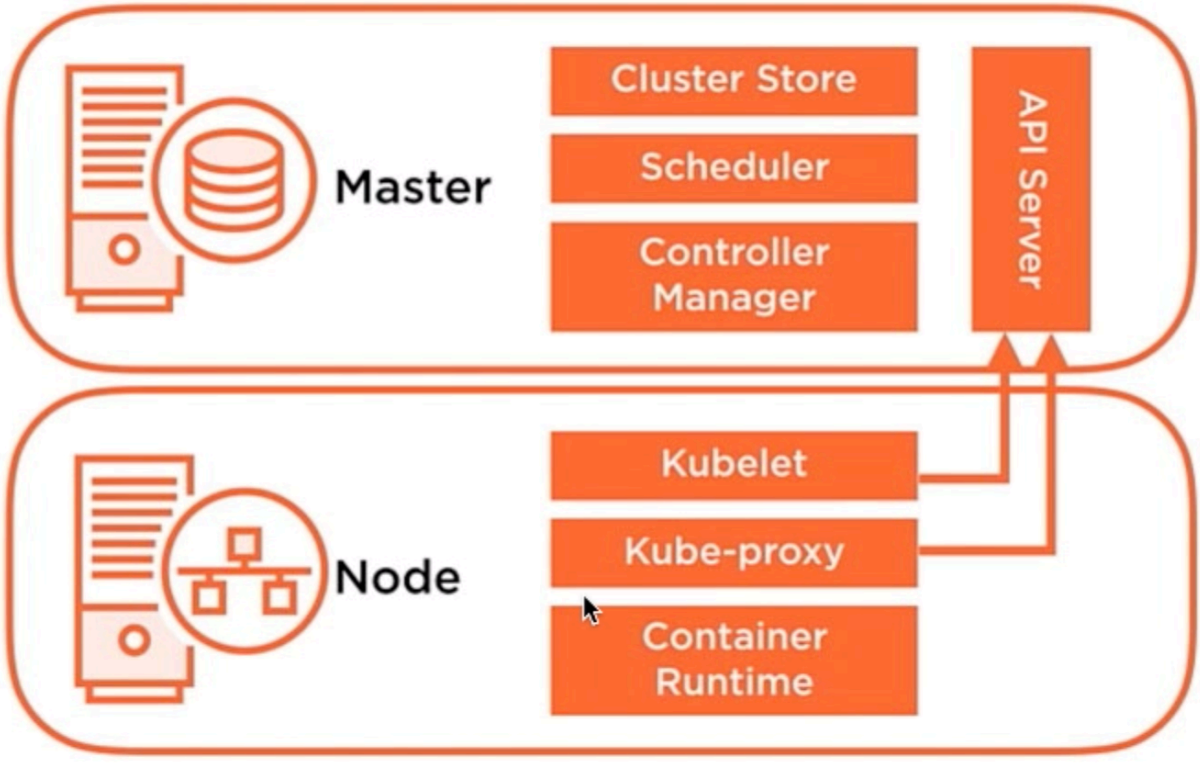
CS199 Kubernetes Notes for Independent Study

Course 1

Make sure all nodes are same OS! (I barfed on ubuntu 16 vs 18 resolv.conf BS!)

Master
  — API Server
  — Cluster Store (etc)
  — Scheduler
  — Controller Manager
Node
  — Kubelet
  — Kube Proxy
  — Container Runtime



“Taint” the master to prevent “user” pods from running on master

Installing k8s:

———— All of this below is worth keeping but basically ignore ————

Demo1

```
swapoff -a
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key
add -
sudo bash -c 'cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF'
sudo apt-get update
apt-cache policy kubelet

sudo apt-get install kubernetes-cni=0.6.0-00
sudo apt-get install kubelet=1.13.1-00 kubeadm=1.13.1-00  kubectl=1.13.1-00

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

sudo add-apt-repository    "deb [arch=amd64] https://download.docker.com/linux/
ubuntu \
$(lsb_release -cs) \
stable"
sudo apt-get update
sudo apt-get install docker-ce=18.06.0~ce~3-0~ubuntu
sudo docker run hello-world

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

Demo2

```
#only on master
wget https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/
hosted/rbac-kdd.yaml
wget https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/
hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml

sudo kubeadm init —pod-network-cidr=192.168.0.0/16
(Or on b1s VM's)
sudo kubeadm init --pod-network-cidr=192.168.0.0/16 --ignore-preflight-
errors=NumCPU
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

kubectl apply -f rbac-kdd.yaml
kubectl apply -f calico.yaml

———— All of this ^ is worth keeping but basically ignore ————

insure running ubuntu 18.04
lsb_release -a
Kubernetes master doesn't handle swap drives gracefully, turn off and insure enuff RAM, at least 2GB
swapoff -a

#install, enable, test docker
sudo apt-get update
sudo apt install docker.io
sudo systemctl start docker; sudo systemctl enable docker
sudo docker run hello-world

#install kubeadm, which in turn installs kubelet, and kubectl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
sudo apt-get update
sudo apt-get install kubeadm

# hold packages so they don't auto update
sudo apt-mark hold kubeadm kubelet kubectl docker.io
sudo apt-mark showhold

# initialize Kubernetes using private network 192.168.0.0/16. This network has nothing to
# to do with the VM's in the cluster. This is the separate network exclusive to pod networking
# this is the "flat" network that the CNI "overlay" simulates in software.
sudo kubeadm init --pod-network-cidr=192.168.0.0/16

kubeadm will spit out a string to join nodes into the cluster...it might looks like below
kubeadm join 10.0.2.4:6443 --token yle863.v15hdkt9kpxyepfh --discovery-token-ca-cert-hash sha256:e314db22ddc0633bf5942175dae53dafaf8efa76b651020fec702a58f88a0a08

```
# copy conf files and secure directory
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

# setup cmdline auto completion for kubectl
echo 'source <(kubectl completion bash)' >>~/.bashrc

# note: Coredns pods sit in pending until calico/weave-net?)

——— ignore below, doesn’t appear to work with calico ———

wget https://docs.projectcalico.org/manifests/calico.yaml
kubectl apply -f calico.yaml

Posted issue here: https://discuss.projectcalico.org/t/newb-q-cant-ping-pod-network-from-master-iaas-k8s-1-20-4-in-azure-on-ubuntu-18-04-w-calico/218/2

——— ignore ^ doesn’t appear to work with calico ———

# Install Weave (from https://medium.com/@patnaikshekhar/creating-a-kubernetes-cluster-in-azure-using-kubeadm-96e7c1ede4a)

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


kubectl get pods --all-namespaces
kubectl get pods --all-namespaces --watch


setup node:

# insure running ubuntu 18.04, disable swap
lsb_release -a
swapoff -a

#install, enable, test docker
sudo apt-get update
sudo apt install docker.io
sudo systemctl start docker
sudo systemctl enable docker
sudo docker run hello-world
```

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key
add -
sudo apt-add-repository "deb http://apt.kubernetes.io/ kubernetes-xenial main"
sudo apt-get update
sudo apt-get install kubeadm

sudo apt-mark hold kubeadm kubelet kubectl docker.io
sudo apt-mark showhold



# join node to the cluster
# Assuming worst case scenario, you no longer know the join string output from
the master on install
# if you lose join string. Below cmd likely doesn't return output.  On the MASTER:
kubeadm token list
# above should return empty, if so create new token:
kubeadm token create
kubeadm token list
# output will look like below:
# fgrwpb.uhyapbanbzc89gsm   23h         2021-05-21T17:12:31Z
authentication,signing   <none>  system:bootstrappers:kubeadm:default-node-
token
# copy 1st string above, i.e. frgwpb...  <- this need to be pasted into
KUBE_ADM_TOKEN_LIST below

# Next get discovery token ca cert hash:
openssl x509 -pubkey -in /etc/kubernetes/pki/ca.crt | openssl rsa -pubin -outform
der 2>/dev/null | openssl dgst -sha256 -hex | sed 's/^.* //'
# copy the output of above command   <- this need to be pasted into
OPENSSL_X509 below

# Now on the node, join the cluster
# kubeadm join 10.0.2.4:6443 --token KUBE_ADM_TOKEN_LIST --discovery-
token-ca-cert-hash sha256:OPENSSL_X509

Kubernetes rest API is CRUD (create, read, update, delete)

———— k8s plural sight course 1 section 4

Important kubectl cmds:
  create/apply: create a resource
  run: run a pod
```

explain: documentation (similar to man)
delete: delete a resource
get: list resources
describe: describe resource info—more information than get typically
exec: execute a command on the container
logs: view logs on a container

Samples of kubectl output formats: wide, yaml, json

kubectl cluster-info
kubectl get nodes
kubectl get nodes -o wide
kubectl get pods (nada!, no namespace)
kubectl get pods —all-namespaces
Or
kubectl get pods —namespace kube-system
kubectl get pods —namespace kube-system -o wide

Things to note. All kube-system pods run on the actual IP's (i.e. the 10.0.2.0/24 ip's) except for coredns and the calico-cube-controllers pod which lives on the pod network (i.e. 192.168.0.0/16)

kubectl get all —all-namespaces (get everything from all namespaces)
(Usually shows services, daemonsets, replicasets, deployments, pods

kubectl api-resources (gives all currently know api objects

kubectl explain pod (documentation of a pod)
kubectl explain pod.spec
kubectl explain pod.spec.containers

kubectl describe node (this is good for looking at resource usage)
kubectl describe node c1-master1 (same as above, but on single node)
kubectl describe (can be used on more than just a node resource….duh)

kubectl -h (help, can be used after commands)

Imperative (one thing at a time) vs declarative (declare end state in code)

We can do things declarative with our own manifest (typically written in yaml or json) and by doing a kubectl apply

Basic manifest has an apiVersion, kind (object, like a pod), metadata (like a name,

like nginx-pod), spec (like a container)

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

Do things imperatively:
kubectl run hello-world-pod --image=gcr.io/google-samples/hello-app:1.0 (this creates a single pod)
kubectl create deployment hello-world-deploy --image=gcr.io/google-samples/hello-app:1.0 (this creates a deployment of hello-world-deploy)

Now, you can ssh to work node (c1-node1) and run "sudo docker ps" and see that the node is running two containers (but the same image ID). One is the pod (hello-world) and the other is the deploy (hello-world-deploy)

kubectl logs hello-world (tab tab helpful here. This shows the logs from the pod—which seem pretty terse)

kubectl exec -it hello-world-pod -- /bin/sh (super helpful to be able to execute cmds on the pod for troubleshooting. Also notice that once in the pod can run ifconfig and IP matches pod ip from a kubectl get pods -o wide)

kubectl get replicaset

kubectl describe deployment hello-world-deploy
kubectl describe pods hello-world-deploy (ok, this is a lil tricky 'cause this is a deployment, but is still a pod. Note, you cannot run kubectl describe deployment

hello-world-pod)

Next expose the deployment as a service.

kubectl expose deployment hello-world-deploy —port=80 —target-port=8080
(This will expose the deployment on port 80 of the hello-world-deploy service running on 8080)

kubectl get service hello-world-deploy (shows the cluster-ip of the service. ONLY available inside the cluster)

kubectl describe service hello-world-deploy (shows details about the service including endpoints and ports)

Strange things to note. Can ping the pod IP from the master:

parker@c1-master1:~$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
hello-world-deploy-57b47545df-xbzzl 1/1 Running 0 5m54s **10.44.0.1** c1-node1 <none> <none>

Can access hello-world app:
parker@c1-master1:~$ curl http://10.44.0.1:8080
Hello, world!
Version: 1.0.0
Hostname: hello-world-deploy-57b47545df-xbzzl
parker@c1-master1:~$

Can also access the service:
parker@c1-master1:~$ kubectl get service hello-world-deploy
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
hello-world-deploy ClusterIP 10.110.208.9 <none> 80/TCP 5m30s
parker@c1-master1:~$ curl http://10.110.208.9:80
Hello, world!
Version: 1.0.0
Hostname: hello-world-deploy-57b47545df-xbzzl
parker@c1-master1:~$

BUT, cannot ping the cluster IP:
parker@c1-master1:~$ ping 10.110.208.9
PING 10.110.208.9 (10.110.208.9) 56(84) bytes of data.

^C
--- 10.110.208.9 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5115ms

kubectl get endpoints hello-world-deploy (will list the endpoints of service)

kubectl get deployment hello-world-deploy -o yaml (this outputs the configuration of the deployment as yaml—can also be json)

Doing things declaratively.
—export option is deprecated. Instead, can use yq.
kubectl get deployments.apps hello-world-deploy -o yaml | yq eval 'del(.metadata.resourceVersion, .metadata.uid, .metadata.annotations, .metadata.creationTimestamp, .metadata.selfLink, .metadata.managedFields)' - > deployment-hello-world-deploy.yaml
kubectl get service hello-world-deploy -o yaml | yq eval 'del(.metadata.resourceVersion, .metadata.uid, .metadata.annotations, .metadata.creationTimestamp, .metadata.selfLink, .metadata.managedFields)' - > service-hello-world-deploy.yaml

The use kubectl apply to apply configuration.
kubectl apply -f deployment-hello-world-deploy.yaml
kubectl apply -f service-hello-world-deploy.yaml

Make change to running config
kubectl edit deployment hello-world-deploy

Course 2

Request to API server generally happen over HTTPS over REST using JSON. The API server is stateless. All state is stored in ETCD (or the cluster store)

Kubernetes API objects (kind): Pods, Deployments, Services, PersistentVolumes

Every manifest includes apiVersion, kind, metadata, and spec

Core API groups.
  Core (or Legacy group): Includes core components, like Pods Node, Namespace PersistentVolume, PersistentVolumeClaim
  Named API groups (common example: Storage API group), like apps (deployment), storage.k8s.io (StorageClass), rbac.authorization.k8s.io (Role)

The request to API server doesn't need to happen with kubectl. Can happen with any client, including curl that respects the API.
Generally yaml is converted to json before being sent to API server

A "context" is basically a cluster.

kubectl config get-contexts (gets a list of possible clusters to connect t)

kubectl config use-context kubernetes-admin@kubernetes

kubectl cluster-info (helps verify which "context" or cluster you are connected to).

kubectl api-resources (lists all of the available resources and which api version they are part of)

kubectl api-resources --api-group=storage.k8s.io (shows only the api resources that are part of storage.k8s.io extension).

kubectl api-verisons (lists all of the versions the api knows about). Try kubectl api-versions | sort

-v 6 (or 7 or 8, or 9, or 10, up to 100?). -v 6 seems to be the verbosity of choice.

-v flag can be used with watch

A good way to authenticate to the api server is running kubectl proxy & (good for later running curl cads)

-v can be used with kubectl logs. You can see that before grabbing log, kubectl checks tomato sure the pod exist before requesting logs. When you delete an object it sends a delete, and then checks to see if it exists.

Namespaces are a security and a naming boundary for role-based access controls. A resource can only be in 1 namespace. A resource can have the same name as long as they are in separate namespaces

Resources, like pods, deployments, services, etc are generally namespaced. Physical things, like nodes and volumes are generally not namespaces.

You get 4 namespaces "out of the box." Default, Kube-public (usually things shared by all namespaces, like configmaps), Kube-system (which includes the api-server, controller-manager, clusterstore, and the scheduler). When you delete a namespace it deletes all of the resources in the namespace. Namespaces cannot

have uppercase (and certain other) characters.

A label is a key/value pair. An object can have multiple labels associated with it.

Kubernetes uses labels internally. Controllers and services use labels to tie a service to a set of pods. Kubernetes uses labels to tie certain pods to certain nodes (if they have special hardware—like SSD's or GPU's).

kubectl get namepsaces (lists all namespaces)

kubectl api-resources --namespaced=true (list all of api-resources that can be namespaced)
kubectl api-resources --namespaced=false (list of all api-resources that cannot be namespaced)

kubectl describe namespaces (show where or not there are any labels or annotations, or resource limits with any namepspace). This also shows that a namespace has a "status" which in this case is active.

k create namespace playground1 (imperatively creates namespace playground1

k create namespace Playuground1 (this fails! Cannot have caps and must start/end with alphanumeric)
The Namespace "Playground1" is invalid: metadata.name: Invalid value: "Playground1": a lowercase RFC 1123 label must consist of lower case alphanumeric characters or '-', and must start and end with an alphanumeric character (e.g. 'my-name', or '123-abc', regex used for validation is '[a-z0-9]([-a-z0-9]*[a-z0-9])?')

Declaratively is fairly shORT:
apiVersion: v1
kind: Namespace
metadata:
  name: playgroundinyaml

k apply -f namespace.yaml

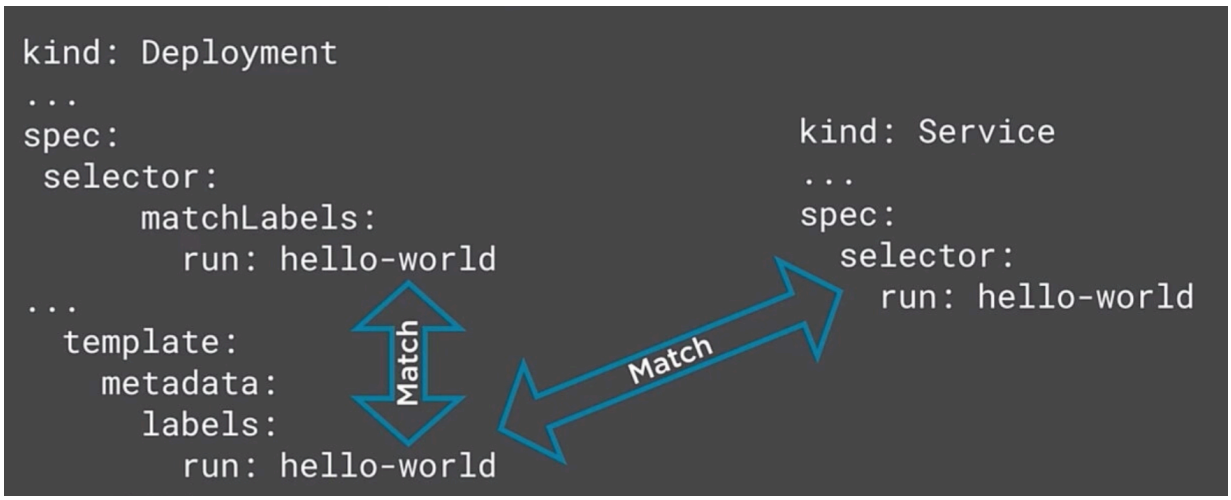
Create a deployment in the playground namespace (declaratively):
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
  namespace: playground1

-n is the same as —namespace

k delete pods -all -n playground1 (deletes all pods in the playground1 namespace, if the pods were part of a deployment, they will be recreated)

To delete the whole namespace (which will kill the deployment)
k delete namespace playground1

This is how you tie a deployment to a service using labels.



k get pods —show-labels (shows all of the labels associated with pods)

k get pod --selector tier=prod (show only pods with tier=prod)
k get pod -l tier=prod (-l is the same as —selector)
k get pod -l tier=prod --show-labels (only pod with tier=prod and shows all of the labels)

k get pod -l 'tier=prod,app=MyWebApp' --show-labels (shows only tier=prod AND app=MyWebApp, and shows labels)
k get pods -l ‘tier in (prod,qa)’ —show-labels (shows pods in prod OR qa, and shows labels)

k get pod -L tier (shows pods using label tier as a column)
k get pod -L tier,app (shows pods using labels tier and app as columns)

k label pod nginx-pod-1 tier=nonprod —overwrite (—overwrite required to overwrite a val. This will change tier label to nonprod)

k label pod nginx-pod-1 another=randolabel (add a new label another, with value randolabel)

k label pod nginx-pod-1 another- (this will remove another label)

k label pod —all tier=non-prod (this will label whatever pods not currently non-prod to non-prod)

k delete pod -l tier=non-prod (this will delete all pods with the label tier=non-prod —in our case all of the pods in default ns)

When you create a deployment, a replicates is also created. The replicaset can be named something like hello-world-54575d5b77

k describe deployment hello-world (shows labels, and shows the name of the replicaset)
k describe replicaset hello-world-54575d5b77 (shows a new selector pod-template-hash).

pod-template-hash is how the replicaset knows which pods are associated with the replicaset

k get pods —show-labels (will show the same pod-template-hash label as the replicaset)

If you relabel the pod-template-hash for a pod in the replicaset with say a “k label pod hello-world-54575d5b77-5pp45 pod-template-hash=DEBUG —overwrite”, kubernetes will launch a new pod because the replicaset is out of compliance. This is a useful technique for debugging a pod.

k describe service hello-world (returns 5 ip addresses as endpoints—odd because replicaset has 4)

The service at this point only matches on app: hello-world (it doesn’t use template-pod-hash), so if you want to kick the pod out of the load balancer, you need to “k label pod hello-world-54575d5b77-5pp45 app=DEBUG —overwrite”

Can also see the labels of a node with a k get no --show-labels (this is where you’ll also see the master shown as the control-plane)
k label node c1-node1 disk=local_ssd

The scheduler is what decides on what node to place a pod.

nodeSelector is used to tie a pod to a node

You can move pods or workloads with nodeSelector to avoid pinning workloads to a particular node by name.

Annotations are a lot like labels, but are only used for external tooling or people for information. Good for comments. Things like build or release information good for annotations. Annotations are usually done in metadata section of the yaml.

Pods are a unit of abstraction above containers. A pod is a unit of work. It is the scheduler's job to figure out where to put that unit of work. A pod is unit of deployment. A pod has resources associated with it, generally storage and networking. A single pod can run multiple containers, but that is generally uncommon. Multi-container pods usually have 2 containers that have a producer/consumer relationship.

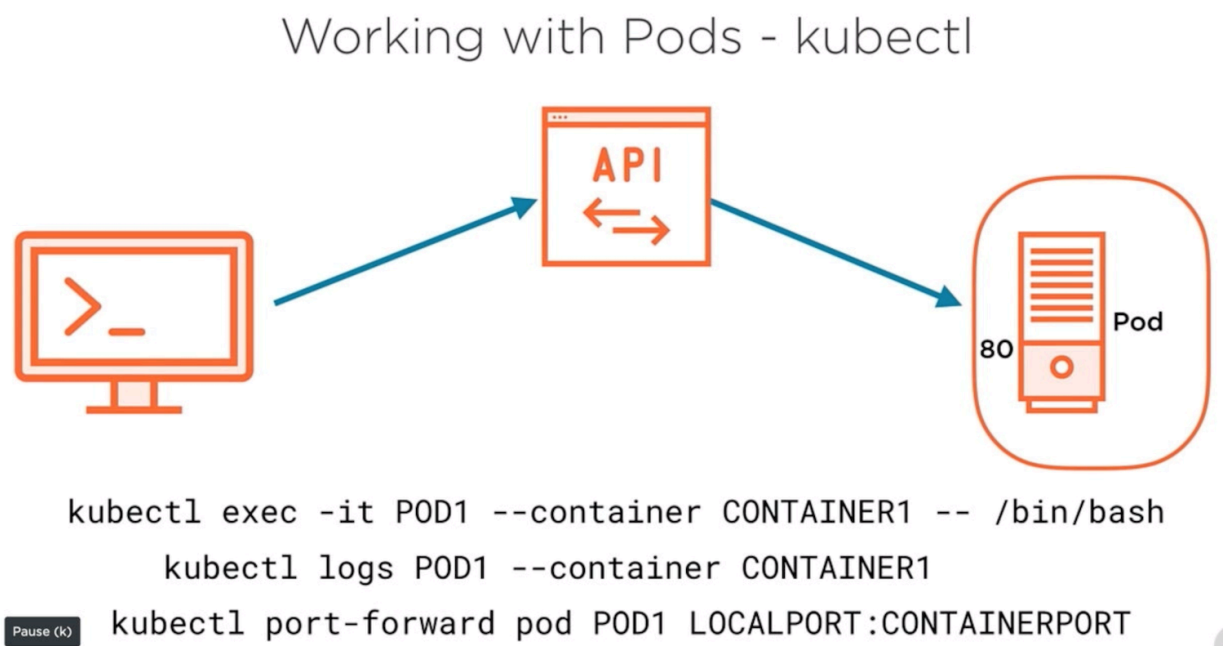
A single container pod usually only has 1 process running in the container.

The controller is responsible for starting/stopping pods and keeping apps in the desired state. And for scaling the application.

You don't want to run a "bare" or "naked" pod. A "bare or naked" pod is one that is not under the controller, daemonset, or replicaset control. K8s will not know how to deal with it if and when it fails.

When you fire off a bash shell inside of a container, you're actually going to the API server, then down to the Kubelet on the node, which then hands it off to the pod and streams all of the output back over the API server.

If you don't have a direct connection to the application running on a pod you can use port forwarding.

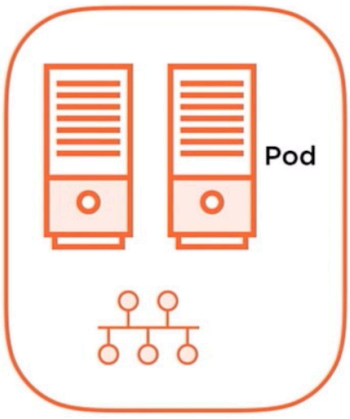


Running a multi-container pod is helpful when you have 2 processes that share a single resource.

Bad idea to put app server container and database container on the same pod for 1. Recovery 2. Scalability.

Multi-container pods share the same container network, so they must communicate over localhost/loopback. Be careful of port conflicts since they are one the same network

Shared Resources Inside a Pod - Networking

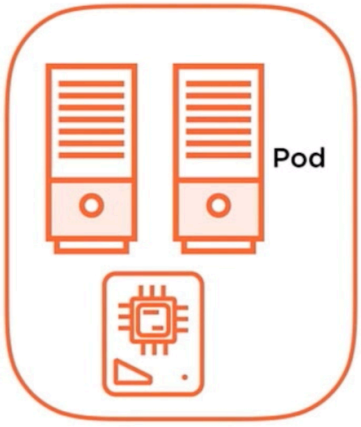


Shared loopback interface, used for communication over localhost

Be mindful of application port conflicts

Each container has its own filesystem, Volumes are defined at pod level and are shared amongst containers in the pod, so the volumes can be mounted into the containers and used to exchange data.

Shared Resources Inside a Pod - Storage



Each container image has it's own file system

Volumes are defined at the Pod level

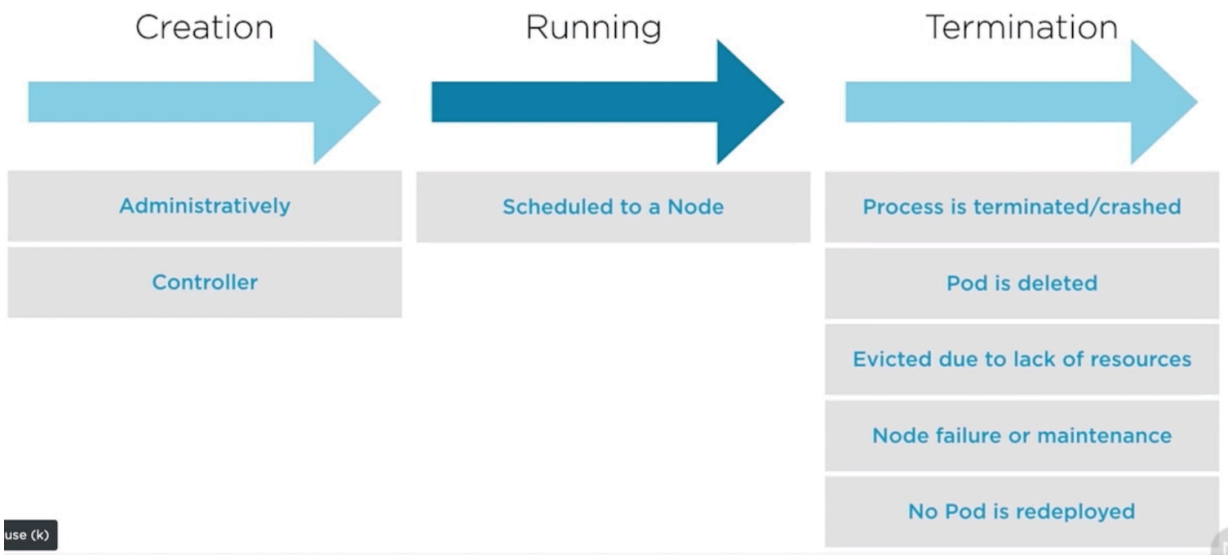
Shared amongst the containers in a Pod

Mounted into the containers' file system

Common way for containers to exchange data

This is the pod lifecycle and ways a pod can be put into various states:

Pod Lifecycle



use (k)