K8s = Kubernetes

asdfas234$$

Diagram: https://app.lucidchart.com/documents/view/b1b62a6e-1266-41dd-b48d-3c44252fb88f/fB.lPxz788ce

Containers: Are a bit like virtual machines, but they are smaller and start up faster.

Wrap software in independent portable packages, making it easy to quickly run software in a variety of environments.

Helps to ensure high availability and make it easy to scale resources.

Containers let's you to run a variety of software components across a cluster of generic software.

Container Runtime: Software that execute containers and manage container images on a node.

Docker is leading the container runtime market, but there are others like Lxd,Rkt (rocket) or Containerd.

Container orchestration tool (cot): The goal is to automate our application infrastructure and make it easy to manage.

If I want to scale up several containers to handle additional load, deploy new code to my entire cluster or ensure that multiple instances of a piece of software are spread across multiple servers for HA I can do it manually, taking a lot of time, or using a container orchestration tool.

Kubernetes: Is a container orchestration tool.
Was created by google to manage it's infrastructure. Is an open source software maintained by several companies and is leading by far the cot market.

https://kubernetes.io/docs/tutorials/kubernetes-basics/
https://vimeo.com/245778144/4d1d597c5e

Can relocate the app components anytime. Takes care of the service discovery, scaling, load balancing, self healing, etc.

Under the hood, kubernetes uses an API server to manipulate docker (or the container run time)

run: sudo ps docker → you will see all the containers there.

POD: Smallest unit of kubernetes. Can be either a single container or group of containers.
All containers in a pod have the same IP
All pods in a cluster have unique IPs in the same IP space (range).

IP Space: If one pod in the worker 1 have 10.10.10.x the other pods in the same and other worker nodes will have an IP from the same range.

All the pods in the same cluster are in the same IP space.

Software defined network (SDN): Flannel by Coreos, Weave Net by WeaveWorks,etc.

CNI: Container network interface

All k8s networking have to follow these rules:

1. All containers can communicate with all other containers without NAT
2. All nodes can communicate with all containers (and vice versa) without NAT
3. The IP address that the container sees itself as is the IP that other see it as.

Flannel: https://github.com/coreos/flannel/
It's very simple to use, doesn't have network policies like Calico or others,

You only have to run this command flannel installs everything:

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

Flannel runs flanneld on each host.

DNS:
Containers that are in one namespace than need to reach containers in another namespace they have to do a full lookup to the DNS server for a record of the other namespace.

Minikube: Runs a single kubernetes node cluster using a virtual machine on my computer.

Web UI (Dashboard): Dashboard UI. Web based kubernetes user interface. Created by google and more companies. Quite similar to GCP UI.

Components: Kubernetes consist on multiple components working together in order to provide the overall functionality of a k8s cluster. Many of them are running as pods.

For control-plane (Master Node)

Can not contain pods or be responsible for running the application components.
Can be replicated for HA.

- **API Server:** Serves the k8s API, the primary interface for the cluster. Simple REST based web API. All cluster components (scheduler,controller manager,kubelet, kube-proxy, etc) are pointing towards it to request information.

- **Scheduler:** Assigns your app to a worker node. Auto-detects which pod to assign to which node based on resources requirements, hardware constraints, etc.

- **Controller Manager:** Daemon. Maintains the cluster. Handles node failures, replicating components, maintaining the correct amount of pods, etc.

  Operates the cluster controllers:
  - Node Controller: Responsible for noticing and responding when nodes go down.
  - Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system
  - Endpoints Controller: Populates the Endpoint object, example: joins services and pods
  - Service Account & Token Controllers: Creates default account and API access tokens for new namespaces.

- **etcd**: Provides distributed, synchronized data storage for the cluster state. Not only stores the data, ensures that is synchronized across all the nodes. Stores the cluster configuration.

  Is a database where all of the objects within the k8s cluster are described.

  Recommended to back this up.

For all the individual nodes (workers)

Responsible for running the application, monitoring it and providing the services that it needs.

- **kubelet:** Daemon**.** Runs and manages the containers on the node and talks to the API server.

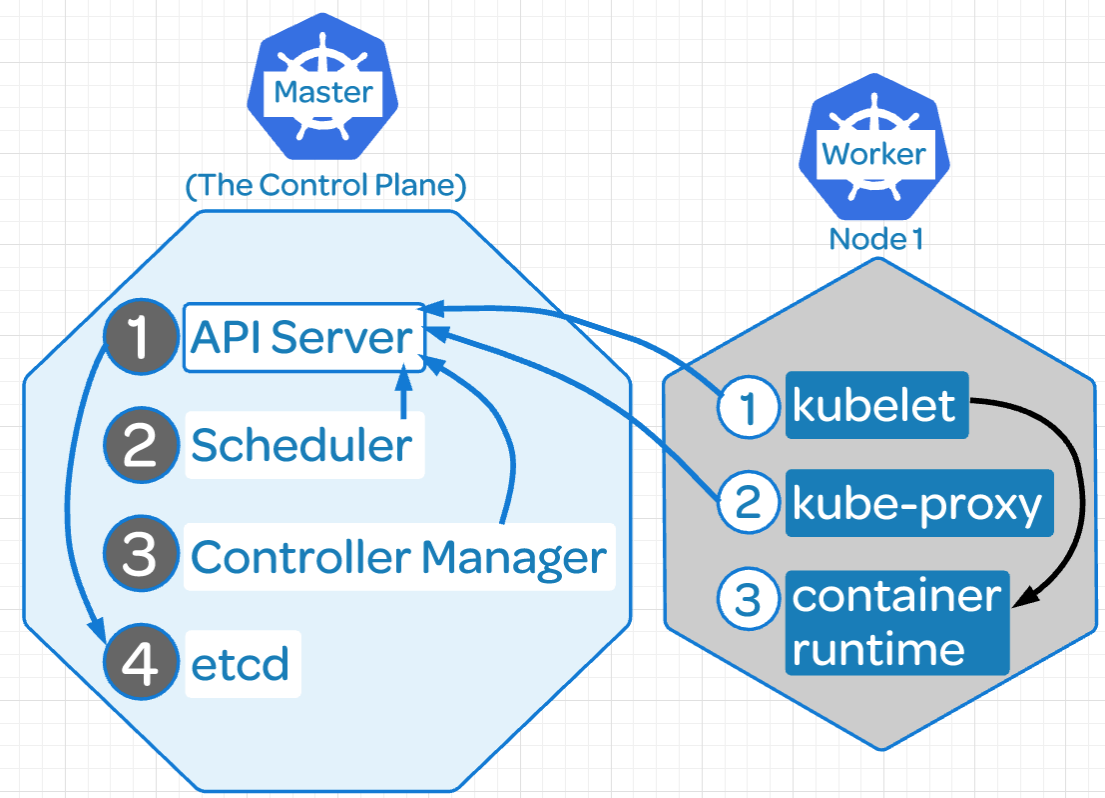
  Middle man between k8s API and the container runner. When a container need actually to be ran on a node, control-plane reaches out on that node and then the kubelet on that node instructs the container runner to go ahead and run that container.

  Is not running as a pod, runs as a service: sudo systemctl status kubelet

- **kube-proxy:** Load balances traffic between application components.

  Handles networking communications between nodes. Routes traffic. Writes firewall rules to the routing tables of each nodes to allow the traffic to be routed properly.

- **Container runtime:** Daemon**.**The program that runs the container (docker, rkt, containerd)



Declarative intent: Instead of having to manually check the cluster, k8s does this automatically it just says which status should be in.

Keystrokes: https://kubernetes.io/docs/reference/kubectl/cheatsheet/

See version:

- **docker:** sudo docker version
- **kubeadm:** sudo kubeadm version
- **kubectl:** sudo kubectl version

Get Nodes: kubectl get nodes

See all pods: kubectl get pods --all-namespaces -o wide
kubectl get pods -n kube-system
kubectl get pods -n <namespace> -w (w is like tail -f)
kubectl get pods name_of_the pod -o yaml (to see it's configuration)
kubectl get nodes -o yaml
kubectl get nodes -o name
kubectl get nodes -o wide

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

Describe nodes: kubectl describe nodes
kubectl describe

See deployment: kubectl get deployment
See services: kubectl get svc

Component Status: kubectl get componentstatus

Reset kubeadm: kubeadm reset → will remove the worker node from the cluster, then you can join the node again

Check logs: journalctl -u kubelet
/var/log/pods/(pod number)
kubectl get po --namespace kube-system -l app=flannel
kubectl logs --namespace kube-system **<POD_ID>** -c kube-flannel
kubectl logs --namespace kube-system **kube-flannel-ds-amd64-2mwxm** -c kube-flannel

See docker available versions: apt-cache madison docker-ce

Services: systemctl status kubelet

Configuration files: /etc/kubernetes/manifests/kube-controller-manager.yaml
Kube configuration: root/.kube/config

Check web server: kubectl exec busybox -- curl 10.244.1.4

Get inside a node:
kubectl exec -it nginx-7cdbd8cdc9-q9x78 /bin/bash
it stands for interactive

Do a loop to test autoscaling: hpa
while true; do wget -q -O http:/<your address here>; done

Check hpa: kubectl get hpa
Check hpa in live: whatch “kubectl get hpa”

Check who is the leader in HA:
kubectl get endpoints kube-scheduler -n kube-system -o yaml

Find the label applied to the etcd pod on the master node:
kubectl get pods --all-namespaces --show-labels -o wide

Delete a pod:
This will delete the pod but if is in a deployment the pod will be created again straigh away:
root@andrewjobson1c:~# **kubectl delete pods -n myy-ns test-5bdd698494-7xx8b**

pod "test-5bdd698494-7xx8b" deleted

Delete a deployment:
Here we are deleting the deployment test2 that have the namespace myy-ns
kubectl delete deployments test2 -n myy-ns

Alias:
alias kc='kubectl'
alias kgp='kubectl get pods'
alias kgs='kubectl get svc'
alias kgc='kubectl get componentstatuses’
alias kctx='kubectl config current-context’
alias kcon='kubectl config use-context’
alias kgc='kubectl config get-context'

Check state of etcd: etcdctl endpoint health

Enable kubernetes autocompletion:

There are two ways to do it:
1 -Source the completion script in your ~/.bashrc file:
echo 'source <(kubectl completion bash)' >>~/.bashrc

2 -Add the completion script to the /etc/bash_completion.d directory:
kubectl completion bash >/etc/bash_completion.d/kubectl

K8s Deployments:

Type of object on k8s. Allows you to automate the management of the pods.
Let’s you to specify a desired state for a set of pods.

Scaling: With a deployment you can specify the number of replicas (pods) you want and the deployment will create or remove pods to meet that number of replicas.

Deployment makes sure that the replicas are evenly spread out across your multiple worker nodes for HA and a lot more useful functionality like that.

Updates: With deployments, you can change the deployment image to a new version of the image. The deployment will gradually replace existing containers with the new version in order to avoid downtime, not like ReplicaSets.

Instead of removing the old pods and spin up the new ones will gradually spin the new ones and gradually replace and remove the old pods on an incremental fashion without incurring in any downtime for the users.

Self-Healing: If one pod is destroyed in the deployment will be immediately replaced by a new one by the deployment.

If the deployment says that I need 4 replicas for my application and anything happens to reduce that number, the deployments will spin the necessary replicas to have 4 working.

```
cat <<EOF | kubectl create -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.4
        ports:
        - containerPort: 80
EOF
```

It worked:
deployment.apps/nginx-deployment created

root@andrewjobson1c:~# **kubectl get po**

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-9b644dcd5-wgbdc	1/1	Running	0	44s
nginx-deployment-9b644dcd5-ws6xk	1/1	Running	0	44s

root@andrewjobson1c:~# **kubectl describe deployment nginx-deployment**

```
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Fri, 05 Jul 2019 08:36:18 +0000
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx
Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType:           RollingUpdate
```

```
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.15.4
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type          Status  Reason
  ----          ------  ------
  Available     True    MinimumReplicasAvailable
  Progressing   True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-9b644dcd5 (2/2 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  3m20s  deployment-controller  Scaled up replica set nginx-deployment-9b644dcd5 to 2
```

Deleting a pod, the deployment will spin up one new pod and will be operative is seconds.

```
root@andrewjobson1c:~# kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-9b644dcd5-wgbdc    1/1     Running   0          44s
nginx-deployment-9b644dcd5-ws6xk    1/1     Running   0          44s
```

```
root@andrewjobson1c:~# kubectl delete pod nginx-deployment-9b644dcd5-ws6xk
pod "nginx-deployment-9b644dcd5-ws6xk" deleted
```

This will take a few seconds

```
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-9b644dcd5-wgbdc    1/1     Running   0          44s
nginx-deployment-9b644dcd5-nvlmb    1/1     Running   0          7s
```

In bold the new pod created seven second ago.

Another Example:

Deploying nginx v1 with two replicas.

Update the nginx image, from v1 to v2:

root@andrewjobson1c:~/k8s_files# **kubectl set image deployment.v1.apps/example-deployment nginx=darealmc/nginx-k8s:v2**
deployment.apps/example-deployment image updated

See how the deployment takes care of removing the old pods (nginx v1) and creating the new ones (nginx v2)

First create a new one, example-deployment-56d5dfc8**7f** (second 29)
The remove one of the old ones (second 25)
After that creates the second one
And then remove the last one of the old pods (second 22)
All this to avoid to lose the website.

root@andrewjobson1c:~/k8s_files# **kubectl describe deployments example-deployment**
[...]
OldReplicaSets: <none>
NewReplicaSet: example-deployment-56d5dfc87f (2/2 replicas created)
Events:

Type	Reason	Age	From	Message
Normal	ScalingReplicaSet	11m	deployment-controller	Scaled up replica set example-deployment-6d98cb87fb to 2
Normal	ScalingReplicaSet	**29s**	deployment-controller	**Scaled up** replica set example-deployment-56d5dfc8**7f** to 1
Normal	ScalingReplicaSet	**25s**	deployment-controller	**Scaled down** replica set example-deployment-6d98cb87fb to 1
Normal	ScalingReplicaSet	**25s**	deployment-controller	**Scaled up** replica set example-deployment-56d5dfc8**7f** to 2
Normal	ScalingReplicaSet	**22s**	deployment-controller	**Scaled down** replica set example-deployment-6d98cb87**fb to 0**

Services:

Provide load balanced usage for a dynamic list of pods.

Pods are constantly being created and destroyed. To allow us to access to the application inside the pods we are using services.

Services are an abstraction layer on top of a set of pods. You can access the services rather than the pods.

If I do a request to a k8s service I'm going to get a response from one of the pods that is part of that service. And even if the list of the pods changes, the service is going to ensure that my request is always routed to one of the currently active pods that is part of the service.

```
cat << EOF | kubectl create -f -
kind: Service
apiVersion: v1
metadata:
  name: nginx-service  → name that the service will have
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
```

```
    port: 80
    targetPort: 80
    nodePort: 30080  → allows the service to be exposed externally, there are other better ways.
  type: NodePort
EOF
```

- Check that the service is working:

root@andrewjobson1c:~# **kubectl get svc**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	2d6h
nginx-service	NodePort	10.110.240.122	<none>	**80:30080/TCP**	12h

The k8s service is there by default.

Do a call to the service internally:

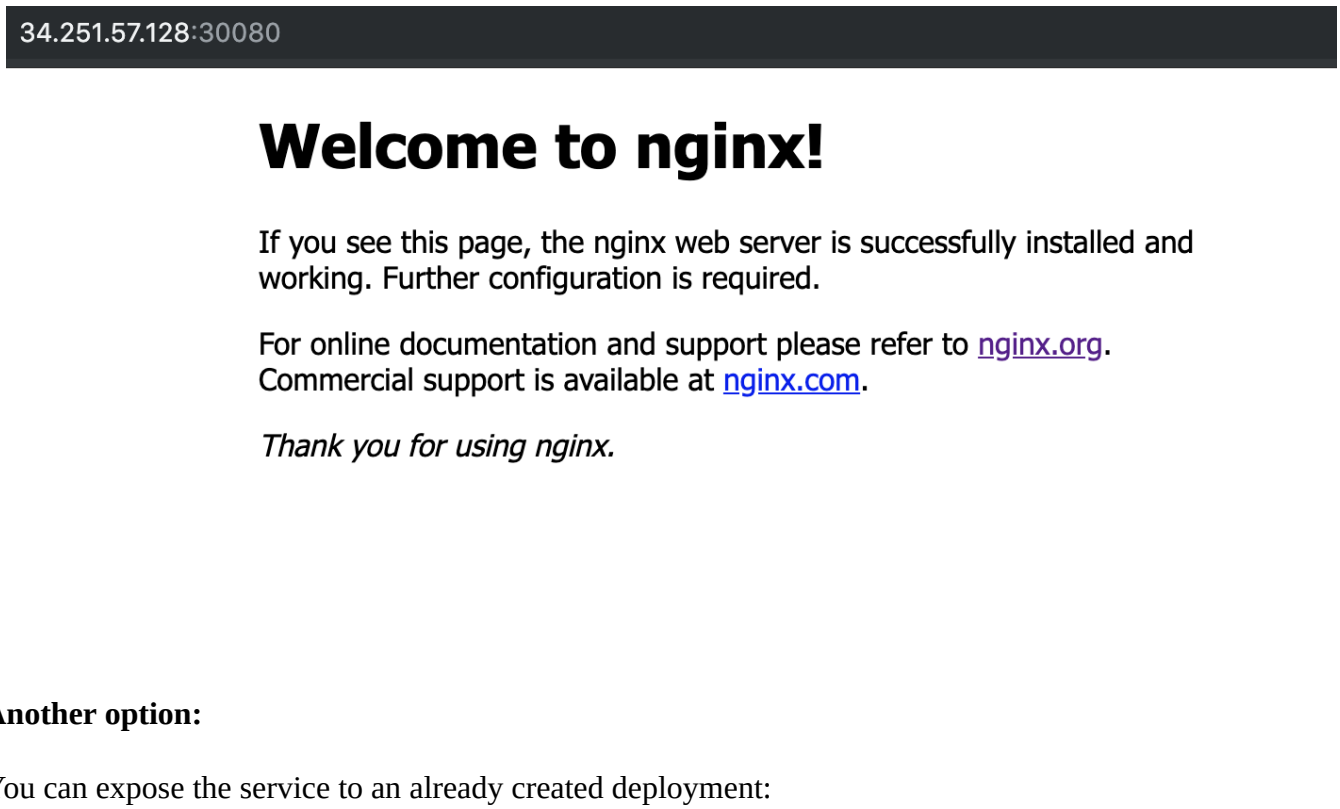
root@andrewjobson1c:~# **curl localhost:30080**

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Do a call externally. Use the IP of the master node + the nodeport, 300800 in this example:

On your browser:

http://34.251.57.128:30080/



Another option:

You can expose the service to an already created deployment:

root@ip-10-0-1-109:~# **kubectl expose deployment nginx --port 80 --type NodePort**
service/nginx exposed

More tests:
We have 2 nginx pods and we created the service: my-awsome-service

root@andrewjobson1c:~/k8s_files# **cat service-example.yml**
kind: Service
apiVersion: v1
metadata:
name: my-awesome-service
spec:
selector:
app: nginx
ports:
- protocol: TCP
port: 32768
targetPort: 80

See the IPs of all the nodes of the service: endpoints
root@andrewjobson1c:~/k8s_files# **kubectl describe service my-awesome-service**
Name: **my-awesome-service**
Namespace: default
Labels: <none>
Annotations: <none>
Selector: app=nginx
Type: ClusterIP
IP: **10.108.195.105**

Port: <unset> 32768/TCP
TargetPort: 80/TCP
Endpoints: 10.244.1.24:80,10.244.2.17:80
Session Affinity: None
Events: <none>

Check if nginx is running: Pick the port we configured when ran the service, 32768 in this example:

root@andrewjobson1c:~/k8s_files# **curl --head 10.108.195.105:32768**
HTTP/1.1 **200 OK**
Server: nginx/1.15.7
Date: Wed, 31 Jul 2019 14:29:19 GMT
Content-Type: text/html
Content-Length: 141
Last-Modified: Sat, 15 Dec 2018 04:13:27 GMT
Connection: keep-alive
ETag: "5c147f67-8d"
Accept-Ranges: bytes

Check the DNS from another container: because the pods can see the DNS

nslookup my-awsome-service

Will tell you the IP of the service, 10.108.195.105, and the whole domain my-awsome-service.default.svc.cluster.local

Lets scale 2 pods more: kubectl scale rs/frontend --replicas=4

Now we have 4 IPs
kubectl describe service my-awesome-service
[...]
Endpoints: 10.244.1.24:80,10.244.1.25:80,10.244.2.17:80 + 1 more...
[...]

Microservices:

Essentially microservices are an application architecture.

Until now we were using monolithic architecture, all the parts of the application are combined in one large executable file.

Microservices are small, independent services that work together to form a whole application.

Advantages:

- **Scalability:** If you need extra resources for a part of your application you can scale it for that specific part, not for the whole application like in monolithic applications.
- **Cleaner code:** The more small parts of the code, the cleaner tend to be and more easy to understand. Is easy to maintain because you can change code in one area without affecting the others.

- **Reliability:** If you have a bug in one area is less likely that the whole application is gonna be affected.
- **Variety of tools:** Different parts of the application can use different languages, frameworks and tools. The right tool can be used for every job.

K8s helps a lot to manage all this.

On the other hand microservices are quite complex to maintain.

Sample of microservice application: https://github.com/linuxacademy/robot-shop

Deploy microservices: Robot-shop

We are going to see how microservices works and how fast are the services deployed and running.

Download the robo-shop git repo that contains several ymls with deploys and services for each one of the applications needed to run it.

Mysql
mongodb
rabbitmq
nginx
java
nodejs
python
etc,

A lot of different technologies running separately and working over k8s that will bring the website up and working correctly.

1) Clone the repository:
cd ~/
git clone https://github.com/linuxacademy/robot-shop.git

2) Create a namespace: To put together all the services and be able to call them at once.

root@andrewjobson1c:~# **kubectl create namespace robot-shop**
namespace/robot-shop created

3) Deploy the app:
kubectl -n robot-shop create -f ~/robot-shop/K8s/descriptors/

Will run the ymls and will create the deploys and sevices.

Check how the containers are being created in live, like with tail -f

kubectl get pods -n robot-shop -w

Now you can access to the web application via browser: http://<ip_of_master_server>:30080 (port on web-service.yml)

4) Add an exta mongodb container:

Due the high number of users connecting to the web we need to scale another mongodb container:

kubectl edit deployment mongodb -n robot-shop

Change: replicas to 2

Check the status of the deployment:
kubectl get deployment mongodb -n robot-shop

Scale:

kubectl scale <?> nginx –replicas=5
Will add 5 nodes until the deployment nginx, so will modify the configuration and then always will have 5 replicas unless we change it.

Build HA kubernetes cluster:

Highly Available Topologies in Kubernetes

You can set up a HA k8s cluster:

- **Stacked control plane nodes:** This approach requires less infrastructure. The etcd members and control plane nodes are co-located.
- **External etcd cluster:** This approach requires more infrastructure. The control plane nodes and etcd members are separated.

All components on kubernetes can be replicated but only some of them can operate simultaneously.

Control manager and scheduler can only have 1 instance running at the same time, the rest will be on stand by.

*** The leader elect option** controls which master node is active and who is in stand by.

How to check who is the leader elect option:
The leader election mechanism is used by creating an end point resource, you can see it on the kube-scheduler yml:

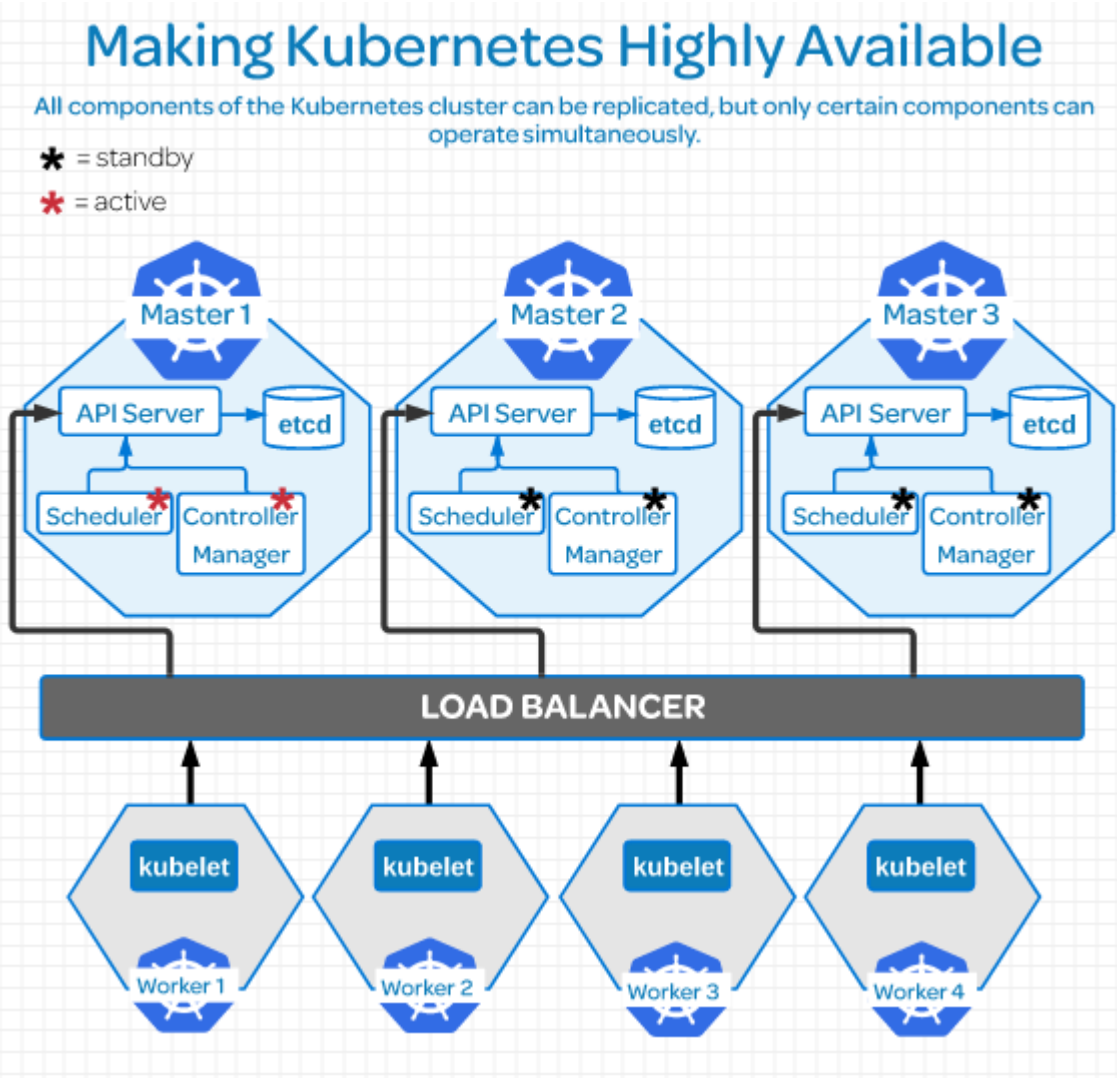
kubectl get endpoints **kube-scheduler** -n kube-system -o yaml
kubectl get endpoints **kube-controller-manage**r -n kube-system -o yaml

root@andrewjobson1c:~# kubectl get endpoints **kube-controller-manager** -n kube-system -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  annotations:
    control-plane.alpha.**kubernetes.io/leader**: '{"**holderIdentity**":"**andrewjobson1c**.mylabserver.com_147bf56d-abfb-11e9-8f0e-02ea1481d388","leaseDurationSeconds":15,"acquireTime":"2019-07-21T21:04:17Z","renewTime":"2019-07-21T22:44:48Z","leaderTransitions":0}'
  creationTimestamp: "2019-07-21T21:04:17Z"
  name: kube-controller-manager
  namespace: kube-system
  resourceVersion: "8939"
  selfLink: /api/v1/namespaces/kube-system/endpoints/kube-controller-manager
  uid: 1a656ae9-abfb-11e9-a69f-02ea1481d388

Once becoming the leader it must periodically update the resource, happening every 2 second by default.

Minimum Resources:

HA Cluster	Required nodes	CPU	RAM	Total
Stacked control plane nodes	3 Control Plane 3 Worker	2 each node	2 GB each node	6 nodes 12 CPU 12 GB Ram
External etcd cluster	3 Control Plane 3 etcd hosts 3 Worker	2 each node	2 GB each node	9 nodes 18 CPU 18 GB Ram



Replicate etcd:

****

****Due to the distributed nature of etcd it's instances can be in:

- **Stacked etcd topology:** Each control plane node creates a local etcd member, and the etcd member communicated only with the kube-apiserver of this node.
- **External etcd topology:** Etcd is external to the k8s cluster.

Etcd uses Raft Consensus Algorithm (Graphical explanation)which requires a majority in order to progress to the next state.

There must be more than half taking part on that state change. In order to have a majority you must have an ODD number of etcd instances therefore the minimum number of instances must be 3 to have a etcd external topology cluster.

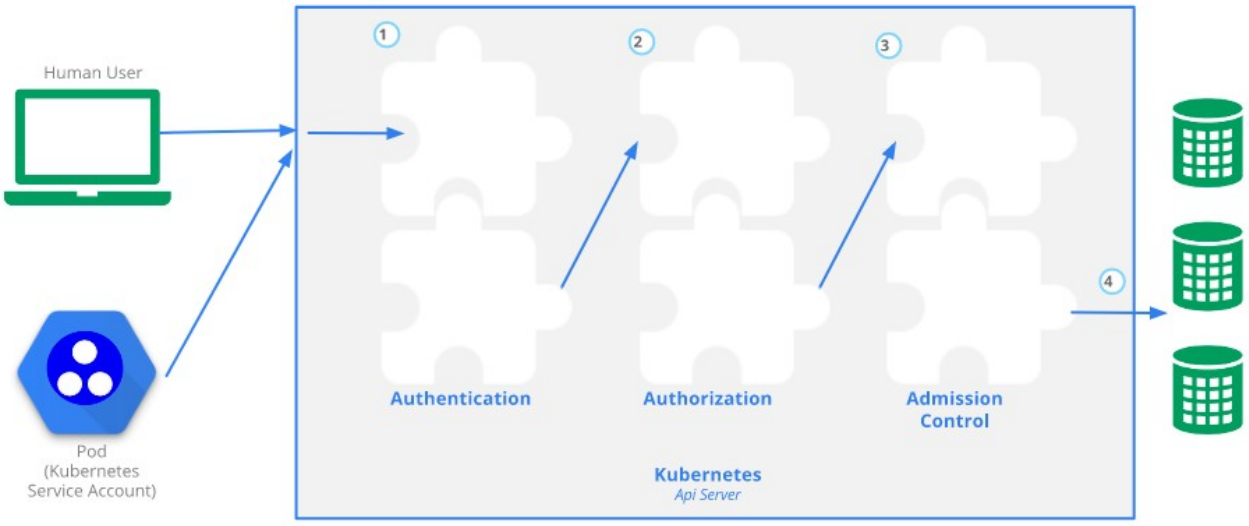
Having 2 etcd instances is worst than having one because this reason.

On larger dev environments most of the time you won't need more than 7 etcd instances.

Links:
Creating Highly Available Kubernetes Clusters with kubeadm
Operating a Highly Available etcd Cluster
Easy explanation of etcd consensus

Configuring Secure Cluster Communications:

Controlling Access to the Kubernetes API



Kubectl , command line tool: It's a translator to the k8s API

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

Who can use CRUD, only:

1. The client, like Users
2. PODs

Let's create a POD:

1. We make the kubectl command to create a POD
2. It sends a request to the API server via http POST request
3. Goes to Authentication
4. Authorization
5. Admission
6. Resource Validation
7. The state is stored on the persistent datastore etcd
8. API server returns a response.

Different plugins/modules: On each Authentication, authorization and Admission control phases there are multiple plugins allowing you find out different things about the request.

Authentication → Who is sending the request.
Authentication method can be determined by http heather or certificate.
Once found the request returns the username, user id and groups the client belongs to back to the API server.

Authentication modules include: Client Certificates, Password, and Plain Tokens, Bootstrap Tokens, and JWT Tokens (used for service accounts).

Authorization → Can this authenticated user perform the requested action on the requested resource?
Example: Can this user create a POD in this namespace?

Adminssion → If your request is Create, Modify or Delete a resource, like a pod, it's sent to Admission Control.
If the request is to READ data, the Admission Control steep is skipped and is sent to ETCD

In admission control the requests goes through all the plugins allowing the plugins to modify the resource for different reasons.

Authentication - Authorization - Admission - Review

Resource Validation → The API server validates the object.

ETCD → The API server stores the object in etcd and returns a response. This is the object store.

Self signed certificate: Created when we create the cluster and is stored in **$USER/.kube/config**
It contains:

- certificate-authority-data
- Master server address: server: https://172.31.127.122:6443
- The Cluster: cluster: kubernetes
- The client certificate: client-certificate-data
- Client Key: client-key-data

This certificate contains the root certificate for the API server's certificate, which is used in place of the system default root certificate.

If the system have multiple users, then the creator needs to share the certificate with other users.

All requests come from two different sources: (both go through the same auth process we saw above)

- The client, like a user
- A POD

Request from Users: **RBAC** (Role Based Access Control) this authorization module is used to prevent unauthorized users to modify the cluster state.

Manage RBAC: You manage RBAC by creating four resources in two groups:

Roles and cluster roles: What can be done?
Role: Namespaces resources
Cluster Role: Cluster level resources

Role Binding and cluster role binding: Who can do it?
Role Binding: Namespace Resources
Cluster Role Binding: Cluster lever resources

Examples:

Role: Create a Role named "pod-reader" that allows user to perform "get", "watch" and "list" on pods:
kubectl create role pod-reader --verb=get --verb=list --verb=watch –resource=pods

Cluster Role: Create a ClusterRole named "pod-reader" that allows user to perform "get", "watch" and "list" on pods:
kubectl create clusterrole pod-reader --verb=get,list,watch –resource=pods

Role Binding: Within the namespace "acme", grant the permissions in the admin ClusterRole to a user named "bob":
kubectl create rolebinding bob-admin-binding --clusterrole=admin --user=bob --namespace=acme

Cluster Role Binding: Across the entire cluster, grant the permissions in the cluster-admin ClusterRole to a user named "root":
kubectl create clusterrolebinding root-cluster-admin-binding --clusterrole=cluster-admin --user=root

Requests from PODs: **Services Accounts:** Is how the POD authenticates to the API server

Configure Service Accounts for Pods

Represents the identity of the APP running on the POD and a token file holds the service account authentication token.

We can view this token file by going over to our POD by a shell and we can cat out the token:

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

When the app uses this token to connect the API server, the plugin authenticates the service account and passes the services account back to the API server core.

Service account by default: If is not specified when creating the pod, the service account applied will be the default one.

This means that if you create the pod into another namespace, the service account will have permissions only for that namespace and not the rest, default namespace included, example:

Get the service accounts in the cluster:

```
root@andrewjobson1c:~# kubectl get serviceaccounts
NAME      SECRETS   AGE
default       1         43h
```

If the service account is not specified on the PODs manifest then the default service account will be applied.

View service account token:

```
root@andrewjobson1c:~# kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-fsfdg   kubernetes.io/service-account-token   3      43h
```

End to End Tests, e2e:

E2e tasks are good for two main reasons:

1. Ensures that performance and response of the app running in a pod is not compromised due the performance in a cluster.
2. Ensures that the pods are in good health, there are no resource constraints or problem with deploying resources.

Tools:

Kubetest: Interface for launching and running e2e tests.
Ju ju deploying cluster: Cloud validation and testing
Each cloud provider have his own version of a testing suite for their hosted kubernetes services.

Tests:

Run a simple nginx deployment:

```
root@andrewjobson1c:~# kubectl run nginx --image=nginx
kubectl run --generator=deployment/apps.v1 is DEPRECATED and will be removed in a future version. Use kubectl run --generator=run-pod/v1 or kubectl create instead.
deployment.apps/nginx created
```

View the deployments in your cluster:

root@andrewjobson1c:~# **kubectl get deployments**

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
busybox	1/1	1	1	33h
nginx	1/1	1	1	31s
node-hello	1/1	1	1	33h

View the pods in the cluster:

kubectl get pods
kubectl get pods --all-namespaces

Use port forwarding to access a pod directly:

root@andrewjobson1c:~# **kubectl port-forward nginx-7cdbd8cdc9-q9x78 8081:80**
Forwarding from 127.0.0.1:8081 -> 80
Forwarding from [::1]:8081 -> 80

root@andrewjobson1c:~# **curl --head http://127.0.0.1:8081**
HTTP/1.1 200 OK
Server: nginx/1.17.2
Date: Wed, 24 Jul 2019 22:17:23 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 23 Jul 2019 11:45:37 GMT
Connection: keep-alive
ETag: "5d36f361-264"
Accept-Ranges: bytes

Get Logs from a pod:

^Croot@andrewjobson1c:~# **kubectl logs pod/nginx-7cdbd8cdc9-q9x78**
127.0.0.1 - - [24/Jul/2019:22:17:23 +0000] "HEAD / HTTP/1.1" 200 0 "-" "curl/7.47.0" "-"

ReplicaSets: Set of pods that all share the same labels.

Are missing some of the features that we need to truly have HA and escalability. To do so we should use Deployments.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
    tier: frontend
  matchExpressions:
    - {key: tier, operator: In, values: [frontend]}
 template:
  metadata:
    labels:
      app: nginx
      tier: frontend
  spec:
    containers:
    - name: nginx
      image: darealmc/nginx-k8s:v1
      ports:
      - containerPort: 80
```

Create one: kubectl create -f replicas-example.yml

See the info or the replica pods: kubectl describe **rs/**frontend

See the IP of one pod:

kubectl get pods

kubectl describe pod frontend-9jrcw

```
[…]
IP:             10.244.2.15
Controlled By:  ReplicaSet/frontend
[...]
```

Check if nginx is running:

curl 10.244.2.15

```
<head>
  <h1> This is the Correct container! Congratulations!</h1>
</head>
<body>
  <img src='k8slogo.png' height="550" width="550">
</body>
```

curl --head 10.244.2.15

```
HTTP/1.1 200 OK
Server: nginx/1.15.7
Date: Wed, 31 Jul 2019 13:25:47 GMT
Content-Type: text/html
Content-Length: 141
Last-Modified: Sat, 15 Dec 2018 04:13:27 GMT
Connection: keep-alive
ETag: "5c147f67-8d"
Accept-Ranges: bytes
```

Scale to 4 replicas:

root@andrewjobson1c:~# **kubectl scale rs/frontend --replicas=4**

replicaset.extensions/frontend scaled

Reduce to 1 replica:

root@andrewjobson1c:~# **kubectl scale rs/frontend --replicas=1**

replicaset.extensions/frontend scaled

Delete the ReplicaSet:

root@andrewjobson1c:~# **kubectl delete rs/frontend**
replicaset.extensions "frontend" deleted

Logs:

1 - Clone the metrics server:
git clone https://github.com/linuxacademy/metrics-server

2 – Deploy the directory of the repository:
kubectl apply -f ~/metrics-server/deploy/1.8+/

3 - Access the metrics API: Makes sure that the API is responding
kubectl get --raw /apis/metrics.k8s.io/

4 – Check status nodes and pods:
kubectl top nodes
kubectl top pods --all-namespaces

Get the CPU and memory of pods with a label selector:
kubectl top pod -l run=<pod_name>

Get the CPU and memory of a specific pod:
kubectl top pod <pod_name>

Get the CPU and memory of the containers inside the pod:
kubectl top pods <pod_name> --containers

Upgrading the Kubernetes Cluster

The CKA curriculum refers to the API Server version, v1.14.1.
So you have to look for the kubelet version (server version), 1.13.8 in this example.
kubectl version --short

The process creates new images for the components: kube-apiserver, kube-controller-manager, kube-scheduler, etcd

And will swap to them once are working to minimize the downtime.

Check versions:

Kubectl version: We need to know the Server Version
root@andrewjobson1c:~# **kubectl version --short**
Client Version: **v1.13.5**

Server Version: **v1.13.8 → API server that we need to upgrade**

Kubelet version:

root@andrewjobson1c:~# **kubectl get nodes**

NAME	STATUS	ROLES	AGE	VERSION
andrewjobson1c.mylabserver.com	Ready	master	10d	**v1.13.5**
andrewjobson2c.mylabserver.com	Ready	<none>	10d	**v1.13.5**
andrewjobson3c.mylabserver.com	Ready	<none>	10d	**v1.13.5**

All versions on all servers on the cluster:

root@andrewjobson1c:~# **kubectl describe nodes |grep -i version**

```
 Kernel Version:           4.4.0-1088-aws
 Container Runtime Version:  docker://18.6.1
 Kubelet Version:          v1.13.5
 Kube-Proxy Version:       v1.13.5
 Kernel Version:           4.4.0-1088-aws
 Container Runtime Version:  docker://18.6.1
 Kubelet Version:          v1.13.5
 Kube-Proxy Version:       v1.13.5
 Kernel Version:           4.4.0-1088-aws
 Container Runtime Version:  docker://18.6.1
 Kubelet Version:          v1.13.5
 Kube-Proxy Version:       v1.13.5
```

Controller-Manager version:

kubectl get pods -n kube-system kube-controller-manager-andrewjobson1c.mylabserver.com -o yaml

[…]
image: k8s.gcr.io/kube-controller-manager**:v1.13.8**
[…]

Upgrade versions:

On the Master node:

1 – Unhold kubeadm → to allow the update
dpkg --get-selections |grep kube → to see what is holded
kubeadm hold

root@andrewjobson1c:~# **sudo apt-mark unhold kubeadm**
Canceled hold on kubeadm.

dpkg --get-selections |grep kube
kubeadm install

2 – Install kubeadm:
sudo apt install -y kubeadm=1.14.1-00

3 – Hold kubeadm again:
root@andrewjobson1c:~# **apt-mark hold kubeadm**
kubeadm set on hold.

4 – Confirm the new version:
root@andrewjobson1c:~# **kubeadm version**

kubeadm version: &version.Info{Major:"1", Minor:"14", **GitVersion:"v1.14.1"**, GitCommit:"b7394102d6ef778017f2ca4046abbaa23b88c290", GitTreeState:"clean", BuildDate:"2019-04-08T17:08:49Z", GoVersion:"go1.12.1", Compiler:"gc", Platform:"linux/amd64"}

5 – Upgrade the rest of the components: Will show you what can we upgrade without doing anything else:

sudo kubeadm upgrade plan

[…]

COMPONENT	CURRENT	AVAILABLE
API Server	v1.13.8	v1.14.4
Controller Manager	v1.13.8	v1.14.4
Scheduler	v1.13.8	v1.14.4
Kube Proxy	v1.13.8	v1.14.4
CoreDNS	1.2.6	1.3.1
Etcd	3.2.24	3.3.10

You can now apply the upgrade by executing the following command:
kubeadm upgrade apply v1.14.4

root@andrewjobson1c:~# **kubeadm upgrade apply v1.14.1**
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -oyaml'
[upgrade/version] **You have chosen to change the cluster version to "v1.14.1"**
[upgrade/versions] Cluster version: v1.13.8
[upgrade/versions] kubeadm version: v1.14.1
[upgrade/confirm] **Are you sure you want to proceed with the upgrade? [y/N]:**

[…]
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.14.1". Enjoy!

6 – Done, check versions:
root@andrewjobson1c:~# **kubectl version --short**
Client Version: v1.13.5
Server Version: **v1.14.1**

7 – Unhold , Upgrade and hold back kubectl:
apt-mark unhold kubectl
apt install -y kubectl=1.14.1-00
apt-mark hold kubectl

Now client and server have the same version:
root@andrewjobson1c:~# **kubectl version --short**
Client Version: v1.14.1
Server Version: v1.14.1

8 – Kubelet now:
apt-mark unhold kubelet
apt install -y kubelet=1.14.1-00
apt-mark hold kubelet

Now we have the master node updated:
root@andrewjobson1c:~# **kubectl get nodes**

NAME	STATUS	ROLES	AGE	VERSION
andrewjobson1c.mylabserver.com	Ready	master	10d	**v1.14.1**
andrewjobson2c.mylabserver.com	Ready	<none>	10d	v1.13.5
andrewjobson3c.mylabserver.com	Ready	<none>	10d	v1.13.5

9 – Now do the same to the worker nodes
Skip the master server parts, points 5 and 6.

Operating System Upgrades within a Kubernetes Cluster:

Remove a worker node for Maintenance:

1- Remove the pods of the node avoiding new ones to be scheduled there:
kubectl draing <node_name> --ignore-daemonsets

2- Watch as the node change status:
kubectl get nodes -w
You will see the node status as Ready, **SchedulingDisabled**

3- Schedule pods to the node after the maintenance:
kubectl uncordon <node_name>

Totally decommission a node:

1- Drain the node:
kubectl draing <node_name> --ignore-daemonsets

2- Delete the node from the cluster:
kubectl delete node <node_name>

3- Install a new server:
Install everything, docker,kubeadm,kubectl and kubelet

4- Join it to the cluster:
On the master server, generate the token: sudo kubeadm token generate

On the master server, generate the join command:
sudo kubeadm token create <paste the output of the previous command> --ttl 2h --print-join-command

On the new worker node, copy the string created on the previous command and paste it on the new worker node.

5- Verify that the node has been added correctly: kubectl get nodes

---

Backing Up and Restoring a Kubernetes Cluster:

Backing up the etcd Store
etcd Disaster Recovery Examples

1- Get the etcd binaries:
wget https://github.com/etcd-io/etcd/releases/download/v3.3.12/etcd-v3.3.12-linux-amd64.tar.gz

2- Unzip the compressed binaries:
tar xvf etcd-v3.3.12-linux-amd64.tar.gz

3- Move the files into /usr/local/bin:
sudo mv etcd-v3.3.12-linux-amd64/etcd* /usr/local/bin

4- Take a snapshot of the etcd datastore using etcdctl:
root@andrewjobson1c:~# 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
Snapshot saved at snapshot.db

5- View the help page for etcdctl:
ETCDCTL_API=3 etcdctl --help

6- View that the snapshot was successful:
root@andrewjobson1c:~# ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshot.db

```
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| 886df69b |   518353 |       1021 |     4.1 MB |
+----------+----------+------------+------------+
```

7- Store the snapshot on the certificates folder and do a backup:
/etc/kubernetes/pki/etcd/
sudo tar -zcvf etcd.tar.gz etcd

8- Copy the etcd directory outside of the server

Restore:
To do the restore operation you need the datastore snapshot and all the certificates files.

When you do the restore it creates and entirely new etc directory.

No matter how many nodes do you have in your cluster you must restore always the same snapshot.

The restore operation overwrites the memory ID and the cluster ID so no longer identify with the original cluster at all. The restore operation create a new entire cluster and replaces the etc keyspaces from that backup. You totally wipe and way the previous cluster.

If the node that is down is completely lost, the new node will need to have the same IP as the original.

Security: https://github.com/kabachook/k8s-security?utm_sq=g7fg3gt0uh

---

Cluster Communications:

Pods and Node Networking:
https://linuxacademy.com/cp/courses/lesson/course/4018/lesson/1/module/327

Networking on k8s uses the old linux network namespaces. Another (better) explanation.

Namespaces: Cgroups are a metering and limiting mechanism, they control how much resources (CPU, memory) you can use. Namespaces limit what you can see.

Linux kernel provides 6 types of namespaces: pid, net, mnt, uts,ipc and user.

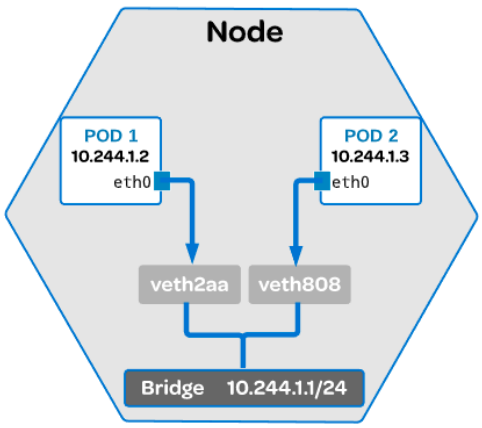
For instance, a process inside a pid namespace only sees processes in the same namespace. Thanks to the mnt namespace it's possible to attach a process to it's own filesystem.

So network (net) namespaces provide a brand-new network stack for all processes within the namespace.

Network namespaces: Contains its own network resources: interfaces, routing tables, etc.

Creates 2 interfaces, one on the node side (veth) and another in the POD side (will be renamed as eth0).

Networking within a Node: Minerva diagram.



Each POD has an IP address associated with it. It receives this IP address from the virtual ethernet (veth) interface pair. One of this veth is given to the POD and renamed as eth0.

Bridge: Common linux ethernet bridge. It discovers the destination using an ARP request. Used to communicate on other veth network on the same node.

Commands:

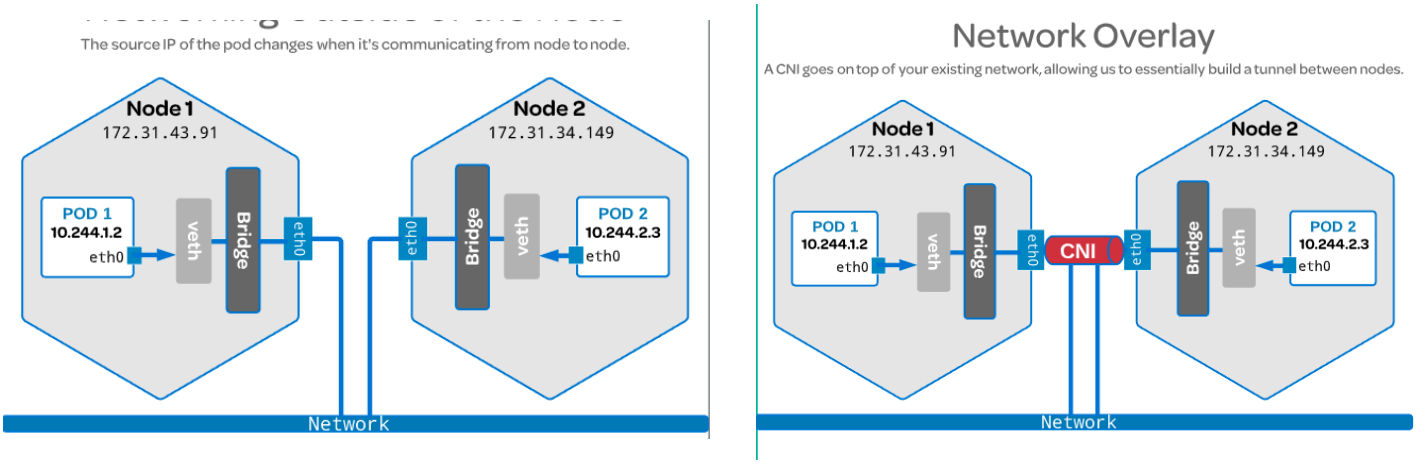
View which node is running on: kubectl get pods -o wide
See the running containers in that node: sudo docker ps (pick the container id needed)
Get the container PID: sudo docker inspect --format '{{ .State.Pid }}' (container id)

Get the IP address: nsenter -r (container PID) → you will see eth0@if6 or something similar. In this example if6 is the interface 6 of the pod, so to know which one is do an ifconfig on the node and count until the 6th one, you will see that is a veth interface seeing how are they linked.

Networking Outside of the Node and CNI: Minerva diagram.

https://linuxacademy.com/cp/courses/lesson/course/4018/lesson/2/module/327

The CNI is managing the communications between nodes.



Once the CNI is installed it installs a network agent on each node, which ties the CNI interface. Encapsulates the packets changing the source and destination from one node to another.

There is a mapping in the userspace where you can program all the POD IP addresses to the node IP addresses and when the packet reaches the other node is decapsulated and when reaches the bridge is moved to the proper POD.

The CNI is installed via plugin, is not a native k8s solution, like Flannel as CNI for instance.

When you run the below command you will install flannel as CNI:
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

If you take a look to the yml you will see that is installing a lot of stuff, like cluster roles, cluster role binding, services, etc, etc. This kind of yml is called **multi resouce yml.**
To use the CNI you must notify the kubelet. Kubelet needs to know that the CNI is being used by setting the network plugging flag equal to the CNI.

We do that when we install the k8s cluster. Each CNI have a different command, flannel has this one:

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

Service Networking: Linux academy. Minerva

Communicate outside of the cluster. When someone is trying to interact with your application or the app is trying to reach something outside of the cluster.

Pods are coming and going, or you can have problems with a single pod, then is easy to remove this failing one and change it for a brand new.

The cluster keep track of all this and determine where to send traffic coming in using services. Avoiding to have a down time because a pod have been moved or deleted because have a new IP address or so.

A service provides just one virtual interface, behind the pods can be moved around, deleted, etc and from the outside everything is going to this virtual interface.

When a service is created in an API server, the API server notifies all kube-proxy agents about it. Kube-proxy in this case is used to keep track of the endpoints and update IP tables so the traffic can be routed accordingly.

Endpoint: Object in the API server created automatically with the service, keeps a cache of the IP addresses of the pods in that service. Is a correlation between IP address plus port.

Commands:

View the spec of the services: kubectl get services -o yaml (you can see the cluster IP) If you ping the cluster IP won't work because a service is a logical grouping of IP and port pair.

View the services in the cluster: kubectl get services

View IP tables rules: sudo iptables-save |grep KUBE |grep nginx

Output example:

-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.109.185.62/32 -p tcp -m comment --comment "default/nginx: cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ

-s 10.244.0.0/16 : source, take anything from that range
-d 10.109.185.62/32: destination, send it to the IP of the service.

Get endpoints: kubetcl get endpoints

Load Balancer and Ingress Rules: Linux Academy Minerva

Load Balancer: Extension of the NodePort type of service.
It's balancing the external access to the app through all the nodes that contain it. You need a public IP.

You can create it as a Service, you have to specify on "spec" the "type" LoadBalancer instead of a node type port service. We don't specify a node port, k8s will choose this for us.

Ingress Rules: Like a load balancer but you can access multiple services with just a single IP address, with the lb you have only one external IP for each service.

COREDNS: Minerva. Linux academy.

K8s used to use Kube-dns, now is using Coredns as a native K8s DNS solution.

Have backwards compatibility with kube-dns, uses better the memory to avoid buffer overruns, supports TLS or Dot over DNS, easy to integrate with etcd and cloud benders to pull authoritative data into coredns.

Plugin and simplified architecture. Allows dns entries without touching the service discovery part.

We can see 2 pods running in the master node as a deployments.

There is a service performing loadbalancing for the dns server.

Commands: To troubleshooting DNS stuff you can create a busybox to run commands over it.

See coredns pods: kubectl get pods -n kube-system
See coredns deployments: kubectl get deployments -n kube-system
Get KUBE-DNS services: kubectl get services -n kube-system (you get kube-dns because it's kept because it's a legacy service).

Create busybox pod: kubectl create -f busybox.yaml
See pod resolve.conf: kubectl exec -t busybox – cat /resolve.conf (you will see in the "search" category different cluster.local entries which means that all queries for these dns entries will resolve)
Do nslookup: kubectl exec -t busybox -- nslookup kubernetes (this will work as explained in the previous command or use the whole hostname of the pod, service, etc.)

Get logs: kubectl logs coredns-34354fsfe-sdxw -n kube-system

Headless services: another example, and another more clear. Service without a clusterIP, will respond with a set of IPs instead of just one.

You have to create a service and you have to set the clusterIP policy to "None".
Specify the app that you want to use the service.

DNS policies on pods: You can configure specific dns policies for a pod.
The default is cluster first which inherit the name resolution from the node the pod is running on.

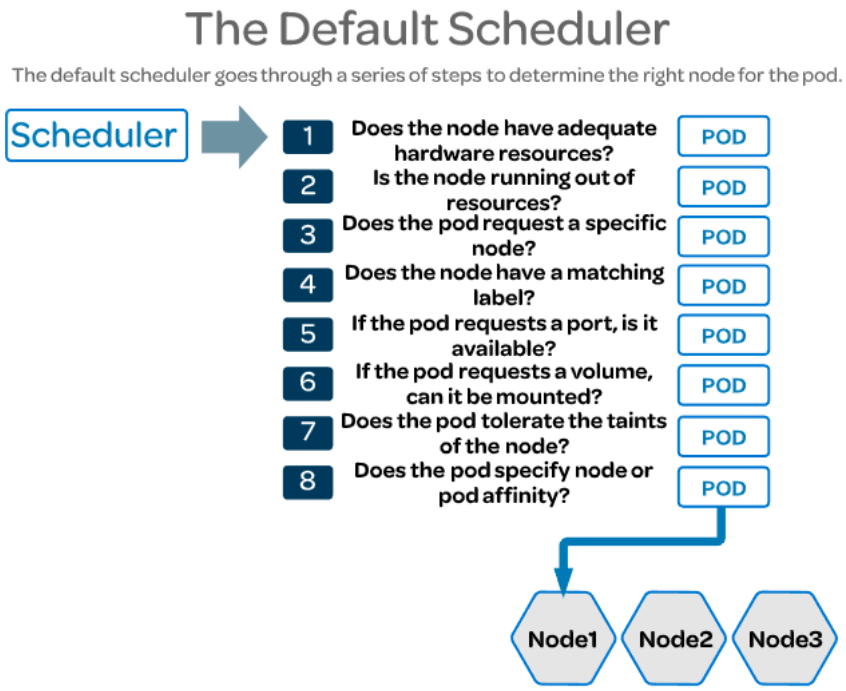
So set spec.dnsPolicy: to “None” and specify the spec.dnsConfig:nameservers , spec.dnsConfig:searches and spec.dnsConfig:options to the ones that you need.

Once deployed that pod we can see in it’s /etc/resolve.conf the information added in the nameservers and search selectors before.

Pod Scheduling

Linux academy Minerva K8s documentation
What is this: To send the pod to the best node based on cpu, ram, disk, etc.

Default scheduler: The system does this automatically.



The scheduler does this automatically, will do a reasonable placement (e.g. spread your pods across nodes, not place the pod on a node with insufficient free resources, etc.). You can pod schedule manually too.

Why to manually schedule: You may want to schedule a pod to a particular node to save resources, or because the node have ssd, etc. There are many more possibilities as to why you would want to create your own scheduling rules.
Example: I have 2 availability zones in two different regions, AZ1 and AZ2 In AZ1 I have dedicated and shared servers. I want to send the pods to AZ1 and to the dedicated servers. Scheduling it manually I can do it.

Default scheduler: The system does this automatically.
If after all the checks scheduler have found more than one node he will select one using a round robin fashion.

nodeSelector: Used on Pods. Is the simplest recommended form to manually assign a pod to a node. Field of Pod.Spec that specifies a key-value pair. You have to create a label in the node and then you have to specify the nodeSelector in your Pod configuration:
Example:
Add the label disktype=ssd to the node kubernetes-foo…….
kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal **disktype=ssd**

Then add the nodeSelector in your pod yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Affinity and anti-Affinity: Is like nodeSelector but with more options, like language more expressive, indicate soft or hard preference,etc.

The key enhancements are:

1. The affinity/anti-affinity language is more expressive. The language offers more matching rules besides exact matches created with a logical AND operation;

2. You can indicate that the rule is "soft"/"preference" rather than a hard requirement, so if the scheduler can't satisfy it, the pod will still be scheduled;

3. You can constrain against labels on other pods running on the node (or other topological domain), rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located

The affinity feature consists of two types of affinity: "node affinity" and "inter-pod affinity/anti-affinity".

Node affinity is like the existing nodeSelector (but with the first two benefits listed above).

Inter-pod affinity/anti-affinity constrains against pod labels rather than node labels, as described in the third item listed above, in addition to having the first and second properties listed above.

Node affinity: Similar to nodeSelector, have to types:

requiredDuringSchedulingIgnoredDuringExecution: Hard. The rules MUST met it. Is like nodeSelector but with more expressive syntax.

preferredDuringSchedulingIgnoredDuringExecution: Soft. Specify preferences that the scheduler will try to met but will not guarantee.

Inter-pod affinity and anti-affinity: It looks for the labels on the pods already running on the node instead of labels on the nodes.

Commands:

Check the nodes: kubectl get nodes

Label a node:
kubectl label nodes <node-name> <label-key>=<label-value>
kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd

Check labels: kubectl get nodes --show-labels

Full list of labels for a node: kubectl describe node "nodename"

Label node 1 as being located in availability zone 1:
kubectl label node chadcrowell1c.mylabserver.com availability-zone=**zone1**

Label node 2 as being located in availability zone 2:
kubectl label node chadcrowell2c.mylabserver.com availability-zone=**zone2**

Label node 1 as dedicated infrastructure:
kubectl label node chadcrowell1c.mylabserver.com share-type=**dedicated**

Label node 2 as shared infrastructure:
kubectl label node chadcrowell2c.mylabserver.com share-type=**shared**

This is the yaml for the deployment, pref-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pref
spec:
  selector:
    matchLabels:
      app: pref
  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

Configure Multiple Schedulers:

Until now we were using the scheduler that is by default in the cluster, but we can create our own. Then you can replace my scheduler with the built in one or make both to work together.

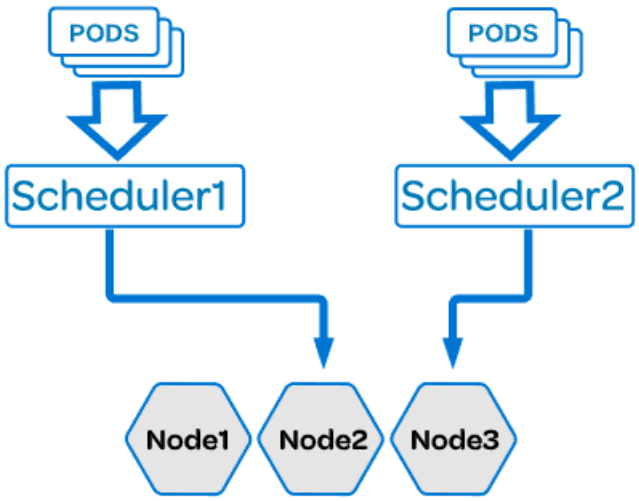
To do so, you have to:

- Create the scheduler (my-scheduler.yaml), copy it from here and modify the image to use yours, docker build required.
- Create a new RBAC stuff: role, role binding, cluster role and cluster role binding pointing to the new scheduler.
- Edit the exixting kube-scheduler add the new scheduler in rules.apiGroups.resouceNames.
- Create the new scheduler.
- Create the pods pointing to the new scheduler,

See whole configuration in /home/joaquim/Documents/repos/configure_multiple_schedulers

Multiple Schedulers

Kubernetes allows you to use multiple schedulers with different rules simultaneously.



Command Reference

Kubernetes Documentation

`kubectl create -f role.yaml`	Create the Role
`kubectl create -f rolebinding.yaml`	Create the RoleBinding
`kubectl create -f clusterrole.yaml`	Create the ClusterRole
`kubectl create -f clusterrolebinding.yaml`	Create the ClusterRoleBinding
`kubectl edit clusterrole system:kube-scheduler`	Edit the existing ClusterRole
`kubectl create -f my-scheduler.yaml`	Create the new scheduler
`kubectl get pods -n kube-system`	View the kube-system pods
`cat pod1/2/3.yaml`	View the pods for each scheduler
`kubectl create -f pod1/2/3.yaml`	Create the pods for each scheduler
`kubectl get pods -o wide`	View the pods on each node

Scheduling Pods with Resource Limits and Label Selectors:

Linux Academy
Minerva
Udemy

Taints and Tolerations:

Taints: → on the nodes
Tells if the node can be scheduled or not.

The master node can not but the kube-system pods have a Toleration that allows them to be moved to the master node

Tolerations: → on the pods
Allow you to tolerate a taint, for example you don't want to schedule nginx or other app pods to the master node but the kube-system pods must be moved there, so the k-system pods will have a toleration for a NoSchedule taint.

Commands:

See nodes/pods capacity,etc:
kubectl describe node → will; show all of them
kubectl describe node name_of_the_node

kubectl describe pod → will; show all of them
kubectl describe pod name_of_the_pod

Look at Capacity and Allocatable sections.

Check in the master node for the taints. You can see

kubectl describe node node1
[...]
Taints: node-role.kubernetes.io/master:**NoSchedule** --→ this is by default.
node.kubernetes.io/disk-pressure:NoSchedule
Unschedulable: false
[...]

Checking a worker node:
kubectl describe node node2
[…]
Taints: node.kubernetes.io/disk-pressure:NoSchedule
Unschedulable: false
[...]

Checking a kube-system POD:
kubectl get pod kube-proxy-k7g85 -n kube-system -o yaml |less

- effect: NoSchedule → so this means that you can jump this restriction in the node.
  key: node.kubernetes.io/unschedulable
This pod will tolerate a node that is unschedulable.

CPU and memory requests:

You can specify the minimum amount of cpu and ram that a pod will need. Then the scheduler can move the pod to the node that have these resources.

The scheduler doesn't looks for every individual resource to determine the best node.
Instead it uses the sum of resources requested by existing pods deployed on that node.

This is because the pod may not be using the full amount of resources requested at any given time.

Here you can see different limits in spec.containers.image.resources

```
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
```

```
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
```

The **m** in 800m and 1000m means milicpu.

Hard limits: You can specify the limit of memory a pod can use.

Is not specified a pod can use the whole memory of the node. To avoid this we can apply limits.
This is not happening with the CPU that won't let the pod to consume all of it.

```
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
```

Use the top command: This will show you the resources from the node, not the container.

kubectl exec -it POD_NAME top
kubectl exec -it busybox top

Modify memory limit:
Let's say we have a pod called elephant that have a 10 Mi limit and we want to increase it to 20 Mi.

1st Generate the pod definition file from the existing elephant pod:
kubectl get pod elephant -o yaml > pod.yaml
2nd Edit pod.yaml to increase the memory limit.
3rd Delete the current elephant pod with the old limit: kubectl delete pod elephant
4th Create the new elephant pod with the new mem limit: kubectl create -f pod.yaml

DaemonSets and Manually Scheduled Pods:

Minerva
DaemonSet

Like replicasets, it helps you to deploy multiple instances of pods but it runs only one copy of the pod in each node.
When a new node is added to the cluster, a replica of the pod is automatically added to that node.
And if a node is removed from the cluster, the replica of the pod in that node will be deleted.

Deamonsets ensures that one copy of the pod is always present in all nodes of the cluster.

Why to use it:
For instance if you want to deploy a monitoring solution or a log collector for each node in the cluster.

Kube-proxy and the networking solution, like flannel, weave-net, etc, are daemon sets because have to run one instance on each node.

Create a daemon-set: Is like to create a replicaset, it only varies in the kind, which have to be DaemonSet instead of ReplicaSet.

Commands:

Get daemonsets: kubectl get daemonsets
kubectl get daemonsets -n kube-system
kubectl get ds -A →all namespaces

Describe it: kubectl describe daemonset name_of_it
kubectl describe ds name_of_it

Different ways to create a pod:

kubectl run: Deploys a docker container by creating a pod straight away. You have to specify the name the pod will have and the image to pull, which will be downloaded from the docker hub repository.

kubectl run nginx --image=nginx
kubectl run redis --image=redis
kubectl run my-jenkins --image=jenkins

Do kubectl run but not create the pod, send the output to a yaml file:
kubectl run pod_name –image=my_image **--dry-run=client -o yaml** > my_file.yaml

kubectl run redis --image=redis **--dry-run=client -o yaml** > redis.yaml

This will create this file:
cat redis.yaml

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: redis
  name: redis
spec:
  containers:
  - image: redis
    name: redis
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

Then you can run this yaml to create the pod: kubectl create -f redis.yaml

Manually creating a yaml: You can use .yaml .ymnl or .json files to do it.

All yaml have to have 4 mandatory fields:

apiVersion: Version of the k8s api you are using to create objects. Depending on the object you want to create you have to use the right version. It’s data is a String.

Kind:	**Version:**
POD	v1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1

kind: Type of object you want to create, could be PODs, deployments, etc, etc. It's data is a String.

metadata: Data about the object, it's name, labels, etc. It's data is a Dictionary. Everything under the metadata have to be placed a little bit to the right, names, labels, etc are children of metadata.
The spaces have to be the same between siblings.

Example:
metadata:
  name: myapp-pod → string (str)
  labels: → dictionary (dict) → you can add any label, type,location, etc.
    app: myapp

metadata is parent of name and labels.
name and labels are siblings and children of metadata
app is children of labels

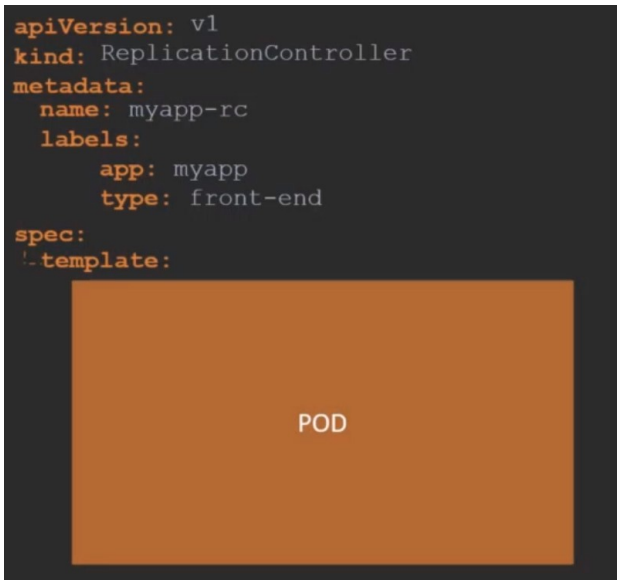
spec: Info about the containers, how much do we need, where to find the images,etc.

spec:
  containers: → List (lst)/Array
    - name: nginx-container → the dash (-) indicates that this is the first item in the list.
      image nginx

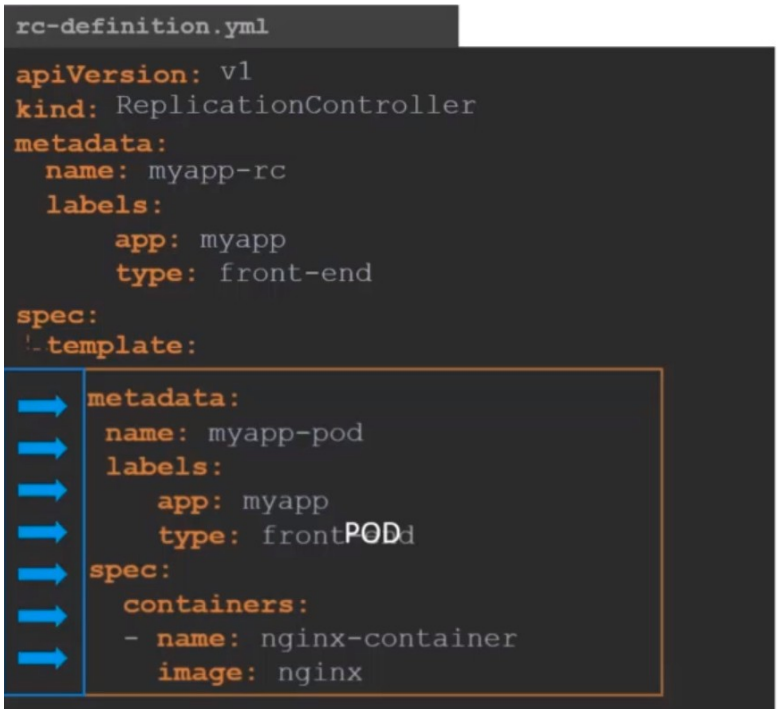
    - name: backend-container → you can specify more containers
      image: redis

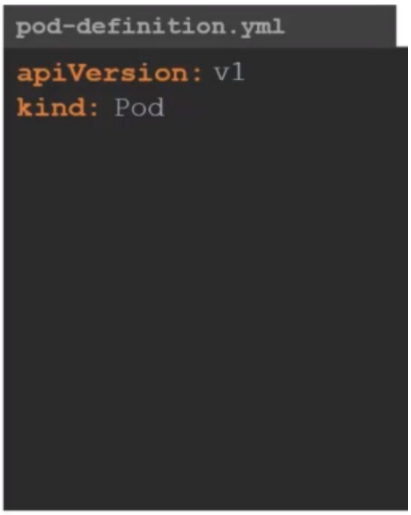
Replication Controllers: Is the older technology that is being replaced by the replicaSet.
The RC creates multiples instances of a pod, so in *spec* (specification) we create *template* section under spec to provide a pod template to be use by the RC to create replicas.

To define the pod template we can paste the metadata and spec info from a previous pod like in this example for instance. Skipping the apiversion and kind fields.



The *metadata* and *spec* fields coming from the pod must have the proper space from the left to make them childrens of spec.template, like we can see below.





Here we have nested two definition files together. The RC being the parent and the pod definition being the child.

Now we have to add the ***replicas*** property to the spec of the RC to indicate how many replicas do we need for the RC.

The final file looks like this. Now we can deploy the RC: kubectl create -f rc-definition.yaml
This will create the replicationcontroller “myapp-rc”

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
 template:
    metadata:
     name: myapp-pod
     labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

 replicas: 3
```

How to see the replication controllers:
kubectl get replicationcontroller

If you check the pods you will see that we have 3 with starting with the name: myapp-rc-… indicating that they are all created by the replication controller

```
> kubectl get pods
NAME             READY     STATUS    RESTARTS   AGE
myapp-rc-4lvk9   1/1       Running   0          20s
myapp-rc-mc2mf   1/1       Running   0          20s
myapp-rc-px9pz   1/1       Running   0          20s
```

ReplicaSets: Are more or less like the replication controller.

The **apiVersion:** field must be **apps/v1** if you add v1 you will have an error like this, v1 has no support for replica set.

```
error: unable to recognize "replicaset-
definition.yml": no matches for /, Kind=ReplicaSet
```

There is a major difference between RC and RS, is the field **selector**

Selector section: Helps the RS to identify what pods fall under it. It is mandatory to include it.

Why to add the selector section: Because rs can also manage pods not created as part of the rs creation.

You can specify the selector in the replication controller but is not mandatory. In replica sets it is mandatory to include it.

Match Labels: Under selector we must specify the matchLabels: field
Matches the labels specified under it to the labels in the pod.
The replicaset selector also provides many other options for matching labels that were not available in the RC.

To create the replicaset:
kubectl create -f replicaset-definition.yaml

To see the replicasets:
kubectl get replicaset

If you check the pods you will see that the name starts with the name of the replicaset + the random number.

Delete RS: kubectl delete replicaset myapp-replicaset → also deletes all underlying Pods

Here a rs definition, notice the selector being a sibling of template and replicas.

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

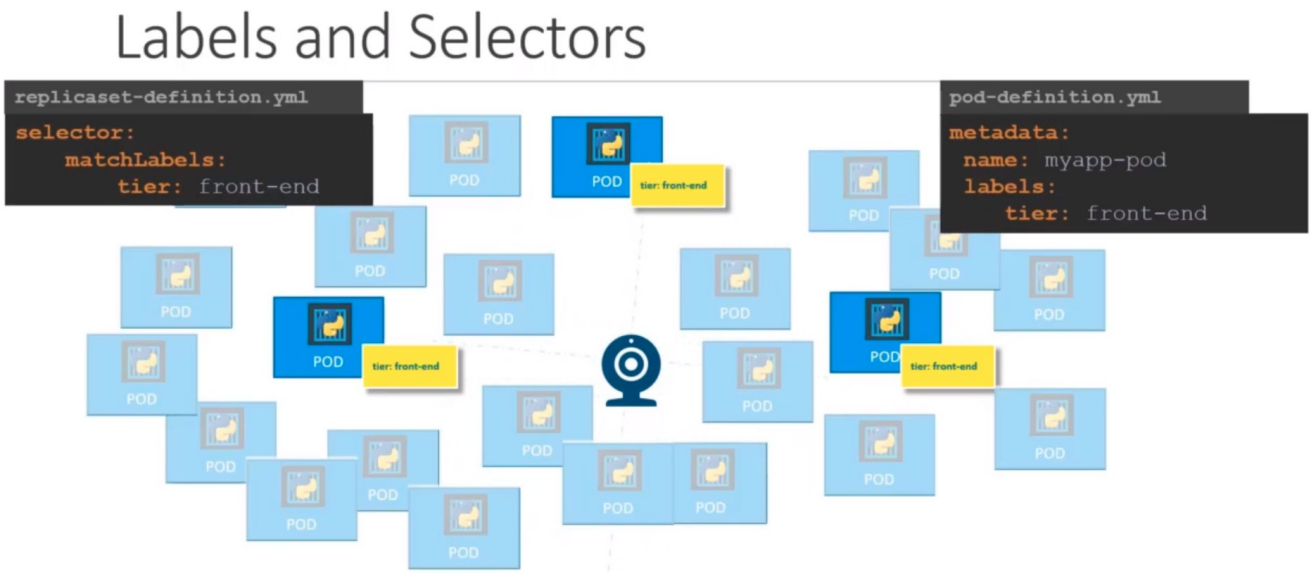
Labels and Selectors: You can use replica set to monitor already created pods, in case one fails the rs will replace it.

The role of the rs is to monitor the pods and if any of them fail deploy new ones. RS is in fact a process that monitors the pods.

How does the RS know what pods to monitor, there could be hundreds of pods in the cluster running different applications.

This is where labeling our pods during creation comes in handy. We can provide this labels as a filter for RS.

In the RS, under the selector section, we use the matchLabels filter and provide the same label that we used when creating the pod. This way the RS know which pods to monitor.



Scale Replica Set: There are different ways:
For instance we have 3 replicas and we want to scale to 6:

- **Updating the replicas in the definition file:** change in the yaml 3 for 6. Then run:
  kubectl replace -f replicaset-definition.yaml

- **Scale command over the .yaml:**
  kubectl scale --replicas=6 -f replicaset-definition.yaml

- **Scale command directly into the process:**
  kubectl scale --replicas=6 replicaset (this is the type) myapp-replicaset (this is the name)
  This option does not modify the .yaml, you will still have 3 replicas in the definition file.

- There are also options available to automatically scaling the RS based on load but this will be treated in the future.

Deployments:

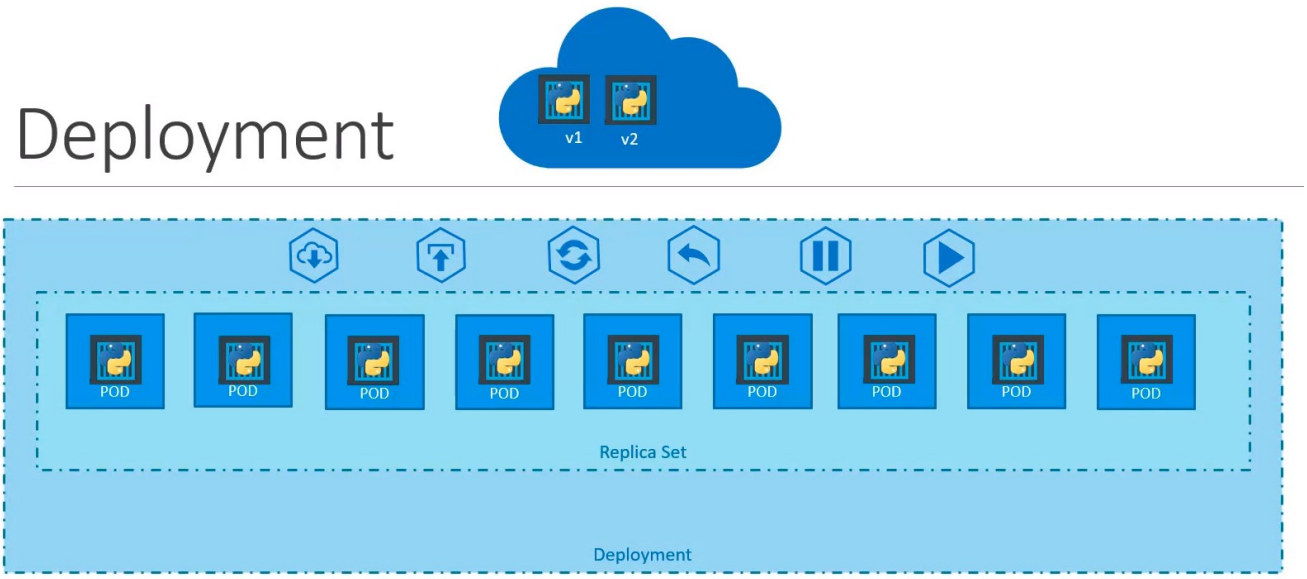
What is a deployment: It's a layer more of abstraction of a replica set. It's a k8s object that becomes higher in the hierarchy. First pod then replica set then deployment. A replicaset is created automatically when a deployment is created.

- **A deployment let's you to:**
  - Create several instances of your application. This is achieved using replicasets.

  - Rolling updates: Upgrade sequentially your app. When you update the version of the code or the version of your app, to avoid having downtime, you can upgrade it pod by pod instead of all at the same time.

- Roll Back: When the upgrade performed resulted in an unexpected error. You can roll it back.

- Pause / Resume: When you want to do multiple changes in the environment, like upgrading nginx, scaling new environment,modifying the resource allocation, etc, you can pause the environment, make the changes then resume it to roll out all the changes together.



How to create it / Definition: Is exactly similar to replica set.
Just change Kind for: Deployment and in metadata.name you can specify myapp-deployment or something similar. The rest is the same than a replicaset.

kubectl apply -f deployment-definition.yml

kubectl get deployments

kubectl get replicaset → because a deployment automatically creates a rs. Will have the deployment name specified in metadata.name of the deployment definition.

kubectl get pods → because the replicaset will create the pods requested in the definition. The pods will have the name specified for the deployment.

kubectl get all → will show you pods, rs and deployment at once.

Till now we can not see much difference between deployment and replicaset, we will see it in future lessons.

Edit deployment:
kubectl edit deployment my_deployment

Create deployment from cli:
kubectl create deployment httpd-frontend --image=httpd:2.4-alpine

Scale deployment:
kubectl scale deployment httpd-frontend --replicas=3

Tip for the exam:

You won't be able to copy and paste definitions to create yml files.

You must create objects using the kubectl run command, you have to get used to it:

Deprecated since v1.17 https://kubernetes.io/docs/reference/kubectl/conventions/
Create an NGINX Pod:
kubectl run --generator=run-pod/v1 nginx --image=nginx

Generate POD Manifest YAML file (-o yaml). **Don't create it**(--dry-run)
kubectl run --generator=run-pod/v1 nginx --image=nginx --dry-run -o yaml

Create a deployment
kubectl create deployment --image=nginx nginx

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)
kubectl create deployment --image=nginx nginx --dry-run -o yaml

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run) with 4 Replicas (--replicas=4)
kubectl create deployment --image=nginx nginx --dry-run -o yaml > nginx-deployment.yaml

Save it to a file, make necessary changes to the file (for example, adding more replicas) and then create the deployment.

Namespaces:

What is a namespace: Is used to identify objects (pods,deployments,rs,services, etc) by manually creating a group (namespace) that grups all the objects we want.

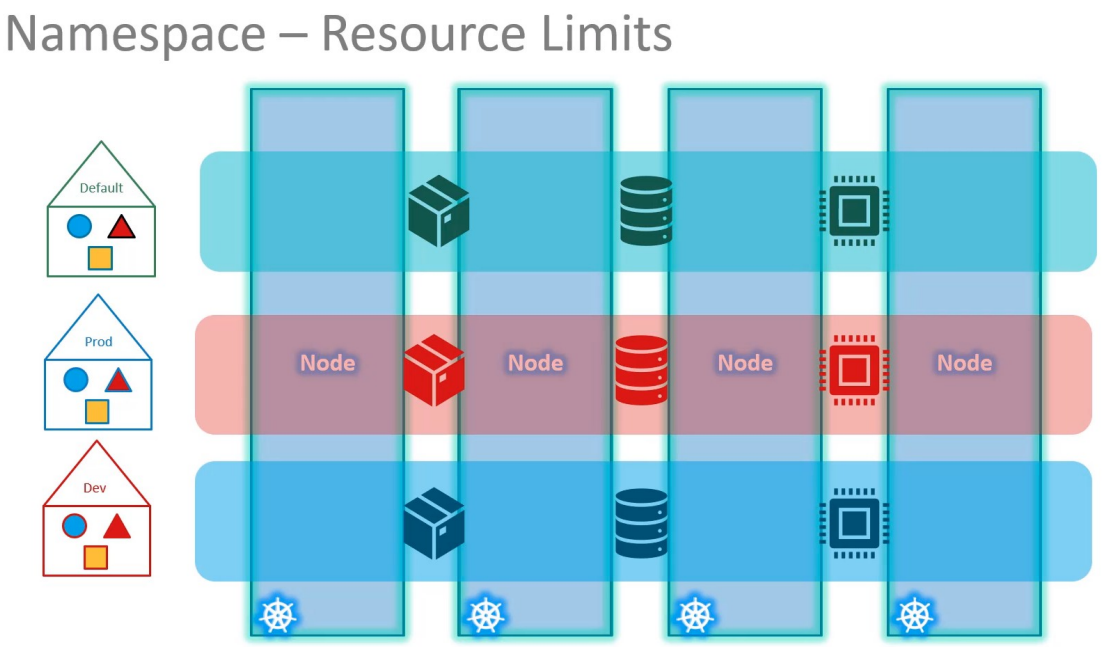
Provides some kind of isolation between different groups.
For instance, you can create a namespace for the dev environment, then any change, like resources, will apply only to dev instead of affecting a prod environment or so.

If we don't specify any group when creating an object, it will be assigned the *default* one.

Known namespaces: kube-system, default, kube-public.

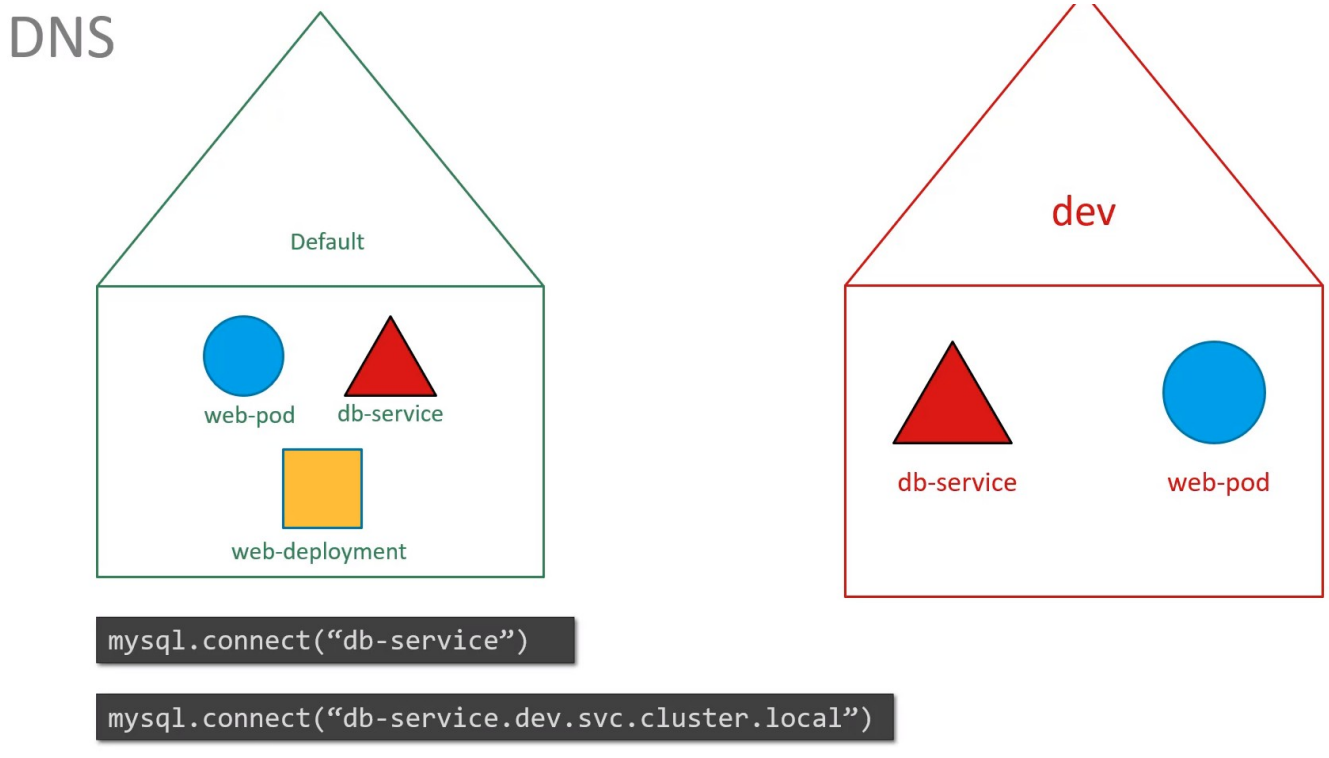
If you have a small environment or you are testing k8s is not necessary to create namespaces.
But if you have a production environment with hundreds or thousands of objects you may need to use them.

You can create namespaces and set up policies to say who can do what. You can assign specific resources for a particular namespace.



Communications: When in the same namespace the objects are reach the others using the name of the service, for instance the web app pod can reach the db pod just using the hostname db-service.

If required the web app pod can reach another service in another namespace as well.



You have to point for the service and the whole hostname of the other cluster, like in the above image, which we point to a service in the dev namespace.

Commands:

kubectl get pods → will show you only the pods in the default namespace.

kubectl get pods --namespace=kube-system → show the pods in the kube-system namespace
kubectl get pods -n kube-system → this does the same

Create a pod specifying the namespace: If you create a pod using the a file the pod is created in the same namespace.
kubectl create -f pod-definition.yaml --namespace=dev → to create it in the dev namespace
kubectl run redis --image=redis --namespace=finance
kubectl run redis --image=redis –dry-run=client -o yaml > pod.yaml

To create an object all the time in the specified namespace:
You have to add the label namespace in the yaml. Then you won't need to specify the namespace for that object.

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 namespace: dev
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

namespace: dev in this example

How to create a new namespace: You have to create a new object called Namespace.

namespace-dev.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
    name: dev
```

Then run: kubectl create -f namespace-dev.yaml

Or without yaml: kubectl create namespace dev

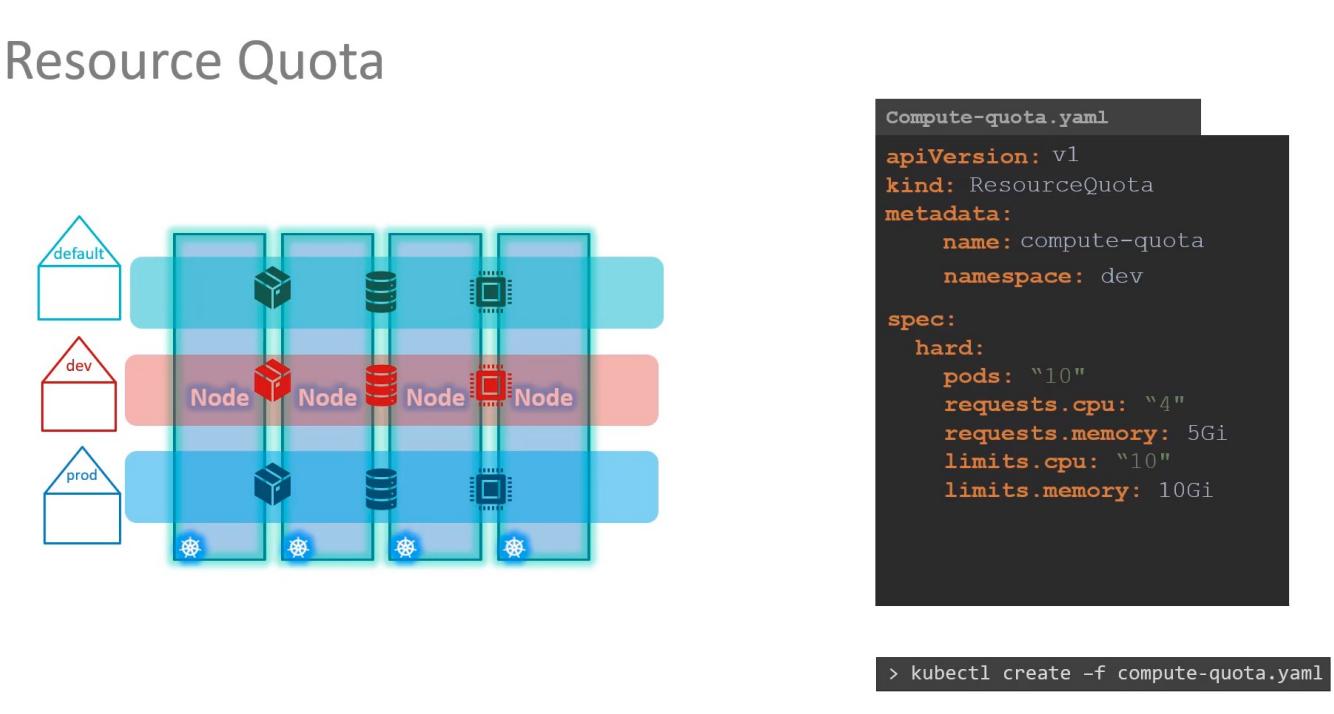
How to change the default namespace: To get that other namespace when typing kubectl get pods

kubectl config set-context $(kubectl config current-context) --namespace=dev

Then you will get the dev namespace by default.
To see the other namespaces you have to specify it as usual:
kubectl get pods --namespaces=default

See all namespaces:
kubectl get pods --all-namespaces
kubectl get pods -A

Limit resources in a namespace: ResourceQuota

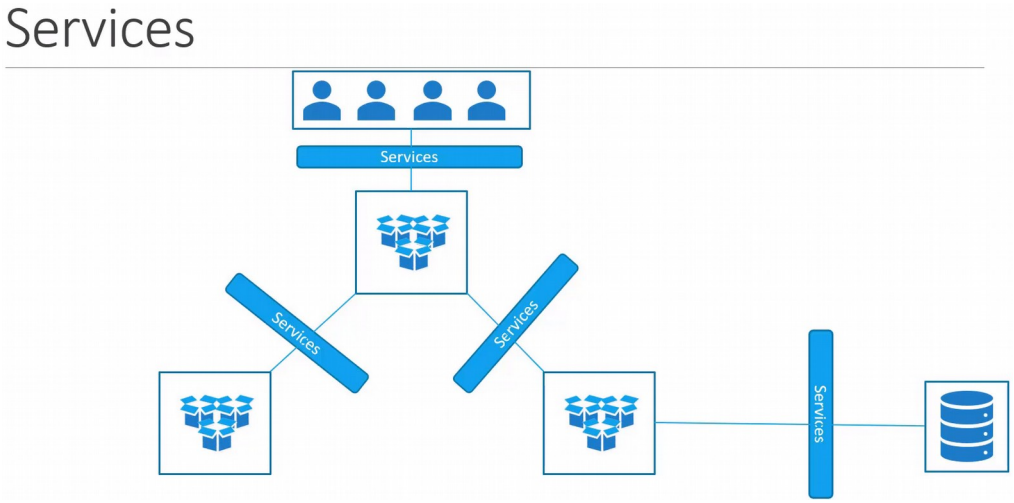


Commands to try: to understand
kubectl get ns
kubectl get ns --no-headers
kubectl -n research get pods --no-headers → research is a namespace from the exercise 35

Services:

A service is an object like deployments, replicaset,pods, etc. Is like a virtual server inside the node.

Enables communications between various components within and outside of the application. Helps us connect applications with other applications or users.



Services Types:

- **ClusterIP:** Is the default one. The service creates a virtual IP inside the cluster to enable communications between different services such as a set of front end servers and a set of backend servers.

- **NodePort:** Is listening to a port on the Node and forwards requests on that node to a port on the POD running the application. The service makes an internal POD accessible on a port on the Node.

- **LoadBalancer:** Provides a load balancer for our application in supported cloud providers, like AWS, GCP,

- **ExternalName:** Newer service. It has no selectors, nor does it define ports or endpoints. It allows the return of an alias to an external service. The redirection happens at the DNS level, not via a proxy or forward. This object can be useful for services not yet brought into the Kubernetes cluster. A simple change of the type in the future would redirect traffic to the internal objects.

NodePort: The service listen to a port on the Node and forwards the requests to the POD. So from outside I have to point to the node IP and to the port on the node that the service have created automatically or us manually, the port have to be between 30000 and 32767.

The service will know which PODS to include by using the selector. Which in the pod is the label, for instance: POD labels.app: myapp Service selector.app: myapp. A Random algorithm selects the PODS. The Service works like a built in load balancer to distribute load across different PODS.

If we have PODS from the same service are in different nodes,k8s will expand the Service across all the nodes and will use the same nodePort in each node. So to access the app from outside you will need to point to each node IP and the same nodePort.

service-definition.yml

```
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
       app: myapp
       type: front-end
```

spec.type: If you don't specify it by default will assign ClusterIp

spec.port.targetPort: Is the POD port. If you don't specify it will assign the port set on spec.port.

spec.port.port: Is the port of the Service and it's mandatory.

spec.port.nodePort: If you don't specify it will assign a random port between 30000 and 32767

selector: Are the labels of the pod. Is how we link the service to the pods.

Create the service: kubectl create -f service-definition.yaml

Check services: kubectl get services
kubectl get svc

Check if works: curl http://IP_of_THE_node:nodePort
If all is fine you have to see "Welcome to nginx" if the app is nginx.

ClusterIP:

Creates automatically a virtual IP inside the cluster to enable communication between different services. You only have to point to an IP to reach the service, even if it is in several nodes.

service-definition.yml

```
apiVersion: v1
kind: Service
metadata:
    name: back-end

spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
       app: myapp
       type: back-end
```

spec.type: If you don't specify it by default will assign ClusterIp

spec.targetPort: If you don't specify it will assign port 80

spec.selector: Uses the selector to link the service to a set of pods. We have to pick the content of selector from the pod definition file, from metadata.labels

Create the service:
kubectl create -f service-definition.yaml

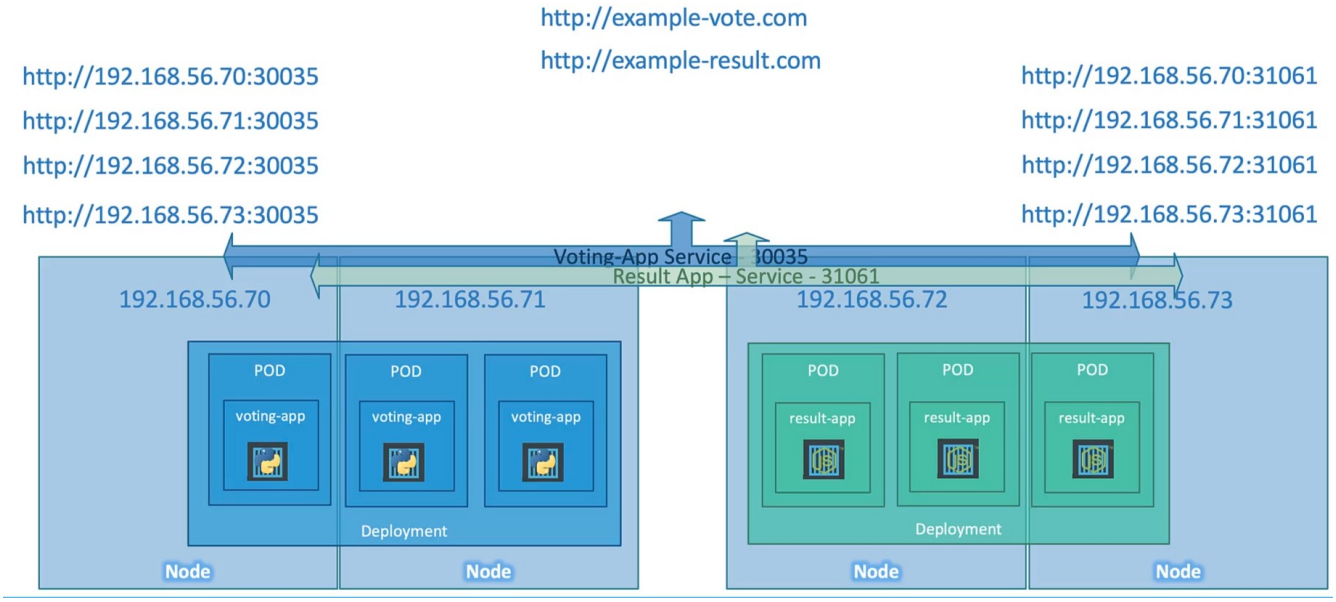
Get services:
kubectl get svc

Load Balancer:

It uses the same configuration than nodePort, only changing the type for LoadBalancer.

Using the nodeport type, if you have different services in different nodes you will need to provide the end user with all the endpoints we have below.

Notice that a service can point to a node in which you don't have pods for that service. For instance, the nodes .72 and .73 don't have the voting-app but if you point to http://192.168.56.73:30035 you will get the voting-app anyway.



To access the voting-app the end user have to access to 4 different end points.
The same to access the result-app.

But the end user wants a single access point. He want to access to a simple url, like example-vote.com or example-result.com This is when the LoadBalancer type comes in.

You have to set up a Load balancer in a virtual machine, like haproxy or nginx and configure it to redirect all the traffic coming from example-vote.com or example-result.com to the proper nodes with the proper port, but you have to install, configure and maintain the load balancer.

You can use the load balancer from the main cloud providers, AWS, GCP, Azure. They manage the load balancer, you only have to create the rules and that's it. This only works with supported cloud platforms. The other will give you problems.

The definition file for the load balancer in k8s is similar to the nodeport definition.

Just change the type for LoadBalancer.

ExternalName:

Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record

with its value. No proxying of any kind is set up.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

Imperative vs Declarative:

Documentation

Imperative, general description: Is to run all the commands one by one, will take longer.

Let's say that you want to install a nginx into a webserver.

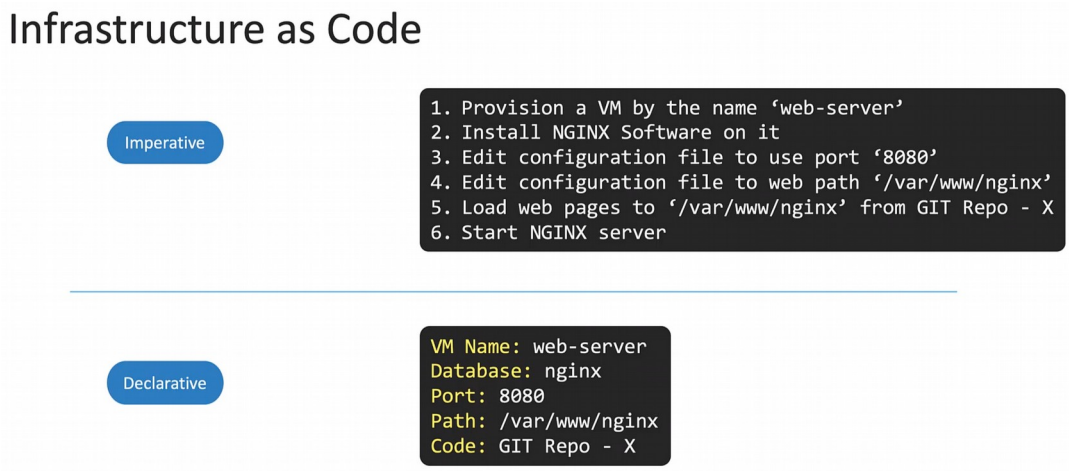
- You have to install the vm called web-server
- You have to install the nginx
- Edit the conf file to use the port 8080. Edit the conf file to point /var/www/nginx
- Load the web pages to the path from the git repository
- Start nginx, etc, etc.

You may find more problems, like there is already a vm called webserver, the nginx version is outdated, etc. This will add more steps to the process.

Declarative, general description: Ansible, puppet,chef, terraform, etc are using the declarative way.

We Declare our requirements. All we say is that we need a vm with nginx, to the port 8080, to the specific folder, using the code in this repo.

And everything that's needed to be done to get this infrastructure in place is done by the system or the software.
You don't have to provide step by step instructions. Orchestration tools are into this category.



Imperative in K8s: We are saying exactly how to bring the infrastructure to our need by creating, updating or deleting objects command by command in the kubectl.

Exam tip: In the exam is better to use the imperative approach because we are creating simple objects and will be faster to create them by clicking the commands one by one instead of creating complex files.

Within imperative approach there are two ways: Docs

- **Imperative commands:** Easiest and fastest. You operate directly to live objects by clicking commands. In the other hand you don’t provide a history record except what is live.
  Example: kubectl create deployment nginx --image nginx

- **Imperative objects:** In imperative object configuration, the kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name. The file specified must contain a full definition of the object in YAML or JSON format.

  Object configuration can be stored on git, reviewed changes before push.
  Requires basic understanding of object schema and require to write a yaml file.

  Examples: kubectl create -f nginx.yaml
  kubectl delete -f nginx.yaml -f redis.yaml
  kubectl replace -f nginx.yaml

Declarative in K8s: kubectl apply. We create a set of files that defines the expected state of the applications and services on a k8s cluster and with a simple kubectl apply command k8s should be able to read the conf file and decide by it self what needs to be done to bring the ifr to the expected state.

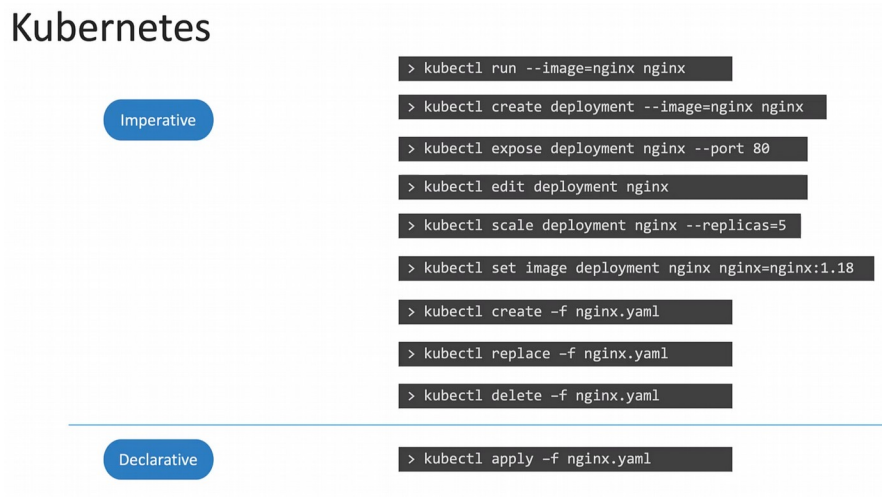
The **apply** command will look at the existing configuration and figure out what changes need to be made to the system.

You can specify the name of the file or if there are several of them you can specify the path to the folder that contains them:

kubectl apply -f nginx.yaml
kubectl apply /path/to/config-files
kubectl diff -f configs/
kubectl apply -f configs/

Another definition from k8s.io:
When using declarative object configuration, a user operates on object configuration files stored locally, however the user does not define the operations to be taken on the files.

Create, update, and delete operations are automatically detected per-object by kubectl. This enables working on directories, where different operations might be needed for different objects.



Exam tip:

As said, in the exam we have to use the imperative commands because is faster but in a company is better to use the declarative one.

For the imperative commands we can use some tricks to confirm that a command will work before running it.

--dry-run: By default as soon as the command is run, the resource will be created.
If you simply want to test your command, use the ***--dry-run=client*** option.
This will not create the resource, instead, tell you whether the resource can be created and if your command is right.

-o yaml: This will output the resource definition in YAML format on the screen.

Examples:

Create an NGINX Pod
kubectl run nginx --image=nginx

Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)
kubectl run nginx --image=nginx --dry-run=client -o yaml

Create a deployment
kubectl create deployment --image=nginx nginx

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml

Save it to a file - (If you need to modify or add some other details)
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml > nginx-deployment.yaml

Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379
kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors)

Or
kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml
(This will not use the pods labels as selectors, instead it will assume selectors as app=redis. You cannot pass in selectors as an option. So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)

Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:

kubectl expose pod nginx --port=80 --name nginx-service --type=NodePort --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors, but you cannot specify the node port. You have to generate a definition file and then add the node port in manually before creating the service with the pod.)

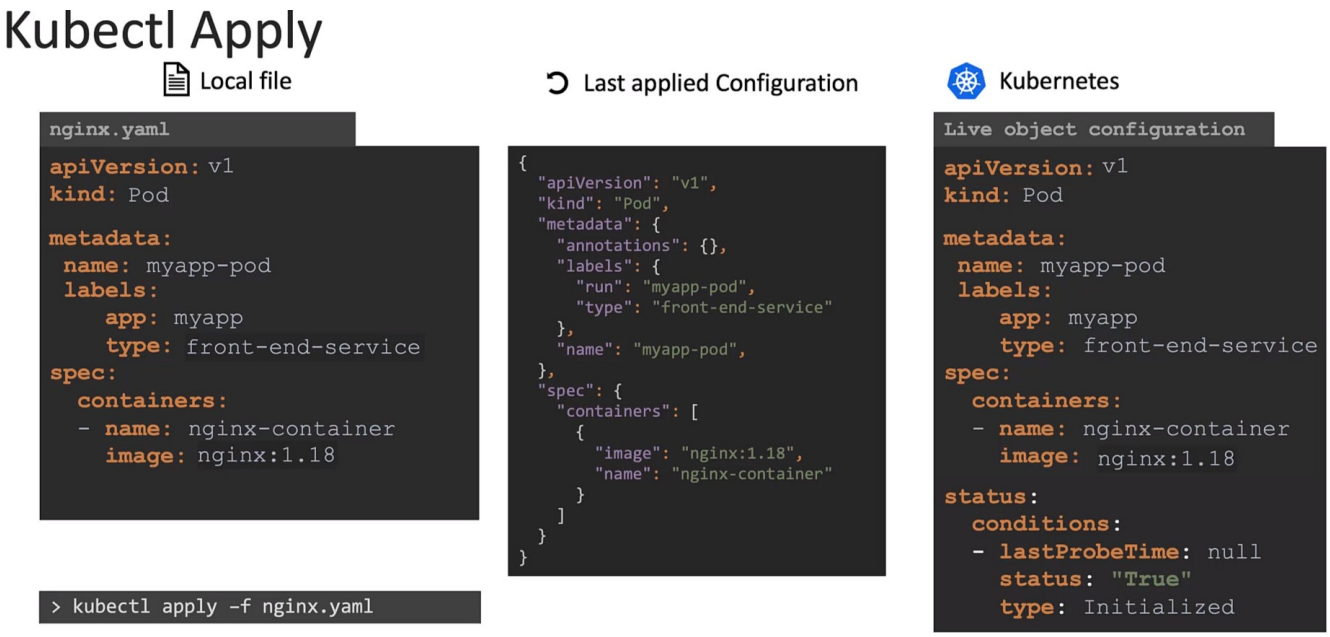
Or

kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml

(This will not use the pods labels as selectors)

Both the above commands have their own challenges. While one of it cannot accept a selector the other cannot accept a node port. I would recommend going with the `kubectl expose` command. If you need to specify a node port, generate a definition file using the same command and manually input the nodeport before creating the service.

Apply Command: How it works internally



kubectl apply can be used to manage objects in the declarative way.

The apply command takes into consideration
The local Configuration file
The live object definition on K8s and
The last applied Configuration before making a decision on what changes to be made.

When you run the apply command, if the object does not already exists, the object is created.

When the object is created a live object configuration similar to the one created locally is created within K8s but with additional fields to store status on the object.

This is the live configuration of the object on the K8s cluster and is how K8s internally stores information about an object.

No matter what approach you use to create the object (imperative/declarative).

But when you use the kubectl apply command to create an object it does something more, the yaml file of the local conf we wrote is converted to a JSON format and it is then stored as the Last Applied Configuration.

For any updates to the object the 3 are compared to identify what changes are to be made on the live object.

For instance, when the nginx image is updated from 1.18 to 1.19 in the local file and then kubectl apply -f nginx.yaml this value is compared with the value in the live configuration.

And if there is a difference the live conf is updated with the new value.

After any change the last applied JSON format is updated to the last version.

Why do we need the last applied configuration?
If we remove a field in the local file, like the type label, and when we run the kubectl apply command it will check if the field is in the Last applied conf. It the file is there the system will remove the field from the Live Object Configuration.

This is how it works

Where stored the Last Applied Configuration:
Is stored in JSON format on the Live Object Configuration on the K8s cluster itself as a field in metadata called annotations labeled as last-applied-configuration

This is only done when you use the kubectl apply command.
kubectl create or kubectl replace commands do NOT store the last applied configuration.

Bear in mind not to mix imperative and declarative approaches.

Scheduling:

Manual Scheduling: Every pod have a field called nodeName which is added automatically by K8s. By default is not set.

You can add into the pod definition yaml the name of the node in which you want to schedule the pod manually, under the spec section:

```
spec:
  nodeName:  worker-node-02
```

Then when you deploy the pod it will be manually scheduled to worker-node-02

IMPORTANT: This only works when you create the pod. Once the POD is created k8s won't let you to schedule it to any node.

How to manually schedule an already created POD to a node: Creating a Binding object.
This works with API calls, via POST request, like the real scheduler actually works, which is with the json format of the POD data.

You can create a binding like the one in the image and convert it to JSON.

Then you can build the API call POST request including that json data and make the call to the node you want to schedule the POD

```
curl --header "Content-Type:application/json" --request POST --data '{"apiVersion":"v1", "kind": "Binding“ …. }
http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/
```

Pod-bind-definition.yaml

```yaml
apiVersion: v1
kind: Binding
metadata:
  name: nginx
target:
  apiVersion: v1
  kind: Node
  name: node02
```

Labels Selectors and Annotations:

Documentation

Label Selector: Via label selector the user can identify a set of objects. Label selector is the core grouping primitives in K8s.

Equality-based requirement:

- = , == → are the same, key equal to value. environment = production, env == prod
- != → inequality, tier != frontend , select all objects with tier as key excluding frontend.

Set-based requirement:

- in: env in (prod,qa). Selects all resources with key env and value equal to prod and qa
- notin: tier notin (fe, be). Select all resources with key equal to tier and value not equal to fe, be.
- exists: partition → selects objects with the key partition, no value is checked. The opposite for !partition

Annotations: Used to record other details for information purposes. Like contact details, like emails, phone numbers etc. Are commentaries like in a script when you use the # Commentary here

See all labels:
kubectl get pods --show-labels

See specific label:
kubectl get pods **--selector** app=App1
kubectl get pods **-l** app=App1

1-A: Show all pods with label ENV (environment) and with the key DEV:
master $ **kubectl get pods --selector=env=dev**

```
NAME        READY   STATUS    RESTARTS   AGE
app-1-6zs75   1/1     Running   0          5m26s
app-1-fdhcj   1/1     Running   0          5m26s
app-1-mc2kg   1/1     Running   0          5m26s
db-1-c4w5x    1/1     Running   0          5m26s
db-1-k24vp    1/1     Running   0          5m26s
db-1-qshsz    1/1     Running   0          5m26s
db-1-zwwdz    1/1     Running   0          5m26s
```

Show me the same than 1-A but without headers:

master $ kubectl get pods --selector=env=dev --no-headers

```
app-1-6zs75   1/1   Running   0     5m33s
app-1-fdhcj   1/1   Running   0     5m33s
app-1-mc2kg   1/1   Running   0     5m33s
db-1-c4w5x    1/1   Running   0     5m33s
db-1-k24vp    1/1   Running   0     5m33s
db-1-qshsz    1/1   Running   0     5m33s
db-1-zwwdz    1/1   Running   0     5m33s
```

Without HEADERS we can count the number of pods like this:

kubectl get pods --selector=env=dev --no-headers |wc -l

See ALL OBJECTS with the same label: (env = prod)

kubectl get **all** --selector=env=prod

master $ **kubectl get all --selector=env**

```
NAME           READY   STATUS    RESTARTS   AGE
pod/app-1-6zs75   1/1     Running   0          16m
pod/app-1-fdhcj   1/1     Running   0          16m
pod/app-1-mc2kg   1/1     Running   0          16m
pod/app-1-zzxdf   1/1     Running   0          16m
pod/app-2-jmpkx   1/1     Running   0          16m
pod/auth          1/1     Running   0          16m
pod/db-1-c4w5x    1/1     Running   0          16m
pod/db-1-k24vp    1/1     Running   0          16m
pod/db-1-qshsz    1/1     Running   0          16m
pod/db-1-zwwdz    1/1     Running   0          16m
pod/db-2-7h5sv    1/1     Running   0          16m

NAME          TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/app-1   ClusterIP   10.100.65.197   <none>        3306/TCP   16m

NAME                 DESIRED   CURRENT   READY   AGE
replicaset.apps/app-1   3         3         3       16m
replicaset.apps/app-2   1         1         1       16m
replicaset.apps/db-1    4         4         4       16m
replicaset.apps/db-2    1         1         1       16m
```

Get the same but without headers:

master $ **kubectl get all --selector=env --no-headers**

```
pod/app-1-6zs75   1/1   Running   0     18m
pod/app-1-fdhcj   1/1   Running   0     18m
pod/app-1-mc2kg   1/1   Running   0     18m
pod/app-1-zzxdf   1/1   Running   0     18m
pod/app-2-jmpkx   1/1   Running   0     18m
pod/auth          1/1   Running   0     18m
pod/db-1-c4w5x    1/1   Running   0     18m
pod/db-1-k24vp    1/1   Running   0     18m
pod/db-1-qshsz    1/1   Running   0     18m
pod/db-1-zwwdz    1/1   Running   0     18m
pod/db-2-7h5sv    1/1   Running   0     18m
service/app-1   ClusterIP   10.100.65.197   <none>   3306/TCP   18m
replicaset.apps/app-1   3   3   3   18m
replicaset.apps/app-2   1   1   1   18m
replicaset.apps/db-1    4   4   4   18m
replicaset.apps/db-2    1   1   1   18m
```

Filter with different labels:

kubectl get pods --selector=env=prod,bu=finance,tier=frontend

Taints and Tolerations:

Taint: Are set on Nodes. Block pods to be scheduled into the Node that have a taint.

Tolerations: Are set on Pods. Pods that have the toleration will be allowed to be scheduled into a Node with a Taint.

How to Taint a NODE:
kubectl taint nodes node-name key=value:taint-effect

Taint Effect: Defines what would happen to pods if they do not tolerate the taint.

- **NoSchedule**: The pods will not be scheduled on the Node.
- **PreferNoSchedule:** The system will try to avoid to schedule the Pod on the Node but that's not guarantied.
- **NoExecute**: New Pods will not be scheduled on the Node and existing Pods in the node will be evicted if they do not tolerate the taint. These last pods may be scheduled to the Node before the taint was applied on the Node, if this is the case this pod will be killed.

Taint Example command:
kubectl taint nodes node1 app=blue:NoSchedule

Set up Toleration: Have to be done in the pod definition file

pod-definition.yml

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

  tolerations:
  - key:"app"
    operator:"Equal"
    value:"blue"
    effect:"NoSchedule"
```

Son of Spec create the tolerations section. All values must be in double quotes.

- key:"app"
operator:"Equal"
value:"blue"
effect:"NoSchedule"

Taint only says to not to schedule Pods in the particular Node, except the Pod have a toleration.
Taint don't guarantee that the Pod with the toleration will only be scheduled to the Node with the taint. The pod can go to any worker node.

Taints and toleration does not tell the pod to go to a particular node, instead it tells the node to only accept nodes with certain tolerations.

Node Affinity: We use it to guarantee that one Pod with a toleration will go to the Node that we want.

Master node: Is technically another node that has all the capabilities to host a Pod plus it runs all the management software.

The master node have a taint to avoid pods to be scheduled there, you can see it with this command:
kubectl describe node kubemaster | grep Taint
Taints: node-role.kubernetes.io/master:NoSchedule

How to add a taint to the master node: (this one already comes by default)
master $ kubectl taint nodes master node-role.kubernetes.io/master:NoSchedule

node/master tainted

How to remove a taint in the master node:
master $ kubectl taint nodes master node-role.kubernetes.io/master:NoSchedule-

node/master untainted

How to see all the options for a object, POD in this example:
kubectl explain pod --recursive |grep -A5 tolerations

I'm searching to the tolerations sections and I will see all the options I can use.
I can copy the hole section and paste it in a pod if I need.

Node Selectors:

Example: You have 3 worker nodes. One of them have bigger space and cpu resources.
You have an app that requires bigger resources.

In the server:
To schedule that app to that particular node we use **nodeSelector**. Is just a key value pair that you set on the node using labels:

kubectl label nodes <node_name> <label_key>=<label_value>
kubectl label nodes node01 size=Large
kubectl label nodes node01 cpu=high

In the pod you have to specify it in the definition as child of spec.

pod-definition.yml

```
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```

```
spec:
    nodeSelector:
      size: Large
```

Limitations: nodeSelector have limitations: Uses a single label as selector.
You can not say, for example, place a pod into a node that have large OR medium space.
Or place the pod into a node that is NOT small in size.
For this we have node Affinity and antiaffinity.

Node Affinity:

Definition: The primary purpose on Node Affinity is to ensure PODS are hosted on particular nodes.

You can do it with nodeSelectors but with these you can not provide advance expressions like OR or NOT.

Node Affinity provides us with advanced capabilities to limit POD placements on specific nodes.
This is also more complex than nodeSelector.

Under PodSpec:

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: disktype
            operator: In
            values:
            - ssd
```

Operators:

- **In** → Ensures that a POD will be placed on a node whose label *disktype* has any value in the list of values specified, *ssd* in this example. You can add more values if needed. In this case would check: have the node this value OR this other value?

- **NotIn** → Node affinity will match the nodes with the label *disktype* that don't have the value *ssd*.

- **Exists** → Just checks the key (label). If there is a key the POD will be placed in that node. Exists doesn't require the value.

- There are more operators: DoesNotExist, Gt, Lt

Type of Node Affinity:
Defines the behavior of the scheduler with respect to nodeAffinity and the stages of the node cycle of the POD.

- **required**DuringScheduling**Ignored**DuringExecution → Will place the POD into the node and if the node is busy and there is no more space or resources the POD won't be placed in any other node and will be lost. This type is used when the placement of the POD is crucial.

- **preferred**DuringScheduling**Ignored**DuringExecution → If the POD placement is less important than running the workload itself, then you can use the preferred type. This is saying to the scheduler try your best to place the POD into the matching node but if you can not find one just place it anywhere.

- Planned: **required**DuringScheduling**Required**DuringExecution → This type is expected in the future. This will evict PODs already working in the node if a label that is not matching is set or removed.

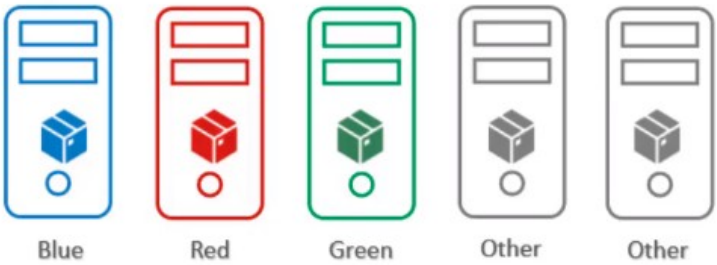
Redirect a live object to a .yaml file:
kubectl get deployments blue -o yaml > blue.yaml

Node Affinity and Taint and Toleration

With taints and tolerations we can inform the scheduler that this node have a taint and this POD have a toleration that allow the scheduler to place it on the node. But we can not guarantee that the POD is gonna be placed in another node.

With Node Affinity we can guarantee that a POD will be placed in the node we want. But we can not avoid other PODs WITHOUT node affinity set on their POD definition to be scheduled there.

So we can use both to guarantee that all the pods will be places in the nodes that we want.



Resource Requirements and Limits:

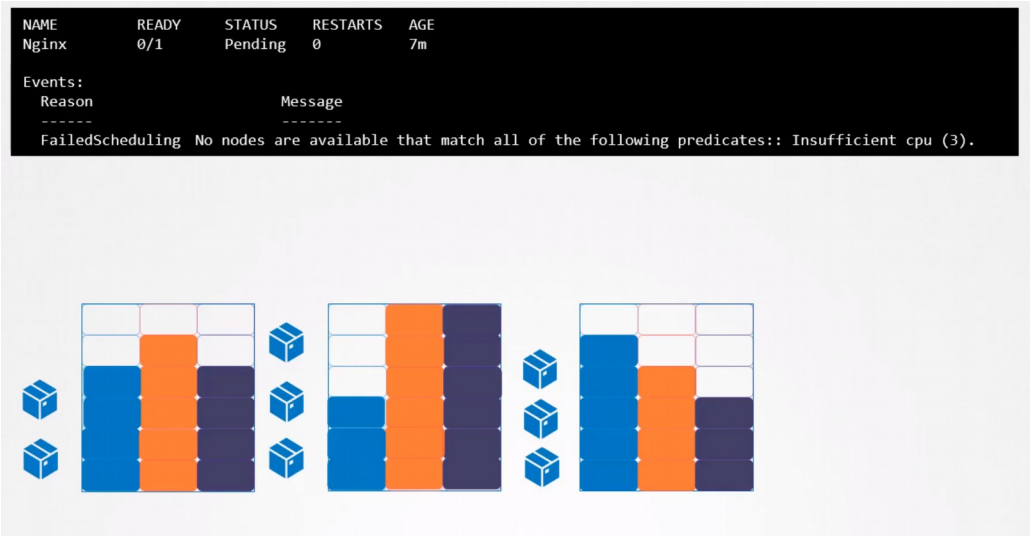
[Documentation](#) [Configure default memory requests](#) [Configure default CPU requests](#)

As we know the scheduler decides which Node a POD goes to.

The scheduler takes into consideration the amount of resources required by a POD and those available on the Nodes.

If there is no sufficient resources in any Node the scheduler holds back scheduling the POD and you will see it in PENDING state.

In the pod description, in the event section, you will see the reason, insufficient cpu or disk or whatever.



Resource request for a container: Is the minimum amount or resources that a container needs to work. We are talking about CPU, Memory and Disk.

When the scheduler tries to place a POD it uses these numbers to identify a node with a sufficient amount of resources available.

How to specify more resources for a POD: If your app needs more than the default values you can specify it in the pod definition yaml. Go to PodSpecContainers and create the child *resources*

```
resources:
  requests:
    memory: "64Mi"   → minimum amount needed
    cpu: "250m"      → minimum amount needed
  limits:
    memory: "128Mi"  → maximum amount allowed
    cpu: "500m"      → maximum amount allowed
```

Create LimitRange for a NameSpace:
You can specify a CPU or memory limit for a namespace creating a LimitRange object.

Then all the objects created in that namespace will apply those limits. You can specify only the requested amount (the minimum the object will get) and the limit (the maximum the object will obtain)

CPU and memory meaning:

CPU: The minimum your can request is 0.1, which is 100m (one hundred millicpu)
Memory: Is measured in bytes. You can use Mi

DaemonSets:

DaemonSets makes sure that you run one copy of the pod you want in each node of the cluster.

Like replicasets takes care of the status of the pods and if one is terminated another is deployed instantly to replace it.

If a new node is added to the cluster, a replica of the pod will we added there too.

This is used for example to deploy pods that control the status (health monitoring) of the nodes or for login purposes, etc.

We are already using daemonsets to deploy kube-system pods, like kube-proxy or for the networking like flannel or wave-net. All of these pods must be in all the nodes, daemonset takes care of this automatically.

Create a DaemonSet:
Is similar to creating a replicaset, except for the **kind: DaemonSet**

daemon-set-definition.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
      - name: monitoring-agent
        image: monitoring-agent
```

replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
      - name: monitoring-agent
        image: monitoring-agent
```

Commands:

Create it: kubectl create -f daemon-set-definition.yaml

View ds: kubectl get daemonsets
kubectl describe daemonsets <name_daemonset>

How many nodes are running kube-proxy pods:
kubectl get pods -A -owide |grep kube-proxy

See the image of a daemonset:
kubectl describe daemonsets.apps -n kube-system kube-flannel-ds-amd64

TO CREATE A DAEMONSET in imperative mode:
kubectl create deployment elasticsearch --image=k8s.gcr.io/fluentd-elasticsearch:1.20 -n kube-system --dry-run=client -oyaml > daemon.yaml → then change the kind from deployment to daemonset.

How does it works:
Before K8s v1.12 was using nodeNames to go to each node-hello

Now is using node affinity rules and the default scheduler.

Static Pods:

Documentation

If you only have a worker node, this means no master or other workers, you can create PODs on it. Only POD, no deployments, rs, etc.

The node have to have installed kubelet. In the kubelet configuration there is a line that points to a folder. You can leave your pod.yaml declaration there. Kubelet is taking a look periodically into this folder and creating or maintaining the pods that if finding there.

Kubelet only works as a POD level. That's why it is only able to create PODs.

Folder:
/etc/kubernetes/manifests → this is designated to store information about pods but you can add there .yaml files with pod definitions and kubelet will pick it up and will maintain it.
Kubelet periodically checks the content of this folder. If you modify the definition, kubelet will detect it and will apply the changes automatically.

Modify the manifest folder:
You can specify which directory will contain the manifests. To do so you have to specify it when you run the kubelet service passing it as an option.

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --pod-manifest-path=/etc/Kubernetes/manifests \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --register-node=true \\
  --v=2
```

Another way to configure it:
Into the kubelet.service skip the –pod-manifest line and add:
--config=kubeconfig.yaml
And inside kubeconfig.yaml add: staticPodPath: /etc/kubernetes/manifest

How to see the pods: once already created
docker ps

How does it work when the node is part of a cluster: K8s can take requests from different places to create PODs.

- The first is through the pod definition file from the static pods folder.
- The second is through an HTTP API endpoint (this is how kube-api server providers input to kubelet)

Kubelet can create both kind of pods, static and the api server ones at the same time.

API server is aware when a static pod is created, if you run kubectl get pods you will see the static pod there labeled as:
static-something-<name-of-the-node>

If there is a k8s cluster, kubelet creates a mirror object on the kube-api server.
What you see from the kube-api server is a read only POD. You can read details of the static pod but you can not edit or delete it. You can only delete it by modifying the file from the node’s manifest folder.

Commands:

How to see the static pods: In a k8s cluster:

– Look for the name of the master node, master in this example, then grep

```
master $ kubectl get pods -A |grep master
kube-system   etcd-master                   1/1    Running   0        6m15s
kube-system   kube-apiserver-master         1/1    Running   0        6m15s
kube-system   kube-controller-manager-master   1/1    Running   0        6m15s
kube-system   kube-scheduler-master         1/1    Running   0        6m15s
```

– You can go to /etc/kuberbetes/manifests and you will see the yamls there, do it in all the nodes.

Find where is the staticPdPath:
ps -aux | grep kubelet → search for --config=/var/lib/kubelet/config.yaml

vim /var/lib/kubelet/config.yaml and search for Path:
staticPodPath: /etc/kubernetes/manifests → default place

staticPodPath: /etc/just-to-mess-with-you → but could be another one, like this one.

Create a static pod:
kubectl run --restart=Never --image=busybox static-busybox --dry-run=client -o yaml --command -- sleep 1000 > /etc/kubernetes/manifests/static-busybox.yaml

---

Multiple Schedulers:

The default scheduler has an algorithm that distributes pods across nodes evenly as well it takes into consideration several conditions specified as taints and tolerations, node affinity etc.

But if you have an app that requires to be placed in some nodes after performing additional checks, you can create your own scheduler to use your algorithm to run your own custom conditions and checks in it.

You can write your own k8s scheduler program, package it and deploy it as the default scheduler or as an additional scheduler in the k8s cluster.

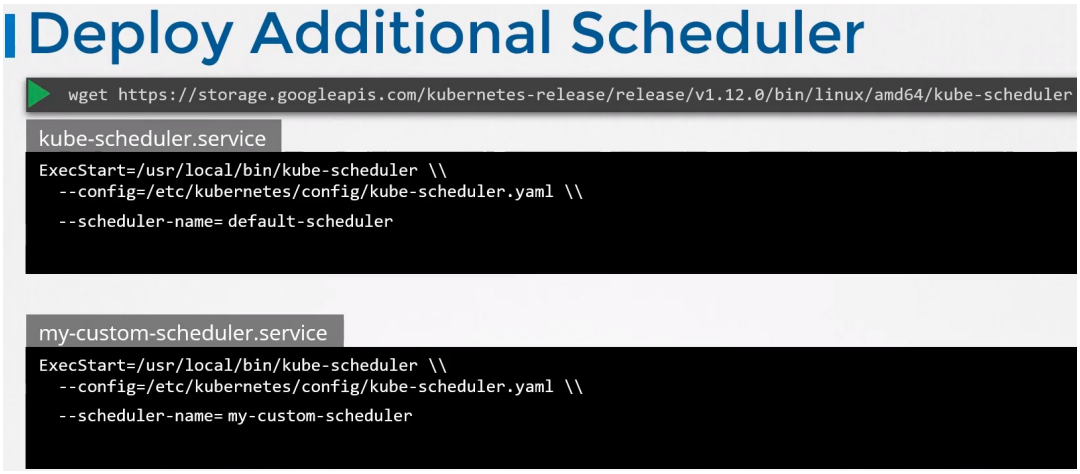
K8s cluster can run multiple schedulers at the same time.

Kubeadm tool deploy the scheduler as a static POD in /etc/kubernetes/manifests/kube-scheduler.yaml
In spec.containers.command you has the command and the associated options to start the scheduler

Create your scheduler: By copying the static scheduler yaml

- cp /etc/kubernetes/manifests/kube-scheduler.yaml ~/my-scheduler.yaml
- Go to kubernetes.io and search for multiple scheduler
- In the ServiceAccount template go to kind=Deployment and in the pod part (spec.containers) copy: - --leader-elect=false and - --scheduler-name=my-scheduler
- Paste it in spec.containers.- command.
- Change leader elect to false: - --leader-elect=false
- Modify the name: kube-scheduler to my-scheduler
- Modify the port if necessary.
- kubectl create -f my-scheduler.yaml
- Confirm that is in running state

You can create a scheduler using kubeadm :



Leader-elect option: when you have multiple copies of the scheduler running in the master node in a HA setup where you have multiple master nodes with kube-scheduler running on it.

If multiple copies of the same scheduler are running on different nodes only one can be active at a time. That's where the leader-elect option comes in.

If you only have a master node you have to set to false the line: --leader-elect=false

If you have HA with several masters you have to set it to true.
You can add the - --lock-object-name=my-custom-schedule → this is to differentiate the new custom scheduler from the default during the leader election process.

/etc/kubernetes/manifests/kube-scheduler.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    image: k8s.gcr.io/kube-scheduler-amd64:v1.11.3
    name: kube-scheduler
```

my-custom-scheduler.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-custom-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    - --scheduler-name=my-custom-scheduler
    - --lock-object-name=my-custom-scheduler


    image: k8s.gcr.io/kube-scheduler-amd64:v1.11.3
    name: kube-scheduler
```

Once done create the pod with **kubectl create** command and look for the new custom scheduler in the kubectl get pods -n=kube-system

Configure a POD or deployment to use the new scheduler:
In spec. create a new field called schedulerName and specify the name of the new scheduler.

spec:
  schedulerName: my-custom-scheduler

And create the pod with kubectl create. This way when the pod is created the right scheduler picks it up to schedule.

If all is fine the pod will appear as running, if something goes wrong will be in pending state or other like crash something.

```
kubectl get pods --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-78fcdf6894-bk4ml	1/1	Running	0	1h
coredns-78fcdf6894-ppr6m	1/1	Running	0	1h
etcd-master	1/1	Running	0	1h
kube-apiserver-master	1/1	Running	0	1h
kube-controller-manager-master	1/1	Running	0	1h
kube-proxy-dgbgv	1/1	Running	0	1h
kube-proxy-fptbr	1/1	Running	0	1h
kube-scheduler-master	1/1	Running	0	1h
my-custom-scheduler	1/1	Running	0	9s
weave-net-4tfpt	2/2	Running	1	1h
weave-net-6j6zs	2/2	Running	1	1h

pod-definition.yaml

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

   schedulerName: my-custom-scheduler
```

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

To know which scheduler picked it up:
kubectl get events

```
kubectl get events

LAST SEEN   COUNT     NAME       KIND  TYPE      REASON      SOURCE                MESSAGE
9s          1         nginx.15   Pod   Normal    Scheduled   my-custom-scheduler   Successfully assigned default/nginx to node01
8s          1         nginx.15   Pod   Normal    Pulling     kubelet, node01       pulling image "nginx"
2s          1         nginx.15   Pod   Normal    Pulled      kubelet, node01       Successfully pulled image "nginx"
2s          1         nginx.15   Pod   Normal    Created     kubelet, node01       Created container
2s          1         nginx.15   Pod   Normal    Started     kubelet, node01       Started container
```

The message says successfully assigned

See logs of my scheduler:
kubectl logs my-custom-scheduler -n kube-system

Logging and Monitoring:

Documents

Monitoring a K8s cluster:

What would you like/can to monitor:

- Node level metrics, like: Number of nodes, are they healthy, performance metrics like CPU, memory, network, disk utilization.

- POD level metrics, like number of PODs, performance like CPU, memory, etc.

K8s does NOT come with a built in monitoring solution. But there are a few open source solutions available, such as:

- Metrics Server → stores the info In-Memory, NO in the disk, so no history data will be available
- Prometheus
- Elastic Stack
- Datadog → property (payment)
- Dynatrace → property (payment)

Metrics Server: It's a slimmed down version of the deprecated Heapster.

- You can have 1 Metrics Server per cluster.
- Retrieves metrics from each of the nodes and pods and stores them into the memory.
- Does NOT stores data in the disk, no historic data will be available.

How are the metrics generated on the nodes:
Kubelet have a subcomponent called cAdvisor (container Advisor), which is responsible for retrieving performance metrics from PODs and exposing them to the kublet API to make the metric available for the Metrics Server.

How to install it:
git clone https://github.com/kodekloudhub/kubernetes-metrics-server.git
cd kubernetes-metrics-server/
kubectl apply -f .

This will deploy a set of pods, services and rules to enable Metrics Server to pull for performance metrics on the cluster.

Once deployed give the Metrics Server some time to collect data.

Commands:
kubectl top node → will provide CPU and Memory from all the nodes in the cluster-a
kubectl top pod → to have CPU and memory from the PODs in the cluster.

Use the watch command to see when the data is coming through:
watch "kubectl top node "

You will see this messages just installed the metrics server Give it a few minutes to collect the data
master $ **kubectl top node**
Error from server (ServiceUnavailable): the server is currently unable to handle the request (get nodes.metrics.k8s.io)

master $ **kubectl top pod**
Error from server (ServiceUnavailable): the server is currently unable to handle the request (get pods.metrics.k8s.io)

Error if is not installed: This is the error you may see if Metrics Server is not installed and you run top

master $ kubectl top pod
Error from server (NotFound): the server could not find the requested resource (get services http:heapster:)

master $ kubectl top node
Error from server (NotFound): the server could not find the requested resource (get services http:heapster:)

After a while you will see it working:
master $ **kubectl top node**
NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%
master 125m 6% 922Mi 48%
node01 2000m 100% 541Mi 14%

master $ **kubectl top pod**
NAME CPU(cores) MEMORY(bytes)
elephant 13m 29Mi
lion 935m 1Mi
rabbit 965m 1Mi

Managing Application Logs:

Login in Docker:
docker run kodekloud/event-simulator → this runs a container that is constantly showing logs

docker run **-d** kodekloud/event-simulator → -d = detached mode, to run in the background without showing the output

docker logs -f <container_id> → this will show the logs again, -f will stream them live

Login in K8s:

View POD logs: kubectl logs -f event-simulator-pod → -f streams the logs live

What if the POD have more than 1 container:
Add into the POD definition, in spec.containers:

- name: <name_of_the_container>
- name: image-processor

Then run the logs command adding the name of the pod and the container:

event-simulator.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: event-simulator-pod
spec:
  containers:
  - name: event-simulator
    image: kodekloud/event-simulator
  - name: image-processor
    image: some-image-processor
```

kubectl logs -f event-simulator-pod image-processor

To see the name of the containers inside a POD:
kubectl logs <POD_name> -c → -c to see the containers
kubectl logs web-app2 -c

Application Lifecycle Management:

Documentation

Rolling Updated and Roll backs:

This is applied in the Deployments.

When you create a deployment, by default and without specifying nothing, under the deployment spec, the below strategy is created:

```
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
  type: RollingUpdate
```

To do rolling updates and roll backs is a basic feature of Deployment objects.

Rollout and Versioning: When you first create a deployment it triggers a Rollout.
A new Rollout created a new Deployment Revision, Revision 1.

When the application is upgraded (when the container version is updated) a new Rollout is triggered and a new Deployment Revision is created named Revision 2.

This helps us keep track of the changes made to our Deployment and enables us to Roll back to a previous version of our Deployment if necessary.

See status of Rollout:
kubectl rollaut status deployment/myapp_deployment

See Revisions and history of Rollout:
kubectl rollout history deployment/myapp_deployment

Deployment Strategy:

Recreate: Not advisable, will cause **Downtime**. Destroys the previous pods on the deployment and then create new ones. This is not used.

For instance, I have 5 instances of an app. Recreate will first destroy all the instances, once are destroyed will create the new 5 instances.

Rolling Update: The default deployment strategy. Sequentially create one pod with the new image version and destroys an old one and so on. **NO downtime**

When do you update your Deployment, by:

- Updating the container version
- Updating the labels
- Updating the number of replicas
- etc

Enable to Record the the revision history:

When you create a deployment you have to add the below line to the command in order to record the command you are typing, is set to none by default, check kubectl create --help |grep record

kubectl create deployment myapp-deployment –image=nginx **--record**

--record → Records the command in the revision history

If you don't do it, NO information will be recorded.

How do you update your Deployment:

Option 1:

Having a Deployment definition file it is easy to modify this file. Once modified the file apply the changes with:

kubectl apply -f deployment-definition.yaml

A new Rollout is triggered and a new Revision of the Deployment is created.

Option 2:

kubectl set image deployment/myapp_deployment \ nginx=ngins:1.9.1

Remember that this option won't update the myapp_deployment.yaml file, therefore we will have different versions of the deployment, one running and the other in the yaml.

See strategy differences with "describe":

kubectl describe deployment myapp_deployment → will show you the difference between Recreate and RollingUpdate

The field *StrategyType* tells you which one are you using.

If you are using RollingUpdate, the default one, you will see in "Events" how the replicaset are created and destroyed sequencially.

With "Recreate" this is going straight away.

How the upgrade is performed under the hood:

When a new Deployment is created it first creates a ReplicaSet automatically, which in turns creates the number of PODs required to meet the number of replicas.

When you upgrade your application, k8s Deployment object creates a new ReplicaSet under the hood and starts deploying the containers there, at the same time taking down the PODs in the old ReplicaSet following a RollingUpdate stragegy.

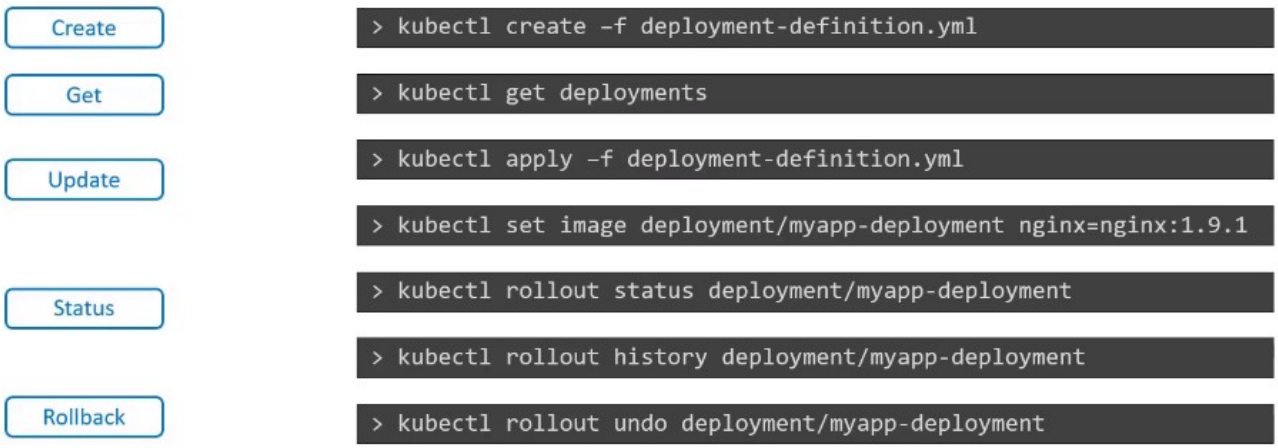
Command:
kubectl get replicasets → will show you both ReplicaSets

Rollback:
Something in the new version of the container is wrong and you want to do a roll back to the previous version of the app.

Command:
kubectl rollout undo deployment/myapp_deployment

Rollback to a certain revision:
kubectl rollout undo deployment frontend --to-revision=2

Commands for rolling updates:



Sends multiple request to a website:
curl-test.sh

```
for i in {1..35}; do
   kubectl exec --namespace=kube-public curl -- sh -c 'test=`wget -qO- -T 2  http://webapp-
service.default.svc.cluster.local:8080/info 2>&1` && echo "$test OK" || echo "Failed"';
   echo ""
```

Define a Command and Arguments for a Container

Documentation

Docker:

When you run an app into docker like nginx or mysql the CMD in the Dockerfile specifies what command to execute when you run the image to create a container.

In nginx or mysql you simply ask the app to start and will be running constantly.

But if you create a container with an OS image, like ubuntu, by default the app (ubuntu) will be stopped with no process running on it, CMD by default specifies only the bash: CMD [“bash”]

bash is not a process like a web or db server, it is a shell that listens for inputs from a terminal and if it can not find the terminal it exits.

Append Command into docker run:
docker run ubuntu sleep 5

How to make this change permanent: Say I want the image to always run the sleep command when it starts.

You have to create your own image from the ubuntu image and specify the sleep command in the Dockerfile:

Different ways to specify the command:

- The command simply as is in the shell form: CMD command param1 → CMD sleep 5
- In a JSON array format: CMD [“command”,”param1”] → CMD [“sleep”,”5”]

ENTRYPOINT: In the Dockerfile the ENTRYPOINT is like the CMD command, it executes what you specify there when you run the container, but it let’s you to pass in a value in the docker run line.

Example:
FROM Ubuntu

ENTRYPOINT [“sleep”]

Then when you run the container you specify how much seconds the sleep command will be executed, 10 in this example:
docker run ubuntu **10**

If you don’t specify any seconds it will give you an error of missing operand, to avoid it use CMD too:

FROM Ubuntu

ENTRYPOINT [“sleep”]

CMD [“5”] → 5 is the default value for the sleep command if nothing is specified when you run the container
docker run ubuntu → without passing any operand will use the default 5 seconds
docker run ubuntu 15 → passing the operand it will be 15 seconds in sleep

You can modify the ENTRYPOINT during the run time:
docker run --entrypoint sleep2.0 ubuntu 15

--entrypoint sleep2.0 → we update(overwrite) the original sleep command to sleep2.0 in the ENTRYPOINT during the run time.

How to apply it into K8s:

In the POD definition as **son of spec.containers**:

command: [“sleep2.0”] → Updates the ENTRYPOINT in the Dockerfile, to the imaginary command sleep2.0

args: [“10”] → Updates the CMD in the Dockerfile, from 5 seconds to 10 seconds in this example.

ENTRYPOINT is the command that will be executed in the runtime
CMD is the default value given

Another option to write it down:
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    command:
    - **"sleep"**
    - **"2000"**

Another option:

```
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    args: ["--color", "green"]
```

or

```
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    command: ["--color", "green"]
```

Environment Variables:

In docker:
docker run -e **APP_COLOR=pink** simple-webapp-color

In K8s: In the definition file, under spec.containers.env as Plain Key Value format

```
env:
   - name: APP_COLOR
     value: pink
```

There are two other ways of setting enviroment variables:
– ConfigMap
        env: APP_COLOR
        valueFrom:
           configMapKeyRef:

– Secrets:
        env: APP_COLOR
        valueFrom:
            secretKeyRef:

ConfigMaps:

When you have a lot of PODs it’s complicated to manage one by one the Environment variables.

ConfigMaps is a centralized way to manage environment variables.

ConfigMaps are used to pass configuration data in the form of key-value pairs in K8s.

There are two faces involved in ConfigMaps:

1. Create the ConfigMap
2. Inject into the POD

Step 1 How to create ConfigMaps:

Imperative:
kubectl create configmap

kubectl create configmap \
 app-config **--from-literal=**APP_COLOR=blue \
 --from-literal=APP_MOD=prod
Or:
 kubectl create configmap my-config --from-literal=key1=config1 --from-literal=key2=config2

kubectl create configmap \
 app-config **--from-file=**app_config.properties

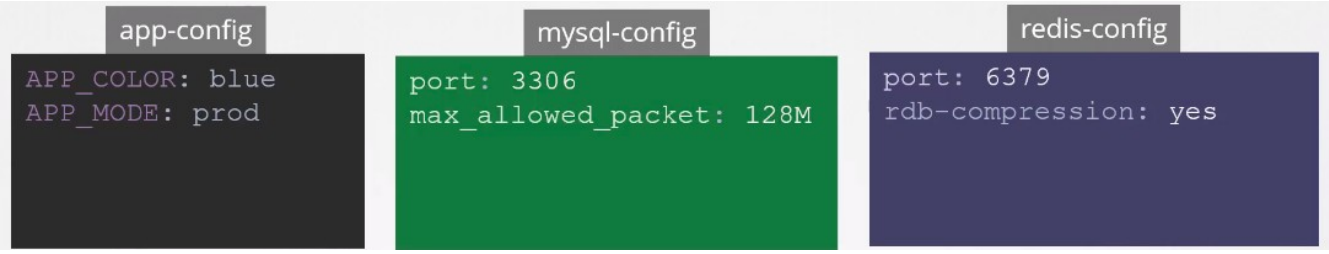
Declarative: We create a definition file.

Like any other object, but instead of spec we have to add **data** field.

config-map.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

You can create as much ConfigMaps as you need:



To view and describe ConfigMaps:

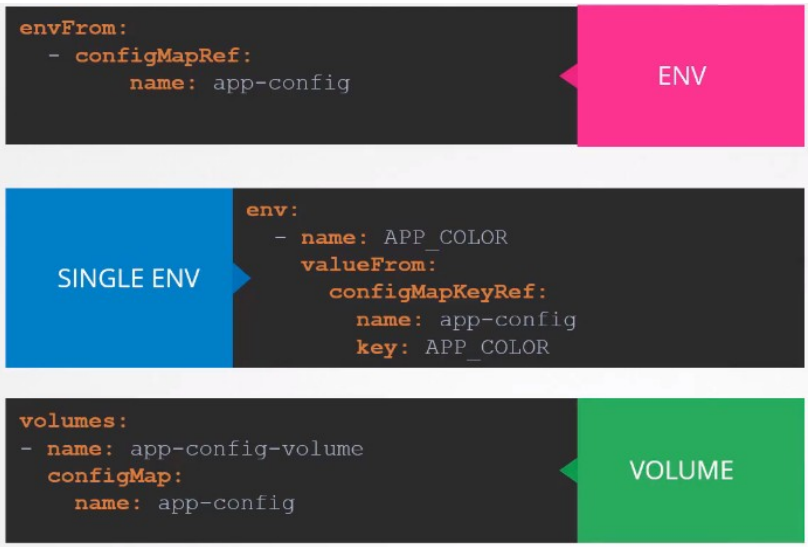
kubectl get configmaps
kubectl describe configmaps

Step 2 Configure the ConfigMap in a POD:

In the POD definition file add it as son of spec.containers.envFrom

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    envFrom:
      - configMapRef:
          name: app-config
```



kubectl explain pods --recursive | grep envFrom -A3

Kubernetes Secrets:

Is like ConfigMaps, you create an object called Secret, stores the necessary data in key-value pairs and you call this object from the POD.

Secrets is stored in an encoded or hash format.

This avoid to store sensitive information like passwords, usernames, hostnames, etc in the PODs and does it in the Secret object.

You can hide the data from being exposed as plain text in the Secret object by using the base64 command in the linux cli, but you can unhide it using the same command so this is only avoiding to show the confidential information to someone that is reading the yaml file but if he have a copy of that file the data will be discovered.

How to create Secrets:

1. Create the Secret
2. Inject it into the POD

Ways to do it:

- Imperative
- Declarative

Imperative:

Using: --from-literal=<key>=<value>

kubectl create secret generic db-secret --from-literal=DB_Host=sql01 --from-literal=DB_User=root --from-literal=DB_Password=password123

This is how looks like:

```
master $ kubectl describe secrets db-secret
Name:         db-secret
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
DB_Password:  11 bytes
DB_User:      4 bytes
DB_Host:      5 bytes
```

Using: --from-file=<path-to-file>

kubectl create secret generic app-secret –from-file=app_secret.properties

Declarative: kubectl create -f <definition.yaml>

secret-data.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: mysql
  DB_User: root
  DB_Password: paswrd
```

You have to hash the sensitive data before, use linux cli:

joaquim@deesktoop:~$ **echo -n 'Password123' |base64**
UGFzc3dvcmQxMjM=

Then you can use it: DB_Password: UGFzc3dvcmQxMjM=

Decode the hash:
joaquim@deesktoop:~$ **echo -n 'UGFzc3dvcmQxMjM=' |base64 --decode**
Password123

Commands:

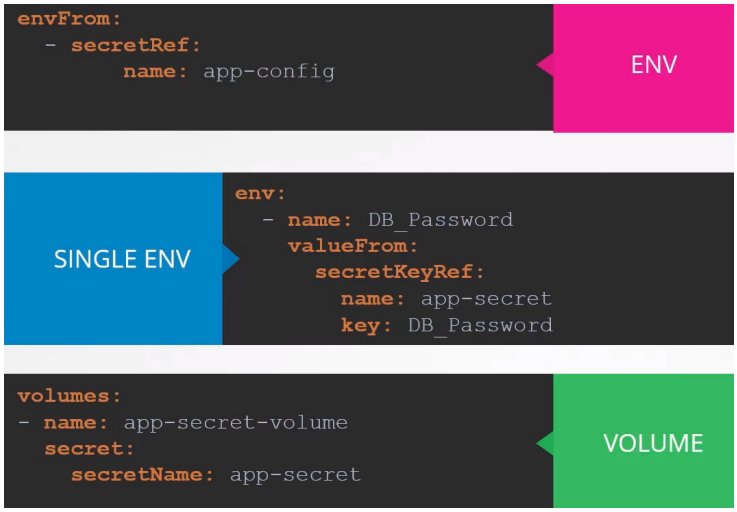
kubectl get secrets
kubectl describe secrets → this hides the values
kubectl describe secrets app-secret -oyaml → to see the hash values too

Configure it to the POD:

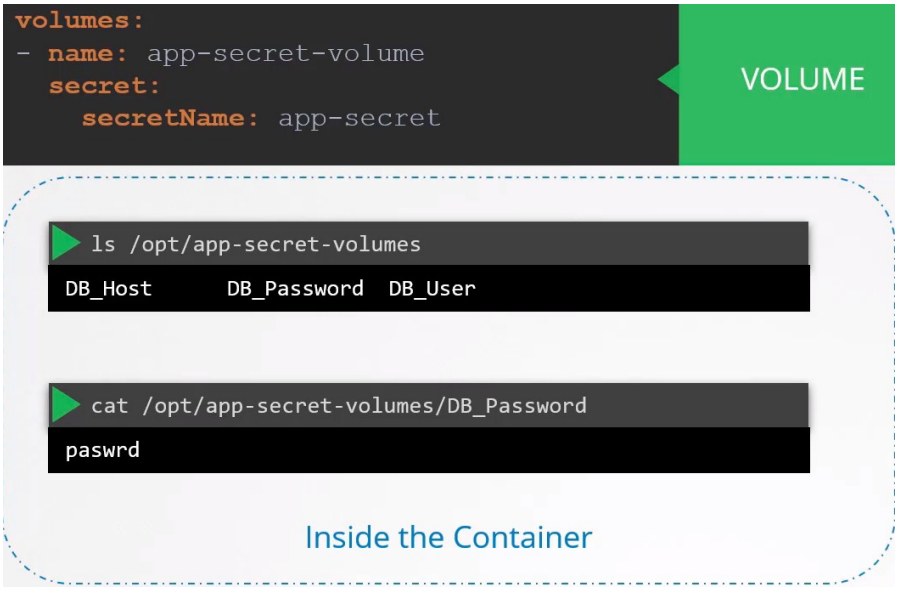
pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    envFrom:
      - secretRef:
          name: app-secret
```

Different ways to inject Secrets in a POD:



With volumes works like this (we have 3 values here, host,pass and user)



Multi-Container PODs:

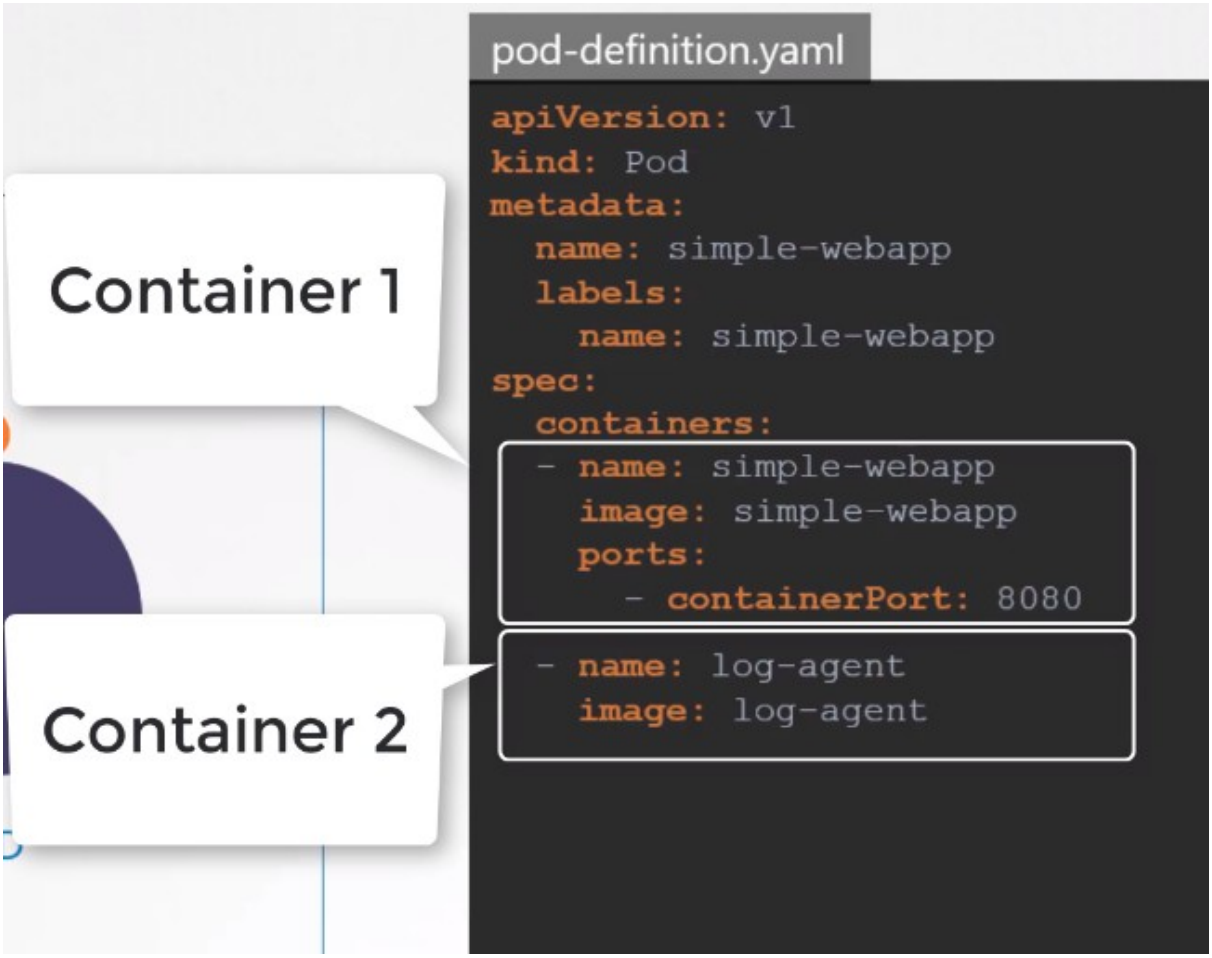
We may need to have in the same POD two or more containers, like a webserver and a logging agent. Then we can scale up and down together sharing the same lifecycle.

They share the same network space (can refer each other as local host) and have access to the same storage volumes.

This way you don’t need to establish services or volume sharing between the pods to enable communications between them.

How to create a multi-container POD:

Add the new container information to the POD definition file:



The container section, under the spec section is an ARRAY (defined by: -).
The reason is an array is to allow multiple containers in a single POD

See logs inside the container:
kubectl -n elastic-stack exec -it app cat /log/app.log

Common patterns for Multi-Pods:
There are 3 common patterns, this is under the CKAD scope:

1. Sidecar → the one we just saw here, for instance one web server and a logging service.
2. Adapter
3. Ambassador

Init Containers:

Documentation

In a Multi-Container Pod context the Init Container is like a normal container/s that is run before the regular one.

You can have several init containers and all of them will run sequentially, if some of them fails the POD will restart itself till the init container works.
Once all the init containers are finished successfully the regular container will start.

Differences from regular containers:

- Init Containers always run to completion
- Each init containers must complete successfully before the next one starts.

When to use it:
When you want to do, and complete, a task before the regular container runs.

Tasks like:

sleep 60

Wait for a Service to be created, using a shell one-line command like:

for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1

Clone a Git repository into a Volume

You don’t need to include these, or other tasks, in your regular app.

Example definition:

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

  # These containers are run during pod initialization
  initContainers:
  - name: install
    image: busybox
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://kubernetes.io
```

Another example: Here we have two init containers.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for myservice; sleep 2; done"]
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for mydb; sleep 2; done"]
```

See logs:

```
kubectl logs myapp-pod -c init-myservice    # Inspect the first init container
kubectl logs myapp-pod -c init-mydb         # Inspect the second init container
```

Get pods: When the init container is running you can see it with kubectl get pods
You will see: Status → Init:0/1
Once the init container is done you will see the usual output: Status → Running

Check it on Description: Will grep everything and highlight init cont
kubectl describe pod purple |grep -iE "init cont|$"

Self Healing Applications:

Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers. The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes. It helps in ensuring enough replicas of the application are running at all times.

Kubernetes provides additional support to check the health of applications running within PODs and take necessary actions through Liveness and Readiness Probes. However these are not required for the CKA exam and as such they are not covered here. These are topics for the Certified Kubernetes Application Developers (CKAD) exam and are covered in the CKAD course.

Cluster Maintenance:

OS Upgrades:
We may need to take down a Node from the cluster to perform maintenance tasks like OS upgrades, hardware maintenance,updating the kernel, applying security patches, etc.

Different Scenarios:

- If the Node came back online immediately the kubelet process starts and the PODs came back online.

- If the Node was down for more than 5 minutes, then the PODs are terminated from that Node. K8s consider them as dead. If the PODs where part of a ReplicaSet then they are recreated on other Nodes.

POD eviction time out: Time it waits for a POD to come back online. This is set on the controller manager with a defautl value of 5 minutes:

kube-controller-manager --pod-evictiontimeout=5m0s ...

When a Node goes offline the Master Node waits up to 5 minutes, then consider the Node dead.

When the Node comes back online after the POD eviction timeout it comes out blank without any PODs scheduled on it.

Drain the Node: You can specify to the PODs of the Node that will be removed for maintenance to be placed in the other worker Nodes in order to avoid losing them.

The PODs are terminated and created in the other worker Nodes. The daemon set PODs are not created in other worker Nodes.

If there are PODs NOT part of a replicaset you won't be able to drain the Node. You will need to --force it to make it happen

The node is also cordoned or marked as unschedulable.

Command:
kubectl drain node01

This will give you this error:
master $ **kubectl drain node01**
node/node01 cordoned
error: unable to drain node "node01", aborting command...

There are pending nodes to be drained:
node01
error: **cannot delete DaemonSet-managed** Pods (use --ignore-daemonsets to ignore): kube-system/kube-flannel-ds-amd64-m7z7m, kube-system/kube-keepalived-vip-xw8tt, kube-system/kube-proxy-kqqpl

kubectl drain node01 --ignore-daemonsets

master $ **kubectl drain node01 --ignore-daemonsets**
node/node01 cordoned
WARNING: ignoring DaemonSet-managed Pods: kube-system/kube-flannel-ds-amd64-m7z7m, kube-system/kube-keepalived-vip-xw8tt, kube-system/kube-proxy-kqqpl
node/node01 drained

ATTENTION: You have to force the drain if there are PODs without being part of a replicaset.
kubectl drain node02 --ignore-daemonsets **--force**

master $ **kubectl drain node02 --ignore-daemonsets --force**
node/node02 already cordoned
WARNING: deleting Pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet: default/hr-app; ignoring DaemonSet-managed Pods: kube-system/kube-flannel-ds-amd64-dlw98, kube-system/kube-keepalived-vip-mzz5h, kube-system/kube-proxy-8gkhv
evicting pod default/blue-8455cd8cd7-nq5mj
evicting pod default/blue-8455cd8cd7-xcc42
evicting pod default/hr-app
evicting pod kube-system/coredns-66bff467f8-qgz2l
pod/hr-app evicted
pod/blue-8455cd8cd7-nq5mj evicted
pod/blue-8455cd8cd7-xcc42 evicted
pod/coredns-66bff467f8-qgz2l evicted
node/node02 evicted

Uncordon the Node:
When the Node comes back online it is still unschedulable, you need to uncordon it, then the PODs can be scheduled on it again.

Command:
kubectl uncordon node-1

Mark a Node unschedulable: It does not terminate or move PODs, just avoid to schedule PODs on it.

Command:
kubectl cordon node-1

Commands:

Get all the pods from a particular Node:

kubectl get pods -A -owide --field-selector spec.nodeName=node01

kubectl get nodes → remember to check the status of the nodes while you are draining and cordoning

---

Kubernetes Software Versions:

kubectl get nodes → shows you the k8s version.

K8s follows a standard software release versioning procedure.

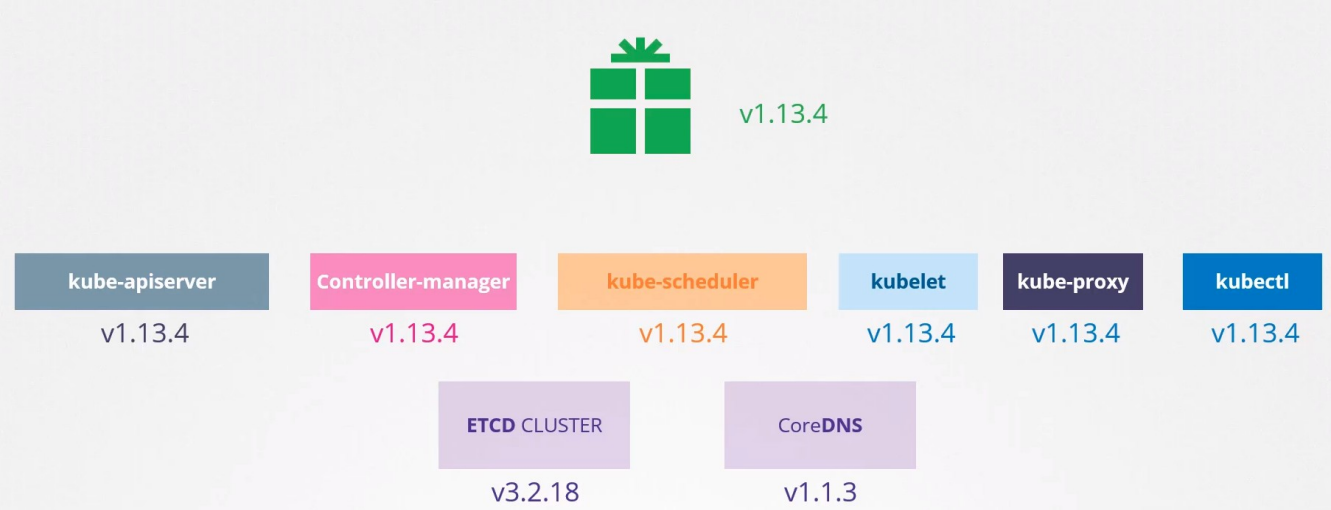
Every few months it comes out with features and functionalities in a Minor release.

Alpha & Beta releases:
All the bug fixes and improvements goes to the tagged Alpha release. In this release the features are disabled by default and maybe have bugs.

From there they make their way to Beta release where the code is well tested, new features are enabled and finally they make their way to the main stable release.

Where to find all the releases:
https://github.com/kubernetes/releases

The downloaded package has all the control plane controllers in it, all of them of the same version except ETCD and CoreDNS because are part of separated projects.



Cluster Upgrade Process:

It is NOT mandatory for all components to have the same release version.

Kube-apiserver is the primary component in the control plane and being the component that all other components talk to NONE of the other components should be at a version HIGHER than kube-apiserver.

The other components can be at the same version or one or 2 versions below, see image:

Kubectl is the exception, it can be 1 version higher.

When to upgrade: At any time K8s supports only 3 minor versions.
If the last one is v1.20 K8s supports v1.20, v1.19 and v1.18.

When v1.21 is released k8s will support v1.21, v1.20 and v1.19

How to upgrade: The recommended approach is to upgrade one minor version at a time.

Upgrade process: Depends on how your cluster is set up.

If your cluster is a Managed Kubernetes Cluster deployed on cloud services provider like Google Cloud or AWS you can upgrade your cluster easily with just a few clicks.

If you deploy your cluster using tools like kubeadm then the tool can help you plan and upgrade the cluster.

If you deploy the cluster from scratch then you manually upgrade the different components of the cluster yourself.

Upgrade with kubeadm: Updating a cluster involves 2 major steps: 1st you update the Master node and then you update the Worker nodes.

Master node upgrade: During the upgrade the Master node goes down but this does NOT mean that the worker nodes and the applications are going down too.

All work loads host in the worker nodes continue to serve users as normal.

Since the Master is down all the management functions are down.

- You can NOT access the cluster using kubectl or other k8s API.
- You can NOT deploy new application or delete or modify the existing ones.
- The Controller Manager don't function either. If a POD fails a new POD won't be created automatically.

But as long and the worker nodes and the PODs are up your applications should be up and users will not be impacted.

Once the upgrade is complete and the cluster is back it should function normally.

Now we have the Master node one version above the rest of the nodes.

Worker Node upgrade strategies: There are different strategies available to upgrade the worker nodes.

- Upgrade the worker nodes at one, PODs are down and apps not working. Will have DOWNTIME.

- Upgrade one node at a time. We upgrade the first node, the workloads are migrated to the other ones. Once is completed we repeat the process sequentially with the other nodes.

- Add new nodes to the cluster, nodes with newer software version. This is specially convenient when you are in a Cloud Environment where you can easely provision new nodes and decomission old ones.
  Nodes with the new software version can be added to the cluster, move the workload over to the new and remove the old node until you finally have all new nodes with the new software version.

How kubeadm upgrade process works:

kubeadm upgrade plan

Will give you information about the current versions, the latest stable version, all the control plane component versions and what versions this can be upgraded too.

It also inform that after upgrading the control plane version you must manually upgrade kubelet on each node (kubeadm does NOT install or upgrade kubelets).

Finally it gives you the command to run for the upgrade:
kubeadm upgrade apply v1.13.4

Also informs that kubeadm tool must be upgraded to the version too.
apt-get upgrade -y kubeadm=1.xx.x.0-00

Let's upgrade:
Remember that we have to upgrade one version each time.

If we are running v1.11 and we want o upgrade to the latest, v1.13, we have to upgrade first to v1.12

1- Upgrade the kubeadm tool: apt-get upgrade -y kubeadm=1.12.0-00

2- Upgrade the cluster: *kubeadm upgrade apply v1.12.0*

At this point you will see in the kubectl get nodes the old versions, v1.11. This is because this command shows the kubelet versions which are not upgraded yet.

3- Upgrade the kubelets, depending on the set up you may not have kubelet components. If we installed the cluster with kubeadm we will have it.
Upgrade kubelet in the master node: *apt-get -y upgrade kubelet=1.12.0-00*
Once is upgraded restart the kubelet service: systemctl restart kubelet

kubectl get nodes will now show that the master node has been upgraded to v1.12.0.

The worker nodes will be at v1.11
4- Now the worker nodes one by one. First evict the workload to another node: *kubectl drain node01*

5- Upgrade the kubeadm and the kubelet packages on node01
apt-get upgrade -y kubeadm=1.12.0-00
apt-get -y upgrade kubelet=1.12.0-00

6- Update the node configuration for the new kubelet version and restart the kubelet service:
kubeadm upgrade node config --kubelet-version v1.12.0
systemctl restart kubelet

We don't have to run kubeadm upgrade apply v1.12.0 in the worker nodes because they don't host kube-apiserver, or kube-scheduler, etc, this is only in the master node.

7- Uncordon the node, to make it scheduable again.
kubectl uncordon node01

Backup & Restore Methods:

Documentation: Operating etcd clusters for Kubernetes

You can store your definition files in github for example, but what happen if someone is creating objects in the imperative way without documenting that information anywhere?

A better approach to backing up resource configuration is to query the **kube-apiserver.**

Commands:

Check ETCD version: kubectl logs -n kube-system etcd-master |grep -i version

Or check the description and search or grep for the Image
kubectl -n kube-system describe pod etcd-master |grep -i image

Address reaching the ETCD cluster from master node:
kubectl -n kube-system describe pod etcd-master |grep "**listen-client-urls**"

Get the certificates location: kubectl -n kube-system describe pod etcd-master
/etc/kubernetes/pki/etcd/

Backup all the running objects:
kubectl get all --all-namespaces -oyaml > all-deploy-services.yaml

View the status of the backup:
ETCDCTL_API=3 etcdctl \
snapshot status snapshot.db
Example:
master $ etcdctl snapshot status /tmp/snapshot-pre-boot.db
d3905d44, 10662, 1485, 3.3 MB

Take snapshot of ETCD db: Here the explanation

Check options available for *snapshot save:*
ETCDCTL_API=3 etcdctl snapshot save **-h**

Usage: ETCDCTL_API=3 etcdctl snapshot save <global_options> <file_to_save>
ETCDCTL_API=3 etcdctl snapshot save <global_options> /tmp/snapshot-pre-boot.db

Mandatory Global options to pass in:

- --cacert="" → --cacert=/etc/kubernetes/pki/etcd/**ca.crt**
- --cert="" → --cert==/etc/kubernetes/pki/etcd/**server.crt**
- --endpoints=[127.0.0.1:2379] → --endpoints=[127.0.0.1]:2379
- --key="" → --key==/etc/kubernetes/pki/etcd/**server.key**

The rest of the global options are optional.

Check if the backup works before running it:
ETCDCTL_API=3 etcdctl **member list** [mandatory global options here]

If you pass the correct parameters you should be able to see listed all the members (nodes) of the cluster.

If the command passed in is wrong will tell you what is failing.

Run the command:
master $ ETCDCTL_API=3 etcdctl snapshot save --endpoints=https://[127.0.0.1]:2379
--cacert=/etc/kubernetes/pki/etcd/ca.crt --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key
/tmp/snapshot-pre-boot.db
Snapshot saved at /tmp/snapshot-pre-boot.db

Restore a Backup: Explanation

Stop kube-apiserver service: service stop kube-apiserver

Check options for *snapshot restore:*
ETCDCTL_API=3 etcdctl snapshot restore -h

Mandatory Options to pass in: All of them except the last two:

- --data-dir="" → location where you want to restore the data
- --initial-advertise-peer-urls="http://localhost:2380" → "https://127.0.0.1:2380"
- --initial-cluster="default=http://localhost:2380" → "master=https//127.0.0.1:2380"
- --initial-cluster-token="etcd-cluster" → give a unique name: etcd-cluster-1
- --name="default" → --name="master"

Mandatory Global options to pass in:

- --cacert="" → --cacert=/etc/kubernetes/pki/etcd/**ca.crt**
- --cert="" → --cert==/etc/kubernetes/pki/etcd/**server.crt**
- --endpoints=[127.0.0.1:2379] → --endpoints=127.0.0.1:2379 → is always the same
- --key="" → --key==/etc/kubernetes/pki/etcd/**server.key**

Command to run:

```
ETCDCTL_API=3 etcdctl --endpoints=https://[127.0.0.1]:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt \
    --name=master \
    --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key \
    --data-dir /var/lib/etcd-from-backup \
    --initial-cluster=master=https://127.0.0.1:2380 \
    --initial-cluster-token=etcd-cluster-1 \
    --initial-advertise-peer-urls=https://127.0.0.1:2380 \
    snapshot restore /tmp/snapshot-pre-boot.db
```

Here the command with the successful output:

```
master $ ETCDCTL_API=3 etcdctl --endpoints=https://[127.0.0.1]:2379 --cacert=/etc/kubernetes/pki/etcd/ca.crt \
>     --name=master \
>     --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key \
>     --data-dir /var/lib/etcd-from-backup \
>     --initial-cluster=master=https://127.0.0.1:2380 \
>     --initial-cluster-token=etcd-cluster-1 \
>     --initial-advertise-peer-urls=https://127.0.0.1:2380 \
>     snapshot restore /tmp/snapshot-pre-boot.db
2020-10-05 08:25:12.258069 I | mvcc: restore compact to 9767
2020-10-05 08:25:12.269542 I | etcdserver/membership: added member e92d66acd89ecf29 [https://127.0.0.1:2380] to cluster 7581d6eb2d25405b
```

Apply the changes to the static ETCD pod: /etc/kubernetes/manifests/etcd.yaml

– Modify the directory where we restored the snapshot: --data-dir=/var/lib/etcd-from-backup

– Add this line: - --initial-cluster-token=etcd-cluster-1

– Replace mountPath: - mountPath: /var/lib/etcd-from-backup

– Replace volumes.hostPath.path: path: /var/lib/etcd-from-backup

As soon as you save the file the changes will be applied.

Wait for the new etcd container spins up: watch docker ps |grep etcd

Check member list: ETCDCTL_API=3 etcdctl **member list** [...]

Restart Daemon: systemctl daemon-reload
Restart etcd: service etcd restart
Start api-server: service kube-apiserver start

Check pods, deployments,svc: kubectl get pods,svc,deployments

PROBLEMS:
Make sure that in the mandatory options the http is specified as http**S**
--initial-cluster=master=https://127.0.0.1:2380
--initial-advertise-peer-urls=https://127.0.0.1:2380

More documentation:
https://www.youtube.com/watch?v=qRPNuT080Hk&ab_channel=CNCF%5BCloudNativeComputingFoundation%5D

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

---

Security:

Secure the servers:

- Root access disabled.
- Password based authentication disabled.
- Only SSH Key based authentication available.
- And all the necessary security measures.

Kube-apiserver: The 1st line of defense must be to control the access to the kube-apiserver itself. We can interact with it via kubectl or accessing the api directly.

- **Who can access the cluster** (kube-apiserver): Is defined by the authentication mechanisms. There are different ways to authenticate to the api-server:
  - Files – Username and Passwords → not secure
  - Files – Usernames and Tokens. → not secure
  - Certificates
  - External Authentication providers like LDAP, Kerberos, etc.
  - Service Accounts →for machines.

- **What can they do:** This is defined by authorization mechanisms:
  - RBAC Authorization → Users are associated to groups with specific permissions.
  - ABAC → Attribute Based Access Control
  - Node Authorization
  - Webhook Mode
  - etc

TLS Encryption: All the comms in the cluster between the different components, api-server, controller manager, etcd, kube proxy, etc is secured by TLS Encryption.

Network Policies: By default all the objects within the cluster can communicate between each other. To limit this you can apply network policies.

Authentication:

Who can access the cluster or it's apps:

- **End Users:** Who access the applications inside the cluster.
- **3rd party applications:** They access the cluster for integration purposes. Other processes, services or applications that require access to the cluster.
- **Human Users:** Developers, admins, etc.

End Users: The security to access the applications is managed by those application.

3rd Party Applications: Service Accounts. This is not part of the CKA curriculum, only in CKAD.
K8s can manage service accounts You can create Service Accounts

Human Users: Accessing the cluster for administrative purposes:

K8s does not manage user accounts natively. It relies in an external source like a file with the user details, certificates or 3rd party identity services like LDAP or Kerberos.

You can NOT create users or list users in k8s.

All user access is managed by the kube-apiserver. Either you are accessing the cluster through the kubectl or the api directly. Api access example: curl https://kube-server-ip:6443/

All of these request go to the kube-apiserver which authenticate the user before processing it.

Authentication Mechanisms: There are different authentication mechanisms that we can configure.

Basic - Files – Username and Passwords: **Method NOT recommended, Not Secure**
You can create a list of users, passwords, UID and groups in a .csv file and use it as a source for user information.

We must specify in the kube-apiserver.service this option:
--basic-auth-file=user-details.csv
And must restart the kube-apiserver to apply the changes. If you set up

If you set up your cluster using the kubeadm tool, you must modify the static pod /etc/kubernetes/manifest/kube-apiserver.yaml and add the line there and restart the api-server.

To authenticate using the basic credentials when accessing the api-server, specify the user and password in a curl like this:
curl https://kube-server-ip:6443/api/v1/pods -u "user1"password123"

Files – Usernames and Tokens: Similar to usernames and passwords. **Method NOT recommended, Not Secure**

You can have a static token (some kind of hash) file instead of the passwods.

To use this option you have to specify it in kube-apiserver as:
--token-auth-file=user-details.csv

To pass it to the api-server use:
curl https://kube-server-ip:6443/api/v1/pods –header "Authorization: Bearer KdfsdDFe34SDFEEF3fs"

TLS Basics & Certificates:

Symmetric Encryption: You encrypt your user and password and you send the data to the destination server, for instance your online bank.

Then you send the key to the destination server to let him to unnencrypt the data.

But a hacker can sniff both communications and will have the encrypted user/password and the key to unencrypt it, so this communication is not secure.

Symmetric encryption uses the same key to encrypt and unencrypt the data and the key have to be exchanged through the network between the sender and the receiver, therefore this is not secure.

Asymmetric Encryption: Instead of using a single key to encrypt and decrypt data, asymmetric encryption uses a pair of keys, a Private key and a Public key.

- **Private key:** Don't have to be shared or sent nowhere. You have to keep it safe.
- **Public Key:** You can share it and send it to everybody.

How Asymmetric encryption works:
You can encrypt your data with the Public key, then you can send it through internet.

If a hacker sniffs the encrypted data won't be able to decrypt it because he don't have the Private key. The Private key is NEVER sent through internet or nowhere. Is always in the destination server.

The Public key is sent and shared with everybody, is public accesible but if a hacker takes it he could only Encrypt data with it, he can NOT Decryp.

Example using SSH asymmetric encryption: You have a server and you don't want to use user/password to access it.

You decided to use keypairs. You generated a public and private keypair.

Command: **ssh-keygen**

It creates two files:

- **id_rsa** → the private key, DON'T share it.
- **id_rsa.pub** → public key, you can share it and send it through internet.

You then send the public key, id_rsa.pub, normally to ~/.ssh/authorized_keys to the remote server. You can send it to as many remote servers you need.

Then you can ssh the remote server:
ssh -i /path_to_the_private_key username@server1

TLS Encryption: Like with SSH, we have to use TLS to encrypt our data using public and private keys.

The way to proceed it's a bit different. We will use Symmetric and Asymmetric encryption all together.

Example:
In this example we will connect to our online bank, all the operation is being carried between the bank server and our browser, we don't have to do or see nothing, only connecting to the website and typing our credentials when the connection is encrypted.

The communications with our bank will be using Symmetric Encryption, so we will send our data encrypted and we will send our Private key to the bank in a safe way to avoid our private key to be sniffed by hackers.

1. From our browser we type the url of our online bank, https://my-bank.com
2. The bank send us its public key (my-bank.pem)
3. Our browser will use the bank's public key, my-bank.pem to encrypt our Symmetric private key.
4. Our browser will send our Symmetric key encrypted with the banks public key to the bank server. If a hacker sniff this communication he could do nothing, he will have the banks public key and our encrypted data. The hacker would only be able to Encrypt things with the bank public key, but he never will be able to Decrypt the content of my data because the hacker will need the private key from the bank to do it.
5. The bank will receive the encrypted data containing our Symetric key and will decrypt it using its private key (which never have been shared).
6. Then the bank will have our Symmetric key unencrypted in his server.
7. Now in our browser we can type our user and password to get inside our bank account. This communication will be encrypted with OUR Symmetric key and will be sent to the bank. Our Symmetric key won't NEVER be sent through internet as plain text.

8. The bank will receive our user and password encrypted and will decrypt it using our Symmetric key received in the step 5 and we will able to log in into our account in a safe way.
9. Any communication will be encrypted with our Symetric key between both sides making it safe.

The hole thing here is that we are passing our Symmetric key to the remote server using Asymmetric encryption.
Once the remote server have our Symmetric key in a safe way the communication can proceed using symmetric encryption, we encrypt the data and the bank server decrypts it using our symmetric key sent previously.

Commands:
Generate Private and Public key pair:
openssl genrsa -out my-bank.key 1024 → generate the private key
openssl rsa -in my-bank.key -pubout > mybank.pem → generate the public key

my-bank.key → private key
mybank.pem → public key

Certificates: A certificate is used to guarantee trust between two parties during a transaction, for example when a user tries to access a webserver, a certificate ensure that the comms between the user and the server is encrypted and the server is who it says it is.

When the server sends the public key, it sends a certificate that has the public key in.
The certificate is like an actual certificate but in a digital format.

It has in formation about:
- Who the certificate is issued to
- The public key of that server
- The location of that server
- Serial Number
- Validity date
- Name (domain) and Alternative names
- Etc

This is how it looks:

```
Certificate:
    Data:
        Serial Number: 420327018966204255
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not After : Feb  9 13:41:28 2020 GMT
        Subject: CN=my-bank.com
 X509v3 Subject Alternative Name:
                DNS:mybank.com, DNS:i-bank.com,
                DNS:we-bank.com,
        Subject Public Key Info:
                    00:b9:b0:55:24:fb:a4:ef:77:73:
                    7c:9b
```

Every Certificate has a **name** on it. The person or subject to who the Certificate is issued to and that is very important as that is the field that helps you validate their identity.

If this is for a webserver this **name** must match what the user types in the URL on his browser.

If the webserver have more that one name this must be specified too in the certificate as alternative names.

Anyone can generate a certificate like this. A hacker can autogenerate a certificate saying that is Google, or your bank.

To avoid this to happen the certificate have to be signed.

You can sign your certificates too , **self sign certificate**, but everybody will know that is NOT a safe certificate. Our browser will detect that.

Browsers have a built in certification validation mechanism and will display a warning message ERR_CERT_AUTHORITY_INVALID if you are using a fake certificate.

Companies are using **Certificate Authority (CA)** which are reliable and famous companies that after validating that you are who you are saying they sign your certificates.

C's like:

- Symantec
- DigiCert
- GlobalSign
- Comodo
- Etc

How do you get your certificates signed by a CA: Using my-bank.key generated previously.

Generate a CSR: Certificate Signing Request. Which is like a Certificate with all our details but WITHOUT SIGNATURE.

```
[cloud_user@bfd51f75391c Documents]$ openssl req -new -key my-bank.key -out my-bank.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [XX]:US
State or Province Name (full name) []:CA
Locality Name (eg, city) [Default City]:
Organization Name (eg, company) [Default Company Ltd]:MyOrg, Inc
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:mydomain.com
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
```

A challenge password []:
An optional company name []:
[cloud_user@bfd51f75391c Documents]$

See the CSR file: openssl req -noout -text -in my-bank.csr

Now we have three documents:
my-bank**.csr** → Certificate signing request just created.
my-bank.key → private key
mybank.pem → public key

Send the CSR to the CA for signing:
Now we can send the csr to the CA for signing.
The CA will verify my details and if all is fine they will sign the certificate and will send it back to us.

Now I have a signed certificate by a CA that the browsers will trust.

If a hacker tries to have a certificate signed by the CA trying to use google, your bank domain will be rejected during the validation phase by the CA and his certificate will be rejected by the CA.

The CA uses different technics to make sure that you are the actual owner of that domain.

How our browser knows that the CA is valid:
In the previous steps the CA signed the certificate, but how our browser can know that the CA that have signed the certificate is a valid one and not a fake that tries to steel your private keys and so.

The CA's themselves have a set of public and private key pairs.

The CA's uses their private keys to sign the certificates and public keys of all the CA's are built in into the browsers. The browser uses the public key of the CA to validate that the certificate was signed by the CA themselves.

You can actually see the public keys of the CA's in the settings of your browser or OS.

How to see the CA certificates in linux:
```
awk -v cmd='openssl x509 -noout -subject' '
>     /BEGIN/{close(cmd)};{print | cmd}' < /etc/pki/ca-trust/extracted/openssl/ca-bundle.trust.crt
subject=CN = ACCVRAIZ1, OU = PKIACCV, O = ACCV, C = ES
subject=C = ES, O = FNMT-RCM, OU = AC RAIZ FNMT-RCM
subject=C = IT, L = Milan, O = Actalis S.p.A./03358520967, CN = Actalis Authentication Root CA
subject=C = US, O = AffirmTrust, CN = AffirmTrust Commercial
subject=C = US, O = AffirmTrust, CN = AffirmTrust Networking
subject=C = US, O = AffirmTrust, CN = AffirmTrust Premium
subject=C = US, O = AffirmTrust, CN = AffirmTrust Premium ECC
subject=C = US, O = Amazon, CN = Amazon Root CA 1
subject=C = US, O = Amazon, CN = Amazon Root CA 2
subject=C = US, O = Amazon, CN = Amazon Root CA 3
subject=C = US, O = Amazon, CN = Amazon Root CA 4
subject=CN = Atos TrustedRoot 2011, O = Atos, C = DE
[...]
```

Validate Internal sites:
Till now we saw how to validate public sites, but if we don't need it because the sites are going to be used within our organization, like for apps like our internal email application or our payroll, etc.

For that you can host your own private CA.

Most of the CA companies have a private offering of their services, a CA server you can deploy internally within your company.

You can then have the public key of your internal CA server installed in all the employees browsers and establish secure connectivity within our organization.

How to add a trusted CA certificate to your browser.

PKI, Public Key Infrastructure: Is the hole infrastructure including the CA, the servers, the people and the process of generating, distributing and maintaining digital certificates is know as PKI

Clarification:
You can encrypt data with both public and private keys and **ONLY** decrypt data with the other.
You can NOT encrypt data with one and decrypt it with the same.

You must be careful what you encrypt your data with. If you encrypt your data with your PRIVATE key then remember ANYONE with your public key will be able to decrypt your data.

Naming Convenctions:
Usually **Private keys** contain the word "key" in its name:

- server.key
- server-key.pem
- client.key
- client-key.pem

Public keys:

- server.crt
- server.pem
- client.crt
- client.pem

How to encrypt and decrypt your own documents:

Encrypt using the public key generated before:
openssl rsautl -encrypt -pubin -inkey **mybank.pem** -in plaintext.txt -out encrypted.txt

Decrypt using the private key generated before:
openssl rsautl -decrypt -inkey **my-bank.key** -in encrypted.txt -out plaintext.txt

---

TLS in K8s:

Server Certificates: The one containing the public key of the server, my-bank.pem in the previous example.

Root Certificates: The CA (Certificate Authority) have it's own private/public keys that uses to sign server certificates. These are called Root certificates.

Client Certificates: The ones in the client side.

All the communications in the cluster must be secure. Communications between nodes, services, clients, etc.

All the servers have to have Server Certificates and all the clients have to have Client Certificates.

Server Certificates for servers:

Kube-apiserver → Exposes an HTTPS API endpoint that other components as well as external users use to manage the cluster. It is a server and requires server certificates to communicate with his clients. We generate a certificate containing the public key: apiserver.crt and the private key: apiserver.key

ETCD-Server: Stores all the information about the cluster. Requires a certificate and a private key, etdcserver.crt and etcdserver.key

Kubelet server: On the worker node. Exposes an https API endpoint that the kube-apiserver talks to to interact with the worker nodes. Kunelet.cert and kubelet.jey

Client Certificates for clients:

We are the clients that access the kube-apiserver . The admin user requires an admin.crt and admin.key to access the kube-apiserver.

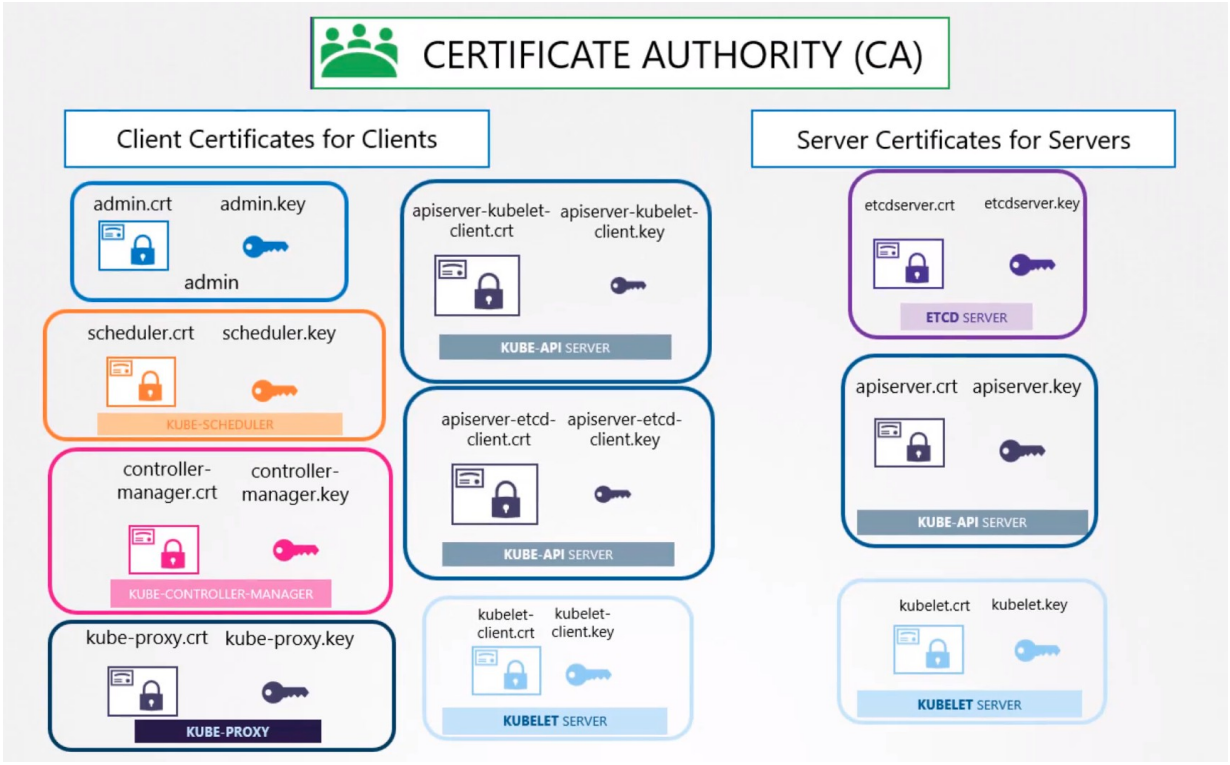
Kube-scheduler: Is a client that access the kube-apiserver too. Also requires a certificate for authentication to the kube-apiserver.

Kube-control-manager: It access too the kube-apiserver. Also requires a certificate for authentication to the kube-apiserver.

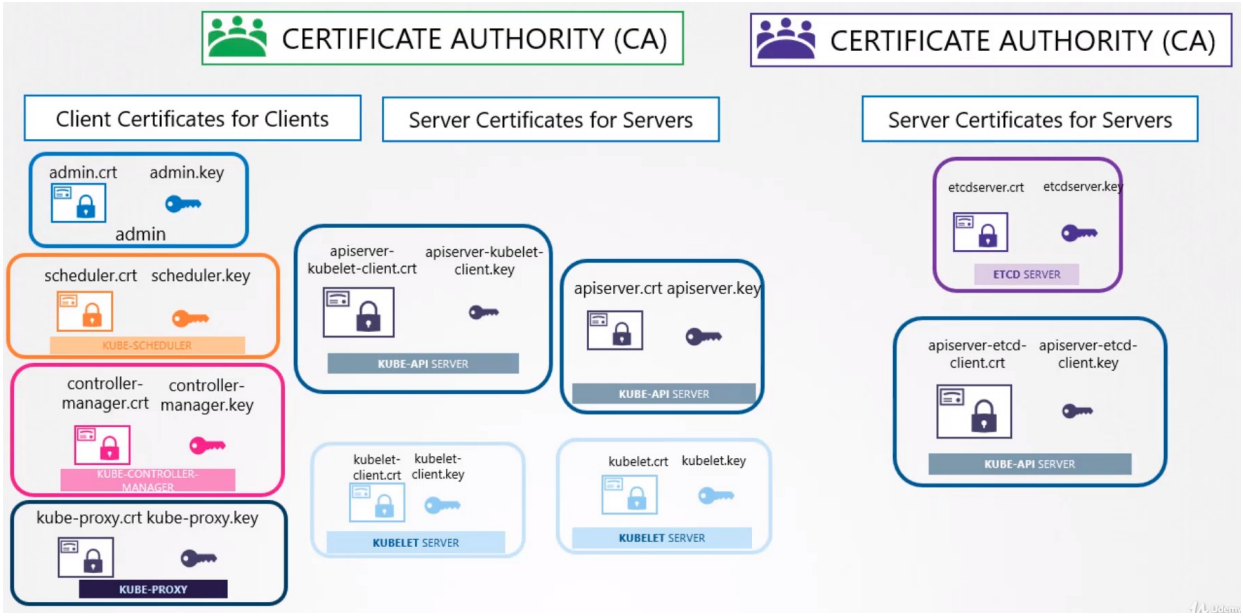
Kube-proxy: Also requires a certificate for authentication to the kube-apiserver.

ETCD: Is a server but the api server is the only one that communicates with him, therefore for the api server etcd is like another client and it needs to authenticate. Api server can use his keys to authenticate or can generate new ones to communicate with etcd: apisever-etcd-client.crt and apiserver-etcd-client.key.

We will need a CA to sign all the certificates. K8s requires to have at least a CA, but in fact you can have more than one.



One for all the components in the cluster and another specific for etcd



TLS in Kubernetes - Certificate Creation: The Hard Way

To generate certificates there are several tools available:

- EASYRSA
- OPENSSL
- CFSSL
- etc.

Following the last image we will start generating the CA certificate.

Root certificate:

CA Certificate:

- **Create the private key:** openssl genrsa -out **ca.key** 2084
- **Create the csr:** openssl req -new -key **ca.key** -subj "/CN=KUBERNETES-CA" -out **ca.csr**
- **See the csr:** openssl req -noout -text -in **ca.csr**
- **Sign the csr:** openssl x509 -req -in **ca.csr** -signkey **ca.key** -out **ca.crt**
- **See the crt:** openssl x509 -in **ca.crt** -text -noout

Since the certificate is for the CA itself is self signing it using it's own private key generated in the previous step.

We will use the CA key/pair to sign all the certificates going forward.

Client Certificates:

Admin User:

1. **Generate the private key:** openssl genrsa -out **admin.key** 2084
2. **Create the csr:** openssl req -new -key admin.key -subj "/CN=kube-admin/O=**system:masters**" -out admin.csr
3. **Sign certificate:** openssl x509 -req -in **admin.csr** -CA **ca.crt** -CAkey **ca.key** -CAcreateserial -out **admin.crt**

2 – About the name, it doesn't have to be kube-admin could be anything but this is the name that kubectl client authenticates with and when you run the kubectl command. In the audit logs and elsewhere this is the name you will see.

How do you differentiate the Admin user from any other users. The user account needs to be identified as an ADMIN user and not just another basic user.

To do so add the group details for the user in the certificate.

In this case a group named SYSTEM:MASTERS exists on the k8s cluster with administrative privileges.

You must this information in the csr as you can see in the step 2.

The O= parameter is equivalent to this when you don't specify -subj when creating the csr:
Organization Name (eg, company) [Default Company Ltd]:

3 – You must create a CASerial when using the x509 option, other wise the certificate will be generated but will show this error:

[root@redhat8:~/certificates]# **openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt**
Signature ok
subject=CN = kube-admin, O = system:masters
Getting CA Private Key
ca.srl: No such file or directory
139876449756992:error:06067099:digital envelope routines:EVP_PKEY_copy_parameters:different parameters:crypto/evp/p_lib.c:93:
139876449756992:error:02001002:system library:fopen:No such file or directory:crypto/bio/bss_file.c:72:fopen('ca.srl','r')
139876449756992:error:2006D080:BIO routines:BIO_new_file:no such file:crypto/bio/bss_file.c:79:
[root@redhat8:~/certificates]#

You can autogenerate a serial add this option when you sign the certificate: -Cacreateserial

Correct output:
[root@redhat8:~/certificates]# openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key **-CAcreateserial** -out admin.crt
Signature ok
subject=CN = kube-admin, O = system:masters
Getting CA Private Key
[root@redhat8:~/certificates]#

Kube-scheduler: Is a system component, so it name must be prefix with the keyboard SYSTEM:

- **Create the private key:** openssl genrsa -out scheduler.key 2084
- **Generate the csr:** openssl req -new -key scheduler.key -subj "/CN=**SYSTEM:**KUBE-SCHEDULER" -out scheduler.csr
- **Sign the certificate:** openssl x509 -req -in scheduler.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out scheduler.crt

Kube-controller-manager: The same here, you must prefix SYSTEM:

- **Create the private key:** openssl genrsa -out controller-manager.key 2084
- **Generate the csr:** openssl req -new -key controller-manager.key -subj "/CN=SYSTEM:KUBE-CONTROLLER-MANAGER" -out controller-manager.csr
- **See the csr:** openssl req -noout -text -in controller-manager.csr
- **Sign the certificate:** openssl x509 -req -in controller-manager.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out controller-manager.crt
- **See the certificate:** openssl x509 -in controller-manager.crt -text -noout

Kube-Proxy: Here is NOT necessary to add system: to the CN

- **Create the private key:** openssl genrsa -out kube-proxy.key 2084
- **Generate the csr:** openssl req -new -key kube-proxy.key -subj "/CN=KUBE-PROXY" -out kube-proxy.csr
- **Sign the certificate:** openssl x509 -req -in kube-proxy.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out kube-proxy.crt

Server Certificates:

ETCD Server:

- **Generate the private key:** openssl genrsa -out etcdserver.crt 2084
- **Generate the csr:** openssl req -new -key etcdserver.key -subj "/CN=ETCD-SERVER" -out etcdserver.csr
- **Sign the certificate:** openssl x509 -req -in etcdserver.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out etcdserver.crt

ETCD Server can be deployed as a cluster across multiple servers as HA. In that case we must generate PEER certificates for each member of the etcd cluster.

- **Generate the private key:** openssl genrsa -out **etcdpeer1**.key 2084 → each etcd peer server will have a different number, the next one will be etcdpeer2.key and so on.
- **Generate the csr:** openssl req -new -key e**tcdpeer1.key** -subj "/CN=**ETCD-PEER**" -out etcdpeer1.csr
- **Sign the certificate:** openssl x509 -req -in etcdpeer1.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out etcdpeer1.crt

Once the Certificates have been generated specify them when starting the ETCD server.

```
cat etcd.yaml
- etcd
    - --advertise-client-urls=https://127.0.0.1:2379
    - --key-file=/path-to-certs/etcdserver.key
    - --cert-file=/path-to-certs/etcdserver.crt
    - --client-cert-auth=true
    - --data-dir=/var/lib/etcd
    - --initial-advertise-peer-urls=https://127.0.0.1:2380
    - --initial-cluster=master=https://127.0.0.1:2380
    - --listen-client-urls=https://127.0.0.1:2379
    - --listen-peer-urls=https://127.0.0.1:2380
    - --name=master
    - --peer-cert-file=/path-to-certs/etcdpeer1.crt
    - --peer-client-cert-auth=true
    - --peer-key-file=/etc/kubernetes/pki/etcd/peer.key
    - --peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
    - --snapshot-count=10000
    - --trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

Kube-API Server:

Kube-apiserver is the most popular of all components within the cluster everyone talks to it and every operation goes through the kube-apiserver, if you need information you talk to the api-server.

It goes by many names and aliases within the cluster. It's real name is KUBE-API SERVER but some call it kubernetes, other kubernetes.default, kubernetes.default.svc, kubernetes.default.svc.cluster.local, or by it's IP: 10.96.0.1, 172.17.0.87.

So all of these names must be present in the certificate generated for the kube-apiserver. Only then those refering to the api-server by these names would be able to establish a valid connection.

- **Generate the private key:** openssl genrsa -out apiserver.key 2084
- **Create the csr passing openssl.cnf as an option:** openssl req -new -key apiserver.key -subj "/CN=kube-apiserver" -out apiserver.csr **-config openssl.cnf**
- **Sign the certificate:** openssl x509 -req -in apiserver.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out apiserver.crt

openssl.cnf → is giving error when you create the csr

```
[ req ]
req_extensions = v3_req
[ v3_req ]
basicConstraints=CA:FALSE
keyUsage = nonRepudiation,
subjectAltName = @alt_names
[ alt_names ]
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster.local
IP.1 = 10.96.0.1
IP.2 = 172.17.0.87
```

```
[root@redhat8:~/certificates]# openssl req -new -key apiserver.key -subj "/CN=kube-apiserver" -out apiserver.csr -config openssl.cnf
Error Loading request extension section v3_req
140432861652800:error:2206D06C:X509 V3 routines:X509V3_parse_list:invalid null name:crypto/x509v3/v3_utl.c:360:
140432861652800:error:22097069:X509 V3 routines:do_ext_nconf:invalid extension
string:crypto/x509v3/v3_conf.c:93:name=keyUsage,section=nonRepudiation,
140432861652800:error:22098080:X509 V3 routines:X509V3_EXT_nconf:error in extension:crypto/x509v3/v3_conf.c:47:name=keyUsage,
value=nonRepudiation,
```

Kubelet Server: Is a https API server that runs on each node responsible for managing he node. Kube-apiserver talks to kubelet to monitor it and to send information about what pods to schedule on that node. You will need certificates to communicate with the api-server.

Each kubelet certificate will be named with the name of the node.

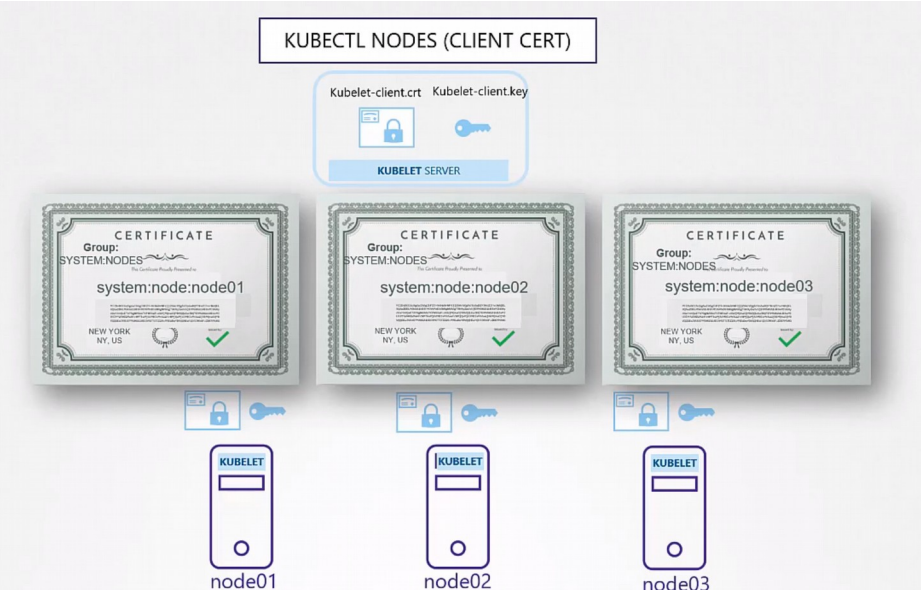
- **Generate the private key:** openssl genrsa -out **kubelet1**.key 2084
- **Generate the csr:** openssl req -new -key kubelet1.key -subj "/CN=**node01**" -out kubelet1.csr
- **Sign the certificate:** openssl x509 -req -in kubelet1.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out kubelet.crt

Once the cert is created use it in the kubelet-config.yaml file. You must do this for each node in the cluster.

kubelet-config.yaml (node01)

```
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  x509:
    clientCAFile: "/var/lib/kubernetes/ca.pem"
authorization:
  mode: Webhook
clusterDomain: "cluster.local"
clusterDNS:
  - "10.32.0.10"
podCIDR: "${POD_CIDR}"
resolvConf: "/run/systemd/resolve/resolv.conf"
runtimeRequestTimeout: "15m"
tlsCertFile: "/var/lib/kubelet/kubelet-node01.crt"
tlsPrivateKeyFile: "/var/lib/kubelet/kubelet-
node01.key"
```

Kubelet Client Certificates: Used by the kubelet to communicate with the API-Server
CN = system:node:node01
O = SYSTEM:NODES



CA Certificate Root Certificate:

For clients to validate the certificate sent by the server and viceversa they all need a copy of the CA public certificate, ca.crt or the name you have chosen.

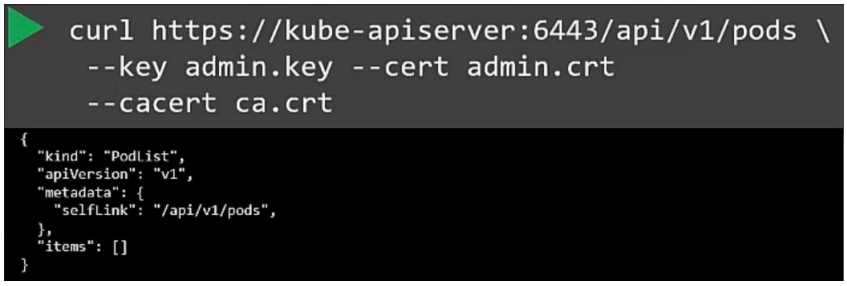
In K8s for its various components to verify each other they all need a copy of the CA root certificate, ca.crt.

When you configure a server or a client with certificates you must specify where the ca.crt root is.

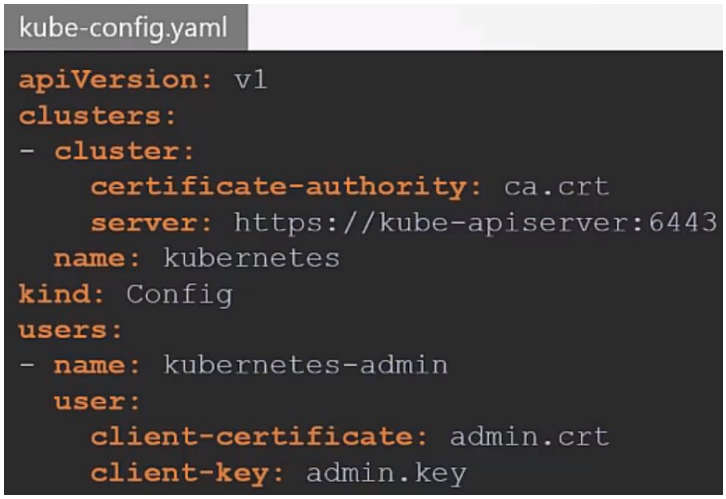
Where to use the Certificates:

Take the admin certificate for instance. You can manage the cluster by using this certificate instead of a user/password. This way is much more secure.

One option: You can use it in an API call specifying: This API call will show you the podlist
curl https://kube-apiserver:6443/api/v1/pods \
 --key admin.key --cert admin.crt
 --cacert ca.crt



Other option is to use a configuration file called kube-config.yaml, where you specify all the information you need. That is what most of the k8s clients uses.



View Certificate Details:

How to see certificates in an existing environment.

You have to know how the cluster was set up. There are different solutions to set up a cluster and they use different methods to generate and manage certificates.

The hard way: Deploy the cluster from scratch. You generate all the certificates by yourself.
/etc/systemd/system/kube-apiserver.service

It deploy all the components as native services on the nodes.

Kubeadm: Automated provisioning tool that autogenerates all the certificates in the cluster.
/etc/kubernetes/manifests/kube-apiserver.yaml

It deploys all the components as PODs.

In this example we will use a cluster created by kubeadm.

Get a template like this:

kubeadm

Component	Type	Certificate Path	CN Name	ALT Names	Organization	Issuer	Expiration
kube-apiserver	Server						
kube-apiserver	Server						
kube-apiserver	Server						

Example spreadsheet

Get the certificate path: Go to /etc/kubernetes/manifests/kube-apiserver.yaml
And check for the folder in which all the certificates are stored:

/etc/kubernetes/pki

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.32
    - --allow-privileged=true
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --disable-admission-plugins=PersistentVolumeLabel
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
    - --secure-port=6443
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-cluster-ip-range=10.96.0.0/12
    - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
```

Now you have the folder go certificate by certificate looking inside for more details, expiry date, etc, in order to fill the spreadsheet.

Open the certificate:
openssl x509 -in etcdserver.crt -text -noout

Name of the certificate: Subject: CN = ETCD-SERVER
Alternative names: If it have them.
Expiry date: Not After : Nov 11 10:30:10 2020 GMT
Issuer: C = XX, L = Default City, O = Default Company Ltd, CN = KUBERNETES-CA

Alternative names for apiserver.crt:
X509v3 Subject Alternative Name:
DNS:master, DNS:kubernetes, DNS:kubernetes.default, DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster.local, IP Address:10.96.0.1, IP Address:172.17.0.24

Follow the same procedure to identify the information about the rest of the certificates.

Checking Logs: When you set up the cluster is good to check the logs in order to see if the certs had any problem.

The Hard Way: journalctl -u etcd.service -l
Kubeadm: kubectl logs etcd-master

Check logs with kubectl down: If you need to check logs when core components like api-server or etcd-server are down the kubectl command line won't work:
docker ps -a
docker logs <container_ID>

Certificates API: CertificateSigningRequest

CA server: The CA server is just a pair of key and certificate files. Whoever have access to these certificates can sign CSR, create as many users as they want and put as many privileges as they want.

You have to place the root certificates in a fully secure server. That server becomes your CA server. The certificate key file is safely stored in that server and ONLY in that server.

If you want to sign certificates you can only doing it by logging into that server.

Kubeadm stores the CA pair of files in the Master Node.

Built in Certificates API: So far we've been signing certificates manually. But when the number of users increase and the team grows you need a better automated way to manage the CSR and as well rotate the certificates when they are expired.

How it works: **CertificateSigningRequest**
K8s have a built in Certificates API, now you are sending the CSR directly to K8s through an API call.

Until now a new user sends the CSR to the admin and the admin logs in into the CA server, signs the certificate, and sends the new_user.crt to the new user.

This time when the admin receives the CSR instead of logging into the Master Node (CA server), he creates a k8s object called **CertificateSigningRequest**

Once the object is created can be seen by admins of the cluster.
The request can be reviewed, approved, denied or deleted by using kubectl commands.
This certificate can be extracted and shared with the user.

How to do it:

1. A user generates a key: openssl genrsa -out peter.key 2084
2. Generates a CSR: openssl req -new -key peter.key -subj"/CN=peter" -out peter.csr
3. Then sends the csr to the admin.
4. The admin takes the csr and creates a **CertificateSigningRequest** object.
5. The CertificateSigningRequest object is creates like any other k8s object using a manifest file with the usual fields:
    1. **apiVersion**: certificates.k8s.io/v1beta
    2. **kind:** CertificateSigningRequest
    3. **metadata**: peter
    4. **spec:**
        1. **groups:** system-authenticated
        2. **usages:** digital signature, key encipherment, server auth
        3. **requests:** the CSR certificate, but encoded with base64

jane-csr.yaml

```
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: jane
spec:
  groups:
  - system:authenticated
  usages:
  - digital signature
  - key encipherment
  - server auth
  request:
      LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRV
      FVRVNULS0tLS0KTUlJQ1dEQ0NBVUFDQVFB
      d0V6RVJNQThHQTFVRUF3d0libVYzTFhWel
      pYSXdnZ0VpTUEwR0NTcUdTSWIzRFFFQgpB
      UVVBQTRJQkR3QXdnZ0VLQW9JQkFRRE8wV0
      pXK0RYc0FKU0lyanBObzV2UklCcGxuemcr
      NnhjOStVVndrS2kwCkxmQzI3dCsxZUVuT0
      41TXVxOTlOZXZtTUVPbnJ
```

How to code / decode the certificate:
cat jane.csr | base 64
echo "the_cert_here" | base64 --decode

How to see it: Once the object is created all admins can see it by running:
kubectl get csr

How to approve it:
kubectl certificate approve peter

K8s signs the certificate using the CA key pairs and generates a CRT for the user.

This CRT certificate can be extracted and shared with the user

View the CRT in yaml:
kubnectl get csr peter -o yaml

You will see the key paste the encoded text and decode it with:
echo my_encoded_cert.txt | base64 --decode

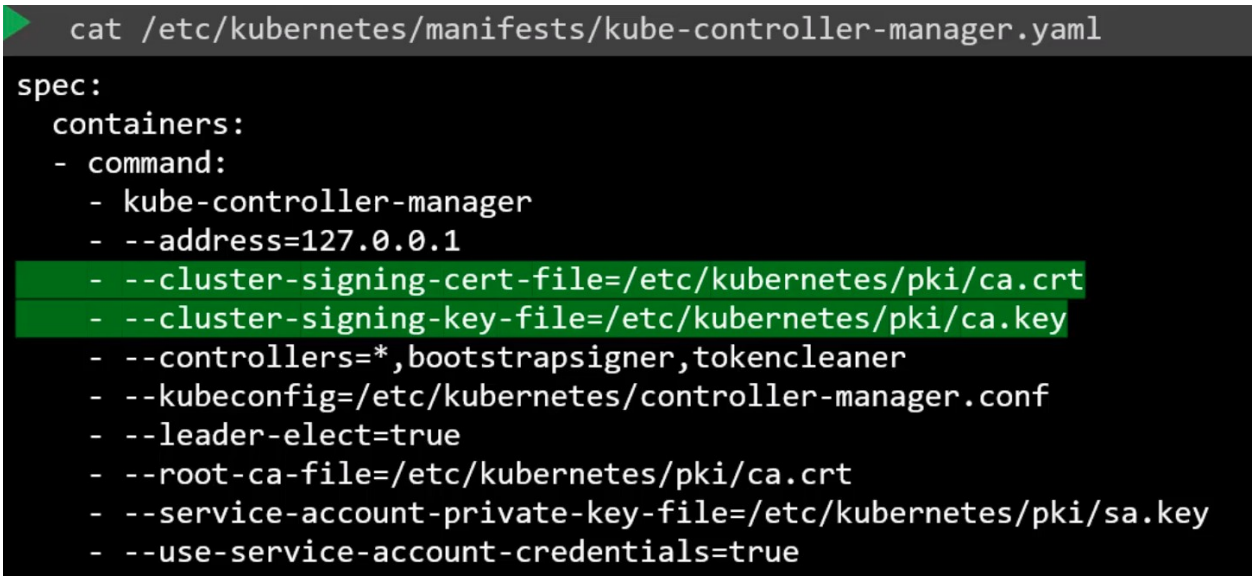
Reject a csr:
kubectl certificate deny peter
kubectl delete csr peter

Which part of the cluster manages this:
The Controller Manager is the responsible for the certificate related operations.

It has controllers on it called CSR-APPROVING, CSR-SIGNING, etc, that are responsible for that specific task.

We know that to sign certificates we need the CA server's root certificate (ca.crt) and private key (ca.key)

In the controller manager configuration, /etc/kubernetes/manifests/kube-controller-manager.yaml we have to specify where both keys are stored, see image below:



KubeConfig:

You already know that you can query the K8s REST API using curl to get a list of PODs for example:

```
curl https://my-kube-playground:6443/api/v1/pods \
 --key admin.key
 --cert admin.crt
 --cacert ca.crt
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
  },
  "items": []
}
```

You can do it too using kubectl like in the image below:

```
kubectl get pods
     --server my-kube-playground:6443
     --client-key admin.key
     --client-certificate admin.crt
     --certificate-authority ca.crt
```

Where you are passing:
Destination server: Like in the API call. This is the Kube-Apiserver
Path to the keys: ca.crt (root certificate) , my private key and my crt.

To avoid to having to type all the time in the kubectl where the keys are and the API server, we use **KubeConfig** file and we write there all that information.

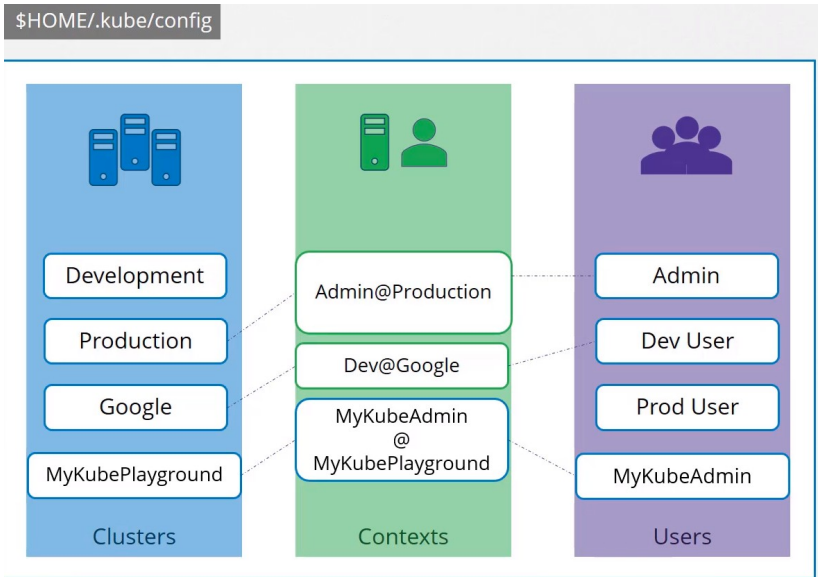
Where is this file: ~/.kube/config
By default kubectl looks into this directory.

The file is already there, that's why you don't have to specify any path to the file explicitly in the kubectl command.

KubeConfig file format:

It have 3 sections:

- **Clusters**: The clusters you have access to. Example: Dev, Test, Prod, AWS, GCP, etc.
- **Users**: User accounts with which you have access to these environments, example: Admin, dev users, prod users, etc.
- **Context:** Marry this together, define which user account will be used to access which cluster. Example: You could create a context named admin@Production that will use the admin account to acess the Production cluster. Or Dev@Google → developers accessing the Google cluster.



You are not creating any user or configuring any kind of user access or authorization in the cluster.

You are using existing users and existing privileges and defining what user you are going to use to access what cluster.

```
apiVersion: v1
kind: Config

clusters:
- name: my-kube-playground
  cluster:
    certificate-authority: ca.crt
    server: https://my-kube-playground:6443


contexts:
- name: my-kube-admin@my-kube-playground
  context:
    cluster: my-kube-playground
    user: my-kube-admin


users:
- name: my-kube-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

```
apiVersion: v1
kind: Config
current-context: dev-user@google
clusters:
- name: my-kube-playground  (values hidden…)
- name: development
- name: production
- name: google

contexts:
- name: my-kube-admin@my-kube-playground
- name: dev-user@google
- name: prod-user@production

users:
- name: my-kube-admin
- name: admin
- name: dev-user
- name: prod-user
```

controlplane $ **cat my-kube-config**
apiVersion: v1
kind: Config

clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: KUBE_ADDRESS

- name: development
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: KUBE_ADDRESS

- name: kubernetes-on-aws
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: KUBE_ADDRESS

- name: test-cluster-1
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    server: KUBE_ADDRESS

contexts:
- name: test-user@development
  context:
    cluster: development
    user: test-user

- name: aws-user@kubernetes-on-aws
  context:
    cluster: kubernetes-on-aws
    user: aws-user

- name: test-user@production

```
  context:
    cluster: production
    user: test-user

- name: research
  context:
    cluster: test-cluster-1
    user: dev-user

users:
- name: test-user
  user:
    client-certificate: /etc/kubernetes/pki/users/test-user/test-user.crt
    client-key: /etc/kubernetes/pki/users/test-user/test-user.key
- name: dev-user
  user:
    client-certificate: /etc/kubernetes/pki/users/dev-user/developer-user.crt
    client-key: /etc/kubernetes/pki/users/dev-user/dev-user.key
- name: aws-user
  user:
    client-certificate: /etc/kubernetes/pki/users/aws-user/aws-user.crt
    client-key: /etc/kubernetes/pki/users/aws-user/aws-user.key

current-context: test-user@development
```

You don't have to create any object, like kubectl create -f something.
You have to leave the file as is and is read by the kubectl command and the required values are used.

Current-context: In a config file that have several contexts the current-context specifies which one will be the default context to use.

Commands:

View the config file: kubectl config view

View a custom kubeConfig file: kubectl config view --kubeconfig=my-custom-config

Change the current-context in the default config file:
kubectl config use-context prod-user@production → **or the name of the context**

Change the current-context in a custom config file:
kubectl config --kubeconfig=/root/my-kube-config use-context research

See help:
kubectl config -h

Namespaces: What if a Cluster have different namespaces, can you configure a Context to a particular namespace? Yes

You can add the field “namespace” into the cluster section. Example: namespace: finance

```
contexts:
- name: admin@production
  context:
    cluster: production
    user: admin
    namespace: finance
```

Certificates: You can specify the full path to the certificates in the config file or you can paste there the certificate encoded in base64.

certificage-authority: /etc/kubernetes/pki/ca.crt

certificate-authority-data: <here the cert encoded in base64>

To decode it: echo “<here the cert encoded in base64>” | base64 --decode

---

API Groups:

All resources in the K8s are grouped into different API groups.

We can get and modify information from the cluster using API calls.

We query the API-Server with http requests to get/modify information in the cluster.

How to do an API Call: API call to get the pods. We have to pass always the key, cert, and root cert:

```
curl https://my-kube-playground:6443/api/v1/pods \
  --key admin.key
  --cert admin.crt
  --cacert ca.crt
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
  },
  "items": []
}
```

Kubectl Proxy: Is an http proxy service created by kubectl to access the kube-apiserver.

We can skip the authentication part (root cert, private key and certificate) by using Kubectl Proxy.

We have to start the client. Then the proxy will use the credentials inside our .kube/config file to access the cluster and won't be necessary to add them into the API calls.

How it works:

Start the client: *kubectl proxy*

This will launch a proxy service locally on port 8001

Don't get confused with kube-proxy and kube**ctl** proxy

Kube proxy is used to enable connectivity between pods and services across different nodes in the cluster.

API Calls: We will get the information in JSON format.

Get the version:
curl https://kube-master:6443/**version**

Get a list of the pods:
curl https://kube-master:6443/**api/**v1/pods

API Groups:
The K8s API is gruped into multiple subgrups based on their purposes, such as /**version**, /**api**,/metrics, /healthz, /logs, etc.

Version API group: You can see the version of the cluster

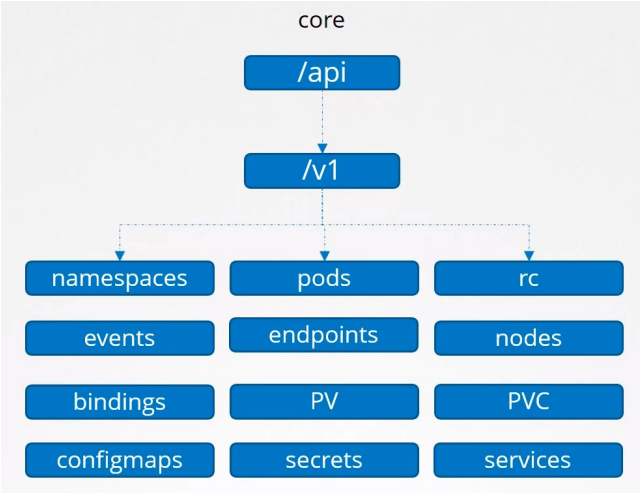
Metrics and healthz: Used to monitor the health on the cluster.

Logs: Used for integrating with 3rd party log application.

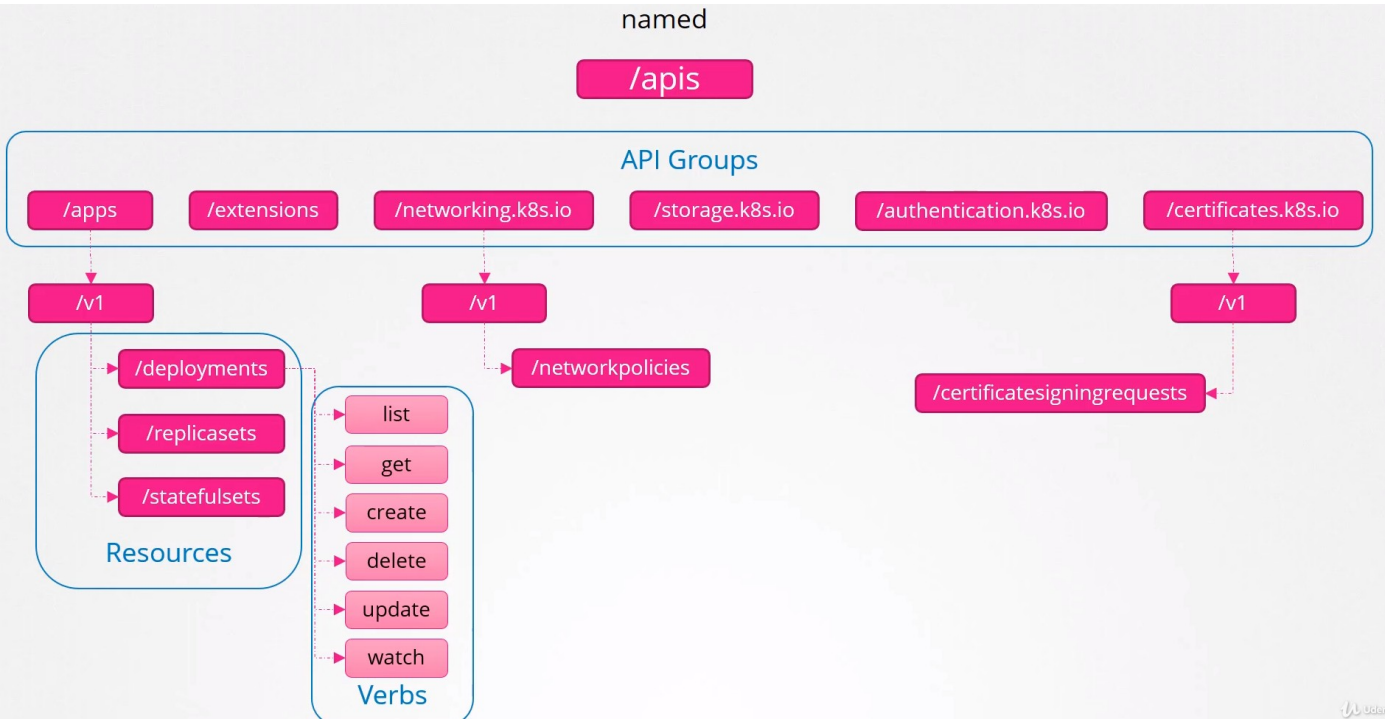
Api and Apis: Responsible for the cluster functionality.

Core Api Group: Where all core functionalities exists, like pods, namespaces, secrets, etc, etc.

Some of them are in the image below:



Named Apis groups: Here the groups are more organized, see some of them in the image below:



How to see the available API groups:

The K8s API reference page can tell you what API is for each object. Select an object and the first section on the documentation page shows it's group details, for instance POD shows v1

Or you can make an API call to the cluster:

curl http://localhost:6443 -k

curl http://localhost:6443 -k |grep "name" → to get all the name resource groups.

Real example: Will show you the hole list of options you have
curl https://controlplane:6443 --key dev-user.key --cert dev-user.crt --cacert /etc/kubernetes/pki/ca.crt -k

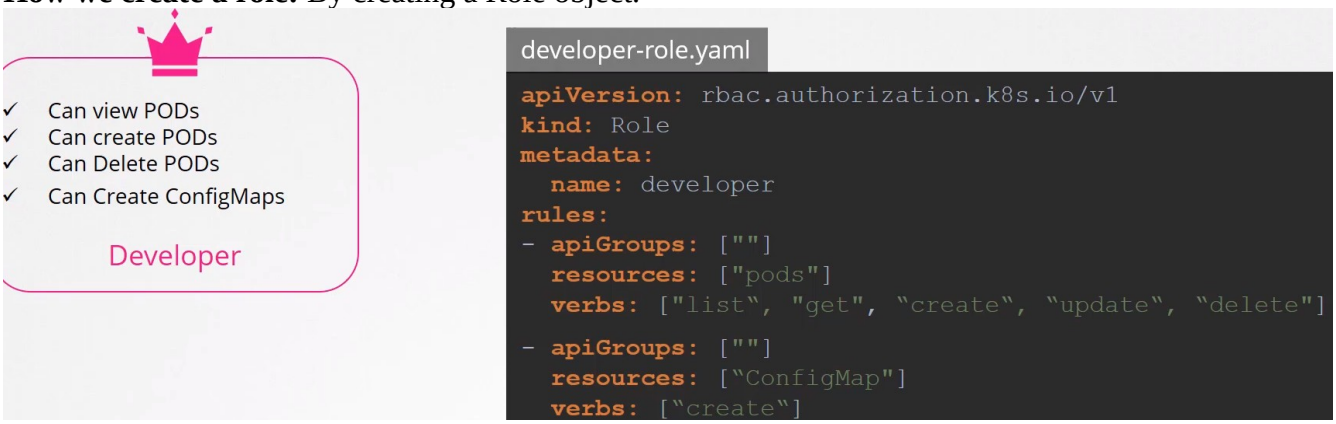
the dev-user keys are in /etc/kubernetes/pki/users/

Another real example: Will show you all the pods the cluster have
curl https://controlplane:6443/api/v1/pods --key dev-user.key --cert dev-user.crt --cacert /etc/kubernetes/pki/ca.crt

---

RBAC:

How we create a role: By creating a Role object.



kind: Role

We've named it as *developer*.

Rules: Each rule have 3 sections: The same sections we saw in the previous API Groups chapter, see Named Apis Groups

- **apiGroups:** Like /apps or /extensions, etc. We can leave the core group as black ""
- **resources:** Like /pods or /replicasets , etc
- **verbs:** Like get, list,create, update, etc

We can add as many rules as we need, in this example we have added another rule to allow developer to create ConfigMaps.

Role Binding: The next step is to link (bind) that role created to the user that we want, dev-user in this example. To do so we are creating role binding objects.

devuser-developer-binding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: devuser-developer-binding
subjects:
- kind: User
  name: dev-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: developer
  apiGroup: rbac.authorization.k8s.io
```

Kind: RoleBinding

metadata:
name: devuser-developer-binding
namespace: → add the namespaces you need. Remove this field if you want to use only the default namespace

Subjects: Is where we specify the user details.

RoleRef: Is where we provide details of the role we've created.

Commands:

To view the Roles: kubectl get roles

List rolebindings: kubectl get rolebindings

View more details about a role: kubectl describe role developer
The same with role bindings: kubectl describe rolebindings devuser-developer-binding

How to check if a particular user have access to a particular resource in the cluster:

kubectl auth **can-i** create deployments
kubectl auth can-i delete nodes

Check another user, only if you have admin rights:

kubectl auth can-i create deployments **--as dev-user**
kubectl auth can-i create pods --as dev-user

Specifying namespace:
kubectl auth can-i create pods --as dev-user **--namespace test**

Resource Names: You can restrict access to the pods in the given namespace buy adding resourceNames into the Rules section:

Say that the namespace have 5 pods and I want only to provide access to the deveoper to two of these pods, like we can see in the image below:

Cluster Roles & Cluster Role Bindings:

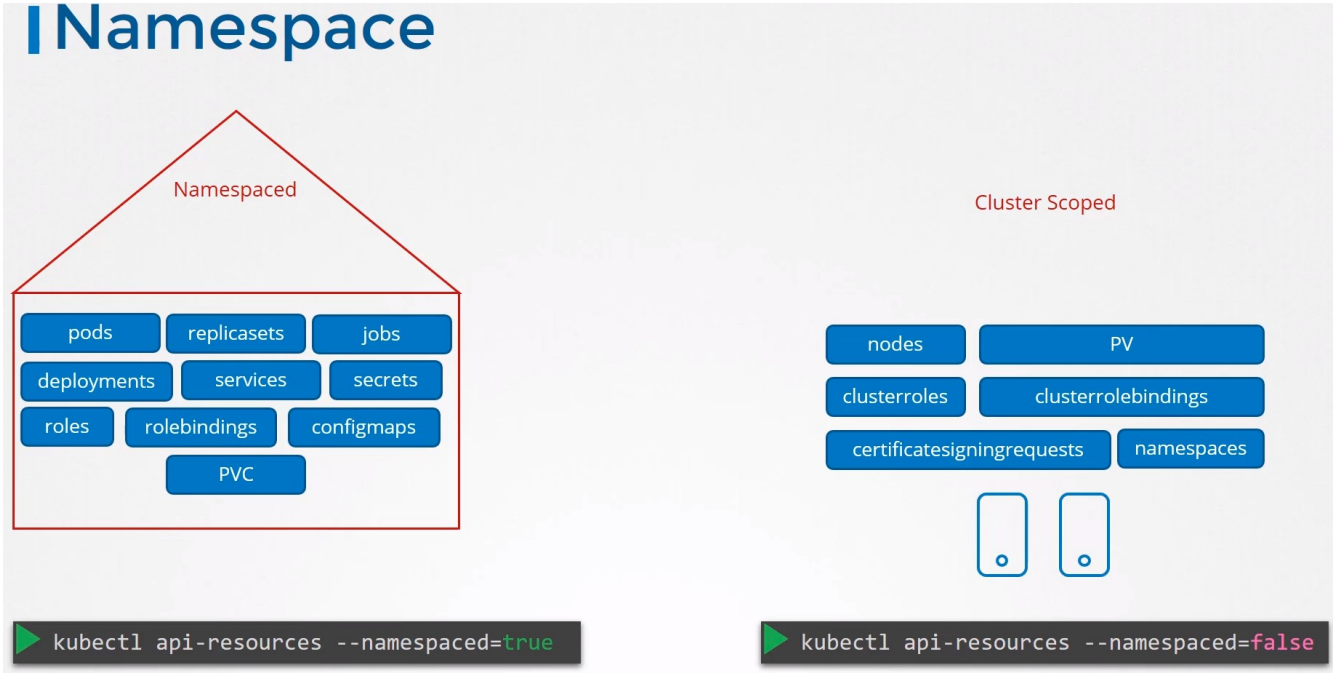
Namespaces are helping in grouping pods, deployments, services, etc.

But what about other resources like Nodes, you can NOT group it into a Namespace.
Those are Cluster wide or Cluster scope resources that can not be associated to any particular namespace.

Resources are categorized as:

- Namespaced
- Cluster Scoped

In the image below we can see resources that we can create as namespace.
We can assign them into a particular namespace and if we don't specify nothing will be assigned to the default namespace:



The Cluster Scoped resources are those where you DON'T specify a namespace.

The image contains only a few resources in both sides, namespaced and cluster scoped.

To see the whole list run the commands you can see in the image.

How to authorize users to cluster wide resources: like nodes, persistent volumes, etc.

By using cluster role and cluster role bindings

Cluster roles are like role except they are for cluster wide resources.

Examples:

– A cluster Admin role can be created to provide a cluster administrator permissions to view, create and delete nodes.

- A storage Admin role can be created to authorize a storage admin to create Pvs (persistent volumes)

Create a Cluster Role:

Is the same than creating a Role, only changing:

Kind: ClusterRole

rules.resources: Specify the nodes

cluster-admin-role.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-administrator
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["list", "get", "create", "delete"]
```

Create a Cluster Role binding:
The same than a cluster role, we have to link the user to the role created before.

Kind: ClusterRoleBinding

cluster-admin-role-binding.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-admin-role-binding
subjects:
- kind: User
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-administrator
  apiGroup: rbac.authorization.k8s.io
```

You can combine Roles and Cluster Roles:

You can create a Cluster Role in which specify that the user can have access to PODs for instance.

Then the user will have access to the Nodes (Cluster Scoped) and to the PODs (Namespaced) or to the resources you specify.

Have in mind that the user will have access to all the PODs in all namespaces of the k8s cluster.

Image Security:

Basics of image names: We are deploying pods using images.

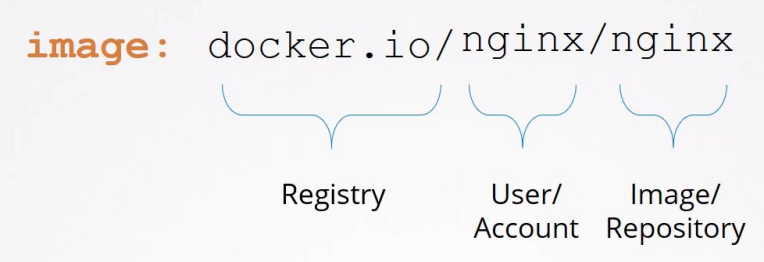
spec.containers.image: Is where we specify which image to install in the pod.

We've used the simplest way to specify the image:

```
spec:
  containers:
  - name: nginx
    image: nginx
```

But we are really specifying to pull the image from the docker.io repository, using the nginx user and pulling the nginx image

This is the full path that an image have:



Registry:
K8s by default takes the images from docker's default registry, Dockerhub, the DNS is docker.io, so you don't need to specify it if you pull the image from there.

There are many other popular registries like the one from Google:
gcr.io/kubernetes-e2e-test-images/dnsutils

Private Registry: When you have applications built in house that shouldn't be made available to the public hosting an internal private registry may be a good solution.

Many cloud providers such as Azure, AWS or GCP provides a private registry for your cloud account by default.

Your private registry: You can have your own private registry. Can be accessed using a set of credentials.

From Docker perspective to run a container using a private image stored in your own registry you first have to log in into it using the docker login command:

```
docker login my-own-private-registry.io
```

Once succesfully loged run the app using the image from the private registry:

docker run my-own-private-registry.io/apps/internal-app

Account User:

When you pull nginx for instance is actually nginx/nginx which is the user or account name.
If you create an account in docker by your selve the user account must be specified like:
my_user/nginx

If you don't provide an account name it assumes that to be the same as the repository name, nginx in this example.

In the pod definition file:

In spec.containers.image we specify the full path to the image in our private registry.

But how we do specify the credentials?

We create a Secret object:

```
kubectl create secret docker-registry regcred \
    --docker-server= private-registry.io    \
    --docker-username= registry-user  \
    --docker-password= registry-password  \
    --docker-email= registry-user@org.com
```

Then specify it into the pod definition yaml:

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: private-registry.io/apps/internal-app
  imagePullSecrets:
  - name: regcred
```

Run a private registry in docker:

Log in into the private registry:
```
docker login private-registry.io
```

Run a private image once logged in:
```
docker run private-registry.io/apps/internal-app
```

Security Context:

What is: Defines privilege and access control settings for a POD or Container.

You can configure it at a POD level or/and a Container level. If the same privileges are in the POD and Container, the container overwrites the POD's ones.

How to configure it: In the POD definition file.

In a POD:
```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
```

In a Container:
```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Capabilities: Only to the Container.

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Commands:

See the user running a POD: kubectl exec <pod_name> -- whoami

Open a shell into a POD: kubectl exec -it <pod_name> -- sh
kubectl exec -it <pod_name> -- bash

Run a command with NO permissions:

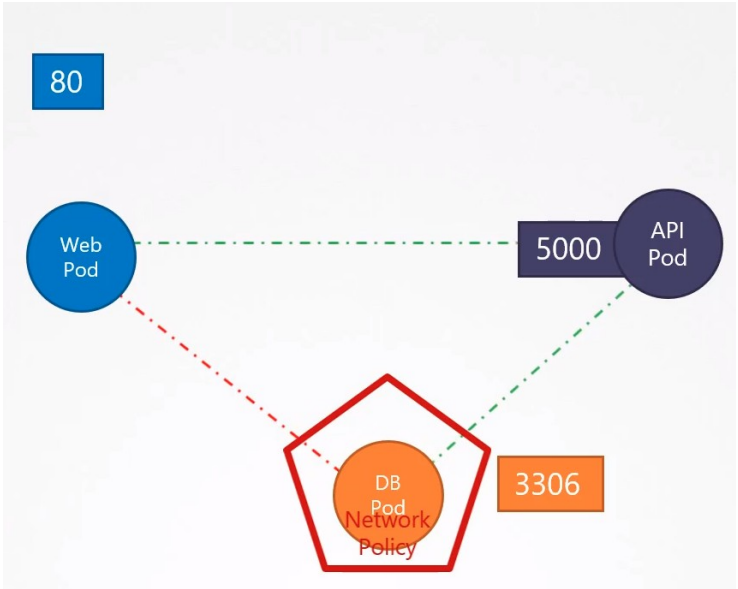
controlplane $ **kubectl exec -it ubuntu-sleeper -- date -s '19 APR 2012 11:14:00'**
date: cannot set date: Operation not permitted
Thu Apr 19 11:14:00 UTC 2012
command terminated with exit code 1

Network Policy:

Docs

All Allowed rule: K8s is configured to allows traffic by default from any POD to any other POD or services within the cluster.

What if you want to deny traffic to a POD:
Let's say that you are running a web server, an API server and a db, see image below:



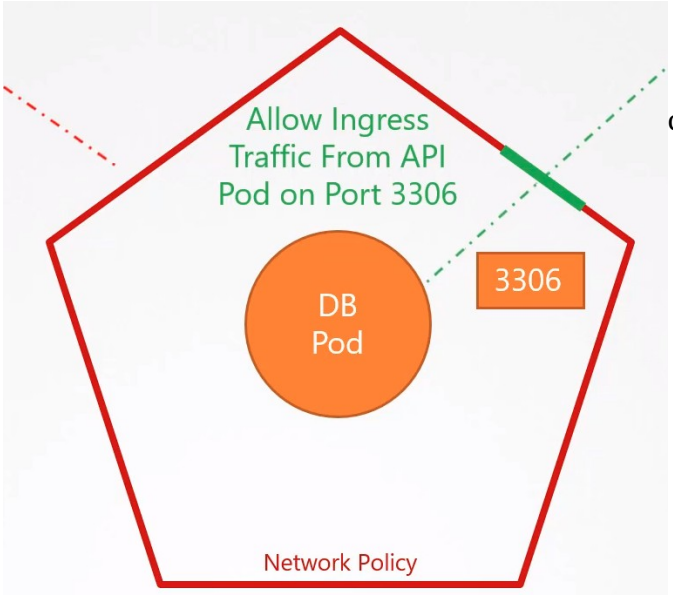
You want to communicate the web pod with the api pod. But by security reasons you want to avoid communications between the web pod and the db pod.

This is where network policy comes in.

A network policy is just another K8s object.

You link the Network Policy to one or more pods. You can define rules within the network policy.

Following the above example we want to allow only network traffic from the Ingres port 3306, the one coming from the API pod:



Have in mind that **Ingres** and **Egress** can be configured

The Network Policy will block all the traffic to the POD and will only allow traffic that matches that particular rule.

Network solutions supporting it:
Calico
Cilium
Kube-router
Romana
Weave Net

NOT Supporting it:
Flannel

Definition:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    ports:
    - protocol: TCP
      port: 5978
```

Commands:

See network policy: kubectl get netpol
You will see the name of the policy and the pod-selector name (the label)

```
controlplane $ kubectl get netpol
NAME           POD-SELECTOR   AGE
payroll-policy   name=payroll        29m
```

See wich PODs have the label payroll: kubectl get pods -l name=payroll

```
controlplane $ kubectl get pods -l name=payroll
NAME     READY   STATUS    RESTARTS   AGE
payroll   1/1          Running    0                26m
```

We can see that the POD that have the label payroll have the same name.

Show labels: kubectl get pods --show-labels

controlplane $ **kubectl get pods --show-labels**

NAME	READY	STATUS	RESTARTS	AGE	LABELS
external	1/1	Running	0	47m	name=external
internal	1/1	Running	0	47m	name=internal
mysql	1/1	Running	0	47m	name=mysql
payroll	1/1	Running	0	47m	name=payroll

STORAGE:

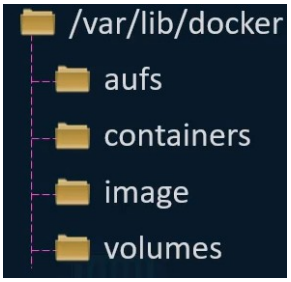
Storage in Docker:

When talking about Storage in Dockers we have two concepts:

- **Storage Drivers Plugins**
- **Volume Drivers Plugins**

How Docker stores data in the local file system:
When you install Docker in a system it creates this folder structure at: /var/lib/docker

Docker stores all its data in that folder and subfolders by default. Data like files related to images and containers running on the local host.



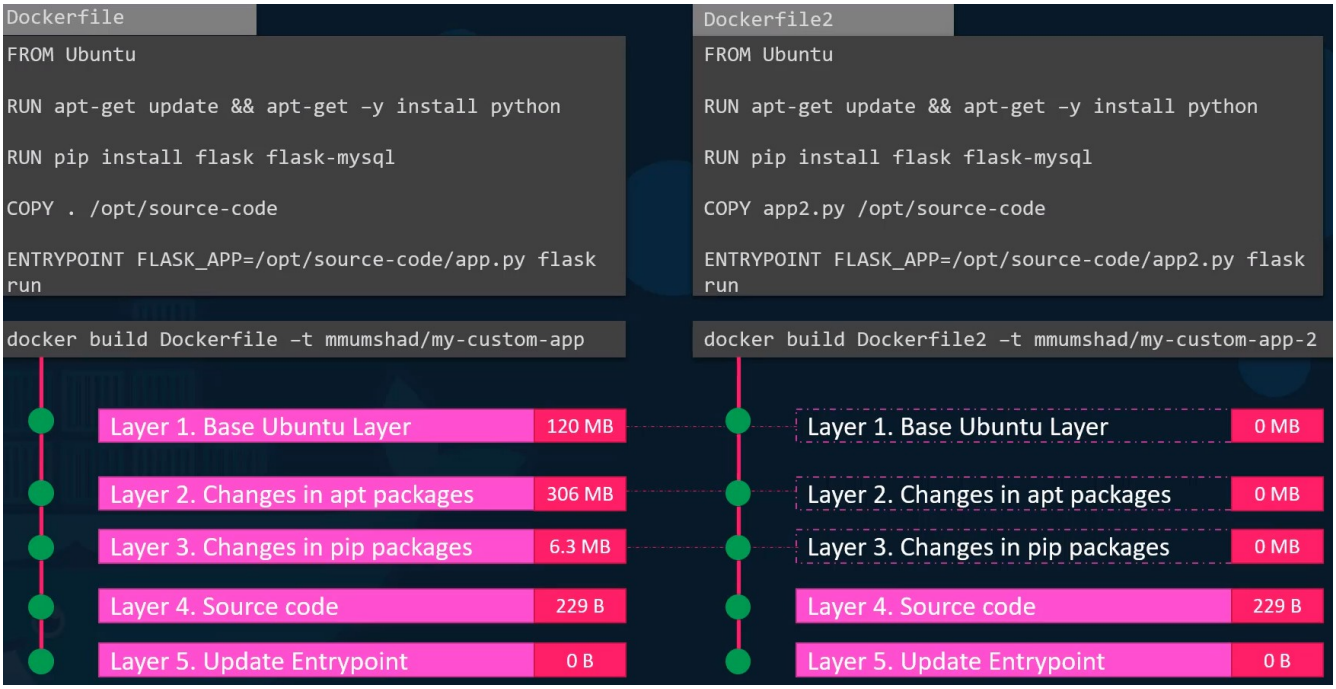
/var/lib/docker/<storage_driver> → Is used to store the layers. Layers have names like these:

ls **/var/lib/docker/overlay2** → **overlay2 = storage driver**
16802227a96c24dcbeab5b37821e2b67a9f921749cd9a2e386d5a6d5bc6fc6d3
377d73dbb466e0bc7c9ee23166771b35ebdbe02ef17753d79fd3571d4ce659d7
3f02d96212b03e3383160d31d7c6aeca750d2d8a1879965b89fe8146594c453d
ec1ec45792908e90484f7e629330666e7eee599f08729c93890a7205a6ba35f5

These are the cryptographic IDs of the layers.

Docker Layered Architecture: When Docker builds images it builds it in a LAYER architecture.

Each line of instruction on the **Dockerfile** creates a new layer in the Docker image with just the changes from the previous layer.



In the picture above we can see the Dockerfile (left) to create a Docker Image containing ubuntu and the different layers that is creating for each line of instructions.

1. The first layer is a base ubuntu OS which weighs 120MB.
2. The second instruction creates the second layer that installs all the apt packages.
3. Then the 3rd layer install the python packages and so on.

Since each layer only stores the changes from the previous layer is reflected in the size as well.

Understanding the benefits of layers: Now we will create a second application (right).

This application, Dockerfile2, have a different Dockerfile but is very similar to the first one.

It uses the same base image as Ubuntu, uses the same flask and python dependencies, but uses a different source code to create a different application and a different entry point.

When you run the docker build command to build a new image for this application, since the first 3 layers of both the applications (dockerfile and dockerfile2) are the same docker is NOT going to build the first 3 layers.

Instead it reuses the same 3 layers it built from the first application from the cache and only creates the last 2 layers with the new sources and the new entry point.

This way Docker build images faster and efficiently saves disk space.

This is also applicable if you update your application code. When you only modify the code, like app.py in this example, Docker simply reuses all the previous layers from cache and quickly rebuilds the application image by updating the latest source code.

This saves a lot of time during rebuilds and updates.

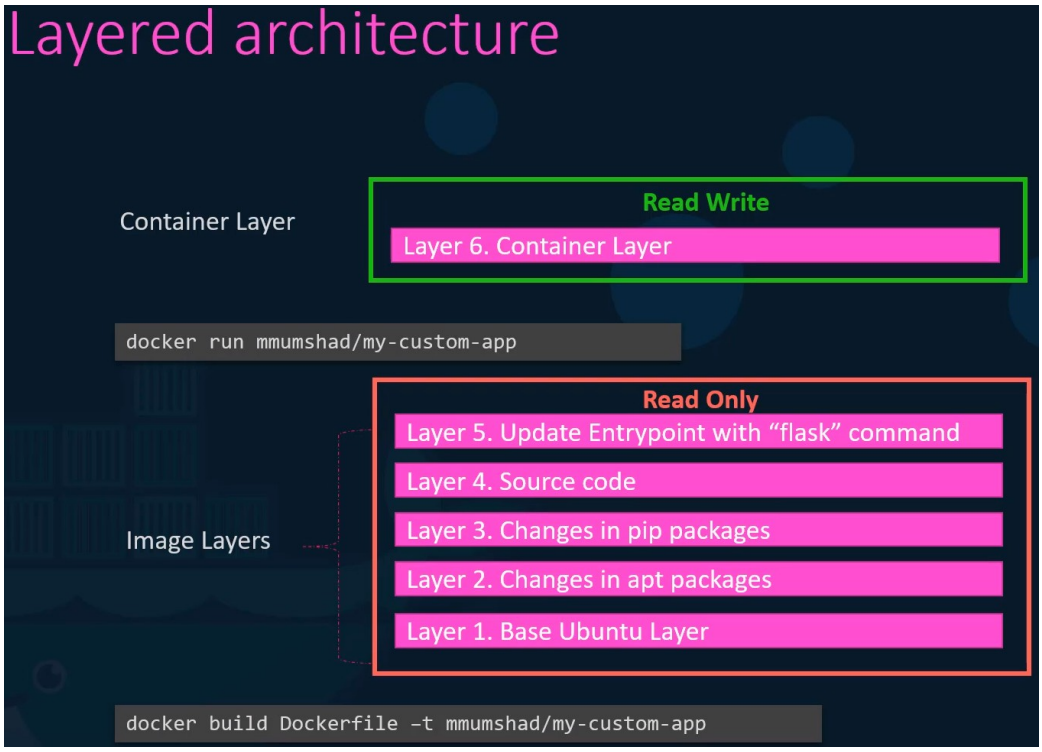
Image Layers: All the layers are created when you run the docker build command to form the final docker image.

Once the build is complete you can not modify the contents of these layers. They are READ-ONLY and you can only modify them by initiating a new build.

Container Layer: When you run a container based off on these image using the docker run command, docker creates a container based on these layers and creates a new thin WRITABLE-LAYER on top of the image layer.

The writable layer is used to store data created by the container such as logs files used by the applications, any temporary files generated by the container or just any file modified by the user in that container.

The life of this layer is only as the container is alive. When the container is destroyed, this layer and the changes stored on it are also destroyed.



COPY-ON-WRITE:

Docs

We have now a running container from the image created previously.

We have a read/write layer that allows me to generate and store files there.

I want to modify the source code in that container (layer 4 in last example) to test a change for example.

I can not modify the source code in the image because is read only.

But I can copy the source code into the read/write layer and then from there I'm able to do the proper modifications of the code.

All future modifications will be done on this copy of the source code in the rewrite layer. This is called **Copy-on-Write** mechanism.

Have in mind that if we delete the POD the data will be deleted too.

Persistent Volumes: To preserve the data stored in the read/write container layer.

We may have a container with a db and we want to persist the data after deleting that container.

To do so we create **persistent volumes** and we bind them into the container that we want.

How to do it:

1st – Create a volume: *docker volume create **data_volume***
It creates the folder data_volume under: /var/lib/docker/volumes/data_volume

2nd – Mount the volume: inside the docker container's read/write layer (-v)
*docker run -v **data_volume**:/var/lib/mysql mysql*

We create the mysql container from the mysql image and we mount it into the newly created data_volume

We specify /var/lib/mysql → location inside the container under mysql stores data by default.

This will create a container called mysql and will store the content of /var/lib/mysql into the volume created before, data_volume

Now if you delete the container the data will be still active.

Volume Mounting: What if you didn't created a volume

If I create a container specifying a volume that doesn't exists, Docker will create that volume for you:

docker run -v **data_volume2**:/var/lib/mysql mysql

data_volume2 doesn't exists, but docker will create it and will bind it to /var/lib/mysql

This is called volume mounting.

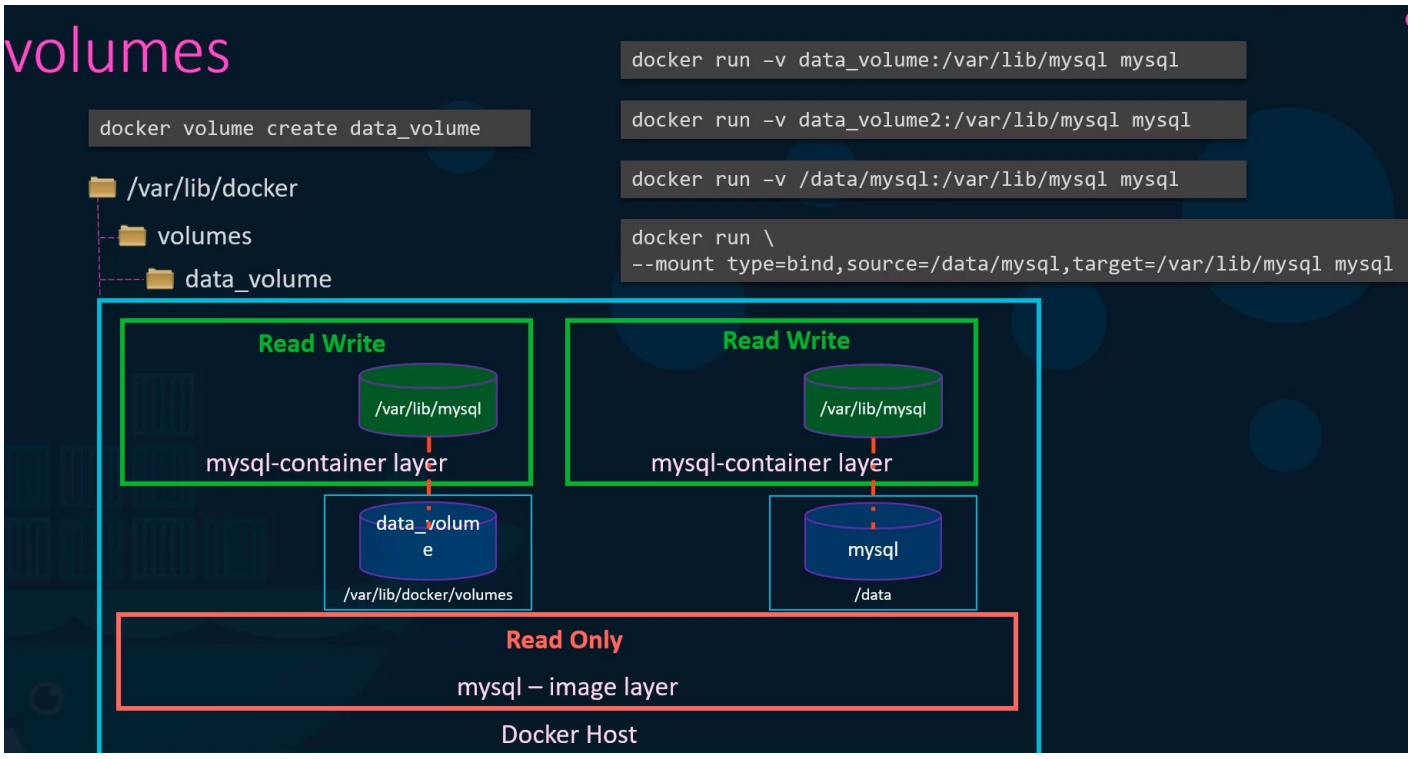
Bind mounting: If you have a external drive in which you want to store the volumes you have to specify the full path into this other drive.

docker run -v **/external_drive/data:**/var/lib/mysql mysql

Two types of volume mounting:

1. Volume mounting: Mounts a volume on the volume directory.

2. Bind Mount: Mounts a directory from any location on the docker host.



New command to create it:
docker run -v is the old way.

The new one more specific:

docker run --mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql

Storage Drivers:
Are the responsible of maintaining the architecture layer, creating writable layers, moving files, coping writing, etc.

Some of the common storage drivers:
- AUFS
- ZFS
- BTRFS
- Device Mapper
- Overlay
- Overlay2

The selection of the Storage Drivers depends on the underlying OS in use.

For instance, Ubuntu uses AUFS (which is not available in CentOS). Newer informations says that could be using Overlay2.

For centos or fedora Device Mapper would be a good option.

Docker will choose the best Storage Driver based on the host OS.

Different Storage Drivers provide different performance, stability, etc.

Volume Drivers:
If you want to persist storage you must create volumes. Volumes are handled by volume driver plugins.

Volume driver plugins: **Local** (default), helps create a volume on the local host and stores its data in /var/lib/docker_volume directory.

There are more plugins that allow you to create a volume in a 3rd party solutions like:
- Local (default)
- Azure File Storage
- Convoy
- DigitalOcean Block Storage
- Flocker
- gce-docker
- GlusterFS
- NetApp
- RexRay
- Portworx
- Wvware vSphere Storage
- etc

Some of these storage drivers supports different storage providers, for instance RexRay can be used to provision storage on AWS EBS, AWS S3, Google Persistent Desk, Openstack, etc.

Command to attach your persistent volume to AWS EBS:
docker run -it --name mysql --volume-driver rexray/ebs --mount src=ebs-vol,target=/var/lib/mysql mysql

Container Storage Interface (CSI)

In the past K8s used alone Docker as a container runtime engine and all the code to work with Docker was embebed in the K8s source code.

Then other container runtimes came out such as rkt or cri-o it was important to open up and extend suppport to different container runtimes and not be dependent on the K8s source code.

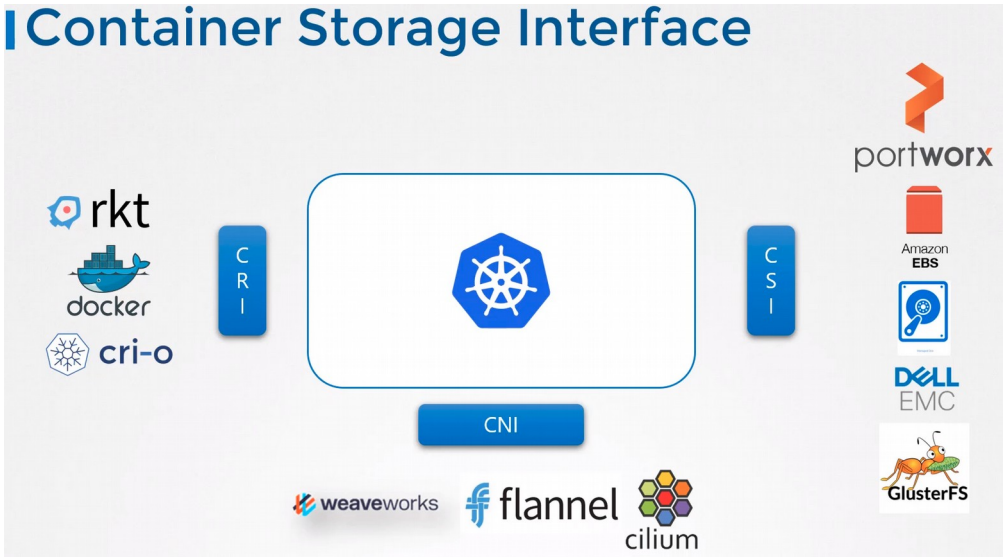
And that's how the Container Runtime Interface (CRI) came to be.

The **CRI** is a STANDARD that defines how an orchestration solution like K8s would communicate with container runtimes like Docker.

So in the future if a new container runtime is developed they can simply follow the CRI Standard and that new container runtime would work with K8s without really having to work with the K8s developers or touching the source code.

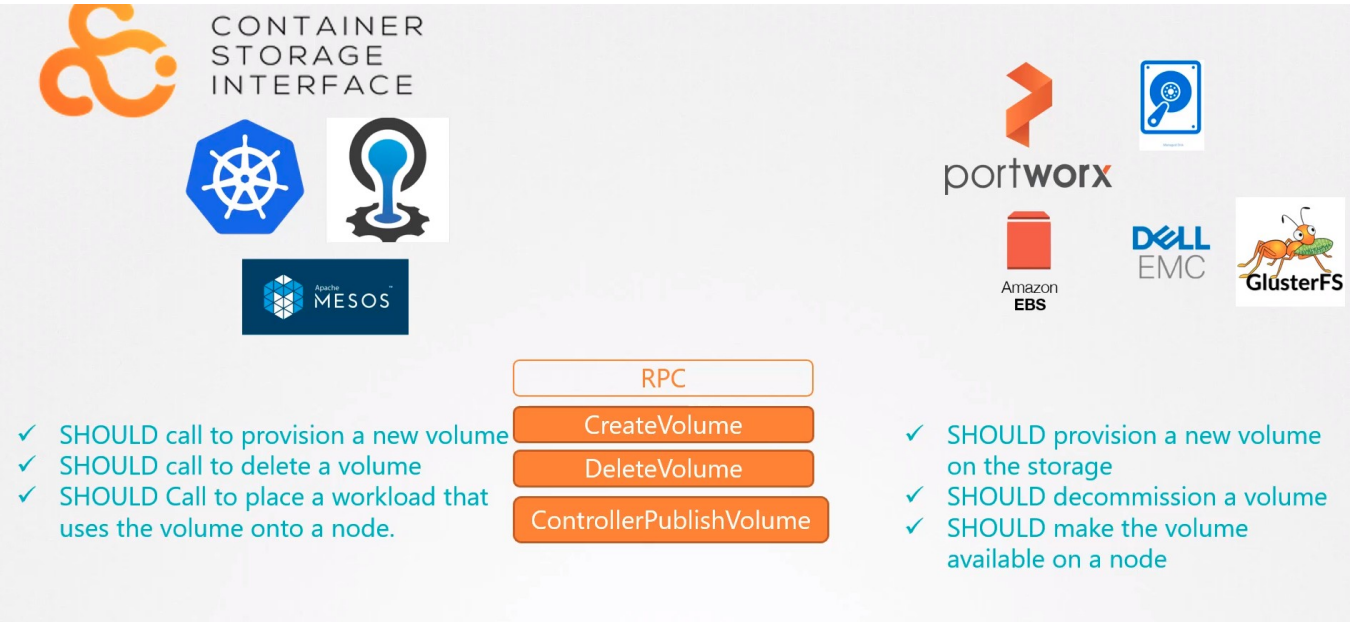
Similarly to extend support for different Networking solutions the Container Networking Interface (**CNI**) Standard was introduced. Now any new Networking vendors could simply develop it plugin based on the CNI standards and make their solution work with K8s.

Container Storage Interface (CSI): Was develop to support multiple storage solutions.
With CSI you can now write your own drivers for your own storage to work with K8s.



CSI it is not a K8s specific standard. It is meant to be an universal standard and if implemented allows any container orchestration tool to work with any storage vendor with a supported plugin.

K8s, Cloud Fondry and Apache Mesos are on board with CSI.



Volumes:

Volumes in Docker:
Docker containers are meant to be **transient in nature** which means that lasts a short period of time.

They are called upon were required to process data and destroyed when is finished. The same happens with the data, the data is destroyed along with the container.

To persist data processed by the containers we attach a volume to the containers when they are created.

The data processed by the container is now placed in this volume thereby retainigh it permanently. Even if the container is deleted the data generated or processed by is retained.

Volumes in K8s:

Like in docker the pods created in K8s are **trancient in nature.**
We can attach a volume to the POD to store and keep the data generated in the POD.

Example:
Let's take a look to a simple implementation of volumes:

We create a single pod with a script that generate a random number between 0 and 100 and send the output to /opt/number.out to the read/write layer inside the container.

Then the pod gets deleted with the random number.

To retain the number generated by the POD we create a volume.

And the volume needs storage. We can configure the storage in different ways.

Create the volume:
In this example we will configure it in a directory on the host (Node1).
We specify a path to the directory /data on the host:

```
hostPath:
    path: /data
    type: directory
```

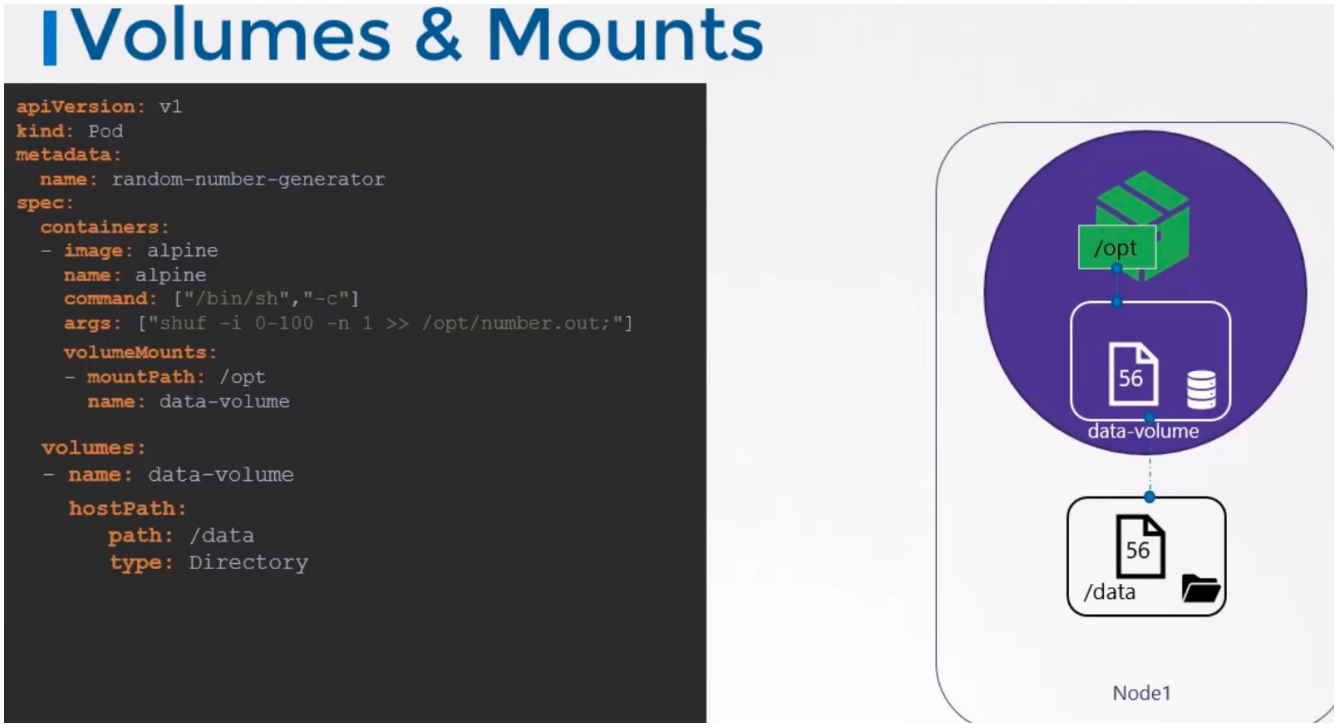
This way all the files generated in the volume will be stored in the directory /data in the node.

Mount the volume: Once the volume is created to access it from the container we have to mount the volume to a directory inside the container.

```
Spec.containers.volumeMounts:
                        - mountPath: /opt
                          name: data-volume
```

The randon number obtained when the script shuf -i … is executed will be written in the /opt directory inside the container and because is mount into the /data directory of the host, the random number (56 in this example) will be preserved in the host.

If the POD is deleted the number (56) still lives in the host.



Volume Storage Options:

We've just used the hostPath option to configure a directory on the host. This works fine for a single node.

However this is not recommended for use in a multinode cluster.

This is because the PODs will use the /data directory on all the nodes and the data in each node won't be the same.

To centralize the data in a single point we configure an external replicated cluster solution.

K8s supports several types of different storage solutions such as:

- NFS
- GlusterFS
- Flocker
- Ceph
- Scaleio
- Fiber channel
- AWS EBS
- GCP
- Azure Disk
- etc, etc.

Example: Configure AWS EBS as the storage option for the volume

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

Type of volumes:

- EmptyDir
- HostPath
- Cloud Volumes
- NFS
- Persistent Volume Claim (PVC)
- And many more

Persistent Volumes (PV):

Docs

Object **created by the sysadmin** within k8s cluster that centralices the volume management.

PVs are resources in the cluster.

Difference between Volumes and Persistent Volumes:

You have to create a volume in each pod. If you have to update the volume you have to modify it in all the pods that contain that volume. So if you want to create a volume into 10 PODs you have to specify it 10 times. The same if you want to update the volume information.

Persistent volumes centralices this by creating an object which the PODs can refeer too.

Definition: PV is a cluster wide pool of storage volumes configured by an administrator to be used by users deploying applications in the cluster. The users can now select storage from this pool using persistent volume claims.

spec.accessModes: Defines how a volume should be mounted on the host. There are 3 modes:

- ReadOnlyMany: All the PODs can read it
- ReadWriteOnce: Only one POD can read/write
- ReadWriteMany: All the PODs can read/write

spec.capacity: Specify the amount of storage to be reserved for that persistent volume.

Volume Type: Here the storage solution, like awsElasticBlockStore:
Remember not to use hostPath in a production environment because is storing it in the particular host in which the POD is and not in a centralized solution.

Persistent Volume Claims (PVC)

PV and PVC are separated objects in the K8s namespace.

An Admin creates a set of PV and an user creates a PVC to use the storage.

Once the PVC are created, K8s binds the PV to claims based on the request and properties set on the volume.

Properties such as capacity, access modes, volume modes, storage class.

You can use labels to particulary use a specific volume using labels and selectors:

```
PV:
        labels:
          name: my-pv

PVC:
        selector:
          matchLabels:
            name: my-pv
```

Every PVC is bound to a single PV.

A PVC with a small storage capacity request can match a PV with a bigger one if all the other criteria matches and there are no better options.

There is a one to one relationship between claims and volumes, so no other claims can utilize the remaining capacity in the volume.

Pending state: If there are no volumes available, the PVC will remain in a pending state until newer volumes are made available to the cluster.

Once new volumes are available the claim will automatically be bound to the newly available volume.

Create a PVC:

pvc-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

```
kubectl create -f pvc-definition.yaml
```

Volume created previously:

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

See the status of the claim:

```
kubectl get persistentvolumeclaim
NAME        STATUS    VOLUME    CAPACITY   ACCESS MODES
myclaim     Pending
```

Delete a PVC:
kubectl delete persistentvolumeclaim myclaim

What happen to the volume when a PVC is deleted:
You can choose what is to happen to the volume:

By default it is set to:

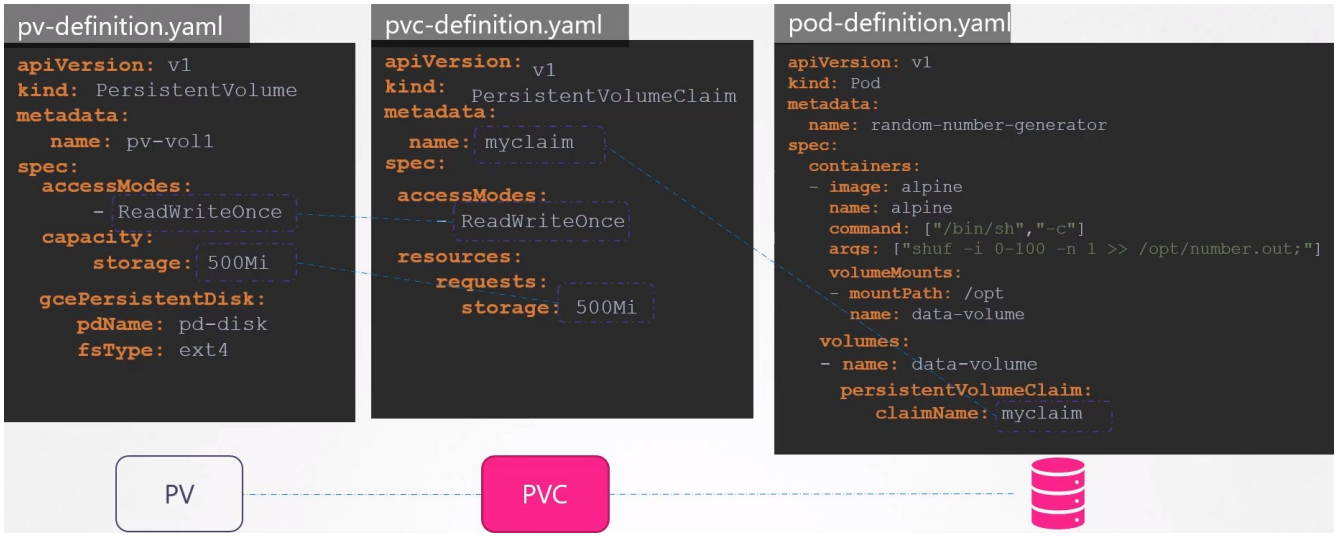
persistentVolumeReclaimPolicy: Retain → the PV will remain until is manually deleted by the Admin. NOT available to reuse by any other Claims

persistentVolumeReclaimPolicy: Delete → Deletes the PV, this is freeing up storage on the end storage device.

persistentVolumeReclaimPolicy: Recycle → The data in the data volume will be removed before making it available to other claims.

How to call the PVC in the PODs, RS and Deployments:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```



Storage Class: Not requested for the CKA exam

Static Provisioning of volumes: You have to manually create the disk in the cloud provider or the storage vendor, then you have to create the pv too in the k8s cluster too.

Dynamic Provisioning of volumes: Storage classess helps to create the disk in the cloud provider automatically and to attach that to parts when a claim is made.

You can define a provisioner such as AWS EBS that can automatically provision storage on the AWS storage and attach that to parts PODs wheb a claim is made.

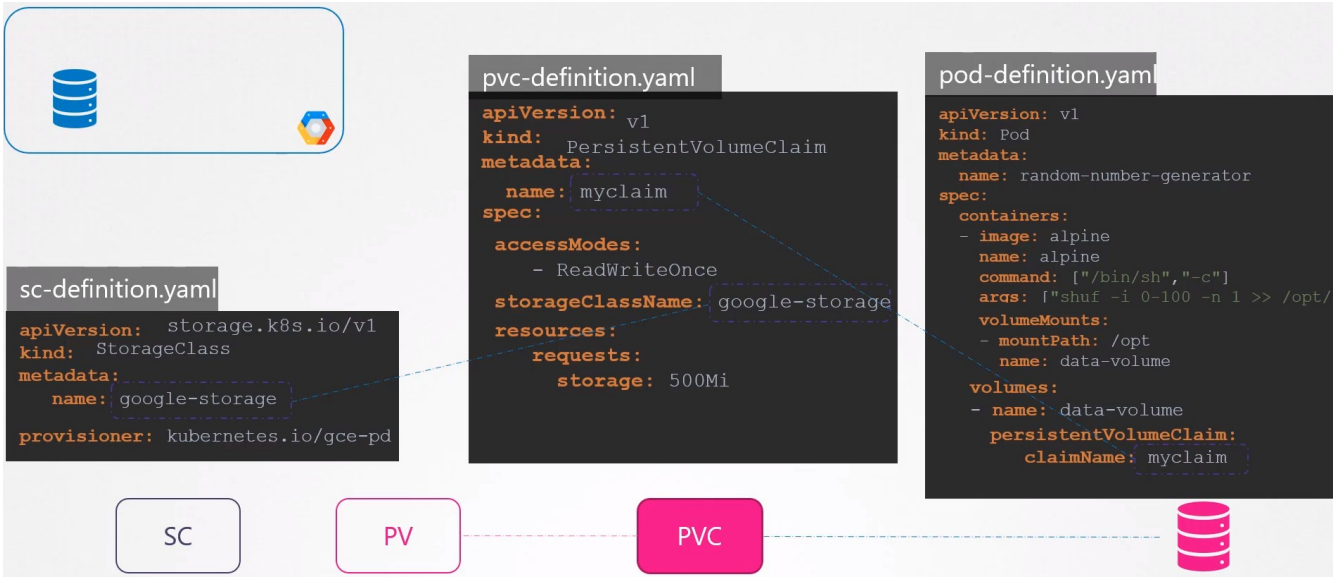
You do that by creating a StorageClass object.

sc-definition.yaml

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

So no longer we need to manualy create a PV because any associated storage is gonna be created automatically when a storage class is created. The PV is gonna be created automatically by the SC

How to call the StorageClass from the PVC:



Networking:

Namespaces: Implementation of the Linux Kernel. Used to isolate resources.

By default each Linux system has only one namespace. All system resources, filesystem,hostname, process ID, network intefaces, etc, belongs to this single namespace.

But you can create additional namespaces and organize resources across them in order to isolate them.

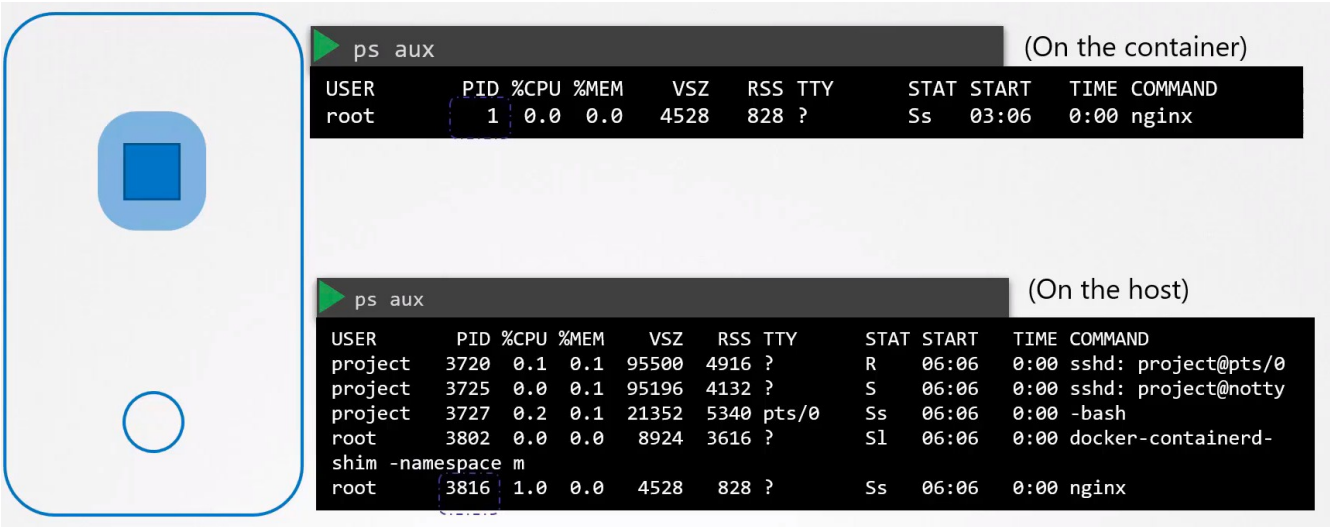
There are several kind of namespaces:

- Mount (mnt)
- Process ID (pid)
- Network (net)
- Inter-process comminications (ipc)
- UTS
- User ID (user)

Single Host example / Process Namespace: We have 1 container running in a host.
The container is isolated form everything using a namespace.

If you are inside the container and check the processes you will see only one, with the PID 1.

But if you check the proceses in the host you can see all the process in the host including the one from the container, with PID **3816** in the image below:



The same process running inside and outside the container with different PID.

This is how Namespaces work.

Network Namespaces: Used in Linux, containers, etc to implement network isolation.

Create a Network NS in a Linux host:

ip netns add red
ip netns add blue

List the Network NS:

ip netns

List interfaces in the host:

ip link

List interfaces in the NS:

ip netns exec red ip link → the ip link command will be executed inside the red NS

or

ip -n red link → the shorter command, does the same (-n = netns)

```
[root@centos8 ~]# ip -n red link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

You only can see the loopback interface and not the rest of the interfaces of the host.

List ARP table:

[root@centos7 ~]# **ip netns exec red arp** → you won't see any entry.

If you run the arp command in the host you will see entries:

```
[root@centos8 ~]# arp
Address                  HWtype  HWaddress           Flags Mask            Iface
_gateway                 ether   0a:7e:ef:58:df:77   C                     eth0
```

Routing table:

```
[root@centos8 ~]# ip netns exec red route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface

[root@centos8 ~]# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    100    0        0 eth0
172.31.16.0     0.0.0.0         255.255.240.0   U     100    0        0 eth0
192.168.122.0   0.0.0.0         255.255.255.0   U     0      0        0 virbr0
```

Establishing connectivity for the NS: For only TWO interfaces

Between the NS itselfs: The network NS has no network connectivity.They have NO interfaces on their own and can not see the underline host network.

We first will establish connectivity bewteen the net NS themselves.

Like connecting two physical machines together using a cable to an ethernet interface on each machine, you can connect two NS together using a Virtual Ethernet Pair (veth) or a Virtual cable often refered as a **Pipe,** is like a virtual cable with two interfaces on either end.

Create the Pipe (cable): between the two NS

[root@centos8 ~]# **ip link add veth-red type veth peer name veth-blue** → Creating one (veth-red) the other will be created too.

Attatch each interface to the appropiate NS:

[root@centos8 ~]# ip link set veth-red netns red
[root@centos8 ~]# ip link set veth-blue netns blue

Assingn IP addresses to each NS: Add the netmask. Add a rule to Firewalld/Iptables if doesn't ping.

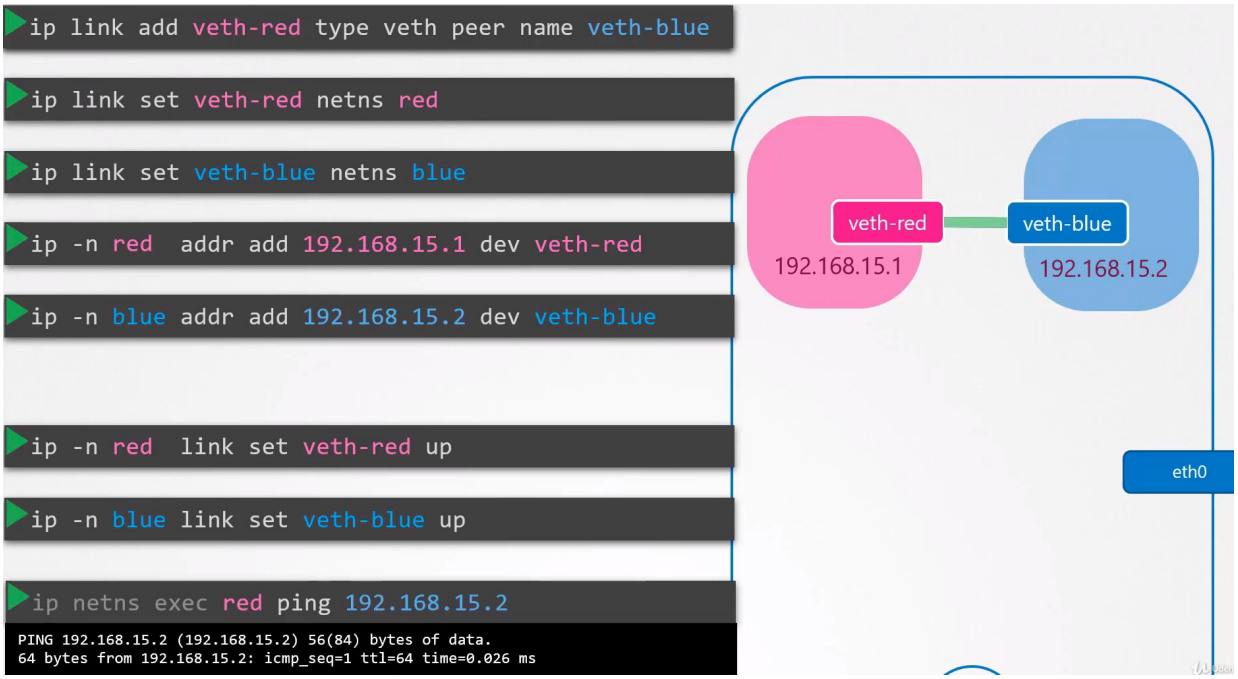
[root@centos8 ~]# ip -n red addr add 192.168.15.1/24 dev veth-red
[root@centos8 ~]# ip -n blue addr add 192.168.15.2/24 dev veth-blue

Bring the intetfaces up:

[root@centos8 ~]# ip -n red link set veth-red up
[root@centos8 ~]# ip -n blue link set veth-blue up

Ping the interfaces:

[root@centos8 ~]# **ip netns exec red ping 192.168.15.2**
PING 192.168.15.2 (192.168.15.2) 56(84) bytes of data.
64 bytes from 192.168.15.2: icmp_seq=1 ttl=64 time=0.029 ms
64 bytes from 192.168.15.2: icmp_seq=2 ttl=64 time=0.034 ms



Checks:

ARP:

```
[root@centos8 ~]# ip netns exec red arp
Address             HWtype  HWaddress           Flags Mask          Iface
192.168.15.2        ether   de:15:83:2a:b9:e2   C                   veth-red

[root@centos8 ~]# ip netns exec blue arp
Address             HWtype  HWaddress           Flags Mask          Iface
192.168.15.1        ether   a2:4d:46:c0:54:97   C                   veth-blue
```

Arp of the host: Can not see the new NS network

```
[root@centos8 ~]# arp
Address             HWtype  HWaddress           Flags Mask          Iface
_gateway            ether   0a:7e:ef:58:df:77   C                   eth0
```

IP Link:

```
[root@centos8 ~]# ip -n blue link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
5: veth-blue@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether de:15:83:2a:b9:e2 brd ff:ff:ff:ff:ff:ff link-netns red
```

IP link before starting the interface:

```
[root@centos8 ~]# ip -n blue addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
5: veth-blue@if6: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether de:15:83:2a:b9:e2 brd ff:ff:ff:ff:ff:ff link-netns red
    inet 192.168.15.2/32 scope global veth-blue
       valid_lft forever preferred_lft forever
```

IP ADDR:

```
[root@centos8 ~]# ip -n blue addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
5: veth-blue@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether de:15:83:2a:b9:e2 brd ff:ff:ff:ff:ff:ff link-netns red
    inet 192.168.15.2/32 scope global veth-blue
       valid_lft forever preferred_lft forever
    inet 192.168.15.2/24 scope global veth-blue
       valid_lft forever preferred_lft forever
    inet6 fe80::dc15:83ff:fe2a:b9e2/64 scope link
       valid_lft forever preferred_lft forever
```

Network Namespaces:

```
[root@centos8 ~]# ip netns
blue (id: 1)
red (id: 0)
```

Establishing connectivity for the Network NS: For Multiple NS

Connect two network namespaces it's easy. But if you want to connect more is better to use a virtual switch, which will avoid us to having to create and link one by one.

There are multiple solutions to do it, we will use **Linux Bridge.**

We will create a virtual interface. This v-interface will be detected by the host as a normal network interface and for the Network Namespaces will be like a switch.

Create the internal bridge network / Virtual Switch: We name it: v-net-0
[root@centos8 ~]# **ip link add v-net-0 type bridge**

For the host will be like another interface. You can see it with ip link:
[root@centos8 ~]# **ip link**
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP mode DEFAULT group default qlen 1000
link/ether 0a:2a:14:c3:ac:89 brd ff:ff:ff:ff:ff:ff
[...]
7: **v-net-0:** <BROADCAST,MULTICAST> mtu 1500 qdisc noop **state DOWN** mode DEFAULT group default qlen 1000
link/ether 72:16:6d:f3:15:4f brd ff:ff:ff:ff:ff:ff

Bring it up:
[root@centos8 ~]# **ip link set dev v-net-0 up**

Connect the NS to the virtual network switch:

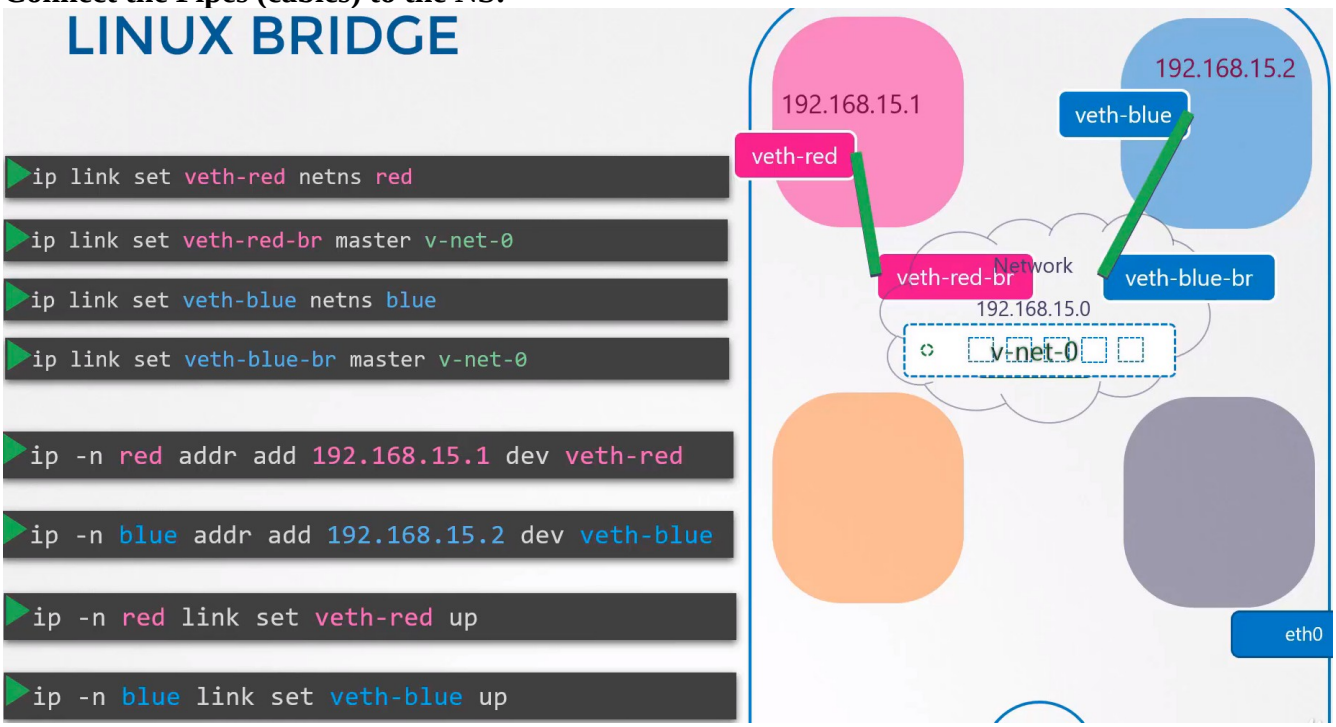
First delete the link created before between the blue and red Network Namespaces.

If you delete one the other gets deleted automatically.
[root@centos8 ~]# **ip -n red link del veth-red**

Create new Pipes (cables) with the bridge:
[root@centos8 ~]# **ip link add veth-red type veth peer name veth-red-br**
[root@centos8 ~]# **ip link add veth-blue type veth peer name veth-blue-br**

TIP: If you have to delete an interface created by mistake:
Flush it first: ip addr flush dev veth-reeeed-br
Then delete it: ip link delete veth-reeeed-br

Connect the Pipes (cables) to the NS:



[root@centos8 ~]# ip link set veth-red netns red
[root@centos8 ~]# ip link set veth-red-br master v-net-0

[root@centos8 ~]# ip link set veth-blue netns blue
[root@centos8 ~]# ip link set veth-blue-br master v-net-0

Set IP Addresses for the NS:
[root@centos8 ~]# ip -n red addr add 192.168.15.1/24 dev veth-red
[root@centos8 ~]# ip -n blue addr add 192.168.15.2/24 dev veth-blue

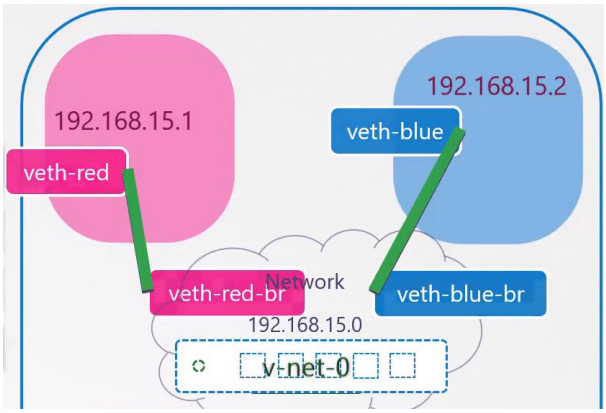
Turn the devices up:
[root@centos8 ~]# ip -n blue link set veth-blue up
[root@centos8 ~]# ip -n red link set veth-red up

Do the same for the other namespaces: orange and gray.

Turn up veth-red-br and veth-blue-br:
ip link set veth-red-br up
ip link set veth-blue-br up

Set an IP for the v-net-0 interface:
[root@centos8 ~]# ip addr add 192.168.15.5/24 dev v-net-0

Connect 4 Network Namespaces with a bridge:



Create the namespaces:
[root@centos8 ~]# ip netns add red
[root@centos8 ~]# ip netns add blue
[root@centos8 ~]# ip netns add orange
[root@centos8 ~]# ip netns add black

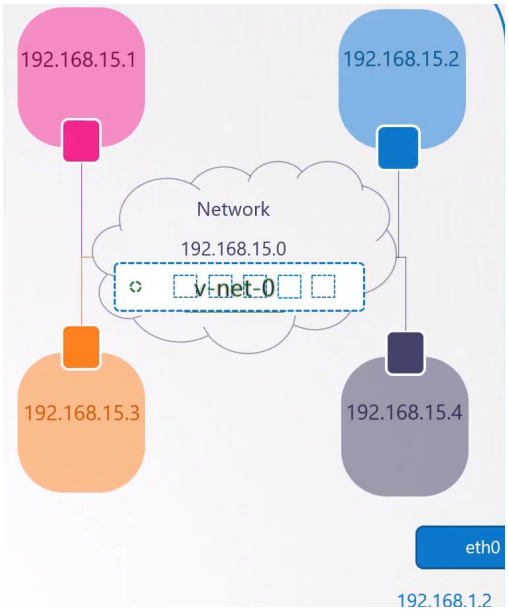
Create the Bridge (switch) interface:
[root@centos8 ~]# ip link add v-net-0 type bridge

Turn Up the bridge interface:
[root@centos8 ~]# ip link set v-net-0 up

Create AND link the veth interfaces with veth-br:
[root@centos8 ~]# ip link add veth-red type veth peer name veth-red-br
[root@centos8 ~]# ip link add veth-blue type veth peer name veth-blue-br
[root@centos8 ~]# ip link add veth-orange type veth peer name veth-orange-br
[root@centos8 ~]# ip link add veth-black type veth peer name veth-black-br

Attach the veth int to the Namespace:
[root@centos8 ~]# ip link set veth-red netns red
[root@centos8 ~]# ip link set veth-blue netns blue
[root@centos8 ~]# ip link set veth-orange netns orange
[root@centos8 ~]# ip link set veth-black netns black

Attach the veth-br to the v-net-0:
[root@centos8 ~]# ip link set veth-red-br master v-net-0
[root@centos8 ~]# ip link set veth-blue-br master v-net-0
[root@centos8 ~]# ip link set veth-orange-br master v-net-0
[root@centos8 ~]# ip link set veth-black-br master v-net-0



Set IP addresses:
[root@centos8 ~]# ip -n red addr add 192.168.15.1/24 dev veth-red
[root@centos8 ~]# ip -n blue addr add 192.168.15.2/24 dev veth-blue
[root@centos8 ~]# ip -n orange addr add 192.168.15.3/24 dev veth-orange
[root@centos8 ~]# ip -n black addr add 192.168.15.4/24 dev veth-black

Turn interfaces UP:
[root@centos8 ~]# ip -n red link set veth-red up
[root@centos8 ~]# ip -n blue link set veth-blue up
[root@centos8 ~]# ip -n orange link set veth-orange up
[root@centos8 ~]# ip -n black link set veth-black up

[root@centos8 ~]# ip link set veth-red-br up
[root@centos8 ~]# ip link set veth-blue-br up
[root@centos8 ~]# ip link set veth-orange-br up
[root@centos8 ~]# ip link set veth-black-br up

Ping:
[root@centos8 ~]# ip netns exec red ping 192.168.15.2
PING 192.168.15.2 (192.168.15.2) 56(84) bytes of data.
64 bytes from 192.168.15.2: icmp_seq=1 ttl=64 time=0.062 ms
64 bytes from 192.168.15.2: icmp_seq=2 ttl=64 time=0.045 ms
64 bytes from 192.168.15.2: icmp_seq=3 ttl=64 time=0.046 ms
64 bytes from 192.168.15.2: icmp_seq=4 ttl=64 time=0.045 ms

Checks:

Check Network Namespaces:
[root@centos8 ~]# ip netns

ARP:
[root@centos8 ~]# arp
[root@centos8 ~]# ip netns exec red arp

IP LINK:
[root@centos8 ~]# ip link
[root@centos8 ~]# ip -n blue link

IP ADDR:
[root@centos8 ~]# ip addr
[root@centos8 ~]# ip -n blue addr

Routing Table:
[root@centos8 ~]# ip route
[root@centos8 ~]# ip netns exec red route

Ping:
[root@centos8 ~]# ip netns exec red ping 192.168.15.2

Establish connectivity between the host and the Network Namespaces:

Add IP to v-net-0:
[root@centos8 ~]# ip addr add 192.168.15.5/24 dev v-net-0

Ping from localhost to the Namespace network:
[root@centos8 ~]# ping 192.168.15.1
PING 192.168.15.1 (192.168.15.1) 56(84) bytes of data.
64 bytes from 192.168.15.1: icmp_seq=1 ttl=64 time=0.081 ms
64 bytes from 192.168.15.1: icmp_seq=2 ttl=64 time=0.057 ms

[root@centos8 ~]# ping 192.168.15.5
PING 192.168.15.5 (192.168.15.5) 56(84) bytes of data.
64 bytes from 192.168.15.5: icmp_seq=1 ttl=64 time=0.067 ms
64 bytes from 192.168.15.5: icmp_seq=2 ttl=64 time=0.069 ms

Reach another host in the LAN network from the Namespace:

Add entry into the NamespaceS routing table:
[root@centos8 ~]# ip netns exec red ip route add 172.31.16.0/20 via 192.168.15.5
[root@centos8 ~]# ip netns exec blue ip route add 172.31.16.0/20 via 192.168.15.5
[root@centos8 ~]# ip netns exec orange ip route add 172.31.16.0/20 via 192.168.15.5
[root@centos8 ~]# ip netns exec black ip route add 172.31.16.0/20 via 192.168.15.5

Ping: → 172.31.17.210 is the IP of eth0 in the host
[root@centos8 ~]# ip netns exec red ping 172.31.17.210
PING 172.31.17.210 (172.31.17.210) 56(84) bytes of data.
64 bytes from 172.31.17.210: icmp_seq=1 ttl=64 time=0.168 ms
64 bytes from 172.31.17.210: icmp_seq=2 ttl=64 time=0.070 ms

IP Route:

```
[root@centos8 ~]# ip netns exec red route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.31.16.0     192.168.15.5    255.255.240.0   UG    0      0        0 veth-red
192.168.15.0    0.0.0.0         255.255.255.0   U     0      0        0 veth-red
```

Enable Nat on the host:

```
[root@centos8 ~]# iptables -t nat -A POSTROUTING -s 192.168.15.0/24 -j MASQUERADE
```

We add a new rule in the PostRouting chain to Masquerade or Replace the FROM address on all packets coming from the source network (192.168.15.0) with it's own IP address.

That way anyone receiving these packages outside the network will think they are coming from the host (172.31.16.0/20) and not from within the Namespaces.

Reach Internet from the Namespaces:

Add default route from NS to v-net-0 IP:

```
[root@centos8 ~]# ip netns exec blue ip route add default via 192.168.15.5
[root@centos8 ~]# ip netns exec red ip route add default via 192.168.15.5
[root@centos8 ~]# ip netns exec orange ip route add default via 192.168.15.5
[root@centos8 ~]# ip netns exec black ip route add default via 192.168.15.5
```

Ping to internet:

```
[root@centos8 ~]# ip netns exec blue ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=50 time=0.932 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=50 time=0.926 ms
```

Connectivity from Internet to the Namespaces:

Forward any traffic coming from Inet to the port 80 to the blue NS IP port 80:

```
[root@centos8 ~]# iptables -t nat -A PREROUTING --dport 80 --to-destination 192.168.15.2:80 -j DNAT
```

Docker Networking:

Scenario: We have a single host running containers.

Networking options:

- **None network:** The Docker container is NOT attached to any network. The container can not reach the outside world and no one on the outside world can reach the container. If there are multiple containers they can not reach each other neither.

- **Host network:** The container is attached to the host network, without network isolation. If you deploy a nginx it will listen and use the port 80 in the host. And NO other container could use the same port.

- **Bridge network:** An internal private network is created. The docker hosts and containers are attached to. Has 172.17.0.0 by default and each device connecting to this network get an IP of this network.

Bridge Network: When Docker is installed in a host it creates a bridge network by default.

docker network ls → will show you the networks in the host. You will see the bridge there plus one *host* and one *none*.
Docker creates the interface with the name Brige

But the host creates the interface with the name Docker0
ip link → will show you the host interfaces.

Docker uses the same techique similar to namespaces. Docker creates the bridge interface by running:
ip link add docker0 type bridge

Docker assigns the to the interface docker0 the IP 172.17.0.1 → *ip addr* to see it

Docker creates a network namespace for each container created.

ip netns → to list all the NS

docker inspect <namespace> → will show you the container associated to the namespace
docker inspect b4d34e345b2

Docker attaches the Namespace to the bridge the same way we did with the normal namespaces.

It creates a Pipe (virtual cable) creating two intercaces on each end.
ip link → **vethbb1c343@if7**
ip -n b4d34e345b2 link → **eth0@if8**

And assing an IP to the container: *ip -n b4d34e345b2 addr*

Summary:
1. Docker creates the Network Namespaces
2. Creates a pair of interfaces
3. Attaches one interface to the container and the other interface to the bridge network.

Port mapping:
We have a docker nginx, only the other containers will be able to see its webpage.
curl http://172.17.0.3:80

To make it reachable from outside run the docker in this way:
docker run -p 8080:80 nginx

You tell docker to map the port 80 on the local host to 8080 in the container.
Now the external users can point to the host ip and the port 8080:

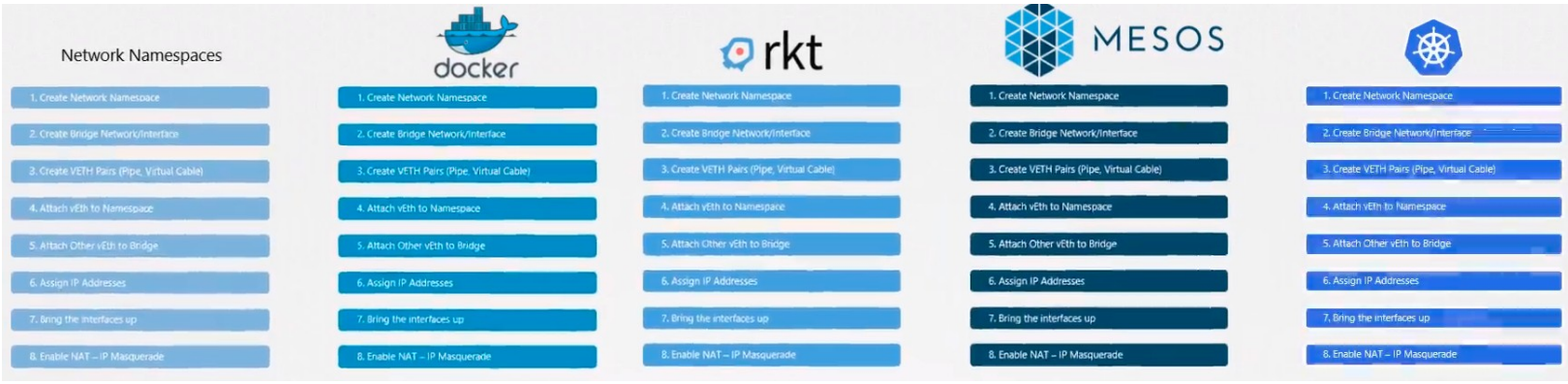
curl http://192.168.1.10:8080

Docker creates natrules using IP tables to do the mappings:
iptables -t nat -A DOCKER -j DNAT --dport 8080 --to-destination 172.17.0.3:80

Check the rule:
iptables -nvL -t nat

CNI - Container Network Interface:

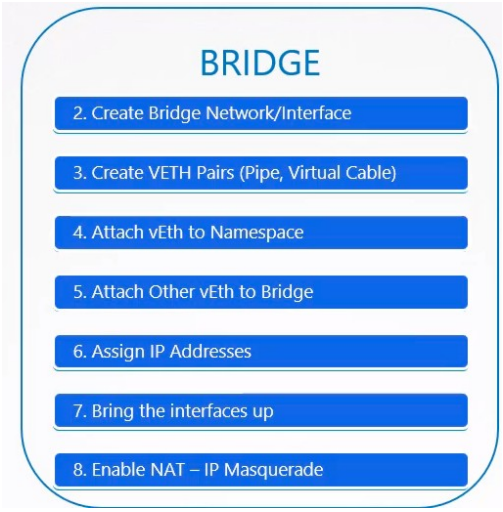
All the container solutions works the same way creating the network.



Instead of developing the same solution multiple times they created a single standard approach that everyone can follow.

We take all these ideas from the different solutions and move all the networking portions of it into a single program or code. We will call this program or code a Plugin. There are multiple plugins.

We will use the Bridge plugin as example:



We created a program (plugin) that performs all the required task to get the container attached to a bridge network.

We can do it by running this command: bridge add 2e34dcf34 /var/run/netns/ 2e34dcf34
We specify the bridge program to add the container 2e34dcf34 to a particular network namespace.

The bridge program takes care of the rest so the container runtime is released from those tasks.

Example: When k8s or rkt creates a new container they call the bridge program and pass the container id and namespace to get networking configured for that container.

To create a bridge plugin a container runtime company have to follow a list of standard solutions to do it, the CNI.

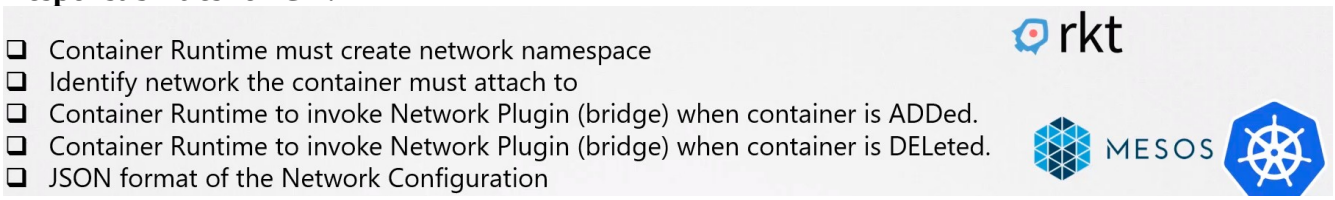
CNI: Set of standards that define how programs should be developed to solve networking challenges in a container runtime environment.

The programs are refered to as plugins. There are several of them:

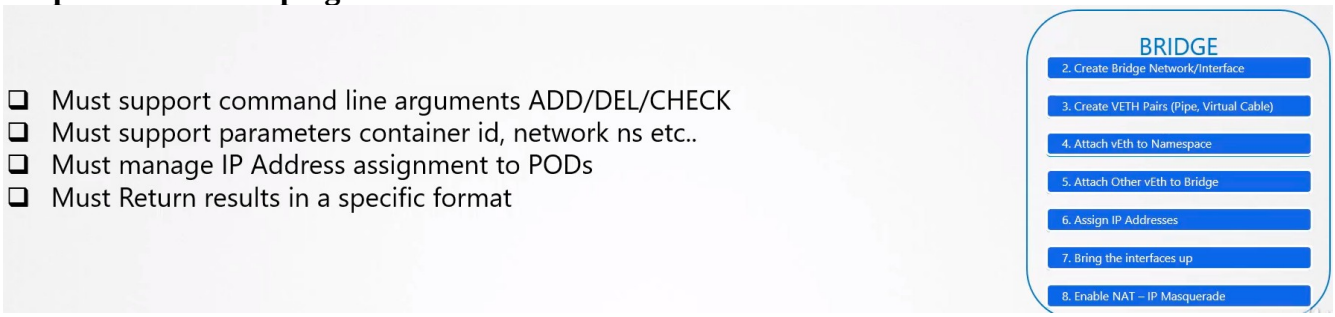
- Bridge
- Vlan
- IPVlan
- MacVlan
- Windows
- Dhcp
- host-local
- 3rd party plugins, weawe, flannel, cilium, nsx, calico, etc.

CNI defines a set of responsabilities that a container runtime (CR) and plugins have to follow#:

Responsabilities for CR:



Responsabilities for plugins:



As long as the CR and the plugins are following the standard they can live together in harmony.
Any CR should be able to work with any plugin.

Docker does NOT implement CNI:
Docker uses it's own standard, CNM, Container Network Model.

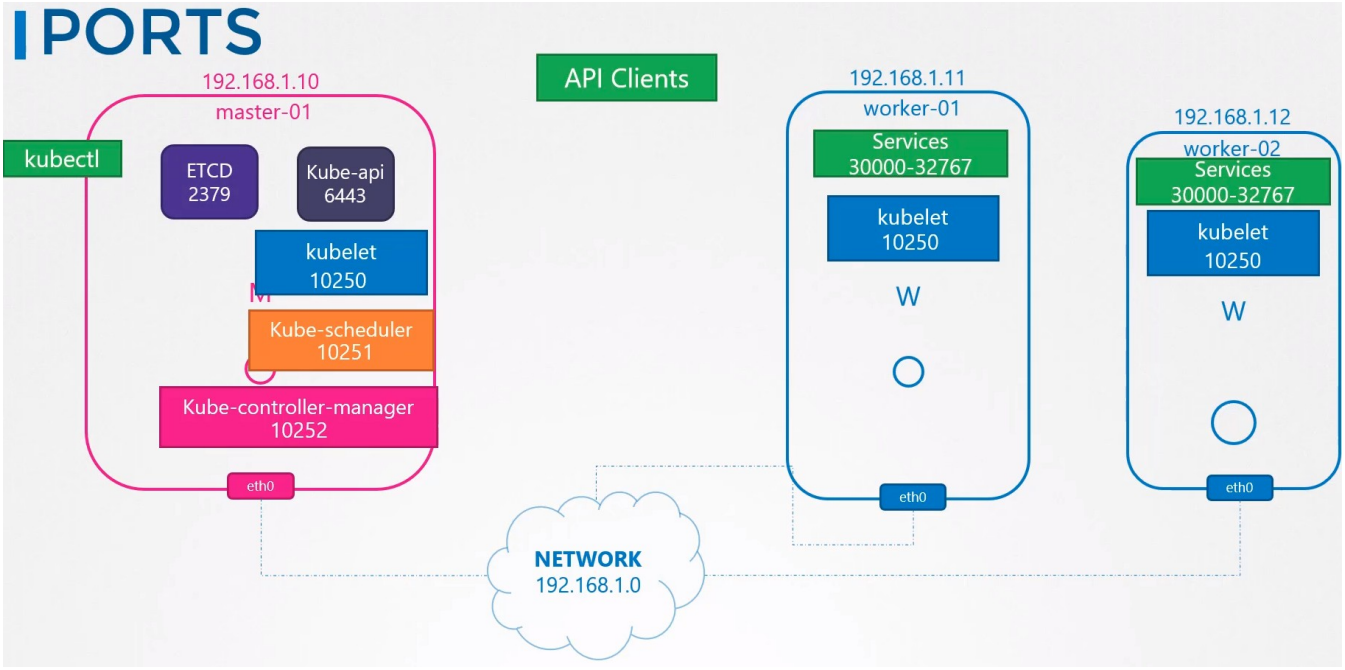
To use it you have to create a network container with a NONE network and then invoke the bridge plugin.

docker run --network=none nginx
bridge add 2e34dcf34 /var/run/netns/ 2e34dcf34

This is how K8s does it.

When k8s creates docker containers it creates them on the none network.
It then invoke the configured CNI plugins to take care of the rest of the configuration.

Cluster Networking:



Cluster components ports list

Commands to troubleshoot:

ip route show default

CNI Exam Instalation:

Exam Tip:

You can be asked to install a CNI. Normally you can choose the one that you want.

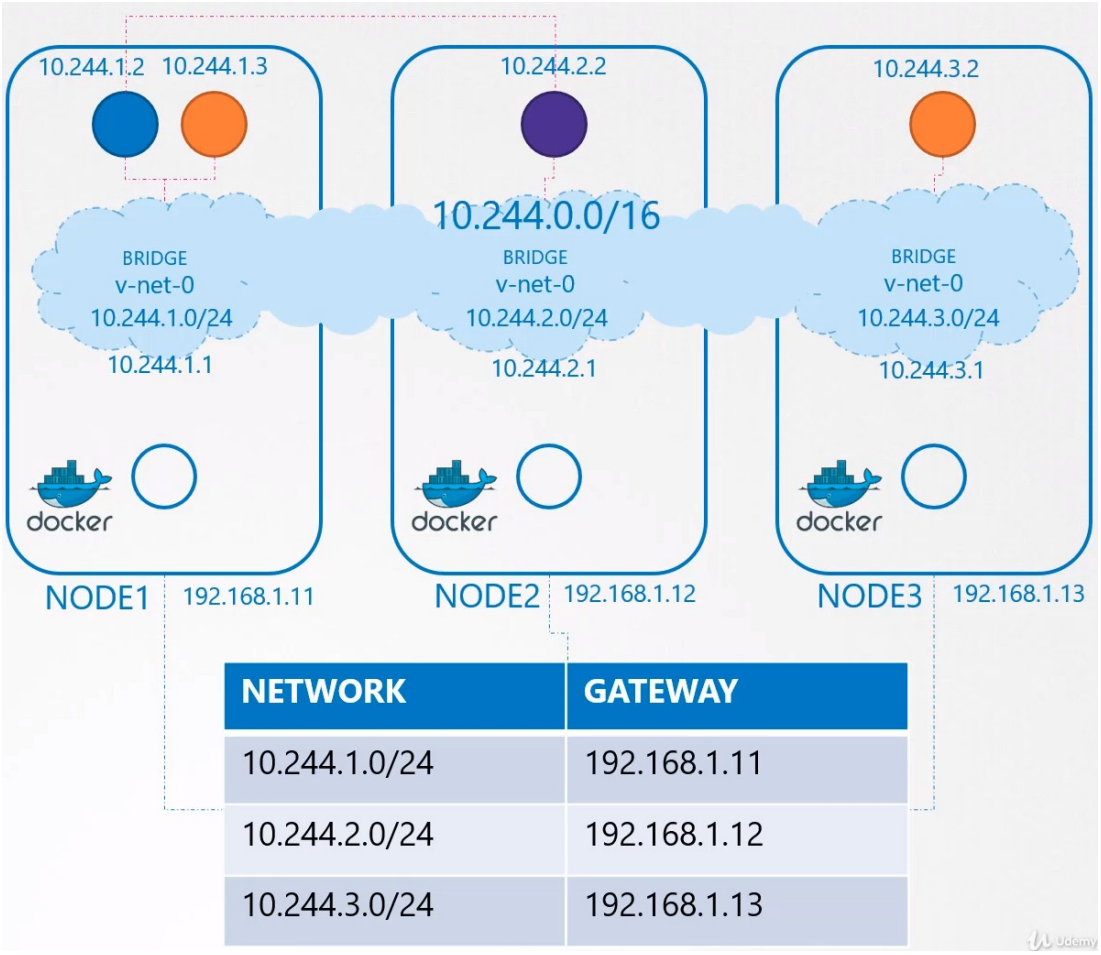
But in kubernetes.io you can not see how to install them, you only have the links to the external websites, and is not allowed to access external websites.

There is a page in k8s docs that explain how to install weave network addon, is the step two:
Weave network addon

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

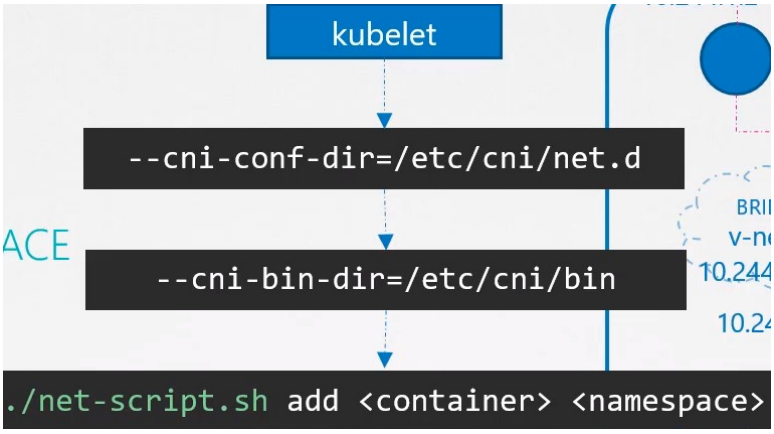
POD Networking:

K8s creates a router where specifies the network and which gateway to use to reach it.



NETWORK	GATEWAY
10.244.1.0/24	192.168.1.11
10.244.2.0/24	192.168.1.12
10.244.3.0/24	192.168.1.13

Kubelet on each node is the one creating containers and when the container is created the kubelet looks at the CNI configuration passed as CLI argument when it was run and identifies the network creation script name.

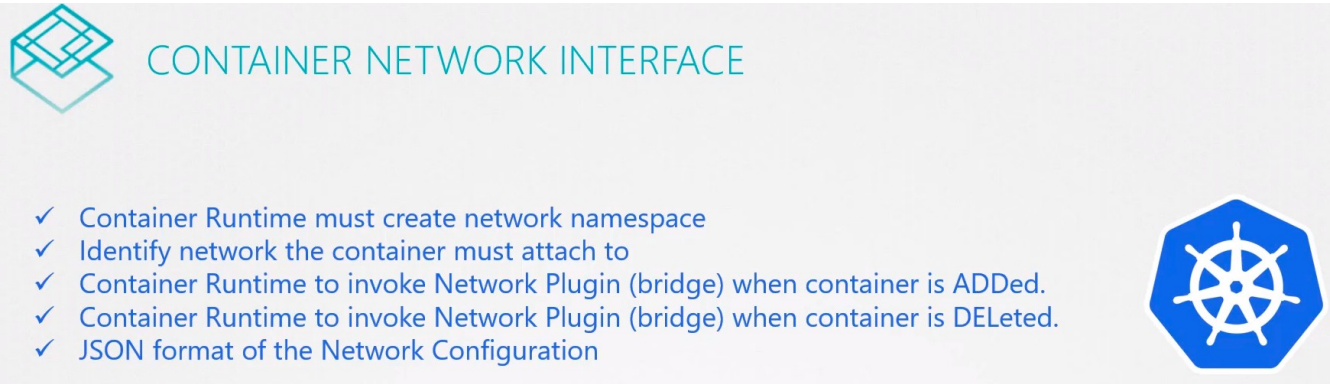


It then looks at the CNI bin directory define the scripts and then executes the script with the container id and the namespace.

Then the script takes care of the rest.

CNI in kubernetes: How k8s is configured to use the network plugins.

CNI defines the responsabilities of container runtime, k8s in this case:



Where do we specify the CNI plugins for k8s to use. The CNI plugin must be invoke by the k8s component responsible for creating containers, kubelet.

Kubelet must invoke the cni plugin AFTER the container is created.

The CNI plugin is configured in the kubelet service on each node in the cluster.

kubelet.service

```
ExecStart=/usr/local/bin/kubelet \\
  --config=/var/lib/kubelet/kubelet-config.yaml \\
  --container-runtime=remote \\
  --container-runtime-endpoint=unix:///var/run/containerd/containerd.sock \\
  --image-pull-progress-deadline=2m \\
  --kubeconfig=/var/lib/kubelet/kubeconfig \\
  --network-plugin=cni \\
  --cni-bin-dir=/opt/cni/bin \\
  --cni-conf-dir=/etc/cni/net.d \\
  --register-node=true \\
  --v=2
```

You can see it too in the kubelet process:

```
ps -aux | grep kubelet
root      2095  1.8  2.4 960676 98788 ?        Ssl  02:32   0:36 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroupfs --cni-bin-dir=/opt/cni/bin --cni-conf-dir=/etc/cni/net.d --network-plugin=cni
```

See all the CNI plugins (binaries):

```
ls /opt/cni/bin
bridge  dhcp  flannel  host-local  ipvlan  loopback  macvlan  portmap  ptp  sample  tuning  vlan  weave-ipam  weave-net  weave-plugin-2.2.1
```

See the configuration files,where kubelet looks to find out which plugin needs to be used.

```
ls /etc/cni/net.d
10-bridge.conf
```

In this case kubelet finds the bridge configuration file. If there are more than one conf file kubelet selects it in alphabethical order.

Look at bridge.conf: Format defined by the CNI standard.

```
cat /etc/cni/net.d/10-bridge.conf
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

IsGateway: Defines wheter the bridge network should get an IP address assigned to it so it can act as a gateway.

ipMasq: Defines if a NAT rule should be added for IP masquerading.

ipam: Defines ipam configuration. This is where you specify the subnet or the range of IP addresses that will be assigned to PODs and any necessary routes.

type: host-local → The IP addresses are managed localy on this host, unlike DHCP server mantaining them remotely. You can set the type to DHCP to configure an external DHCP server.

CNI weave:

WeaveWorks: Is one solution based in CNI. It deploys an agent or service in each node.

Each agent in each node communicate with the others to exchange information regarding nodes, networks and pods within them.

Each agent or peer stores a topology of the entire set up where they know the pods and their IPs on the other nodes.

Weave creates its own bridge on the nodes and names it *weave*. Then assings an IP address to each network.

A single POD may be attached to multiple bridge networks. You could have a POD attached to the Weave bridge and to the Docker bridge created by Docker.

What route the packet takes to reach destination depence on the route configured on the container.

When a packet is sent from one POD to another on another node Weave intercepts the packet and identifies that is in a separated network it then encapsules this packet into a new one with new sorce and destination and sends it across the newtork.

Once on the other side the other Weave agent retrieves the packet decapsulates it and routes the packet to the right POD.

How to deploy Weave on a K8s cluster:
Weave and Weave peers can be deployed as services or deamons on each node in the cluster **manually** or if k8s is set up already an easier way to do that is to deploy as PODs in the cluster.

Once the base k8s system is ready with Nodes and networking configured correctly between the Nodes and the basic control plane components are deployed Weave can be deployed in the cluster with kubectl apply command:

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

This deploys all the necessary components required for Weave in the cluster.

The Weave Peers are deployed as Daemonsets, which ensures that one POD of the given kind, weave net in this case, is deployed on each node in the cluster.

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.extensions/weave-net created
```

If you deploy the cluster with the **kubneadm** tool and Weave plugin you can see the weave peers as PODs deployed on each Node.

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

```
NAME                              READY     STATUS    RESTARTS   AGE       IP              NODE
NOMINATED NODE
coredns-78fcdf6894-99khw          1/1       Running   0          19m       10.44.0.2       master    <none>
coredns-78fcdf6894-p7dpj          1/1       Running   0          19m       10.44.0.1       master    <none>
etcd-master                       1/1       Running   0          18m       172.17.0.11     master    <none>
kube-apiserver-master             1/1       Running   0          18m       172.17.0.11     master    <none>
kube-scheduler-master             1/1       Running   0          17m       172.17.0.11     master    <none>
weave-net-5gcmb                   2/2       Running   1          19m       172.17.0.30     node02    <none>
weave-net-fr9n9                   2/2       Running   1          19m       172.17.0.11     master    <none>
weave-net-mc6s2                   2/2       Running   1          19m       172.17.0.23     node01    <none>
weave-net-tbzvz                   2/2       Running   1          19m       172.17.0.52     node03    <none>
```

For troubleshooting view the logs:

```
kubectl logs weave-net-5gcmb weave –n kube-system
```

```
INFO: 2019/03/03 03:41:08.643858 Command line options: map[status-addr:0.0.0.0:6782 http-addr:127.0.0.1:6784 ipalloc-range:10.32.0.0/12 name:9e:96:c8:09:bf:c4 nickname:node02 conn-limit:30
datapath:datapath db-prefix:/weavedb/weave-net host-root:/host port:6783 docker-api: expect-npc:true ipalloc-init:consensus=4 no-dns:true]
INFO: 2019/03/03 03:41:08.643980 weave  2.2.1
INFO: 2019/03/03 03:41:08.751508 Bridge type is bridged_fastdp
INFO: 2019/03/03 03:41:08.751526 Communication between peers is unencrypted.
INFO: 2019/03/03 03:41:08.753583 Our name is 9e:96:c8:09:bf:c4(node02)
INFO: 2019/03/03 03:41:08.753615 Launch detected - using supplied peer list: [172.17.0.11 172.17.0.23 172.17.0.30 172.17.0.52]
INFO: 2019/03/03 03:41:08.753632 Checking for pre-existing addresses on weave bridge
INFO: 2019/03/03 03:41:08.756183 [allocator 9e:96:c8:09:bf:c4] No valid persisted data
INFO: 2019/03/03 03:41:08.761033 [allocator 9e:96:c8:09:bf:c4] Initialising via deferred consensus
INFO: 2019/03/03 03:41:08.761091 Sniffing traffic on datapath (via ODP)
INFO: 2019/03/03 03:41:08.761659 ->[172.17.0.23:6783] attempting connection
INFO: 2019/03/03 03:41:08.817477 overlay_switch ->[8a:31:f6:b1:38:3f(node03)] using fastdp
INFO: 2019/03/03 03:41:08.819493 sleeve ->[172.17.0.52:6783|8a:31:f6:b1:38:3f(node03)]: Effective MTU verified at 1438
INFO: 2019/03/03 03:41:09.107287 Weave version 2.5.1 is available; please update at https://github.com/weaveworks/weave/releases/download/v2.5.1/weave
INFO: 2019/03/03 03:41:09.284907 Discovered remote MAC 8a:dd:b5:14:8f:a3 at 8a:dd:b5:14:8f:a3(node01)
INFO: 2019/03/03 03:41:09.331952 Discovered remote MAC 8a:31:f6:b1:38:3f at 8a:31:f6:b1:38:3f(node03)
INFO: 2019/03/03 03:41:09.355976 Discovered remote MAC 8a:a5:9c:d2:86:1f at 8a:31:f6:b1:38:3f(node03)
```

See the CNI plugin configured: Or do: kubectl get pods -A → you will see the plugin configured.

controlplane $ **ls -l /etc/cni/net.d/**
-rw-r--r-- 1 root root 318 Nov 2 10:12 10-weave.conflist

controlplane $ **cat /etc/cni/net.d/10-weave.conflist**

```
{
   "cniVersion": "0.3.0",
   "name": "weave",
   "plugins": [
      {
         "name": "weave",
         "type": "weave-net",
         "hairpinMode": true
      },
      {
         "type": "portmap",
         "capabilities": {"portMappings": true},
         "snat": true
      }
   ]
}
```

controlplane $ **ls /opt/cni/bin/**
bandwidth bridge dhcp firewall host-device host-local ipvlan loopback macvlan portmap ptp sbr static tuning vlan weave-ipam weave-net weave-plugin-2.7.0

IP Address Management - Weave - IPAM

Who is responsible to ensuring there are no duplicated IPs assigned: CNI Plugin is responsible on managing the IP address assignment to PODs.

CNI plugin comes with 2 built in plugins to manage the IP assignement:

- DHCP plugin
- host-local → Manages a list of all the IP addresses. Each Node have one list.

host-local configuration:

```
cat /etc/cni/net.d/net-script.conf
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "net-script",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

How WeaveWorks manages IP addresses:

By default alocates the range 10.32.0.0/12 for the entire network, from 10.32.0.1 to 10.47.255.254, which gives 1.048.574 IPs.

The peers decide to split the IPs iqually of this range between each Nodes.
You can configure the ranges to be assigned to the nodes passing a few options when deploying the weave network to the cluster.

Service Networking:

Kubelet: Ran in every node as daemon. Is listening the kube-apiserver and is responsible for creating PODs.

Each kubelet in each node watches the changes on the cluster through the kube-apiserver and everytime a POD needs to be created it create it on the node.

It then invokes the CNI plugin to configure networking for that POD.

Kube-proxy: Similarly each node runs kube-proxy which watches the changes in the cluster through kube-apiserver and everytime a new Service is to be created kube-proxy gets into action.

Unlike PODs, services are not created on each node because are cluster wide concept.

Services are virtual objects,there are no proceses or namespaces or interfaces for a service.

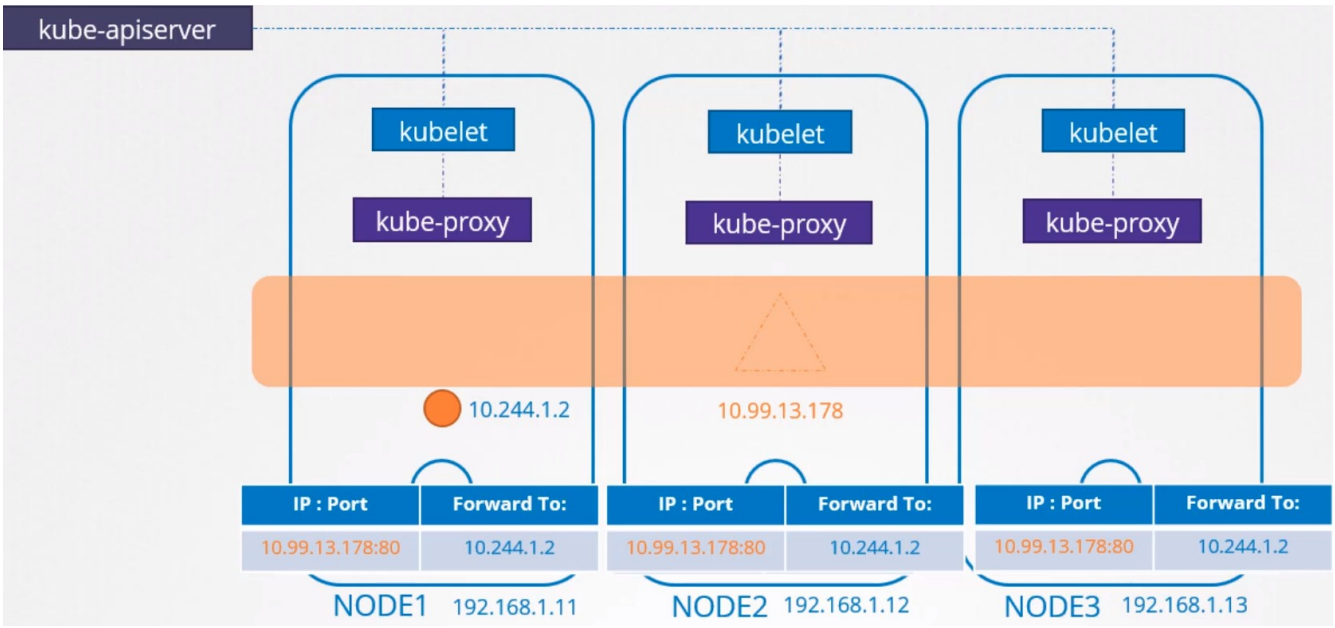
How do they get an IP and how are we able to access it:

When a service object is created an IP address is assigned to it from a predefined range.

The kube-proxy running on each node gets that IP and creates forwarding rules on each node in the cluster saying:

Any traffic coming to this IP:PORT (of the service) should go to the IP:PORT (of the POD)

Whenever a Service is created or deleted, the kube-proxy component creates or deletes these rules.



How these Rules are created:

Kube-proxy supports different ways, such as:

- userspace → where kube-proxy listens on a port for each service and proxies connections to pods.
- ipvs rules
- iptables

Where to set the rules:
Configuring the proxy service: kube-proxy --proxy-mode [userspace | iptables | ipvs] …

If nothing is specified configures the iptables mode.

How IPtables are configured and how to view them on the nodes:
iptables -L -t net |grep <name_of_the_service>

```
kubelet get service
NAME            TYPE        CLUSTER-IP       PORT(S)          AGE
db-service      ClusterIP   10.103.132.104   3306/TCP         12h

iptables -L -t net | grep db-service
KUBE-SVC-XA5OGUC7YRHOS3PU  tcp  --  anywhere   10.103.132.104   /* default/db-service: cluster IP */ tcp dpt:3306
DNAT                       tcp  --  anywhere   anywhere         /* default/db-service: */ tcp to:10.244.1.2:3306
KUBE-SEP-JBWCWHHQM57V2WN7  all  --  anywhere   anywhere         /* default/db-service: */
```

These rules mean:
Any traffic going to the IP and port of the service: 20.103.132.104 : 3306
should have its destination address changed to 10.244.1.2 : 3306 which is the IP of the POD

This is done by adding a DNAT rule to Iptables.

Similarly when you create a Service of type NodePort kube-proxy created IPtables rules to forward all traffic coming on a port on all nodes to the respective backend PODs.

See kube-proxy entries on logs:

```
cat /var/log/kube-proxy.log
I0307 04:29:29.883941       1 server_others.go:140] Using iptables Proxier.
I0307 04:29:29.912037       1 server_others.go:174] Tearing down inactive rules.
I0307 04:29:30.027360       1 server.go:448] Version: v1.11.8
I0307 04:29:30.049773       1 conntrack.go:98] Set sysctl 'net/netfilter/nf_conntrack_max' to 131072
I0307 04:29:30.049945       1 conntrack.go:52] Setting nf_conntrack_max to 131072
I0307 04:29:30.050701       1 conntrack.go:83] Setting conntrack hashsize to 32768
I0307 04:29:30.050701       1 proxier.go:294] Adding new service "default/db-service:3306" at 10.103.132.104:3306/TCP
```

Where you can see what rules kube-proxy uses (IPtables in this example) and one entry when a new service was added.

The location of the file may vary depending on the instalation.

How IP ranges are assigned:
IP range that will be assigned to a Service: kube-api-server **--service-cluster-ip-range** ipNet (default is 10.0.0.0/24)

ps aux |grep kube-api-server → to see which range you were assigned

How to see the IP range assigned to Nodes:

kubectl get nodes -owide → to get the IP of the nodes

ip addr |grep ens3 → to see the range

controlplane $ **ip addr |grep ns3**
2: ens3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
inet **172.17.0.89/16** brd 172.17.255.255 scope global ens3

See range of IP configured for PODs:
kubectl logs -n kube-system weave-net-cqfhr weave
[...]
INFO: 2020/11/03 14:30:38.854635 Command line options: map[conn-limit:200 datapath:datapath db-prefix:/weavedb/weave-net docker-api: expect-npc:true host-root:/host http-addr:127.0.0.1:6784 ipalloc-init:consensus=1 **ipalloc-range:10.32.0.0/12** metrics-addr:0.0.0.0:6782 name:a6:f4:07:2c:14:fd nickname:controlplane no-dns:true no-masq-local:true port:6783]
[...]

See IP range configured for Services:

controlplane $ cat /etc/kubernetes/manifests/**kube-apiserver**.yaml
[...]
- --service-cluster-ip-range=10.96.0.0/12
[...]

See type of proxy configured:

controlplane $ kubectl logs -n kube-system kube-proxy-sx2wk
[…]
I1103 14:29:53.692964 1 server_others.go:186] Using **iptables Proxier.**
[...]

DNS in K8s:

How you can address a service or pod from another pod.

The FQDN for services is:

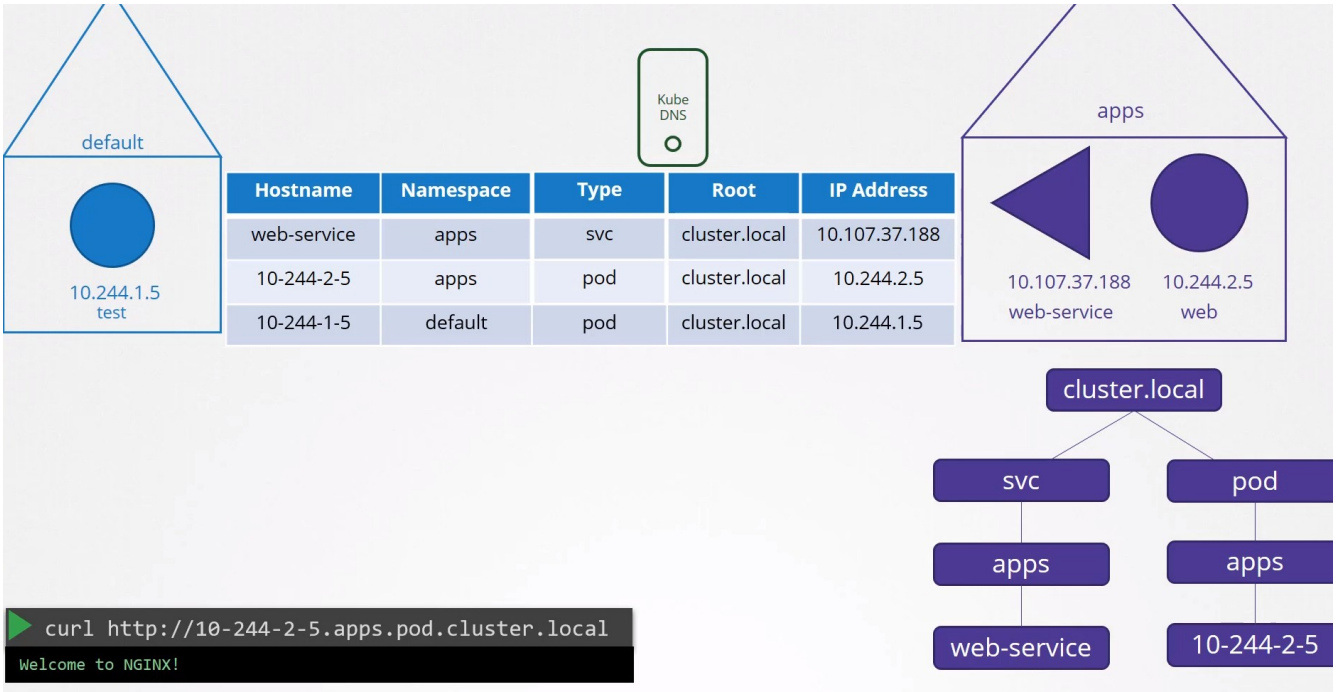
Name of the service: *web-service* in this example

Namespace in which the service is: *apps* in this example. If we are querying the service from the same namespace is not necessary to add the FQDN, only http://web-service will be ok

Type: *svc* for service. Pod for pod, and so on.

Root: cluster.local → it's always the same.

Full url: http://web-service.apps.pod.cluster.local



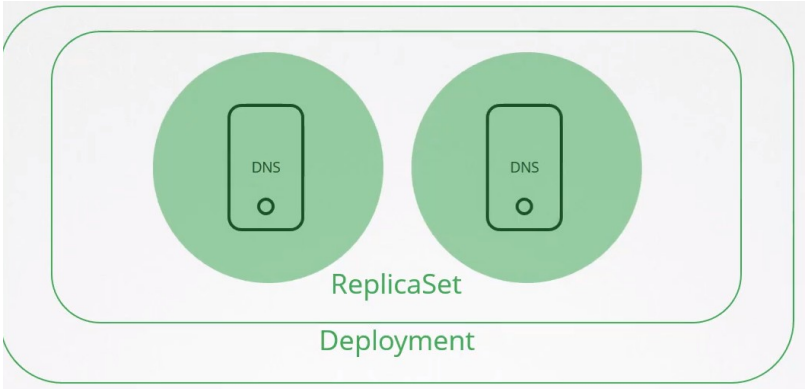
In the case of a POD: You have to enable to reach the pod by it's hostname. Once enabled follows the same procedure more or less than the svc, except:

Hostname: Thakes the IP of the pod and changes the dots (.) for dashes (-) → 10-244-1-5

CoreDNS in Kubernetes:

CoreDNS is the recommended DNS server since k8s 1.12. Previously kube-dns was used.

How is CoreDNS set up in the cluster: It is deployed as two PODs for redundancy as part of a ReplicaSet within a Deployment.



The PODs run **./Coredns** as executable, the same executable that we ran when execute CoreDNS ourselves.

./Coredns requires a config file, which is in **/etc/coredns/Corefile**

```
cat /etc/coredns/Corefile
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
       pods insecure
       upstream
       fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    proxy . /etc/resolv.conf
    cache 30
    reload
}
```

Whithin this file you have a number of plugins configured, highlited in orange.

Plugins are used for handling errors, reporting health, monitoring metrics, cache, etc.

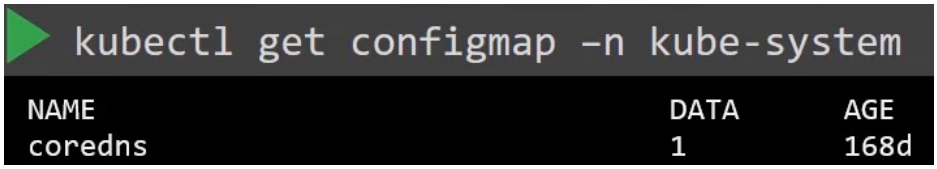
The plugin that makes coredns work with k8s is the kubernetes plugin.
That is where the TLD of the cluster is set, in this case **cluster.local**

Every record in the coredns DNS server falls into this domain.

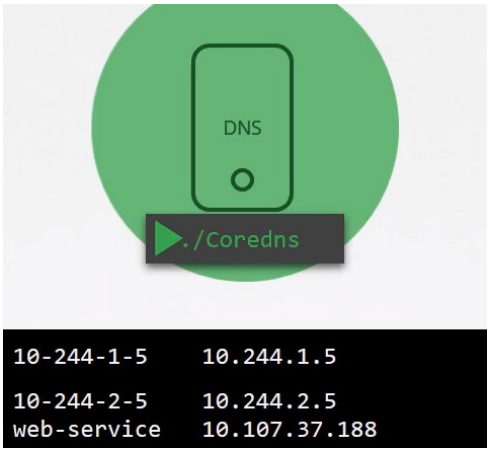
Within kubernetes plugin, the *pods insecure* option is what is responsible for creating a POD DNS record in the cluster. Is the one that convert the IP of the pod into dashes like 10-244-1-5.
This is not set by default so you need to specify it if you want to reach PODs by FQDN.

proxy . /etc/resolv.conf → Any record that this DNS server can not solve, for example www.google.com it it forwarded to the nameserver specified in /etc/resolve.conf which is set to use the nameserver from the k8s Node.

This Corefile is passed into the POD as a ConfigMap object.That way if you need to modify the configuration you can edit the ConfigMap object.



With CoreDNS up and running with the appropiate plugin it watches the k8s cluster for new PODs and Services and every time a pod or svc is created it adds a record for it in it's db.



How to point the PODs to the DNS server:
When we deploy CoreDNS solution it also creates a Service to make it available to other components within the cluster.

The service is named as **kube-dns** by default and the IP of this service is added by the **kubelet** as nameserver to the /etc/resolve.conf configuration of all the PODs when are created.

kubectl get service -n kube-system

If you take a look to the kubelet config file you will see the IP of the DNS server and domain on it:

```
cat /var/lib/kubelet/config.yaml
...
clusterDNS:
- 10.96.0.10
clusterDomain: cluster.local
```

Once the pods are configured with the nameserver you can now resolve other pods and services.
You can use the domain name to access the services:

```
curl http://web-service
curl http://web-service.default
curl http://web-service.default.svc
curl http://web-service.default.svc.cluster.local
```

nslookup or host command: When you run it it will get the FQDN and the IP:

```
host web-service
web-service.default.svc.cluster.local has address 10.97.206.196
```

/etc/resolve.conf
A part of the nameserver it contains the search entry which allows you to find web-service using any name:

```
cat /etc/resolv.conf
nameserver              10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local
```

```
host web-service
web-service.default.svc.cluster.local has address 10.97.206.196
```

```
host web-service.default
web-service.default.svc.cluster.local has address 10.97.206.196
```

```
host web-service.default.svc
web-service.default.svc.cluster.local has address 10.97.206.196
```

However PODs must use the whole FQDN because there is no entry for them in the search option.

Commands:

See where is the conf file for the DNS solution:
kubectl describe -n kube-system **deployments**.apps coredns |grep -A2 Args

```
    Args:
      -conf
      /etc/coredns/Corefile
```

How is the CoreFile conf passed to the PODs:
kubectl describe -n kube-system deployments.apps coredns

Search for Volumes.Type, it will tell you if is a ConfigMap and its name

See the root domain:
cat /var/lib/kubelet/config.yaml → clusterDomain: cluster.local

or

kubectl describe configmaps -n kube-system coredns → kubernetes cluster.local

Which domain to curl from one POD to another: in the same Namespace
Check the svc and POD for labels:

kubectl describe svc **test-service** → Selector: name=test AND TargetPort: 80/TCP

kubectl get pod test --show-labels → LABELS name=test

Now we know that to curl the test pod we have to do: curl http://test-service:80 if you are in the same Namespace.

Which domain to curl from one pod to another: In different Namespaces

Get the name of the destination Service.
Get the name of the destination Namespace.

curl http://service.namespace

Ingress:

Without ingress:

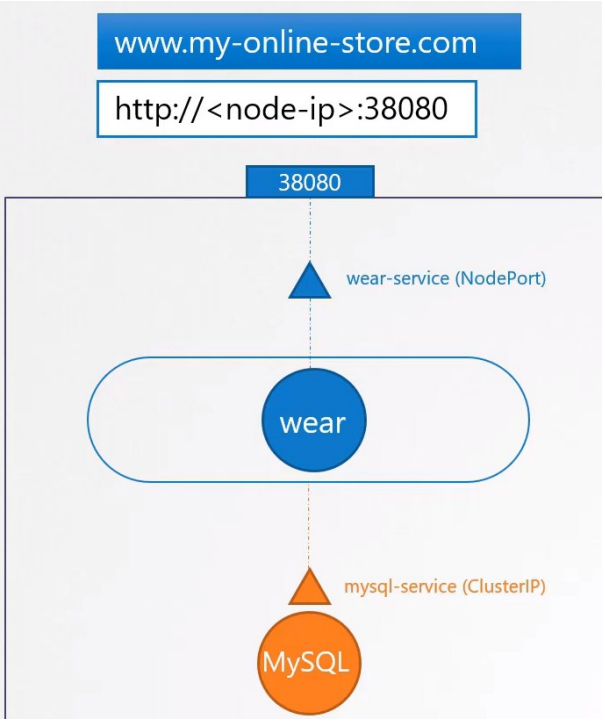
Basic Configuration: With NodePort Service App hosted on prem.
You have a web application that runs a clothes e-shop → **www.my-online-store.com**

You bild the app into a Docker image and deploy it into k8s cluster as a POD in a Deployment.

The app needs a db so I deploy a mysql as a POD and create a ClusterIP Service called *mysql-service* to make it accesible to my application. The application is now working (internally).

To make accesible the application to the outside world you create a NodePort Service called *wear-service* and make the application available on a high Port on the Nodes in the cluster.

Now the external users can reach the web application by typing the IP of one of our Nodes and the port 38080

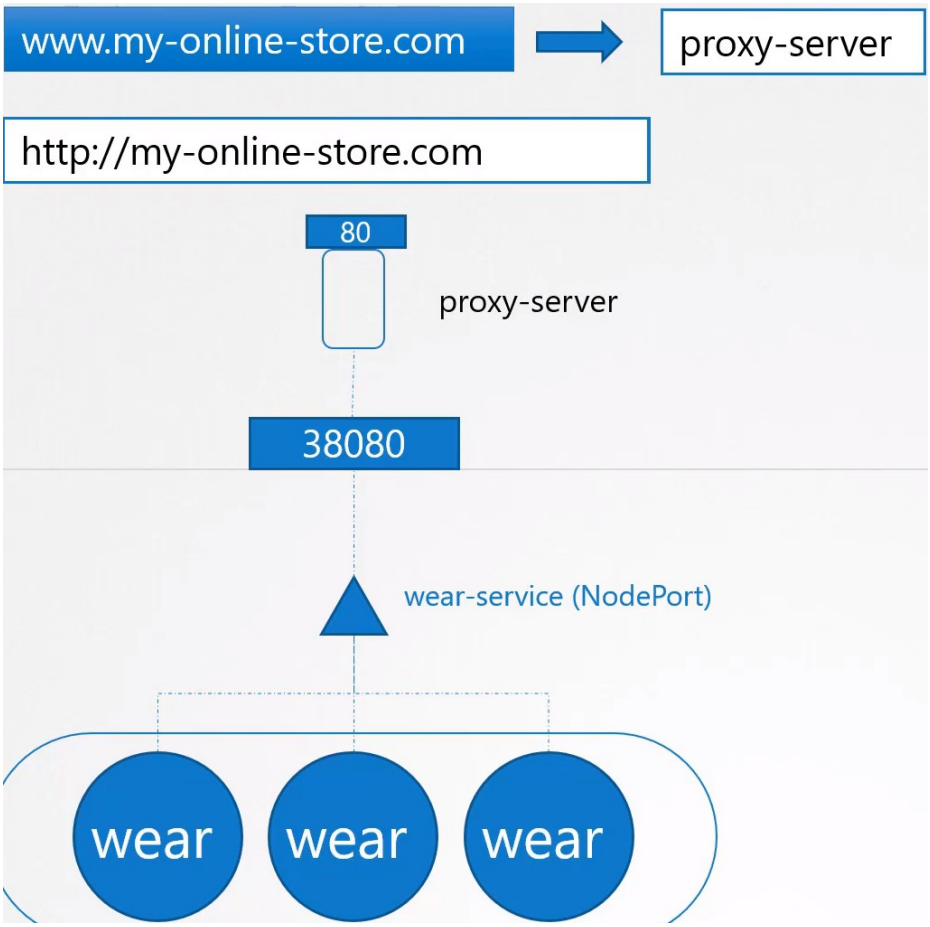


Advanced Configuration: App hosted on prem.

This is a production application and you don't want the users to type the IP addresses of the nodes to get the app everytime, so you configure your DNS to point the IP of the nodes.

The users are now able to reach the web app by typing http://my-online-store.com:38080

Now you don't want the users to remember a high Port number. NodePort Service is using high ports from 30000 to >38000, so you use a proxy server as additional layer between the DNS server and the cluster that proxies request to port 80 to port 38080 on your nodes.



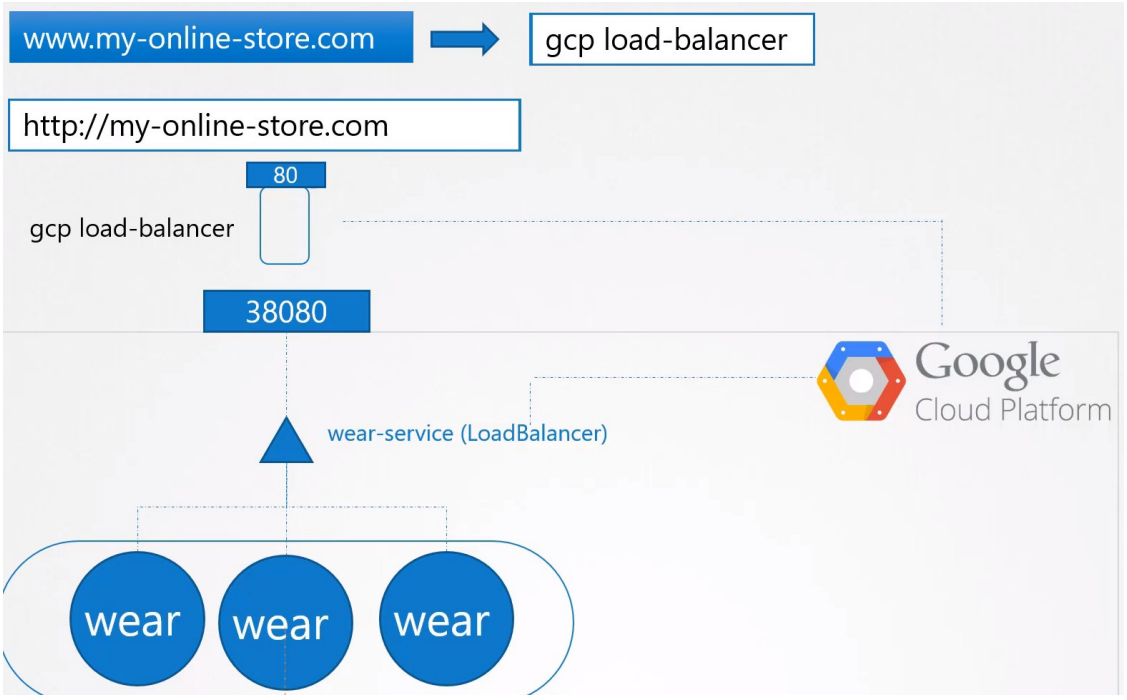
You then points your DNS to the proxy server, and users can now access my app by typing the domain name only: http://my-online-store.com

App hosted in cloud provider: Service type LoadBalancer

K8s does the same than it did for a NodePort, it provisions a high port for the service (38080 in this example) but in addition to that k8s also sends the request to the cloud provider, GCP in this example, to provision a Network LoadBalancer for this service.

On receiving the request GCP automatically deploy a LoadBalancer configured to route trafic to the service ports on all the Nodes and return it's information to k8s.

The LoadBalacer has an external IP that can be provided to users to acess the application, in this case we set the DNS to point to this IP and users access the application using the url http://my-online-store.com



App hosted in cloud provider: Several Loadbalancers

The business is growing and you have new services for the customers, for example a video streaming service.

You want your users to be able to access the video streaming service by going to: http://my-online-store.com/**watch**

You want to make available your old application accesible at http://my-online-store.com/**wear**

The developers developed the new video streaming application as a completely different app that has nothing to do with the existing one, however it is deployed in the same cluster as a separate Deployment.

You create a Service called *video-service* with type LoadBalancer.

K8s provisions the port 38282 in the Nodes and also provides a Network Load Balancer on the Cloud Provider.

The new load balancer has a new IP. Remember you must pay for each Load Balancer which will affect your bill.

How do you redirect traffic between both URL's? You will need another LoadBalancer or Proxy to redirect traffic based on the URL of both services, and every new service introduced you have to reconfigure the LB.

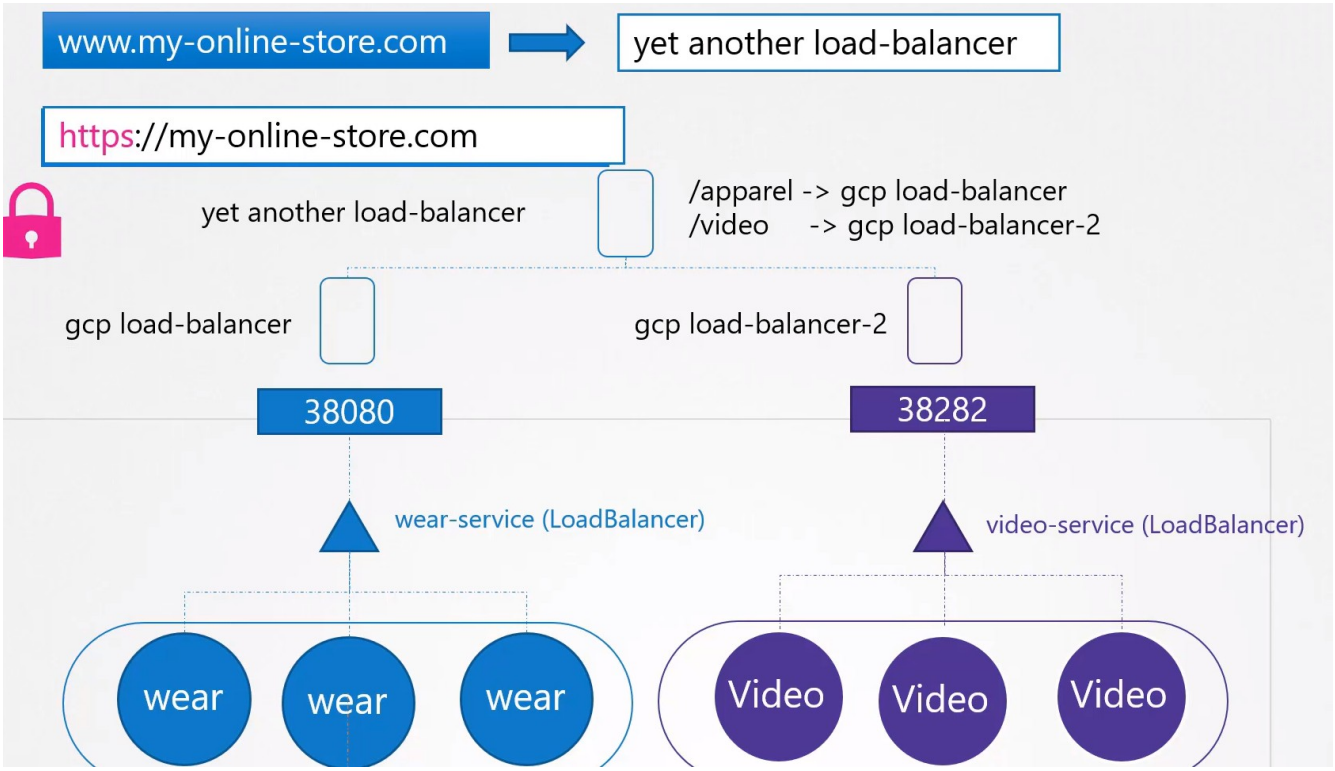
Finally you also need to enable **SSL** for the apps to allow the users to access the apps using https
You can configure that at different levels, either at the application level itself or at the Proxy/LB level.

You don't want your developers to implement it in the app as they would do it in different ways.

You want it to be configured at one place with minimal maintenance.

That's a lot of different configuration and all of this becomes difficult to manage when your app scales.

It requires to involve different individuals in different teams, you need to configure the FW rules for each new service and is expensive as well as for each service a new Cloud native LB needs to be provisioned.



Ingress:

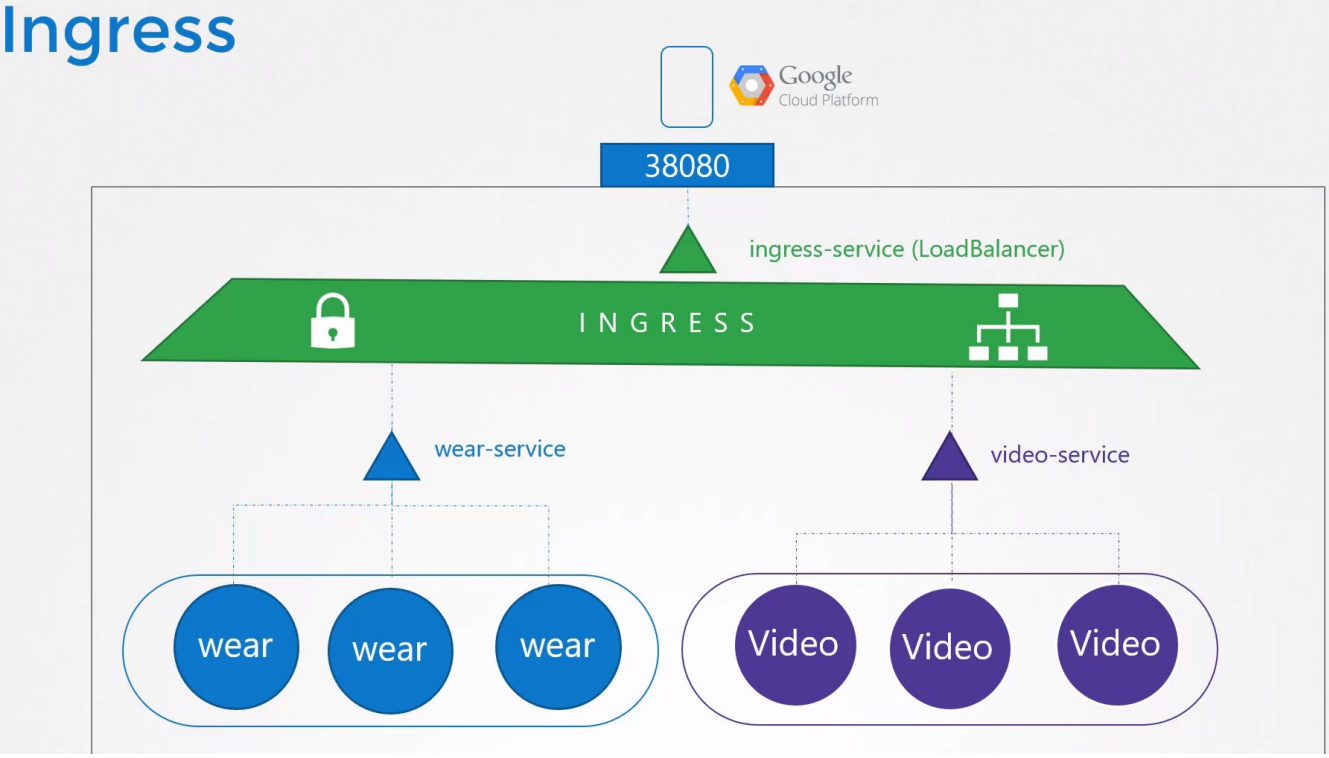
Ingress helps our users access your application using a single externally accessible URL that you can configure to route to different services within your cluster based on the URL path.

At the same time implement SSL security as well.

Think on Ingress as a layer 7 load balancer built in to the K8s cluster like can be configured using k8s primitives just like any other object in k8s.

You still need to expose it to make it accessible outside the cluster. You still need to publish as a NodePort or as cloud native LoadBalancer. But that is just a one time configuration.

You are going to perform all your load balancing, authentication, SSL and URL based routing configurations on the Ingress controller.

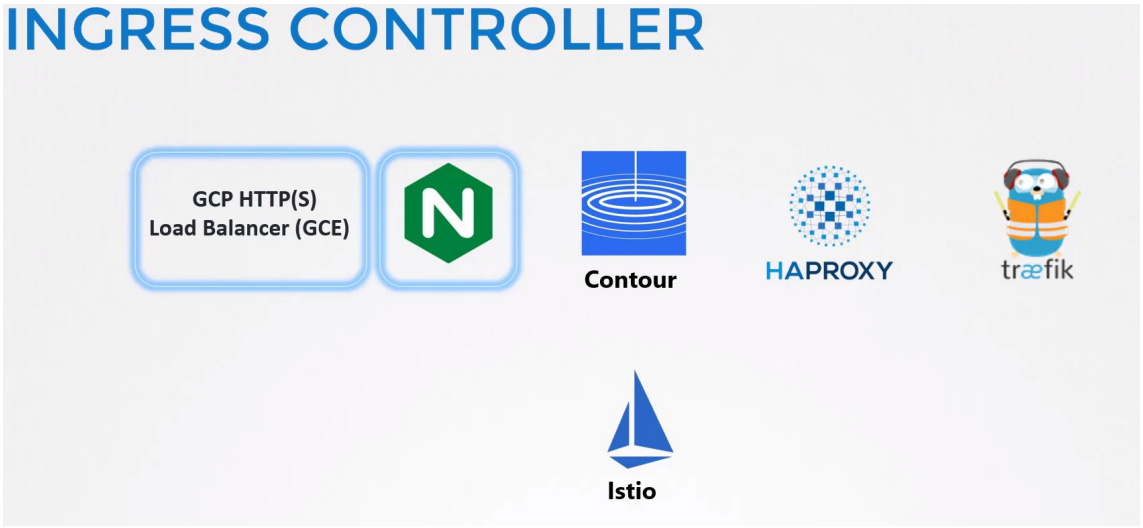


Ingress parts:

1. **Ingress Controller:** Supported solution deployed. K8s does NOT come with a built-in Ingress Controller by default. There are several IC solutions like Nginx, Haproxy, Istio, GCE (google), etc. GCE and Nginx are supported and maintained by the k8s project.

2. **Ingress Resources:** Are created using definition files like PODs, Deployment, Services, etc.

Ingress Controller (IC): We will use **Nginx** as an IC from now for this explanation.



These IC are not just another load balancer or nginx server, the load balancer components are just a part of it.

The IC have additional intelligence built into them to monitor the k8s cluster for new definitions or ingress resources and configure the Nginx server accordingly.

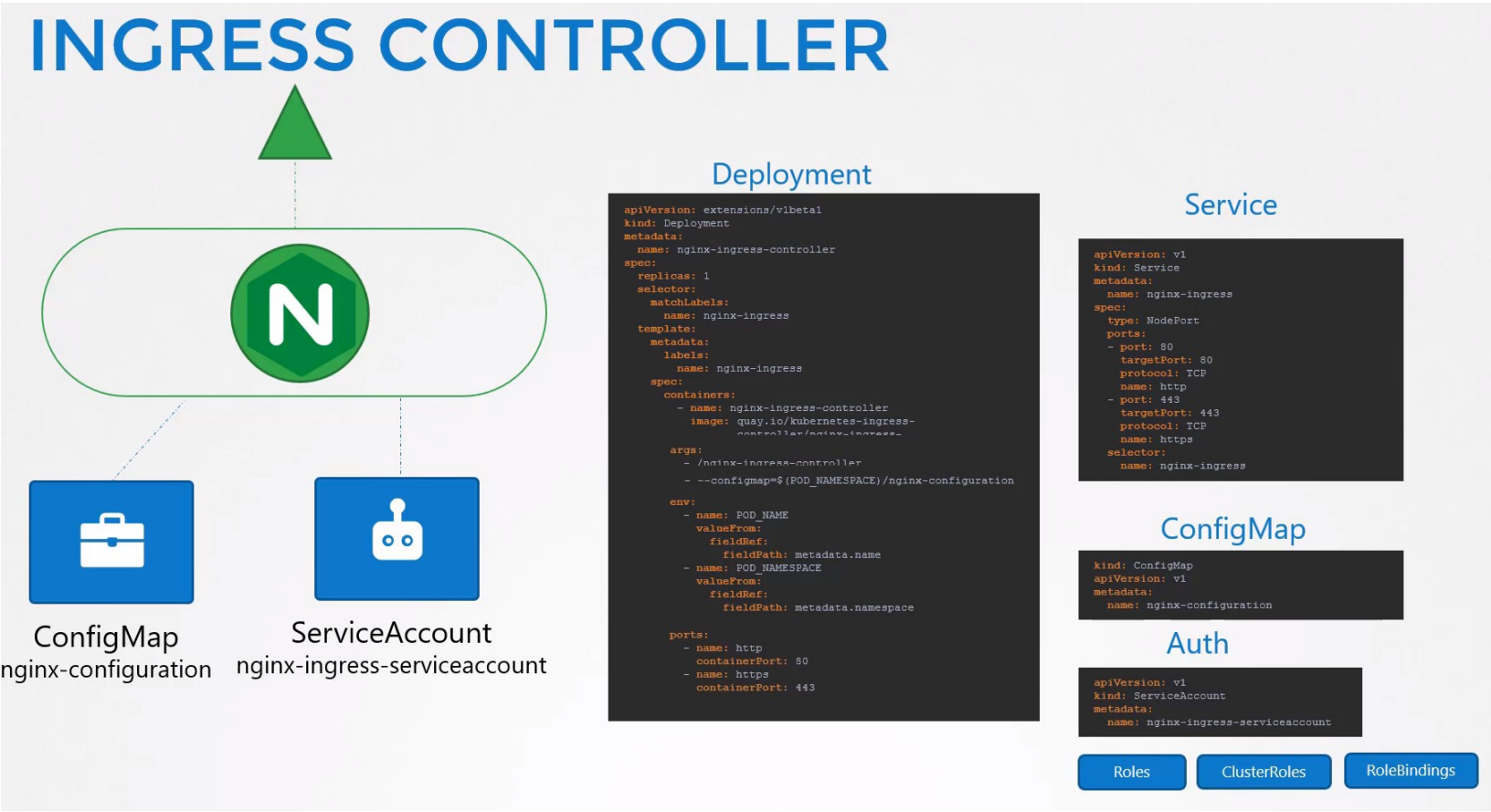
Deploy an Ingress Controller: An Nginx controller is deployed as just another Deployment in k8s but with more objects:

- **Deployment**: Which have to contain:
  - Deployment definition file named as nginx-ingress-controller.
  - 1 replica and a simple POD definition template
  - We will label it as: nginx-ingress
  - The Image will be an special build of Nginx build especifically to be used as an IC in k8s.
  - You must pass an args containing:
    - The nginx programs which are stored in /nginx-ingress-controller.
    - The config maps where the nginx configuration options are defined.
  - Environment variables containing the POD name and Namespace it is deployed to, Nginx service requires this to read the configuration data from within the POD.
  - The ports used by the IC, happens to be 80 and 443.

- **ConfigMaps:** Nginx has a set of configuration options such as:
  - The path to store the logs.
  - Keep-alive threshold.
  - SSL settings.
  - Session timeout,
  - etc.
    We will put all this into a ConfigMap in order to decuple this configuration data from the nginx controller image. It can have nothing configured yet, but we can add settings in the future.

- **Service:** To expose Ingress Controller to the externall world.
  - Type: NodePort
  - Nginx Label Selector: nginx-ingress → to link the Service to the Deployment.

- **ServiceAccount:** Nginx controller have additional intelligence built in to them to monitor the k8s cluster for Ingress resources and configure the underlining Nginx server when something is changed. We create a ServiceAccount with the correct Roles, Clusterroles and RoleBindings.



Ingress Resource: Set of rules and configurations applied on the Ingress Controller.

You can configure rules to simply forward incoming traffic to a single application or route traffic to different applications based on URL, so if one user goes to my-online-store.com/**wear** then route to one of the applications or if the user goes to the /watch URL then route to the video app, etc.

Or you could route users based on the domain name itself, for example if the user visits **wear**.my-online-store.com then route the user to the wear application or if he visits watch.my-online-store.com route the user to the video app.

Basic configuration:

Ingress-wear.yaml

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
 backend:
      serviceName: wear-service
      servicePort: 80
```

```
kubectl create -f Ingress-wear.yaml
ingress.extensions/ingress-wear created
```

```
kubectl get ingress
NAME           HOSTS     ADDRESS   PORTS
ingress-wear   *         80        2s
```

The traffic is redirected to the application Services.

The backend section defines where the traffic will be routed to.

If it's a single backend then you don't really have any rules, just specify the service name and port of the backed.

Rules: You use rules when you want to route trafic based on different conditions.

For example, you create the Rule 1 to handle the trafic that reaches your domain name, my-online-store.com

Then create Rule 2 when users reach your cluster using the domain name **wear.**my-online-store.com

Use Rule 3 to handle trafic to **watch**.my-online-store.com

And use Rule 4 to handle everything else.

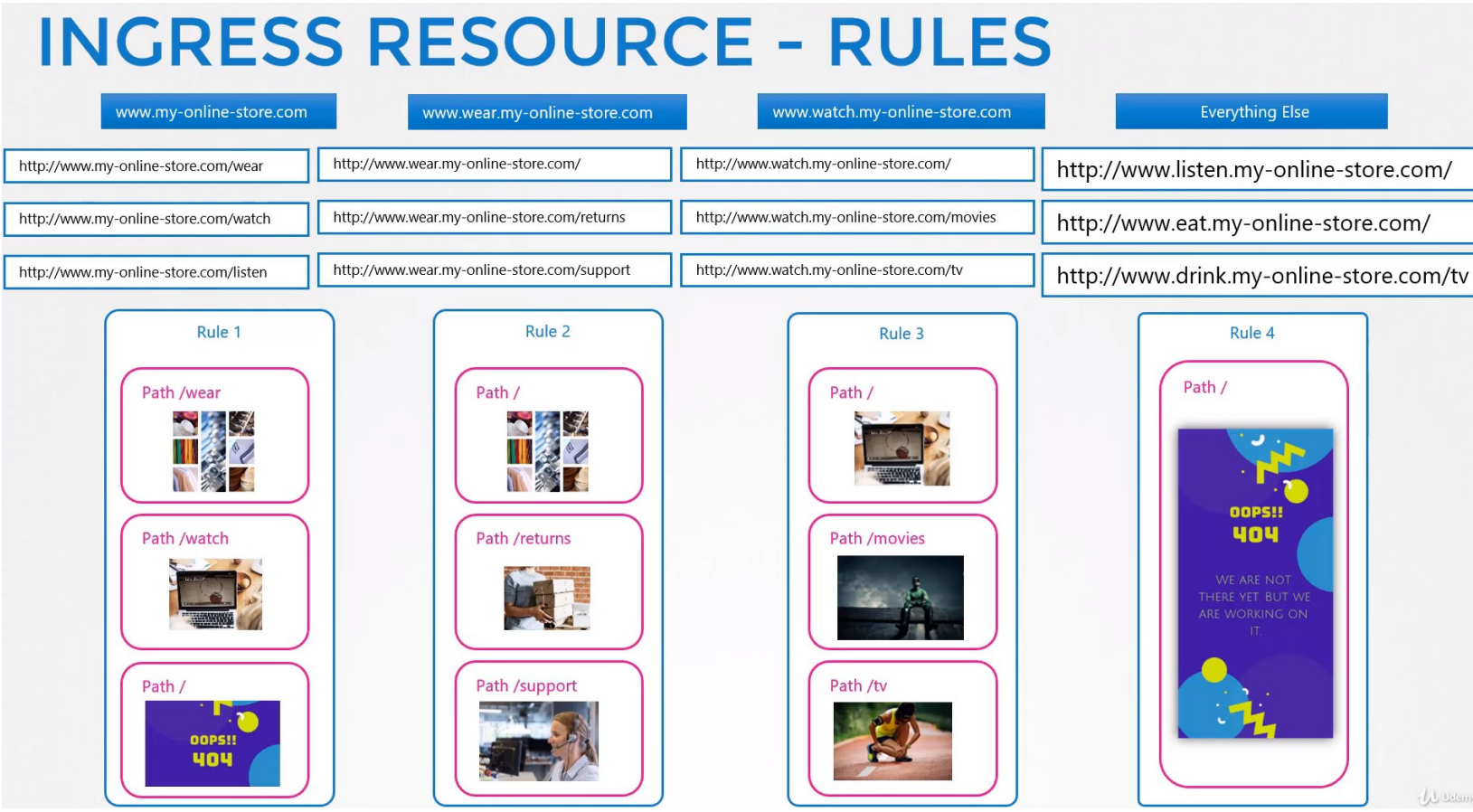
Within each Rule you can handle different paths, for example:

Rule 1 → my-online-store.com/wear
my-online-store.com/watch
my-online-store.com/anything_else → 404 error

Rule 2 → wear.my-online-store.com/
wear.my-online-store.com/returns
wear.my-online-store.com/support

Rule 3 → watch.my-online-store.com/
watch.my-online-store.com/movies
watch.my-online-store.com/tv

Rule 4 → Everything else redirects to 404
everything_else.my-online-store.com/
everything_else.my-online-store.com/fuck



Remember: You have Rules at the top for each Host or Domain name and within each Rule you have different paths to route trafic based in the URL.

How to configure Ingress resources:

One path for each URL → /wear and /watch

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

Basic Ingress to compare:

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

We will need to create 2 Services, wear-service and watch-service

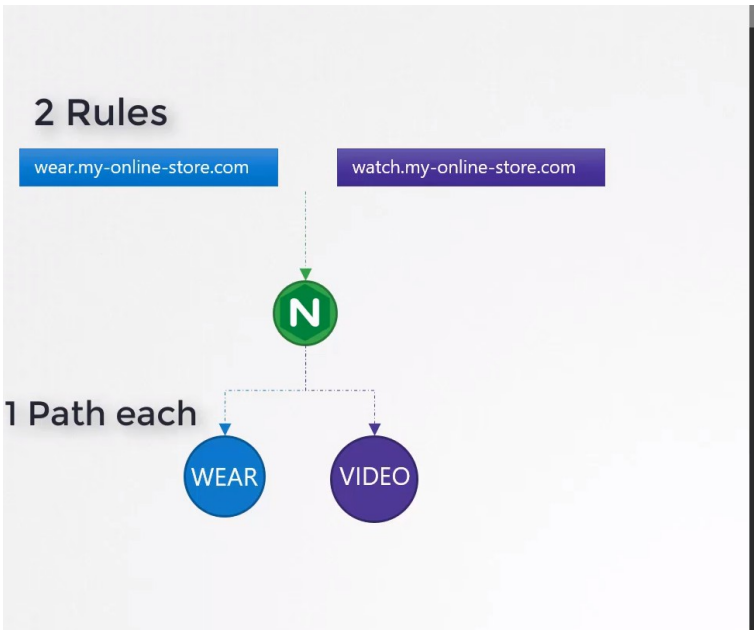
Watch the rules created:

```
kubectl describe ingress ingress-wear-watch
Name:             ingress-wear-watch
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path  Backends
  ----  ----  --------
  *
        /wear    wear-service:80 (<none>)
        /watch   watch-service:80 (<none>)
Annotations:
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  CREATE  14s   nginx-ingress-controller  Ingress default/ingress-wear-watch
```

Notice the Default backend: If a user tries to access a URL that does not match any of these rules then the user is directed to the Service specified as the Default backend, in this case the Service name is default-http-backend:80

You must remember to deploy such a service.

Using Domain names or Host names: (3rd type of configuration)



Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```

Compare the two configurations:
Splitting trafic by URL had just one rule and we split the trafic with two paths.

To split traffic by host name we use two rules and one path specification in each rule.

<u>- Splitting traffic by url:</u>
Have just une rule and 2 paths.
You can see it like this: path: */wear
So is looking for everything that has /wear at the end.

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

<u>- Splitting traffic by Domain/Hostname:</u>
We use two rules and one path specification on each rule.

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```

Summary:

Ingress Resources Rules: You have 3 options:

- **Path:** You provide a list of paths, like /testpath, each of wich has an associate backend defined as service.name, service.port and service.port.name. If no host is specified the rules apply to all the inbound traffic coming from the IP address. If there are also a Host specified both rules, path and host must match the content of an incoming request before the LB redirects the traffic to the referenced server

- **Host:** If a host is provided, for example foo.bar.com, the rules apply to that host.

- **Backend:** Combination of Service and port names.

Default Backend: Configured in the Ingress Controller. You use it to send there the traffic that doesn't match the rules.

Instead of rejecting everyting that is not matching the rules, it sends the no matched traffic to the default backend.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-resource-backend
spec:
  defaultBackend:
    resource:
      apiGroup: k8s.example.com
      kind: StorageBucket
      name: static-assets
  rules:
    - http:
        paths:
          - path: /icons
            pathType: ImplementationSpecific
            backend:
              resource:
                apiGroup: k8s.example.com
                kind: StorageBucket
                name: icon-assets
```

Design and Install a Kubernetes Cluster:

For testing environmets:

Kubeadm tool can provision a single node or a multinode cluster but you must provision the required host with the supported configuration yourselve.

The different between kubeadm and **minikube** is that minikube provisions the VM's by itselve.

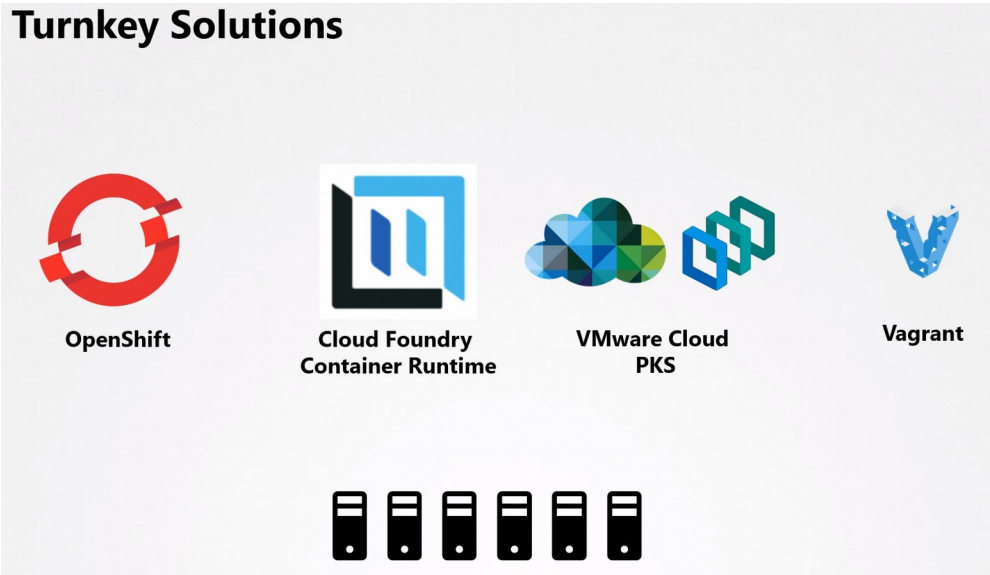
For production environments:

Turnkey Solutions: You provide and maintain the cluster to use it **privately** within your organization.

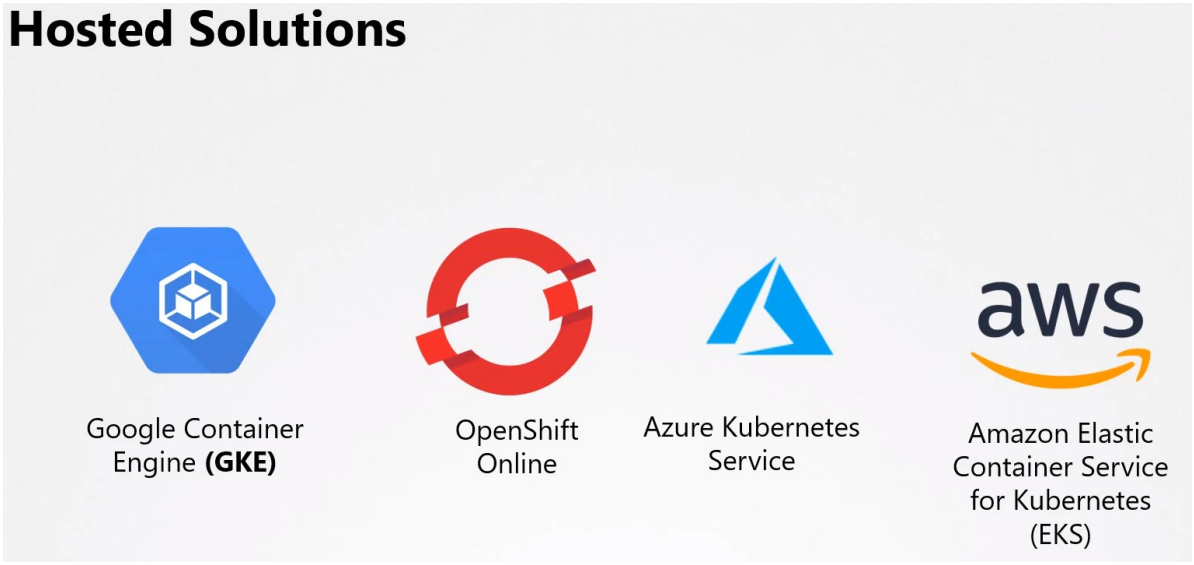
Hosted Solutions: A cloud provider of your election will take care of deploying and maintaining the VM's.

Turnkey Solutions	Hosted Solutions (Managed Solutions)
• You Provision VMs	• Kubernetes-As-A-Service
• You Configure VMs	• Provider provisions VMs
• You Use Scripts to Deploy Cluster	• Provider installs Kubernetes
• You Maintain VMs yourself	• Provider maintains VMs
• Eg: Kubernetes on AWS using KOPS	• Eg: Google Container Engine (GKE)

Turnkey solutions that helps you to deploy k8s cluster privately in your DC.
These are few of the many k8s certified solutions.



Hosted Solutions:



Configure HA:

If you have a single master node and it fails the apps in the workers will still run until new pods,etc, need to be scheduled,recreated, etc.

Since the kubeapi-server is not available you can not access the cluster to manange it through kubectl or through API.

This is why you have to consider multiple master nodes in a production environment.

HA is to have reduncancy to avoid single point of failure.

HA for different control plane components:

API-server:
Can be in **Active-Active** mode running in the master nodes. Api-server listens the kubectl calls in https://master1:6443, this is configured in the kube-config file.

But with two or more masters you can have a load balancer in the front of the master nodes that split traffic between the API-server, we point the kubectl utility to the load balancer.

Scheduler & Controller Manager: They watch the state of the cluster and takes actions. They run in an **Active-Standby mode.**

Example: Controller manager have controllers like the replication controller that are constantly watching the state of PODs and taking necessary actions. If multiple instances of this run in paralel we could create duplicated resources.

Who decides which node is active and pasive: This is achieved by a **Leader Election process.**

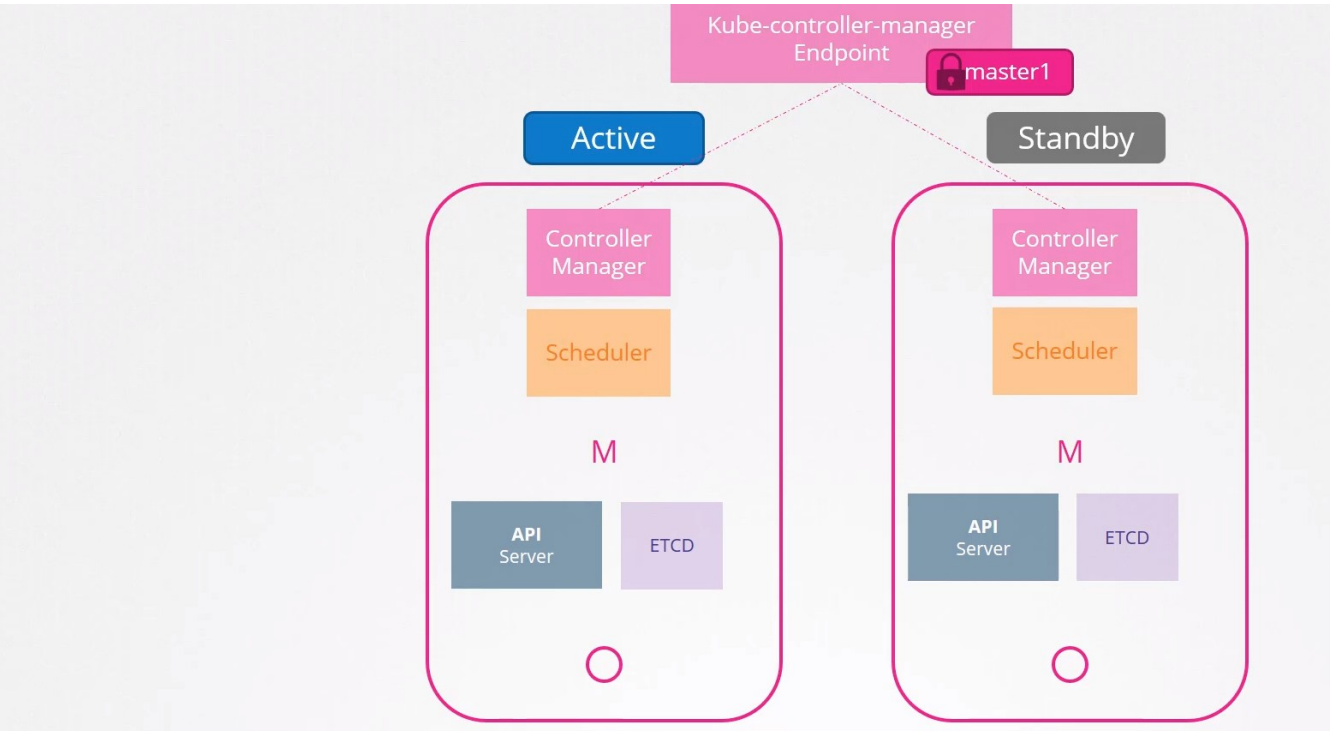
For example, the controller manager. When the cm is configured it must specify the leader elect option wich is by default set to true.

Then when the cm process starts it tries to gain a lease or a lock on an end point object named as kube-controller-manager endpoint.

Whichever process first updates the endpoint with this information gains the lease and becomes the active of the two, the other becames passive.

It will be the leader for 15 seconds by default, this is specified as the --leader-elecgt-lease-duration.

The lease is renewed every 10s. And all the nodes tries to be the leader every 2 seconds.




```
kube-controller-manager --leader-elect true  [other options]
                        --leader-elect-lease-duration 15s
                        --leader-elect-renew-deadline 10s
                        --leader-elect-retry-period 2s
```

The **scheduler** follows the same process.

ETCD:

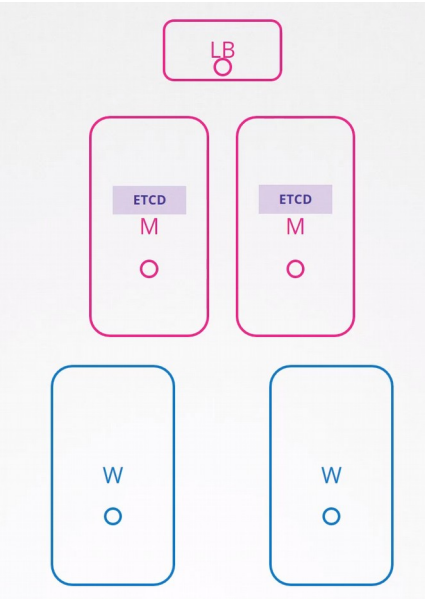
There are two topologies you can configure in k8s:

- **Stacked Topology:** ETCD is inside the master node with the other components
  - Easier to set up
  - Easier to manage
  - Fewer Servers
  - Risk during failures
- **External ETCD Topology:** ETCD have its own set of servers for HA
  - Less Risky
  - Harder to Setup
  - More Servers

Api-server is the only component talking to ETCD.
In the etcd configuration file /etc/systemd/system/kube-apiserver.service is specified which etcd servers are you using.

```
cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --etcd-servers=https://10.240.0.10:2379,https://10.240.0.11:2379
```

Summary:



For HA we will need a Load Balancere nodem more than 1 master nodes and more than 1 ETCD nodes if we want reduncancy.

ETCD in HA:

Quick recap of ETCD:

What is ETCD: Distributed reliable key-value store that is simple, secure and fast.

What is a key-value store: Stores the data in documents or pages (NoSQL) instead of tables (Relational,SQL).

Each individual gets a document and all the information regarding him is stored in that file which you can modify without affecting the rest of the documents.

You transact using data format like json or yaml

What is a distributed system: In ETCD. You have 3 etcd nodes for example.
You can Read in all of them but only can Write in one to avoid having different data in different nodes.

The etcd node leader node will be the only one receiving the Write requests and then will replicate it to the rest of the etcd nodes.

The leading election process is carried out internally by using **RAFT protocol** which is a Distributed Consensus.

How RAFT works: In a 3 node cluster. When the cluster is set up we have 3 nodes that NOT have a leader elected.

RAFT algorism have random timers for initiating request. The first one to finish the timer sends out a request to the other nodes requesting permission to be the leader.

The other nodes receiving the request respond with their vote and the node assumes the leading role.

The leader send out notification at regular intervales to other nodes informing that is alive and continues asuming the role of the leader.

In case the leader is not sending the keep alive notifications, the rest of the nodes initiates the leading election process among themselves and a new leader is identified.

Now a Write commes in and is replicated to the other etcd nodes in the cluster.

A write is not considered completed until it is replicated to the rest of the nodes.

What happen if one node is down: A write request is considered to be complete if it can be written on the majority of the nodes of the cluster.

If you have 3 nodes and 1 is down there are still two nodes up, and 2 is the majority/quorum of 3 so the write will be replicated to the other node and will be considered as completed.

If the 3rd node comes online, then the data is copied to that as well.

Quorum (majority of nodes): Minimum number of nodes for the cluster to function properly or make a succesfully Write.

For any given number of nodes Quorum is the total number of nodes divided by 2 plus 1:
N/2 + 1

In case of 3 nodes we know that 2 nodes is the quorum → 3 = 3/2 + 1 = 2.5 ~= 2

In case of 2 nodes the quorum is 2 → 2/2 + 1 = 2
So the if one node fails the Write won't be processed because we have lost the quorum. Therefore to have two nodes is the same than having 1.

That's why is recommended on having at least 3 etcd instances for HA. That way it offers a fault tolerance of at least one node.

If you lose one you can still have quorum and the cluster will continue to function.

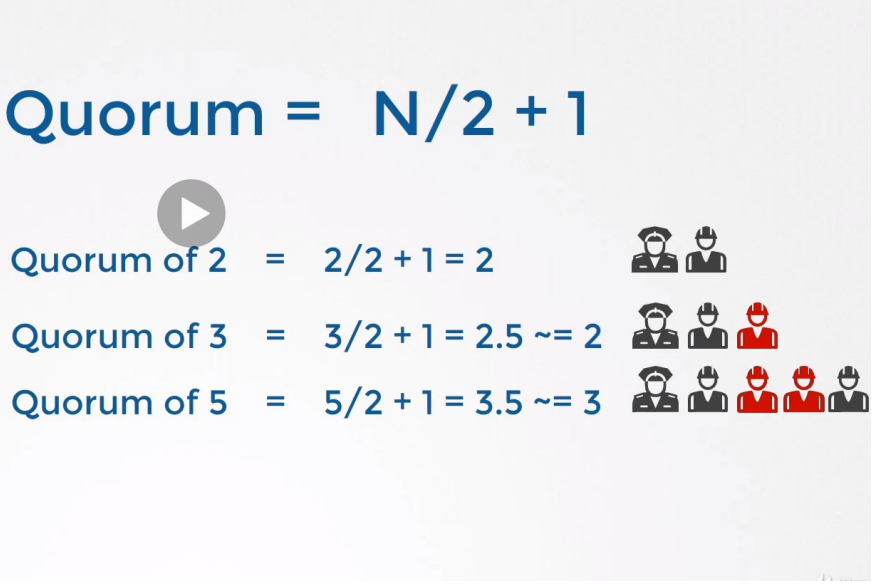
Fault Tolerance: Instances + Quorum gives you the fault tolerance.

The number of nodes you can afford to lose while keeping the cluster alive.

At least you have to have 3 Nodes for HA. Use always Odd numbers (3,5,7), having 5 nodes is prefered than having 6 in case you segment the network for example.

5 Nodes is enough to have full tolerance.

Instances	Quorum	Fault Tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3



Instances	Quorum	Fault Tolerance
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

3 nodes (instances) is ok, but if you want a higher level of fault tolerance 5 is better. More than 5 is unnecessary.

Install ETCD on a server:

```
wget -q --https-only \
    "https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"

tar -xvf etcd-v3.3.9-linux-amd64.tar.gz

mv etcd-v3.3.9-linux-amd64/etcd* /usr/local/bin/

mkdir -p /etc/etcd /var/lib/etcd

cp ca.pem kubernetes-key.pem kubernetes.pem /etc/etcd/
```

Pass to etcd.service the etcd nodes information for HA:

etcd.service

```
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
  --advertise-client-urls https://${INTERNAL_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster peer-1=https://${PEER1_IP}:2380,peer-2=https://${PEER2_IP}:2380 \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
```

Store and retrieve data: Once ETCD is installed use the ETCDCTL to operate it.
ETCDCTL have two versions, v2 and v3. One each of them have different commands, v2 is default but we will use v3

To use v3 → export ETCDCTL_API=3

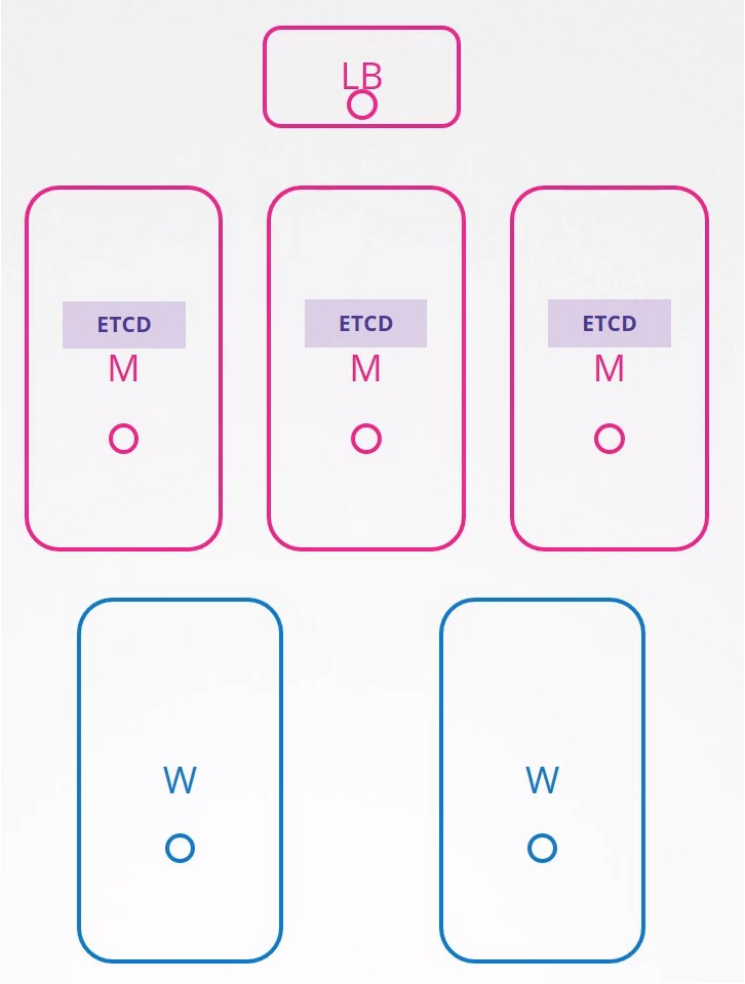
To write → etcdctl put name john

To retrieve data → etcdctl get name

To get all keys → etcdctl get / --prefix --keys-only

Summary:

3 master node with Stacked ETCD topology:

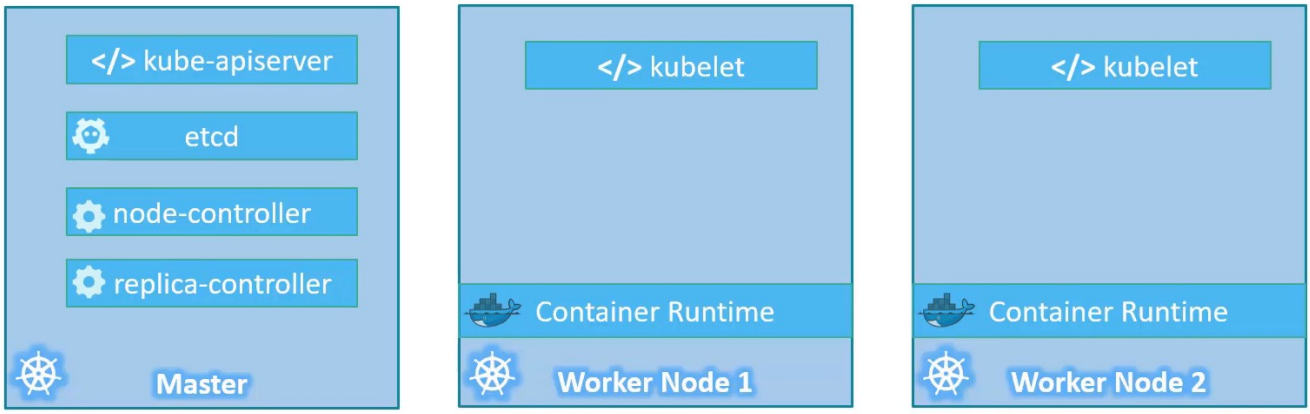


Kubeadm:

Can be used to bootstrap a k8s cluster.

Kubeadm tool helps us to set up a multinode cluster following k8s best practices.

K8s cluster have different components:



Installing all of these components on all the nodes, configuring them,point them to each other and setting up ssl certificates is a tedious task. Kubeadm helps us by taking care of all of those tasks.

Steps to set up a k8s cluster using kubeadm:

1. Create the servers and designate one node as master and the other as worker nodes.
2. To install a container run time on all the nodes.
3. Install kubeadmin tool on all the nodes.
4. Initialize the master server. During this process all the required components are installed and configured on the master server.
5. POD Network. K8s requires a special networking solution between the master and the worker nodes (weave, flannel, etc)
6. Join the worker nodes to the master node.

Full manual

Tips:

Get the join command:

```
kubeadm token create --print-join-command
kubeadm token create --help
```

Install CNI weave: It's only necessary on the Master node

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

Autofill:

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

Troubleshooting:

Application Failure:

Always do a map of all the objects related and go one by one to find out what is failing.

What we have: Two tier application with a web and a db server.

What is failing: Users report issues when accessing the application.

What to check:

1. Check the application front end by accessing the app via web. Check if the webserver is accessinble by using curl to the app.

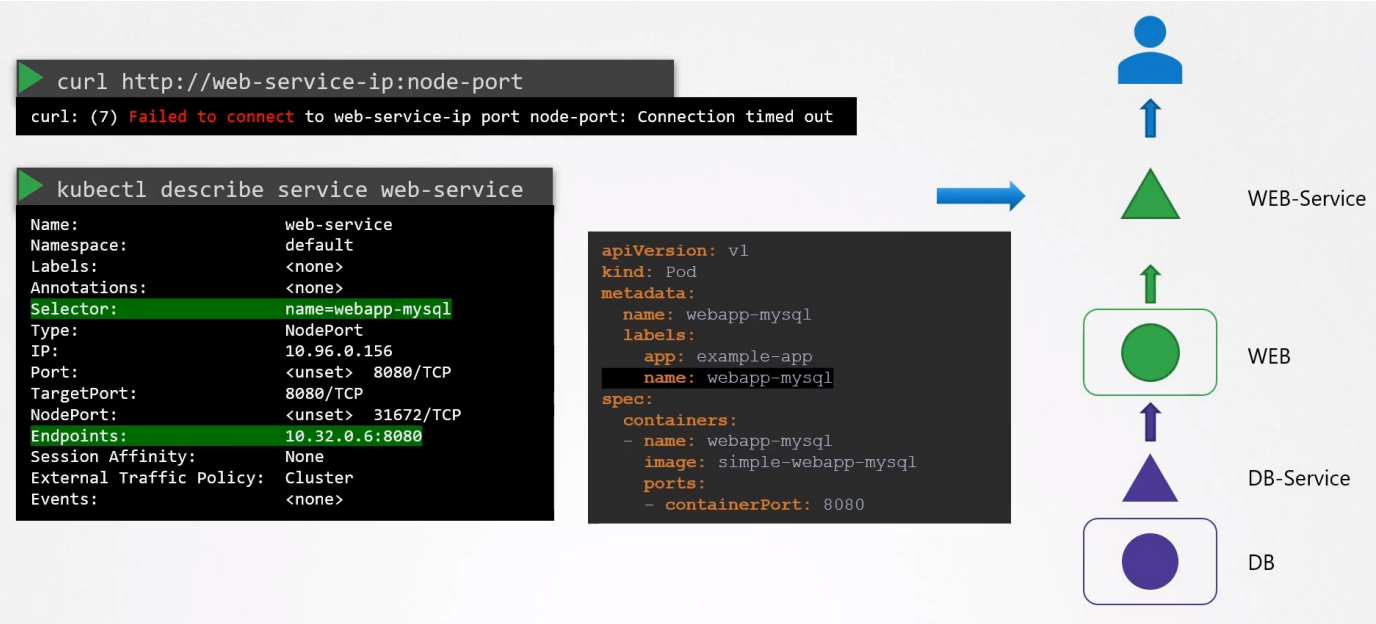
2. Check the WEB-Service: Check if have discovered the endpoint. Compare if the selectors are the same in the Service and the Pod and make sure they match.

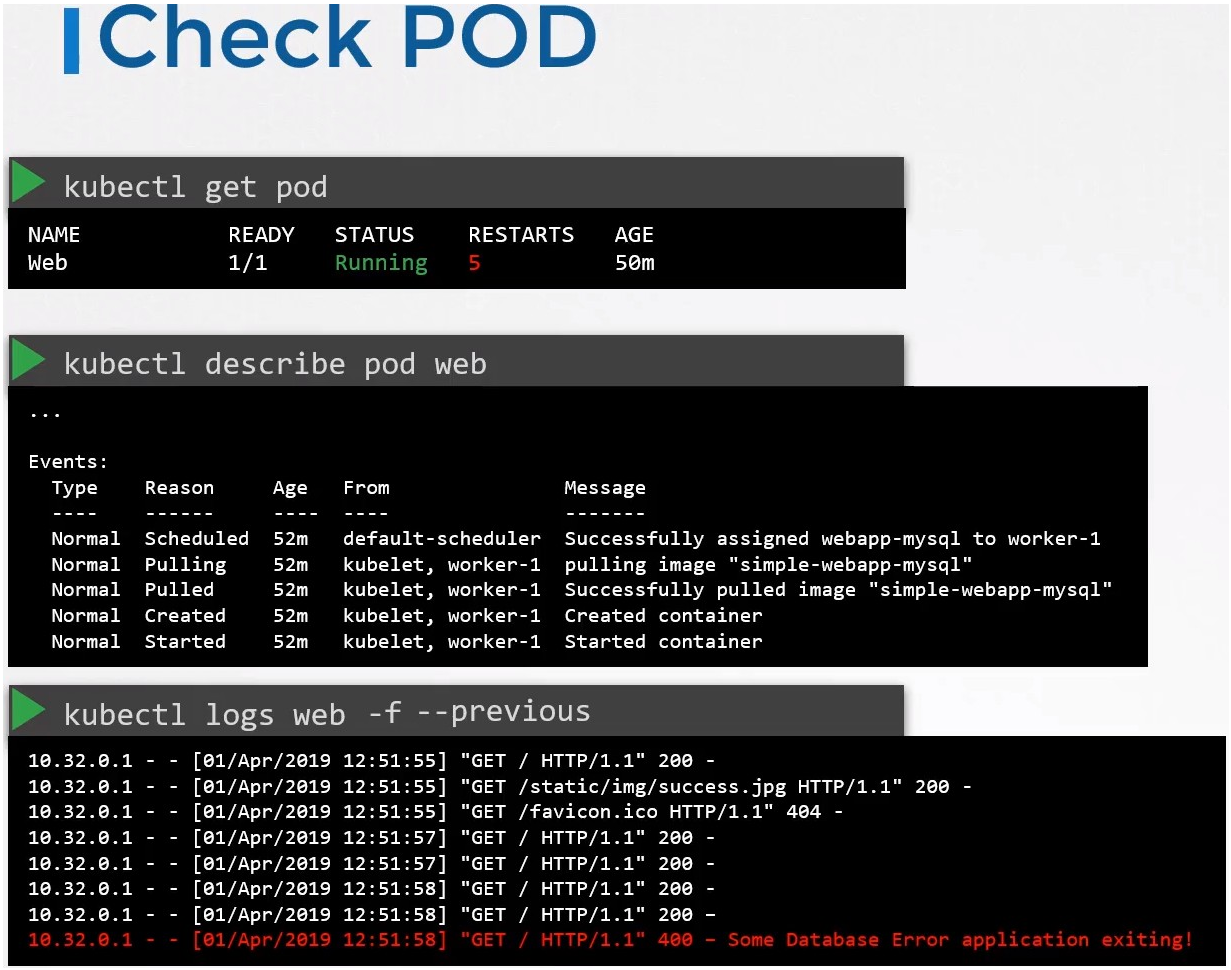
3. Check the Pod: Make sure is Running. The status and the restarts will give you an idea if the app in the Pod is running or is getting restarted. Check the events of the Pod using *describe* and *logs* commands.
   1. kubectl get pod
   2. kubectl describe pod nginx
   3. kubectl logs nginx -f --previous → previous to see the logs of the previous Pod.

4. Check the DB-Service: Same checks than in the step 2.

5. Check the DB Pod: Check the logs of the pod and look for any error in the DB.

Troubleshooting apps documentation





Commands:

Get all objects of one namespace:
kubectl -n <namespace> get all
kubectl -n alpha get all

Get endpoints:
kubectl -n <namespace> get endpoints
kubectl -n delta get endpoints
k -n delta get ep

Control Plane Failure:k get -n kube-system all

Check the status of the nodes:
k get nodes

Check status of the pods on the cluster:
k get pods
k get pods -n kube-system → if the control plane services were deployed with kubeadm or similar

Check the control plane components: If were deployed as services (not using kubeadm)

On the Master node:
service kube-apiserver status

service kube-controller-manager status

service kube-scheduler status

On the worker node:
service kubelet status

service kube-proxy status

Check logs for control plane controller:
k logs kube-apiserver-master -n kube-system

journalctl -u kube-apiserver

Worker Node Failure:

Check the status of the nodes:
k get nodes

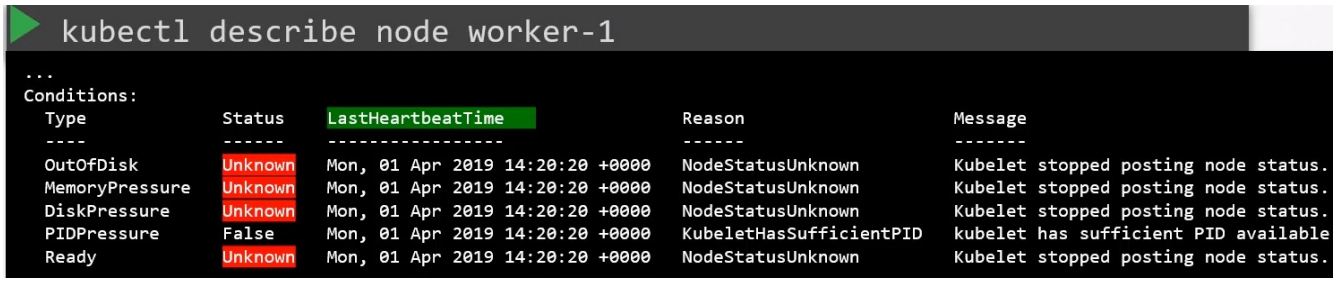
Check node description: If they are not ready
k describe node worker1

```
kubectl describe node worker-1
...
Conditions:
  Type             Status  LastHeartbeatTime                 Reason                       Message
  ----             ------  -----------------                 ------                       -------
  OutOfDisk        False   Mon, 01 Apr 2019 14:30:33 +0000   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   Mon, 01 Apr 2019 14:30:33 +0000   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Mon, 01 Apr 2019 14:30:33 +0000   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Mon, 01 Apr 2019 14:30:33 +0000   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Mon, 01 Apr 2019 14:30:33 +0000   KubeletReady                 kubelet is posting ready status. AppArmor enabled
```

Conditions types: When a node ran out of disk space, memory, disk preasure or PID the flag is set to True, if all is fine the Ready flag is True.

k get -n kube-system all

Possible Node Down: When is not communicating with the cluster. The status will be Unknown



Check if the node is crushed, if so bring it back up.

Top: Check the cpu utilization, roge programs, etc.

Disk space: df -h

Free -m: Check the memory and the swap

Kubelet Status:
service kubelet status

Kubelet Logs:
sudo journalctl -u kubelet

Kubelet configuration:
/etc/systemd/system/kubelet.service.d/10-kubeadm.conf

/var/lib/kubelet/config.yaml

/etc/kubernetes/kubelet.conf

Check Cluster info: In the afected node
kubectl cluster-info

kubectl cluster-info dump

Kubelet SSL Certificates: Ensure are not expired, are part of the correct group and were issued by the right CA.

openssl x509 -in /var/lib/kubelet/worker-1.crt -text

```
openssl x509 -in /var/lib/kubelet/worker-1.crt  -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            ff:e0:23:9d:fc:78:03:35
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = KUBERNETES-CA
        Validity
            Not Before: Mar 20 08:09:29 2019 GMT
            Not After : Apr 19 08:09:29 2019 GMT
        Subject: CN = system:node:worker-1, O = system:nodes
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:b4:28:0c:60:71:41:06:14:46:d9:97:58:2d:fe:
                    a9:c7:6d:51:cd:1c:98:b9:5e:e6:e4:02:d3:e3:71:
                    58:a1:60:fe:cb:e7:9b:4b:86:04:67:b5:4f:da:d6:
                    6c:08:3f:57:e9:70:59:57:48:6a:ce:e5:d4:f3:6e:
                    b2:fa:8a:18:7e:21:60:35:8f:44:f7:a9:39:57:16:
                    4f:4e:1e:b1:a3:77:32:c2:ef:d1:38:b4:82:20:8f:
                    11:0e:79:c4:d1:9b:f6:82:c4:08:84:84:68:d5:c3:
                    e2:15:a0:ce:23:3c:8d:9c:b8:dd:fc:3a:cd:42:ae:
                    5e:1b:80:2d:1b:e5:5d:1b:c1:fb:be:a3:9e:82:ff:
                    a1:27:c8:b6:0f:3c:cb:11:f9:1a:9b:d2:39:92:0e:
                    47:45:b8:8f:98:13:c6:4d:6a:18:75:a4:01:6f:73:
                    f6:f8:7f:eb:5d:59:94:46:d8:da:37:75:cf:27:0b:
                    39:7f:48:20:c5:fd:c7:a7:ce:22:9a:33:4a:30:1d:
                    95:ef:00:bd:fe:47:22:42:44:99:77:5a:c4:97:bb:
                    37:93:7c:33:64:f4:b8:3a:53:8c:f4:10:db:7f:5f:
                    2b:89:18:d6:0e:68:51:34:29:b1:f1:61:6b:4b:c6:
```

Network Failure:

k get -n kube-system all

Check if the CNI is up and running: If is not installed, do it:
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')"

Check endpoints:
k get endpoints -A

Check Daemonsets:
k get ds / k describe ds ...

Check ConfigMaps: To see if the path in the daemonsets are the same than in the configmaps

Check DNS:
Have the DNS and endpoint? kubectl -n kube-system get ep kube-dns (it should have more than 1 IP)
kube-system kube-dns 10.32.0.2:53,10.32.0.3:53,10.32.0.2:9153 + 3 more... 10m

If not endpoint: Check the service and look for the labels selectors:
k describe -n kube-system svc kube-dns