods have the old image (nginx:1.7.8)

```
$ kubectl rollout undo deploy nginx
```

8. Do an on purpose update of the deployment with a wrong image nginx:1.91

```
$ kubectl set image deploy nginx nginx=nginx:1.19
```

9. Verify that something's wrong with the rollout

```
$ kubectl rollout status deploy nginx
```

10. Return the deployment to the second revision (number 2) and verify the image is nginx:1.7.9

```
$ kubectl rollout undo deploy nginx --to-revision=4
```

11. Check the details of the third revision rollout (number 4)

```
$ kubectl rollout history deploy nginx --revision=4
```

12. Scale the deployment to 5 replicas

```
$ kubectl scale --current-replicas=2 --replicas=5 deploy nginx
```

13. Autoscale the deployment, pods between 5 and 10, targetting CPU utilization at 80%

```
$ kubectl autoscale deploy nginx --min=5 --max=10 --cpu-percent=80
```

14. Pause the rollout of the deployment

```
$ kubectl rollout pause deploy nginx
```

15. Update the image to nginx:1.9.1 and check that there's nothing going on, since we paused the rollout

```
$ kubectl set image deploy nginx nginx=nginx:1.9.1
$ kubectl rollout history deploy nginx
```

16. Resume the rollout and check that the nginx:1.9.1 image has been applied

```
$ kubectl rollout resume deploy nginx
$ kubectl rollout history deploy nginx
```

17. Delete the deployment and the horizontal pod autoscaler you created

```
$ kubectl delete deploy nginx
$ kubectl delete hpa nginx
```

Jobs

1. Create a job with image perl that runs default command with arguments "perl -Mbignum=bpi -wle 'print bpi(2000)'"

```
apiVersion: batch/v1
kind: Job
metadata:
  name: perl
  namespace: default
spec:
  template:
    spec:
      containers:
        - name: perl
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

2. Wait till it's done, get the output

```
$ kubectl logs perl-h9sbz
```

3. Create a job with the image busybox that executes the command 'echo hello;sleep 30;echo world'

```
apiVersion: batch/v1
kind: Job
```

```
metadata:
  name: busybox-job
  namespace: default
spec:
  template:
    spec:
      containers:
        - name: busybox-job
          image: busybox
          command: ["/bin/sh"]
          args: ["-c", "echo hello; sleep 30; echo world"]
      restartPolicy: OnFailure
```

4. See the status of the job, describe it and see the logs

```
$ kubectl get job busybox-job
$ kubectl describe job busybox-job
$ kubectl logs job/busybox-job
```

5. Create the same job, make it run 5 times, one after the other. Verify its status and delete it

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox-job
  namespace: default
spec:
  completions: 5
  template:
    spec:
      containers:
        - name: busybox-job
          image: busybox
          command: ["/bin/sh"]
          args: ["-c", "echo hello; sleep 30; echo world"]
      restartPolicy: OnFailure
```

6. Create the same job, but make it run 5 parallel times

```
apiVersion: batch/v1
kind: Job
metadata:
  name: busybox-job
  namespace: default
spec:
  completions: 5
  parallelism: 5
  template:
    spec:
      containers:
        - name: busybox-job
          image: busybox
          command: ["/bin/sh"]
          args: ["-c", "echo hello; sleep 30; echo world"]
      restartPolicy: OnFailure
```

CronJobs

1. Create a cron job with image busybox that runs on a schedule of "*/1 * * * *" and writes 'date; echo Hello from the Kubernetes cluster' to standard output

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: busybox-cronjob
  namespace: default
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: busybox-cronjob
              image: busybox
              command: ["/bin/sh"]
              args: ["-c", "date; echo Hello from Kubernetes cluster"]
          restartPolicy: OnFailure
```

2. See its logs and delete it

```
$ kubectl get cj
$ kubectl get jobs --watch
$ kubectl get po --show-labels
$ kubectl logs busybox-cronjob-1556066760-smzn7
$ kubectl delete cj busybox-cronjob
```

Configuration (18%)

ConfigMaps

1. Create a configmap named config with values foo=lala,foo2=lolo

```
$ kubectl create configmap config --from-env-file=app.properties
# or
$ kubectl create configmap config --from-literal=foo=lala --from-literal=foo2=lolo
```

2. Display its values

```
$ kubectl get cm config -o yaml --export
# or
$ kubectl describe cm config
```

3. Create and display a configmap from a file, giving the key 'special'

```
$ kubectl create configmap config --from-file=special=app.properties
```

4. Create a configMap called 'options' with the value var5=val5. Create a new nginx pod that loads the value from variable 'var5' in an env variable called 'option'

```
$ kubectl create configmap options --from-literal=var5=val5
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      env:
        - name: option
          valueFrom:
            configMapKeyRef:
              name: options
              key: var5
  restartPolicy: Never
```

```
# to check env variables
$ kubectl exec nginx -it -- env
```

5. Create a configMap 'anotherone' with values 'var6=val6', 'var7=val7'. Load this configMap as env variables into a new nginx pod

```
$ kubectl create configmap anotherone --from-literal=var6=val6 --from-literal=var7=val7
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      envFrom:
        - configMapRef:
            name: anotherone
  restartPolicy: Never
```

```
# to check env variables
$ kubectl exec nginx -it -- env
```

6. Create a configMap 'cmvolume' with values 'var8=val8', 'var9=val9'. Load this as a volume inside an nginx pod on path '/etc/lala'. Create the pod and 'ls' into the '/etc/lala' directory.

```
$ kubectl create cm cmvolume --from-literal=var8=val8 --from-literal=var9=val9
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  volumes:
    - name: config-volume
      configMap:
        name: cmvolume
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - mountPath: /etc/lala
          name: config-volume
  restartPolicy: Never
```

```
# to check
$ kubectl exec nginx -it -- /bin/sh
# cd to /etc/lala and you should see two file var8 and var 9
$ cat var8 # will show value val8
```

Security Context

1. Create the YAML for an nginx pod that runs with the UID 101. No need to create the pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  securityContext:
    runAsUser: 101
  containers:
    - name: nginx
      image: nginx
  restartPolicy: Never
```

2. Create the YAML for an nginx pod that has the capabilities "NET_ADMIN", "SYS_TIME" added on its single container

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      securityContext:
        capabilities:
          add: ["NET_ADMIN", "SYS_TIME"]
  restartPolicy: Never
```

Requests and limits

1. Create an nginx pod with requests cpu=100m,memory=256Mi and limits cpu=200m,memory=512Mi

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        requests:
          cpu: "100m"
          memory: "256Mi"
        limits:
          cpu: "200m"
          memory: "512Mi"
  restartPolicy: Never
```

Secrets

1. Create a secret called mysecret with the values password=mypass

```
$ kubectl create secret generic mysecret --from-literal=password=mypass
```

2. Create a secret called mysecret2 that gets key/value from a file

```
$ kubectl create secret generic mysecret2 --from-file=username
```

3. Get the value of mysecret2

```
$ kubectl get secret mysecret2 -o yaml --export
$ echo YWRtaW4K | base64 -d
```

4. Create an nginx pod that mounts the secret mysecret2 in a volume on path /etc/foo

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: nginx
spec:
  volumes:
    - name: myvolume
      secret:
        secretName: mysecret2
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - mountPath: /etc/foo
          name: myvolume
  restartPolicy: Never
```

5. Delete the pod you just created and mount the variable 'username' from secret mysecret2 onto a new nginx pod in env variable called 'USERNAME'

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      env:
        - name: USERNAME
          valueFrom:
            secretKeyRef:
              key: username
              name: mysecret2
  restartPolicy: Never
```

Service Accounts

1. See all the service accounts of the cluster in all namespaces

```
$ kubectl get serviceaccounts --all-namespaces
# or
$ kubectl get sa --all-namespaces
```

2. Create a new serviceaccount called 'myuser'

```
$ kubectl create serviceaccount myuser
# or
$ kubectl create sa myuser
```

3. Create an nginx pod that uses 'myuser' as a service account

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  serviceAccountName: myuser
  containers:
    - name: nginx
      image: nginx
  restartPolicy: Never
```

Observability (18%)

Liveness and Readiness probes

1. Create an nginx pod with a liveness probe that just runs the command 'ls'. Save its YAML in pod.yaml. Run it, check its probe status, delete it.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      livenessProbe:
        exec:
          command: ["ls"]
  restartPolicy: Never
```

2. Modify the pod.yaml file so that liveness probe starts kicking in after 5 seconds whereas the period of probing would be 10 seconds. Run it, check the probe, delete it.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      livenessProbe:
        initialDelaySeconds: 5
        periodSeconds: 10
        exec:
          command:
            - ls
  restartPolicy: Never
```

3. Create an nginx pod (that includes port 80) with an HTTP readinessProbe on path '/' on port 80. Again, run it, check the readinessProbe, delete it.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      readinessProbe:
        httpGet:
          port: 80
          path: /
        initialDelaySeconds: 10
        timeoutSeconds: 3
  restartPolicy: Never
```

Logging

1. Create a busybox pod that runs 'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done'. Check its logs

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
    - name: busybox
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "i=0; while true; do echo \"$i: $(date)\"; i=$((i+1));
sleep 1; done"]
  restartPolicy: Never
```

2. Get CPU/memory utilization for nodes (heapster must be running)

```
$ kubectl top nodes
```

Services and Networking (13%)

1. Create a pod with image nginx called nginx and expose its port 80

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    ports:
      - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
```

2. Confirm that ClusterIP has been created. Also check endpoints

```
$ kubectl get svc nginx-svc
$ kubectl get ep
```

3. Get pod's ClusterIP, create a temp busybox pod and 'hit' that IP with wget

```
$ kubectl get svc nginx-svc
$ kubectl run busybox --rm --image=busybox -it -- sh
```

4. Convert the ClusterIP to NodePort and find the NodePort port. Hit it using Node's IP. Delete the service and the pod

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  selector:
    app: myapp
  ports:
  - port: 80
    targetPort: 80
  type: NodePort
```

5. Create a deployment called foo using image 'dgkanatsios/simpleapp' (a simple server that returns hostname) and 3 replicas. Label it as 'app=foo'. Declare that containers in this pod will accept traffic on port 8080 (do NOT create a service yet)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: foo
spec:
  replicas: 3
  selector:
    matchLabels:
      app: foo
  template:
    metadata:
      labels:
        app: foo
    spec:
      containers:
      - name: foo
        image: dgkanatsios/simpleapp
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 8080
```

```
$ kubectl get po -o wide # IP Column
```

6. Create a service that exposes the deployment on port 6262. Verify its existence, check the endpoints

```
apiVersion: v1
kind: Service
metadata:
  name: foo-svc
spec:
  selector:
    app: foo
  ports:
  - port: 6262
    targetPort: 8080
```

```
$ kubectl get ep
```

7. Create a temp busybox pod and connect via wget to foo service. Verify that each time there's a different hostname returned. Delete deployment and services to cleanup the cluster

```
$ kubectl run busybox --rm --image=busybox -it sh
wget -O- http://10.106.123.25:6262
```

8. Create an nginx deployment of 2 replicas, expose it via a ClusterIP service on port 80. Create a NetworkPolicy so that only pods with labels 'access: true' can access the deployment and apply it

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            memory: "128Mi"
            cpu: "500m"
        ports:
        - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
    - from:
      - podSelector:
          matchLabels:
            access: 'true'
```

```
 # Cannot access the pod as label access=true is missing
$ kubectl run busybox --image=busybox --restart=Never --rm -it -- wget -O-
http://IP:80
 # Can access the pod as label access=true is present
$ kubectl run busybox --image=busybox --restart=Never --labels=access=true --
rm -it -- wget -O- http://IP:80
```

State Persistence (8%)

1. Create busybox pod with two containers, each one will have the image busybox and will run the 'sleep 3600' command. Make both containers mount an emptyDir at '/etc/foo'. Connect to the second busybox, write the first column of '/etc/passwd' file to '/etc/foo/passwd'. Connect to the first busybox and write '/etc/foo/passwd' file to standard output. Delete pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    name: busybox
spec:
  volumes:
    - name: myvolume
      emptyDir: {}
  containers:
  - name: busybox
    image: busybox
    resources:
      limits:
        memory: "128Mi"
```

```yaml
        cpu: "500m"
    ports:
      - containerPort: 80
    command: ["/bin/sh"]
    args: ["-c", "sleep 3600;"]
    volumeMounts:
      - name: myvolume
        mountPath: /etc/foo
  - name: busybox2
    image: busybox
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    ports:
      - containerPort: 80
    command: ["/bin/sh"]
    args: ["-c", "sleep 3600;"]
    volumeMounts:
      - name: myvolume
        mountPath: /etc/foo
```

```bash
# connect to container busybox2
$ kubectl exec busybox -c busybox2 -it -- /bin/sh
cat /etc/passwd | cut -f 1 -d ':' > /etc/foo/passwd
# connect to container busybox
$ kubectl exec busybox -c busybox -it -- /bin/sh
cat /etc/foo/passwd
```

2. Create a PersistentVolume of 10Gi, called 'myvolume'. Make it have accessMode of 'ReadWriteOnce' and 'ReadWriteMany', storageClassName 'normal', mounted on hostPath '/etc/foo'. Save it on pv.yaml, add it to the cluster. Show the PersistentVolumes that exist on the cluster.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myvolume
spec:
  storageClassName: normal
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
  hostPath:
    path: /etc/foo
```

```
$ kubectl get pv
```

3. Create a PersistentVolumeClaim for this storage class, called mypvc, a request of 4Gi and an accessMode of ReadWriteOnce and save it on pvc.yaml. Create it on the cluster. Show the PersistentVolumeClaims of the cluster. Show the PersistentVolumes of the cluster.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  resources:
    requests:
      storage: 4Gi
  storageClassName: normal
  accessModes:
    - ReadWriteOnce
```

```
$ kubectl get pvc
```

4. Create a busybox pod with command 'sleep 3600', save it on pod.yaml. Mount the PersistentVolumeClaim to '/etc/foo'. Connect to the 'busybox' pod, and copy the '/etc/passwd' file to '/etc/foo'.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox2
  labels:
    name: busybox2
spec:
  volumes:
    - name: myvolume
      persistentVolumeClaim:
        claimName: mypvc
  containers:
  - name: busybox2
    image: busybox
    args:
      - /bin/sh
      - -c
      - sleep 3600
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    ports:
      - containerPort: 80
    volumeMounts:
      - name: myvolume
        mountPath: /etc/foo
```

```
$ kubectl exec busybox -it -- /bin/sh
cp /etc/passwd /etc/foo/passwd
$ kubectl exec busybox2 -it -- /bin/sh
ls /etc/foo
```

5. Create a busybox pod with 'sleep 3600' as arguments. Copy '/etc/passwd' from the pod to your local folder.

```
$ kubectl run busybox --image=busybox -- sleep 3600
$ kubectl cp busybox:/etc/passwd .
cat passwd
```

Multi-container Pods (10%)

1. Create a Pod with two containers, both with image busybox and command "echo hello; sleep 3600". Connect to the second container and run 'ls'.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  labels:
    name: busybox
spec:
  volumes:
    - name: myvolume
      emptyDir: {}
  containers:
  - name: busybox
    image: busybox
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    ports:
      - containerPort: 80
    command: ["/bin/sh"]
    args: ["-c", "echo hello; sleep 3600"]
  - name: busybox2
    image: busybox
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
    ports:
      - containerPort: 80
    command: ["/bin/sh"]
    args: ["-c", "echo hello; sleep 3600"]
```

```
$ kubectl exec busybox -c busybox2 -it -- /bin/sh
ls
exit
```