Kubernetes

Contents

[Core Concepts 2](#_Toc55663747)

[Kubernetes Architecture 2](#_Toc55663748)

[ETCD 2](#_Toc55663749)

[Kube-API Server 4](#_Toc55663750)

[Controller Manager 5](#_Toc55663751)

[Kube Scheduler 6](#_Toc55663752)

[Kubelet 7](#_Toc55663753)

[Kube Proxy 7](#_Toc55663754)

[PODs 8](#_Toc55663755)

[ReplicaSets 8](#_Toc55663756)

[Deployments 9](#_Toc55663757)

[Namespaces 11](#_Toc55663758)

[Services 12](#_Toc55663759)

[Kubernetes Imperative and Declarative 15](#_Toc55663760)

[Scheduling 17](#_Toc55663761)

[Manually Scheduling 17](#_Toc55663762)

[No Scheduler 18](#_Toc55663763)

[Labels and Selectors 18](#_Toc55663764)

[Annotations 19](#_Toc55663765)

[Taints and Tolerations 19](#_Toc55663766)

[Logging & Monitoring 21](#_Toc55663767)

[Application Lifecycle Management 21](#_Toc55663768)

[Cluster Maintenance 21](#_Toc55663769)

[Security 21](#_Toc55663770)

[Storage 21](#_Toc55663771)

[Networking 21](#_Toc55663772)

[Design and Install a Kubernetes Cluster 21](#_Toc55663773)

[Install “Kubernetes the kubeadm way” 21](#_Toc55663774)

[End to End Tests on a Kubernetes Cluster 21](#_Toc55663775)

[Troubleshooting 21](#_Toc55663776)

[Other Topics 21](#_Toc55663777)

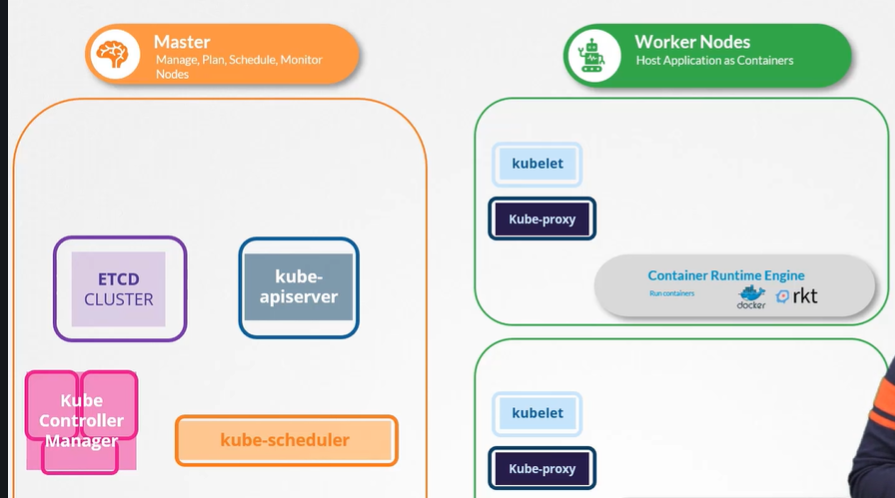
[Lightning Labs 21](#_Toc55663778)

[Mock Exams 21](#_Toc55663779)

# Core Concepts

## Kubernetes Architecture

Two types of Nodes:

1. **Worker Nodes:** Host Application as Containers
   1. **Container Runtime Engine**
   2. **kubelet**
   3. **kube-proxy**
2. **Master Nodes:** Manage, Plan, Schedule, Monitor Nodes
   1. **ETCD Cluster**
   2. **kube-scheduler**
   3. **Controller-Managers**
   4. **kube-apiserver**

## ETCD

ETCD is a distributed reliable **key-value** store that is simple, secure & fast.

#### Install ETCD

1. Download Binaries
2. Extract it
3. Run ETCD Service (Port 2379)

#### Operate ETCD

.etcdctl set key1 value1

This command sets an entry in ETCD Store.

.etcdctl get key1

This command gets the entry under `key1` from ETCD Store.

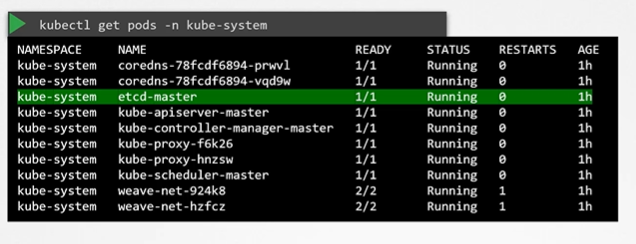
.etcdctl

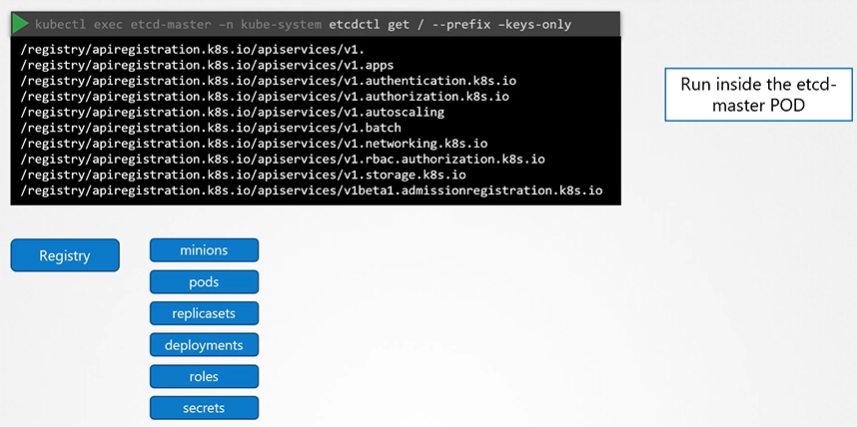
This command is the .etcdctl help command.

#### ETCD Cluster

It records every detail of Kubernetes entity like:

1. Nodes
2. PODs
3. Configs
4. Secrets
5. Accounts
6. Roles
7. Bindings
8. Others





(Optional) Additional information about ETCDCTL Utility  
  
ETCDCTL is the CLI tool used to interact with ETCD.  
  
ETCDCTL can interact with ETCD Server using 2 API versions - Version 2 and Version 3.  By default its set to use Version 2. Each version has different sets of commands.

For example, ETCDCTL version 2 supports the following commands:

1. etcdctl backup
2. etcdctl cluster-health
3. etcdctl mk
4. etcdctl mkdir
5. etcdctl set

Whereas the commands are different in version 3

1. etcdctl snapshot save
2. etcdctl endpoint health
3. etcdctl get
4. etcdctl put

To set the right version of API set the environment variable ETCDCTL\_API command

export ETCDCTL\_API=3

When API version is not set, it is assumed to be set to version 2. And version 3 commands listed above don't work. When API version is set to version 3, version 2 commands listed above don't work.

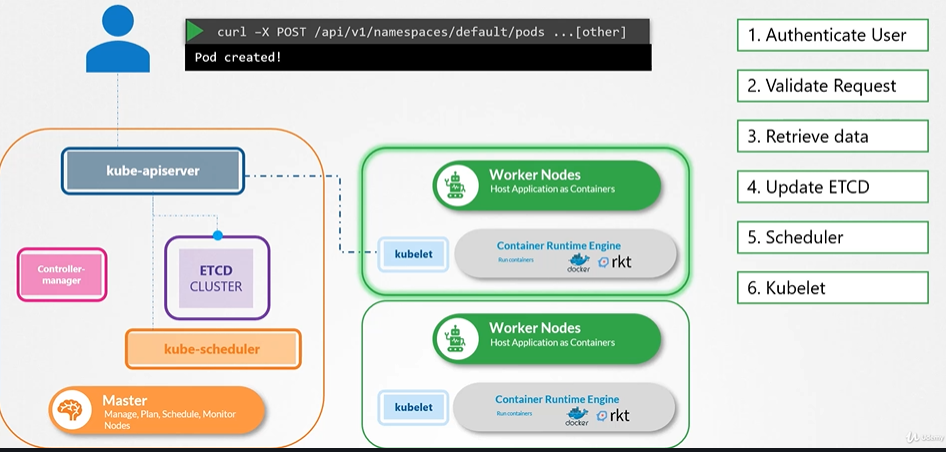
Apart from that, you must also specify path to certificate files so that ETCDCTL can authenticate to the ETCD API Server. The certificate files are available in the etcd-master at the following path. We discuss more about certificates in the security section of this course. So don't worry if this looks complex:

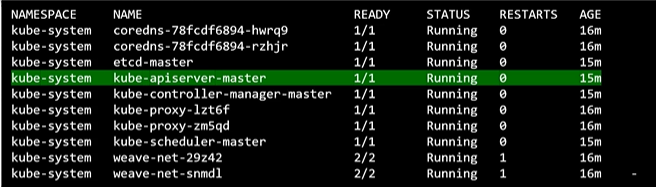
1. --cacert /etc/kubernetes/pki/etcd/ca.crt
2. --cert /etc/kubernetes/pki/etcd/server.crt
3. --key /etc/kubernetes/pki/etcd/server.key

So, for the commands I showed in the previous video to work you must specify the ETCDCTL API version and path to certificate files. Below is the final form:

## Kube-API Server

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances.

  
Only kube-apiserver talk to ETCD cluster, rest of the control-plane components talk to ETCD cluster via kube-apiserver.

kubectl get pods -n kube-system

This command gives us all control-plane components running in kube-system namespace.

To view the manifest of kube-apiserver(kubeadm setup): cat /etc/kubernetes/manifests/kube-apiserver.yaml

To view options of kube-apiserver(non-kubeadm setup):   
cat /etc/system/system/kube-apiserver.service  
ps -aux | grep kube-apiserver

## Controller Manager

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

1. **Node Controller**: Responsible for noticing and responding when nodes go down.
2. **Replication controller**: Responsible for maintaining the correct number of pods for every replication controller object in the system.
3. **Endpoints controller**: Populates the Endpoints object (that is, joins Services & Pods).
4. **Service Account & Token controllers**: Create default accounts and API access tokens for new namespaces.

* Watch Status
* Remediate Situation
* Node Monitor Period = 5s
* Node Monitor Grace Period = 40s
* POD Eviction Timeout = 5m

#### View kube-controller-manager – kubeadm

kubectl get pods -n kube-system

To view the manifest of kube-controller-manager (kubeadm setup): cat /etc/kubernetes/manifests/kube-controller-manager.yaml

To view options of kube-controller-manager (non-kubeadm setup):   
cat /etc/system/system/kube-controller-manager.service  
ps -aux | grep kube-controller-manager

## Kube Scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

* Filter Nodes
* Rank Nodes

Factors taken into account for scheduling decisions include:

1. Individual and collective resource requirements
2. Hardware/software/policy constraints
3. Affinity and anti-affinity specifications
4. Data Locality
5. Inter-workload interference
6. Deadlines

#### View kube-scheduler – kubeadm

kubectl get pods -n kube-system

To view the manifest of kube-scheduler (kubeadm setup): cat /etc/kubernetes/manifests/kube- scheduler.yaml

To view options of kube-scheduler (non-kubeadm setup):   
cat /etc/system/system/kube- scheduler.service  
ps -aux | grep kube- scheduler

## Kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Kubeadm does not deploy Kubelets.

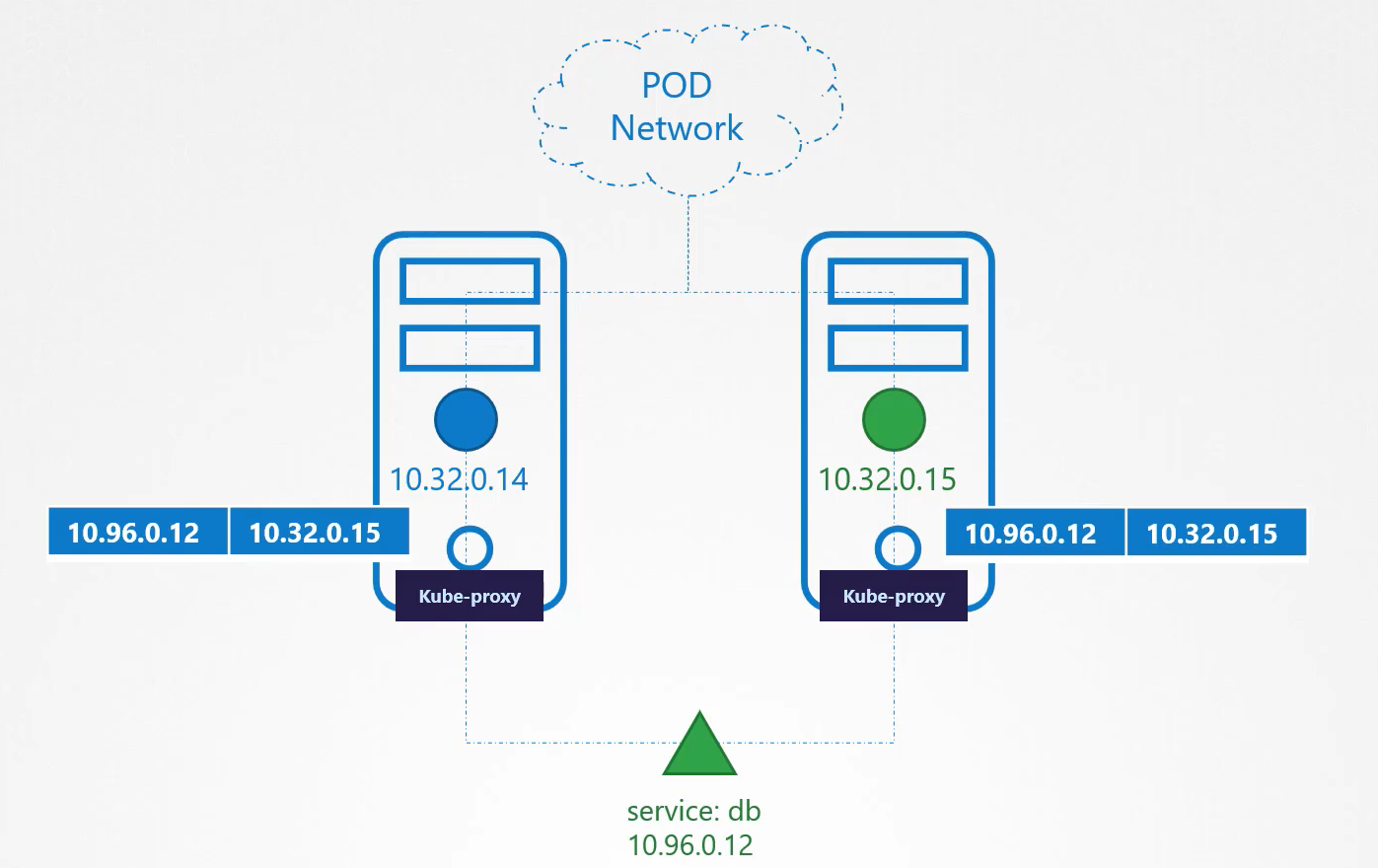
ps -aux | grep kubelet

## Kube Proxy

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept.

kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.



ps -aux | grep kube-proxy

### PODs

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

A Pod (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers.

A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled.

#### Using PODs

Usually Pods are not required to be created directly, even singleton Pods.

Instead, create them using workload resources such as Deployment or Job.

*If your Pods need to track state, consider the StatefulSet resource.*

Pods in a Kubernetes cluster are used in two main ways:

1. **Pods that run a single container**
2. **Pods that run multiple containers that need to work together:** A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. (**Multi-container PODs**)

#### Pod Definition YAML

apiVersion: v1  # version of Kubernetes API  
kind: Pod       # kind of Kubernetes object to be created   
metadata:       # data about the Kubernetes object `kind`  
  name: myapp-pod  
  labels:  
    app: myapp  
    type: front-end  
spec:           # specification about Kubernetes object `kind`  
  containers:  
    - name: nginx-container  
      image: nginx

kubectl pod-creation command (using yaml): kubectl create -f pod-definition.yml  
kubectl pod-creation command: kubectl run nginx --image=nginx  
kubectl pod-get command: kubectl get pods  
kubectl pod-deletion command: kubectl delete pod nginx

To get information about all the pods: kubectl get pods  
To describe all the creation information of a pod: kubectl describe pod myapp-pod

### ReplicaSets

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

* A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria.
* A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.
* A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features.

|  |  |
| --- | --- |
| ReplicaSets | ReplicationController |
| 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: 4   selector:      matchLabels:       type: front-end | apiVersion: v1 kind: ReplicationController metadata:   name: myapp   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: 4 |

Commands:

1. To create a replicaset: kubectl create -f replicaset-definition.yml
2. To get all replicasets: kubectl get replicaset

#### Labels and Selectors

Using labels and selectors, the ReplicaSet knows which pods to monitor in a cluster of pods.

#### Scaling Replicas

1. Update the yaml and run the command: kubectl replace -f replicaset-definition.yml
2. We can also use the command: kubectl scale --replicas=6 -f replicaset-definition.yml,   
   or, kubectl scale --replicas=6 -f replicaset myapp-replicaset but this does not change **replicaset-definition.yml.**

### Deployments

A Deployment provides declarative updates for Pods and ReplicaSets.



The contents of **deployment-definition.yml** is as follows:

apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: myapp-deployment  
  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: 4  
  selector:   
    matchLabels:  
      type: front-end

Commands:

1. To create a deployment: kubectl create -f deployment-definition.yml
2. To get deployments: kubectl get deployments

BONUS TIP:

1. Create a NGINX Pod: kubectl run nginx --image=nginx
2. Generate POD Manifest YAML file (-o yaml). Don't create it (--dry-run): kubectl run nginx --image=nginx --dry-run=client -o yaml
3. Create a deployment: kubectl create deployment --image=nginx nginx
4. Generate Deployment YAML file (-o yaml). Don't create it (--dry-run): kubectl create deployment --image=nginx nginx --dry-run=client -o yaml
5. 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=client -o yaml > nginx-deployment.yaml
6. Save it to a file, make necessary changes to the file (for example, adding more replicas) and then create the deployment.

### Namespaces

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

* Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.
* Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces cannot be nested inside one another and each Kubernetes resource can only be in one namespace.
* Namespaces are a way to divide cluster resources between multiple users (via resource quota).



The namespace-dev.yml is follows:

apiVersion: v1  
kind: Namespace  
metadata:   
  name: dev

Commands:

1. To create a pod in a specific namespace: kubectl create -f pod-definition.yml --namespace=dev
2. To create a namespace (with manifest): kubectl create -f namespace-dev.yml
3. To create a namespace (without manifest): kubectl create namespace dev
4. To set the default namespace: kubectl config set-context $(kubectl config current-context) --namespace=dev
5. To get all namespaces: kubectl get pods --all-namespaces

#### 

#### Resource Quota

The compute-quota.yml is as follows:

apiVersion: v1  
kind: ResourceQuota  
metadata:  
  name: compute-quota  
  namespace: dev  
spec:  
  hard:  
    pods: "10"  
    requests.cpu: "4"  
    requests.memory: 5Gi  
    limits.cpu: "10"  
    limits.memory: 10Gi

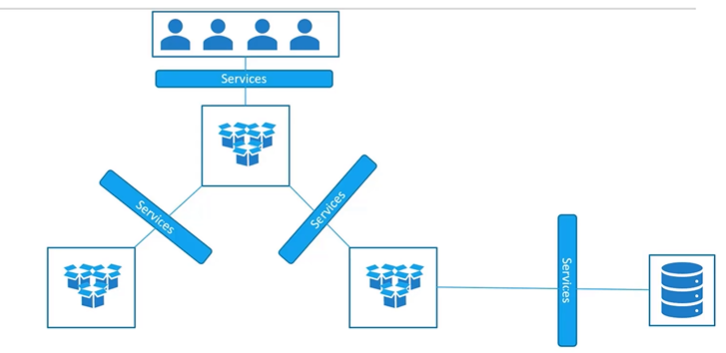
Commands:

1. To create a resource-quota (with manifest): kubectl create -f compute-quota.yml

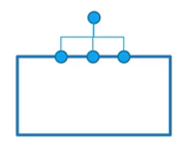
### Services

An abstract way to expose an application running on a set of Pods as a network service.

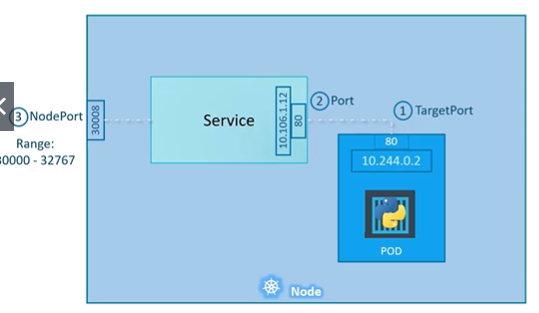
With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.



#### Service Types:

1. NodePort  
   
2. ClusterIP  
   
3. LoadBalancer

#### Services NodePort



The **nodeport-service-definition.yml** is as follows:  
apiVersion: v1  
kind: Service  
metadata:  
  name: myapp-service  
spec:  
  type: NodePort  
  ports:  
    - targetPort: 80    # port on target container  
      port: 80          # port on service  
      nodePort: 30008   # port on node  
  selector:  
    app: myapp  
    type: front-end

Commands:

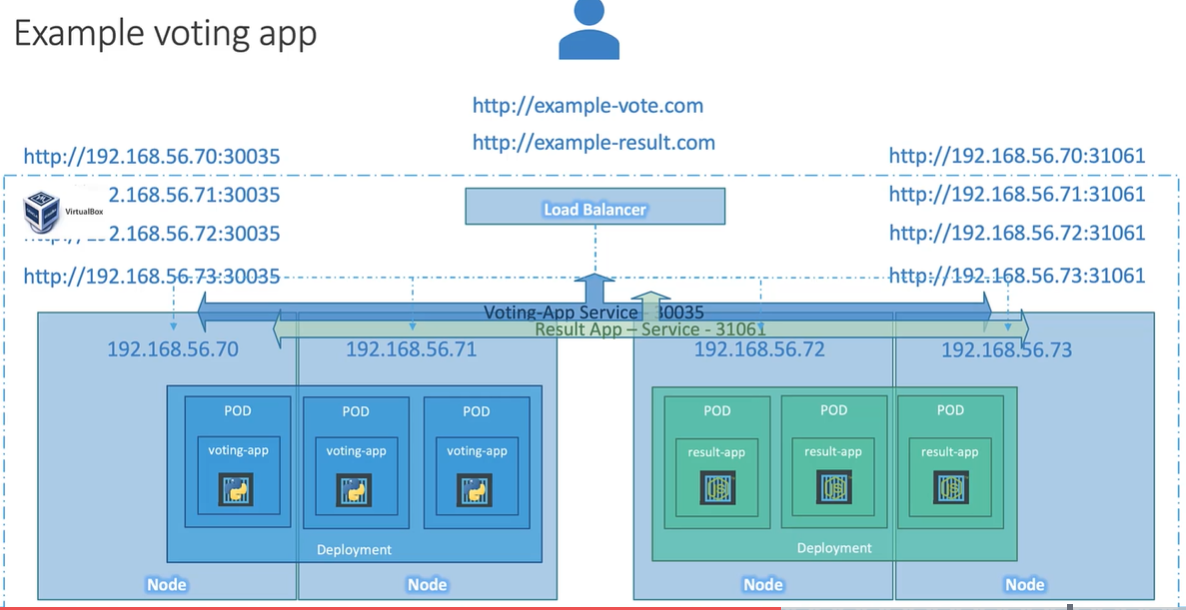
1. To create a service: kubectl create -f service-definition.yml
2. To get services: kubectl get services

#### Services ClusterIP

The **clusterip-service-definition.yaml** is as follows:

apiVersion: v1  
kind: Service  
metadata:  
  name: back-end  
spec:  
  type: ClusterIP  
  ports:  
    - targetPort: 80    # port where the backend is exposed  
      port: 80          # port where the service is exposed  
  selector:  
    app: myapp  
    type: back-end

#### Services LoadBalancer

Typical solution:

Instead we can use native load balancer.

The **loadbalancer-service-definition.yml** is as follows:

apiVersion: v1  
kind: Service  
metadata:  
  name: myapp-service  
spec:  
  type: LoadBalancer  
  ports:  
    - targetPort: 80    # port where the backend is exposed  
      port: 80          # port where the service is exposed  
      nodePort: 30008

### Kubernetes Imperative and Declarative

#### Imperative Commands

##### Create objects

##### kubectl run --image=nginx nginx kubectl create deployment --image=nginx nginx kubectl expose deployment nginx --port 80

kubectl run httpd --image=httpd:alpine --port=80 --expose

##### Update Objects

kubectl edit deployment nginx  
kubectl label pods labelex owner=michael  
kubectl scale deployment nginx --replicas=5  
kubectl set image deployment nginx nginx=nginx:1.18  
kubectl create -f nginx.yml  
kubectl replace -f nginx.yml  
kubectl delete -f nginx.yml

#### Declarative Commands

kubectl apply -f nginx.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

IMPORTANT:

kubectl create deployment does not have a --replicas option. You could first create it and then scale it using the kubectl scale command.

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

You can then update the YAML file with the replicas or any other field before creating the deployment.

**Service**

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

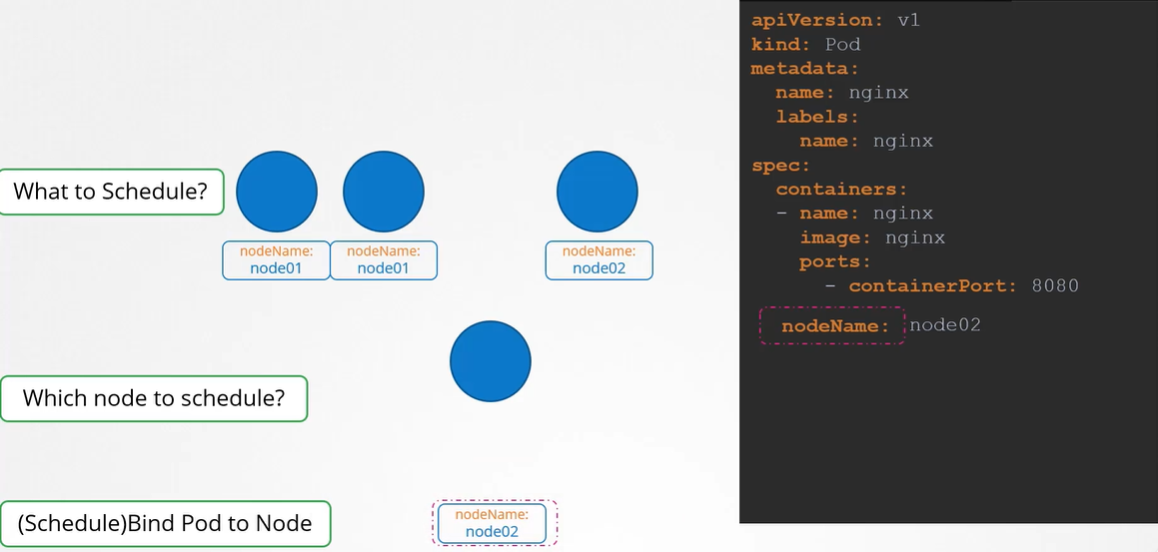
(This will not use the pods labels as selectors)

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.

# Scheduling

## Manually Scheduling

The scheduler goes through all the nodes and finds those nodes who’s this property is not set. These are the nodes which will be used for scheduling. Once identified, it sets this property of the node in the pod along side with it and schedules the pod on that node, creating a binding object.



## No Scheduler

Create a **pod-bind-definition.yml** and manually schedule the pods on the node.

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

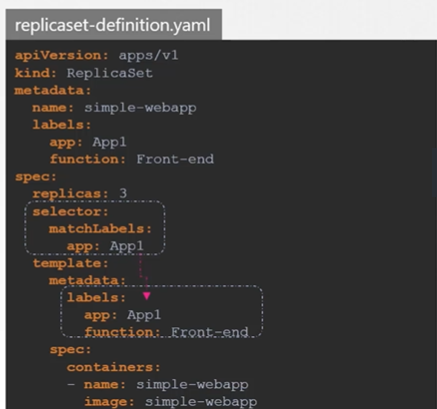
Then send a POST request to binding API:

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

This will schedule the pods which label on the specified node.

## Labels and Selectors

Using selectors to select the pod with given label:

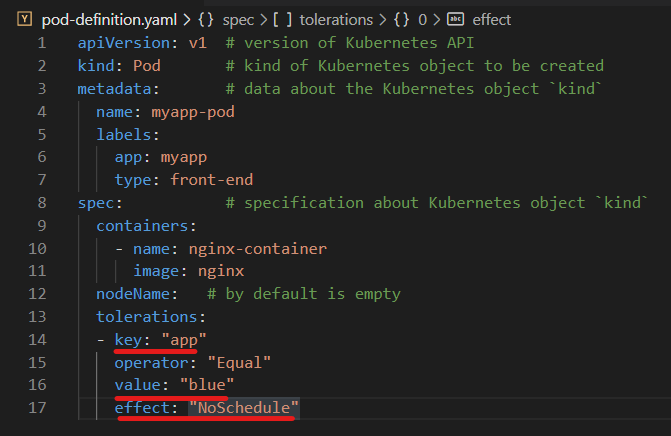
kubectl get pods --selectors app=App1

### Annotations

## Taints and Tolerations

They are used to restrict the pod to be scheduled to particular nodes.

*Taints* allow a node to repel a set of pods. *Tolerations* are applied to pods, and allow the pods to schedule onto nodes with matching taints.

****Taints and tolerations work together to ensure the pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

**Taints – Node:** kubectl taint nodes node-name key-value: taint-effect

E.g., kubectl taint nodes node1 app=blue :NoSchedule

##### Taint Effects:

In particular,

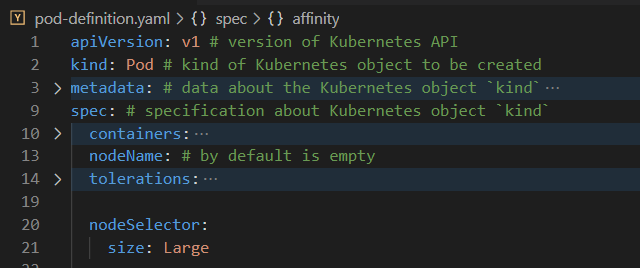
* if there is at least one un-ignored taint with effect NoSchedule then Kubernetes will not schedule the pod onto that node
* if there is no un-ignored taint with effect NoSchedule but there is at least one un-ignored taint with effect PreferNoSchedule then Kubernetes will try to not schedule the pod onto the node
* if there is at least one un-ignored taint with effect NoExecute then the pod will be evicted from the node (if it is already running on the node), and will not be scheduled onto the node (if it is not yet running on the node).

**Removing Tolerations – Pods:**

E.g., kubectl taint nodes master/controlplane node-role.kubernetes.io/master:NoSchedule-

## Node Selectors

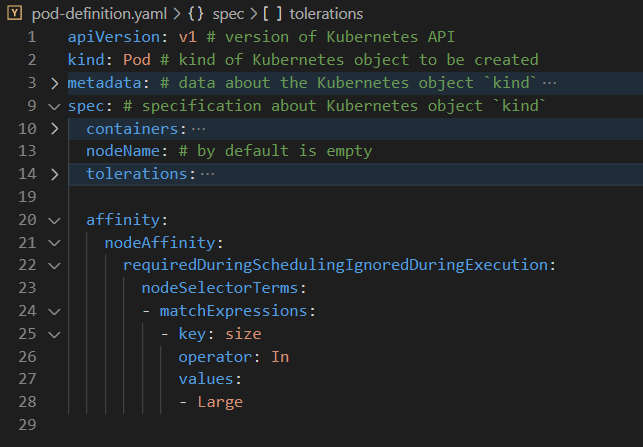
nodeSelector is the simplest recommended form of node selection constraint. nodeSelector is a field of PodSpec. For the pod to be eligible to run on a node, the node must have each of the indicated key-value pairs as labels (it can have additional labels as well). The most common usage is one key-value pair.

1. Run kubectl get nodes to get the names of your cluster’s nodes.
2. Pick out the one that you want to add a label to , and then run kubectl label nodes <node-name> <label-key>=<label-value> to add a label to the node you’ve chosen.   
   E.g., kubectl label nodes kubernetes-foo-node-1.c.a-robinson.internal disktype=ssd
3. You can verify that it worked by re-running kubectl get nodes --show-labels and checking that the node now has a label. You can also use kubectl describe node "nodename" to see the full list of labels of the given node.
4. Take whatever pod config file you want to run, and add a nodeSelector section to it, like this.
5. When you then run kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml, the Pod will get scheduled on the node that you attached the label to. You can verify that it worked by running kubectl get pods -o wide and looking at the "NODE" that the Pod was assigned to.

## Node Affinity

Node affinity is conceptually similar to nodeSelector -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

There are currently two types of node affinity, as follows:

1. requiredDuringSchedulingIgnoredDuringExecution- "only run the pod on nodes with Intel CPUs",
2.  preferredDuringSchedulingIgnoredDuringExecution- "try to run this set of pods in failure zone XYZ, but if it's not possible, then allow some to run elsewhere".

# Logging & Monitoring

# Application Lifecycle Management

# Cluster Maintenance

# Security

# Storage

# Networking

# Design and Install a Kubernetes Cluster

# Install “Kubernetes the kubeadm way”

# End to End Tests on a Kubernetes Cluster

# Troubleshooting

# Other Topics

# Lightning Labs

# Mock Exams