

Kubernetes Basic

Deploying and Accessing an Application

Let's launch a simple Nginx web server and access its default web page, Execute below command to pull the Nginx image from Docker Hub and create a deployment called **myweb**:

```
kubectrl run --image=nginx:latest myweb
```

OR

```
kubectrl run --generator=run-pod/v1 --image=nginx mynginx1
```

(Note: both the commands are deprecated; Kubernetes recommend us to use create command.)

It packages and deploys the container in a Kubernetes-specific artifact called a *Pod*

To access the web server running inside the Pod, (here target pod is port of pod)

Syntax:

```
kubectrl expose pod <pod name> --port=80 --target-port=<pod port> --type=<NodePort for Service>
```

ex:

```
kubectrl expose pod myweb-59d7488cb9-jvnwn --port=80 --target-port=80 --type=NodePort
```

The Pod is now exposed on every Node of the cluster on an arbitrary port.

(For a NodePort service, Kubernetes allocates a port from a configured range (default is 30000-32767), and each node forwards that port, which is the same on each node, to the service. It is possible to define a specific port number, but you should take care to avoid potential port conflicts.)

P.S <https://kubernetes.io/docs/concepts/services-networking/service/>

Note: A Service can map any incoming port to a targetPort. By default and for convenience, the **targetPort** is set to the same value as the **port** field.

Note: Once you expose pod on every node, it may create problem for you

- If something gets wrong with exposed pod then again and again you have to expose pod manually.
- because it exposes only one pod, let imagine we have multiple pod, in that case this strategy won't work.
- Instead of exposing individual pod we can expose deployment using below command

```
kubectrl expose deployment <deployment name> --port=80 --target-port=<pod port> --type=<NodePort for Service>
```

Benefits: drawback of above points are resolved.

`kubectl get pods -o wide` -----> it will let you know that how many pods are running on which node
VMs or Get the status of the service

`kubectl get pods --all-namespaces` -----> list all the nodes

To get the NodePort of the `myweb` deployment, run the following command.

Syntax: `kubectl get svc <pod name>`

Ex: `kubectl get svc myweb-59d7488cb9-jvnwn`

Useful Kubectl commands

`kubectl exec -it <pod> -- /bin/bash`

`kubectl describe pod <Pod name>` ? Describe pod

`kubeadm token create --print-join-command` -> Regenerate kubeadm join token

`kubectl get deployments` ---? Get info on current deployments

`kubectl get rs` ---? get info about the replica sets

`kubectl get pods --show-labels` ---? Get pods and show the labels attached to those pods.

`kubectl rollout status deployment/myweb` --? get deployment status

`kubectl set image deployment/helloworld-deployment k8s-demo=k8s-demo2` - -

? run the k8s-demo with the image label version2

`kubectl edit deployment/helloworld-deployment` -> Edit deployment object

then check status of deployment and number of pod.

`kubectl rollout status deployment/helloworld-deployment` -> get the status of the rollout

`kubectl rollout history deployment/helloworld-deployment - get the rollout history` -> useful for
rollback to a specific version.

`kubectl rollout undo deployment/helloworld-deployment` -> rollback to previous version

`kubctl rollout undo deployment/helloworld-deployment --to-revision=N` -> roll back to a specific
version.

`kubectl delete rc <replicationcontroller deployment name>` -> to delete replication controller
deployment

```
kubectl get pods redis-master --template='{{(index (index .spec.containers 0).ports 0).containerPort}}{{"\n"}}'
```

-> Verify that the Redis server is running in the pod and listening on port:

LAB:

In a two-node cluster, a web application has in-memory cache such as redis. We want the web-servers to be co-located with the cache as much as possible

Lab 2: Deploy Jenkins on Kubernetes

```
kubectl run --image=Jenkins myjenkins
kubectl expose pod <pod name> --port=8080 --target-port=8080 --type=NodePort
kubectl get svc
then browse <VM IP>:<Port>
to get password: login to pod
kubectl exec -it <pod> -- /bin/bash
```

Assignment: Expose deployment instead of pod

Lab 3: My first Manifest file.

vi myfirstk8sfile.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: mytomcat
spec:
  containers:
  - name: mytomcat
    image: tomcat
    ports:
    - containerPort: 80
```

kubectl create -f myfirstk8sfile.yml

Lab 4: vi nginxdeployment.yml

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

Note: make sure key: value pair of matchLabels and key: value pair of labels should match

```
kubectl create -f <yaml file name>.yaml
kubectl create -f nginxdeployment.yml
kubectl get pods
kubectl get pods -o wide
kubectl get services
kubectl get deployments
kubectl delete deployments <deployment name>
```

(The create command can be used to create a pod directly, or it can create a pod or pods through a Deployment. It is highly recommended that you use a Deployment to create your pods. It watches for failed pods and will start up new pods as required to maintain the specified number. If you don't want a Deployment to monitor your pod (e.g. your pod is writing non-persistent data which won't survive a restart, or your pod is intended to be very short-lived), you can create a pod directly with the create command.

You need a **deployment** object - or other Kubernetes API objects like a **replication controller** or **replicaset** - that needs to keep the **replicas** (pods) alive (that's kind of the point of using kubernetes).

What you will use in practice for a typical application are:

1. **Deployment object** (where you will specify your apps container/containers) that will host your app's container with some other specifications.
2. **Service object** (that is like a grouping object and gives it a so-called virtual IP (cluster IP) for the pods that have a certain label - and those pods are basically the app containers that you deployed with the former **deployment** object).

You need to have the **service** object because the pods from the deployment object can be killed, scaled up and down, and you can't rely on their IP addresses because they will not be persistent.

So you need an object like a **service**, that gives those pods a stable IP.

)

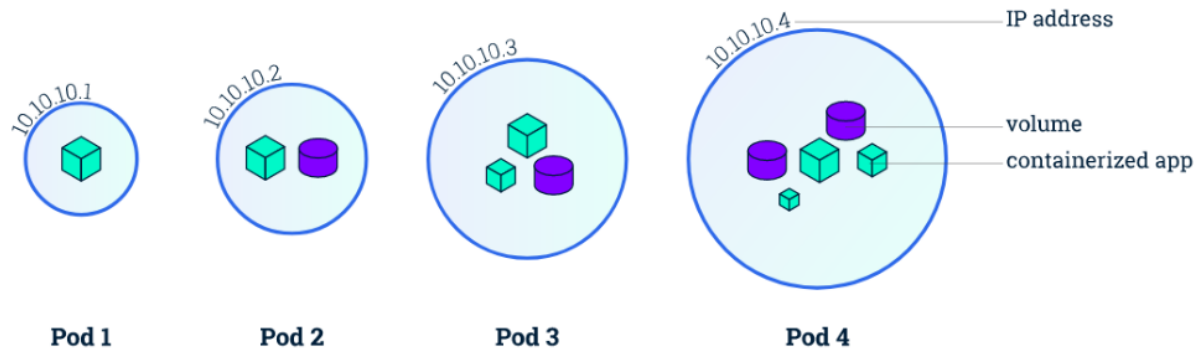
Replicaset and Replication Controller:

The major difference is that the rolling-update command works with Replication Controllers, but won't work with a Replica Set. This is because Replica Sets are meant to be used as the backend for Deployments

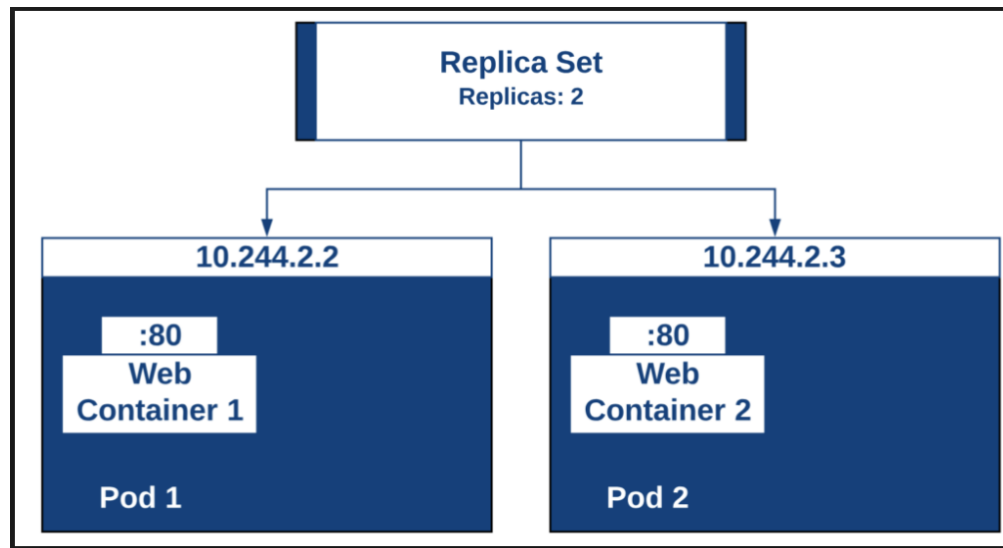
(Installations Step: <https://kubernetes.io/docs/setup/independent/install-kubeadm/>)

Access Dashboard through GCP: <https://cloud.google.com/kubernetes-engine/docs/how-to/exposing-apps>)

Pods: Pods are nothing but they are the smallest deployable unit. Its small group of tightly coupled containers it shared network and data within It's router table.

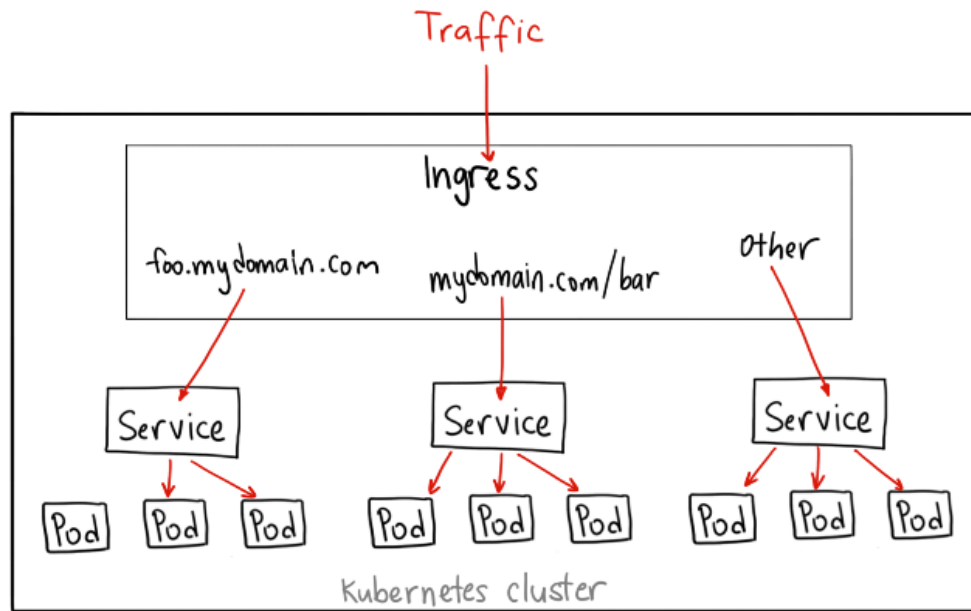


Replica Sets: It runs those copies replicas of pods. It starts to kill pods if necessary. It handles pods failures and it does help check as needed.

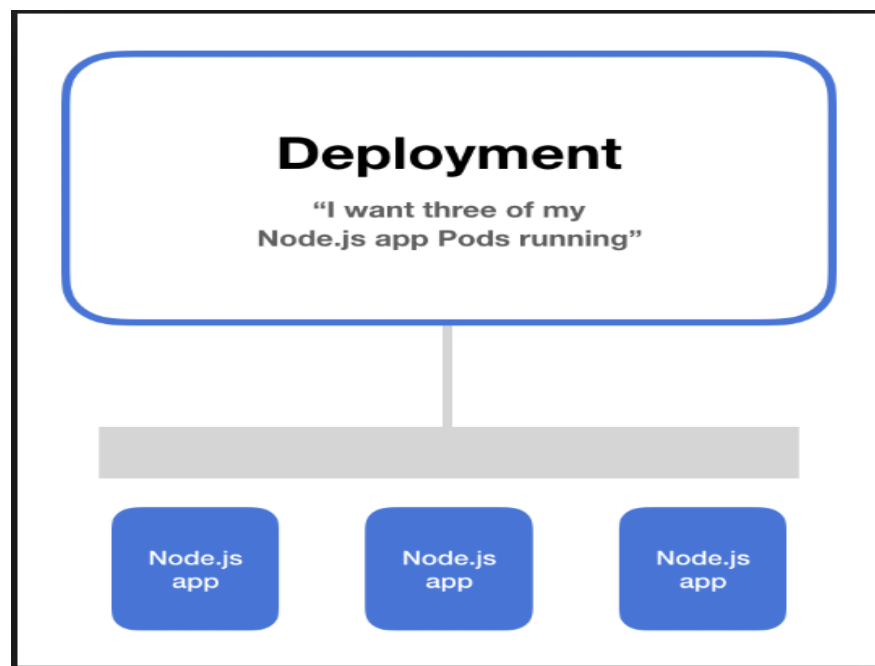


Ingress: It exposes services to outside the world, it maps your url to the services. You configure access by creating a collection of rules that define which inbound connections reach which services.

Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a “smart router” or entrypoint into your cluster.



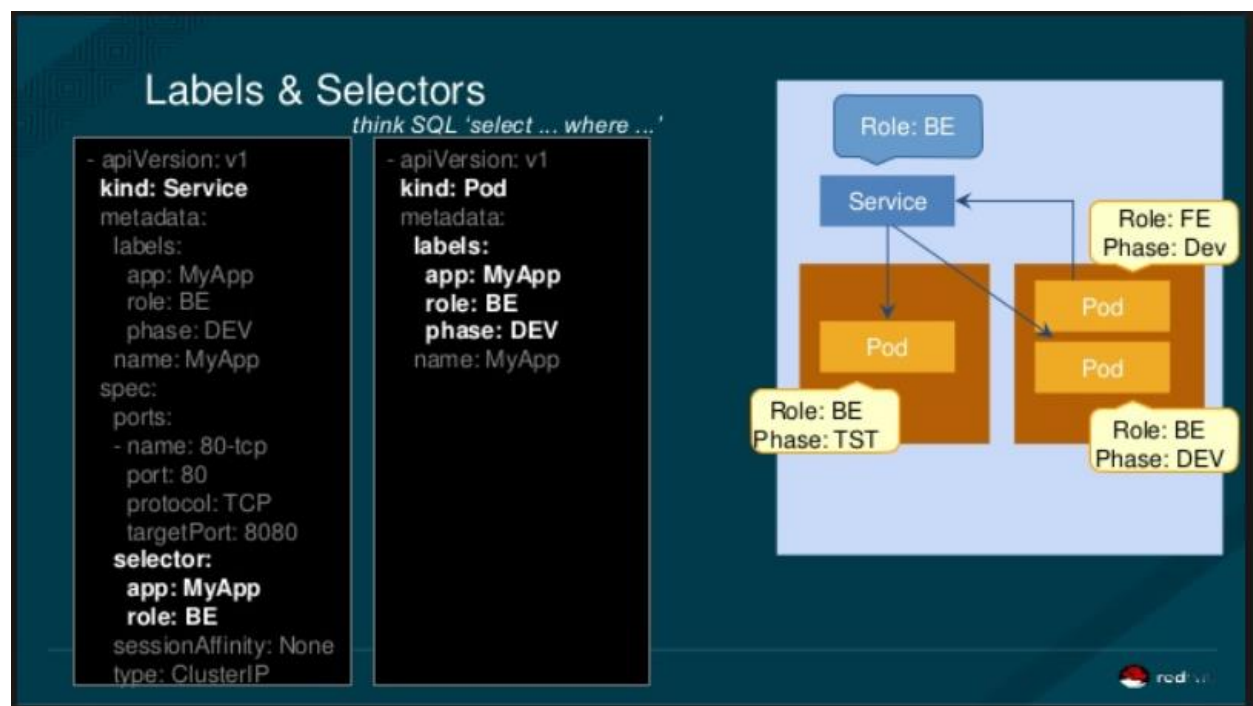
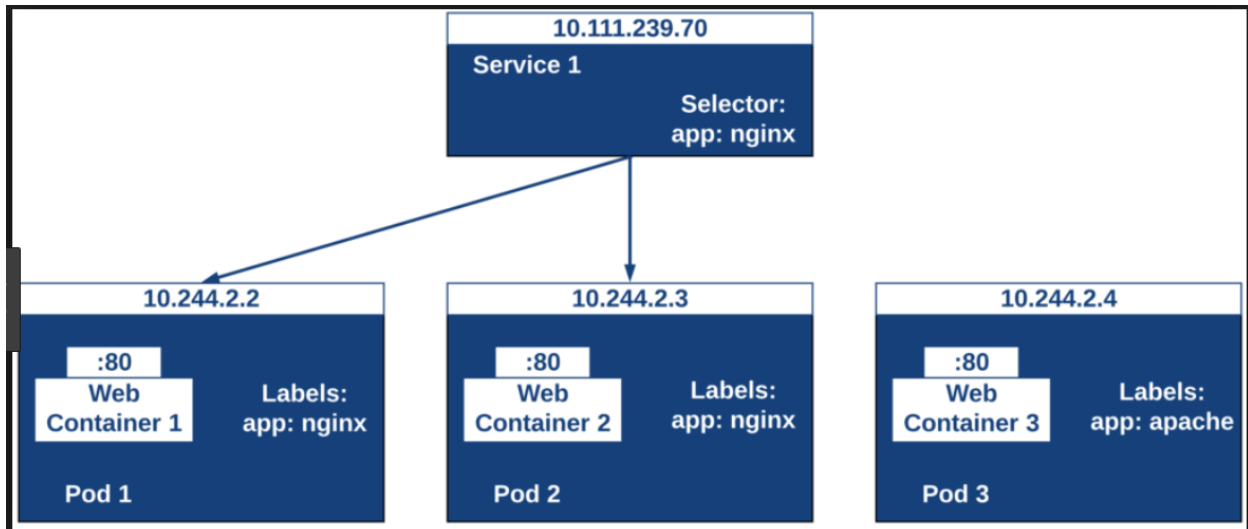
Deployments: It can be used for declaration of your application which contain the image tag etc, and environmental variables data volumes.



Namespaces: created namespaces for your environments example test, staging, production,

Services: Pods can be exposed internally or externally through services, services enable that discovery of pod by associating a set of pods to a specific criteria.

Selector: Pods are associated to services via key value pair called labels and selectors, and in any pod the labels that matches the selector will automatically be discovered by the services.



Labels:

Labels are key-value pairs which are attached to pods, replication controller and services. They are used as identifying attributes for objects such as pods and replication controller. They can be added to an object at creation time and can be added or modified at the run time.

Selectors:

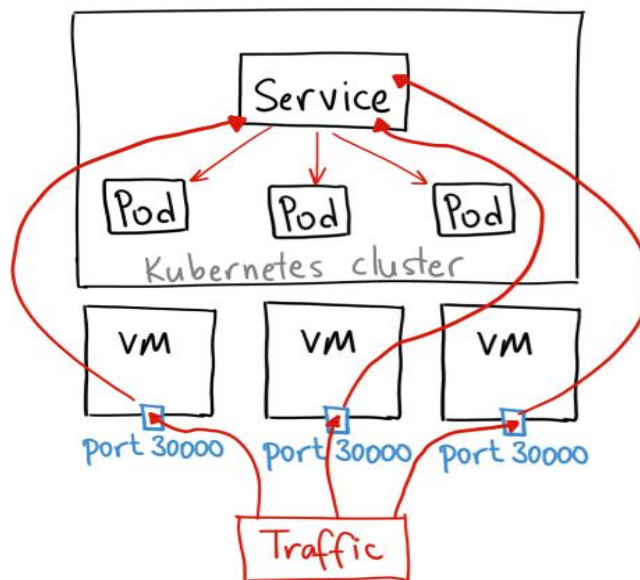
Labels do not provide uniqueness. In general, we can say many objects can carry the same labels.

ClusterIP:

A ClusterIP service is the default Kubernetes service. It gives you a service inside your cluster that other apps inside your cluster can access. There is no external access.

NodePort

A NodePort service is the most primitive way to get external traffic directly to your service. NodePort, as the name implies, opens a specific port on all the Nodes (the VMs), and any traffic that is sent to this port is forwarded to the service.

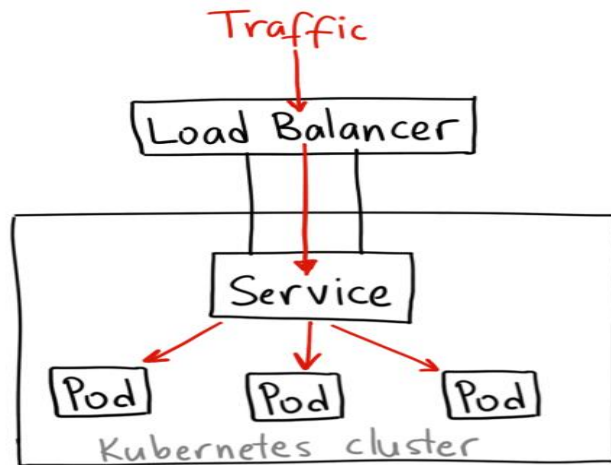


There are many downsides to this method:

1. You can only have once service per port
2. You can only use ports 30000–32767
3. If your Node/VM IP address change, you need to deal with that

LoadBalancer

A LoadBalancer service is the standard way to expose a service to the internet. On GKE, this will spin up a Network Load Balancer that will give you a single IP address that will forward all traffic to your service.



Prefer Ingress over NodePort and LoadBalancer

Life Cycle of Pod:

Phases of a Pod

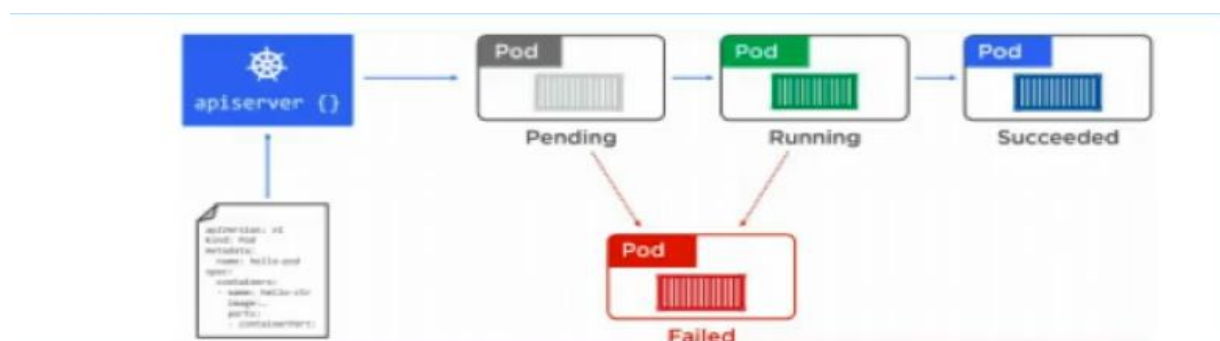
Pending: Accepted by Kubernetes but container not created yet.

Running: Pod bound to a node, all containers created and at least one container is running/starting/restarting

Succeeded: Container(s) exited with status 0

Failed: All containers exit and at least one exited with non-zero status.

Unknown: State of Pod could not be determined due to communication issues with its node



Kubectl Commands Cheat Sheet



Pod & Container Introspection

```
# List the current pods
kubectl get pods
# Describe pod <name>
kubectl describe pod <name>
# List the replication controllers
kubectl get rc
# List the replication controllers in <namespace>
kubectl get rc --namespace=<namespace>
# Describe replication controller <name>
kubectl describe rc <name>
# List the services
kubectl get svc
# Describe service <name>
kubectl describe svc <name>
# Delete pod <name>
kubectl delete pod <name>
# Watch nodes continuously
kubectl get nodes -w
```

Cluster Introspection

```
# Get version information
kubectl version
# Get cluster information
kubectl cluster-info
# Get the configuration
kubectl config view
# Output information about a node
kubectl describe node <node>
```

Debugging

```
# Execute <command> on <service> optionally #
selecting container <$container>
kubectl exec <service> <command> [-c <$container>]
# Get logs from service <name> optionally # selecting
container <$container>
kubectl logs -f <name> [-c <$container>]
# Watch the Kubelet logs
watch -n 2 cat /var/log/kublet.log
# Show metrics for nodes
kubectl top node
# Show metrics for pods
kubectl top pod
```

Quick Commands

```
# Launch a pod called <name>
# using image <image-name>
kubectl run <name> --image=<image-name>
# Create a service described # in <manifest.yaml>
kubectl create -f <manifest.yaml>
# Scale replication controller
# <name> to <count> instances
kubectl scale --replicas=<count> rc <name>
# Map port <external> to # port <internal> on replication
# controller <name>
kubectl expose rc <name> --port=<external> --target-
port=<internal>
# Stop all pods on <n>
kubectl drain <n> --delete-local-data --force --ignore-
daemonsets
# Create namespace <name>
kubectl create namespace <namespace>
# Allow Kubernetes master nodes to run pods
kubectl taint nodes --all node-role.kubernetes.io/master-
```

Objects

```
all
clusterrolebindings
clusterroles
cm = configmaps
controllerrevisions
crd = customresourcedefinition
cronjobs
cs = componentstatuses
csr = certificatesigningrequests
deploy = deployments
ds = daemonsets
ep = endpoints
ev = events
hpa = horizontalpodautoscalers
ing = ingresses
jobs
limits = limitranges
netpol = networkpolicies
no = nodes
ns = namespaces
pdb = poddisruptionbudgets
po = pods
podpreset
podtemplates
psp = podsecuritypolicies
pv = persistentvolumes
pvc = persistentvolumeclaims
quota = resourcequotas
rc = replicationcontrollers
rolebindings
roles
rs = replicaset
sa = serviceaccounts
sc = storageclasses
secrets
sts = statefulsets
```

<https://files.acrobat.com/a/preview/5f7379e6-8331-4b90-ac64-691190c06ad3>

```

1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  spec:
6    revisionHistoryLimit: 5
7    minReadySeconds: 10
8    selector:
9      matchLabels:
10       app: nginx
11       deployer: distelli
12    strategy:
13      type: RollingUpdate
14      rollingUpdate:
15        maxUnavailable: 1
16        maxSurge: 1
17      replicas: 3
18      template:
19        metadata:
20          labels:
21            app: nginx
22            deployer: distelli
23        spec:
24          containers:
25            - name: nginx
26              image: nginx: 1.7.9

```

- replicas – Tells Kubernetes how many pods to create during a deployment. Modifying this field is an easy way to scale a containerized application.

- `spec.strategy.type` – Suppose there is another version of the application that needs to be deployed, and during the deployment phase, we need to update without facilitating an outage. The Rolling Update strategy allows Kubernetes to update a service without facilitating an outage by proceeding to update pods one at a time.
- `spec.strategy.rollingUpdate.maxUnavailable` – The maximum number of pods that can be unavailable during the Rolling update.
- `spec.strategy.rollingUpdate.maxSurge` – The maximum number of pods that can be scheduled above the desired number of pods.
- `spec.minReadySeconds` – An optional Integer that describes the minimum number of seconds, for which a new pod should be ready without any of its containers crashing for it to be considered available.
- `spec.revisionHistoryLimit` – An optional integer attribute that you can use to tell Kubernetes explicitly how many old ReplicaSets to retain at any given time.
- `spec.template.metadata.labels` – Adds labels to a deployment specification
- `spec.selector` – An optional object that tells the Kubernetes deployment controller to only target pods that match the specified labels. Thus, to only target pods with the labels “app” and “deployer” we can make the following modification to our deployment yaml.

Kubernetes Dashboard:

For Ubuntu VM: reset password and install XRDP

```
passwd ubuntu
sudo apt-get install xrdp

sudo apt-get install mate-core mate-desktop-environment mate-notification-daemon

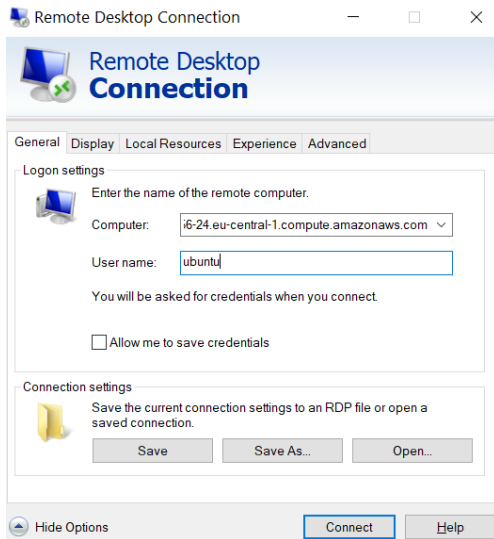
sudo sed -i.bak 's/fi/a #xrdp multiple users configuration \n xfce-session \n' /etc/xrdp/startwm.sh

sudo ufw allow 3389/tcp
```

Please follow the current link [here](https://github.com/kubernetes/dashboard/wiki/Installation)

<https://github.com/kubernetes/dashboard/wiki/Installation>

press windows+R then type mstsc , paste public ip of your Kubernetes master and login as ubuntu user.



Once you logged in

Search for terminal and type

```
kubectl proxy
```

Then open browser and paste below url

<http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/>

To get **token** Run the following commands:

This command will create a service account for a dashboard in the default namespace

```
kubectl create serviceaccount dashboard -n default
```

Add the cluster binding rules to your dashboard account

```
kubectl create clusterrolebinding dashboard-admin -n default --  
clusterrole=cluster-admin --serviceaccount=default:dashboard
```

```
kubectl get secret $(kubectl get serviceaccount dashboard -o  
jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 --  
decode
```

Copy the token and paste in browser.

Assigning Pods to Specific Node

You can constrain a pod to only be able to run on particular nodes or to prefer to run on particular nodes

The recommended approaches all use label selectors to make the selection. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (e.g. spread your pods across nodes, not place the pod on a node with insufficient free resources, etc.) but there are some circumstances where you may want more control on a node where a pod lands, e.g. to ensure that a pod ends up on a machine with an SSD attached to it, or to co-locate pods from two different services that communicate a lot into the same availability zone.

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

ex:

```
kubectl label nodes ip-172-31-41-97 disktype=ssd
```

Verify: `kubectl get nodes --show-labels`

Where: Node name is 'kubernetes-foo-node-1.c.a-robinson.internal'

And label is "disktype=ssd"

Demo: Add a nodeSelector field to your pod configuration,

`vi pod-nginx.yaml`

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

Verify: `kubectl get pods -o wide`