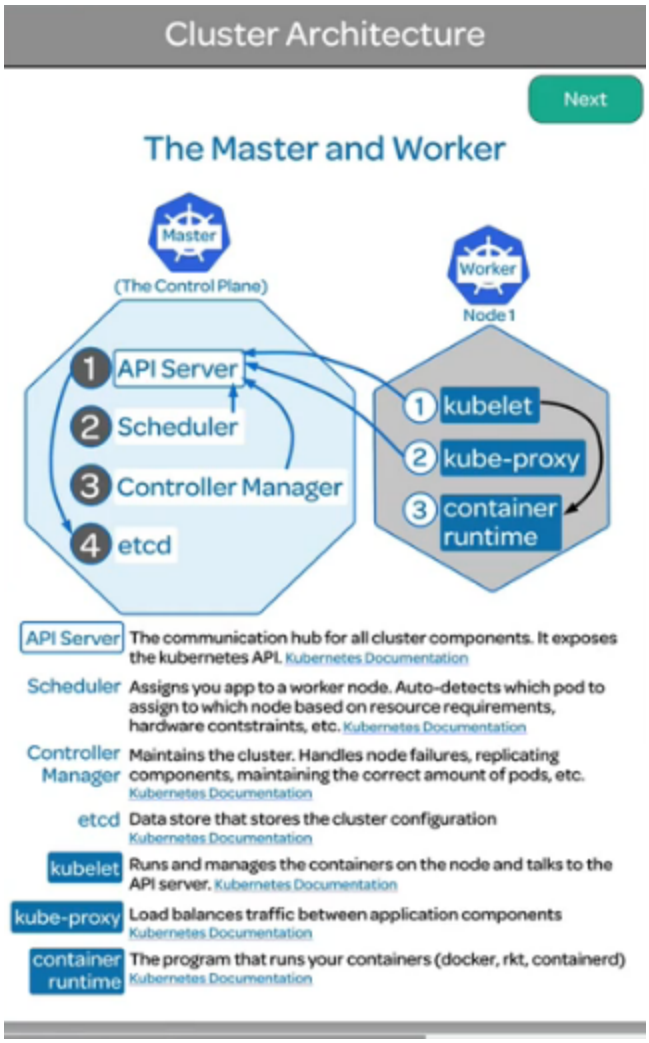
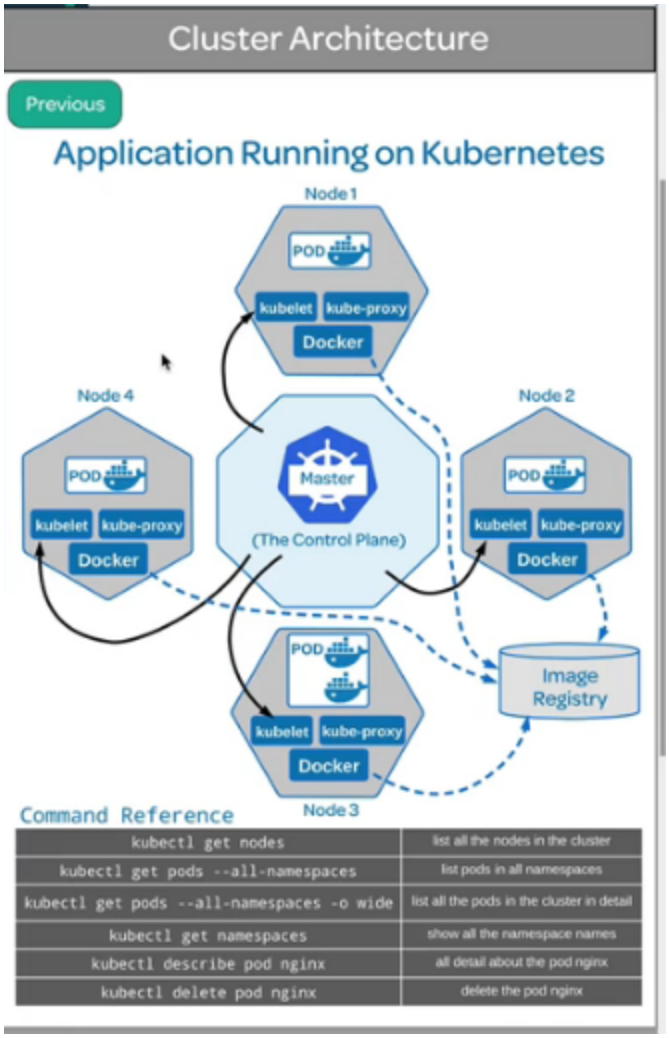
- Core Concepts (19%)
  - Understanding Kubernetes Architecture
    - Kubernetes Cluster Architecture
  - Official Kubernetes Documentation
    - Kubernetes API Primitives
  - Helpful Links
    - Kubernetes Services and Network Primitives
  - Helpful Links:
- Installation, Configuration, and Validation (12%)
  - Building the Kubernetes Cluster
    - Release Binaries, Provisioning, and Types of Clusters
  - Helpful Links
    - Installing Kubernetes Master and Nodes
    - Building a Highly Available Kubernetes Cluster
  - Helpful Links:
    - Configuring Secure Cluster Communications
  - Helpful Links
    - Running End-to-End Tests on Your Cluster
  - Helpful Links:
- Upgrade Kubernetes (Managing the Kubernetes Cluster)
  - Upgrade steps:
  - Node Maintenance
  - Resource Links
  - Backup & Restore
  - LAB
- Networking
  - Pod and Node networking
  - Container network interface (CNI)
  - Service networking
  - Ingress Rules and Load Balancers
  - Cluster DNS
- Scheduling
  - Configuring the Kubernetes Scheduler
  - Running Multiple Schedulers for Multiple Pods
  - Scheduling Pods with Resource Limits and Label Selectors
  - DaemonSets and Manually Scheduled Pods

Core Concepts (19%)

Understanding Kubernetes Architecture

Kubernetes Cluster Architecture





The Kubernetes architecture has two main roles: the master role and the worker role. In this lesson, we will take a look at what each role is and the individual components that make it run. We will even deploy a pod to see the components in action!

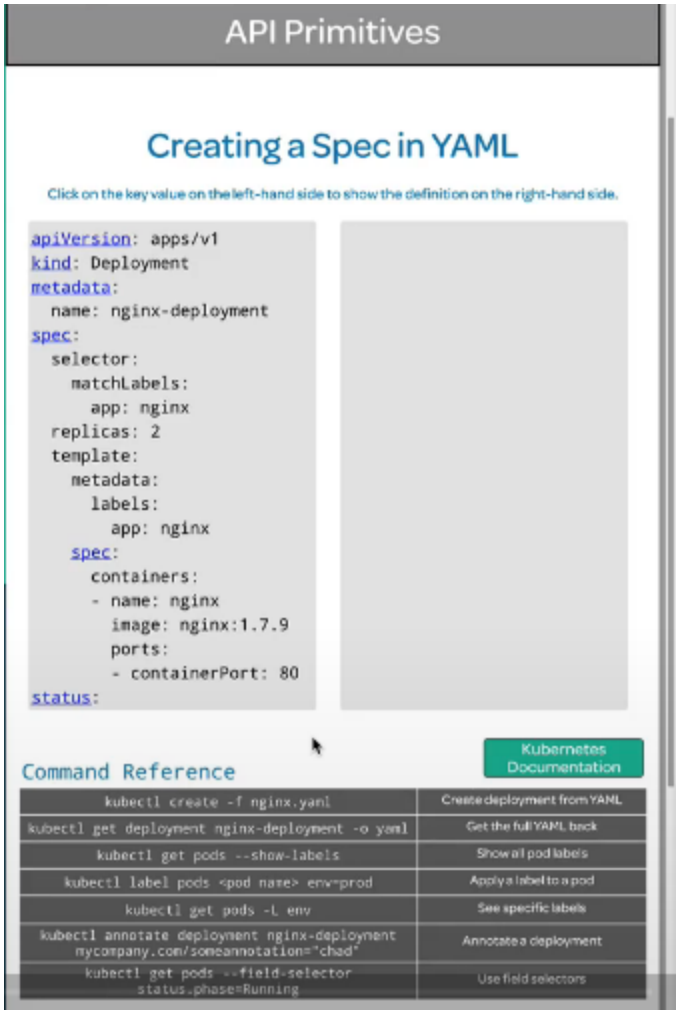
```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
```

```
EOF
```

Official Kubernetes Documentation

- Kubernetes Components Overview
- API Server
- Scheduler
- Controller Manager
- ectd Datastore
- kubelet
- kube-proxy
- Container Runtime

Kubernetes API Primitives



Every component in the Kubernetes system makes a request to the API server. The kubectl command line utility processes those API calls for us and allows us to format our request in a certain way. In this lesson, we will learn how Kubernetes accepts the instructions to run deployments and go through the YAML script that is used to tell the control plane what our environment should look like.

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  selector:

    matchLabels:

      app: nginx
```

```
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Helpful Links

- Spec and Status
- API Versioning
- Field Selectors

Kubernetes Objects are persistent entities in the Kubernetes system

A Kubernetes object is a “record of intent”–once you create the object, the Kubernetes system will constantly work to ensure that object exists. By creating an object, you’re effectively telling the Kubernetes system what you want your cluster’s workload to look like; this is your cluster’s *desired state*

apiVersion:

kind:

Metadata:

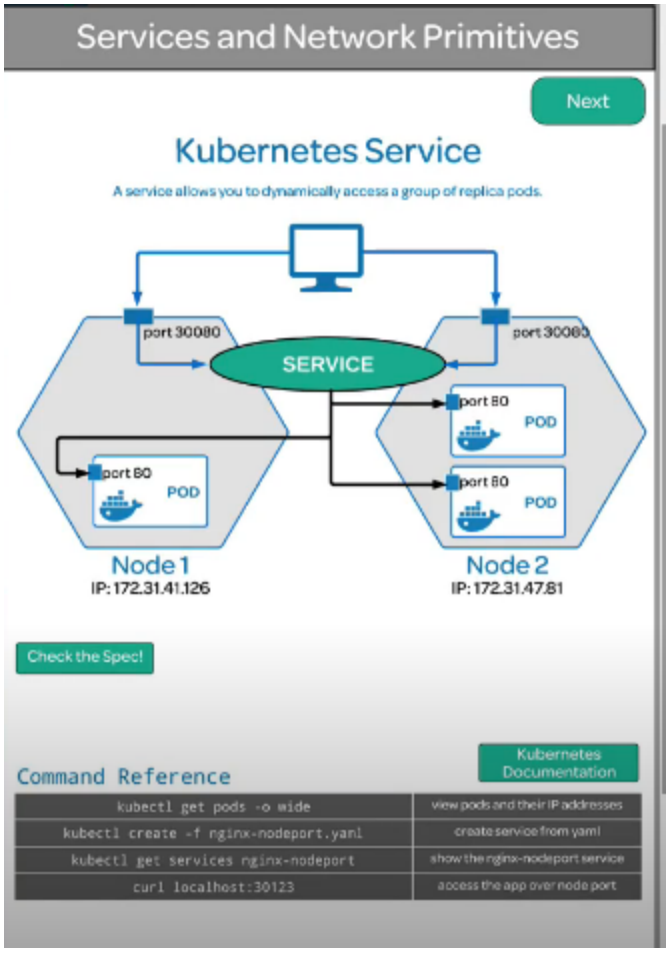
Spec: Desired state

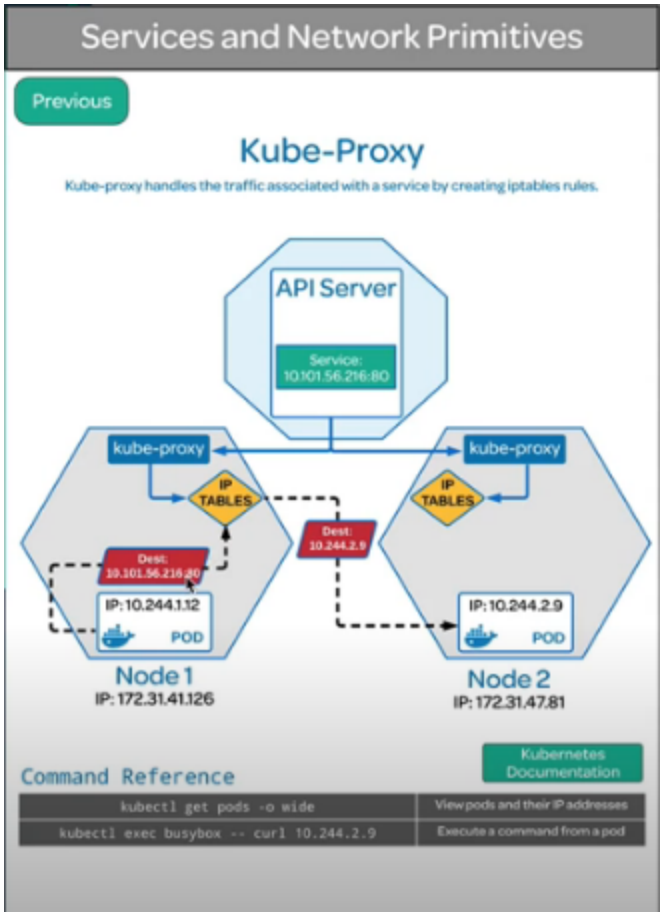
Status: Actual state

Kubectl – iputs in yaml file

Internally calls API passing json as input (Yaml to json)

Kubernetes Services and Network Primitives





Kubernetes services allow you to dynamically access a group of replica pods without having to keep track of which pods are moved, changed, or deleted. In this lesson, we will go through how to create a service and communicate from one pod to another.
The YAML to create the service and associate it with the label selector:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30080
```

```
  selector:
    app: nginx
```

To create the busybox pod to run commands from:

```
cat << EOF | kubectl create -f -
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - name: busybox
    image: radial/busyboxplus:curl
    args:
    - sleep
    - "1000"
EOF
```

Helpful Links:

- Kubernetes Services
- The Role of kube-proxy

Installation, Configuration, and Validation (12%)

Building the Kubernetes Cluster

Release Binaries, Provisioning, and Types of Clusters

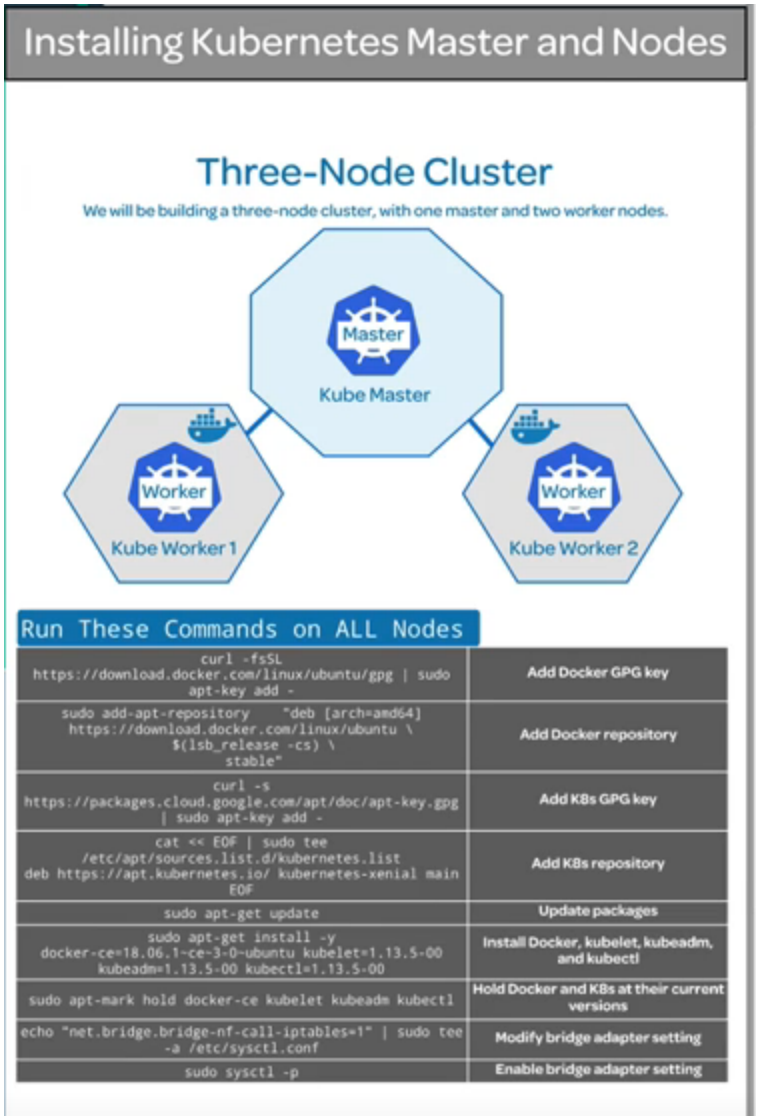
There are many choices when it comes to choosing where to create your Kubernetes cluster. In this lesson, we will focus on what you need to know for the CKA exam — specifically, how the cluster build relates to the types of clusters you will face in the exam.

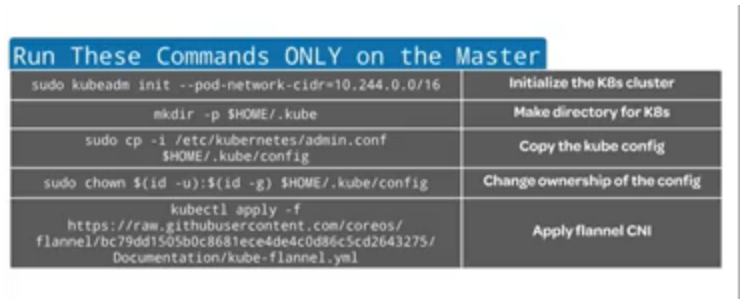
Helpful Links

- Kubernetes Binaries

- Install Minikube
- Kubernetes Documentation

Installing Kubernetes Master and Nodes





Now that we've considered all the different types of clusters and where to locate the tools we need to install a cluster, let's get to it! In this lesson, we will go through the all steps to install a three-node cluster using your Linux Academy cloud servers.
Get the Docker gpg key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Add the Docker repository:

```
sudo add-apt-repository    "deb [arch=amd64] https://download.docker.com/linux/ubuntu
\
   $(lsb_release -cs) \
   stable"
```

Get the Kubernetes gpg key:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
```

Add the Kubernetes repository:

```
cat << EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

Update your packages:

```
sudo apt-get update
```

Install Docker, kubelet, kubeadm, and kubectl:

```
sudo apt-get install -y docker-ce=18.06.1~ce~3-0~ubuntu kubelet=1.13.5-00
kubeadm=1.13.5-00 kubectl=1.13.5-00
```

Hold them at the current version:

```
sudo apt-mark hold docker-ce kubelet kubeadm kubectl
```

Add the iptables rule to `sysctl.conf`:

```
echo "net.bridge.bridge-nf-call-iptables=1" | sudo tee -a /etc/sysctl.conf
```

Enable iptables immediately:

```
sudo sysctl -p
```

Initialize the cluster (run only on the master):

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

Set up local kubeconfig:

```
mkdir -p $HOME/.kube


sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config


sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Apply Flannel CNI network overlay:

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

Join the worker nodes to the cluster:

```
kubeadm join [your unique string from the kubeadm init command]
```

Verify the worker nodes have joined the cluster successfully:

```
kubectl get nodes
```

Compare this result of the kubectl get nodes command:

```
NAME                            STATUS   ROLES    AGE   VERSION
chadcrowell1c.mylabserver.com   Ready    master   4m18s v1.13.5
chadcrowell2c.mylabserver.com   Ready    none     82s   v1.13.5
chadcrowell3c.mylabserver.com   Ready    none     69s   v1.13.5
```

Building a Highly Available Kubernetes Cluster

Building a Highly Available Kubernetes Cluster

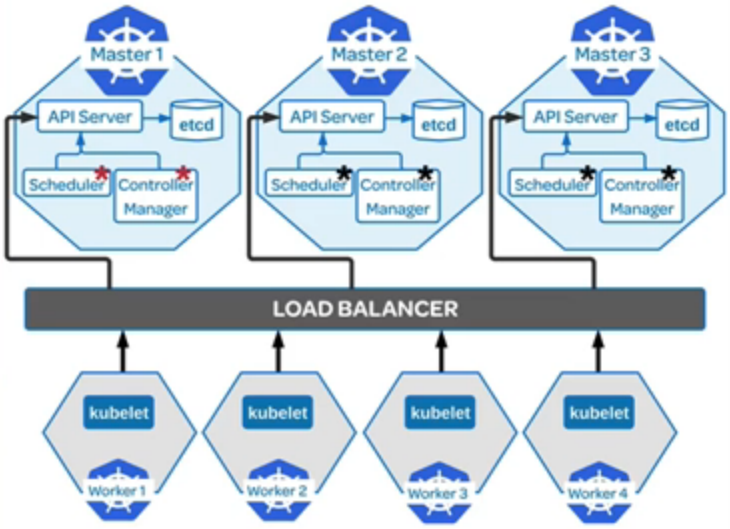
Next

Making Kubernetes Highly Available

All components of the Kubernetes cluster can be replicated, but only certain components can operate simultaneously.

* = standby

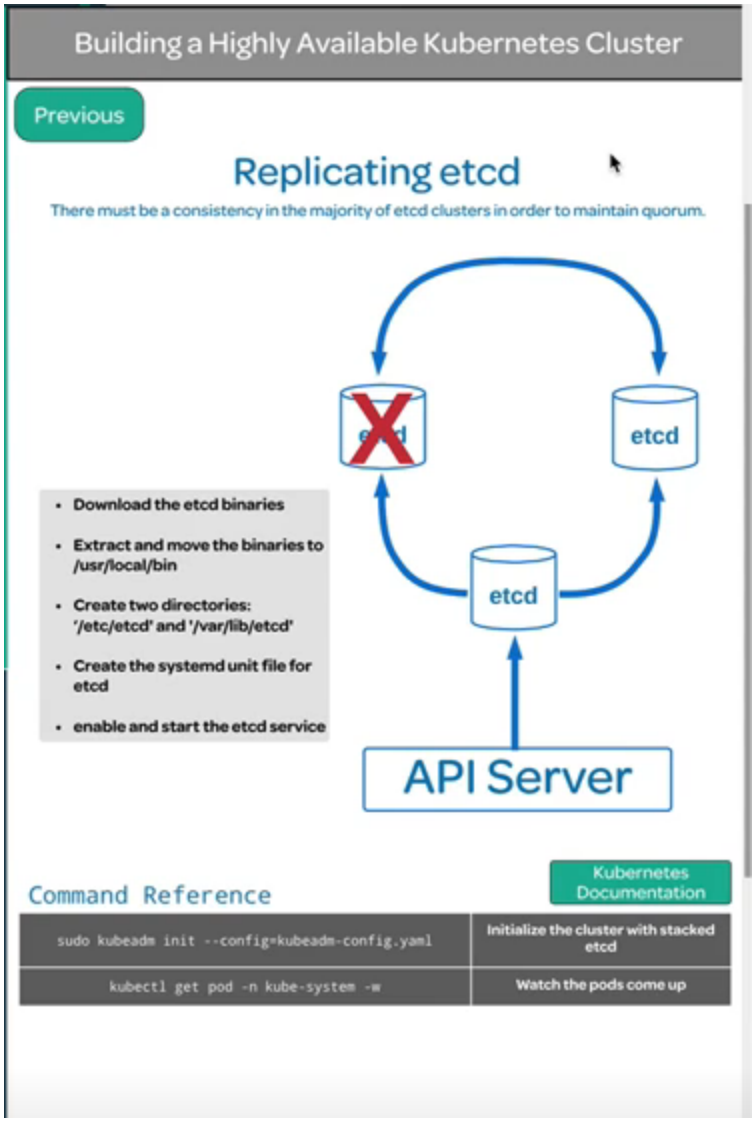
* = active



Command Reference

Kubernetes Documentation

`kubectl get pods -o custom-columns=POD:metadata.name,NODE:spec.nodeName --sort-by spec.nodeName -n kube-system`	View pods in the default namespace
`kubectl get endpoints kube-scheduler -n kube-system -o yaml`	View the kube-scheduler endpoint



You can provide high availability for cluster components by running multiple instances — however, some replicated components must remain in standby mode. The scheduler and the controller manager are actively watching the cluster state and take action when it changes. If multiples are running, it creates the possibility of unwarranted duplicates of pods.
View the pods in the default namespace with a custom view:

```
kubectl get pods -o custom-columns=POD:metadata.name,NODE:spec.nodeName --sort-by
spec.nodeName -n kube-system
```

View the kube-scheduler YAML:

```
kubectl get endpoints kube-scheduler -n kube-system -o yaml
```

Create a stacked etcd topology using kubeadm:

```
kubeadm init --config=kubeadm-config.yaml
```

Watch as pods are created in the default namespace:

```
kubectl get pods -n kube-system -w
```

Helpful Links:

- Creating Highly Available Kubernetes Clusters with kubeadm
- Highly Available Topologies in Kubernetes
- Operating a Highly Available etcd Cluster

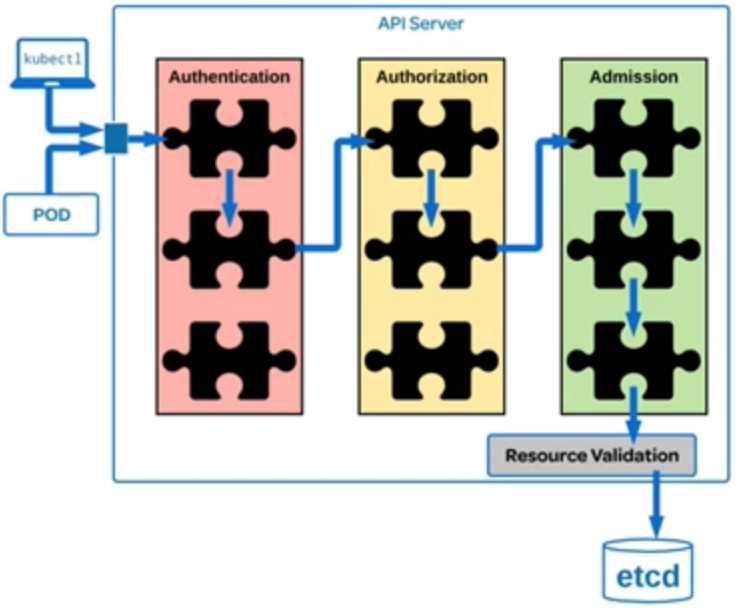
Configuring Secure Cluster Communications

Configuring Secure Cluster Communications

Next

Securing Kubernetes API Access

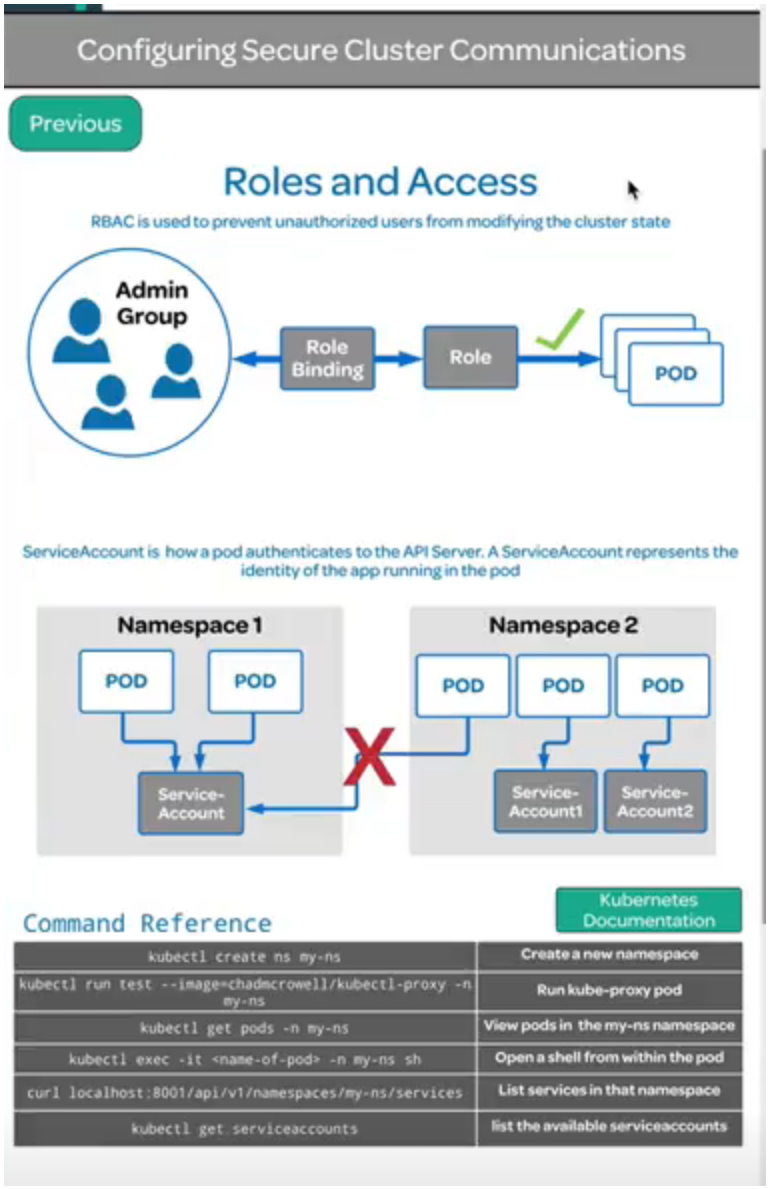
The Kubernetes API server provides a CRUD (Create, Read, Update, Delete) interface for querying and modifying the cluster state over a RESTful API.



Kubernetes Documentation

Command Reference

`cat .kube/config | more`	View the kube config
`kubectl get secrets`	View the service account token



To prevent unauthorized users from modifying the cluster state, RBAC is used, defining roles and role bindings for a user. A service account resource is created for a pod to determine how it has control over the cluster state. For example, the default service account will not allow you to list the services in a namespace.
View the kube-config:

```
cat .kube/config | more
```

View the service account token:

```
kubectl get secrets
```

Create a new namespace named `my-ns`:

```
kubectl create ns my-ns
```

Run the kube-proxy pod in the `my-ns` namespace:

```
kubectl run test --image=chadmcrowell/kubectl-proxy -n my-ns
```

List the pods in the `my-ns` namespace:

```
kubectl get pods -n my-ns
```

Run a shell in the newly created pod:

```
kubectl exec -it <name-of-pod> -n my-ns sh
```

List the services in the namespace via API call:

```
curl localhost:8001/api/v1/namespaces/my-ns/services
```

View the token file from within a pod:

```
cat /var/run/secrets/kubernetes.io/serviceaccount/token
```

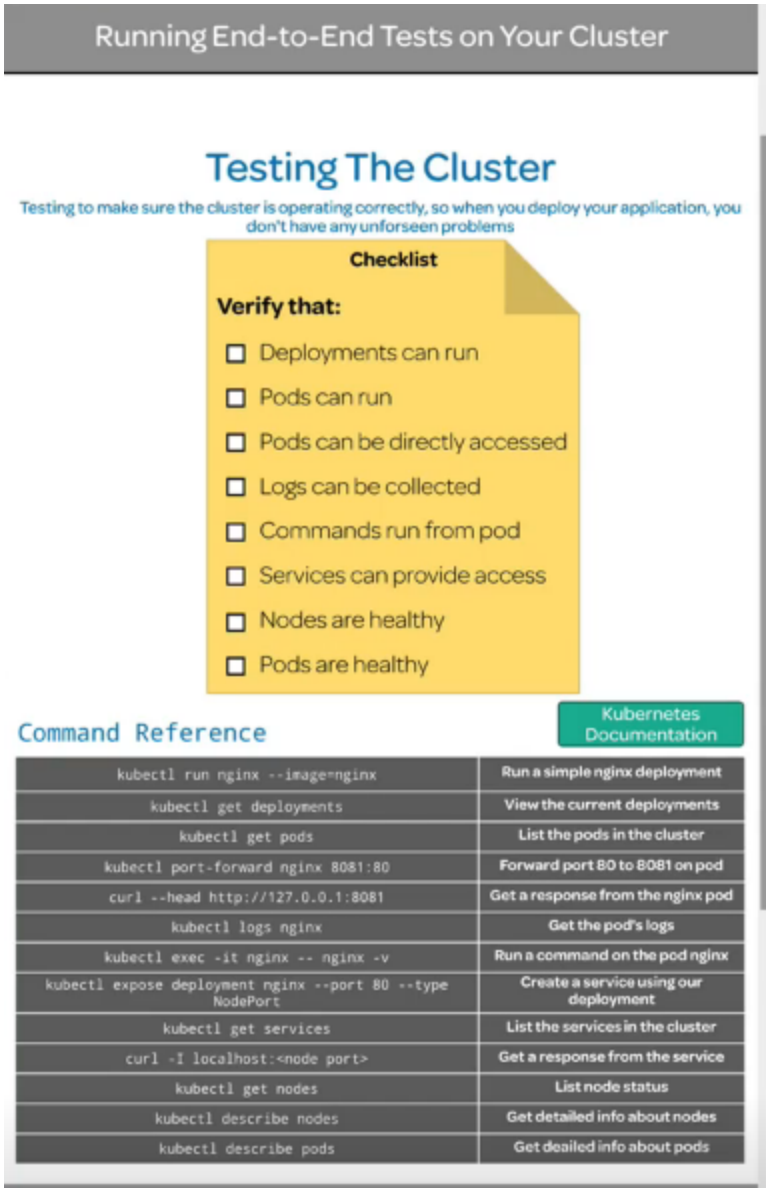
List the service account resources in your cluster:

```
kubectl get serviceaccounts
```

Helpful Links

- Service Accounts
- Cluster Administration Overview
- Securing the Cluster
- Controlling Access to the API
- Authorization
- Using a Proxy to Access the API

Running End-to-End Tests on Your Cluster



Running end-to-end tests ensures your application will run efficiently without having to worry about cluster health problems. Kubetest is a useful tool for providing end-to-end tests — however, it is beyond the scope of this exam. In this lesson, we will go through the practice of testing our ability to run deployments, run pods, expose a container, execute a command from a container, run a service, and check the overall health of our nodes and pods for conditions.
Run a simple nginx deployment:

```
kubectl run nginx --image=nginx
```

View the deployments in your cluster:

```
kubectl get deployments
```

View the pods in the cluster:

```
kubectl get pods
```

Use port forwarding to access a pod directly:

```
kubectl port-forward $pod_name 8081:80
```

Get a response from the nginx pod directly:

```
curl --head http://127.0.0.1:8081
```

View the logs from a pod:

```
kubectl logs $pod_name
```

Run a command directly from the container:

```
kubectl exec -it nginx -- nginx -v
```

Create a service by exposing port 80 of the nginx deployment:

```
kubectl expose deployment nginx --port 80 --type NodePort
```

List the services in your cluster:

```
kubectl get services
```

Get a response from the service:

```
curl -I localhost:$node_port
```

List the nodes' status:

```
kubectl get nodes
```

View detailed information about the nodes:

```
kubectl describe nodes
```

View detailed information about the pods:

```
kubectl describe pods
```

Helpful Links:

- Kubetest
- Test a Juju Cluster

Upgrade Kubernetes (**Managing the Kubernetes Cluster**)

- Upgrade kubeadm (on Master)
- Upgrade components (API server, control manager, scheduler, ectd) (On Master)

    Components that must be upgraded manually after you have upgraded the control plane with 'kubeadm upgrade apply':
    COMPONENT CURRENT AVAILABLE
    Kubelet 3 x v1.12.2 v1.13.8

    Upgrade to the latest stable version:

    COMPONENT CURRENT AVAILABLE
    API Server v1.12.10 v1.13.8
    Controller Manager v1.12.10 v1.13.8
    Scheduler v1.12.10 v1.13.8
    Kube Proxy v1.12.10 v1.13.8
    CoreDNS 1.2.2 1.2.6
    Etcd 3.2.24 3.2.24

- Upgrade kubelet (on Master)
- Upgrade kubelet (on Nodes)
- Upgrade kubectl (On Master)

Kubectl get nodes - version is kubelet version, it is installed on both master and worker nodes

```
master $ kubectl get nodes
NAME     STATUS   ROLES    AGE    VERSION
master   Ready    master   137m   v1.14.0
node01   Ready    <none>   137m   v1.14.0
master $
```

Note : Kubelet is a systemd service and not a POD.

Client version - kubectl version
Server version → API server version

```
eState:"clean", BuildDate:"2019-03-25T15:45:
master $ kubectl version --short
Client Version: v1.14.0
Server Version: v1.14.0
master $
```

Scheduler manager version

```
  startTime: "2019-07-13T14:55:122"
master $ kubectl get pods kube-scheduler-master -n kube-system -o yaml | grep image
    image: k8s.gcr.io/kube-scheduler:v1.14.0
    imagePullPolicy: IfNotPresent
    image: k8s.gcr.io/kube-scheduler:v1.14.0
    imageID: docker-pullable://k8s.gcr.io/kube-scheduler@sha256:cb35b2580cd0d97984106a81dcd0f1d9f63d774e18eeae40caef88d217f36b82
master $
```

Controller manager version

```
master $ kubectl get pods kube-controller-manager-master -n kube-system -o yaml | grep image
    image: k8s.gcr.io/kube-controller-manager:v1.14.0
    imagePullPolicy: IfNotPresent
    image: k8s.gcr.io/kube-controller-manager:v1.14.0
    imageID: docker-pullable://k8s.gcr.io/kube-controller-manager@sha256:433e56decf088553bdbe055610712dc1192453b2265376eea9af4aab9f574
b54
master $
```

ETCD version

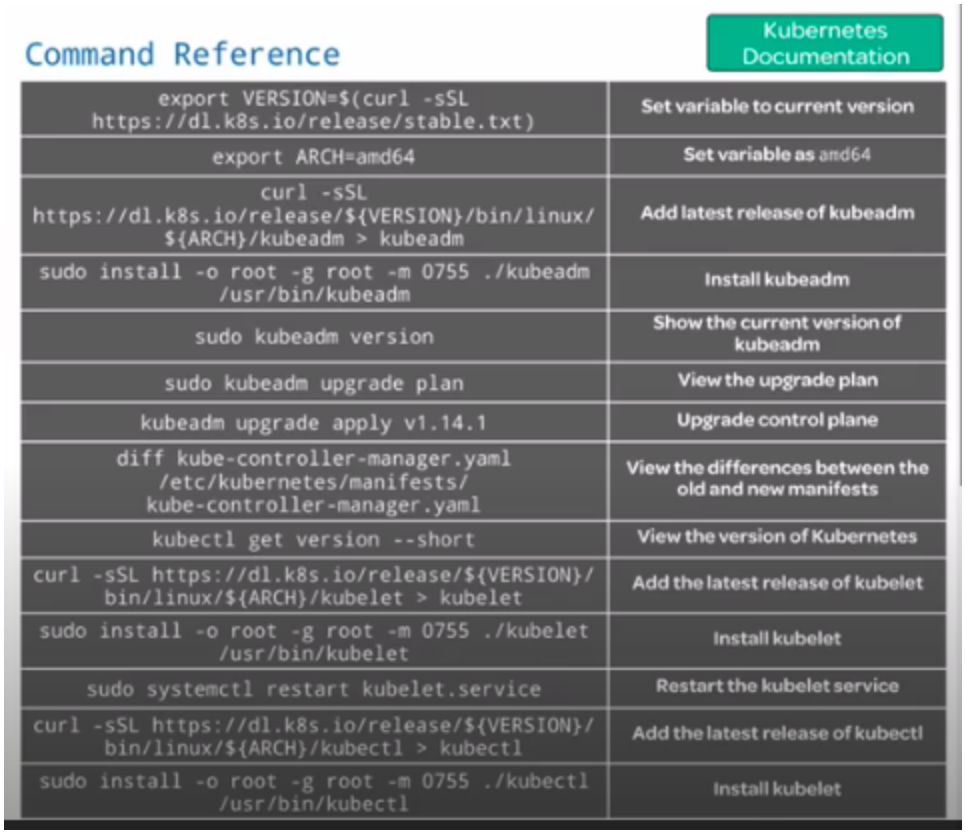
```
master $ kubectl get pods etcd-master -n kube-system -o yaml | grep image
    image: k8s.gcr.io/etcd:3.3.10
    imagePullPolicy: IfNotPresent
    image: k8s.gcr.io/etcd:3.3.10
    imageID: docker-pullable://k8s.gcr.io/etcd@sha256:17da501f5d2a675be46040422a27b7cc21b8a43895ac998b171db1c346f361f7
master $
```

Kubeadm version

```
cloud_user@ip-10-0-1-101:~$ sudo kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"13", GitVersion:"v1.13.5", Git
rsion:"go1.11.5", Compiler:"gc", Platform:"linux/amd64"}
```

Upgrade steps:

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



View the version of the server and client on the master node:
`kubectl version --short`

View the version of the scheduler and controller manager:
`kubectl get pods -n kube-system`
`kube-controller-manager-chadcrowell1c.mylabserver.com -o yaml`

View the name of the kube-controller pod:
`kubectl get pods -n kube-system`

Set the `VERSION` variable to the latest stable release of Kubernetes:

```
export VERSION=v1.14.1
```

Set the `ARCH` variable to the amd64 system:

```
export ARCH=amd64
```

View the latest stable version of Kubernetes using the variable:

```
echo $VERSION
```

Curl the latest stable version of Kubernetes:

```
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > kubeadm
```

Install the latest version of kubeadm:

```
sudo install -o root -g root -m 0755 ./kubeadm /usr/bin/kubeadm
```

Check the version of kubeadm:

```
sudo kubeadm version
```

Plan the upgrade:

```
sudo kubeadm upgrade plan
```

Apply the upgrade to 1.14.1:

```
kubeadm upgrade apply v1.14.1
```

View the differences between the old and new manifests:

```
diff kube-controller-manager.yaml /etc/kubernetes/manifests/kube-controller-manager.yaml
```

Curl the latest version of kubelet:

```
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubelet > kubelet
```

Install the latest version of kubelet:

```
sudo install -o root -g root -m 0755 ./kubelet /usr/bin/kubelet
```

Restart the kubelet service:

```
sudo systemctl restart kubelet.service
```

Watch the nodes as they change version:

```
kubectl get nodes -w
```

Node Maintenance

1. Move the runnings pods on target node to other nodes (DRAIN)
2. CORDON - disable pod scheduling on the node
3. Enable the node after maintenance (UNCORDON)
4. Optional .. Remove the node from cluster
5. Create another node, install components, and join back to cluster..

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

See which pods are running on which nodes:

```
kubectl get pods -o wide
```

Evict the pods on a node:

```
kubectl drain [node_name] --ignore-daemonsets
```

- Kube-proxy and flannel are daemon sets

Watch as the node changes status:

```
kubectl get nodes -w
```

Schedule pods to the node after maintenance is complete:

```
kubectl uncordon [node_name]
```

Remove a node from the cluster:

```
kubectl delete node [node_name]
```

Generate a new token:

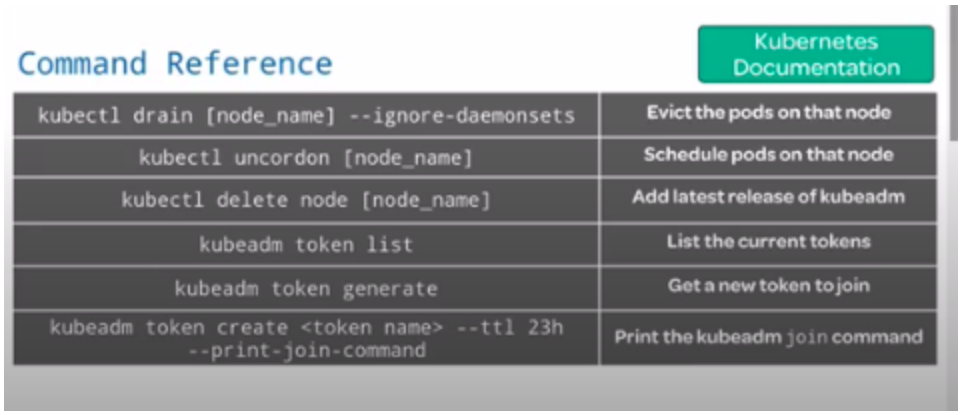
```
sudo kubeadm token generate
```

List the tokens:

```
sudo kubeadm token list
```

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

```
sudo kubeadm token create [token_name] --ttl 2h --print-join-command
```

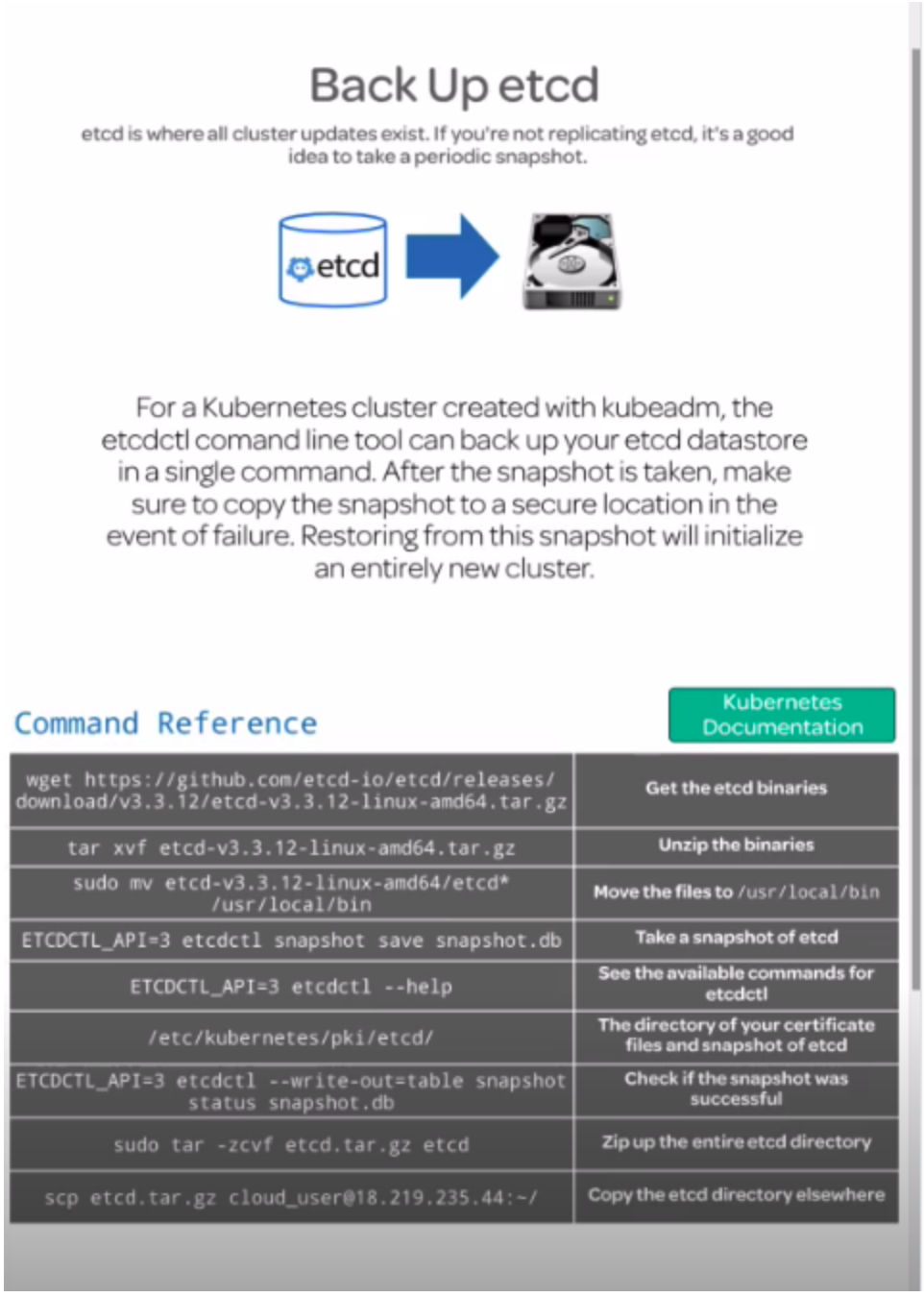


Resource Links

https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-upgrade/
https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG-1.13.md

https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/#maintenance-on-a-node

Backup & Restore



Backing up your cluster can be a useful exercise, especially if you have a single etcd cluster, as all the cluster state is stored there. The etcdctl utility allows us to easily create a snapshot of our cluster state (etcd) and save this to an external location. In this

lesson, we'll go through creating the snapshot and talk about restoring in the event of failure.

Get the etcd binaries:

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

Unzip the compressed binaries:

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

Move the files into `/usr/local/bin`:

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

Take a snapshot of the etcd datastore using etcdctl:

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

View the help page for etcdctl:

```
ETCDCTL_API=3 etcdctl --help
```

Browse to the folder that contains the certificate files:

```
cd /etc/kubernetes/pki/etcd/
```

View that the snapshot was successful:

```
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshot.db
```

Zip up the contents of the etcd directory:

```
sudo tar -zcvf etcd.tar.gz etcd
```

Copy the etcd directory to another server:

```
scp etcd.tar.gz cloud_user@18.219.235.42:~/
```

https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/#backing-up-an-etcd-cluster

https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/recovery.md

LAB

We have been given a three-node cluster that is in need of an upgrade. We must perform the upgrade to all of the cluster components, including kubeadm, kube-controller-manager, kube-scheduler, kubeadm, and kubectl. We need to perform the following to complete this hands-on lab:

- Get version 1.13.5 of kubeadm.
- Upgrade kubeadm and verify it has been installed correctly.
- Before initiating, first plan the upgrade to check for errors.
- Perform the upgrade to v1.13.5 and verify it was installed correctly.
- Get the latest version of kubelet.
- Install kubelet on each node and restart the kubelet service.
- Verify kubelet was installed correctly.
- Get the latest version of kubectl.
- Install the latest version of kubectl.

Learning Objectives

Get version 1.13.5 of kubeadm

Use the following commands to create a variable and get the latest version of kubeadm:

```
export VERSION=v1.13.5
export ARCH=amd64
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm >
kubeadm
```

Install kubeadm and verify it has been installed correctly.

Run the following commands to install kubeadm and verify the version:

```
sudo install -o root -g root -m 0755 ./kubeadm /usr/bin/kubeadm
sudo kubeadm version
```

Plan the upgrade in order to check for errors.

Use the following command to plan the upgrade:

```
sudo kubeadm upgrade plan
```

Perform the upgrade of the kube-scheduler and kube-controller-manager.

Use this command to apply the upgrade (also in the output of upgrade plan):

```
sudo kubeadm upgrade apply v1.13.5
```

Get the latest version of kubelet.

Use the following commands to get the latest version of kubelet on each node:

```
export VERSION=v1.13.5
export ARCH=amd64
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubelet >
kubelet
```

Install kubelet on each node and restart the kubelet service.

Use these commands to install kubelet and restart the kubelet service:

```
sudo install -o root -g root -m 0755 ./kubelet /usr/bin/kubelet
sudo systemctl restart kubelet.service
```

Verify the kubelet was installed correctly.

Use the following command to verify the kubelet was installed correctly:

```
kubectl get nodes
```

Get version 1.13.5 of kubectl

Use the following command to get the latest version of kubectl:

```
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubectl >
kubectl
```

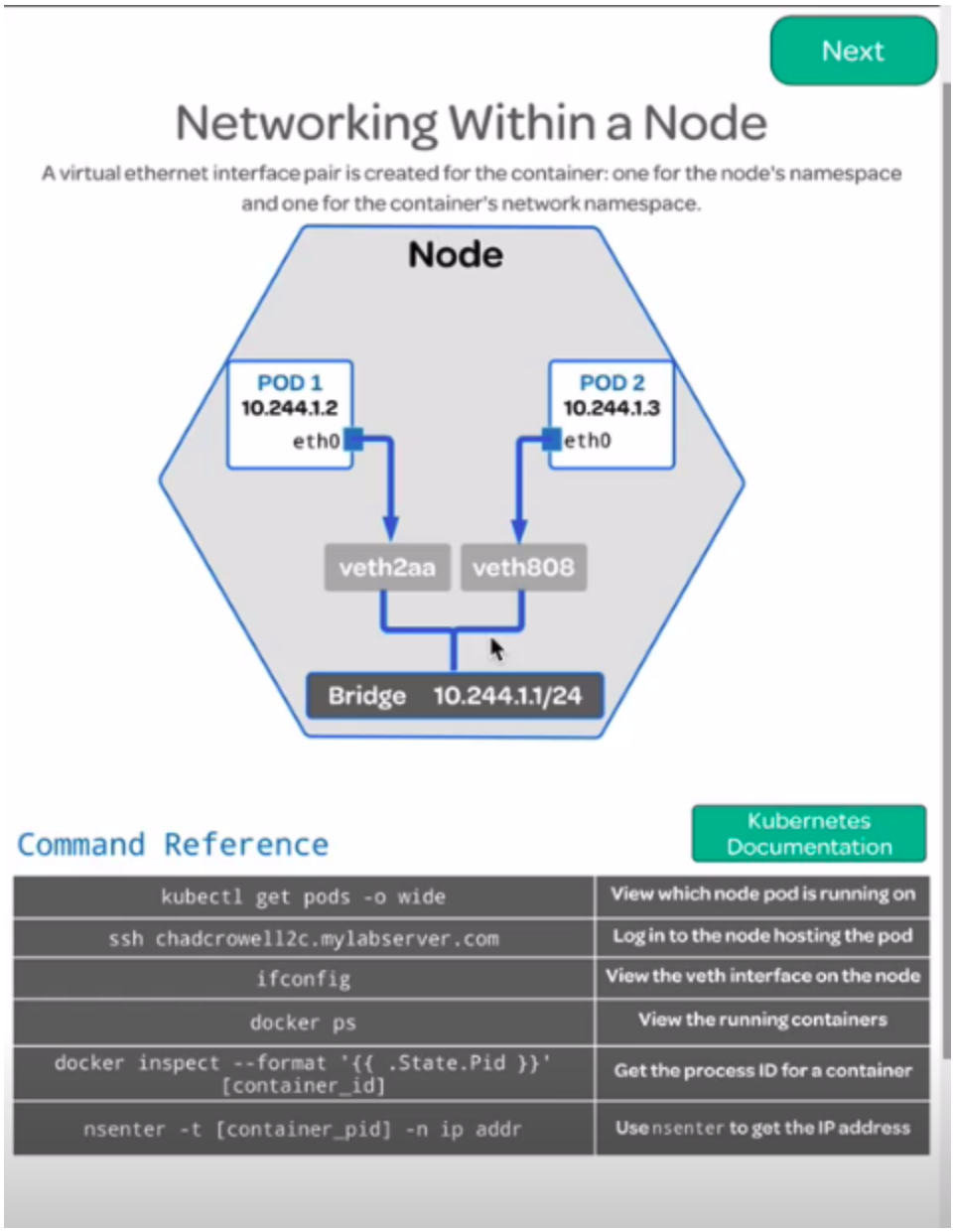
Upgrade kubectl

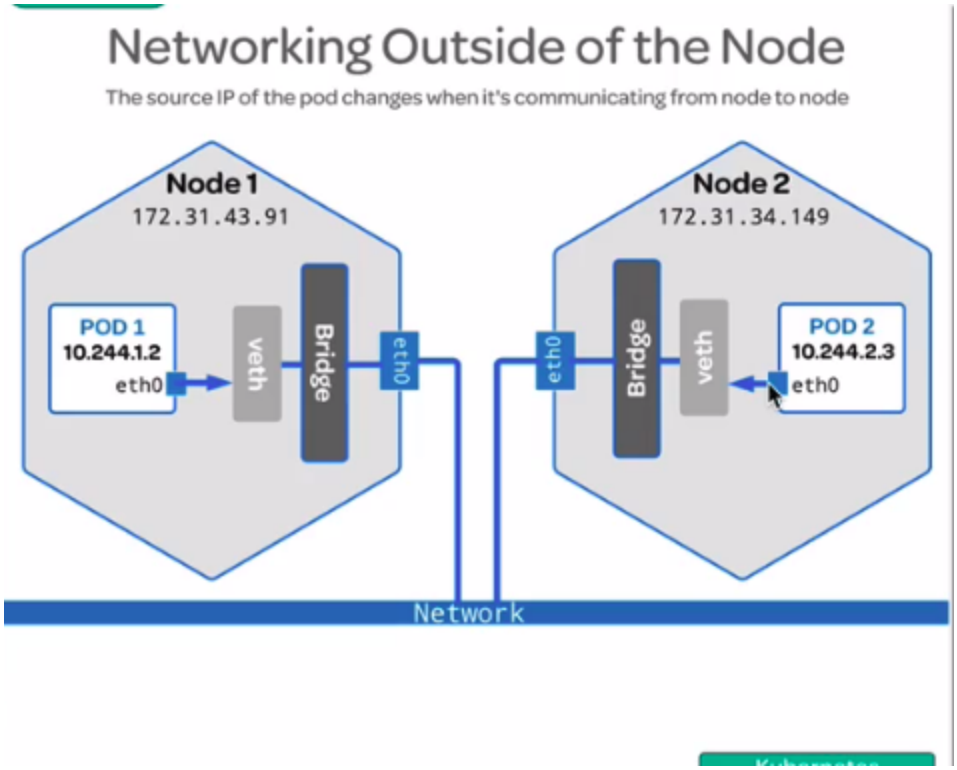
Use the following command to install the latest version of kubectl:

```
sudo install -o root -g root -m 0755 ./kubectl /usr/bin/kubectl
```

Networking

Pod and Node networking





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

See which node our pod is on:

```
kubectl get pods -o wide
```

Log in to the node:

```
ssh [node_name]
```

View the node's virtual network interfaces:

```
ifconfig
```

View the containers in the pod:

```
docker ps
```

Get the process ID for the container:

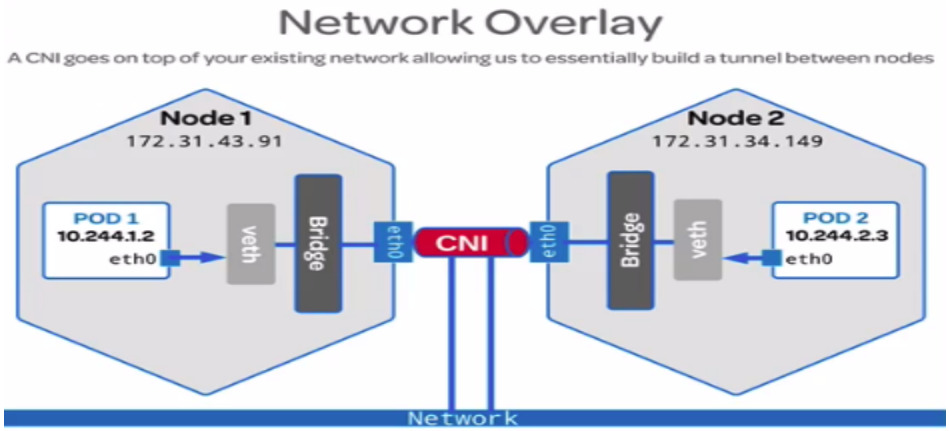
```
docker inspect --format '{{ .State.Pid }}' [container_id]
```

Use nsenter to run a command in the process's network namespace:
`nsenter -t [container_pid] -n ip addr`

https://kubernetes.io/docs/concepts/cluster-administration/networking/

Container network interface (CNI)

- CNI installed as a plugin.. Not part of kubernetes
- Maintains mapping info (POD to Node & Node to POD)
- Uses encapsulation and decapsulation for traffic management..
- Runs on every node as a daemon set..
- Installs a network agent on each node..
- Kubelet needs to be notified on the CNI..
- Kubeadm init --port-CIDR = ???
- Does IP address management and IP assignment



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

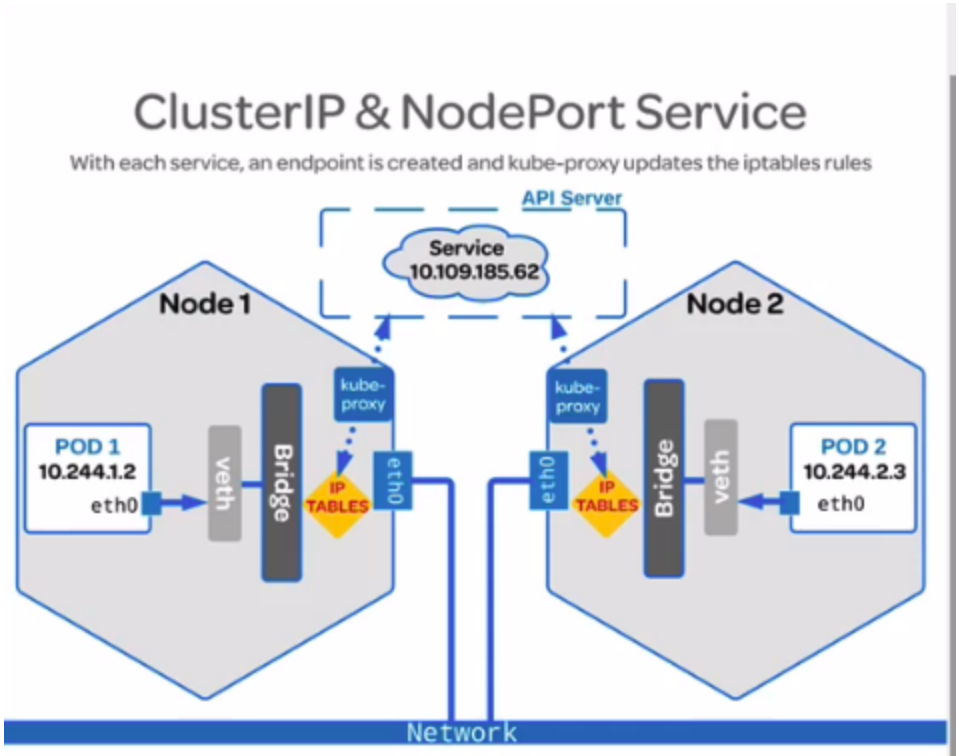
```
kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Documentation/kube-flannel.yml
```

https://github.com/coreos/flannel/blob/master/Documentation/kubernetes.md

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/#pod-network

https://kubernetes.io/docs/concepts/cluster-administration/addons/

Service networking



Command Reference

Kubernetes Documentation

`kubectl get services -o yaml`	View the spec of our service
`ping 10.96.0.1`	Trying to ping a service
`kubectl get services`	View the services in our cluster
`sudo iptables-save | grep KUBE | grep nginx`	View the iptables rules

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

YAML for the nginx `NodePort` service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30080
  selector:
    app: nginx
```

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

```
kubectl get services -o yaml
```

Try and ping the `clusterIP` service IP address:

```
ping 10.96.0.1
```

View the list of services in your cluster:

```
kubectl get services
```

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

```
kubectl get endpoints
```

Look at the iptables rules for your services:

```
sudo iptables-save | grep KUBE | grep nginx
```

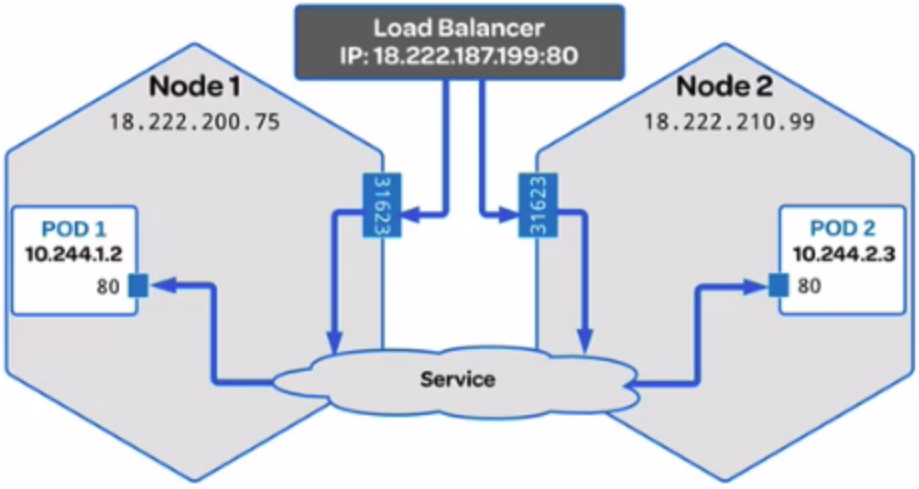
https://kubernetes.io/docs/concepts/services-networking/service/

Ingress Rules and Load Balancers

Next

Load Balancers

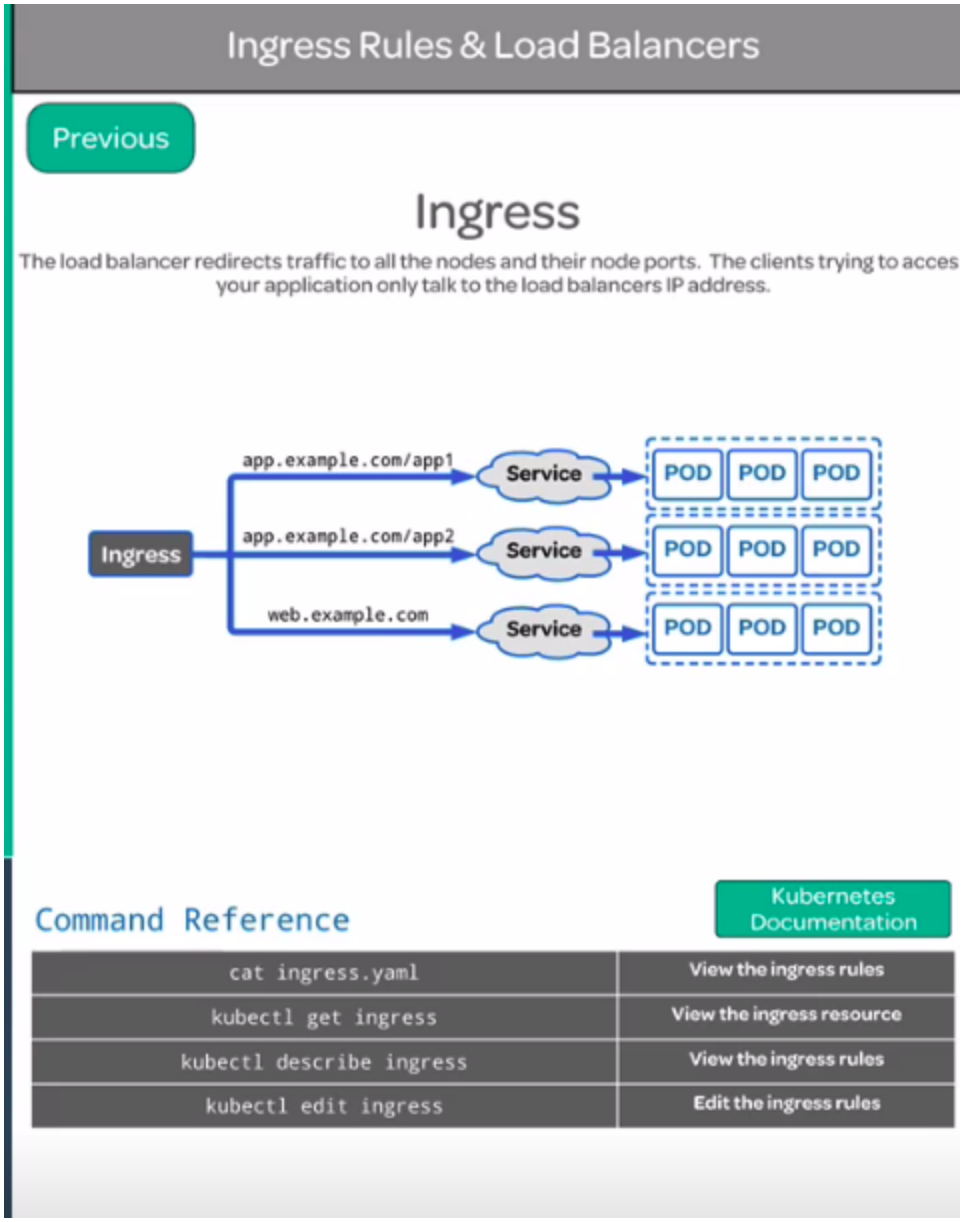
The load balancer redirects traffic to all the nodes and their node ports. The clients trying to access your application only talk to the load balancers IP address.



Command Reference

Kubernetes Documentation

`cat nginx-loadbalancer.yaml`	View the load balancer yaml
`kubectl run kubeserve2 --image=chadmcrowell/kubeserve2`	Create a deployent named kubeserve2
`kubectl get deployments`	View the deployments
`kubectl scale deployment/kubeserve2 --replicas=2`	Scale the deployment up to 2
`kubectl get pods -o wide`	View the 2 pods in the deployment
`kubectl expose deployment kubeserve2 --port 80 --target-port 8080 --type LoadBalancer`	Create the LoadBalancer service
`kubectl get services`	View the LoadBalancer service
`kubectl get services kubeserve2 -o yaml`	View the YAML output for service kubeserve2
`curl http://[external ip address]`	Access the service through the new LoadBalancer



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

distributes traffic to the pods on a node to reduce latency and direct traffic to the appropriate services within your cluster.
View the list of services:

```
kubectl get services
```

The load balancer YAML spec:

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

Create a new deployment:

```
kubectl run kubeserve2 --image=chadmcrowell/kubeserve2
```

View the list of deployments:

```
kubectl get deployments
```

Scale the deployments to 2 replicas:

```
kubectl scale deployment/kubeserve2 --replicas=2
```

View which pods are on which nodes:

```
kubectl get pods -o wide
```

Create a load balancer from a deployment:

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

View the services in your cluster:

```
kubectl get services
```

Watch as an external port is created for a service:

```
kubectl get services -w
```

Look at the YAML for a service:

```
kubectl get services kubeserve2 -o yaml
```

Curl the external IP of the load balancer:

```
curl http://[external-ip]
```

View the annotation associated with a service:

```
kubectl describe services kubeserve
```

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

```
kubectl annotate service kubeserve2 externalTrafficPolicy=Local
```

The YAML for an Ingress resource:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: service-ingress
spec:
  rules:
  - host: kubeserve.example.com
    http:
      paths:
      - backend:
          serviceName: kubeserve2
          servicePort: 80
  - host: app.example.com
    http:
      paths:
      - backend:
          serviceName: nginx
          servicePort: 80
  - http:
      paths:
      - backend:
          serviceName: httpd
          servicePort: 80
```

Edit the ingress rules:

```
kubectl edit ingress
```

View the existing ingress rules:

```
kubectl describe ingress
```

Curl the hostname of your Ingress resource:

```
curl http://kubeserve2.example.com
```

LB -- > NODE -- > SERVICE → IP tables --> POD

POD can be located on any node.. Below settings ensure that traffic is only sent to PODS located to local PODS to avoid additional network hops..

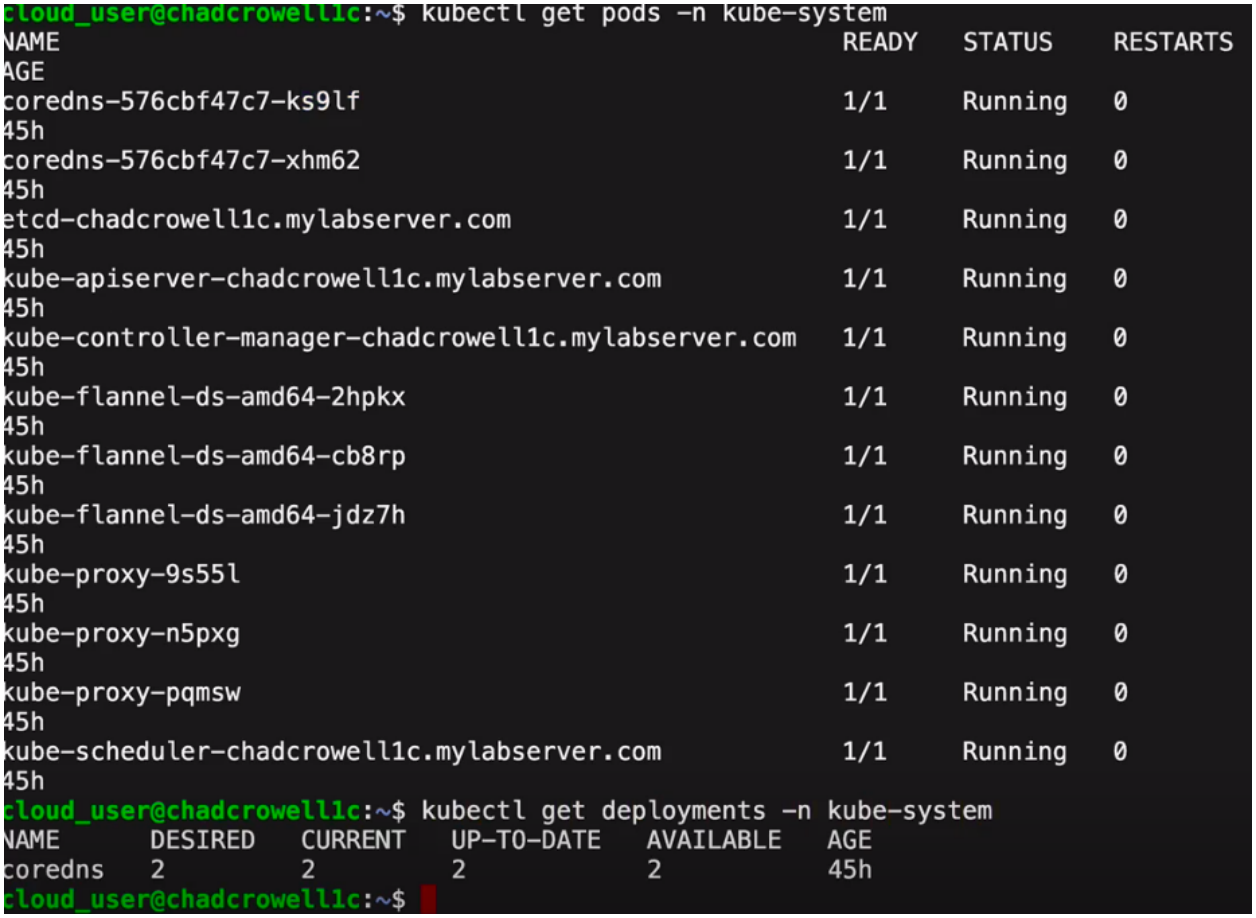
```
chadmcrowell@cloudshell:~ (k8s-cluster-233419)$ kubectl annotate service kube
serve2 externalTrafficPolicy=Local
service "kubeserve2" annotated
chadmcrowell@cloudshell:~ (k8s-cluster-233419)$ kubectl describe services kub
eserve2
Name:                     kubeserve2
Namespace:                default
Labels:                   run=kubeserve2
Annotations:              externalTrafficPolicy=Local
Selector:                 run=kubeserve2
Type:                     LoadBalancer
IP:                       10.19.247.170
LoadBalancer Ingress:     35.202.6.37
Port:                     <unset>  80/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  32029/TCP
Endpoints:                10.16.1.6:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason                Age              From                Message
  ----    ------                ----             ----                -------
  Normal  EnsuredLoadBalancer   14m              service-controller  Ensured
 load balancer
  Normal  EnsuringLoadBalancer  6s (x2 over 14m)  service-controller  Ensurin
```

Load balancer - One external IP per service
Ingress controller - One external IP , different service.

https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/
https://kubernetes.io/docs/concepts/services-networking/ingress/

Cluster DNS

- As of 1.13, core DNS has replaced kube-dns
- Written in go , supports buffering
- Supports DNS over TLS..
- Has plugin architecture..
- Easy integration with others..
- 2 pods runs on master node..
- Runs as a deployment and have associated service, name is kube-dns for backword compatibility



```
master $ kubectl get services -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   50m
master $
```

All pods running in kube-system name space..

```
master $ kubectl get pods -n kube-system
NAME                              READY   STATUS    RESTARTS   AGE
coredns-fb8b8dccf-m7t4j           1/1     Running   0          34m
coredns-fb8b8dccf-twdtm           1/1     Running   0          34m
etcd-master                       1/1     Running   0          33m
kube-apiserver-master             1/1     Running   0          33m
kube-controller-manager-master    1/1     Running   0          33m
kube-proxy-px8sn                  1/1     Running   0          34m
kube-proxy-ws2mr                  1/1     Running   0          34m
kube-scheduler-master             1/1     Running   0          33m
weave-net-k8gnr                   2/2     Running   1          34m
weave-net-xr7mk                   2/2     Running   0          34m
master $
```

Etcd, controller manager, API server, scheduler ----> Runs as PODS..no deployment
Core DNS -----> Runs as deployment and PODS, have associated service..

```
master $ kubectl get deployments -n kube-system
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
coredns   2/2     2            2           34m
```

Kube-proxy + CNI (weave-net or flannel) -----> runs a daemon set..

```
master $ kubectl get daemonsets -n kube-system
NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
kube-proxy   2         2         2       2            2           <none>          41m
weave-net    2         2         2       2            2           <none>          41m
master $
```

Kubelet ----> runs as systemd service

```
master $ systemctl list-unit-files --state=enabled | grep kubelet
kubelet.service                enabled
master $
```

List all supported resources/objects...

```
master $ kubectl api-resources
NAME                              SHORTNAMES   APIGROUP                       NAMESPACED   KIND
bindings                                                                      true         Binding
componentstatuses                 cs                                          false        ComponentStatus
configmaps                        cm                                          true         ConfigMap
endpoints                         ep                                          true         Endpoints
events                            ev                                          true         Event
limitranges                       limits                                      true         LimitRange
namespaces                        ns                                          false        Namespace
nodes                             no                                          false        Node
persistentvolumeclaims            pvc                                         true         PersistentVolumeClaim
persistentvolumes                 pv                                          false        PersistentVolume
pods                              po                                          true         Pod
podtemplates                                                                  true         PodTemplate
replicationcontrollers            rc                                          true         ReplicationController
resourcequotas                    quota                                       true         ResourceQuota
secrets                                                                       true         Secret
serviceaccounts                   sa                                          true         ServiceAccount
```

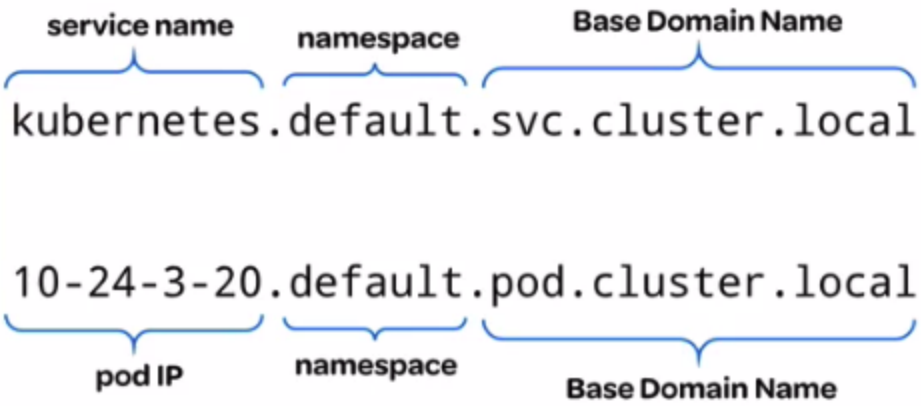
Resolving names..

```
cloud_user@chadcrowell1c:~$ kubectl exec -t busybox -- cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local us-east-2.compute.in
ternal
options ndots:5
cloud_user@chadcrowell1c:~$ kubectl exec -it busybox -- nslookup kubernetes
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes
Address 1: 10.96.0.1 kubernetes.default.svc.cluster.local
cloud_user@chadcrowell1c:~$ kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP           NODE
                         NOMINATED NODE
busybox                       1/1     Running   0          2m46s   10.244.1.7   chadc
rowell2c.mylabserver.com   <none>
kubeserve2-86c4dbddfb-7f2pn   1/1     Running   0          122m    10.244.2.5   chadc
rowell3c.mylabserver.com   <none>
kubeserve2-86c4dbddfb-dx7wb   1/1     Running   0          122m    10.244.1.6   chadc
rowell2c.mylabserver.com   <none>
nginx-dbddb74b8-xnpht         1/1     Running   0          45h     10.244.2.2   chadc
rowell3c.mylabserver.com   <none>
cloud_user@chadcrowell1c:~$ kubectl exec -it busybox -- nslookup 10-244-1-6.default.p
od.cluster.local
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      10-244-1-6.default.pod.cluster.local
Address 1: 10.244.1.6
cloud_user@chadcrowell1c:~$ kubectl exec -it busybox -- nslookup kube-dns.kube-system
.svc.cluster.local
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kube-dns.kube-system.svc.cluster.local
```

Cluster DNS

DNS

Every service defined in the cluster is assigned a DNS name. A Pod's DNS search list will include the Pod's own namespace and the cluster's default domain



Command Reference

Kubernetes Documentation

`kubectl get pods -n kube-system`	View the coredns pods
`kubectl get deployments -n kube-system`	View the coredns deployment
`kubectl get services -n kube-system`	View the kube-dns service
`kubectl create -f busybox.yaml`	Create a busybox pod
`kubectl exec -it busybox -- cat /etc/resolv.conf`	Look at the resolv.conf
`kubectl exec -it busybox -- nslookup kubernetes`	Lookup the kubernetes service DNS
`kubectl exec -ti busybox -- nslookup 10-244-1-2.default.pod.cluster.local`	Lookup a pod DNS
`kubectl exec -it busybox -- nslookup kube-dns.kube-system.svc.cluster.local`	Lookup a service DNS
`kubectl logs [coredns-pod-name]`	get the logs for coredns errors
`kubectl create -f headless-service.yaml`	Create headless service
`kubectl create -f custom-dns.yaml`	Create a custom DNS pod

Headless service…

- Service without a cluster IP..

Explore..

Custom DNS policy

Overwite default DNS configuration
Set own DNS name and search path
dnsPolicy is set to None

```
cloud_user@chadcrowell1c:~$ cat custom-dns.yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 8.8.8.8
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
```

CoreDNS is now the new default DNS plugin for Kubernetes. In this lesson, we'll go over the hostnames for pods and services. We will also discover how you can customize DNS to include your own nameservers.
View the CoreDNS pods in the `kube-system` namespace:

```
kubectl get pods -n kube-system
```

View the CoreDNS deployment in your Kubernetes cluster:

```
kubectl get deployments -n kube-system
```

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

```
kubectl get services -n kube-system
```

Spec for the busybox pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox:1.28.4
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

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

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

Look up the DNS name for the native Kubernetes service:

```
kubectl exec -it busybox -- nslookup kubernetes
```

Look up the DNS names of your pods:

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

Look up a service in your Kubernetes cluster:

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

Get the logs of your CoreDNS pods:

```
kubectl logs [coredns-pod-name]
```

YAML spec for a headless service:

```
apiVersion: v1
kind: Service
metadata:
  name: kube-headless
spec:
  clusterIP: None
  ports:
```

```
  - port: 80
    targetPort: 8080
  selector:
    app: kubserve2
```

YAML spec for a custom DNS pod:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: dns-example
spec:
  containers:
    - name: test
      image: nginx
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 8.8.8.8
    searches:
      - ns1.svc.cluster.local
      - my.dns.search.suffix
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/
https://kubernetes.io/docs/tasks/administer-cluster/dns-debugging-resolution/
https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/
https://github.com/coredns/deployment/tree/master/kubernetes
https://github.com/kubernetes/dns/blob/master/docs/specification.md
https://coredns.io/2018/01/29/deploying-kubernetes-with-coredns-using-kubeadm/

LAB

Learning Objectives

Create an nginx deployment, and verify it was successful.

Use this command to create an nginx deployment:

```
kubectl run nginx --image=nginx
```

Use this command to verify deployment was successful:

```
kubectl get deployments
```

Create a service, and verify the service was successful.

Use this command to create a service:

```
kubectl expose deployment nginx --port 80 --type NodePort
```

Use this command to verify the service was created:

```
kubectl get services
```

Create a pod that will allow you to query DNS, and verify it's been created.

Use the following YAML to create the busybox pod spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  containers:
  - image: busybox:1.28.4
    command:
      - sleep
      - "3600"
    name: busybox
  restartPolicy: Always
```

Use the following command to create the busybox pod:

```
kubectl create -f busybox.yaml
```

Use the following command to verify the pod was created successfully:

```
kubectl get pods
```

Perform a DNS query to the service.

Use the following command to query the DNS name of the nginx service:

```
kubectl exec busybox -- nslookup nginx
```

Record the DNS name.

Record the name of:

```
<service-name>;.default.svc.cluster.local
```

Scheduling

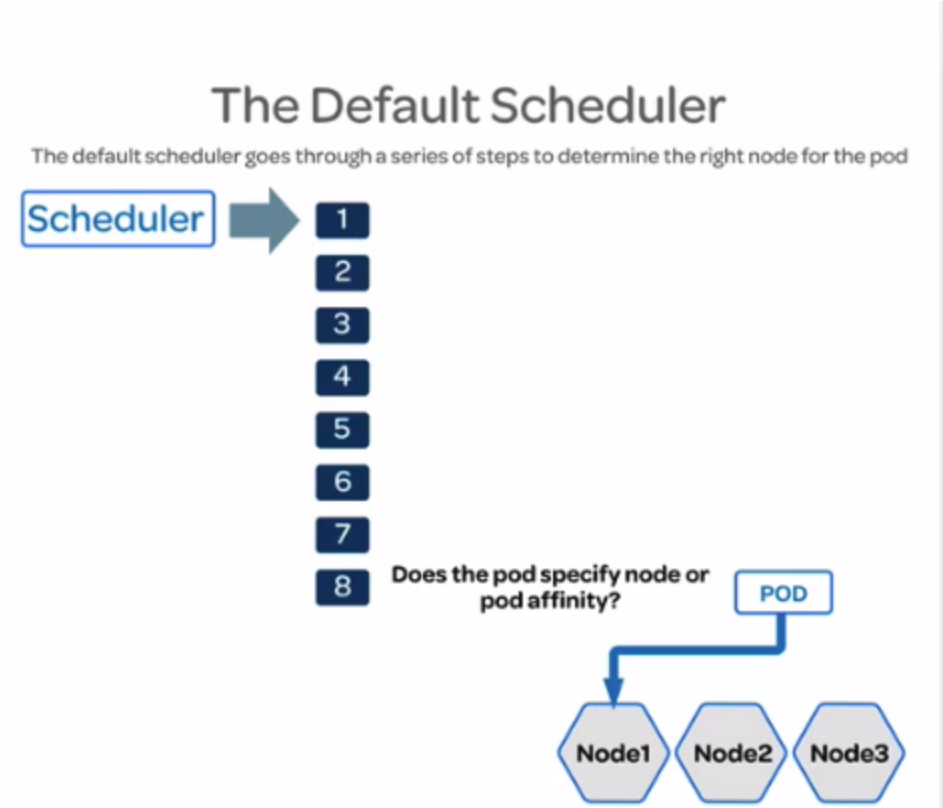
Configuring the Kubernetes Scheduler

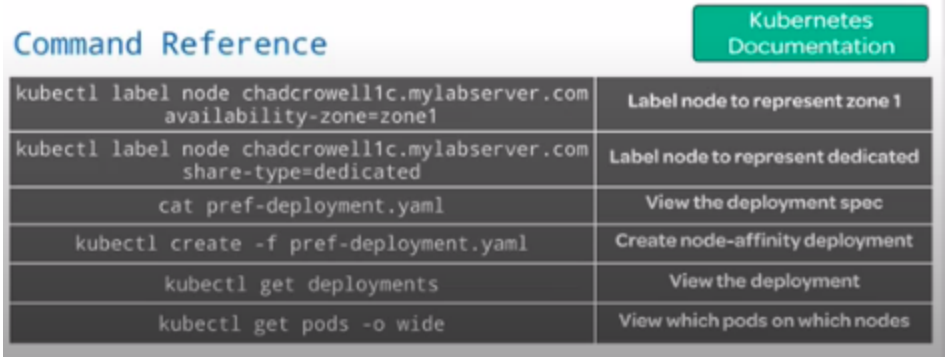
Need for scheduling pods to a specific hosts.

- Licensing..
- Special disk needs..
- Specific availability zones..

Tests for scheduling

https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/





1) Resources available - Requests and limits
2) NodeSelector
3) Requested port available on node ?
4) Requested volumes can be attached to node..
5) Taints and tolerations
6) Affinity and anti affinity

The default scheduler in Kubernetes attempts to find the best node for your pod by going through a series of steps. In this lesson, we will cover the steps in detail in order to better understand the scheduler's function when placing pods on nodes to maximize uptime for the applications running in your cluster. We will also go through how to create a deployment with node affinity.
Label your node as being located in availability zone 1:

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

Label your node as dedicated infrastructure:

```
kubectl label node chadcrowell1c.mylabserver.com share-type=dedicated
```

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

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: pref
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: pref
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 80
            preference:
              matchExpressions:
              - key: availability-zone
                operator: In
                values:
                - zone1
          - weight: 20
            preference:
              matchExpressions:
```

```
              - key: share-type
                operator: In
                values:
                - dedicated
      containers:
      - args:
        - sleep
        - "99999"
        image: busybox
        name: main
```

Create the deployment:

```
kubectl create -f pref-deployment.yaml
```

View the deployment:

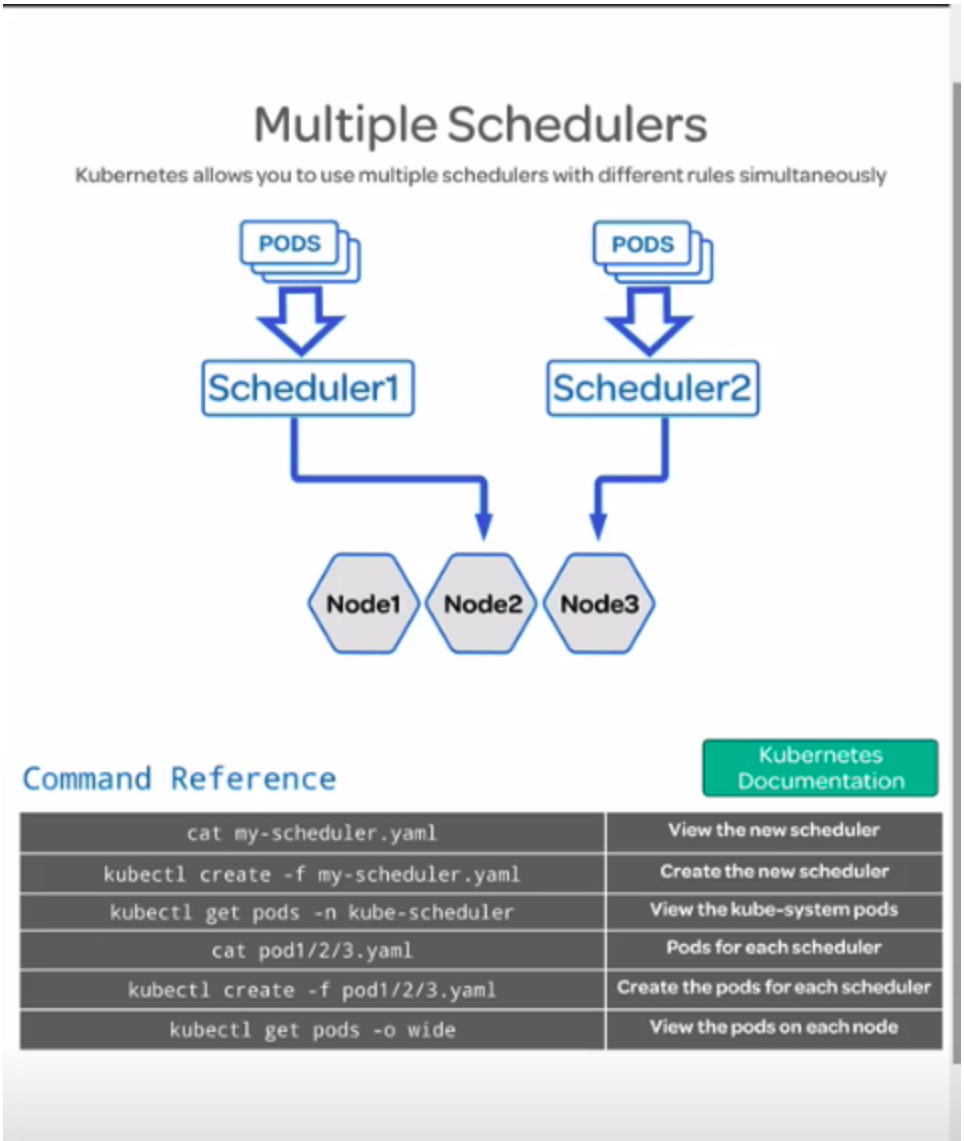
```
kubectl get deployments
```

View which pods landed on which nodes:

```
kubectl get pods -o wide
```

https://kubernetes.io/docs/concepts/configuration/assign-pod-node/
https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#affinity-and-anti-affinity

Running Multiple Schedulers for Multiple Pods



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

```
apiVersion: extensions/v1beta1
```

```
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: my-scheduler
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      containers:
      - command: [/usr/local/bin/kube-scheduler, --address=0.0.0.0,
                  --scheduler-name=my-scheduler, --leader-elect=false]
        image: linuxacademycontent/content-kube-scheduler
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10251
          initialDelaySeconds: 15
        name: kube-second-scheduler
        readinessProbe:
          httpGet:
            path: /healthz
            port: 10251
        resources:
          requests:
            cpu: '0.1'
        securityContext:
          privileged: false
        volumeMounts: []
      hostNetwork: false
      hostPID: false
      volumes: []
```

Run the deployment for `my-scheduler`:

```
kubectl create -f my-scheduler.yaml
```

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

```
kubectl get pods -n kube-system
```

The pod YAML for pod 1:

```
apiVersion: v1
kind: Pod
metadata:
  name: no-annotation
  labels:
    name: multischeduler-example
spec:
  containers:
  - name: pod-with-no-annotation-container
    image: k8s.gcr.io/pause:2.0
```

The pod YAML for pod 2:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-default-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: default-scheduler
  containers:
  - name: pod-with-default-annotation-container
    image: k8s.gcr.io/pause:2.0
```

The pod YAML for pod 3:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
  - name: pod-with-second-annotation-container
    image: k8s.gcr.io/pause:2.0
```

View the pods as they are created:

```
kubectl get pods -o wide
```

https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/

Scheduling Pods with Resource Limits and Label Selectors

Nodes - taints and tolerations

https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/

Affinity - POD property
Taint - Node property
- Don't schedule any POD except ones with matching tolerations..

Tolerations - POD property

```
kubectl taint nodes node1 key=value:NoSchedule
```

```
tolerations:
- key: "key"
  operator: "Equal"
  value: "value"

  effect: "NoSchedule"
```

```
tolerations:
- key: "key"
  operator: "Exists"

  effect: "NoSchedule"
```

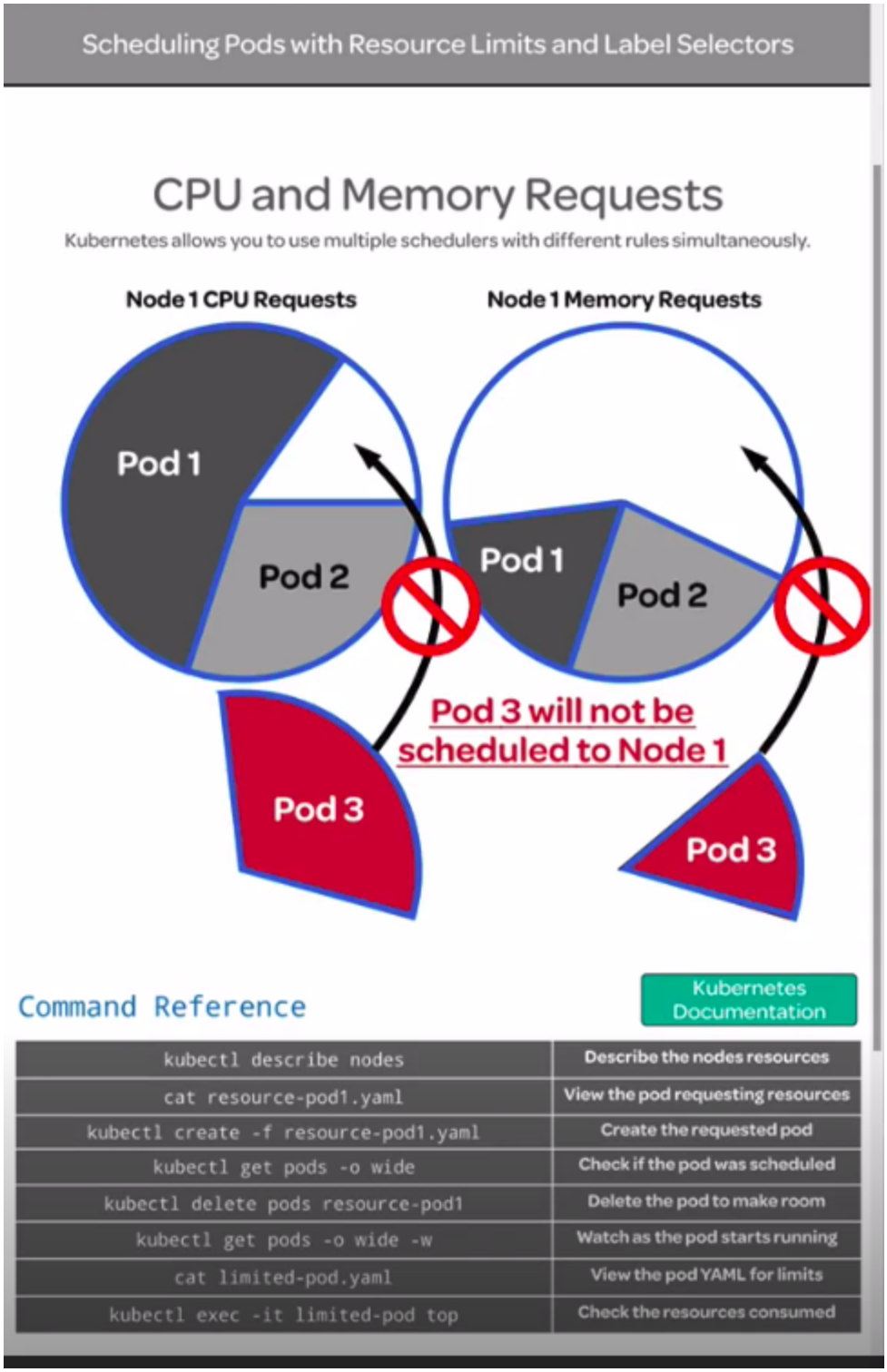
A toleration “matches” a taint if the keys are the same and the effects are the same,

and:

- the `operator` is `Exists` (in which case no `value` should be specified), or

- the `operator` is `Equal` and the `value`s are equal

`Operator` defaults to `Equal` if not specified.



In order to share the resources of your node properly, you can set resource limits and requests in Kubernetes. This allows you to reserve enough CPU and memory for your

application while still maintaining system health. In this lesson, we will create some requests and limits in our pod YAML to show how it's used by the node.
View the capacity and the allocatable info from a node:

```
kubectl describe nodes
```

The pod YAML for a pod with requests:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-pod1
spec:
  nodeSelector:
    kubernetes.io/hostname: "chadcrowell3c.mylabserver.com"
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: pod1
    resources:
      requests:
        cpu: 800m
        memory: 20Mi
```

Create the requests pod:

```
kubectl create -f resource-pod1.yaml
```

View the pods and nodes they landed on:

```
kubectl get pods -o wide
```

The YAML for a pod that has a large request:

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-pod2
spec:
  nodeSelector:
    kubernetes.io/hostname: "chadcrowell3c.mylabserver.com"
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: pod2
    resources:
      requests:
```

```
        cpu: 1000m
        memory: 20Mi
```

Create the pod with 1000 millicore request:

```
kubectl create -f resource-pod2.yaml
```

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

```
kubectl describe resource-pod2
```

Look at the total requests per node:

```
kubectl describe nodes chadcrowell3c.mylabserver.com
```

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

```
kubectl delete pods resource-pod1
```

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

```
kubectl get pods -o wide -w
```

The YAML for a pod that has limits:

```
apiVersion: v1
kind: Pod
metadata:
  name: limited-pod
spec:
  containers:
  - image: busybox
    command: ["dd", "if=/dev/zero", "of=/dev/null"]
    name: main
    resources:
      limits:
        cpu: 1
        memory: 20Mi
```

Create a pod with limits:

```
kubectl create -f limited-pod.yaml
```

Use the exec utility to use the top command:

```
kubectl exec -it limited-pod top
```

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace/

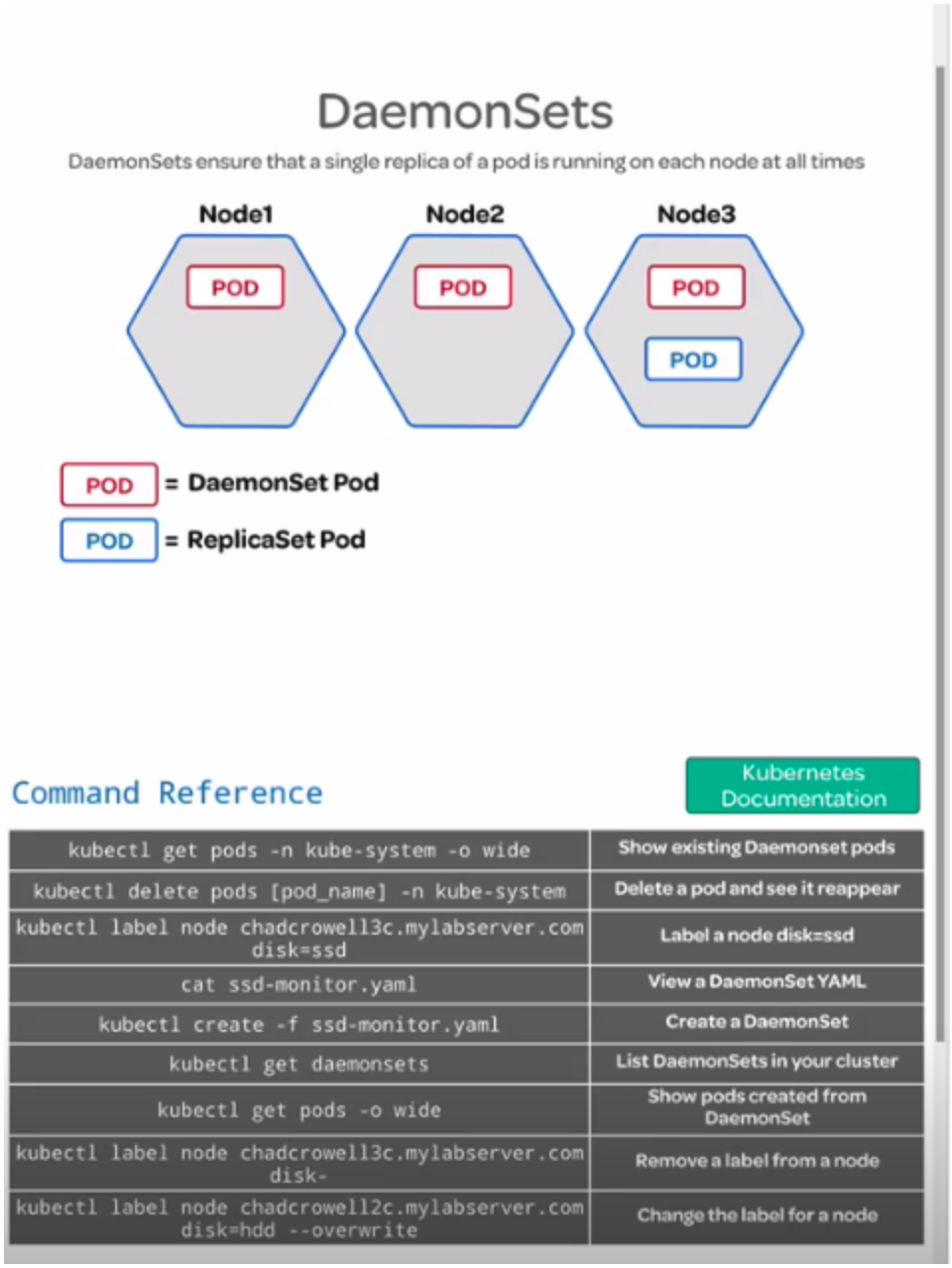
https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/

Resource limits and requests..

Container - top command - only shows node memory…

DaemonSets and Manually Scheduled Pods

- Use no scheduler
- One instance on every node..
- WHen new node is configured, agent is automatically installed
- If pod is delete, new pod is automatically created..
- Ignore taints..



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

Find the DaemonSet pods that exist in your kubeadm cluster:

```
kubectl get pods -n kube-system -o wide
```

Delete a DaemonSet pod and see what happens:

```
kubectl delete pods [pod_name] -n kube-system
```

Give the node a label to signify it has SSD:

```
kubectl label node[node_name] disk=ssd
```

The YAML for a DaemonSet:

```
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - name: main
        image: linuxacademycontent/ssd-monitor
```

Create a DaemonSet from a YAML spec:

```
kubectl create -f ssd-monitor.yaml
```

Label another node to specify it has SSD:

```
kubectl label node chadcrowell2c.mylabserver.com disk=ssd
```

View the DaemonSet pods that have been deployed:

```
kubectl get pods -o wide
```

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

```
kubectl label node chadcrowell3c.mylabserver.com disk-
```

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

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

Pick the label to choose for your DaemonSet:

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

https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

Displaying Scheduler Events

There are multiple ways to view the events related to the scheduler. In this lesson, we'll look at ways in which you can troubleshoot any problems with your scheduler or just find out more information.
View the name of the scheduler pod:
`kubectl get pods -n kube-system`

Get the information about your scheduler pod events:
`kubectl describe pods [scheduler_pod_name] -n kube-system`

View the events in your default namespace:

```
kubectl get events
```

View the events in your `kube-system` namespace:

```
kubectl get events -n kube-system
```

Delete all the pods in your default namespace:

```
kubectl delete pods --all
```

Watch events as they are appearing in real time:

```
kubectl get events -w
```

View the logs from the scheduler pod:

```
kubectl logs [kube_scheduler_pod_name] -n kube-system
```

The location of a systemd service scheduler pod:

```
/var/log/kube-scheduler.log
```

https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/#verifying-that-the-pods-were-scheduled-using-the-desired-schedulers

LAB:

Taint one of the worker nodes to repel work.

1. Use the following command to taint the node:

```
kubectl taint node <node_name> node-type=prod:NoSchedule
```

Schedule a pod to the dev environment.

1. Use the following YAML to specify a pod that will be scheduled to the dev environment:

```
apiVersion: v1
kind: Pod
metadata:
 name: dev-pod
 labels:
   app: busybox
spec:
 containers:
 - name: dev
   image: busybox
   command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

2. Use the following command to create the pod:

```
kubectl create -f dev-pod.yaml
```

Allow a pod to be scheduled to the prod environment.

1. Use the following YAML to create a deployment and a pod that will tolerate the prod environment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: prod
spec:
 replicas: 1
 selector:
   matchLabels:
     app: prod
 template:
   metadata:
     labels:
       app: prod
   spec:
     containers:
     - args:
       - sleep
       - "3600"
       image: busybox
       name: main
     tolerations:
     - key: node-type
       operator: Equal
       value: prod
       effect: NoSchedule
```

2. Use the following command to create the pod:

```
kubectl create -f prod-deployment.yaml
```

Verify each pod has been scheduled and verify the toleration.

1. Use the following command to verify the pods have been scheduled:

```
kubectl get pods -o wide
```

Verify each pod has been scheduled and verify the toleration.

1. Use the following command to verify the pods have been scheduled:

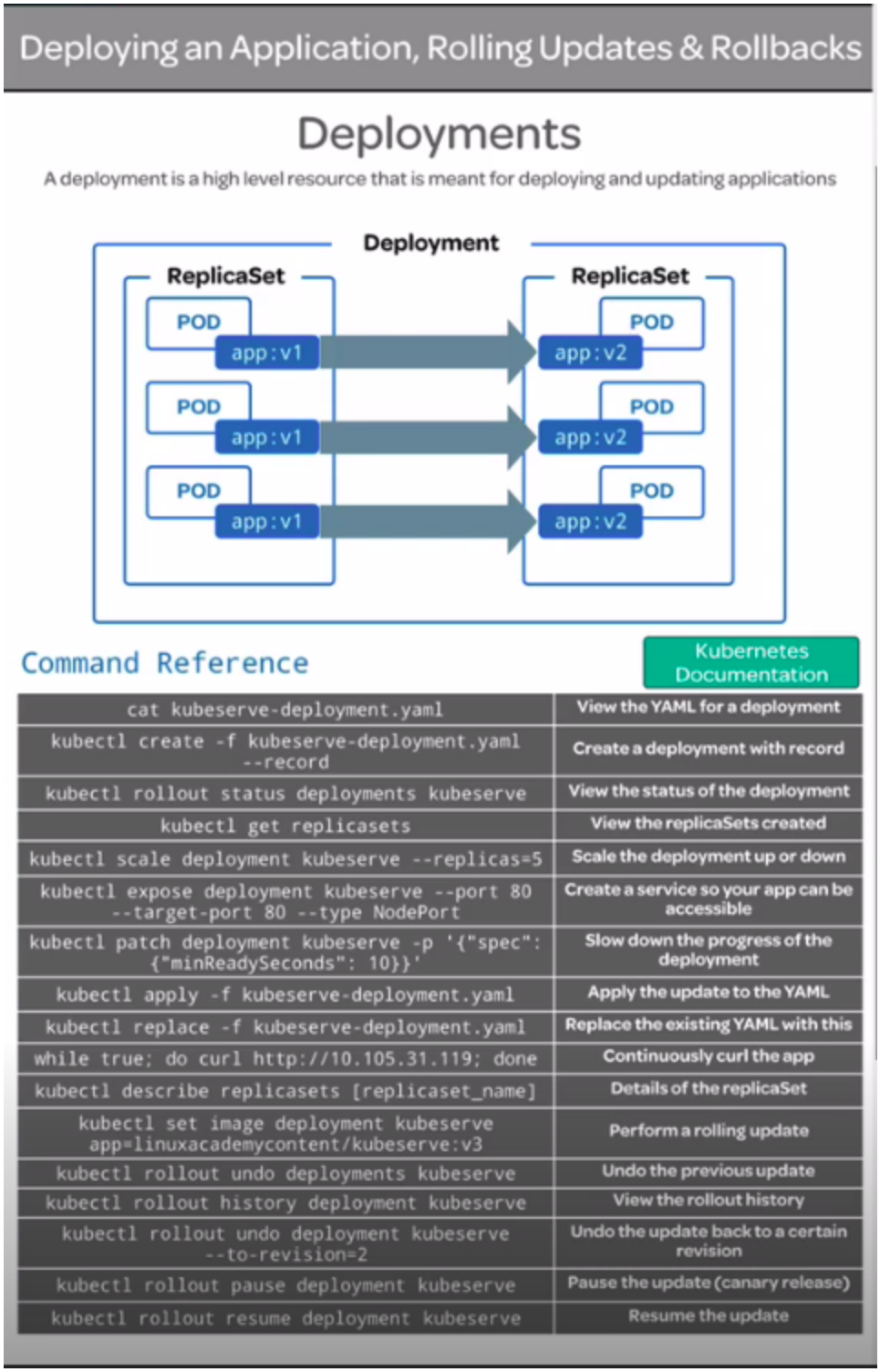
```
kubectl get pods -o wide
```

2. Verify the toleration of the production pod:

```
kubectl get pods <pod_name> -o yaml
```

Application lifecycle management

Deploying an Application, Rolling Updates, and Rollbacks



Deploying an Application, Rolling Updates & Rollbacks

We already know Kubernetes will run pods and deployments, but what happens when you need to update or change the version of your application running inside of the Kubernetes cluster? That's where rolling updates come in, allowing you to update the

app image with zero downtime. In this lesson, we'll go over a rolling update, how to roll back, and how to pause the update if things aren't going well.
The YAML for a deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubeserve
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubeserve
  template:
    metadata:
      name: kubeserve
      labels:
        app: kubeserve
    spec:
      containers:
      - image: linuxacademycontent/kubeserve:v1
        name: app
```

Create a deployment with a record (for rollbacks):

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

Check the status of the rollout:

```
kubectl rollout status deployments kubeserve
```

View the ReplicaSets in your cluster:

```
kubectl get replicasets
```

Scale up your deployment by adding more replicas:

```
kubectl scale deployment kubeserve --replicas=5
```

Expose the deployment and provide it a service:

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

Set the `minReadySeconds` attribute to your deployment:

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

Use `kubectl apply` to update a deployment:

```
kubectl apply -f kubeserve-deployment.yaml
```

Use `kubectl replace` to replace an existing deployment:

```
kubectl replace -f kubeserve-deployment.yaml
```

Run this curl look while the update happens:

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

Perform the rolling update:

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

Describe a certain ReplicaSet:

```
kubectl describe replicasets kubeserve-[hash]
```

Apply the rolling update to version 3 (buggy):

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

Undo the rollout and roll back to the previous version:

```
kubectl rollout undo deployments kubeserve
```

Look at the rollout history:

```
kubectl rollout history deployment kubeserve
```

Roll back to a certain revision:

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

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

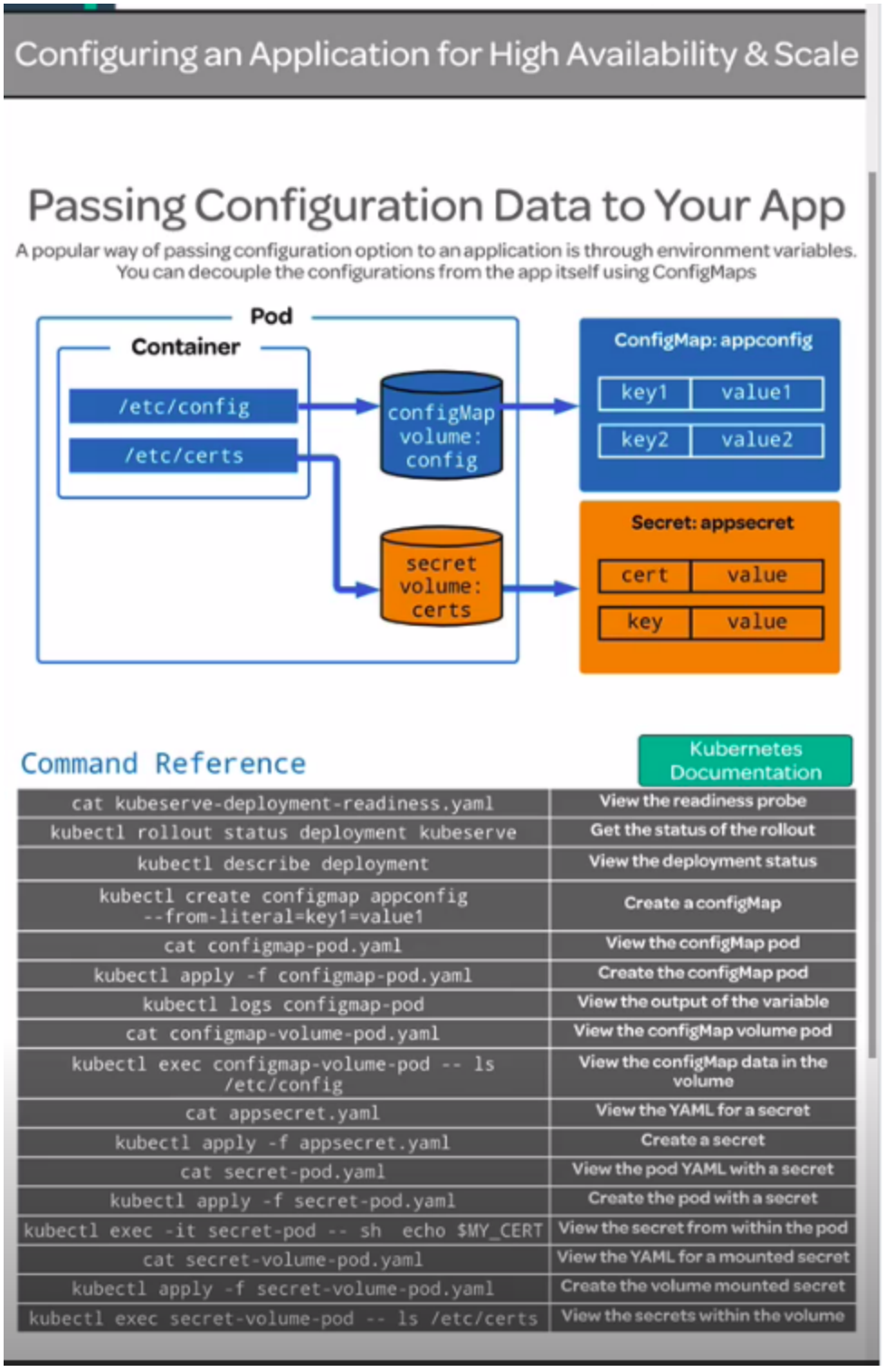
```
kubectl rollout pause deployment kubeserve
```

Resume the rollout after the rolling update looks good:

```
kubectl rollout resume deployment kubeserve
```

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-app/deploy-intro/
https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

Configuring an Application for High Availability and Scale



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

The YAML for a readiness probe:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubeserve
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubeserve
  minReadySeconds: 10
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      name: kubeserve
      labels:
        app: kubeserve
    spec:
      containers:
      - image: linuxacademycontent/kubeserve:v3
        name: app
        readinessProbe:
          periodSeconds: 1
          httpGet:
            path: /
            port: 80
```

Apply the readiness probe:

```
kubectl apply -f kubeserve-deployment-readiness.yaml
```

View the rollout status:

```
kubectl rollout status deployment kubeserve
```

Describe deployment:

```
kubectl describe deployment
```

Create a ConfigMap with two keys:

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

Get the YAML back out from the ConfigMap:

```
kubectl get configmap appconfig -o yaml
```

The YAML for the ConfigMap pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
  - name: app-container
    image: busybox:1.28
    command: ['sh', '-c', "echo $(MY_VAR) && sleep 3600"]
    env:
    - name: MY_VAR
      valueFrom:
        configMapKeyRef:
          name: appconfig
          key: key1
```

Create the pod that is passing the ConfigMap data:

```
kubectl apply -f configmap-pod.yaml
```

Get the logs from the pod displaying the value:

```
kubectl logs configmap-pod
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume-pod
spec:
  containers:
  - name: app-container
    image: busybox
    command: ['sh', '-c', "echo $(MY_VAR) && sleep 3600"]
    volumeMounts:
      - name: configmapvolume
        mountPath: /etc/config
  volumes:
    - name: configmapvolume
```

```
      configMap:
        name: appconfig
```

Create the ConfigMap volume pod:

```
kubectl apply -f configmap-volume-pod.yaml
```

Get the keys from the volume on the container:

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

Get the values from the volume on the pod:

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

The YAML for a secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: appsecret
stringData:
  cert: value
  key: value
```

Create the secret:

```
kubectl apply -f appsecret.yaml
```

The YAML for a pod that will use the secret:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
  - name: app-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
    env:
    - name: MY_CERT
      valueFrom:
        secretKeyRef:
          name: appsecret
          key: cert
```

Create the pod that has attached secret data:

```
kubectl apply -f secret-pod.yaml
```

Open a shell and echo the environment variable:

```
kubectl exec -it secret-pod -- sh
```

```
echo $MY_CERT
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-volume-pod
spec:
  containers:
  - name: app-container
    image: busybox
    command: ['sh', '-c', "echo $(MY_VAR) && sleep 3600"]
    volumeMounts:
      - name: secretvolume
        mountPath: /etc/certs
  volumes:
    - name: secretvolume
      secret:
        secretName: appsecret
```

Create the pod with volume attached with secrets:

```
kubectl apply -f secret-volume-pod.yaml
```

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

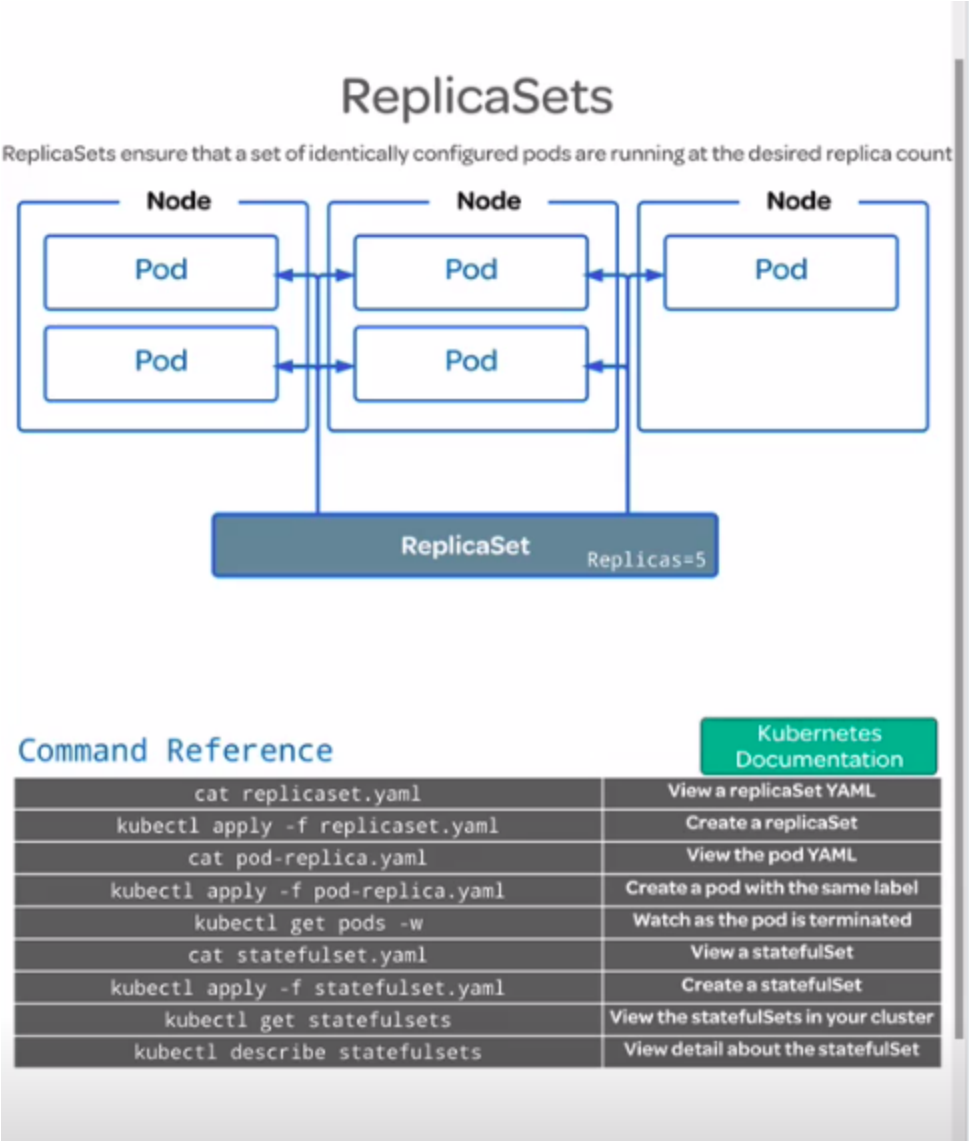
```
kubectl exec secret-volume-pod -- ls /etc/certs
```

Helpful Links

- Scaling Your Application
- Configure Pod ConfigMaps

- Secrets

Creating a Self-Healing Application



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

The YAML for a ReplicaSet:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myreplicaset
  labels:
    app: app
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: main
        image: linuxacademycontent/kubeserve
```

Create the ReplicaSet:

```
kubectl apply -f replicaset.yaml
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: pod1
  labels:
    tier: frontend
spec:
  containers:
  - name: main
    image: linuxacademycontent/kubeserve
```

Create the pod with the same label:

```
kubectl apply -f pod-replica.yaml
```

Watch the pod get terminated:

```
kubectl get pods -w
```

The YAML for a StatefulSet:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Create the StatefulSet:

```
kubectl apply -f statefulset.yaml
```

View all StatefulSets in the cluster:

```
kubectl get statefulsets
```

Describe the StatefulSets:

```
kubectl describe statefulsets
```

Helpful Links

- ReplicaSet
- StatefulSets

LAB:

Create and roll out version 1 of the application, and verify a successful deployment.

1. Use the following YAML named `kubeserve-deployment.yaml` to create your deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: kubeserve
spec:
 replicas: 3
 selector:
   matchLabels:
     app: kubeserve
   template:
     metadata:
       name: kubeserve
       labels:
         app: kubeserve
     spec:
       containers:
       - image: linuxacademycontent/kubeserve:v1
         name: app
```

2. Use the following command to create the deployment:

```
kubectl apply -f kubeserve-deployment.yaml --record
```

3. Use the following command to verify the deployment was successful:

```
kubectl rollout status deployments kubeserve
```

4. Use the following command to verify the app is at the correct version:

```
kubectl describe deployment kubeserve
```

Scale up the application to create high availability.

1. Use the following command to scale up your application to five replicas:

```
kubectl scale deployment kubeserve --replicas=5
```

2. Use the following command to verify the additional replicas have been created:

```
kubectl get pods
```

Create a service, so users can access the application.

1. Use the following command to create a service for your deployment:

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

2. Use the following command to verify the service is present and collect the cluster IP:

```
kubectl get services
```

3. Use the following command to verify the service is responding:

```
curl http://<ip-address-of-the-service>
```

Perform a rolling update to version 2 of the application, and verify its success.

1. Use this curl loop command to see the version change as you perform the rolling update:

```
 while true; do curl http://<ip-address-of-the-service>; done
```

2. Use this command to perform the update (while the curl loop is running):

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

3. Use this command to view the additional ReplicaSet created during the update:

```
 kubectl get replicasets
```

4. Use this command to verify all pods are up and running:

4. Use this command to verify all pods are up and running:

```
kubectl get pods
```

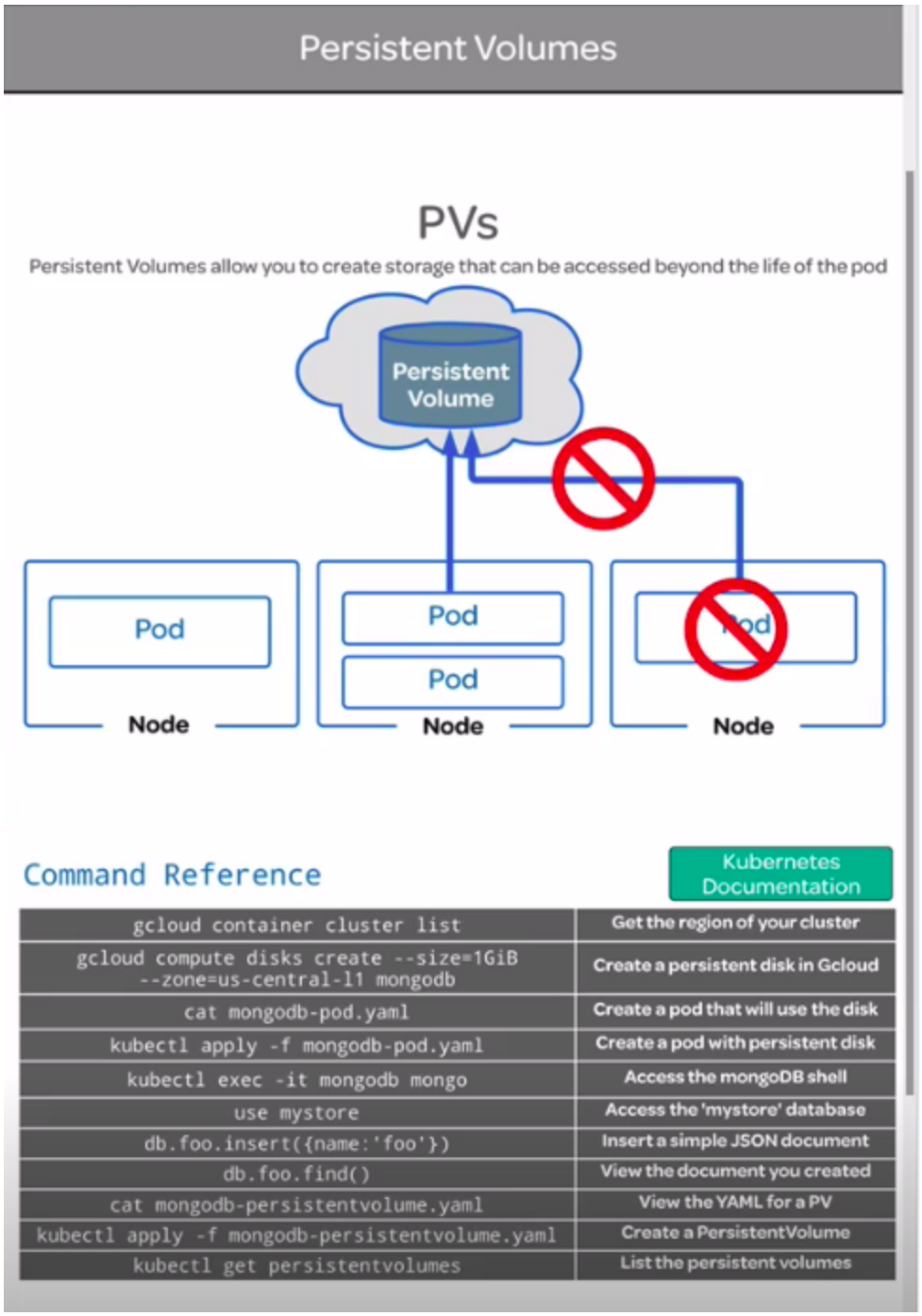
5. Use this command to view the rollout history:

```
kubectl rollout history deployment kubeserve
```

Storage (7%)

Managing Data in the Kubernetes Cluster

Persistent Volumes



In Kubernetes, pods are ephemeral. This creates a unique challenge with attaching storage directly to the filesystem of a container. Persistent Volumes are used to create an abstraction layer between the application and the underlying storage, making it

easier for the storage to follow the pods as they are deleted, moved, and created within your Kubernetes cluster.
In the Google Cloud Engine, find the region your cluster is in:

```
gcloud container clusters list
```

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

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

The YAML for a pod that will use persistent disk:

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  volumes:
  - name: mongodb-data
    gcePersistentDisk:
      pdName: mongodb
      fsType: ext4
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
```

Create the pod with disk attached and mounted:

```
kubectl apply -f mongodb-pod.yaml
```

See which node the pod landed on:

```
kubectl get pods -o wide
```

Connect to the `mongodb` shell:

```
kubectl exec -it mongodb mongo
```

Switch to the `mystore` database in the `mongodb` shell:

```
use mystore
```

Create a JSON document to insert into the database:
`db.foo.insert({name:'foo'})`

View the document you just created:
`db.foo.find()`

Exit from the `mongodb` shell:
`exit`

Delete the pod:
`kubectl delete pod mongodb`

Create a new pod with the same attached disk:
`kubectl apply -f mongodb-pod.yaml`

Check to see which node the pod landed on:
`kubectl get pods -o wide`

Drain the node (if the pod is on the same node as before):
`kubectl drain [node_name] --ignore-daemonsets`

Once the pod is on a different node, access the `mongodb` shell again:
`kubectl exec -it mongodb mongo`

Access the `mystore` database again:
`use mystore`

Find the document you created from before:
`db.foo.find()`

The YAML for a `PersistentVolume` object in Kubernetes:

```
apiVersion: v1
kind: PersistentVolume
metadata:
```

```
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

Create the Persistent Volume resource:

```
kubectl apply -f mongodb-persistentvolume.yaml
```

View our Persistent Volumes:

```
kubectl get persistentvolumes
```

```
LAB:

   -  Create a volume on google cloud
   -  Mount the volume to a mngodb pod
   -  Add some records..
   -  Delete pod
   -  Cretae another POD using same volume
   -  Records should be ratained..
```

Helpful Links:

- Persistent Volumes
- Configure Persistent Volume Storage

Volume Access Modes

Volume Access Modes

AccessModes

By specifying an access mode with your persistent volume(PV), you allow the volume to be mounted to one or many nodes, as well as read by one or many nodes

```
spec:
  capactiy:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  - ReadOnlyMany
```

- **RWO (ReadWriteOnce)** - Only 1 node can mount the volume for writing and reading
- **ROX (ReadOnlyMany)** - Multiple nodes can mount the volume for reading
- **RWX (ReadWriteMany)** - Multiple nodes can mount the volume for writing and reading

```
NAME         CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
mongodb-pv   1Gi        RWO,RWX       Retain          Available
```

Command Reference

Kubernetes Documentation

`cat mongodb-persistentvolume.yaml`	View the access mode in the YAML
`kubectl get pv`	View the existing PVs access mode

```
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
~
```

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

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

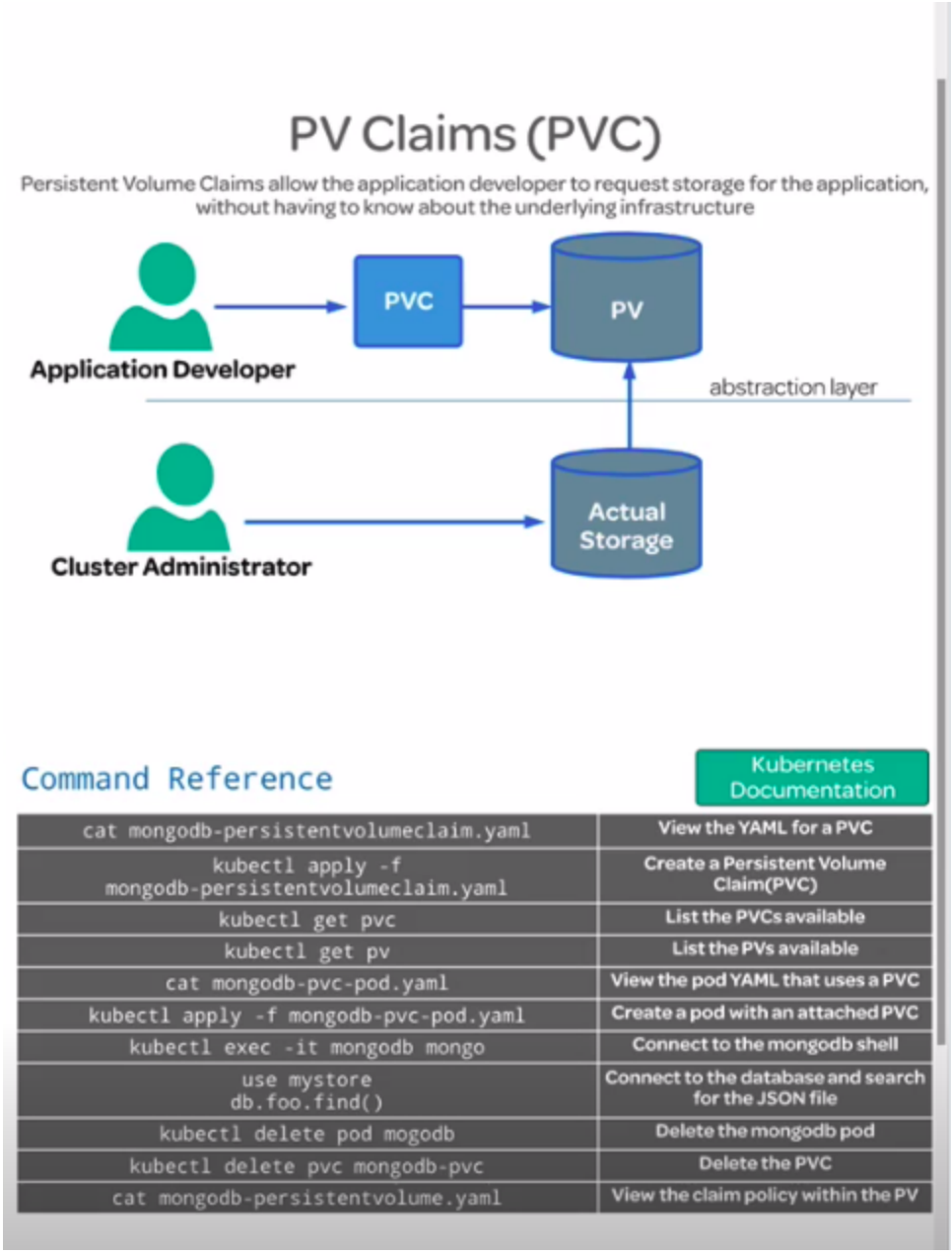
View the Persistent Volumes in your cluster:

```
kubectl get pv
```

Helpful Links:

- Access Modes

Persistent Volume Claims



Retain policy - Data from the disk/volume is not deleted once it is released..
Delete policy - Data is deleted once released..
Persistent Volume Claims (PVCs) are a way for an application developer to request storage for the application without having to know where the underlying storage is. The claim is then bound to the Persistent Volume (PV), and it will not be released until the PVC is deleted. In this lesson, we will go through creating a PVC and accessing storage within our persistent disk.
The YAML for a PVC:
apiVersion: v1
kind: PersistentVolumeClaim

```
metadata:
  name: mongodb-pvc
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
  - ReadWriteOnce
  storageClassName: ""
```

Create a PVC:
kubectl apply -f mongodb-pvc.yaml

View the PVC in the cluster:
kubectl get pvc

View the PV to ensure it’s bound:
kubectl get pv

The YAML for a pod that uses a PVC:

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
  volumes:
  - name: mongodb-data
    persistentVolumeClaim:
      claimName: mongodb-pvc
```

Create the pod with the attached storage:

kubectl apply -f mongo-pvc-pod.yaml

Access the `mogodb` shell:
kubectl exec -it mongodb mongo

Find the JSON document created in previous lessons:
db.foo.find()

Delete the `mongodb` pod:
kubectl delete pod mogodb

Delete the `mongodb-pvc` PVC:
kubectl delete pvc mongodb-pvc

Check the status of the PV:
kubectl get pv

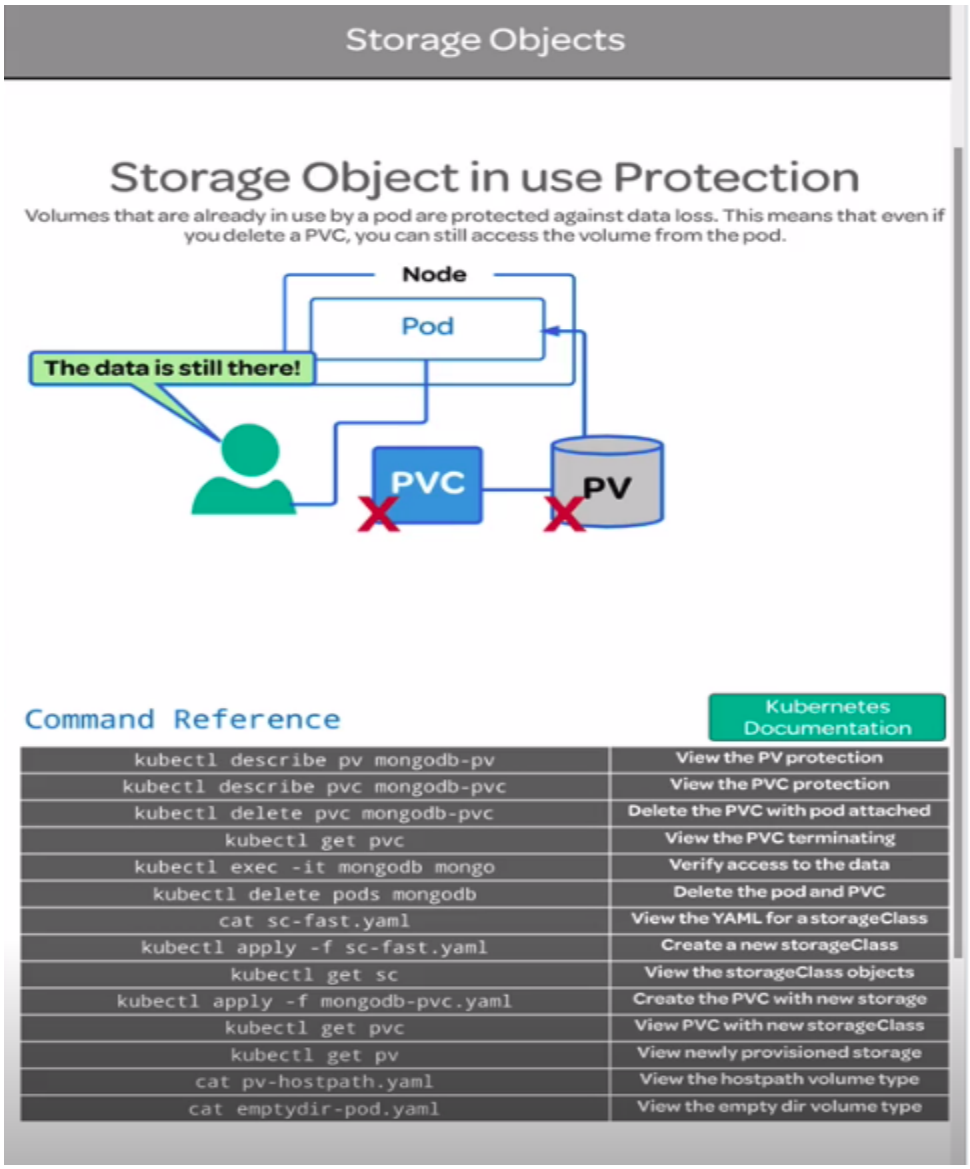
The YAML for the PV to show its reclaim policy:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

Helpful Links

- PersistentVolumeClaims
- Create a PersistentVolumeClaim

Storage Objects



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

See the PV protection on your volume:

```
kubectl describe pv mongodb-pv
```

See the PVC protection for your claim:

```
kubectl describe pvc mongodb-pvc
```

Delete the PVC:

```
kubectl delete pvc mongodb-pvc
```

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

```
kubectl get pvc
```

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

```
kubectl exec -it mongodb mongo
use mystore
db.foo.find()
```

Delete the pod, which finally deletes the PVC:

```
kubectl delete pods mongodb
```

Show that the PVC is deleted:

```
kubectl get pvc
```

YAML for a `StorageClass` object:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Create the StorageClass `type` "fast":

```
kubectl apply -f sc-fast.yaml
```

Change the PVC to include the new `StorageClass` object:

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: fast
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

Create the PVC with automatically provisioned storage:

```
kubectl apply -f mongodb-pvc.yaml
```

View the PVC with new StorageClass:

```
kubectl get pvc
```

View the newly provisioned storage:

```
kubectl get pv
```

The YAML for a `hostPath` PV:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath
spec:
  storageClassName: local-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

The YAML for a pod with an empty directory volume:

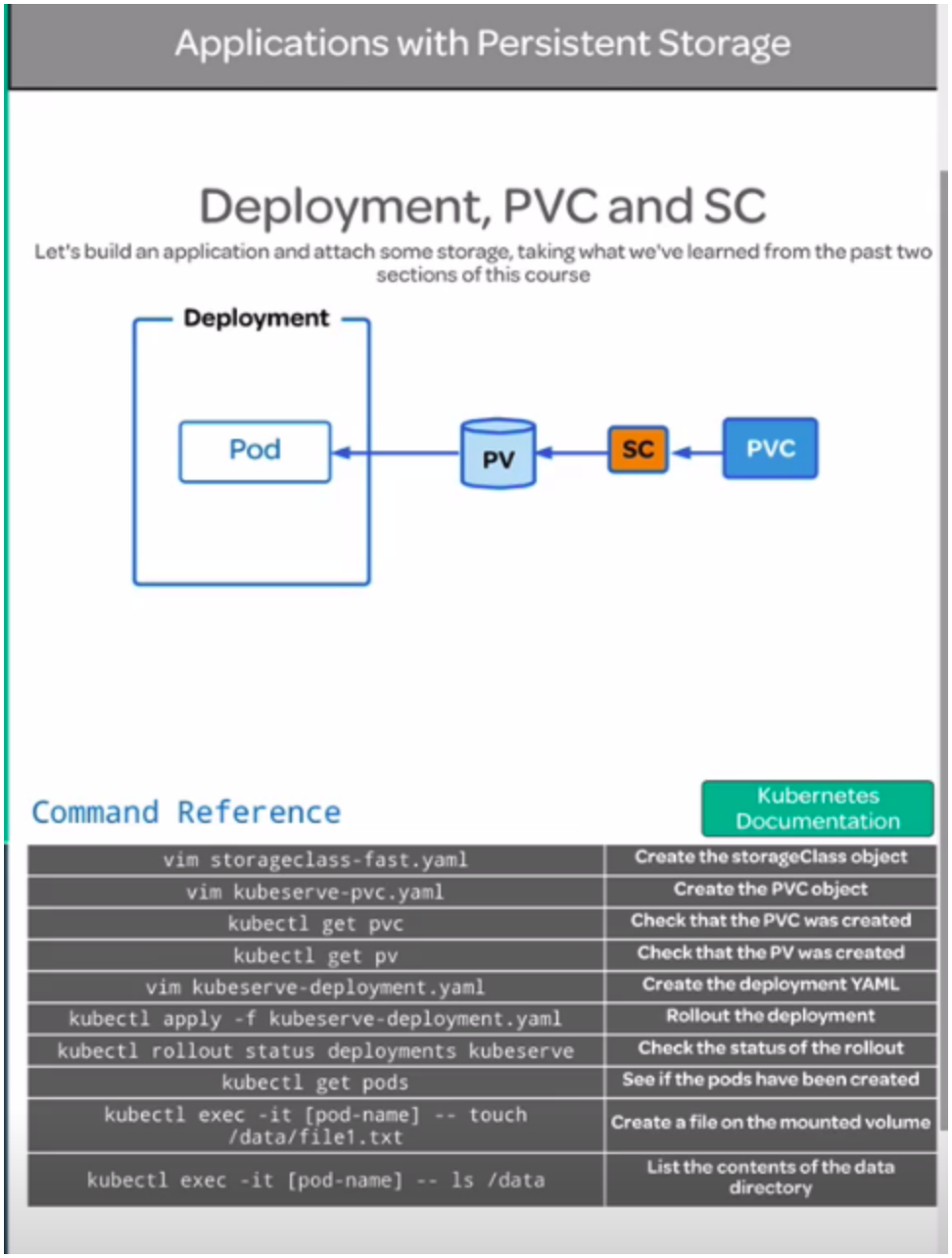
```
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-pod
spec:
  containers:
  - image: busybox
```

```
    name: busybox
    command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]
    volumeMounts:
    - mountPath: /tmp/storage
      name: vol
  volumes:
  - name: vol
    emptyDir: {}
```

Helpful Links:

- Object in Use Protection
- Volumes

Applications with Persistent Storage



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

The YAML for our `StorageClass` object:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
```

```
parameters:
  type: pd-ssd
```

The YAML for our PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: kubeserve-pvc
spec:
  storageClassName: fast
  resources:
    requests:
      storage: 100Mi
  accessModes:
    - ReadWriteOnce
```

Create our `StorageClass` object:

```
kubectl apply -f storageclass-fast.yaml
```

View the `StorageClass` objects in your cluster:

```
kubectl get sc
```

Create our PVC:

```
kubectl apply -f kubeserve-pvc.yaml
```

View the PVC created in our cluster:

```
kubectl get pvc
```

View our automatically provisioned PV:

```
kubectl get pv
```

The YAML for our deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kubeserve
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
      app: kubeserve
  template:
    metadata:
      name: kubeserve
      labels:
        app: kubeserve
    spec:
      containers:
      - env:
        - name: app
          value: "1"
        image: linuxacademycontent/kubeserve:v1
        name: app
        volumeMounts:
        - mountPath: /data
          name: volume-data
      volumes:
      - name: volume-data
        persistentVolumeClaim:
          claimName: kubeserve-pvc
```

Create our deployment and attach the storage to the pods:

```
kubectl apply -f kubeserve-deployment.yaml
```

Check the status of the rollout:

```
kubectl rollout status deployments kubeserve
```

Check the pods have been created:

```
kubectl get pods
```

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

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

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

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

```
LAB :
```

Create a PersistentVolume.

1. Use the following YAML spec for the PersistentVolume named `mongodb-pv.yaml`:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  storageClassName: local-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

2. Then, create the PersistentVolume:

```
kubectl apply -f mongodb-pv.yaml
```

Create a PersistentVolumeClaim.

1. Use the following YAML spec for the PersistentVolumeClaim named `mongodb-pvc.yaml`:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

2. Then, create the PersistentVolumeClaim:

```
kubectl apply -f mongodb-pvc.yaml
```

Create a pod from the mongodb image, with a mounted volume to mount path `/data/db`.

1. Use the following YAML spec for the pod named `mongodb-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
  volumes:
  - name: mongodb-data
    persistentVolumeClaim:
      claimName: mongodb-pvc
```

2. Then, create the pod:

```
kubectl apply -f mongodb-pod.yaml
```

3. Verify the pod was created:

```
kubectl get pods
```

Access the node and view the data within the volume.

1. Connect to the node:

```
ssh <node_hostname>
```

2. Switch to the `/mnt/data` directory:

```
cd /mnt/data
```

3. List the contents of the directory:

3. List the contents of the directory:

```
ls
```

Delete the pod and create a new pod with the same YAML spec.

1. Delete the pod:

```
kubectl delete pod mongodb
```

2. Create a new pod:

```
kubectl apply -f mongodb-pod.yaml
```

Verify the data still resides on the volume.

1. Log in to the node:

```
ssh <node_hostname>
```

2. Switch to the `/mnt/data` directory:

```
cd /mnt/data
```

3. List the contents of the directory:

```
ls
```

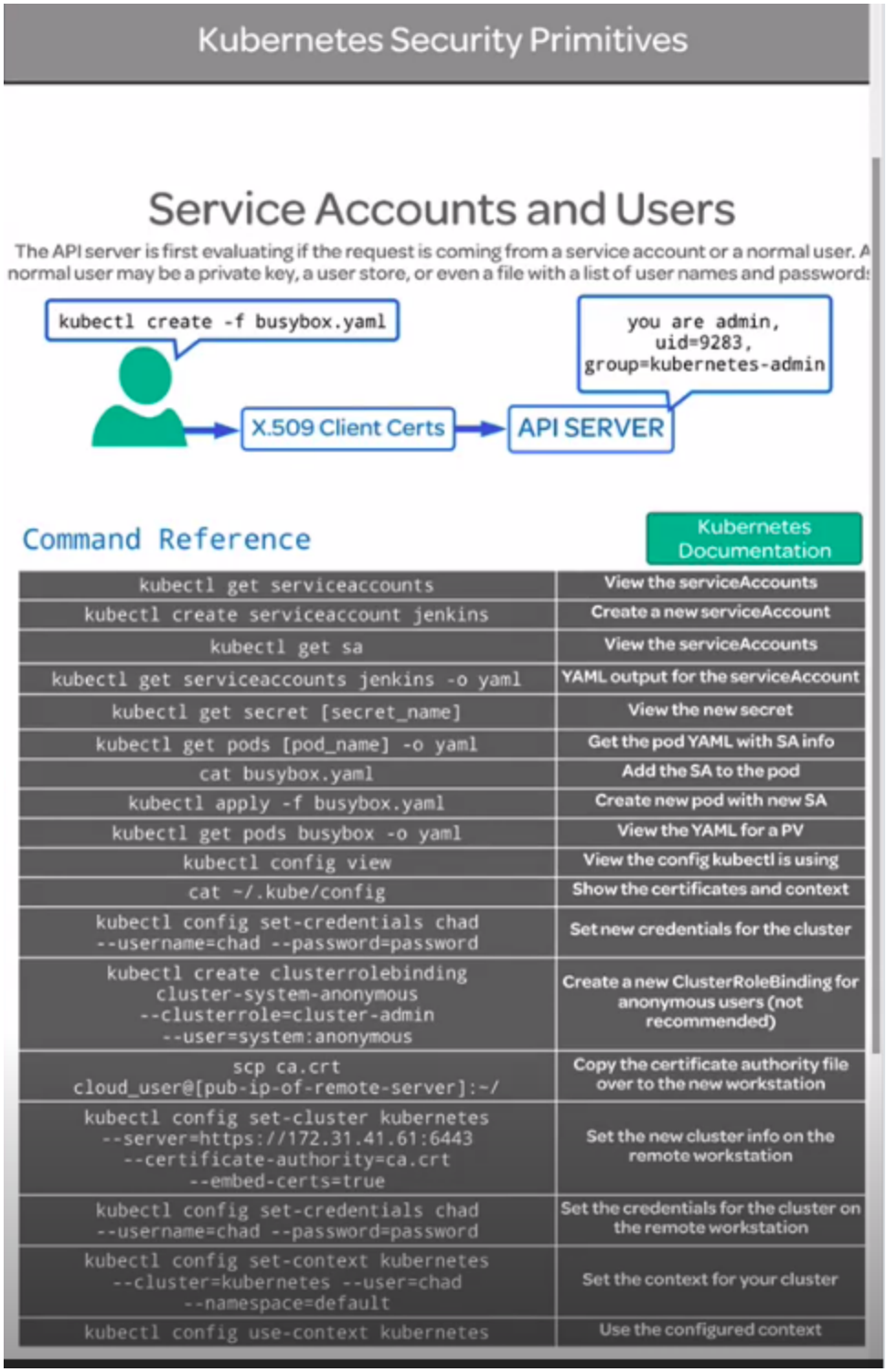
Security (12%)

Securing the Kubernetes Cluster

How to configure a new machine to connect to API server and run commands using kubectl

- Listen one more time..

Kubernetes Security Primitives



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

List the service accounts in your cluster:

```
kubectl get serviceaccounts
```

Create a new `jenkins` service account:

```
kubectl create serviceaccount jenkins
```

Use the abbreviated version of `serviceAccount`:

```
kubectl get sa
```

View the YAML for our service account:

```
kubectl get serviceaccounts jenkins -o yaml
```

View the secrets in your cluster:

```
kubectl get secret [secret_name]
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  serviceAccountName: jenkins
  containers:
  - image: busybox:1.28.4
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

Create a new pod with the service account:

```
kubectl apply -f busybox.yaml
```

View the cluster config that kubectl uses:
`kubectl config view`

View the config file:
`cat ~/.kube/config`

Set new credentials for your cluster:
`kubectl config set-credentials chad --username=chad --password=password`

Create a role binding for anonymous users (not recommended):
`kubectl create clusterrolebinding cluster-system-anonymous --clusterrole=cluster-admin --user=system:anonymous`

SCP the certificate authority to your workstation or server:
`scp ca.crt cloud_user@[pub-ip-of-remote-server]:~/`

Set the cluster address and authentication:
`kubectl config set-cluster kubernetes --server=https://172.31.41.61:6443 --certificate-authority=ca.crt --embed-certs=true`

Set the credentials for Chad:
`kubectl config set-credentials chad --username=chad --password=password`

Set the context for the cluster:
`kubectl config set-context kubernetes --cluster=kubernetes --user=chad --namespace=default`

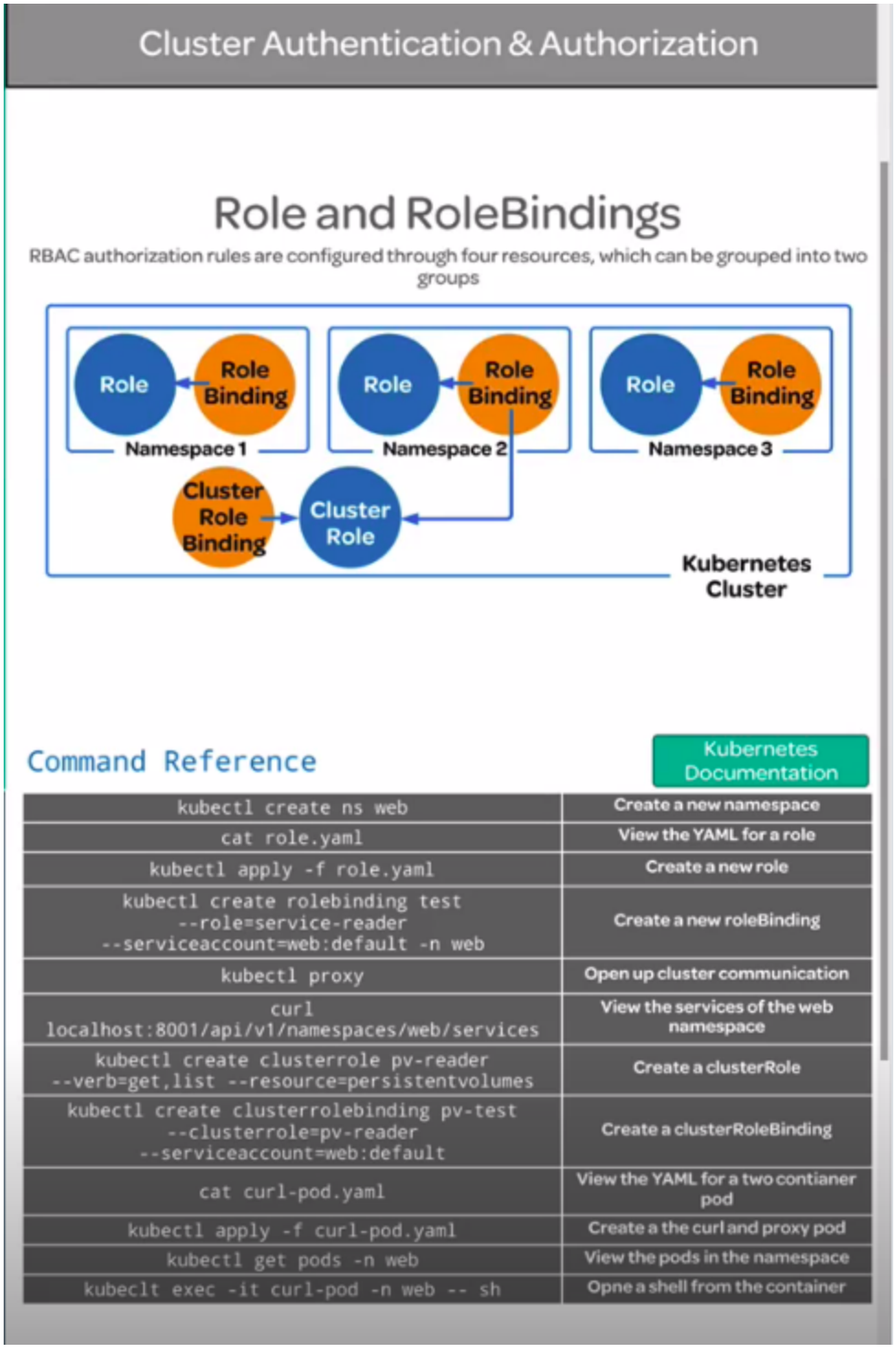
Use the context:
`kubectl config use-context kubernetes`

Run the same commands with kubectl:
`kubectl get nodes`

- Securing the Cluster

- Authentication
- Administer the Cluster via kubectl

Cluster Authentication and Authorization



`kubectl create ns web`	Create a new namespace
`cat role.yaml`	View the YAML for a role
`kubectl apply -f role.yaml`	Create a new role
`kubectl create rolebinding test --role=service-reader --serviceaccount=web:default -n web`	Create a new roleBinding
`kubectl proxy`	Open up cluster communication
`curl localhost:8001/api/v1/namespaces/web/services`	View the services of the web namespace
`kubectl create clusterrole pv-reader --verb=get,list --resource=persistentvolumes`	Create a clusterRole
`kubectl create clusterrolebinding pv-test --clusterrole=pv-reader --serviceaccount=web:default`	Create a clusterRoleBinding
`cat curl-pod.yaml`	View the YAML for a two contianer pod
`kubectl apply -f curl-pod.yaml`	Create a the curl and proxy pod
`kubectl get pods -n web`	View the pods in the namespace
`kubeclt exec -it curl-pod -n web -- sh`	Opne a shell from the container

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

Create a new namespace:

```
kubectl create ns web
```

The YAML for a service role:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: web
  name: service-reader
rules:
- apiGroups: [""]
  verbs: ["get", "list"]
  resources: ["services"]
```

Create a new role from that YAML file:

```
kubectl apply -f role.yaml
```

Create a `RoleBinding`:

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

Run a proxy for inter-cluster communications:

```
kubectl proxy
```

Try to access the services in the `web` namespace:

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

Create a `ClusterRole` to access `PersistentVolumes`:

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

Create a `ClusterRoleBinding` for the cluster role:

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

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

```
apiVersion: v1
kind: Pod
metadata:
  name: curlpod
  namespace: web
spec:
  containers:
  - image: tutum/curl
    command: ["sleep", "9999999"]
    name: main
  - image: linuxacademycontent/kubectl-proxy
    name: proxy
  restartPolicy: Always
```

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

```
kubectl apply -f curl-pod.yaml
```

Get the pods in the `web` namespace:

```
kubectl get pods -n web
```

Open a shell to the container:

```
kubectl exec -it curlpod -n web -- sh
```

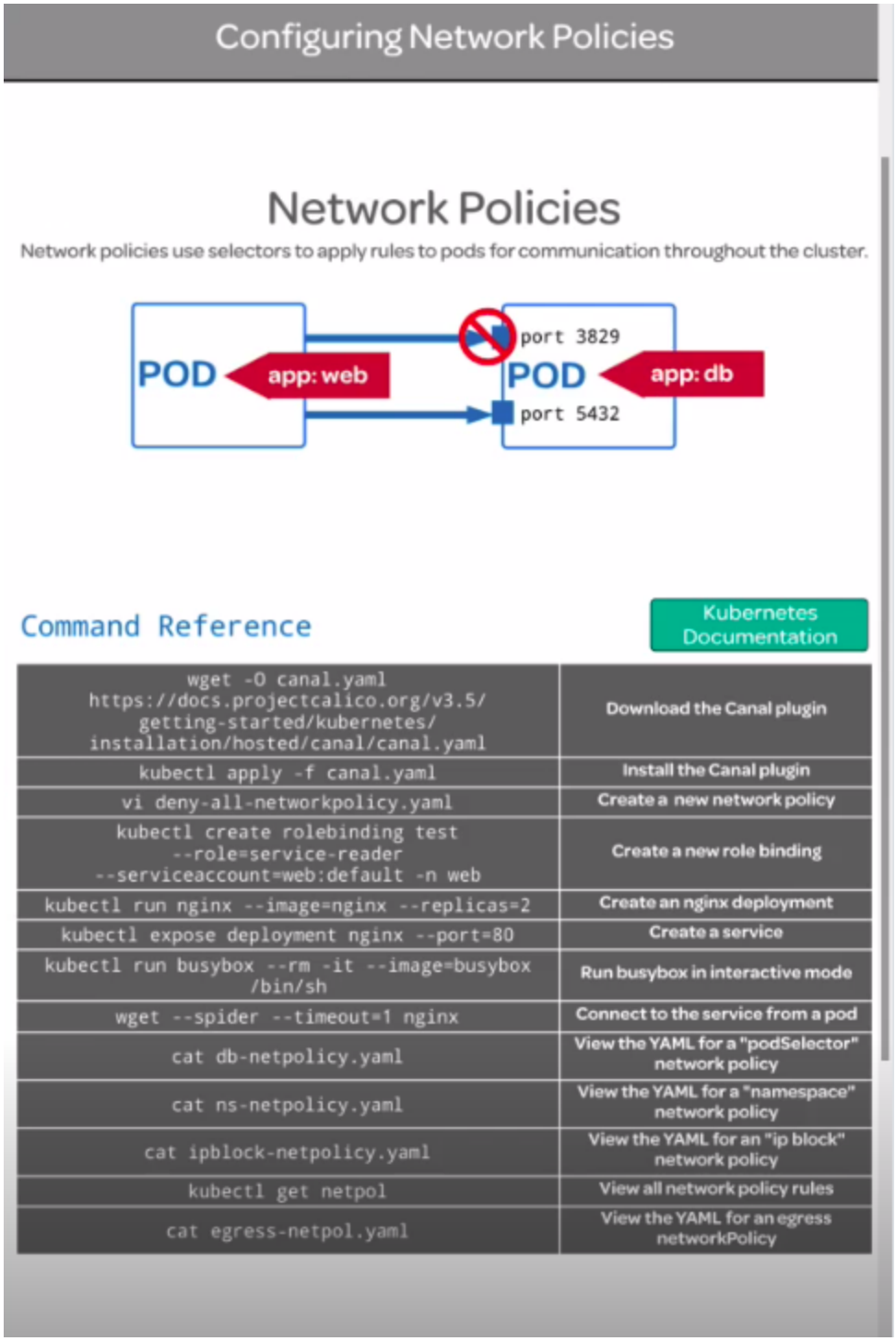
Access `PersistentVolumes` (cluster-level) from the pod:

```
curl localhost:8001/api/v1/persistentvolumes
```

Helpful Links:

- Authorization
- RBAC
- RoleBinding and ClusterRoleBinding

Configuring Network Policies



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

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

Apply the canal plugin:

```
kubectl apply -f canal.yaml
```

The YAML for a `deny-all` NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

Run a deployment to test the NetworkPolicy:

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

Create a service for the deployment:

```
kubectl expose deployment nginx --port=80
```

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

```
kubectl run busybox --rm -it --image=busybox /bin/sh
#wget --spider --timeout=1 nginx
```

The YAML for a pod selector NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-netpolicy
spec:
  podSelector:
```

```
    matchLabels:
      app: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: web
    ports:
    - port: 5432
```

Label a pod to get the NetworkPolicy:

```
kubectl label pods [pod_name] app=db
```

The YAML for a namespace NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ns-netpolicy
spec:
  podSelector:
    matchLabels:
      app: db
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          tenant: web
    ports:
    - port: 5432
```

The YAML for an IP block NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ipblock-netpolicy
spec:
  podSelector:
    matchLabels:
      app: db
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.1.0/24
```

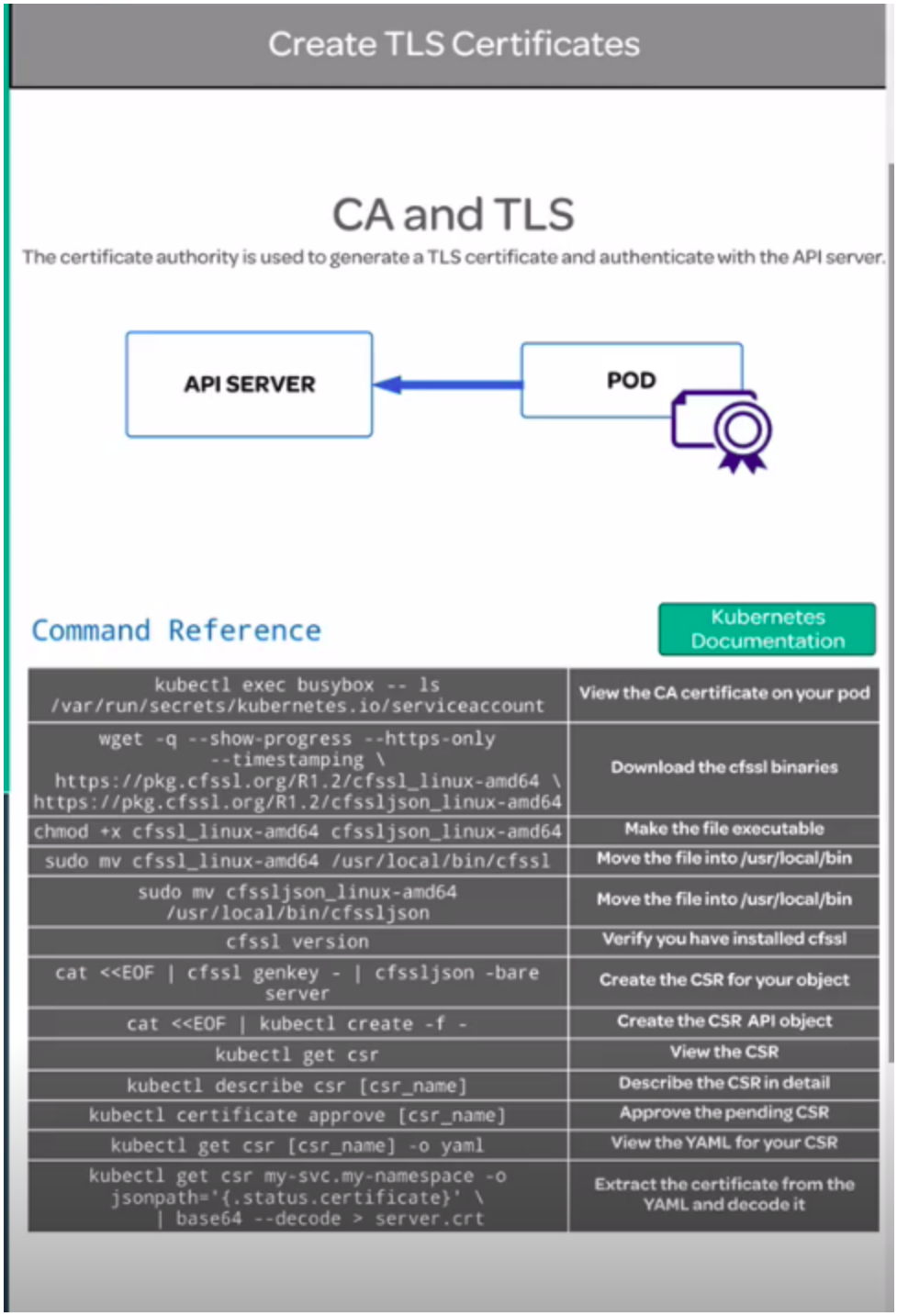
The YAML for an egress NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-netpol
spec:
  podSelector:
    matchLabels:
      app: web
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: db
    ports:
    - port: 5432
```

Helpful Links:

- Network Policies
- Declare Network Policies
- Default Network Policies

Creating TLS Certificates



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

Find the CA certificate on a pod in your cluster:

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

Download the binaries for the cfssl tool:

```
wget -q --show-progress --https-only --timestamping \
  https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 \
  https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64
```

Make the binary files executable:

```
chmod +x cfssl_linux-amd64 cfssljson_linux-amd64
```

Move the files into your `bin` directory:

```
sudo mv cfssl_linux-amd64 /usr/local/bin/cfssl
```

```
sudo mv cfssljson_linux-amd64 /usr/local/bin/cfssljson
```

Check to see if you have cfssl installed correctly:

```
cfssl version
```

Create a CSR file:

```
cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "172.168.0.24",
    "10.0.34.2"
  ],
  "CN": "my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  }
}
EOF
```

Create a `CertificateSigningRequest` API object:

```
cat <<EOF | kubectl create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: pod-csr.web
spec:
  groups:
  - system:authenticated
  request: $(cat server.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

View the CSRs in the cluster:

```
kubectl get csr
```

View additional details about the CSR:

```
kubectl describe csr pod-csr.web
```

Approve the CSR:

```
kubectl certificate approve pod-csr.web
```

View the certificate within your CSR:

```
kubectl get csr pod-csr.web -o yaml
```

Extract and decode your certificate to use in a file:

```
kubectl get csr pod-csr.web -o jsonpath='{.status.certificate}' \
    | base64 --decode > server.crt
```

Helpful Links

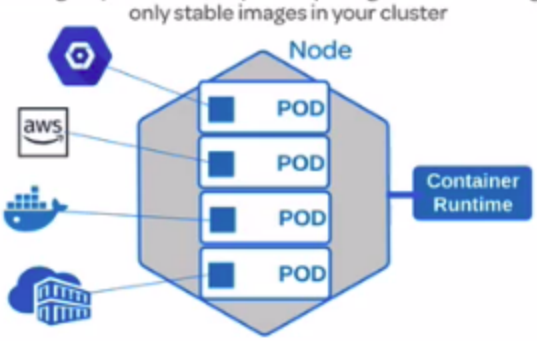
- Managing TLS in the Cluster
- Bootstrapping TLS for Your kubelets
- How kubeadm Manages Certificates

Secure Images

Secure Images

Private Registry

Pulling from a private registry ensures that you are pulling the correct image. It forces you to use only stable images in your cluster



Command Reference

Kubernetes Documentation

`cat busybox.yaml`	View how an image is referenced
`sudo vim /home/cloud_user/.docker/config.json`	View the keys for all registries
`sudo docker login`	Login to docker registry
`sudo docker images`	View the images docker pulled
`sudo docker pull busybox:1.28.4`	Pull another image
`docker login -u podofminerva -p 'otj701c9OucKZOCx5qrRblofcNRf3W+e' podofminerva.azurecr.io`	Login using a different registry
`sudo docker tag busybox:1.28.4 podofminerva.azurecr.io/busybox:latest`	Tag an image to properly push it to your private registry
`docker push podofminerva.azurecr.io/busybox:latest`	Push an image to a private registry
`kubectl create secret docker-registry acr --docker-server=https://podofminerva.azurecr.io --docker-username=podofminerva -docker-password='otj701c9OucKZOCx5qrRblofcNRf3W+e' --docker-email=user@example.com`	Create a new secret with your new private registry keys
`kubectl patch sa default -p '{"imagePullSecrets": [{"name": "acr"}]}'`	Modify the default serviceAccount
`vi acr-pod.yaml`	Build the YAML for a pod using private registry image
`kubectl apply -f acr-pod.yaml`	Create the pod from your private registry
`kubectl get pods`	View the pod running in your cluster

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

View where your Docker credentials are stored:

```
sudo vim /home/cloud_user/.docker/config.json
```

Log in to the Docker Hub:

```
sudo docker login
```

View the images currently on your server:

```
sudo docker images
```

Pull a new image to use with a Kubernetes pod:

```
sudo docker pull busybox:1.28.4
```

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

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

View your stored credentials:

```
sudo vim /home/cloud_user/.docker/config.json
```

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

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

Push the image to your private registry:

```
docker push podofminerva.azurecr.io/busybox:latest
```

Create a new `docker-registry` secret:

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

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

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

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

```
apiVersion: v1
kind: Pod
metadata:
  name: acr-pod
  labels:
    app: busybox
spec:
  containers:
    - name: busybox
      image: podofminerva.azurecr.io/busybox:latest
      command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
      imagePullPolicy: Always
```

Create the pod from the private image:

```
kubectl apply -f acr-pod.yaml
```

View the running pod:

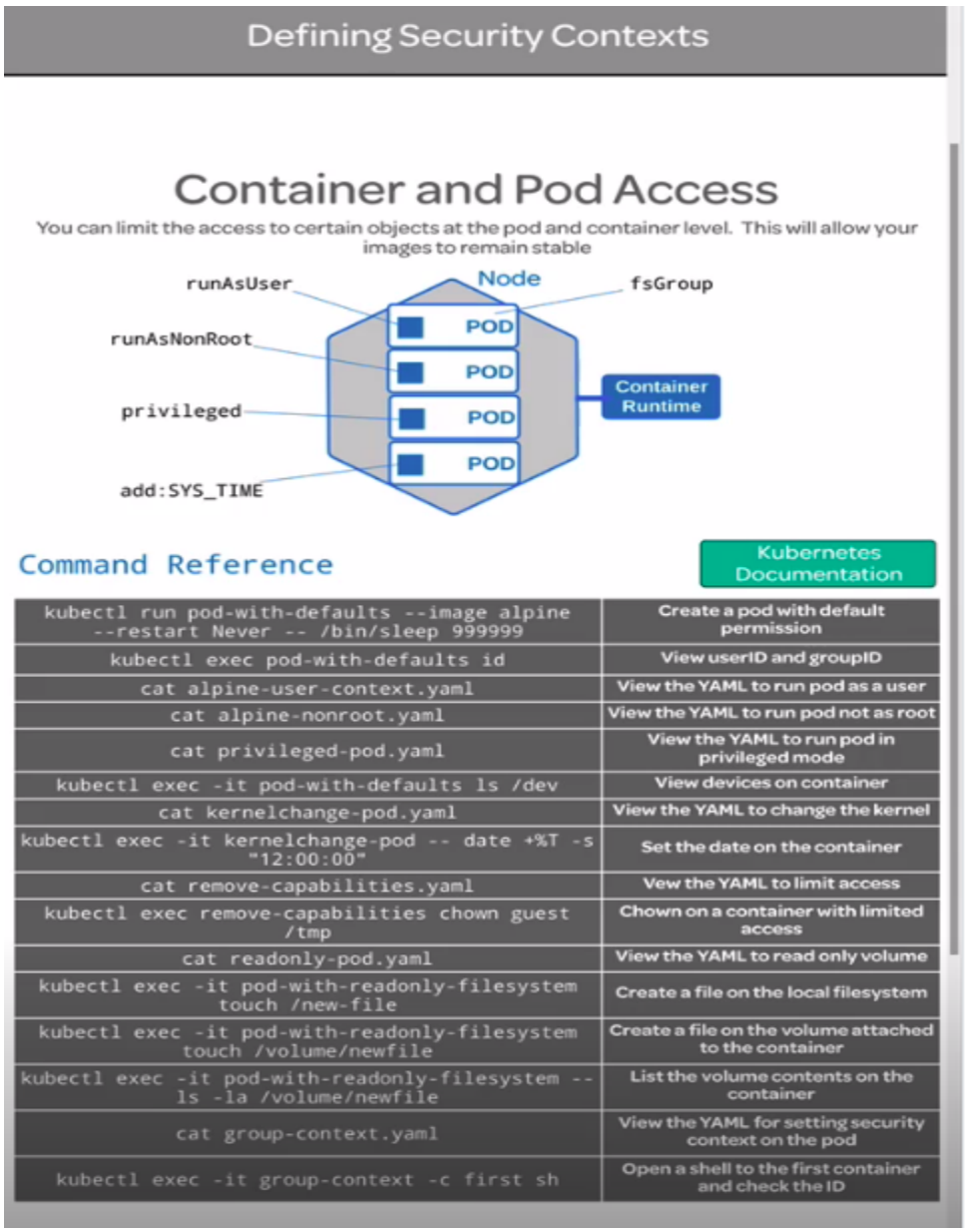
```
kubectl get pods
```

Helpful Links

- Images
- Pull Images from a Private Registry
- Configure Service Accounts
- Add ImagePullSecrets
- 11 Ways (Not) to Get Hacked

Defining Security Contexts

By default container runs as root user.. Not a best practice..



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

Run an alpine container with default security:

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

Check the ID on the container:

```
kubectl exec pod-with-defaults id
```

The YAML for a container that runs as a user:

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine-user-context
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 405
```

Create a pod that runs the container as user:

```
kubectl apply -f alpine-user-context.yaml
```

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

```
kubectl exec alpine-user-context id
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine-nonroot
Spec:la -ltr
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsNonRoot: true
```

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

```
kubectl apply -f alpine-nonroot.yaml
```

View more information about the pod error:

```
kubectl describe pod alpine-nonroot
```

```
POD fails.. As image is running as root and command specified to run as non root user.
```

The YAML for a privileged container pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: privileged-pod
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      privileged: true
```

Create the privileged container pod:

```
kubectl apply -f privileged-pod.yaml
```

View the devices on the default container:

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

View the devices on the privileged pod container:

```
kubectl exec -it privileged-pod ls /dev
```

Try to change the time on a default container pod:

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

```
Not able to set date..

date: can't set date: Operation not permitted

12:00:00
```

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

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: kernelchange-pod
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      capabilities:
        add:
        - SYS_TIME
```

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

```
kubectl run -f kernelchange-pod.yaml
```

Change the time on a container:

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

View the date on the container:

```
kubectl exec -it kernelchange-pod -- date
```

The YAML for a container that removes capabilities:

```
apiVersion: v1
kind: Pod
metadata:
  name: remove-capabilities
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      capabilities:
        drop:
        - CHOWN
```

Create a pod that's container has capabilities removed:

```
kubectl apply -f remove-capabilities.yaml
```

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

```
kubectl exec remove-capabilities chown guest /tmp
```

```
chown: /tmp: Operation not permitted

command terminated with exit code 1
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: readonly-pod
spec:
  containers:
  - name: main
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      readOnlyRootFilesystem: true
    volumeMounts:
    - name: my-volume
      mountPath: /volume
      readOnly: false
  volumes:
  - name: my-volume
    emptyDir:
```

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

```
kubectl apply -f readonly-pod.yaml
```

Try to write to the container filesystem:

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

```
touch: /new-file: Read-only file system

command terminated with exit code 1
```

Create a file on the volume mounted to the container:

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

View the file on the volume that's mounted:

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

```
-rw-r--r--   1 root      root              0 Aug  9 17:34 /volume/newfile
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: group-context
spec:
  securityContext:
    fsGroup: 555
    supplementalGroups: [666, 777]
  containers:
  - name: first
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 1111
    volumeMounts:
    - name: shared-volume
      mountPath: /volume
      readOnly: false
  - name: second
    image: alpine
    command: ["/bin/sleep", "999999"]
    securityContext:
      runAsUser: 2222
    volumeMounts:
    - name: shared-volume
      mountPath: /volume
      readOnly: false
  volumes:
  - name: shared-volume
    emptyDir:
```

Create a pod with two containers and different group permissions:

```
kubectl apply -f group-context.yaml
```

Open a shell to the first container on that pod:

```
kubectl exec -it group-context -c first sh
```

```
imsrv01@imsrv01-Lenovo-ideapad-700-15ISK:~/CKA/cka/security$ kubectl exec -it group-context -c first sh
/ $ id
uid=1111 gid=0(root) groups=555,666,777
/ $ cd /volume
/volume $ ls -ltr
total 0
/volume $ touch file
/volume $ ls -ltr
total 0
-rw-r--r--  1 1111      555                0 Aug  9 17:38 file
```

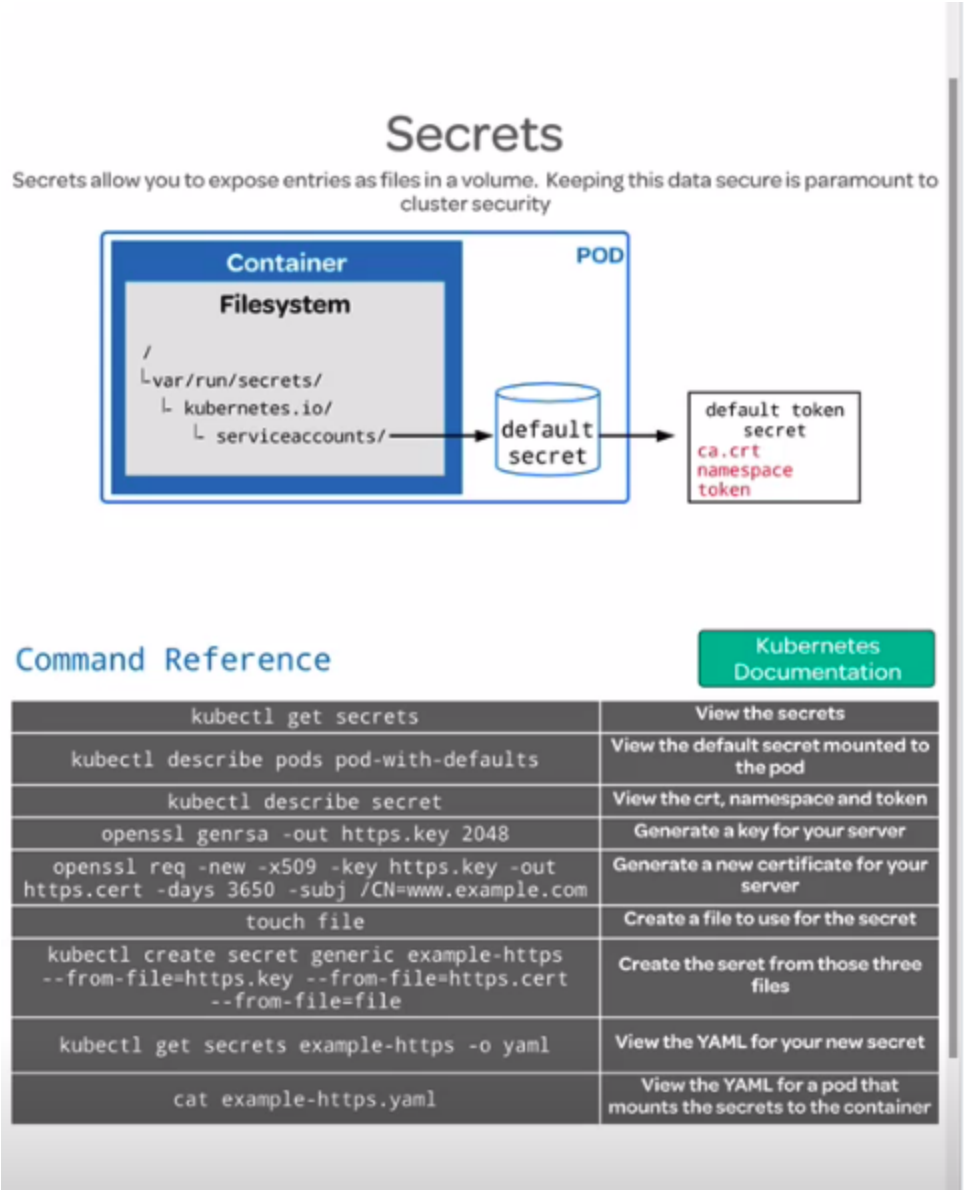
```
/volume $ imsrv01@imsrv01-Lenovo-ideapad-700-15ISK:~/CKA/cka/security$ kubectl exec -it group-context -c second sh
/ $ id
uid=2222 gid=0(root) groups=555,666,777
/ $ cd /volume
/volume $ touch file
touch: file: Permission denied
/volume $ touch file1
/volume $ ls -lrr
```

```
total 0
-rw-r--r--  1 2222      555               0 Aug  9 17:42 file1
-rw-r--r--  1 1111      555               0 Aug  9 17:38 file
```

Helpful Links

- Security Context

Securing Persistent Key Value Store



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

View the default secret mounted to each pod:
kubectl describe pods pod-with-defaults

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

Generate a key for your https server:
openssl genrsa -out https.key 2048

Generate a certificate for the https server:
openssl req -new -x509 -key https.key -out https.cert -days 3650 -subj /CN=www.example.com

Create an empty file to create the secret:
touch file

Create a secret from your key, cert, and file:
kubectl create secret generic example-https --from-file=https.key --from-file=https.cert
--from-file=file

View the YAML from your new secret:
kubectl get secrets example-https -o yaml

Create the configMap that will mount to your pod:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
data:
  my-nginx-config.conf: |
    server {
        listen              80;
        listen              443 ssl;
        server_name         www.example.com;
        ssl_certificate     certs/https.cert;
        ssl_certificate_key certs/https.key;
        ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers         HIGH:!aNULL:!MD5;

        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }
```

```
  }
 sleep-interval: |
  25
```

The YAML for a pod using the new secret:

```
apiVersion: v1
kind: Pod
metadata:
 name: example-https
spec:
 containers:
 - image: linuxacademycontent/fortune
   name: html-web
   env:
   - name: INTERVAL
     valueFrom:
       configMapKeyRef:
         name: config
         key: sleep-interval
   volumeMounts:
   - name: html
     mountPath: /var/htdocs
 - image: nginx:alpine
   name: web-server
   volumeMounts:
   - name: html
     mountPath: /usr/share/nginx/html
     readOnly: true
   - name: config
     mountPath: /etc/nginx/conf.d
     readOnly: true
   - name: certs
     mountPath: /etc/nginx/certs/
     readOnly: true
   ports:
   - containerPort: 80
   - containerPort: 443
 volumes:
 - name: html
   emptyDir: {}
 - name: config
```

```
    configMap:
      name: config
      items:
      - key: my-nginx-config.conf
        path: https.conf
  - name: certs
    secret:
      secretName: example-https
```

Describe the nginx conf via ConfigMap:
kubectl describe configmap

View the cert mounted on the container:
kubectl exec example-https -c web-server -- mount | grep certs
vi

Use port forwarding on the pod to server traffic from 443:
kubectl port-forward example-https 8443:443 &

Curl the web server to get a response:
curl https://localhost:8443 -k

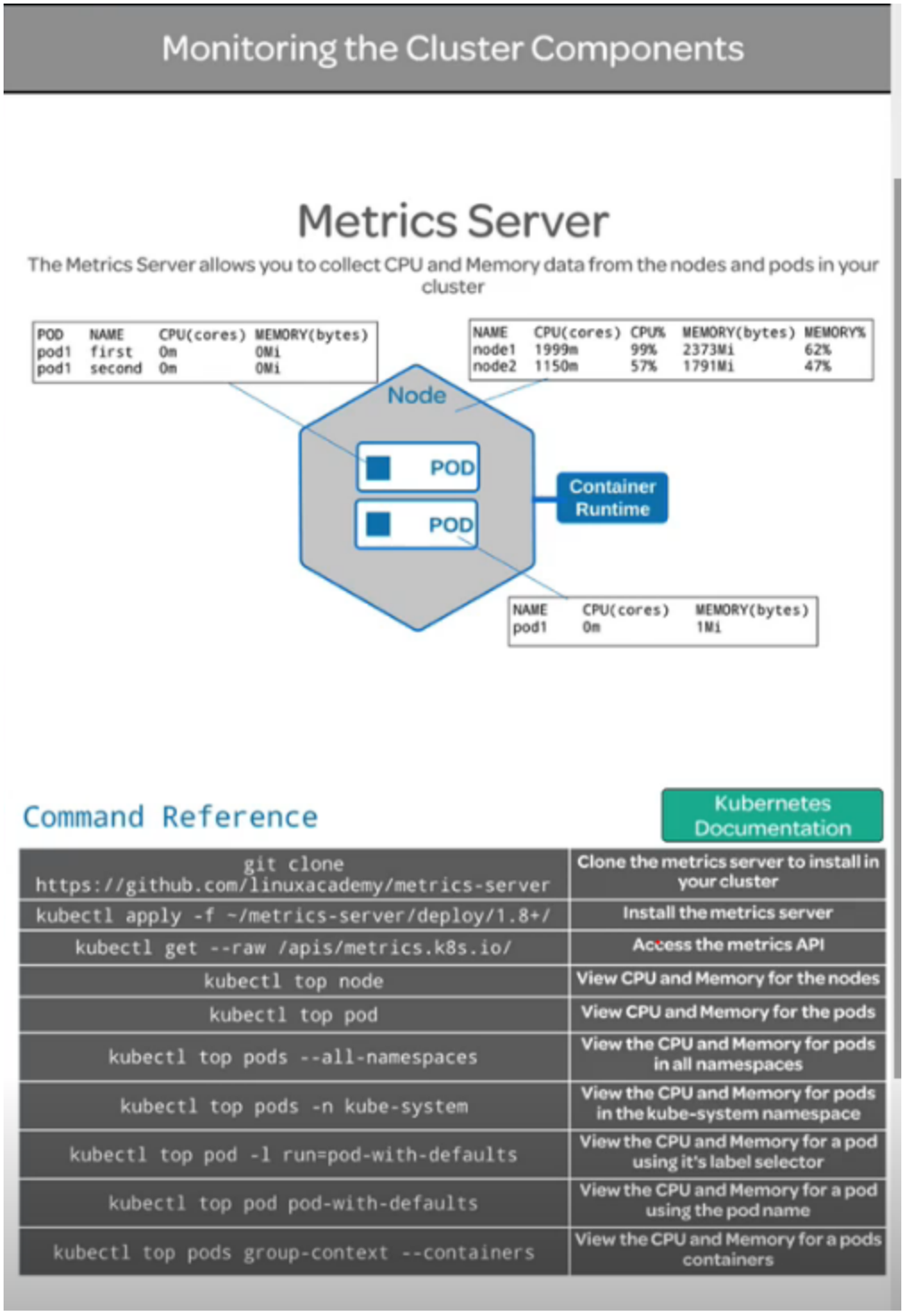
Helpful Links

- Secrets

Logging and Monitoring (5%)

Monitoring Cluster Components

Monitoring the Cluster Components



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

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

Install the metrics server in your cluster:

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

Get a response from the metrics server API:

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

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

```
kubectl top node
```

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

```
kubectl top pods
```

Get the CPU and memory of pods in all namespaces:

```
kubectl top pods --all-namespaces
```

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

```
kubectl top pods -n kube-system
```

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

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

Get the CPU and memory of a specific pod:

```
kubectl top pod pod-with-defaults
```

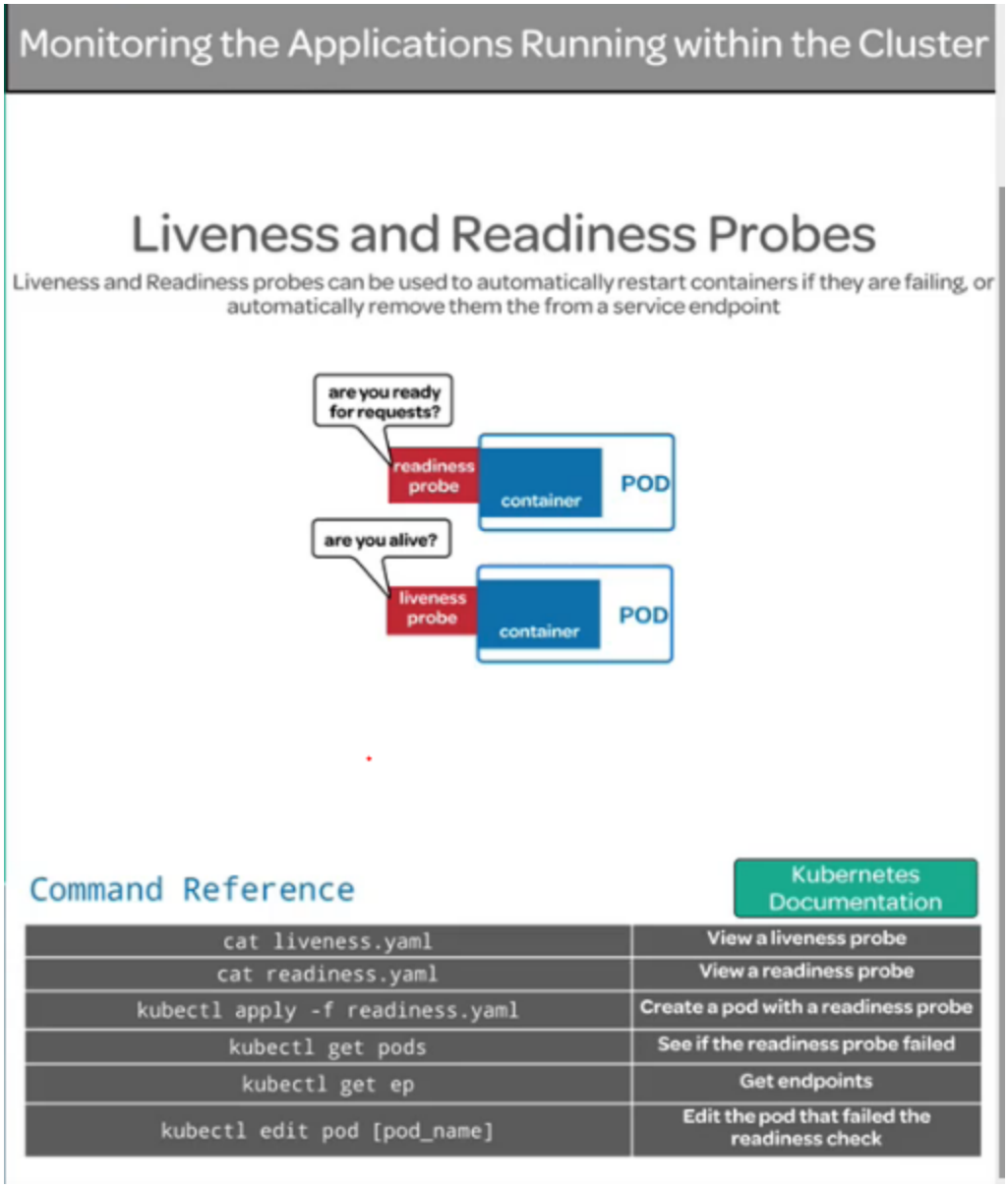
Get the CPU and memory of the containers inside the pod:

```
kubectl top pods group-context --containers
```

Helpful Links

- Monitor Node Health
- Resource Usage Monitoring

Monitoring the Applications Running within a Cluster



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

```
apiVersion: v1

kind: Pod

metadata:

  name: liveness
```

```
spec:
  containers:
  - image: linuxacademycontent/kubeserve
    name: kubeserve
    livenessProbe:
      httpGet:
        path: /
        port: 80
```

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

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nginx
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    readinessProbe:
```

```
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
---
apiVersion: v1
kind: Pod
metadata:
  name: nginxpd
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:191
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
```

Create the service and two pods with readiness probes:

```
kubectl apply -f readiness.yaml
```

Check if the readiness check passed or failed:

```
kubectl get pods
```

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

```
kubectl get ep
```

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

```
kubectl edit pod [pod_name]
```

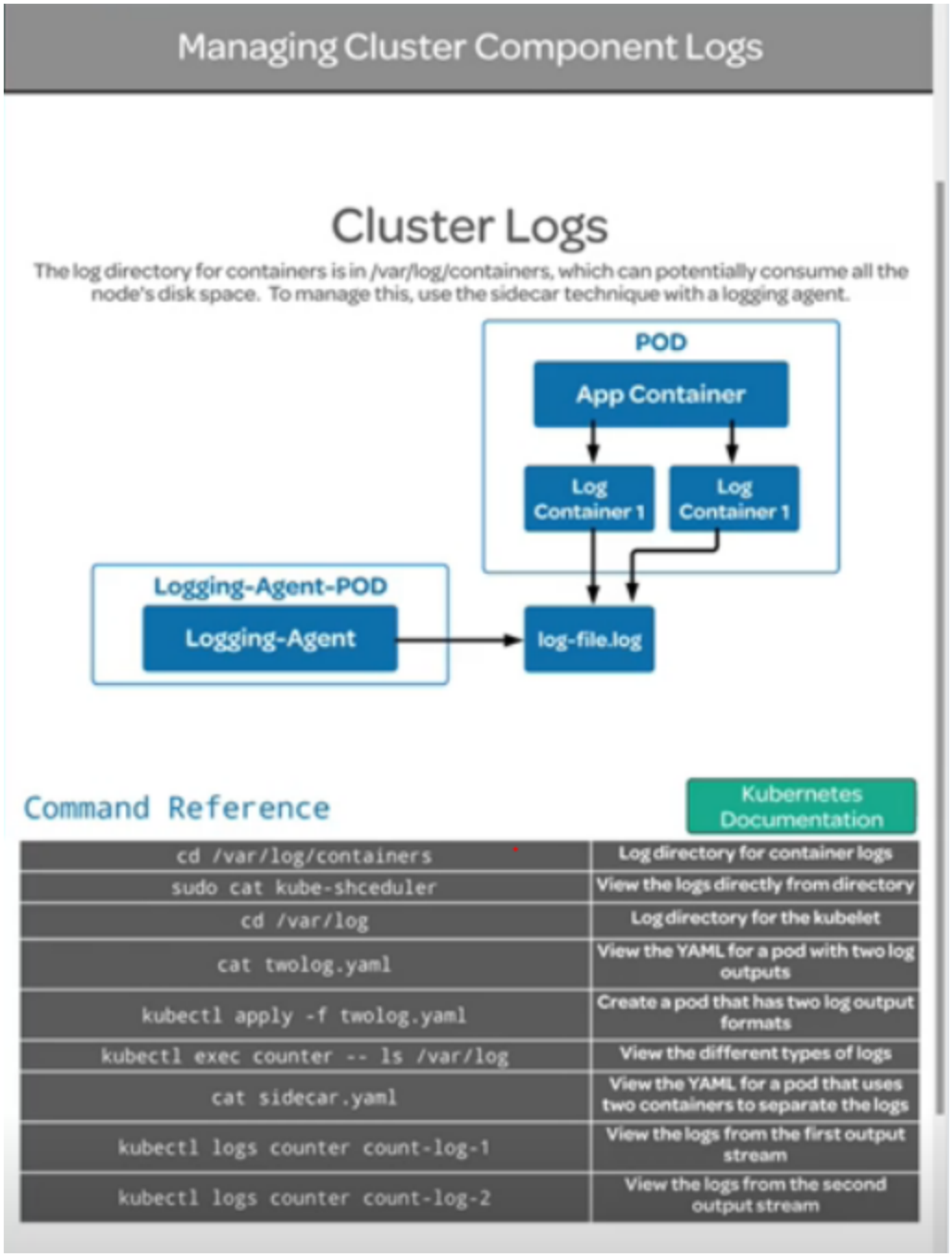
Get the list of endpoints to see that the repaired pod is part of the service again:

```
kubectl get ep
```

Helpful Links

- Container Probes

Managing Cluster Component Logs



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

```
/var/log/containers
```

The directory where kubelet stores its logs:

```
/var/log
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

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

```
kubectl apply -f twolog.yaml
```

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

```
kubectl exec counter -- ls /var/log
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      i=0;
      while true;
      do
        echo "$i: $(date)" >> /var/log/1.log;
        echo "$(date) INFO $i" >> /var/log/2.log;
        i=$((i+1));
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-1
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/1.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: count-log-2
```

```
    image: busybox
    args: [/bin/sh, -c, 'tail -n+1 -f /var/log/2.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```

View the first type of logs separately:

```
kubectl logs counter count-log-1
```

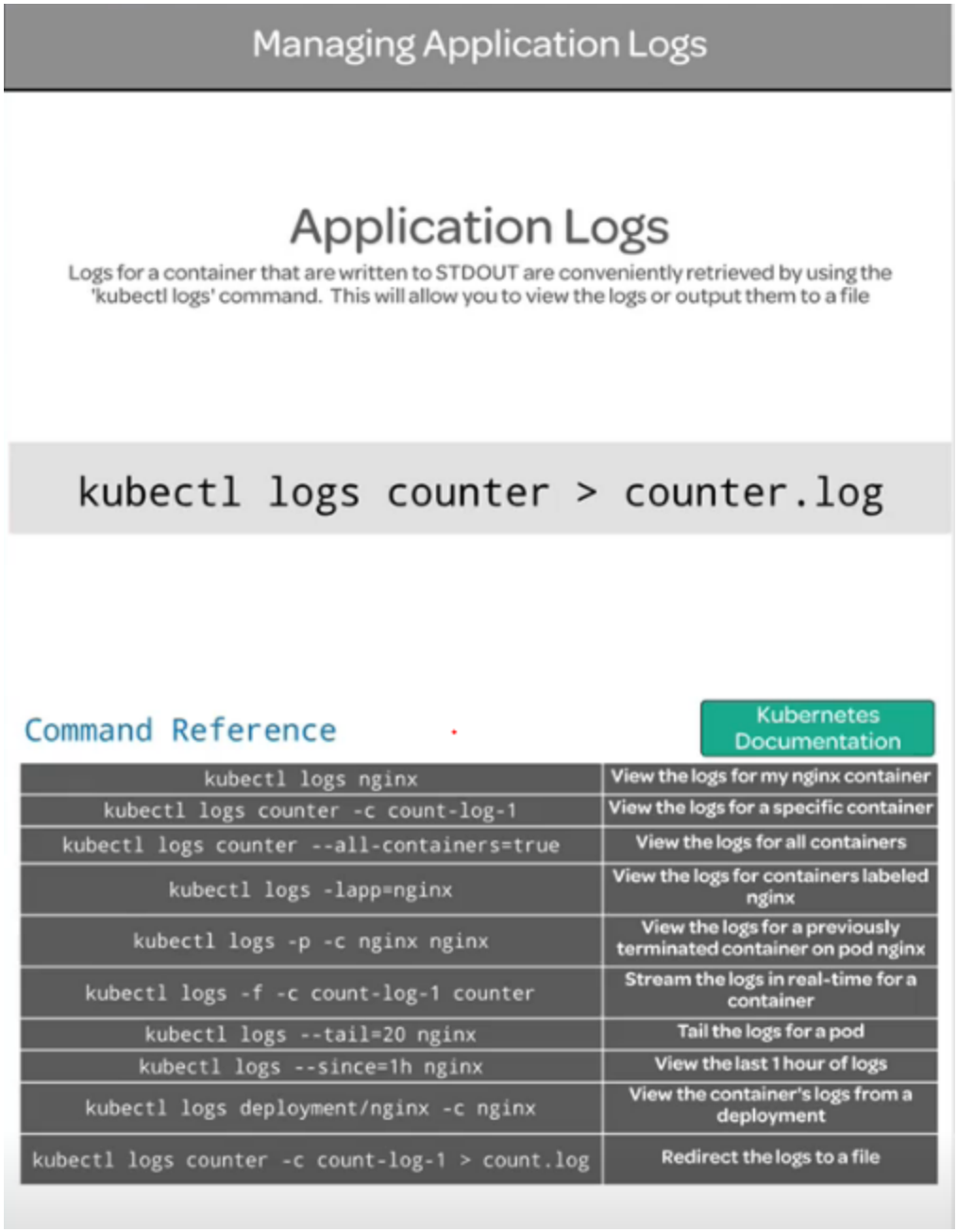
View the second type of logs separately:

```
kubectl logs counter count-log-2
```

Helpful Links

- Logging

Managing Application Logs



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

Get the logs from a pod:

```
kubectl logs nginx
```

Get the logs from a specific container on a pod:

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

Get the logs from all containers on the pod:

```
kubectl logs counter --all-containers=true
```

Get the logs from containers with a certain label:

```
kubectl logs -lapp=nginx
```

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

```
kubectl logs -p -c nginx nginx
```

Stream the logs from a container in a pod:

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

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

```
kubectl logs --tail=20 nginx
```

View the logs from a previous time duration:

```
kubectl logs --since=1h nginx
```

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

```
kubectl logs deployment/nginx -c nginx
```

Redirect the output of the logs to a file:

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

Helpful Links

- Logs

View the Persistent Volume.

Use the following command to view the Persistent Volume within the cluster:

```
kubectl get pv
```

Create a ClusterRole.

Use the following command to create the ClusterRole:

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

Create a ClusterRoleBinding.

Use the following command to create the ClusterRoleBinding:

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

Create a pod to access the PV.

1. Use the following YAML to create a pod that will proxy the connection and allow you to curl the address:

```
apiVersion: v1
kind: Pod
metadata:
  name: curlpod
  namespace: web
spec:
  containers:
  - image: tutum/curl
    command: ["sleep", "9999999"]
    name: main
  - image: linuxacademycontent/kubectl-proxy
    name: proxy
  restartPolicy: Always
```

2. Use the following command to create the pod:

```
kubectl apply -f curlpod.yaml
```

Request access to the PV from the pod.

1. Use the following command (from within the pod) to access a shell from the pod:

```
kubectl exec -it curlpod -n web -- sh
```

2. Use the following command to curl the PV resource:

```
curl localhost:8001/api/v1/persistentvolumes
```

Learning Objectives

Identify the problematic pod in your cluster.

Use the following command to view all the pods in your cluster:

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

Collect the logs from the pod.

Use the following command to collect the logs from the pod:

```
kubectl logs <pod_name> -n <namespace_name>
```

Output the logs to a file.

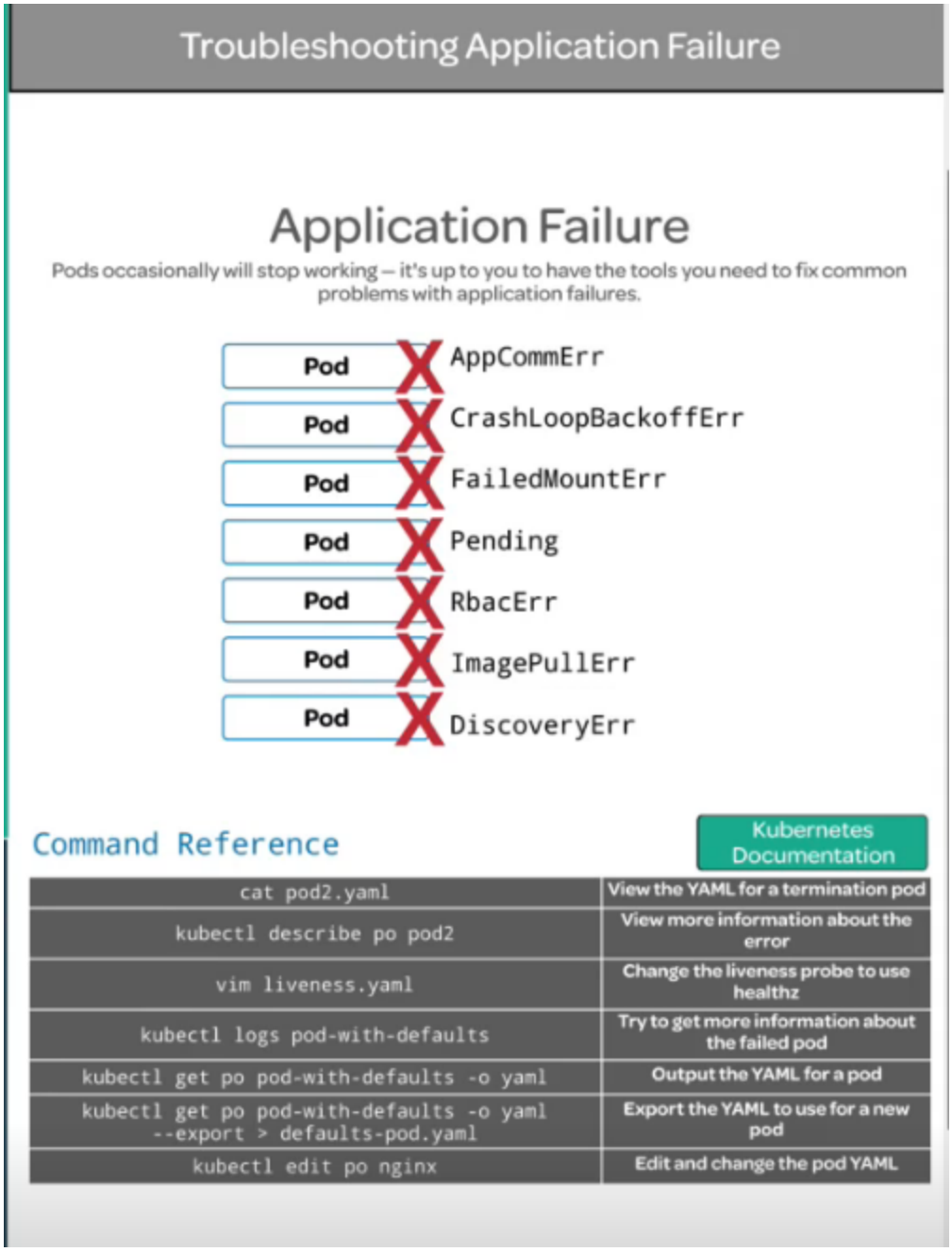
Use the following command to output the logs to a file:

```
kubectl logs <pod_name> -n <namespace_name> > broken-pod.log
```

Troubleshooting (10%)

Identifying Failure within the Kubernetes Cluster

Troubleshooting Application Failure



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

```
apiVersion: v1

kind: Pod

metadata:
```

```
  name: pod2
spec:
  containers:
  - image: busybox
    name: main
    command:
    - sh
    - -c
    - 'echo "I''ve had enough" > /var/termination-reason ; exit 1'
    terminationMessagePath: /var/termination-reason
```

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

```
kubectl describe po pod2
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness
spec:
  containers:
  - image: linuxacademycontent/kubeserve
    name: kubeserve
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8081
```

View the logs for additional detail:

```
kubectl logs pod-with-defaults
```

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

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

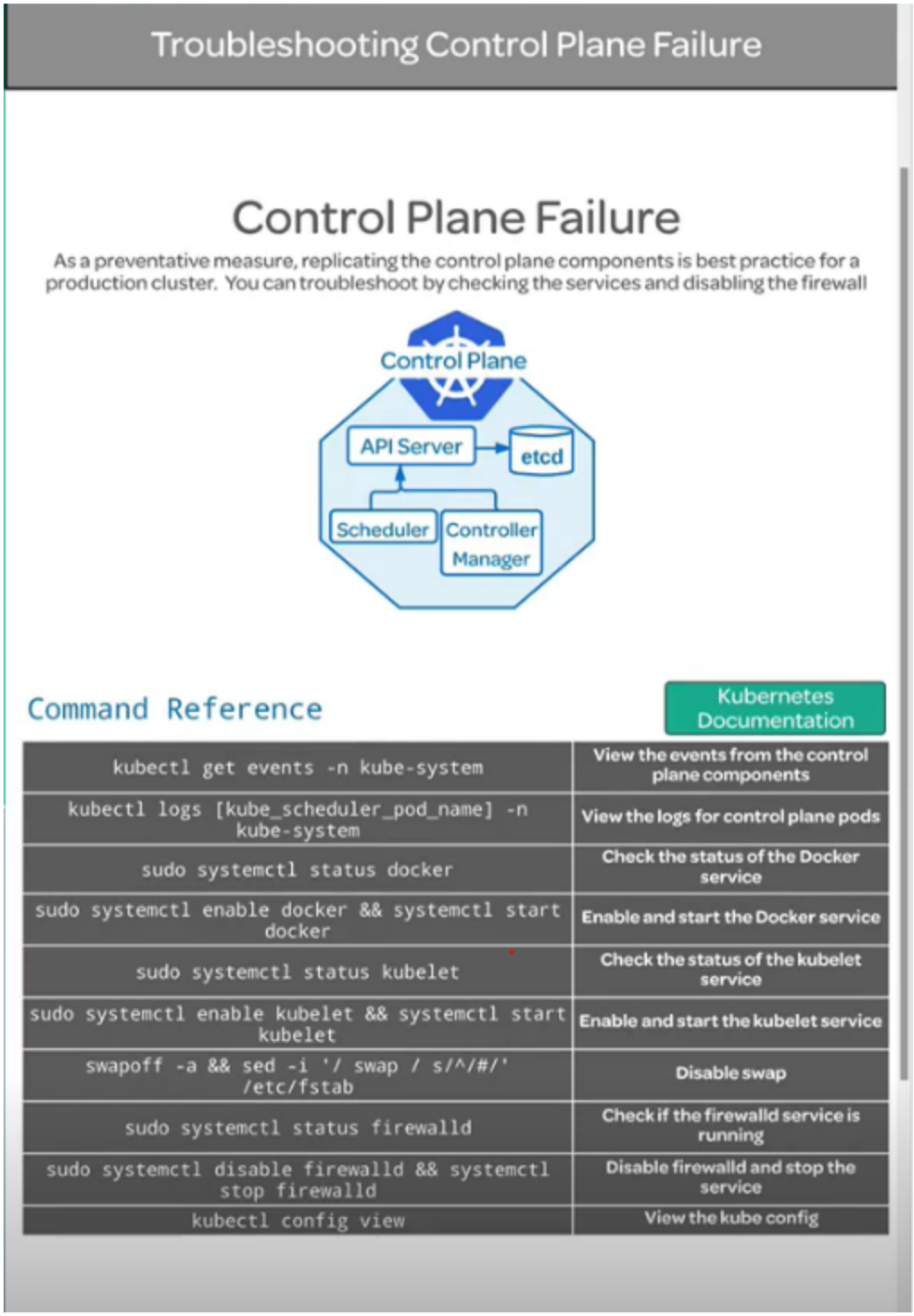
Edit a pod directly (i.e., changing the image):

```
kubectl edit po nginx
```

Helpful Links

- Pod Failure Reasons
- Debug the Application
- Troubleshoot Applications
- The Pod Lifecycle

Troubleshooting Control Plane Failure



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

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

```
kubectl get events -n kube-system
```

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

```
kubectl logs [kube_scheduler_pod_name] -n kube-system
```

Check the status of the Docker service:

```
sudo systemctl status docker
```

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

```
sudo systemctl enable docker && systemctl start docker
```

Check the status of the kubelet service:

```
sudo systemctl status kubelet
```

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

```
sudo systemctl enable kubelet && systemctl start kubelet
```

Turn off swap on your machine:

```
sudo su -

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

Check if you have a firewall running:

```
sudo systemctl status firewalld
```

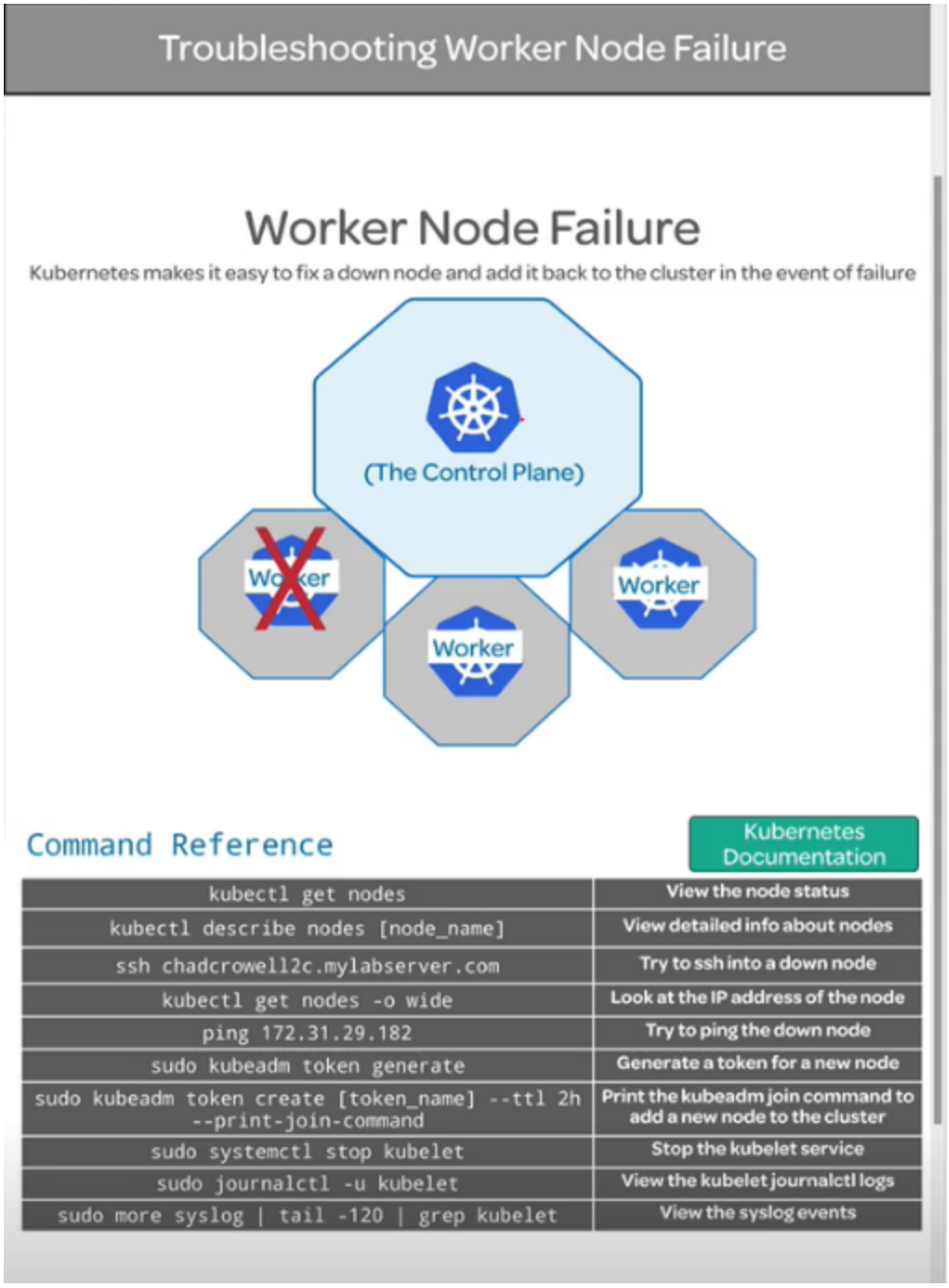
Disable the firewall and stop the firewalld service:

```
sudo systemctl disable firewalld && systemctl stop firewalld
```

Helpful Links

- Cluster Failure Overview
- Master HA

Troubleshooting Worker Node Failure



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

```
kubectl get nodes
```

Find out more information about the nodes with `kubectl describe`:

```
kubectl describe nodes chadcrowell2c.mylabserver.com
```

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

```
ssh chadcrowell2c.mylabserver.com
```

Get the IP address of your nodes:

```
kubectl get nodes -o wide
```

Use the IP address to further probe the server:

```
ssh cloud_user@172.31.29.182
```

Generate a new token after spinning up a new server:

```
sudo kubeadm token generate
```

Create the kubeadm join command for your new worker node:

```
sudo kubeadm token create [token_name] --ttl 2h --print-join-command
```

View the journalctl logs:

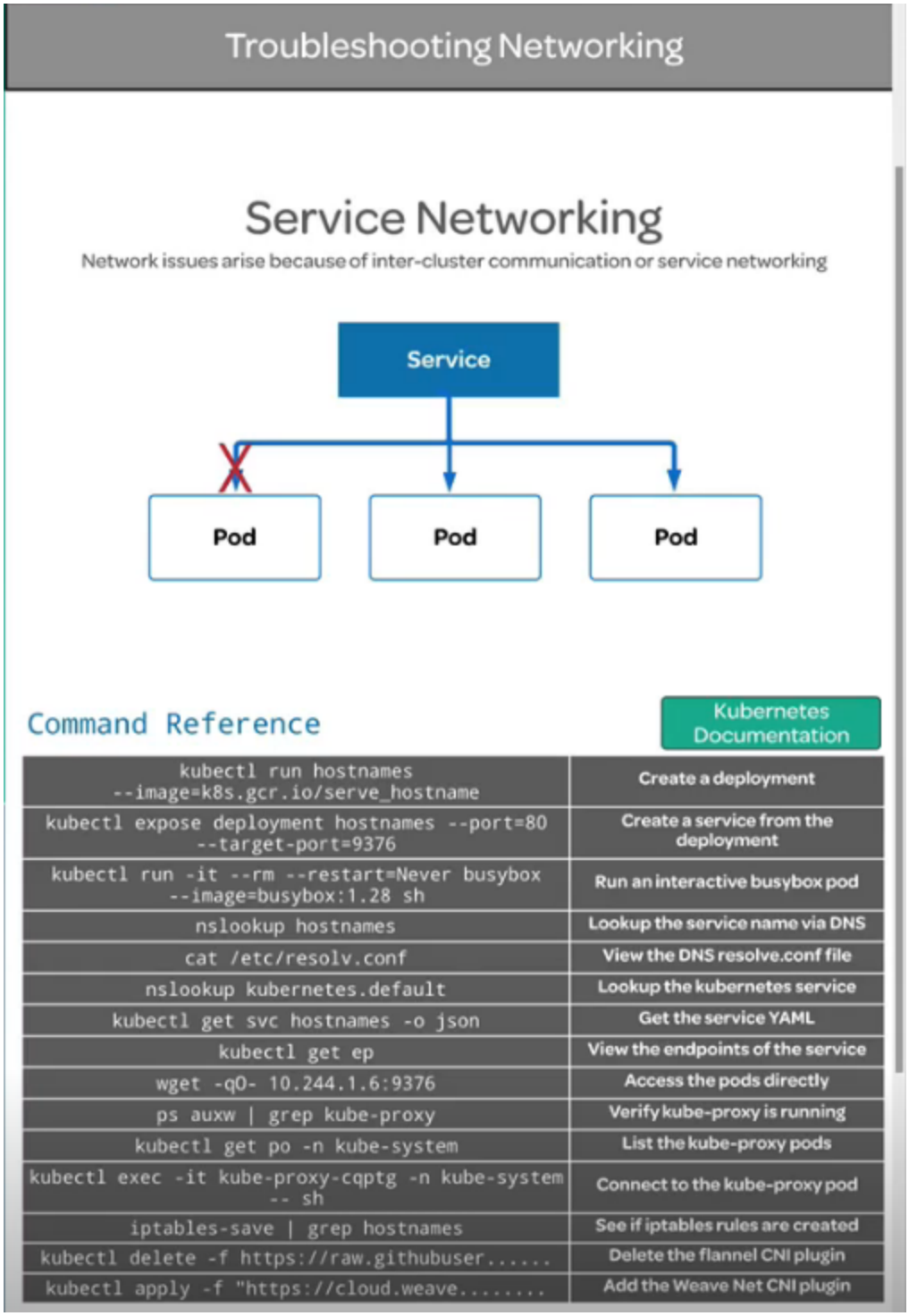
```
sudo journalctl -u kubelet
```

View the syslogs:

```
sudo more syslog | tail -120 | grep kubelet
```

- Nodes
- Explore Nodes

Troubleshooting Networking



Network issues usually start to arise internally or when using a service. In this lesson, we'll go through the many methods to see if your app is serving traffic by creating a service and testing the communication within the cluster.
Run a deployment using the container port 9376 and with three replicas:

```
kubectl run hostnames --image=k8s.gcr.io/serve_hostname \
                      --labels=app=hostnames \
                      --port=9376 \
                      --replicas=3
```

List the services in your cluster:

```
kubectl get svc
```

Create a service by exposing a port on the deployment:

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

Run an interactive busybox pod:

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

From the pod, check if DNS is resolving hostnames:

```
# nslookup hostnames
```

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

```
# cat /etc/resolv.conf
```

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

```
# nslookup kubernetes.default
```

Get the JSON output of your service:

```
kubectl get svc hostnames -o json
```

View the endpoints for your service:

```
kubectl get ep
```

Communicate with the pod directly (without the service):

```
wget -qO- 10.244.1.6:9376
```

Check if kube-proxy is running on the nodes:

```
ps auxw | grep kube-proxy
```

Check if kube-proxy is writing iptables:

```
iptables-save | grep hostnames
```

View the list of `kube-system` pods:

```
kubectl get pods -n kube-system
```

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

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

Delete the flannel CNI plugin:

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

Apply the Weave Net CNI plugin:

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

Helpful Links

- Debugging Services in Kubernetes
- Port Forwarding to Access a Pod
- Troubleshooting
- Installing the CNI Plugin

Learning Objectives

Identify the broken pods.

Use the following command to see what's in the cluster:

```
kubectl get all --all-namespaces
```

Find out why the pods are broken.

Use the following command to inspect the pod and view the events:

```
kubectl describe pod <pod_name>
```

Repair the broken pods.

Use the following command to repair the broken pods in the most efficient manner:

```
kubectl edit deploy nginx -n web
```

Ensure pod health by accessing the pod directly.

Start busybox pod

```
kubectl run busybox --image=busybox --rm -it --
restart=Never -- sh
```

Use the following command to access the pod directly via its container port:

```
wget -qO- <pod_ip_address>:80
```

Practice EXAM 1

Additional Information and Resources

You have been given access to a three-node cluster. Within that cluster, you will be responsible for creating a service for end users of your web application. You will ensure the application meets the specifications set by the developers and the proper resources are in place to ensure maximum uptime for the app. You must perform the following tasks in order to complete this hands-on lab:

- All objects should be in the `web` namespace.
- The deployment name should be `webapp`.
- The deployment should have 3 replicas.
- The deployment's pods should have one container using the `linuxacademycontent/podofminerva` image with the tag `latest`.
- The service should be named `web-service`.
- The service should forward traffic to port 80 on the pods.
- The service should be exposed externally by listening on port 30080 on each node.
- The pods should be configured to check the `/healthz`. endpoint on port 8081, and automatically restart the container if the check fails.
- The pods should be configured to not receive traffic until the endpoint on port 80 responds successfully.

Learning Objectives

check_circle**Create a deployment named `webapp` in the `web` namespace and verify connectivity.**keyboard_arrow_up

1. Use the following command to create a namespace named `web`:

```
kubectl create ns web
```

2. Use the following command to create a deployment named `webapp`:

```
kubectl run webapp --image=linuxacademycontent/podofminerva:latest
--port=80 --replicas=3 -n web
```

check_circle**Create a service named `web-service` and forward traffic from the pods.**keyboard_arrow_up

1. Use the following command to get the IP address of a pod that's a part of the deployment:

```
kubectl get po -o wide -n web
```

2. Use the following command to create a temporary pod with a shell to its container:

```
kubectl run busybox --image=busybox --rm -it --restart=Never -- sh
```

3. Use the following command (from the container's shell) to send a request to the web pod:

```
wget -O- <pod_ip_address>:80
```

4. Use the following command to create the YAML for the service named `web-service`:

```
kubectl expose deployment/webapp --port=80 --target-port=80
--type=NodePort -n web --dry-run -o yaml > web-service.yaml
```

5. Use Vim to add the namespace and the NodePort to the YAML:

```
vim web-service.yaml
```

   Change the name to `web-service`, add the namespace `web`, and add `nodePort: 30080`.

6. Use the following command to create the service:

```
kubectl apply -f web-service.yaml
```

7. Use the following command to verify that the service is responding on the correct port:

```
curl localhost:30080
```

8. Use the following command to modify the deployment:

```
kubectl edit deploy webapp -n web
```

9. Add the liveness probe and the readiness probe:

```
10. livenessProbe:
11.   httpGet:
12.     path: /healthz
```

```
13.      port: 8081
14.  readinessProbe:
15.    httpGet:
16.      path: /
```

```
      port: 80
```

17. Use the following command to check if the pods are running:

```
kubectl get po -n web
```

18. Use the following command to check if the probes were added to the pods:

```
kubectl get po <pod_name> -o yaml -n web --export
```

You have been given access to a two-node cluster. Within that cluster, a `PersistentVolume` has already been created. You must identify the size of the volume in order to make a `PersistentVolumeClaim` and mount the volume to your pod. Once you have created the PVC and mounted it to your running pod, you must copy the contents of `/etc/passwd` to the volume. Finally, you will delete the pod and create a new pod with the volume mounted in order to demonstrate the persistence of data. You must perform the following tasks in order to complete this hands-on lab:

- All objects should be in the `web` namespace.
- The `PersistentVolumeClaim` name should be `data-pvc`.
- The PVC request should be 256 MiB.
- The access mode for the PVC should be `ReadWriteOnce`.
- The storage class name should be `local-storage`.
- The pod name should be `data-pod`.
- The pod image should be `busybox` and the tag should be `1.28`.
- The pod should request the `PersistentVolumeClaim` named `data-pvc`, and the volume name should be `temp-data`.
- The pod should mount the volume named `temp-data` to the `/tmp/data` directory.
- The name of the second pod should be `data-pod2`.

Learning Objectives

Create the `PersistentVolumeClaim`.

1. Use the following command to check if there is already a `web` namespace:

```
kubectl get ns
```

2. Use the following command to create a `PersistentVolumeClaim`:

```
vi data-pvc.yaml #copy and paste from docs
```

3. Change the following in the `data-pvc.yaml` file:

```
name: data-pvc
namespace: web
storageClassName: local-storage
storage: 256Mi
```

4. Use the following command to create the PVC:

```
kubectl apply -f data-pvc.yaml
```

Create a pod mounting the volume and write data to the volume.

1. Use the following command to create the YAML for a busybox pod:

```
kubectl run data-pod --image=busybox --restart=Never -o yaml --dry-run -
- /bin/sh -c 'sleep 3600' > data-pod.yaml
```

1. Use the following command to edit the `data-pod.yaml` file:

```
vi data-pod.yaml
```

1. Add the following to the `data-pod.yaml` file:

```
namespace: web
  volumeMounts:
  - name: temp-data
    mountPath: /tmp/data
volumes:
- name: temp-data
persistentVolumeClaim:
claimName: data-pvc
```

1. Use the following command to create the pod:

```
kubectl apply -f data-pod.yaml
```

1. Use the following command to connect to the pod:

```
kubectl exec -it data-pod -n web -- sh
```

1. Use the following command to copy the contents of the `etc/passwd` file to `/tmp/data`:

```
# cp /etc/passwd /tmp/data/passwd
```

1. Use the following command to list the contents of `/tmp/data`:

```
# ls /tmp/data/
```

Delete the pod and create a new pod and view volume data.

1. Use the following command to delete the pod:

```
kubectl delete po data-pod -n web
```

2. Use the following command to modify the pod YAML:

```
vi data-pod.yaml
```

3. Change the following line in the `data-pod.yaml` file:

```
name: data-pod2
```

4. Use the following command to create a new pod:

```
kubectl apply -f data-pod.yaml
```

5. Use the following command to connect to the pod and view the contents of the volume:

```
kubectl exec -it data-pod2 -n web -- sh
# ls /tmp/data
```

You have been given access to a three-node cluster. You will be responsible for creating a deployment and a service to serve as a front end for a web application. In addition to the web application, you must deploy a Redis database and make sure the web application can only access this database using the default port of 6379. You will first create a `default-deny` network policy, so all pods within your Kubernetes are not able to communicate with each other by default. Then you will create a second network policy that specifies the communication on port 6379 between the web application and the database using their label selectors. You must apply these specifications to your resources in order to complete this hands-on lab:

- Create a deployment named `webfront-deploy`.
- The deployment should use the image `nginx` with the tag `1.7.8`.
- The deployment should expose container port 80 on each pod and contain 2 replicas.
- Create a service named `webfront-service` and expose port 80, target port 80.
- The service should be exposed externally by listening on port 30080 on each node.
- Create one pod named `db-redis` using the image `redis` and the tag `latest`.
- Verify that you can communicate to pods by default.
- Create a network policy named `default-deny` that will deny pod communication by default.
- Verify that you can no longer communicate between pods.
- Apply the label `role=frontend` to the web application pods and the label `role=db` to the database pod.
- Create a network policy that will apply an ingress rule for the pods labeled with `role=db` to allow traffic on port 6379 from the pods labeled `role=frontend`.
- Verify that you have applied the correct labels and created the correct network policies.

Learning Objectives

Create a deployment and a service to expose your web front end.

1. Use the following command to create the YAML for your deployment:

```
kubectl create deployment webfront-deploy --image=nginx:1.7.8  --dry-run -o yaml > webfront-deploy.yaml
```

1. Add container port 80, to have your final YAML look like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: webfront-deploy
  name: webfront-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webfront-deploy
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: webfront-deploy
    spec:
      containers:
      - image: nginx:1.7.8
        name: nginx
        ports:
        - containerPort: 80
        resources: {}
status: {}
```

1. Use the following command to create your deployment:

```
kubectl apply -f webfront-deploy.yaml
```

1. Use the following command to scale up your deployment:

```
kubectl scale deployment/webfront-deploy --replicas=2
```

1. Use the following command to create the YAML for a service:

```
kubectl expose deployment/webfront-deploy --port=80 --target-port=80 --type=NodePort --
dry-run -o yaml > webfront-service.yaml
```

1. Add the name and the nodePort, the complete YAML will look like this:

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: webfront-deploy
  name: webfront-service
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30080
  selector:
    app: webfront-deploy
  type: NodePort
status:
  loadBalancer: {}
```

1. Use the following command to create the service:

```
kubectl apply -f webfront-service.yaml
```

1. Verify that you can communicate with your pod directly:

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

# wget -O- [pod_ip_address]:80
# wget --spider --timeout=1 webfront-service
```

Create a database server to serve as the backend database.

Use the following command to create a Redis pod:

```
kubectl run db-redis --image=redis --restart=Never
```

Create a network policy that will deny communication by default.

1. Use the following YAML (this is where you can use kubernetes.io and search "network policies" and then search for the text "default"):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

1. Use the following command to apply the network policy:

```
kubectl apply default-deny.yaml
```

1. Verify that communication has been disabled by default:

```
kubectl run busybox --rm -it --image=busybox /bin/sh
# wget -O- [pod_ip_address]:80
```

Apply the labels and create a communication over port 6379 to the database server.

1. Use the following commands to apply the labels:

```
kubectl get po
kubectl label po <pod_name> role=frontend
kubectl label po db-redis role=db
kubectl get po --show-labels
```

2. Use the following YAML to create a network policy for the communication between the two labeled pods (copy from kubernetes.io website):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: redis-netpolicy
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - port: 6379
```

3. Use the following command to create the network policy:

```
kubectl apply -f redis-netpolicy.yaml
```

4. Use the following command to view the network policies:

```
kubectl get netpol
```

5. Use the following command to describe the custom network policy:

```
kubectl describe netpol redis-netpolicy
```

6. Use the following command to show the labels on the pods:

```
kubectl get po --show-labels
```

How to Register for the CKA Exam

How to Register

Here is how you register for the Certified Kubernetes Administrator (CKA) exam, and what you will need in order to take the exam

What to Do:

- You **must** show your photo identification
- You **must** have reliable internet, use only Chrome or Chromium browser, have a webcam, have a microphone
- You may reschedule up to 24 hours in advance
- Cost is **$300.00** US. You get 1 free retake!
- Exam is **180** minutes long, choose a time slot when registering
- You may have one additional tab open to one of the following sites (not all):

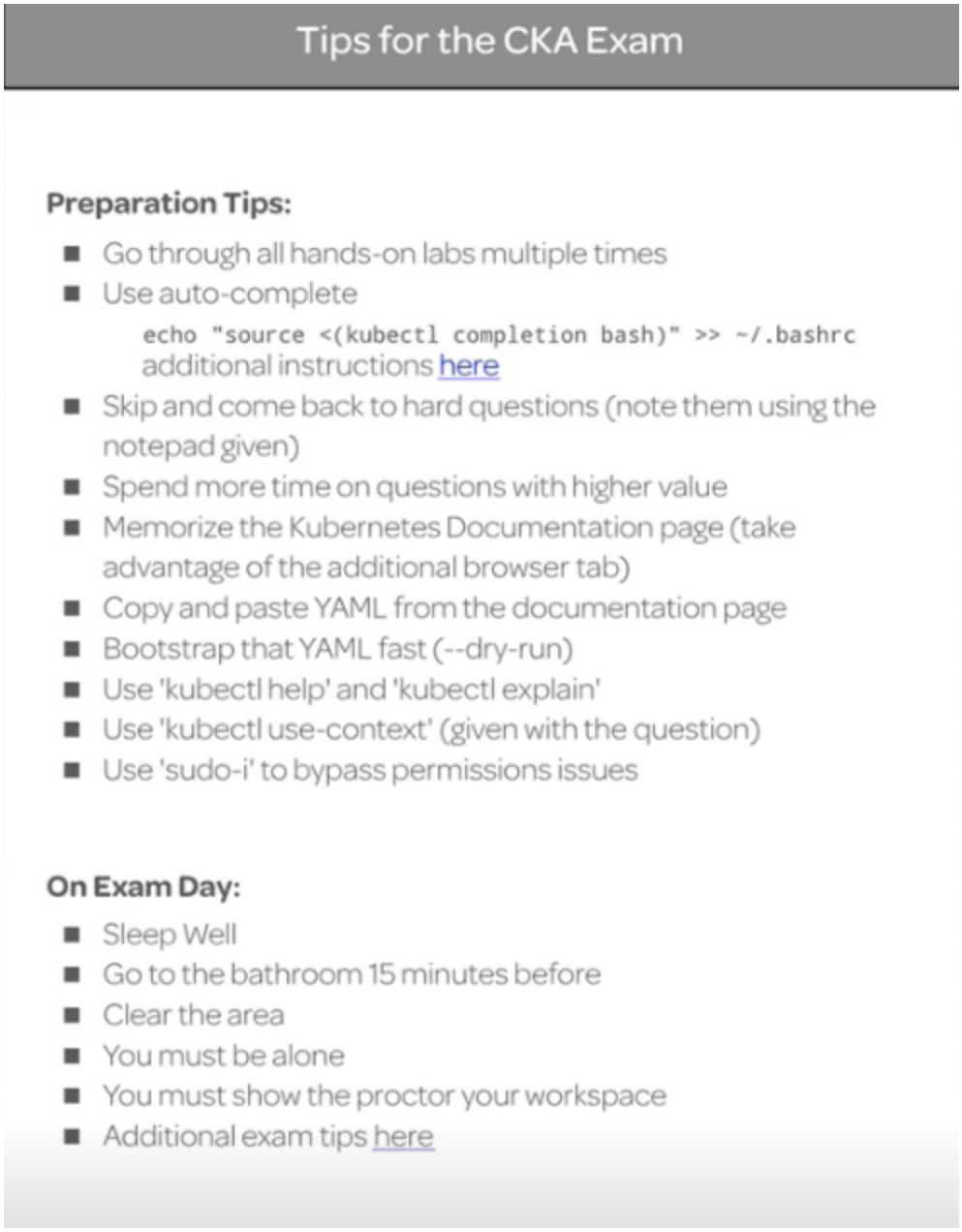
  https://kubernetes.io/docs/ (and all subdomains)
  https://github.com/kubernetes/ (and all subdomains)
  https://kubernetes.io/blog/

- Do NOT click link navigative away from the above domains
- Register through the Linux Foundation here
- Read the handbook here
- Run the Compatibility Check Tool here
- Verify your exam time and begin exam here
- You **must** complete the exam prep checklist before starting the exam
- Check your email to confirm the exam (add psionline.com to your white-list)
- Set the configuration context:

```
kubectl config use-context k8s
```

- The environment is currently running Kubernetes v1.13
- Sudo to rool with 'sudo -i' (view technical instructions by typing 'man lf_exam').

Register



This lesson will give you some tips for exam day.

Use `kubectl completion`

```
source <(kubectl completion bash)
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

https://kubernetes.io/docs/reference/kubectl/cheatsheet/#kubectl-autocomplete

Use `kubectl help`

```
kubectl help
```

https://kubernetes.io/docs/reference/kubectl/overview/#syntax

Use `kubectl explain`

Get documentation of various resources. For instance pods, nodes, services, etc.

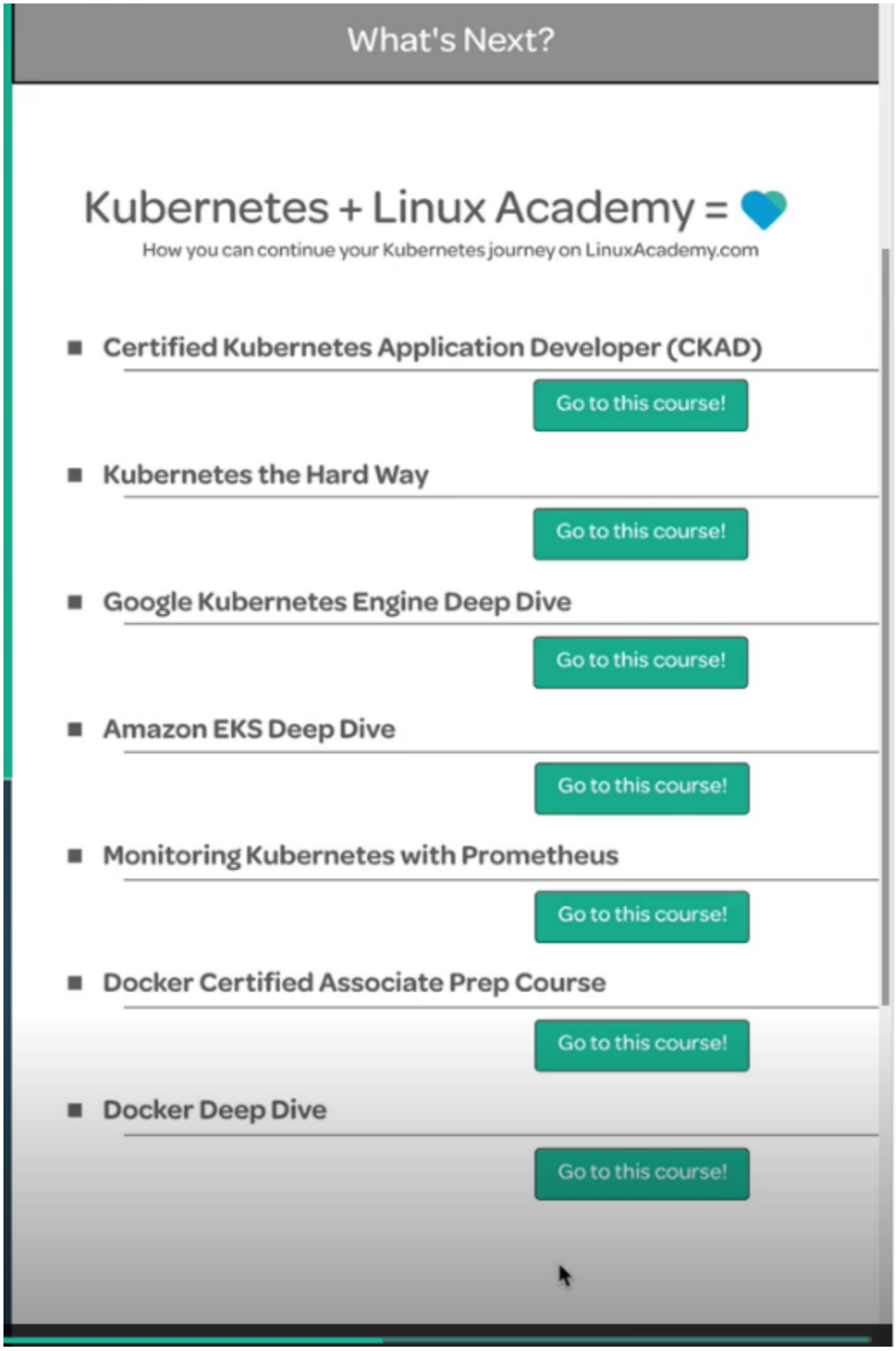
```
kubectl explain deployments
```

https://kubernetes.io/docs/reference/kubectl/overview/#operations

Switch Between Multiple Contexts

```
kubectl config use-context
```

https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/#define-clusters-users-and-contexts


What's Next?
Kubernetes + Linux Academy =
How you can continue your Kubernetes journey on LinuxAcademy.com
Certified Kubernetes Application Developer (CKAD)
Go to this course!
Kubernetes the Hard Way
Go to this course!
Google Kubernetes Engine Deep Dive
Go to this course!
Amazon EKS Deep Dive
Go to this course!
Monitoring Kubernetes with Prometheus
Go to this course!
Docker Certified Associate Prep Course
Go to this course!
Docker Deep Dive
Go to this course!