Day1. What are containers, Intro to Docker, Install Docker, Commands with Docker

Day2· Kube Architecture, Install, BootStrap, Components, Folders, Api -resources, Pods

Day3. YAML, Multipods, Static, lnit pods, Namespace, labels, limits, Env variables"

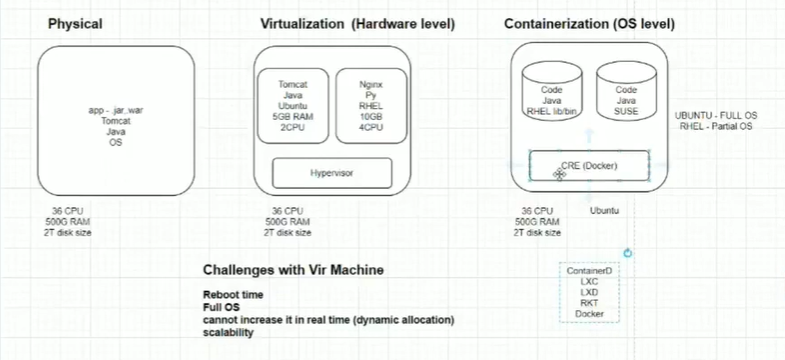
Day4. Controllers, rolling updates/rollbacks, Services, Daemonset

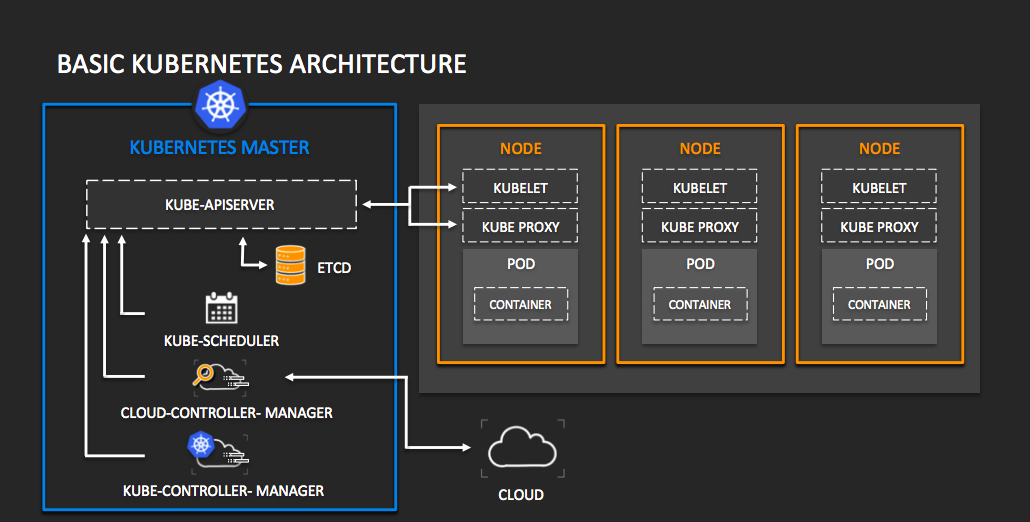
Day5. Storages, ConfigMaps, Secrets, Cloud Disk, Service- LB

Day6. Schedular, Namespaces limits, etcd-backup, Monitoring

Day7. Networks, Security, Contexts, Config file"

Day8. Troubleshooting, lngress, Stateful sets and Headless svc, Helm, HPA(Horizontal pod autoscaling), EXAM tips





Kubernetes is your container orchestration tool.

Kubernetes can create any kind of container but docker can create only docker base container.

**Kubernetes Features:**

* Automated rollouts and rollbacks
* Service discovery and load balancing
* Storage orchestration
* Secret and configuration management
* Automatic bin packing
* Batch execution
* IPv4/IPv6 dual-stack
* Horizontal scaling
* Self-healing
* Designed for extensibility

**Kube Architecture**

Master Node01

**Master Components**

API Server:6443

Scheduler

Controller Manager

ETCD

2CPU, 4GB RAM on each node

**kubectl**

Worker Node03

Kubelet

Docker

Kube-proxy

Worker Node02

Kubelet

Docker

Kube-proxy

Worker Node01

Kubelet

Docker (CRI- container runtime interface)

Kube-proxy

**CNI (Container Network Interface):** It is networking plugin it is dependent on your running container (flannel, weave, Calico etc.)

**Master Node:**

**API Server 🡪** It is responsible for all the API base communications in k8s cluster. Master server components (Scheduler, Controller Manager**,** ETCD) also connect with each other with the help of API server only.

**Scheduler** 🡪 It is helping to create container on worker nodes, as per your requirement.

Like: high resource etc.….

**Controller Manager** 🡪 is a manager and it is a kind of multiple managers, it is not a single component,

It is responsible to manage required container always up and running, also if any node down, it is migrate fault node container to healthy node.

**ETCD** 🡪 it is the database and all information about your running cluster it has, like how many nodes container running etc.… running in you cluster. The database with key value pair.

**Worker Node:**

**Kubelet** 🡪 It is also a container(agent) and It take care all the worker node components whether it is container or etc.… it make sure all the container always up and running, it is taking all the instruction from you master API server

Example: If you want to create a container, The request come to kubelet it sends request to CRI (docker software) to create a container or delete a container.

OPT: then scheduler send a request to api server and api server send request to kubelet create a container on any free worker node.

**Kube-proxy** 🡪 It help in networking, consider if one container running on node01 or one container running on node02,03 inside of your k8s cluster all the container talk to each other and that happen with the help of kube-proxy.

**Installation**

****

1. Update the apt package index and install packages needed to use the Kubernetes apt repository:

sudo apt-get update

sudo apt-get install -y apt-transport-https ca-certificates curl

1. Download the Google Cloud public signing key:

sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg

1. Add the Kubernetes apt repository:

echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list

1. Update apt package index, install kubelet, kubeadm and kubectl, and pin their version:

sudo apt-get update

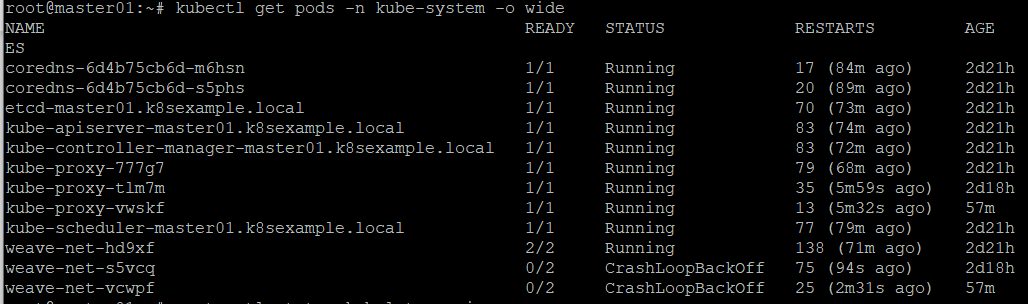
sudo apt-get install -y kubelet kubeadm kubectl

sudo apt-mark hold kubelet kubeadm kubectl

Day 2

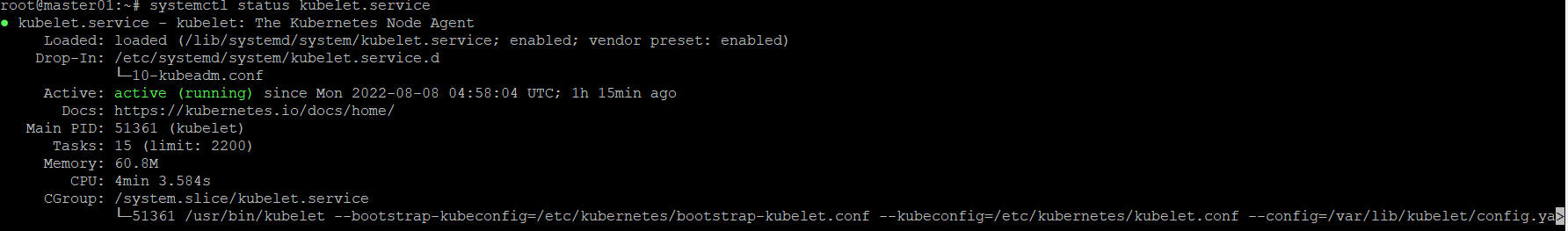
**PODs**

root@master01:~# kubectl get pods -n kube-system -o wide



Q: Why Kubelet is missing

Ans: Because kubelet is running as a service not pod.



Kubectl send request to Api server and Api server check with the scheduler which node is capable to create pod/container, scheduler check the capability with the help from etcd, which node is capable to run the container, now Api server connect with kubelet to create pod/container on node02.

**PODs**

In the Kubernetes POD never change IP until you delete the pod, but the container IP change whenever container get reboot.

**Create pods**

root@master01:~# kubectl api-resources

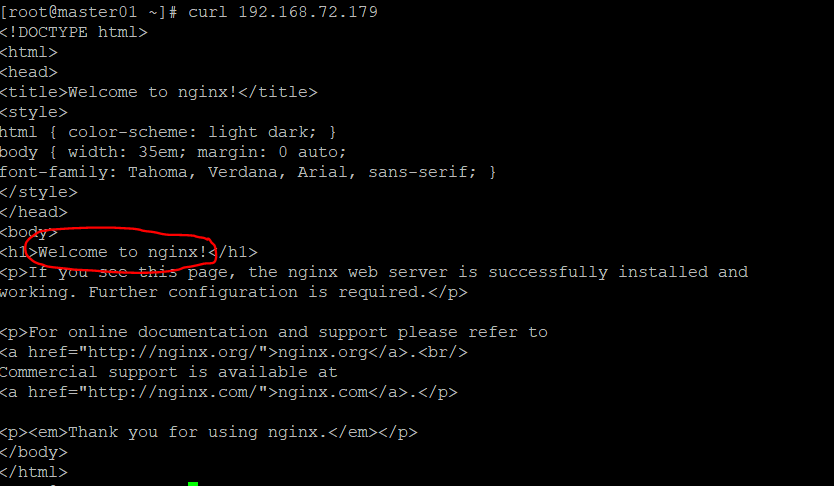
root@master01:~# kubectl run pp --image nginx

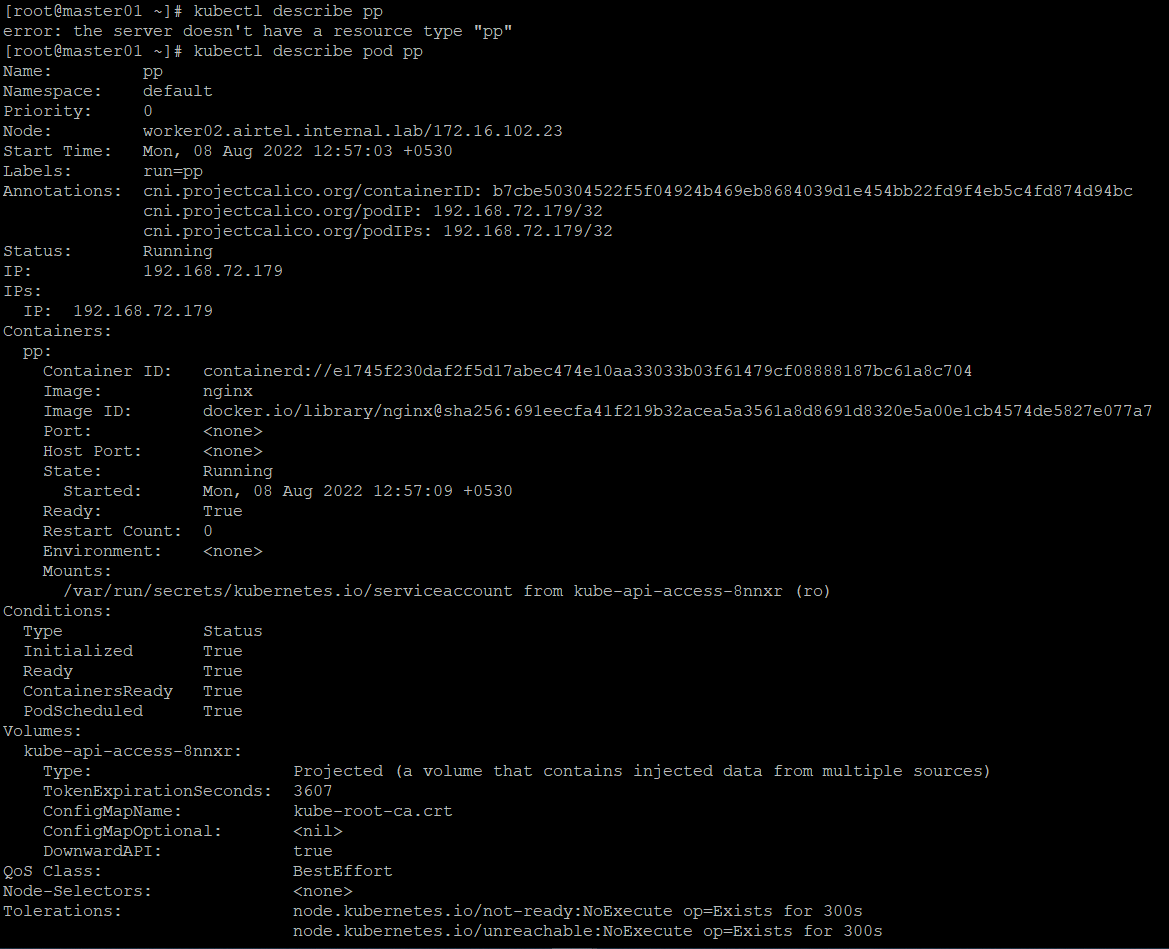
pod/pp created 🡪 pod is created

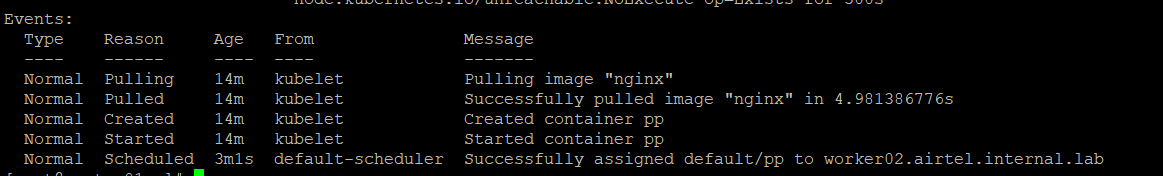
root@master01:~# kubectl get pod -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

pp 1/1 Running 0 38s 192.168.72.179 worker01.k8sexample.local <none> <none>







Please check events where you can see what is going on when you create any pod.

**What is pause container:**

When every you create a pod, it automatically creates container and this container also create a pause container, It recover IP and other details when you delete or restart your container.

Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.

Whenever you create a yaml file pod/container you always have 4 fields.

1. API Version --- API Version defines the versioned schema of this representation of an object.
2. Kind --- Kind is a string value representing the REST resource this object represents.
3. Metadata --- Standard object's metadata.
4. Spec --- Specification of the desired behavior of the pod.
5. Status( Optional ) --- Most recently observed status of the pod.

apiVersion: v1

kind: Pod

metadata:

name: pod1

labels:

app: tomcat

spec:

containers:

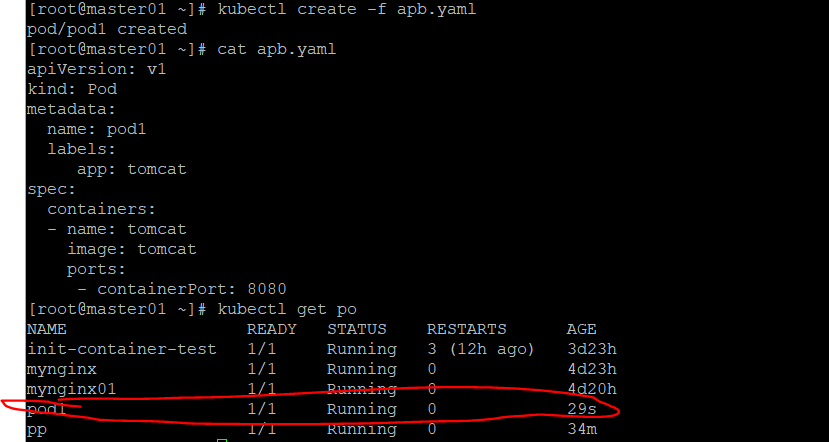
- name: tomcat

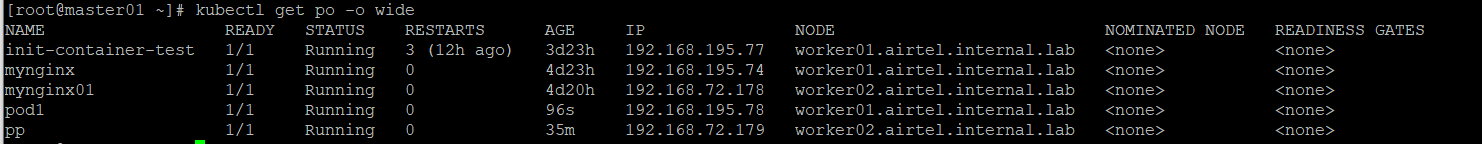
image: tomcat

ports:

- containerPort: 8080

Create pod from YAML file





## **Multi-Container Pod**

This is the 3 type of container

## **Sidecar:**

Sidecars derive their name from motorcycle sidecars. While your motorcycle can work fine without the sidecar, having one enhances or extends the functionality of your bike, by giving it an extra seat. Similarly, in Kubernetes, a sidecar pattern is used to enhance or extend the existing functionality of the container.

## **Ambassador:**

The ambassador pattern derives its name from an Ambassador, who is an envoy and a person a country chooses to represent their country and connect with the rest of the world. Similarly, in the Kubernetes perspective, an Ambassador pattern implements a proxy to the external world. Lets look at an example — If you build an application that needs to connect with a database server, the server configuration, etc, changes with the environment.

Now, the official recommendation to handle these is to use Config Maps, but what if you have legacy code that is already using another way of connecting to the database. Maybe, a properties file, or even worse, a hardcoded set of values. What if you want to communicate with localhost, and you can leave the rest to the admin? You can use the Ambassador pattern for these kinds of scenarios.

So, what we can do is create another container that can act as a TCP Proxy to the database, and you can connect to the proxy via localhost. The sysadmin can then use config maps and secrets with the proxy container to inject the correct connection and auth information.

## **Adapter:**

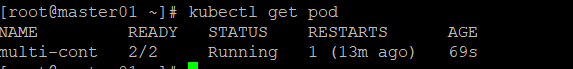
The Adapter is another pattern that you can implement with multiple containers. The adapter pattern helps you standardize something heterogeneous in nature. For example, you’re running multiple applications within separate containers, but every application has a different way of outputting log files.

Now, you have a centralized logging system that accepts logs in a particular format only. What can you do in such a situation? Well, you can either change the source code of each application to output a standard log format or use an adapter to standardize the logs before sending it to your central server. That’s where the adapter pattern comes in.

so, an adapter container transforms the output of the main container.

Example:

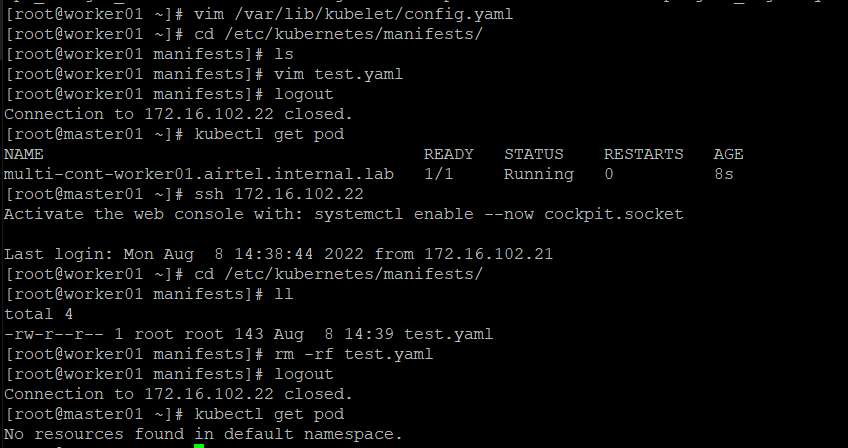
kubectl create -f <https://raw.githubusercontent.com/lerndevops/educka/master/2-pods/multi/multi-cont-pod-ex1.yml>

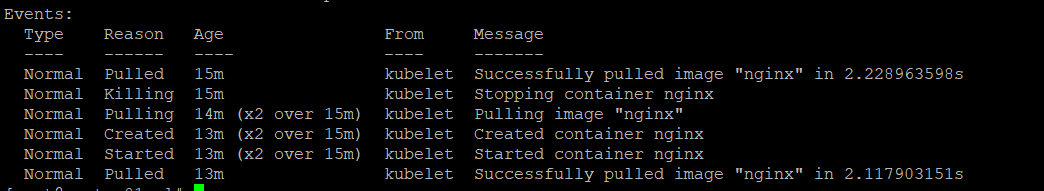


**Statics Pod:**

If you want to create static pod then you need to go on specific pod and go to /var/lib/kubelet/

And check config.yaml check the **StaticPodPath:/etc/Kubernetes/mainfests/** here you create yaml file and check pods on master node.



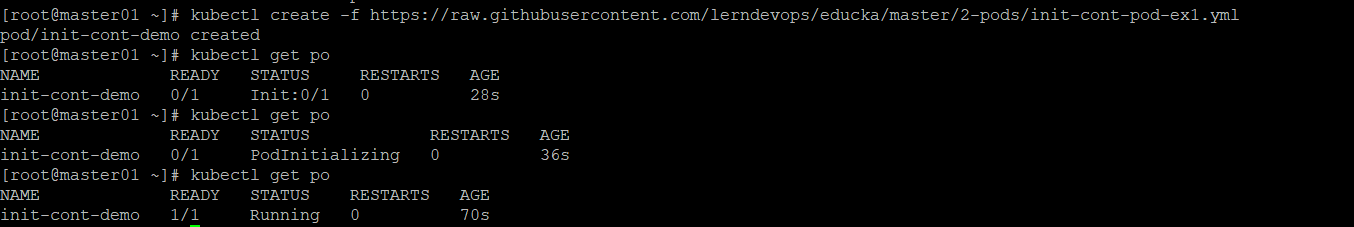


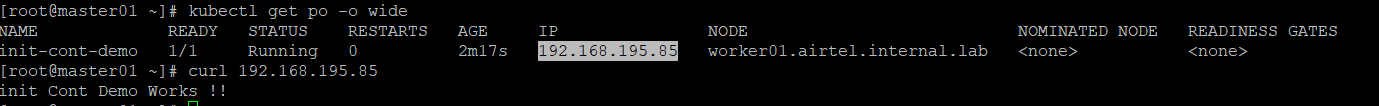
This time you can see scheduler not doing anything on this running pod. It totally skip complete pod deployment process and this pod you cannot delete from master node, you should be delete this pod from running node only.

**Init Container:**

If you use init container, in this situation your pod should not be available until init container not complete init jobs and dry out init container.

**Liveness:** dose not depends on pod availability.



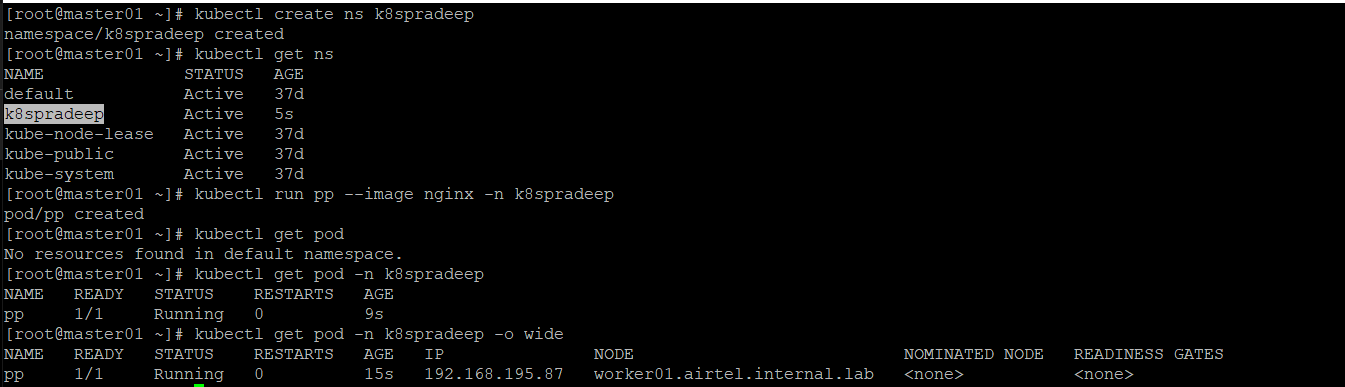


**Namespaces:**

Namespaces are a way to organize clusters into virtual sub-clusters — they can be helpful when different teams or projects share a Kubernetes cluster. Any number of namespaces are supported within a cluster, each logically separated from others but with the ability to communicate with each other. Namespaces cannot be nested within each other.

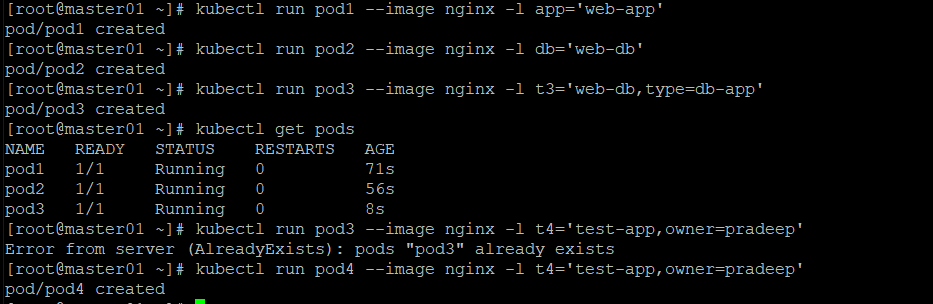
Any resource that exists within Kubernetes exists either in the default namespace or a namespace that is created by the cluster operator. Only nodes and persistent storage volumes exist outside of the namespace; these low-level resources are always visible to every

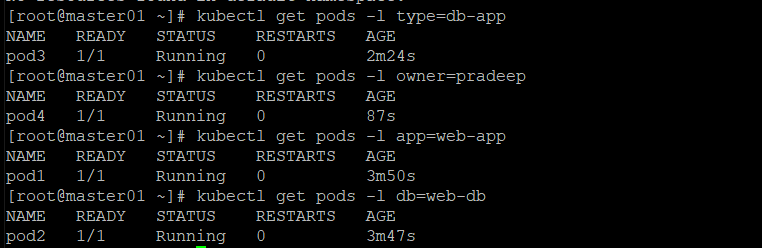
Note:



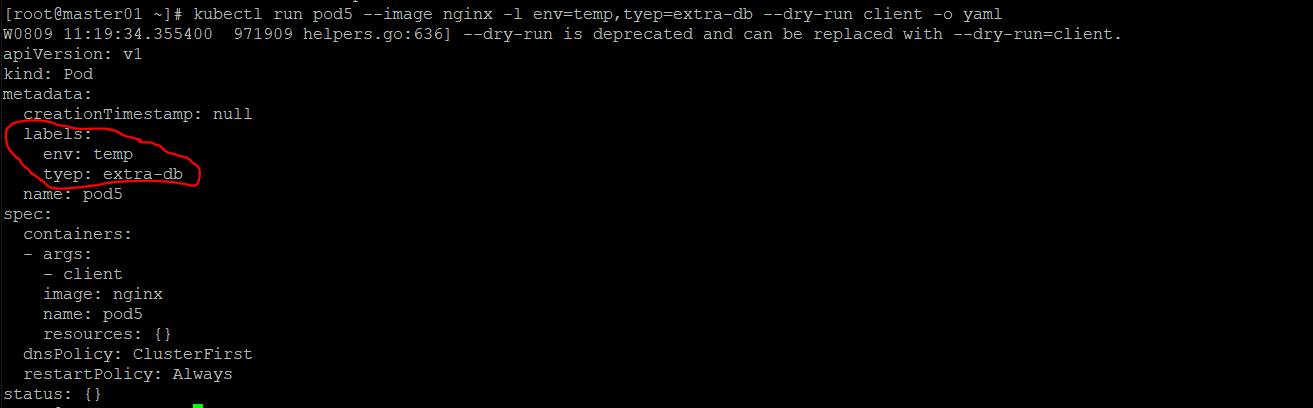
**Labels:**

For identification we use labels for pods.



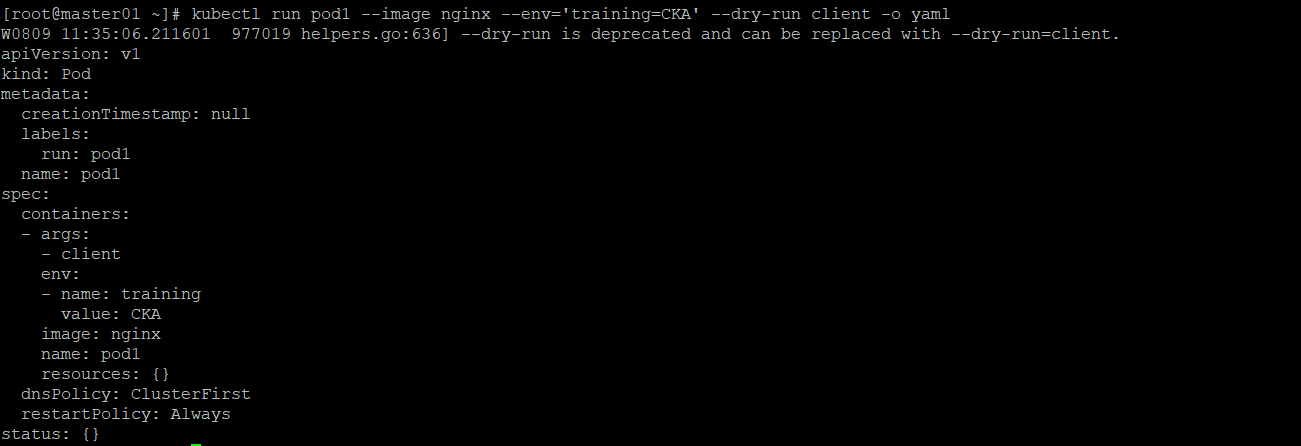


Labels in manifest file.



**Environment variables:**

We want to be providing some kind variables inside our container. We are doing this same as labels.



## **Requests and limits:**

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

For example, if you set a memory request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM.

If you set a memory limit of 4GiB for that container, the kubelet (and [container runtime](https://kubernetes.io/docs/setup/production-environment/container-runtimes)) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error.

Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions.

kubectl apply -f https://k8s.io/examples/pods/resource/cpu-request-limit.yaml --namespace=cpu-example

kubectl get pod cpu-demo --output=yaml --namespace=cpu-example

kubectl top pod cpu-demo --namespace=cpu-example

apiVersion: v1

kind: Pod

metadata:

  name: cpu-demo

  namespace: cpu-example

spec:

  containers:

  - name: cpu-demo-ctr

    image: vish/stress

    resources:

      limits:

        cpu: "1"

      requests:

        cpu: "0.5"

    args:

    - -cpus

    - "2"

**Probes (Liveness and Readiness)**

Liveness check your application is live and Readiness check your application is ready to receive production request.

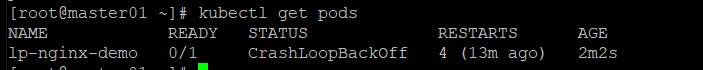
If you application is having a problem and restarting application will solve the problem this means liveness probe work, but if restarting won’t solve the problem because your application misbehaving because pendency is offline like database this check for Readiness probs.

**Liveness**

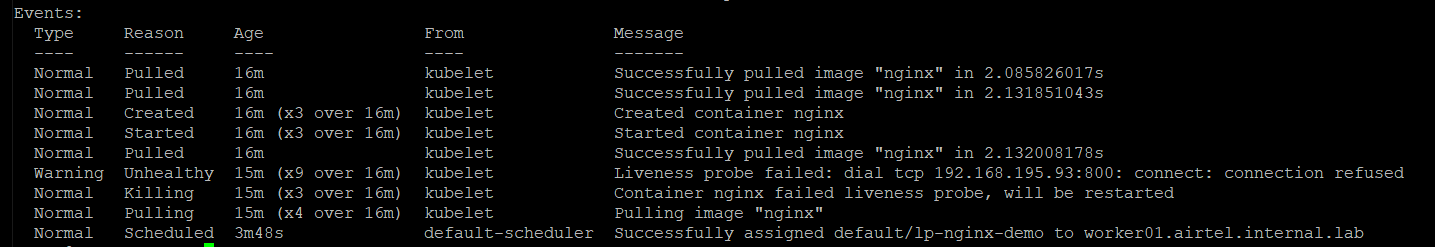
|  |
| --- |
| apiVersion: v1 |
|  | kind: Pod |
|  | metadata: |
|  | name: lp-nginx-demo |
|  | labels: |
|  | app: nginx |
|  | spec: |
|  | terminationGracePeriodSeconds: 0 |
|  | containers: |
|  | - name: nginx |
|  | image: nginx |
|  | ports: |
|  | - containerPort: 80 |
|  | livenessProbe: |
|  | # httpGet: |
|  | # path: / |
|  | # port: 80 |
|  | tcpSocket: |
|  | port: 800 |
|  | initialDelaySeconds: 5 |
|  | periodSeconds: 5 |

[root@master01 ~]# kubectl create -f https://raw.githubusercontent.com/lerndevops/educka/master/2-pods/probes/lp-nginx-pod.yml

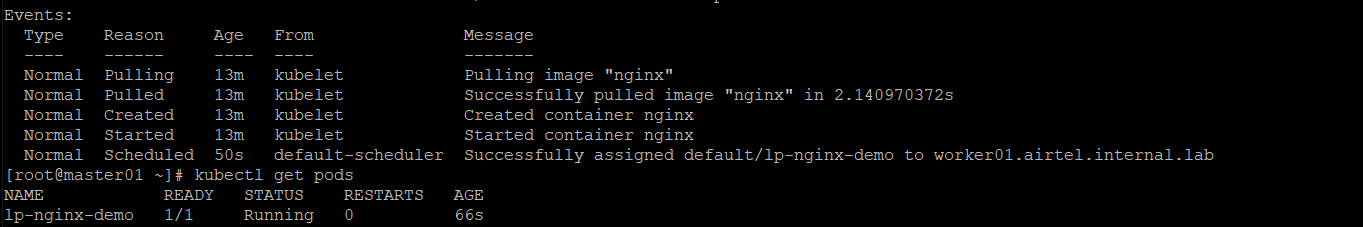
pod/lp-nginx-demo created



It is crashing pod because be give wrong port, this process gone again and again till we not rectify issue.

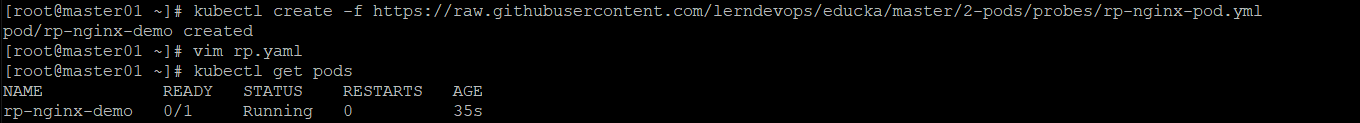


Now we provide the right port check below issue is solved.

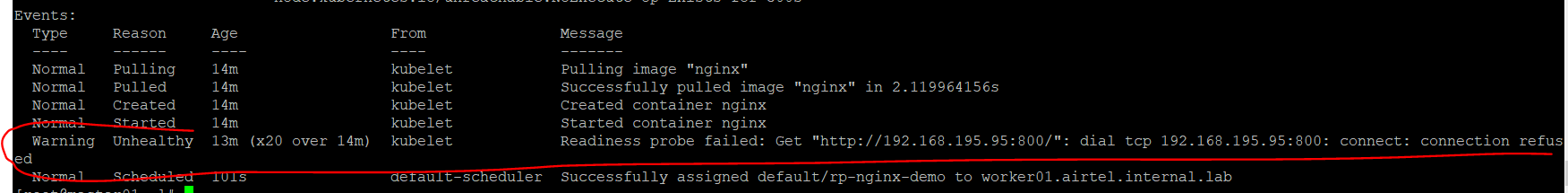


**Readiness**

Kubect create -f https://raw.githubusercontent.com/lerndevops/educka/master/2-pods/probes/rp-nginx-pod.yml



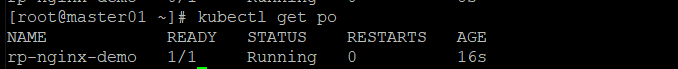
Your application is ready but it is not restarted check READY 0/1 and you “Readiness probe failed: Get "http://192.168.195.95:800/": dial tcp 192.168.195.95:800: connect: connection refused”



Liveness means your container is not live and in readiness your container will not be ready.

Liveness your container keep restarting until unless your liveness probe not passing.

In Readiness probe it won’t be ready it means if your check is failing, it means your container is not ready to solve the purpose. It will keep on waiting it will not be ready until an unless your check is pass.

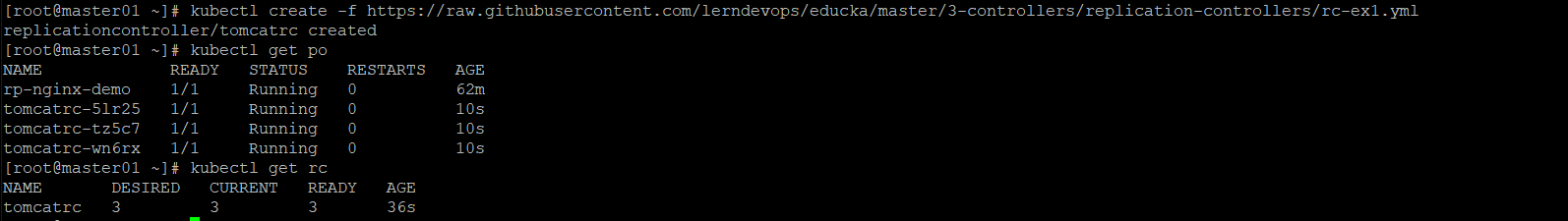


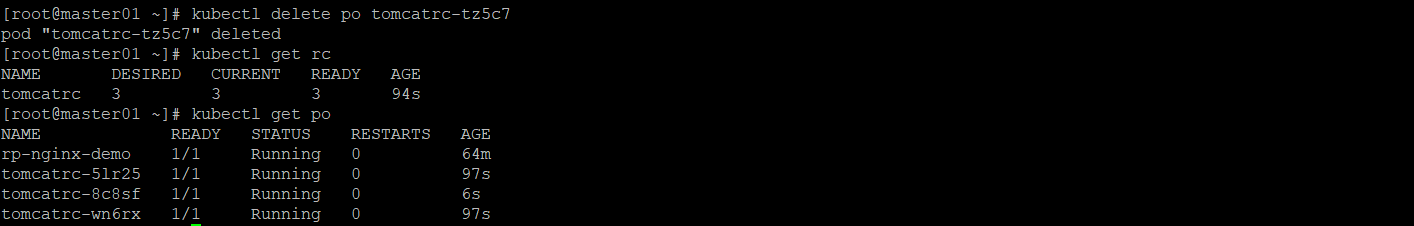
Now your container is ready 1/1 after pass the readiness probes.

**Controllers (Replication Controller and ReplicaSet)**

**Replication Controller:-** It is obsolete technology in k8s

If your pod is corrupted or container get delete with any region, the replication controller manage the state.





## **How a ReplicationController Works**

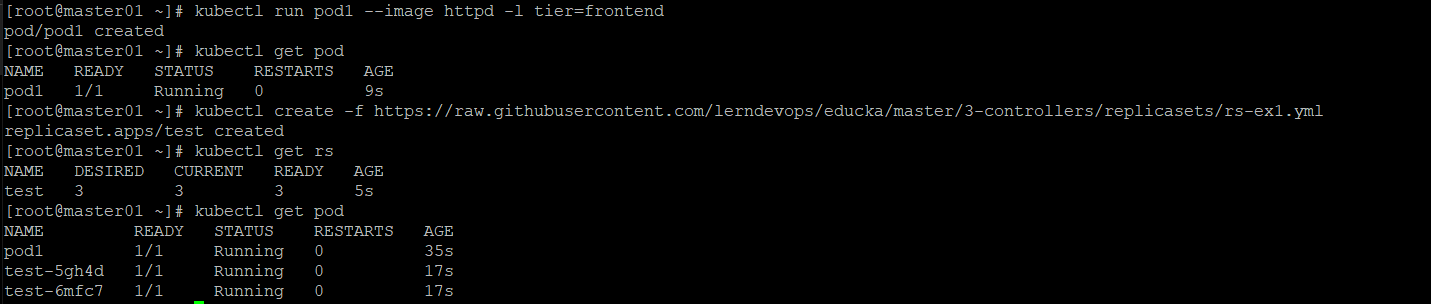
## the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated. For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade. For this reason, you should use a ReplicationController even if your application requires only a single pod. A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

**Challenges in RC:-** If you create 4 pods and one pod also running with same details, it is not counting this running pod this is the drawback of RC, it create 4 new pods no matter what you running. In ReplicaSet if you create 4 pods and one pod running with same configuration then it create 3 new pod and total is 4.

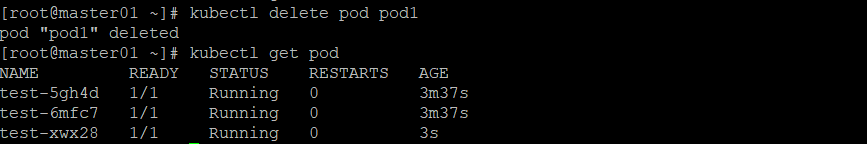
## **Alternatives to ReplicationController**

### ReplicaSet

[ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) is the next-generation ReplicationController that supports the new [set-based label selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#set-based-requirement). It's mainly used by [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) as a mechanism to orchestrate pod creation, deletion and updates. Note that we recommend using Deployments instead of directly using Replica Sets, unless you require custom update orchestration or don't require updates at all.



You can see we create one pod with same label after that we create ReplicaSet pod and it create 3 RS, also check pods you can see it create 2 new pod and use existing 1 pod as well (also see selector and labels).



You can see also If we delete old pod it automatically create new RS with same name.

In ReplicaSet you can scale it in real time.

**Deployment**

Whenever we create any deployment it automatic create ReplicaSet in backend.

You describe a *desired state* in a Deployment, and the Deployment [Controller](https://kubernetes.io/docs/concepts/architecture/controller/) changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

**Use Case**

**The following are typical use cases for Deployments:**

* [Create a Deployment to rollout a ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment). The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
* [Declare the new state of the Pods](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment) by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
* [Rollback to an earlier Deployment revision](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment) if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
* [Scale up the Deployment to facilitate more load](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment).
* [Pause the rollout of a Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#pausing-and-resuming-a-deployment) to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
* [Use the status of the Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#deployment-status) as an indicator that a rollout has stuck.
* [Clean up older ReplicaSets](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#clean-up-policy) that you don't need anymore.

**Services :** services are cluster level resource.

If you want to access you application publicly then use below command for nodeport

[root@master01 ~]# kubectl run pp1 --image nginx

pod/pp1 created

[root@master01 ~]# kubectl expose pod pp1 --name pp1svc --port 80 --target-port 80 --type NodePort

service/pp1svc exposed

[root@master01 ~]# kubectl get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

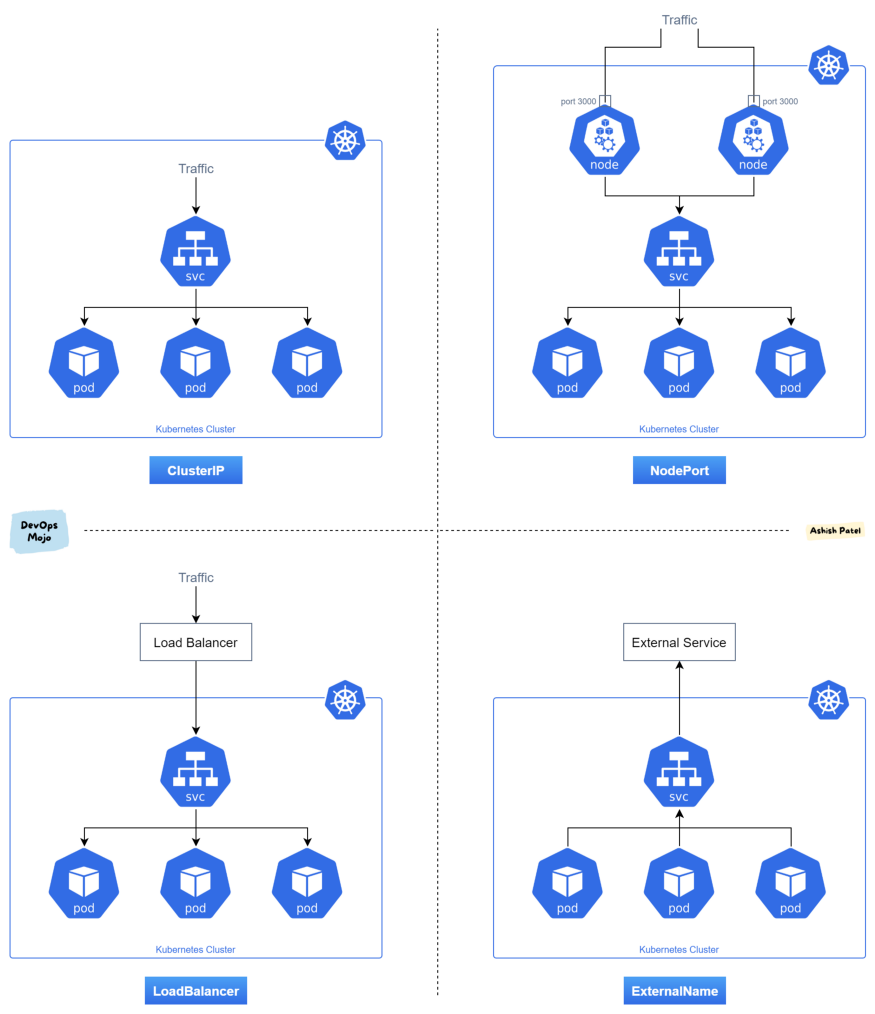
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 4m49s

pp1svc NodePort 10.103.125.25 <none> 80:30501/TCP 42s

kubectl create -f <https://raw.githubusercontent.com/lerndevops/educka/master/3-controllers/replicasets/rs-ex2.yml> (for NodePort)

**Service Types Overview**

Introduction to Service types in K8s — Types of Kubernetes Services.



Kubernetes — Service Types

## TL;DR

There are four types of Kubernetes services — ClusterIP, NodePort, LoadBalancer and ExternalName. The type property in the Service's spec determines how the service is exposed to the network.

Read about Kubernetes [Services](https://medium.com/devops-mojo/kubernetes-services-overview-k8s-service-introduction-why-and-what-are-kubernetes-services-how-works-e6fd4fd4a51a) and [Ingress](https://medium.com/devops-mojo/kubernetes-ingress-overview-what-is-kubernetes-ingress-introduction-to-k8s-ingress-b0f81525ffe2)

**ClusterIP**

* ClusterIP is the default and most common service type.
* Kubernetes will assign a cluster-internal IP address to ClusterIP service. This makes the service only reachable within the cluster.
* You cannot make requests to service (pods) from outside the cluster.
* You can optionally set cluster IP in the service definition file.

## Use Cases

* Inter service communication within the cluster. For example, communication between the front-end and back-end components of your app.

Example:

|  |
| --- |
| apiVersion: v1 |
|  | kind: Service |
|  | metadata: |
|  | name: my-backend-service |
|  | spec: |
|  | type: ClusterIP # Optional field (default) |
|  | clusterIP: 10.10.0.1 # within service cluster ip range |
|  | ports: |
|  | - name: http |
|  | protocol: TCP |
|  | port: 80 |
|  | targetPort: 8080 |

vim svc.yaml

|  |
| --- |
| apiVersion: v1 |
|  | kind: Service |
|  | metadata: |
|  | name: samplesvc |
|  | labels: |
|  | app: frontend |
|  | spec: |
|  | type: NodePort (Remove NodePort,if you want to run this service as cluster) |
|  | ports: |
|  | - port: 3000 |
|  | # nodePort: 30001 #expose , if dont give takes default |
|  | protocol: TCP |
|  | selector: |
|  | tier: frontend |

[root@master01 ~]# kubectl describe svc samplesvc01

Name: samplesvc01

Namespace: default

Labels: app=frontend

Annotations: <none>

Selector: tier=frontend

Type: ClusterIP

IP Family Policy: SingleStack

IP Families: IPv4

IP: 10.102.12.203

IPs: 10.102.12.203

Port: <unset> 3000/TCP

TargetPort: 3000/TCP

Endpoints: 192.168.195.112:3000,192.168.195.113:3000,192.168.72.188:3000

Session Affinity: None

Events: <none>

In Cluster, your service does not expose to the external world, with the help of cluster IP your app pod will connect with your DB pods.

[root@master01 ~]# kubectl describe svc samplesvc

Name: samplesvc

Namespace: default

Labels: app=frontend

Annotations: <none>

Selector: tier=frontend

Type: NodePort

IP Family Policy: SingleStack

IP Families: IPv4

IP: 10.96.141.245

IPs: 10.96.141.245

Port: <unset> 3000/TCP

TargetPort: 3000/TCP

NodePort: <unset> 30601/TCP

Endpoints: 192.168.195.112:3000,192.168.195.113:3000,192.168.72.188:3000

Session Affinity: None

External Traffic Policy: Cluster

Events: <none>

Kubernetes ClusterIP Service

App02

App03

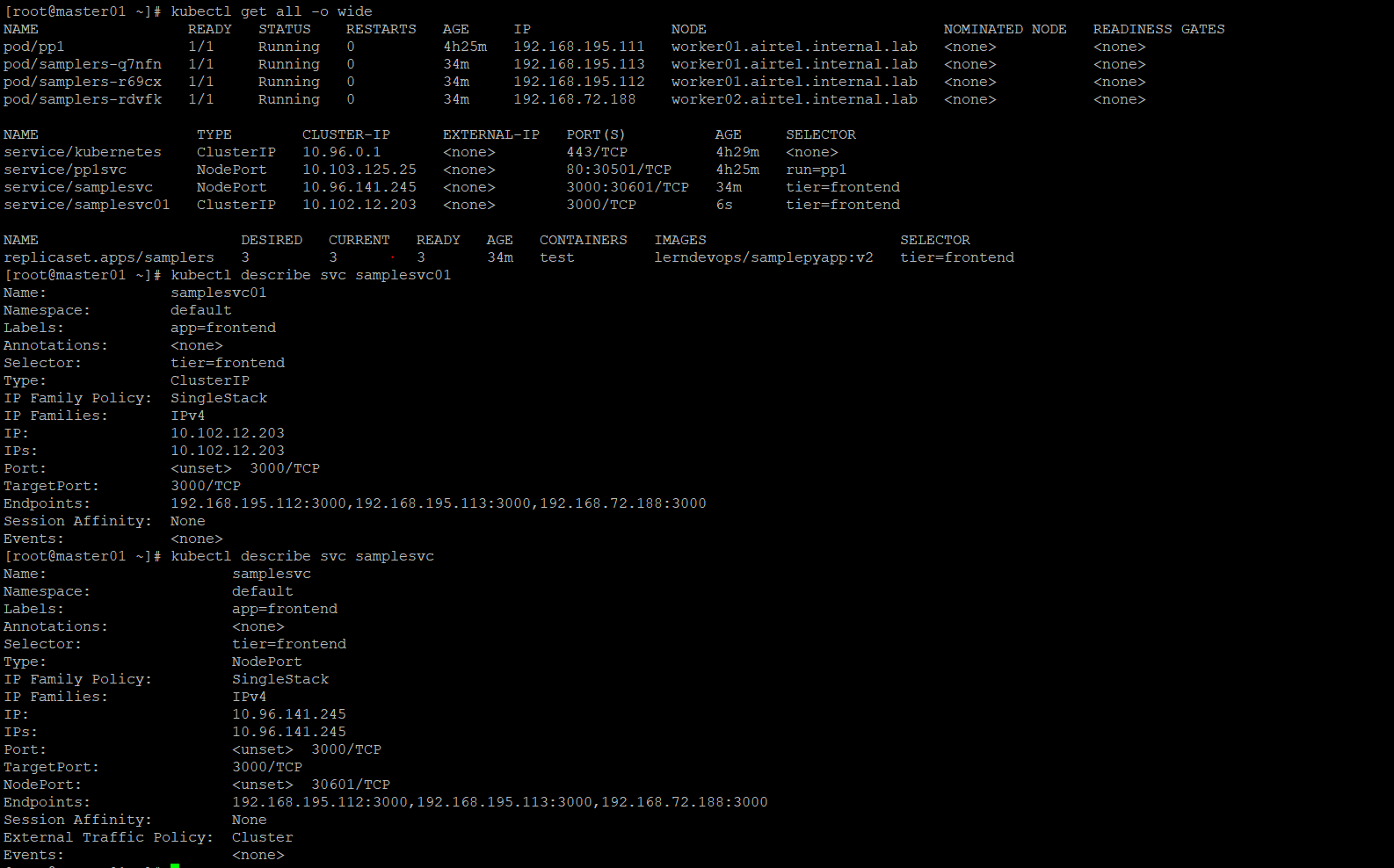
App01

ClusterIP 10.102.12.203

DB02

DB03

DB01



**NodePort**

* NodePort service is an extension of ClusterIP service. A ClusterIP Service, to which the NodePort Service routes, is automatically created.
* It exposes the service outside of the cluster by adding a cluster-wide port on top of ClusterIP.
* NodePort exposes the service on each Node’s IP at a static port (the NodePort). Each node proxies that port into your Service. So, external traffic has access to fixed port on each Node. It means any request to your cluster on that port gets forwarded to the service.
* You can contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
* Node port must be in the range of 30000–32767. Manually allocating a port to the service is optional. If it is undefined, Kubernetes will automatically assign one.
* If you are going to choose node port explicitly, ensure that the port was not already used by another service.

## Use Cases

* When you want to enable external connectivity to your service.
* Using a NodePort gives you the freedom to set up your own load balancing solution, to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes’ IPs directly.
* Prefer to place a load balancer above your nodes to avoid node failure.

## Example

|  |
| --- |
| apiVersion: v1 |
|  | kind: Service |
|  | metadata: |
|  | name: my-frontend-service |
|  | spec: |
|  | type: NodePort |
|  | selector: |
|  | app: web |
|  | ports: |
|  | - name: http |
|  | protocol: TCP |
|  | port: 80 |
|  | targetPort: 8080 |
|  | nodePort: 30000 # 30000-32767, Optional field |

Kubernetes NodePort Service

**LoadBalancer**

* LoadBalancer service is an extension of NodePort service. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
* It integrates NodePort with cloud-based load balancers.
* It exposes the Service externally using a cloud provider’s load balancer.
* Each cloud provider (AWS, Azure, GCP, etc) has its own native load balancer implementation. The cloud provider will create a load balancer, which then automatically routes requests to your Kubernetes Service.
* Traffic from the external load balancer is directed at the backend Pods. The cloud provider decides how it is load balanced.
* The actual creation of the load balancer happens asynchronously.
* Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.

## Use Cases

* When you are using a cloud provider to host your Kubernetes cluster.

This type of service is typically heavily dependent on the cloud provider.

## Example

|  |
| --- |
| apiVersion: v1 |
|  | kind: Service |
|  | metadata: |
|  | name: my-frontend-service |
|  | spec: |
|  | type: LoadBalancer |
|  | clusterIP: 10.0.171.123 |
|  | loadBalancerIP: 123.123.123.123 |
|  | selector: |
|  | app: web |
|  | ports: |
|  | - name: http |
|  | protocol: TCP |
|  | port: 80 |
|  | targetPort: 8080 |

Kubernetes LoadBalancer Service

**ExternalName**

* Services of type ExternalName map a Service to a DNS name, not to a typical selector such as my-service.
* You specify these Services with the `spec.externalName` parameter.
* It 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 established.

## Use Cases

* This is commonly used to create a service within Kubernetes to represent an external datastore like a database that runs externally to Kubernetes.
* You can use that ExternalName service (as a local service) when Pods from one namespace to talk to a service in another namespace.

## Example

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

Kubernetes ExternalName Service

**Ingress**

You can also use [Ingress](https://kubernetes.io/docs/concepts/services-networking/ingress/) to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. It lets you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address.

## **Deployments**

A Deployment provides declarative(rolling) updates for Pods and ReplicaSets.

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.

You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

### Use Case

**The following are typical use cases for Deployments:**

1. Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
2. Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
3. Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
4. Scale up the Deployment to facilitate more load.
5. Pause the rollout of a Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
6. Use the status of the Deployment as an indicator that a rollout has stuck.
7. Clean up older ReplicaSets that you don't need anymore.

[root@master01 ~]# kubectl create deployment pp1 --image nginx

deployment.apps/pp1 created

[root@master01 ~]# kubectl get rs

NAME DESIRED CURRENT READY AGE

pp1-744df99f5c 1 1 1 24s

[root@master01 ~]# kubectl scale deploy pp1 --replicas=3

deployment.apps/pp1 scaled

[root@master01 ~]# kubectl get pod

NAME READY STATUS RESTARTS AGE

pp1-744df99f5c-dtfm4 1/1 Running 0 3m1s

pp1-744df99f5c-k9pzr 1/1 Running 0 4m42s

pp1-744df99f5c-xls6z 1/1 Running 0 3m1s

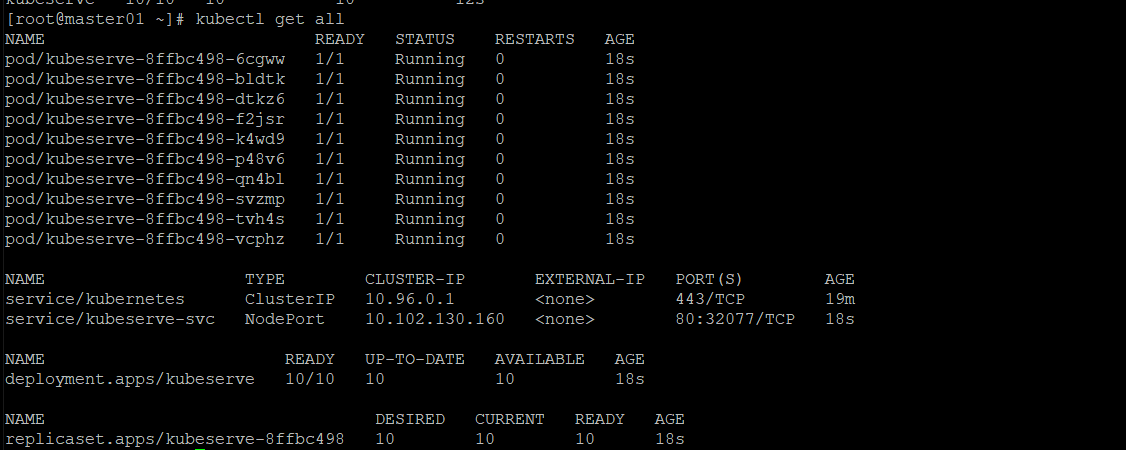
(First one is deployment name, 2nd one is replicaset name, 3rd in pod name)

## **Deployments**

**Rolling updates**

With this function we are updating running application, at the time only container unavailable

Kubectl create -f <https://raw.githubusercontent.com/lerndevops/educka/master/3-controllers/deployments/deployment-ex3.yml>



[root@master01 ~]# kubectl set image deploy kubeserve app=leaddevops/kubeserve:v2

deployment.apps/kubeserve image updated

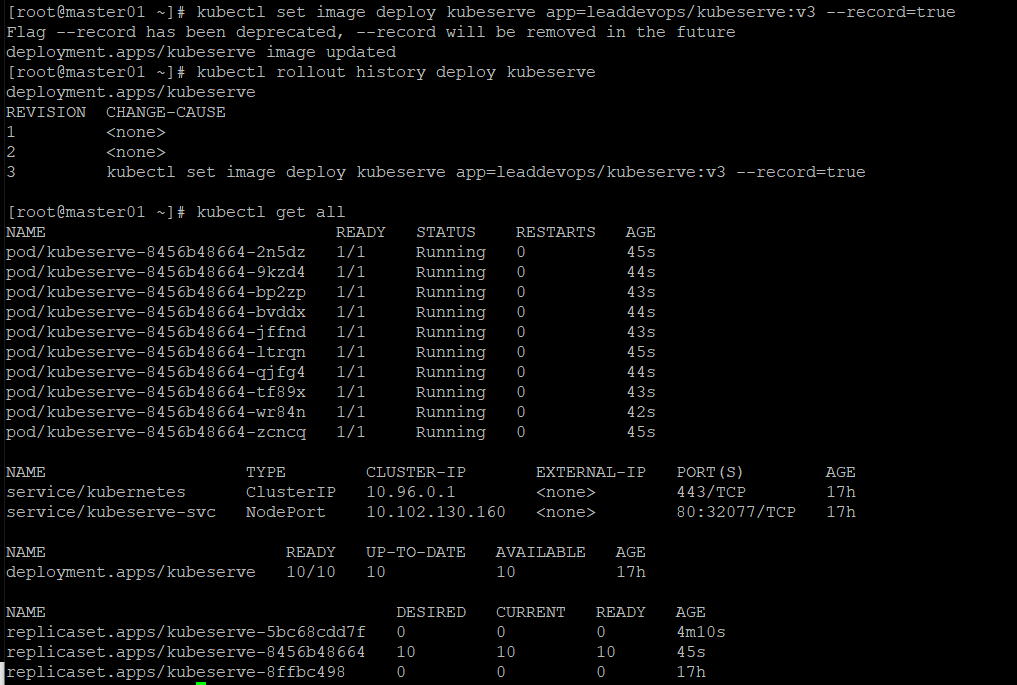
[root@master01 ~]# kubectl rollout history deploy kubeserve

deployment.apps/kubeserve

REVISION CHANGE-CAUSE

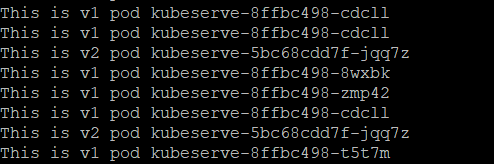
1 <none>

2 <none>



[root@master01 ~]# kubectl rollout undo deployment kubeserve --to-revision=2

deployment.apps/kubeserve rolled back



## **Daemonset**

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod.

As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected.

Deleting a DaemonSet will clean up the Pods it created.

#### Some typical uses of a DaemonSet are:

1. running a cluster storage daemon on every node
2. running a logs collection daemon on every node
3. running a node monitoring daemon on every node

[root@master01 ~]# kubectl create -f https://raw.githubusercontent.com/lerndevops/educka/master/3-controllers/daemonsets/ds-ex1.yml

daemonset.apps/mydaemonset created

[root@master01 ~]# kubectl get ds

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

mydaemonset 2 2 2 2 2 <none> 16s

**Jobs and CronJobs**

Kubernetes Jobs ensure that one or more pods execute their commands and exit successfully.

When all the pods have exited without errors, the Job gets completed.

When the Job gets deleted, any created pods get deleted as well.

A CronJob creates Jobs on a repeating schedule.

One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in Cron format.

**Schedular**

# kubernetes-multiple-scheduler

The Kubernetes scheduler is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity. The scheduler needs to take into account individual and collective resource requirements, quality of service requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, deadlines, and so on. Workload-specific requirements will be exposed through the API as necessary.

Kubernetes ships with its default scheduler. Please refer to the architecture discussion about what scheduler does.

The source code for kubernetes defaule scheduler is - <https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler>

We will take reference from the official kubernetes documentation to implement a second scheduler which will be a replica of the default scheduler. We will name the new scheduler as - **myscheduler** adn the default scheduler will be named as **default-scheduler**. The original demo is referenced at - <https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/>

**Note** that for this demo we will be building kubernetes on the master node. It is highly recommended that you have atleast 6 CPU, 15 GB of memory and 30 GB of storage on the node where you will execute the **make** command

Let’s start off by

* Build kubernetes from source

## **Node Selection** (nodeSelector)

nodeSelector is the simplest recommended form of node selection constraint. You can add the nodeSelector field to your Pod specification and specify the [node labels](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#built-in-node-labels) you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.

You can use any of the following methods to choose where Kubernetes schedules specific Pods:

* [nodeSelector](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodeselector) field matching against [node labels](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#built-in-node-labels)
* [Affinity and anti-affinity](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity)
* [nodeName](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodename) field
* [Pod topology spread constraints](https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#pod-topology-spread-constraints)

## Ex: nodeSelector

|  |
| --- |
| apiVersion: apps/v1 |
|  | kind: Deployment |
|  | metadata: |
|  | name: kubeserve |
|  | spec: |
|  | replicas: 2 |
|  | selector: |
|  | matchLabels: |
|  | app: kubeserve |
|  | template: |
|  | metadata: |
|  | name: kubeserve |
|  | labels: |
|  | app: kubeserve |
|  | spec: |
|  | nodeSelector: |
|  | color: "red" |
|  | containers: |
|  | - image: leaddevops/kubeserve:v1 |
|  | name: app |

[root@master01 ~]# kubectl label node worker02.airtel.internal.lab color=red

node/worker02.airtel.internal.lab labeled

[root@master01 ~]# kubectl get po -o wide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES

kubeserve-5f469ccdb9-7hwpk 1/1 Running 0 83s 192.168.72.157 worker02.airtel.internal.lab <none> <none>

kubeserve-5f469ccdb9-m2qlv 1/1 Running 0 83s 192.168.72.159 worker02.airtel.internal.lab <none> <none>

## Ex: nodeName

|  |
| --- |
| apiVersion: apps/v1 |
|  | kind: Deployment |
|  | metadata: |
|  | name: kubeserve |
|  | spec: |
|  | replicas: 2 |
|  | selector: |
|  | matchLabels: |
|  | app: kubeserve |
|  | template: |
|  | metadata: |
|  | name: kubeserve |
|  | labels: |
|  | app: kubeserve |
|  | spec: |
|  | nodeName: worker02.airtel.internal.lab |
|  | containers: |
|  | - image: leaddevops/kubeserve:v1 |
|  | name: app |

**Affinity and anti-affinity**

# **Node and Pod Affinity and Anti-Affinity**

As we’ve mentioned earlier, nodeSelector is the simplest Pod scheduling constraint in Kubernetes. In Kubernetes 1.2, a new node and Pod affinity feature was added as alpha and graduated to beta in Kubernetes 1.6. The affinity greatly expands the nodeSelector functionality introducing the following improvements:

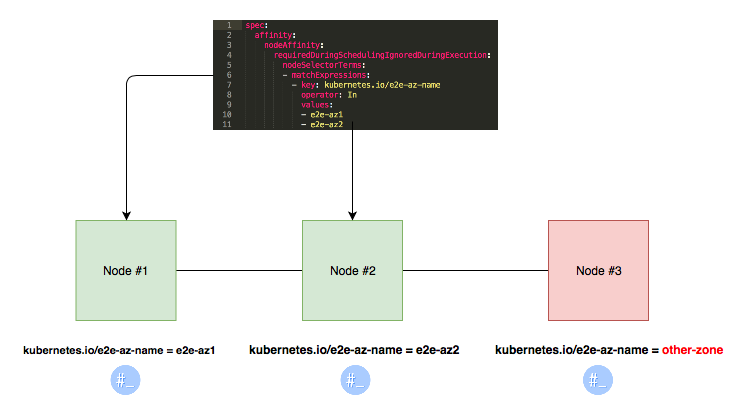
1. Affinity language is more expressive (more logical operators to control how Pods are scheduled).
2. Users can now “soft” scheduling rules. If the “soft” rule is not met, the scheduler can still schedule a Pod onto a specific node.
3. New affinity feature supports Pod co-location. Users can constraint a Pod against the label of another Pod running on a specific node rather than against node labels. With this feature, users can control what Pods end up on the same node and which one don’t. This feature is called “inter-Pod affinity/anti-affinity.”

Essentially, there are two types of affinity in Kubernetes: *node* affinity and *Pod* affinity. Similarly to nodeSelector, node affinity attracts a Pod to certain nodes, whereas the Pod affinity attracts a Pod to certain Pods. In addition to that, Kubernetes supports Pod anti-affinity, which repels a Pod from other Pods. Users can also implement node anti-affinity using logical operators. In what follows, we discuss node and Pod affinity/anti-affinity in more detail.

# **Node Affinity**

Node affinity allows scheduling Pods to specific nodes. There are a number of use cases for node affinity, including the following:

* Spreading Pods across different availability zones to improve resilience and availability of applications in the cluster (see the image below).
* Allocating nodes for memory-intensive Pods. In this case, you can have a few nodes dedicated to less compute-intensive Pods and one or two nodes with enough CPU and RAM dedicated to memory-intensive Pods. This way you prevent undesired Pods from consuming resources dedicated to other Pods.



One of the best features of the current affinity implementation in Kubernetes is the support for “hard” and “soft” node affinity.

With “hard” affinity, users can set a precise rule that should be met in order for a Pod to be scheduled on a node. For example, using “hard” affinity you can tell the scheduler to run the Pod only on the node that has SSD storage attached to it.

As the name suggests, “soft” affinity is less strict. Using “soft” affinity, you can ask the scheduler to try to run the set of Pod in availability zone XYZ, but if it’s impossible, allow some of these Pods to run in the other Availability Zone.

The image above demonstrates the “hard” node affinity. As you see in the manifest, it is defined by the requiredDuringSchedulingIgnoredDuringExecution field of the PodSpec. In its turn, “soft” node affinity is defined by the preferredDuringSchedulingIgnoredDuringExecution field.

**Note**: IgnoredDuringExecution part in both names means that if labels on a node will be changed after the Pod matching these labels is scheduled, it will still continue to run on that node. There is still an unimplemented requiredDuringSchedulingRequiredDuringExecution feature that allows evicting Pods from nodes that no longer satisfy the Pod’s node affinity requirements.

To get the feel of how node affinity works, let’s look at this example:

apiVersion: v1  
kind: Pod  
metadata:  
 name: with-node-affinity  
spec:  
 affinity:  
 nodeAffinity:  
 requiredDuringSchedulingIgnoredDuringExecution:  
 nodeSelectorTerms:  
 - matchExpressions:  
 - key: kubernetes.io/e2e-az-name  
 operator: In  
 values:  
 - e2e-az1  
 - e2e-az2  
 preferredDuringSchedulingIgnoredDuringExecution:  
 - weight: 1  
 preference:  
 matchExpressions:  
 - key: custom-key  
 operator: In  
 values:  
 - custom-value  
 containers:  
 - name: with-node-affinity  
 image: k8s.gcr.io/pause:2.0

As you see, the node affinity is specified under nodeAffinity field of the affinity in the PodSpec. We define “hard” and “soft” affinity rules in the same PodSpec.

The “hard” affinity rule tells the scheduler to place the Pod only onto a node with the label whose key is kubernetes.io/e2e-az-name and the value is either e2e-az1 or e2e-az2 .

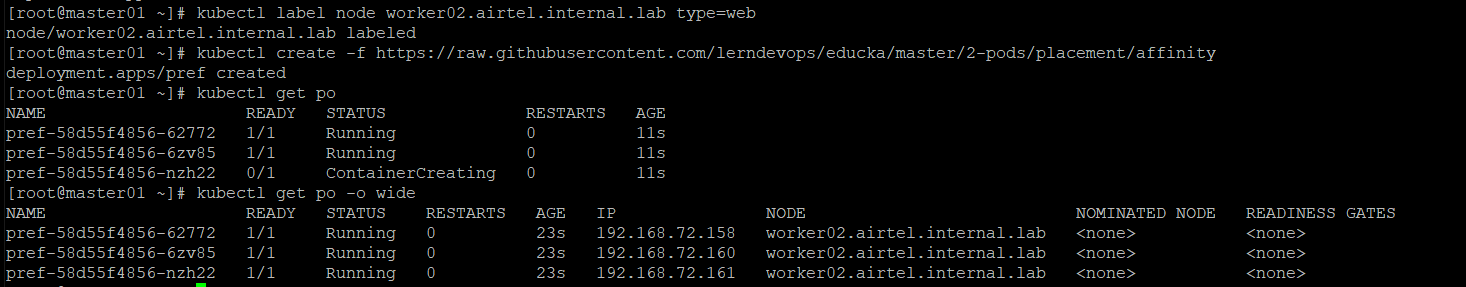
The “soft” rule says that among the nodes that meet “hard” criteria, we prefer the Pod to be placed on the nodes that have a label with the key custom-key and value custom-value. Because this rule is “soft,” if there are no such nodes, the Pod will still be scheduled if the “hard” rule is met.

In the example above, we used the In operator in matchExpressions. This operator matches the key’s value if it includes the “expression” specified. Other supported operators include NotIn , Exists , DoesNotExist , Gt , Lt By the way, you can use NotIn and DoesNotExist to achieve node anti-affinity behavior.

**Ex: nodeAffinity**

|  |
| --- |
| apiVersion: apps/v1 |
|  | kind: Deployment |
|  | metadata: |
|  | name: pref |
|  | spec: |
|  | selector: |
|  | matchLabels: |
|  | app: pref |
|  | replicas: 3 |
|  | template: |
|  | metadata: |
|  | labels: |
|  | app: pref |
|  | spec: |
|  | affinity: |
|  | nodeAffinity: |
|  | preferredDuringSchedulingIgnoredDuringExecution: |
|  | - weight: 80 |
|  | preference: |
|  | matchExpressions: |
|  | - key: zone # assign this label to node |
|  | operator: In |
|  | values: |
|  | - east |
|  | #- west we can add multiple values too |
|  | - weight: 20 |
|  | preference: |
|  | matchExpressions: |
|  | - key: type # assign this label to node |
|  | operator: In |
|  | values: |
|  | - web |
|  | containers: |
|  | - args: |
|  | - sleep |
|  | - "99999" |
|  | image: busybox |
|  | name: main |

Note: requiredDuringSchedulingIgnoredDuringExecution: ->It has to full fill given condition.



# **Taints and Tolerations**

## Taints and Tolerations – Concepts

Taints and tolerations are a mechanism that allows you to ensure that pods are not placed on inappropriate nodes. Taints are added to nodes, while tolerations are defined in the pod specification. When you taint a node, it will repel all the pods except those that have a toleration for that taint. A node can have one or many taints associated with it.

For example, most Kubernetes distributions will automatically taint the master nodes so that one of the pods that manages the control plane is scheduled onto them and not any other data plane pods deployed by users. This ensures that the master nodes are dedicated to run control plane pods.

A taint can produce three possible effects:

**NoSchedule**

The Kubernetes scheduler will only allow scheduling pods that have tolerations for the tainted nodes.

**PreferNoSchedule**

The Kubernetes scheduler will try to avoid scheduling pods that don’t have tolerations for the tainted nodes.

**NoExecute**

Kubernetes will evict the running pods from the nodes if the pods don’t have tolerations for the tainted nodes.

Impact of a taint and tollerations on a K8s cluster

## Use Cases for Taints and Tolerations

### Dedicated Nodes

If you need to dedicate a group of worker nodes for a set of users, you can add a taint to those nodes, such as by using this command:

kubectl taint nodes nodename dedicated=groupName:NoSchedule

Then add tolerations of the taint in that user group’s pods so they can run on those nodes. To further ensure that pods only get scheduled on that set of tainted nodes, you can also add a label to those nodes, e.g., dedicated=groupName. Then use NodeSelector in the deployment/pod spec, which will make sure that pods from the user group are bound to the node group and don’t run anywhere else.

### Nodes with Special Hardware

If there are worker nodes with special hardware, you need to make sure that normal pods that don’t need the special hardware don’t run on those worker nodes. Do this by adding a taint to those nodes as follows:

kubectl taint nodes nodename special=true:NoSchedule

Later on, the pods requiring special hardware can be run on those worker nodes by adding tolerations for the above taint.

### Taint-Based Evictions

A taint with the NoExecute effect will evict the running pod from the node if the pod has no tolerance for the taint. The Kubernetes node controller will automatically add this kind of taint to a node in some scenarios so that pods can be evicted immediately and the node is “drained” (have all of its pods evicted). For example, suppose a network outage causes a node to be unreachable from the controller. In this scenario, it would be best to move all of the pods off the node so that they can get rescheduled to other nodes. The node controller takes this action automatically to avoid the need for manual intervention.

The following are built-in taints:

**node.kubernetes.io/not-ready**

Node is not ready. This corresponds to the NodeCondition Ready attribute being "False".

**node.kubernetes.io/unreachable**

Node is unreachable from the node controller. This corresponds to NodeCondition Ready being "Unknown".

**node.kubernetes.io/memory-pressure**

Node has memory pressure.

**node.kubernetes.io/disk-pressure**

Node has disk pressure. In case of High disk utilization on nodes, it can cause slowness for application so its better to relocate pods.

**node.kubernetes.io/pid-pressure**

Node has PID pressure. Process ID is a limited resource and its saturation can cause down time for applications, so better to relocate pods to somewhere else.

**node.kubernetes.io/network-unavailable**

Node's network is unavailable. As explained above.

**node.kubernetes.io/unschedulable**

Node is unschedulable. Any other reason that will make the node inappropriate for hosting pods, for example if the cluster is being scaled down and the node is being removed.

## How to Use Taints and Tolerations

We will now present a scenario to help you better understand taints and tolerations. Let’s start with a Kubernetes cluster that has worker nodes categorized into different groups, such as front-end nodes and back-end nodes. Let’s assume that we need to deploy the front-end application pods so that they are placed only on front-end nodes and not back-end nodes. We also must ensure that new pods are not scheduled into master nodes because those nodes run control plane components such as etcd.

Let’s start by getting the list of nodes to see what is already tainted by the Kubernetes default installation. Here we are on a cluster created by the Rancher RKE tool.

kubectl get nodes -o=custom-columns=NodeName:.metadata.name,TaintKey:.spec.taints[\*].key,TaintValue:.spec.taints[\*].value,TaintEffect:.spec.taints[\*].effect

NodeName TaintKey TaintValue TaintEffect

cluster01-master-1 node-role.kubernetes.io/controlplane,node-role.kubernetes.io/etcd true,true NoSchedule,NoExecute

cluster01-master-2 node-role.kubernetes.io/controlplane,node-role.kubernetes.io/etcd true,true NoSchedule,NoExecute

cluster01-master-3 node-role.kubernetes.io/controlplane,node-role.kubernetes.io/etcd true,true NoSchedule,NoExecute

cluster01-worker-1 <none> <none> <none>

From the output above, we noticed that the master nodes are already tainted by the Kubernetes installation so that no user pods land on them until intentionally configured by the user to be placed on master nodes by adding tolerations for those taints. The output also shows a worker node that has no taints. We will now taint the worker so that only front-end pods can land on it. We can do this by using the kubectl taint command.

kubectl taint nodes cluster01-worker-1 app=frontend:NoSchedule

node/cluster01-worker-1 tainted

The above taint has a key name app, with a value frontend, and has the effect of NoSchedule, which means that no pod will be placed on this node until the pod has defined a toleration for the taint. We will see what the toleration looks like in later steps.

Let’s try to deploy an app on the cluster without any toleration configured in the app deployment specification.

kubectl create ns frontend

namespace/frontend created

kubectl run nginx --image=nginx --namespace frontend

deployment.apps/nginx created

kubectl get pods -n frontend

NAME READY STATUS RESTARTS AGE

nginx-76df748b9-gjbs4 0/1 Pending 0 9s

kubectl get events -n frontend

LAST SEEN TYPE REASON OBJECT MESSAGE

<unknown> Warning FailedScheduling pod/nginx-76df748b9-gjbs4 0/4 nodes are available: 1 node(s) had taint {app: frontend}, that the pod didn't tolerate, 3 node(s) had taint {node-role.kubernetes.io/controlplane: true}, that the pod didn't tolerate.

<unknown> Warning FailedScheduling pod/nginx-76df748b9-gjbs4 0/4 nodes are available: 1 node(s) had taint {app: frontend}, that the pod didn't tolerate, 3 node(s) had taint {node-role.kubernetes.io/controlplane: true}, that the pod didn't tolerate.

We created a namespace and deployed Nginx using the kubectl run command, but looking at the pod status and cluster events, we see that the pod can’t be scheduled because there are no appropriate worker nodes. Three master nodes have taints that the pod didn’t tolerate and one worker node has a taint that the pod doesn’t tolerate. To successfully place the pod on the worker node, we need to edit the deployment and add a toleration of the taint we configured earlier on the node.

Let’s see what the current deployment YAML looks like.

kubectl get deployment nginx -n frontend -o yaml

apiVersion: apps/v1

kind: Deployment

metadata:

annotations:

deployment.kubernetes.io/revision: "1"

creationTimestamp: "2021-08-29T09:39:37Z"

generation: 1

labels:

run: nginx

name: nginx

namespace: frontend

resourceVersion: "13367313"

selfLink: /apis/apps/v1/namespaces/frontend/deployments/nginx

uid: e46e026e-3a92-4aac-b985-7110426aa437

spec:

progressDeadlineSeconds: 600

replicas: 1

revisionHistoryLimit: 10

selector:

matchLabels:

run: nginx

strategy:

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

type: RollingUpdate

template:

metadata:

creationTimestamp: null

labels:

run: nginx

spec:

containers:

- image: nginx

imagePullPolicy: Always

name: nginx

resources: {}

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

dnsPolicy: ClusterFirst

restartPolicy: Always

schedulerName: default-scheduler

securityContext: {}

terminationGracePeriodSeconds: 30

From the output above, we can see that there is no toleration added in the pod spec. Let’s edit and add one.

kubectl edit deployment nginx -n frontend

deployment.apps/nginx edited

kubectl get deployment nginx -n frontend -o yaml

apiVersion: apps/v1

kind: Deployment

metadata:

annotations:

deployment.kubernetes.io/revision: "3"

creationTimestamp: "2021-08-29T09:39:37Z"

generation: 3

labels:

run: nginx

name: nginx

namespace: frontend

resourceVersion: "13368509"

selfLink: /apis/apps/v1/namespaces/frontend/deployments/nginx

uid: e46e026e-3a92-4aac-b985-7110426aa437

spec:

progressDeadlineSeconds: 600

replicas: 1

revisionHistoryLimit: 10

selector:

matchLabels:

run: nginx

strategy:

rollingUpdate:

maxSurge: 25%

maxUnavailable: 25%

type: RollingUpdate

template:

metadata:

creationTimestamp: null

labels:

run: nginx

spec:

containers:

- image: nginx

imagePullPolicy: Always

name: nginx

resources: {}

terminationMessagePath: /dev/termination-log

terminationMessagePolicy: File

dnsPolicy: ClusterFirst

restartPolicy: Always

schedulerName: default-scheduler

securityContext: {}

terminationGracePeriodSeconds: 30

tolerations:

- effect: NoSchedule

key: app

operator: Equal

value: frontend

Notice the tolerations section of the pod spec: We have added a toleration for the taint so that the pod can be scheduled on the worker node.

Now let’s get the pod’s status and events.

kubectl get events -n frontend

LAST SEEN TYPE REASON OBJECT MESSAGE

3m56s Normal SuccessfulCreate replicaset/nginx-9cf9fd78f Created pod: nginx-9cf9fd78f-khc5z

2s Normal SuccessfulDelete replicaset/nginx-9cf9fd78f Deleted pod: nginx-9cf9fd78f-khc5z

7m7s Normal ScalingReplicaSet deployment/nginx Scaled up replica set nginx-76df748b9 to 1

3m56s Normal ScalingReplicaSet deployment/nginx Scaled up replica set nginx-9cf9fd78f to 1

10s Normal ScalingReplicaSet deployment/nginx Scaled down replica set nginx-76df748b9 to 0

10s Normal ScalingReplicaSet deployment/nginx Scaled up replica set nginx-8cb54bccc to 1

2s Normal ScalingReplicaSet deployment/nginx Scaled down replica set nginx-9cf9fd78f to 0

kubectl get pods -n frontend

NAME READY STATUS RESTARTS AGE

nginx-8cb54bccc-g4htt 1/1 Running 0 38s

The pod has now been allowed to run on the tainted node. If there are other worker nodes in the cluster, and they are not tainted, then this pod can also land on those free nodes. To make sure that this pod lands on the nodes that are dedicated to front-end pods, then aside from taint and toleration, we need to label the front-end nodes (e.g., app=frontend) and then use NodeSelector in the pod deployment spec so that the pod is only scheduled on front-end nodes.

## Conclusion

Taints and Tolerations provide advanced pod scheduling where tainted nodes control which pods can be scheduled on them. They are easier to manage as compared to other custom scheduling methods such as affinities. Nodes with special hardware, dedicating nodes for a group of users, and taint based pod evictions are some of the known use cases for taints and tolerations.

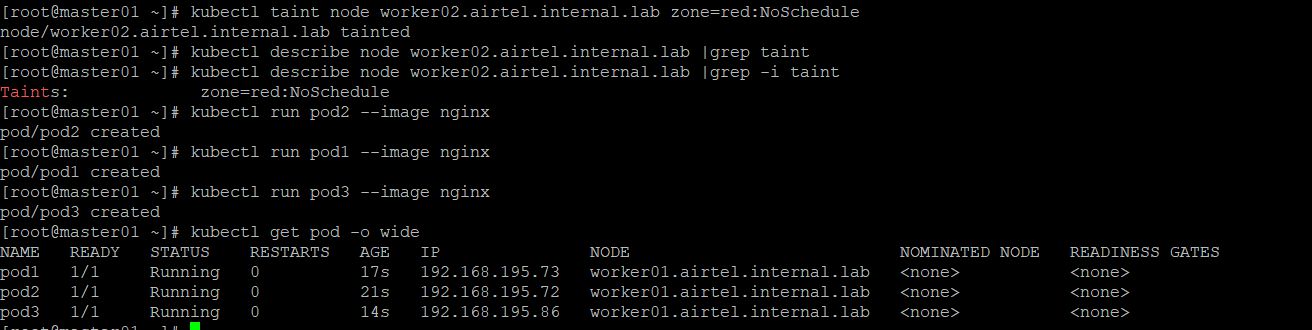
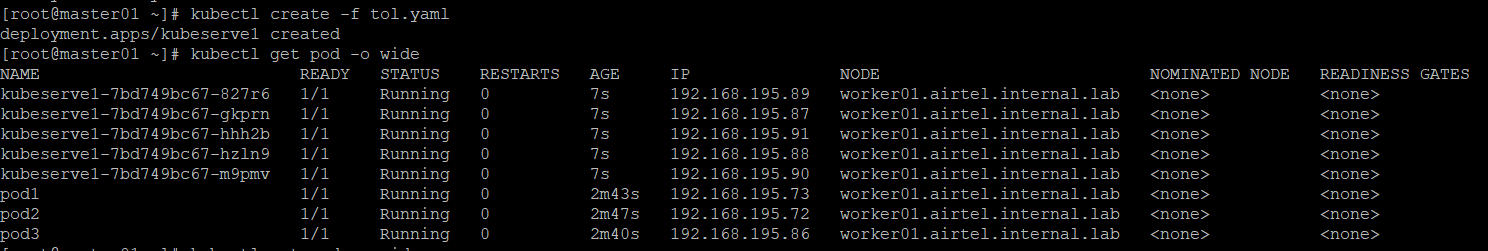
**NoSchedule**

[root@master01 ~]# kubectl taint node worker02.airtel.internal.lab zone=red:NoSchedule

node/worker02.airtel.internal.lab tainted

[root@master01 ~]# kubectl describe node worker02.airtel.internal.lab |grep -i taint

Taints: zone=red:NoSchedule

[root@master01 ~]# cat tol.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: kubeserve1

spec:

replicas: 5

selector:

matchLabels:

app: kubeserve

template:

metadata:

name: kubeserve

labels:

app: kubeserve

spec:

tolerations:

- key: color

operator: "Equal"

value: "red"

effect: "NoSchedule"

containers:

- image: leaddevops/kubeserve:v1

name: app

**Note: NoExecute**

It means you do not schedule any pod on this host also after NoExecute running pods also leave this node immediately.

[root@master01 ~]# kubectl taint node worker02.airtel.internal.lab situation=serious:NoExecute

node/worker02.airtel.internal.lab tainted

[root@master01 ~]# kubectl describe node worker02.airtel.internal.lab |grep -i taint

Taints: zone=red:NoSchedule

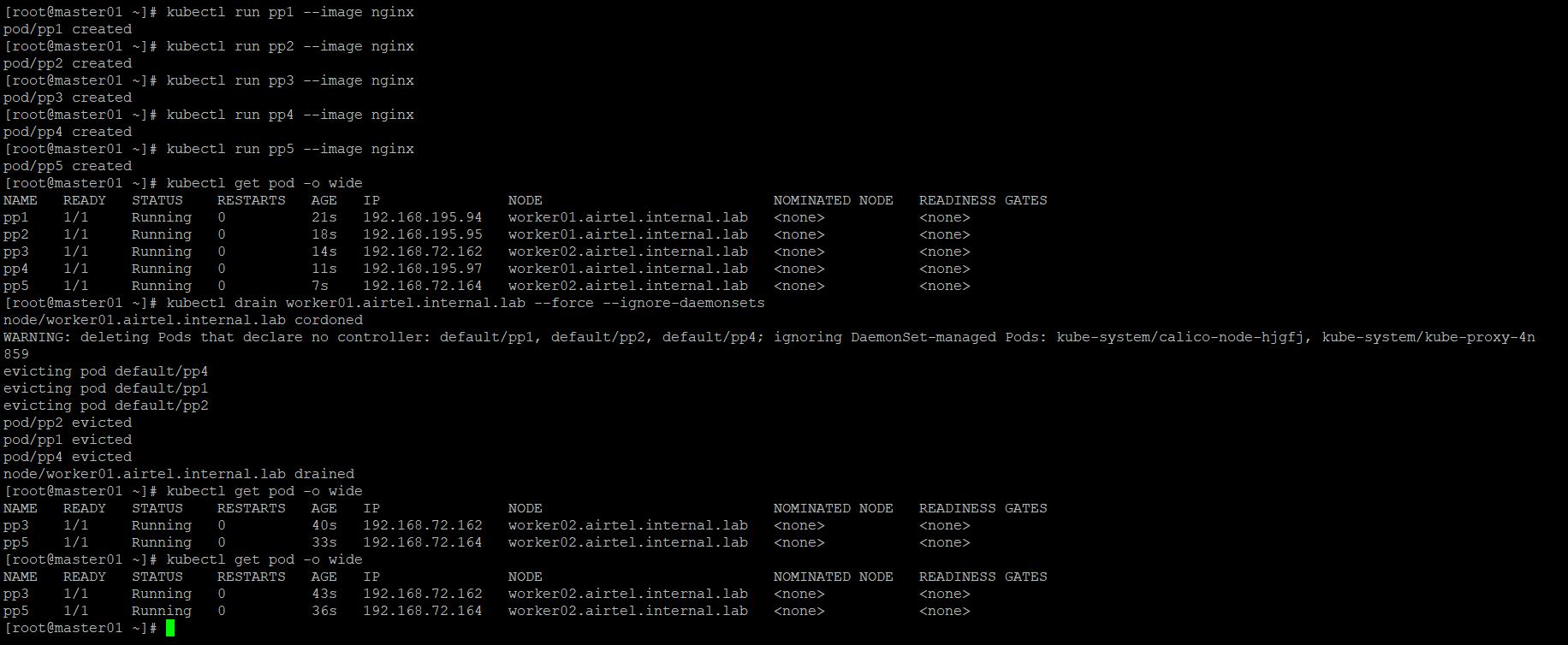
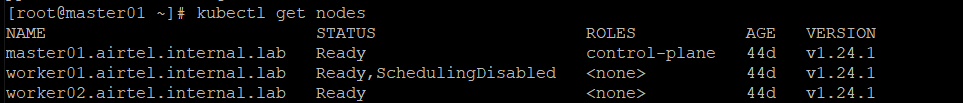
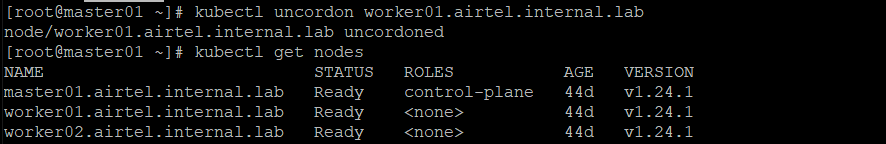
[root@master01 ~]# kubectl taint node worker02.airtel.internal.lab situation=serious:NoExecute-

node/worker02.airtel.internal.lab untainted

**Drain, Cordon, or Uncordon**

* Drain node in preparation for maintenance same as kubectl drain.
* Cordon will mark the node as unschedulable.
* Uncordon will mark the node as schedulable.
* The given node will be marked unschedulable to prevent new pods from arriving.
* Then drain deletes all pods except mirror pods (which cannot be deleted through the API server).

[root@master01 ~]# kubectl drain master01.airtel.internal.lab --force --ignore-daemonsets

**NameSpaces Limit**

Object: you can do this with resource quota.

Hardware: you can create namespace base on the hardware resources. Like CPU and RAM

[root@master01 ~]# kubectl create -f https://raw.githubusercontent.com/lerndevops/educka/master/7-security/namespaces/resource-quotas/object-counts-demo.yml

namespace/prod created

resourcequota/object-counts created

[root@master01 ~]# cat object-counts-demo.yml

|  |
| --- |
| apiVersion: v1 |
|  | kind: Namespace |
|  | metadata: |
|  | name: prod |
|  | labels: |
|  | name: production |
|  | --- |
|  | apiVersion: v1 |
|  | kind: ResourceQuota |
|  | metadata: |
|  | name: object-counts |
|  | namespace: prod |
|  | spec: |
|  | hard: |
|  | pods: "4" |
|  | replicationcontrollers: "2" |
|  | services: "10" |
|  | count/deployments.apps: "2" |

[root@master01 ~]# kubectl get ns

NAME STATUS AGE

default Active 44d

kube-node-lease Active 44d

kube-public Active 44d

kube-system Active 44d

prod Active 68s

[root@master01 ~]# kubectl get resourcequota -n prod

NAME AGE REQUEST LIMIT

object-counts 89s count/deployments.apps: 0/2, count/replicasets.apps: 0/2, pods: 0/4, replicationcontrollers: 0/2, services: 0/10

[root@master01 ~]# kubectl -n prod create deploy dep --image nginx

deployment.apps/dep created

[root@master01 ~]# kubectl get resourcequota -n prod

NAME AGE REQUEST LIMIT

object-counts 3m57s count/deployments.apps: 1/2, count/replicasets.apps: 1/2, pods: 2/4, replicationcontrollers: 0/2, services: 0/10

[root@master01 ~]# kubectl -n prod scale deployment dep --replicas=3

deployment.apps/dep scaled

[root@master01 ~]# kubectl get resourcequota -n prod

NAME AGE REQUEST LIMIT

object-counts 6m10s count/deployments.apps: 1/2, count/replicasets.apps: 1/2, pods: 4/4, replicationcontrollers: 0/2, services: 0/10

[root@master01 ~]# kubectl -n prod get deploy

NAME READY UP-TO-DATE AVAILABLE AGE

dep 3/3 3 3 4m4s

**Hardware Quota**

[root@master01 ~]# kubectl craete -f https://raw.githubusercontent.com/lerndevops/educka/master/7-security/namespaces/resource-quotas/compute-resources-demo.yml

|  |
| --- |
| apiVersion: v1 |
|  | kind: Namespace |
|  | metadata: |
|  | name: teama |
|  | labels: |
|  | name: development |
|  | --- |
|  | apiVersion: v1 |
|  | kind: ResourceQuota |
|  | metadata: |
|  | name: compute-resources-demo |
|  | namespace: teama |
|  | spec: |
|  | hard: |
|  | requests.cpu: "1" |
|  | requests.memory: 1Gi |
|  | limits.cpu: "2" |
|  | limits.memory: 2Gi |

[root@master01 ~]# kubectl get resourcequota -n teama

NAME AGE REQUEST LIMIT

compute-resources-demo 67s requests.cpu: 0/1, requests.memory: 0/1Gi limits.cpu: 0/2, limits.memory: 0/2Gi

**ETCD**

How to take etcd backup

## Install etcdctl:

export RELEASE="3.3.13"

wget https://github.com/etcd-io/etcd/releases/download/v${RELEASE}/etcd-v${RELEASE}-linux-amd64.tar.gz

tar xvf etcd-v${RELEASE}-linux-amd64.tar.gz

cd etcd-v${RELEASE}-linux-amd64

sudo mv etcdctl /usr/local/bin

## etcd snapshot explanation

the idea is to create a snapshot of the etcd database. This is done by communicating with the running etcd instance in Kubernetes and asking it to create a snapshot. in order to communicate with the etcd pod in Kubernetes, we need to:

## **backup ETCD Data:**

mkdir /tmp/etcd-backup

## ETCDCTL\_API=3 etcdctl --endpoints=172.16.102.21:2379 --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key snapshot save /tmp/etcd-backup/172.16.102.db

## remove /var/lib/etcd folder, delete couple of deployments/pods

## restore ETCD Data:

ETCDCTL\_API=3 etcdctl snapshot restore /etcd-backup/etcd-snapshot-latest.db --initial-cluster etcd-restore=[https:// 172.16.102.21:2380](https://10.128.0.42:2380/) --initial-advertise-peer-urls=[https:// 172.16.102.21:2380](https://10.128.0.42:2380/) --name etcd-restore --data-dir /var/lib/etcd

**Top**

Command can be used to retrieve snapshots of resource utilization of pods or nodes in your Kubernetes cluster. Resource utilization is an important thing to monitor for Kubernetes cluster owners. In order to monitor resource utilization, you can keep track of things like CPU, memory, and storage.

**What is kubectl Top command?**[**​**](https://signoz.io/blog/kubectl-top/#what-is-kubectl-top-command)

As mentioned before, kubectl - pronounced (Kube: c-t-l) - is a CLI for running commands that can help you interact with a Kubernetes interface or resources in a k8 cluster. These resources include pods, deploy, replica set, etc.

A kubectl top is a command used to list all the running nodes and pods along with their resource utilization. It provides you a snapshot of resource utilization metrics like CPU, memory, and storage on each running node.

Each node in Kubernetes comes with cAdvisor, which is an open-source agent that monitors resource usage about containers. kubectl command gets resource utilization metrics from cAdvisor via the metrics-server.

To obtain these metrics, you need to run the kubectl top command which shows the CPU, memory, and network utilization for the containers, pods, or nodes. For the kubectl top command to work, you need to have metrics API installed. You can find instructions to install metrics API [here](https://github.com/kubernetes-sigs/metrics-server).

Now that you have a brief understanding of the concepts let’s see how the kubectl top command operates.

**Using kubectl top node**[**​**](https://signoz.io/blog/kubectl-top/#using-kubectl-top-node)

Running the kubectl top node command lists metrics of the current node which would look like this:

[root@master01 ~]# kubectl top node

NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%

master01.airtel.internal.lab 180m 4% 2272Mi 29%

worker01.airtel.internal.lab 74m 1% 1632Mi 21%

worker02.airtel.internal.lab 85m 2% 1556Mi 20%

[root@master01 ~]# kubectl top pod

No resources found in default namespace.

[root@master01 ~]# kubectl top pod -n kube-system

NAME CPU(cores) MEMORY(bytes)

calico-kube-controllers-6766647d54-phtbq 3m 97Mi

calico-node-5584c 18m 228Mi

calico-node-8xscm 26m 231Mi

calico-node-hjgfj 27m 231Mi

coredns-6d4b75cb6d-52qs7 1m 30Mi

coredns-6d4b75cb6d-wszgx 2m 30Mi

etcd-master01.airtel.internal.lab 11m 77Mi

kube-apiserver-master01.airtel.internal.lab 61m 537Mi

kube-controller-manager-master01.airtel.internal.lab 11m 60Mi

kube-proxy-4n859 1m 42Mi

kube-proxy-fb66f 1m 32Mi

kube-proxy-xlddb 1m 28Mi

kube-scheduler-master01.airtel.internal.lab 3m 29Mi

metrics-server-5b84dbf469-cjn6d 5m 30Mi

[root@master01 ~]# kubectl top pod -n prod

NAME CPU(cores) MEMORY(bytes)

dep-b47697d5f-25g5p 0m 3Mi

dep-b47697d5f-2g4tw 0m 3Mi

dep-b47697d5f-gqzrj 0m 3Mi

pod1 0m 3Mi

**Monitoring**

In the monitoring you can check your infra details.

kubect create -f <https://github.com/lerndevops/educka/blob/master/monitoring/metrics-server/metrics-server-v0.5.yml>

kubect create -f <https://github.com/lerndevops/educka/blob/master/monitoring/prometheus/prometheus.yaml>

**Volumes**

**EmptyDir**

An emptyDir volume is first created when a Pod is assigned to a node, and exists as long as that Pod is running on that node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each container. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

**Note:** A container crashing does not remove a Pod from a node. The data in an emptyDir volume is safe across container crashes.

Some uses for an emptyDir are:

* scratch space, such as for a disk-based merge sort
* checkpointing a long computation for recovery from crashes
* holding files that a content-manager container fetches while a webserver container serves the data

Depending on your environment, emptyDir volumes are stored on whatever medium that backs the node such as disk or SSD, or network storage. However, if you set the emptyDir.medium field to "Memory", Kubernetes mounts a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that unlike disks, tmpfs is cleared on node reboot and any files you write count against your container's memory limit.

**Ex:**

**cat educka/5-storage/volumes/emptyDir-vol-ex2.yml**

apiVersion: v1

kind: Pod

metadata:

name: myvolumes-pod

spec:

containers:

- image: centos

imagePullPolicy: IfNotPresent

name: myvolumes-container-1

command: ['sh', '-c', 'echo The Bench Container 1 is Running ; sleep 3600']

volumeMounts:

- mountPath: /demo1

name: demo-volume

- image: ubuntu

imagePullPolicy: IfNotPresent

name: myvolumes-container-2

command: ['/bin/bash', '-c', 'echo The Bench Container 2 is Running ; sleep 3600']

volumeMounts:

- mountPath: /demo2

name: demo-volume

- image: alpine

imagePullPolicy: IfNotPresent

name: myvolumes-container-3

command: ['sh', '-c', 'echo The Bench Container 3 is Running ; sleep 3600']

volumeMounts:

- mountPath: /demo3

name: demo-volume

volumes:

- name: demo-volume

emptyDir: {}

If you want to see all the running container inside the POD use below command

[root@master01 ~]# kubectl get pods myvolumes-pod -o jsonpath="{.spec.containers[\*].name}"

### hostPath

**Warning:**

HostPath volumes present many security risks, and it is a best practice to avoid the use of HostPaths when possible. When a HostPath volume must be used, it should be scoped to only the required file or directory, and mounted as ReadOnly.

If restricting HostPath access to specific directories through AdmissionPolicy, volumeMounts MUST be required to use readOnly mounts for the policy to be effective.

A hostPath volume mounts a file or directory from the host node's filesystem into your Pod. This is not something that most Pods will need, but it offers a powerful escape hatch for some applications.

For example, some uses for a hostPath are:

* running a container that needs access to Docker internals; use a hostPath of /var/lib/docker
* running cAdvisor in a container; use a hostPath of /sys
* allowing a Pod to specify whether a given hostPath should exist prior to the Pod running, whether it should be created, and what it should exist as

In addition to the required path property, you can optionally specify a type for a hostPath volume.

The supported values for field type are:

| Value | Behavior |
| --- | --- |
|  | Empty string (default) is for backward compatibility, which means that no checks will be performed before mounting the hostPath volume. |
| DirectoryOrCreate | If nothing exists at the given path, an empty directory will be created there as needed with permission set to 0755, having the same group and ownership with Kubelet. |
| Directory | A directory must exist at the given path |
| FileOrCreate | If nothing exists at the given path, an empty file will be created there as needed with permission set to 0644, having the same group and ownership with Kubelet. |
| File | A file must exist at the given path |
| Socket | A UNIX socket must exist at the given path |
| CharDevice | A character device must exist at the given path |
| BlockDevice | A block device must exist at the given path |

Watch out when using this type of volume, because:

* HostPaths can expose privileged system credentials (such as for the Kubelet) or privileged APIs (such as container runtime socket), which can be used for container escape or to attack other parts of the cluster.
* Pods with identical configuration (such as created from a PodTemplate) may behave differently on different nodes due to different files on the nodes
* The files or directories created on the underlying hosts are only writable by root. You either need to run your process as root in a [privileged Container](https://kubernetes.io/docs/tasks/configure-pod-container/security-context/) or modify the file permissions on the host to be able to write to a hostPath volume

#### hostPath configuration example

[root@master01 volumes]# cat hostPath-vol.yml

apiVersion: v1

kind: Pod

metadata:

name: hostpath-vol-pod1

spec:

containers:

- name: test-container

image: tomcat

volumeMounts:

- name: hostpath-volume

mountPath: /usr/local/tomcat/logs

volumes:

- name: hostpath-volume

hostPath:

path: /tmp/logs # directory location on host

type: Directory # this field is optional

**PV and PVC**

**Persistent Volume**

1 GB Volume

**Storage Class**

5 GB Volume

2 GB Volume

**4 GB**

**Persistent Volume Claim**

10 GB Volume

20 GB Volume

**Pods**

## Kubernetes PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs)

Storage management is essential in Kubernetes, especially in  large environments where many users deploy multiple Pods. The users in this environment often need to configure storage for each Pod, and when making a change to existing applications, it must be made on all Pods, one after the other. To mitigate this time-consuming scenario and separate the details of how storage is provisioned from how it is consumed, we use PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs).

A **PersistentVolume (PV)** in Kubernetes is a pool of pre-provisioned storage resources in a Kubernetes cluster, that can be used across different user environments. Its lifecycle is separate from a Pod that uses the PersistentVolume.

A **PersistentVolumeClaim (PVC)**, is a process of storage requests from PVs by the users in Kubernetes. Kubernetes binds PVs with the PVCs based on the request and property set on those PVs. Kubernetes searches for PVs that correspond to the PVCs’ requested capacity and specified properties, so that each PVC can bind to a single PV.

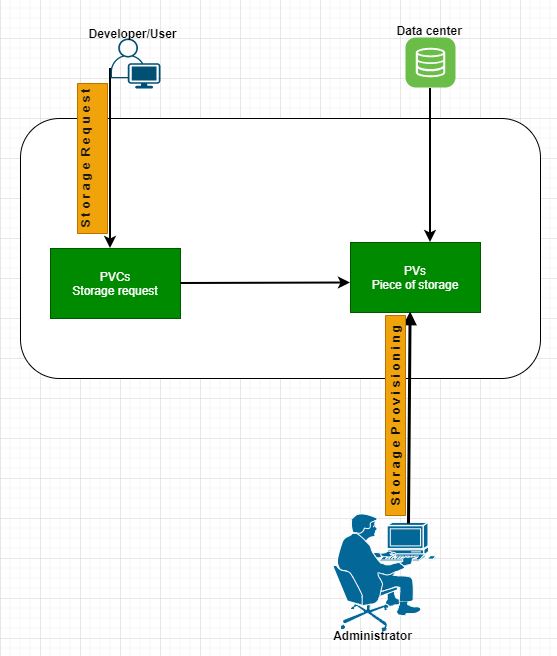
When there are multiple matches, you can use labels and selectors to bind a PVC to the right or a particular PV. This helps guard against a situation where a small PVC binds to a larger PV, since PVs and PVCs have a one-to-one relationship. When this happens, the remaining storage in the bound PVs are inaccessible to other users.

**NOTE:** Both the PVCs and the Pod using them must be in the same namespace.

## The Difference Between PVs and PVCs in Kubernetes

PVs and PVCs differ in provisioning, functionalities, and the person responsible for creating them, specifically :

* PVs are created by the cluster administrator or dynamically by Kubernetes, whereas users/developers create PVCs.
* PVs are cluster resources provisioned by an administrator, whereas PVCs are a user’s request for storage and resources.
* PVCs consume PVs resources, but not vice versa.
* A PV is similar to a node in terms of cluster resources, while a PVC is like a Pod in the context of cluster resource consumption.



## Difference between Volumes and PersistentVolumes

Volumes and PersistentVolumes differ in the following ways:

* A Volume separates storage from a container but binds it to a Pod, while PVs separate storage from a Pod.
* The lifecycle of a Volume is dependent on the Pod using it, while the lifecycle of a PV is not.
* A Volume enables safe container restart and allows sharing of data across containers, whereas a PV enables safe Pod termination or restart.
* A separate manifest YAML file is needed to create a PV, but not required for a volume.

## PersistentVolumes and PersistentVolumeClaims Lifecycle

The communication between PVs and PVCs  consists of the following stages:

* **Provisioning:** This is the process of creating a storage volume, which can be done **statically** or **dynamically**.
  + **Static:** The static way of provisioning a storage volume is when  PVs are created before PVCs by an Administrator and exist in the Kubernetes API, waiting to be claimed by a user’s storage request, using PVC.
  + **Dynamic:**  The dynamic way of storage provisioning involves creating PVs automatically, using StorageClass instead of manual provisioning of PersistentVolumes. Kubernetes will dynamically provision a volume specifically for this storage request, with the help of a StorageClass. (More about StorageClass and how it is used to provision PVs dynamically in the next part of this series).
* **Binding:** When an Administrator has provisioned PVs and users make a specific amount of storage requests (PVCs), a control loop in the master finds a matching PV and binds them together. If a matching volume does not exist, the claim will remain unbound until a volume match is created.
* **Using:** Pods use claims as volumes. The Kubernetes API checks the claim to find a bound PV and mounts it in the Pod for the users. When a claim is already bound to a PV, the bind remains unchanged as long as the user wants it. So user A’s bound PV can not be taken over by user B, if it is still in use by user A.
* **Reclaiming:** When a user is done with the volume usage, the resources must be freed up by deleting the PVC objects from Kubernetes. This practice will free the PV from the PVC, allowing reclamation of the resources, making it available for future use. What happens to the volumes afterwards is determined by the PersistentVolumeReclaimPolicy (PVRP) value specified in the configuration file. The PVRP provides three options of what you can do to a PersistentVolume after the claim has been deleted:  **retain**, **delete**, or **recycle**.
  + **Retain:** This is the default reclaim policy. It is a process that allows manual deletion of the PV by the Administrator. When a PVC is deleted, the PV remains intact, including its data, thus making it unavailable for use by another claim.  However, the Administrator can still manually reclaim the PersistentVolume.
  + **Delete:** When the reclaim policy is set to delete, the PV deletes automatically when the PVC is deleted and makes the space available. It is important to note that the dynamically provisioned PVs inherit the reclaim policy of their StorageClass, which defaults to **Delete**.
  + **Recycle:** This type of reclaim policy will scrub the data in the PV and make it available for another claim.

## **How to Create a PersistentVolume**

The following steps will guide you through how to create a PersistentVolume and how to use it in a Pod. Before continuing, it is imperative to have a basic knowledge of [volume, volumeMounts, and volume types](https://www.kubermatic.com/blog/keeping-the-state-of-apps-1-introduction-to-volume-and-volumemounts/) such as hostPath, emptyDir, among others, in order to effectively follow this hands-on practice. A running Kubernetes cluster and a kubectl command-line tool must be configured to talk to the cluster. If you do not have this, you can achieve this by simply creating a Kubernetes cluster on any environment with [KubeOne](https://github.com/kubermatic/kubeone#getting-started). Refer to [Getting Started](https://github.com/kubermatic/kubeone#getting-started) for instructions. Alternatively, you can go to the [Kubernetes playground](https://labs.play-with-k8s.com/) to practice. In that case, you might also need a cloud provider (GKE, AWS, etc.) access or credentials to provision storage.

Follow the below steps to create a PersistentVolume:

**Step 1:** Create the YAML file.

$ vim pv-config.yaml

**Step 2:** Copy and paste the below configuration file into the YAML manifest file created above.

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-volume

spec:

capacity:

storage: 3Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/app/data"

The configuration above shows properties with different functionalities. In addition to the properties from the previous Kubernetes objects exercises, let’s look at **accessModes**, **capacity**, and **storage** **properties**:

* **accessModes**: This property defines how a PV should be mounted on the host. The value can be **ReadWriteOnce**, **ReadOnlyMany**, or **ReadWriteMany**. Below is a basic description of  these values:
  + **ReadWriteOnce:** You can mount the PV as read-write by a single node.
  + **ReadOnlyMany:** You can mount the PV as read-only by multiple nodes.
  + **ReadWriteMany:** You can mount the PV as read-write by multiple nodes.
* **capacity.storage:** This property is used to set the amount of storage needed for the volume.

**Step 3:** Create the Persistent Volume using kubectl create command.

$ kubectl create -f pv-config.yaml

persistentvolume/my-volume created

**Step 4:** Check the created PV to see if it is available.

$ kubectl get pv

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE

my-volume 3Gi RWO Retain Available ⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀⠀ 60s

**Step 5:** Check the description of the PersistentVolume by running kubectl describe command.

$ kubectl describe pv my-volume

Labels: <none>

Annotations: <none>

Finalizers: [kubernetes.io/pv-protection]

StorageClass:

Status: Available

Claim:

Reclaim Policy: Retain

Access Modes: RWO

VolumeMode: Filesystem

Capacity: 3Gi

Node Affinity: <none>

Message:

Source:

Type: HostPath (bare host directory volume)

Path: /app/data

HostPathType:

Events: <none>

As seen above, the PersistentVolume is available and ready to serve a PVC storage request. The status will change from **available** to **bound** when it has been claimed.

**NOTE:** It is not advisable to use the hostpath volume type in a production environment.

## **How to Create a PersistentVolumeClaim in Kubernetes**

Now that the PersistentVolume has been successfully created, the next step is to create a PVC that will claim the volume. Creating a PersistentVolumeClaim is similar to the method used to create the PersistentVolume above, with a few differences in terms of its properties and values. The value of the kind property will be **PersistentVolumeClaim**. The **resources.request.storage** field will also be added, and the value will be the provisioned PV capacity value (so 3Gi in our case).

**The configuration will look like the manifest file below:**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-claim

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 3Gi

**Step 1:** Create a YAML file.

$ vim pvc-config.yaml

**Step 2:** Copy and paste the above configuration file into the YAML file created above.

**Step 3:** Create the PVC by running kubectl create command.

$ kubectl create -f pvc-config.yaml

persistentvolumeclaim/my-claim created

**Step 4:** Check the status of the PVC by running kubectl get command.

$ kubectl get pvc my-claim

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

my-claim Bound my-volume 3Gi RWO 7m

**Step 5:** Check the description of the PVC.

$ kubectl describe pvc my-claim

Name: my-claim

Namespace: default

StorageClass:

Status: Bound

Volume: my-volume

Labels: <none>

Annotations: pv.kubernetes.io/bind-completed: yes

pv.kubernetes.io/bound-by-controller: yes

Finalizers: [kubernetes.io/pvc-protection]

Capacity: 3Gi

Access Modes: RWO

VolumeMode: Filesystem

Mounted By: <none>

Events: <none>

As shown above, the status indicates a **“Bound”** state, which means the PVC and PV are now bound together. Now, check the status of the PV once again with kubectl get command.

$ kubectl get pv my-volume

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM ⠀⠀⠀STORAGECLASS REASON AGE

my-volume 3Gi RWO Retain Bound default/my-claim ⠀⠀⠀ 11m

The output shows that the status has changed from **“available”** when it is not yet bound to **“Bound”** because the PV has been claimed by the created PVC.

**Step 6:** Delete the PVC using kubectl delete command.

$ kubectl delete pvc my-claim

persistentvolumeclaim "my-claim" deleted

**Step 7:** Check both the PVC and PV status with kubectl get command.

$ kubectl get pvc my-claim

No resources found in default namespace.

$ kubectl get pv my-volume

NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM ⠀⠀⠀STORAGECLASS REASON AGE

my-volume 3Gi RWO Retain Released default/my-claim ⠀⠀ 21m

The output shows that the PV status has now changed from a **“Bound”** to a **“Released”** state, after the PVC is deleted.

## **PersistentVolumes States**

A PV has different states, can be in any of these, and each has its own meaning, described as follows:

* **Available:** The PV is free and not yet claimed or bounded.
* **Bound:** The PV has been claimed or bounded.
* **Released:** The bounded PVC has been deleted, and the PV is free from its previous bounded state.
* **Failed:** The PV has failed its automatic reclamation.

## **PersistentVolumeClaims States**

Each PVC, like the PV, has its own states that represent its current status.

* **Bound**: The PVC is bound to a PV.
* **Pending:** The PVC can not bind to a PV. This could be due to a higher storage request than the PV capacity, or the PV accessMode value is different from that of the PVC, etc.

## **How to Use PersistentVolumeClaim in a Pod**

A Pod can access storage with the help of a PVC, which will be used as a volume. PVC can be used in a Pod by first declaring a **“volumes”** property in the Pod manifest file and specifying the claim name under the declared volume type **“persistentVolumeClaim”** property. It is essential that both the PVC and the Pod using it exist in the same namespace. This will allow the cluster to find the claim in the Pod’s namespace and use it to access the PersistentVolume that is bound to the PVC. Once created, the applications in the containers can read and write into the storage.The complete configuration file will look like this:

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

containers:

- name: stclass-test

image: nginx

volumeMounts:

- mountPath: "/app/data"

name: my-volume

volumes:

- name: my-volume

persistentVolumeClaim:

claimName: my-claim

The below steps will guide you through how to use a claim in a Pod. The PV and PVC have to be provisioned before creating the Pod. The claim name inside the Pod must also match the claim name in the running PVC.

**Step 1:** Create a YAML file.

$ vim pvc-pod.yaml

**Step 2:** Copy and paste the above Pod manifest file into the YAML file created above and create the Pod with kubectl create command.

$ kubectl create -f pvc-pod.yaml

pod/my-pod created

**Step 3:** Check the status and the description of the Pod.

$ kubectl get pod my-pod

NAME READY STATUS RESTARTS AGE

my-pod 1/1 Running 0 6m50s

$ kubectl describe pod my-pod

Name: my-pod

Namespace: default

Priority: 0

Node: node01/172.17.0.57

Start Time: Tue, 01 Dec 2020 11:44:08 +0000

Labels: <none>

Annotations: <none>

Status: Running

IP: 10.244.1.2

IPs:

IP: 10.244.1.2

Volumes:

my-volume:

Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)

ClaimName: my-claim

ReadOnly: false

default-token-nlmxj:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-nlmxj

Optional: false

QoS Class: BestEffort

**Step 4:** Exec into the Pod to test the PVC use case.

$ kubectl exec -it my-pod -- bin/bash

root@my-pod:/#

**Step 5:** Use df -h together with the path to confirm the mount point.

root@my-pod:/# df -h /app/data

Filesystem Size Used Avail Use% Mountedon

/dev/mapper/host01--vg-root 191G 22G 159G 13% /app/data

**Step 6:** Change into the mount directory and create a file using echo command.

root@my-pod:/# cd /app/data

root@my-pod:/app/data# echo "I love Kubermatic" > file.txt

**Step 7:** Check the created file and data using ls and cat **commands**, then exit the Pod once this has been confirmed.

root@my-pod:/app/data# ls

file.txt

root@my-pod:/app/data# cat file.txt

"I love Kubermatic"

root@my-pod:/app/data# exit

**Step 8:** Delete and recreate the Pod.

$ kubectl delete pod my-pod

pod "my-pod" deleted

$ kubectl get pods

No resources found in default namespace.

Recreate the Pod and check the status:

$ kubectl create -f pvc-pod.yaml

pod/my-pod created

$ kubectl get pod my-pod

NAME READY STATUS RESTARTS AGE

my-pod 1/1 Running 0 5s

**Step 9:** Exec into the Pod and check the previous file and data created if they exist in the new Pod.

$ kubectl exec -it my-pod -- bin/bash

root@my-pod:/# df -h /app/data

Filesystem Size Used Avail Use% Mountedon

/dev/mapper/host01--vg-root 191G 22G 159G 13% /app/data

root@my-pod:/# cd /app/data

root@my-pod:/app/data# ls

file.txt

root@my-pod:/app/data# cat file.txt

"I love Kubermatic"

You can see that the file and data created in the deleted Pod above are still there and were taken up by the new Pod.

**ConfigMaps**

Many applications rely on configuration which is used during either application initialization or runtime.

Most of the times there is a requirement to adjust values assigned to configuration parameters.

ConfigMaps is the kubernetes way to inject application pods with configuration data.

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.

#### A ConfigMap is an API object used to store non-confidential data in key-value pairs in the cluster store (ETCD).

#### Pods can consume ConfigMaps as

* environment variables,
* command-line arguments,
* or as configuration files in a volume.

# **Ultimate Guide to ConfigMaps in Kubernetes**

How do you manage your application's configuration? For a Python or Node.js application, where do you store configuration? How do you set connection strings, analytics keys, and service URLs?

If you're using Kubernetes, the answer is ConfigMaps.

With this guide, tutorial, and examples, you’ll learn how to use ConfigMaps in Kubernetes. We’ll teach you how to create ConfigMaps. Then, you'll learn how to mount them in volumes and use them as environment variables.

## **What is a ConfigMap in Kubernetes?**

A ConfigMap is a dictionary of configuration settings. This dictionary consists of key-value pairs of strings. Kubernetes provides these values to your containers. Like with other dictionaries (maps, hashes, ...) the key lets you get and set the configuration value.

## **Why would you use a ConfigMap in Kubernetes?**

Use a ConfigMap to keep your application code separate from your configuration.

It is an important part of creating a Twelve-Factor Application.

This lets you change easily configuration depending on the environment (development, production, testing) and to dynamically change configuration at runtime.

## **What is a ConfigMap used for?**

A ConfigMap stores configuration settings for your code. Store connection strings, public credentials, hostnames, and URLs in your ConfigMap.

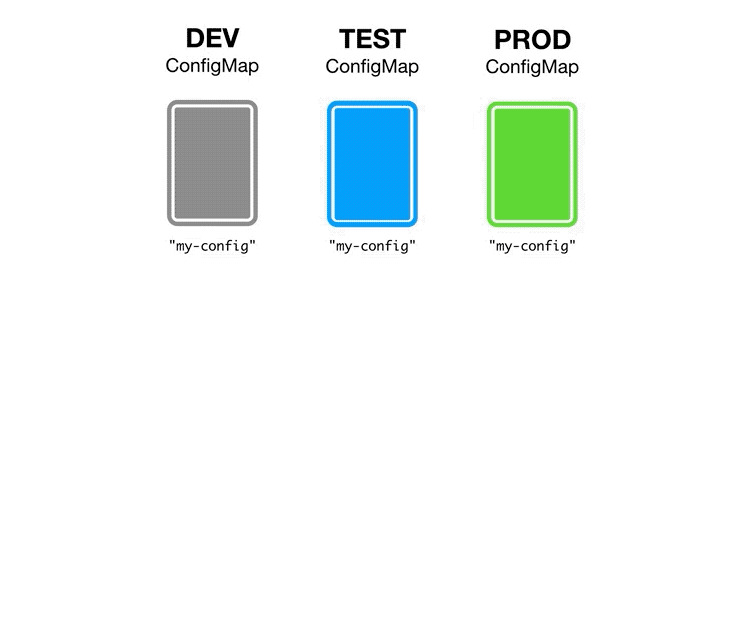
## **How does a ConfigMap work?**

Here's a quick animation I made showing how a ConfigMap works in Kubernetes.

First, you have multiple ConfigMaps, one for each environment.

Second, a ConfigMap is created and added to the Kubernetes cluster.

Third, containers in the Pod reference the ConfigMap and use its values.



## **How to create a ConfigMap in YAML? How to mount a ConfigMap as a Volume?**

Defining the ConfigMap in YAML and mounting it as a Volume is the easiest way to use ConfigMaps.

The official documentation waaay overcomplicates this.

Here's the method I use. It's simpler and easier for when you're starting to learn about ConfigMaps.

My favourite way is to define the ConfigMap dictionary in a YAML file.

This lets you create that ConfigMap like any other Kubernetes resources using `kubectl apply -f $file.yaml`. After that, you mount the ConfigMap as a Volume in your Pod's YAML specification.

### 1. Define the ConfigMap in a YAML file.

Create a YAML file setting the key-value pairs for your ConfigMap.

|  |  |
| --- | --- |
|  | kind: ConfigMap |
|  | apiVersion: v1 |
|  | metadata: |
|  | name: example-configmap |
|  | data: |
|  | # Configuration values can be set as key-value properties |
|  | database: mongodb |
|  | database\_uri: mongodb://localhost:27017 |
|  |  |
|  | # Or set as complete file contents (even JSON!) |
|  | keys: | |
|  | image.public.key=771 |
|  | rsa.public.key=42 |

[view raw](https://gist.github.com/matthewpalmer/33016359f49c88acc12e86eda232f14a/raw/240e535a5e493b907ce441e4cabafdb35547d87d/config-map.yaml)[config-map.yaml](https://gist.github.com/matthewpalmer/33016359f49c88acc12e86eda232f14a#file-config-map-yaml)hosted with ❤ by [GitHub](https://github.com/)

### 2. Create the ConfigMap in your Kubernetes cluster

Create the ConfigMap using the command kubectl apply -f config-map.yaml

### 3. Mount the ConfigMap through a Volume

Each property name in this ConfigMap becomes a new file in the mounted directory (`/etc/config`) after you mount it.

|  |  |
| --- | --- |
|  | kind: Pod |
|  | apiVersion: v1 |
|  | metadata: |
|  | name: pod-using-configmap |
|  |  |
|  | spec: |
|  | # Add the ConfigMap as a volume to the Pod |
|  | volumes: |
|  | # `name` here must match the name |
|  | # specified in the volume mount |
|  | - name: example-configmap-volume |
|  | # Populate the volume with config map data |
|  | configMap: |
|  | # `name` here must match the name |
|  | # specified in the ConfigMap's YAML |
|  | name: example-configmap |
|  |  |
|  | containers: |
|  | - name: container-configmap |
|  | image: nginx:1.7.9 |
|  | # Mount the volume that contains the configuration data |
|  | # into your container filesystem |
|  | volumeMounts: |
|  | # `name` here must match the name |
|  | # from the volumes section of this pod |
|  | - name: example-configmap-volume |
|  | mountPath: /etc/config |

[view raw](https://gist.github.com/matthewpalmer/01ea5d5f82032e5f78b3612bf155ebd5/raw/be5cb743e7eaafa6979d94a166df662d9c5c4aa4/pod.yaml)[pod.yaml](https://gist.github.com/matthewpalmer/01ea5d5f82032e5f78b3612bf155ebd5#file-pod-yaml)hosted with ❤ by [GitHub](https://github.com/)

Attach to the created Pod using `kubectl exec -it pod-using-configmap sh`. Then run `ls /etc/config` and you can see each key from the ConfigMap added as a file in the directory. Use `cat` to look at the contents of each file and you’ll see the values from the ConfigMap.

You can then read the configuration settings using Python/Node.js/PHP from this file.

## **How to use a ConfigMap with Environment Variables and `envFrom`?**

You can consume a ConfigMap via environment variables in a running container using the `envFrom` property.

### 1. Create the ConfigMap

Create the ConfigMap using the example from the previous section.

### 2. Add the `envFrom` property to your Pod's YAML

Set the `envFrom` key in each container to an object containing the list of ConfigMaps you want to include.

|  |  |
| --- | --- |
|  | kind: Pod |
|  | apiVersion: v1 |
|  | metadata: |
|  | name: pod-env-var |
|  | spec: |
|  | containers: |
|  | - name: env-var-configmap |
|  | image: nginx:1.7.9 |
|  | envFrom: |
|  | - configMapRef: |
|  | name: example-configmap |

[view raw](https://gist.github.com/matthewpalmer/c9649e7769e3e7e5e3790f5b3a4e6908/raw/c992552c067bd2a44d36c8c372381099d2678d77/pod-env-var.yaml)[pod-env-var.yaml](https://gist.github.com/matthewpalmer/c9649e7769e3e7e5e3790f5b3a4e6908#file-pod-env-var-yaml)hosted with ❤ by [GitHub](https://github.com/)

Attach to the created Pod using `kubectl exec -it pod-env-var sh`. Then run `env` and see that each key from the ConfigMap is now available as an environment variable.

## **What are the other ways to create and use ConfigMaps?**

There are three other ways to create ConfigMaps using the `kubectl create configmap` command. I prefer the methods used above, but here are your options.

1. Use the contents of an entire directory with kubectl create configmap my-config --from-file=./my/dir/path/
2. Use the contents of a file or specific set of files with kubectl create configmap my-config --from-file=./my/file.txt
3. Use literal key-value pairs defined on the command line with kubectl create configmap my-config --from-literal=key1=value1 --from-literal=key2=value2

You can get more information about this command using kubectl create configmap --help.

**Secret**

**What Are Kubernetes Secrets?**

A Kubernetes secret is an object storing sensitive pieces of data such as usernames, [passwords](https://phoenixnap.com/blog/strong-great-password-ideas), tokens, and keys. Secrets are created by the system during an app installation or by users whenever they need to store sensitive information and make it available to a pod.

If passwords, tokens, or keys were simply part of a pod definition or container image, they could be accidentally exposed during Kubernetes operations. Therefore, the most important function of the secret is to prevent accidental exposure of the information stored in it while at the same time making it available wherever the user needs it.

**Note:** Secrets are not the only way to manage sensitive information in Kubernetes. By using the system’s declarative nature, it is easy to integrate third-party information management solutions.

## **Using Kubernetes Secrets**

When you create a secret, it needs to be referenced by the pod that will use it. To make a secret available for a pod:

1. Mount the secret as a file in a volume available to any number of containers in a pod.

2. Import the secret as an environment variable to a container.

3. Use kubelet, and the imagePullSecrets field.

The following sections explain how to create Kubernetes secrets, as well as how to decode and access them.

### Create Kubernetes Secrets

To create a Kubernetes secret, apply one of the following methods:

* Use kubectl for a command-line based approach.
* Create a configuration file for the secret.
* Use a generator, such as Kustomize to generate the secret.

**Note:** A secret must have a name that is a valid [DNS](https://phoenixnap.com/kb/what-is-domain-name-system-works) subdomain name.

**Create Secrets Using kubectl**

1. To start creating a secret with kubectl, first create the files to store the sensitive information:

echo -n '[username]' > [file1]

echo -n '[password]' > [file2]

Using echo -n to append text to a file

The -n option tells echo not to append a new line at the end of the string. The new line is also treated as a character, so it would be encoded together with the rest of the characters, producing a different encoded value.

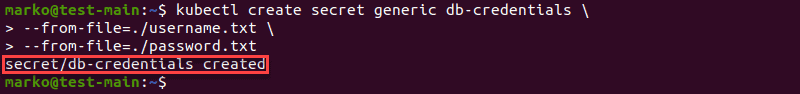
2. Now, use kubectl to create a secret using the files from the previous step. Use the generic subcommand to create an Opaque secret. Also, add the --from-file option for each of the files you want to include:

kubectl create secret generic [secret-name] \

--from-file=[file1] \

--from-file=[file2]

The output confirms the creation of the secret:



3. To provide keys for values stored in the secret, use the following syntax:

kubectl create secret generic [secret-name] \

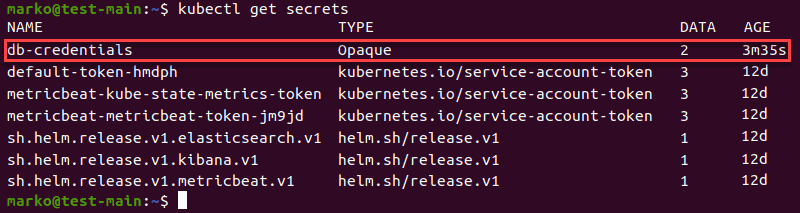
--from-file=[key1]=[file1] \

--from-file=[key2]=[file2]

4. Check that the secret has been successfully created by typing:

kubectl get secrets

The command shows the list of available secrets – their names, types, number of data values they contain, and their age:

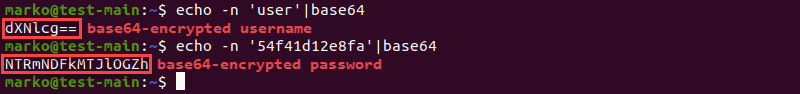


**Create Secrets in a Configuration File**

1. To create a secret by specifying the necessary information in a configuration file, start by encoding the values you wish to store:

echo -n '[value1]' | base64

echo -n '[value2]' | base64



2. Now create [a yaml file](https://phoenixnap.com/blog/what-is-yaml-with-examples) using a text editor. The file should look like this:

apiVersion: v1

kind: Secret

metadata:

name: newsecret

type: Opaque

data:

username: dXNlcg==

password: NTRmNDFkMTJlOGZh

3. Save the file and use the kubectl apply command to create the secret:

kubectl apply -f [file]

Using kubectl apply to create a secret from a file

**Create Kubernetes Secret with Generators**

Generators such as Kustomize help quickly generate secrets.

1. To create a secret with Kustomize, create a file named kustomization.yaml and format it as follows:

secretGenerator:

- name: db-credentials

files:

- username.txt

- password.txt

The example above states db-credentials as the name of the secret and uses two previously created files, username.txt, and password.txt**,** as data values.

2. Alternatively, to provide the unencrypted, literal version of the data values, include the literals section with key-value pairs you wish to store:

secretGenerator:

- name: db-credentials

literals:

- username=user

- password=54f41d12e8fa

3. Save the file and use the following command in the folder where kustomization.yaml is located:

kubectl apply -k .

The output confirms the creation of the secret:

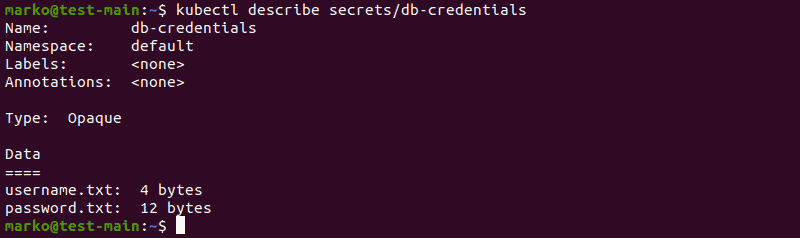
Using kubectl apply -k to create a secret using the Kustomize generator

### Use kubectl describe to See Created Secrets

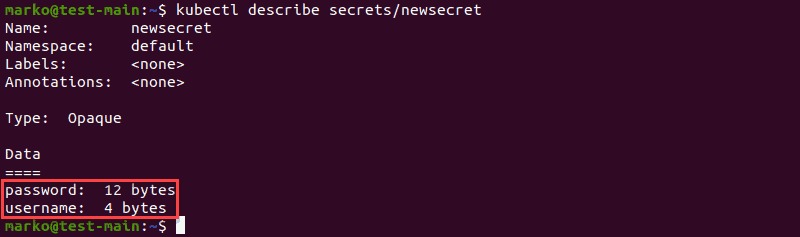
The kubectl describe command shows basic information about Kubernetes objects. Use it to view the description of a secret.

kubectl describe secrets/[secret]

The first example shows the secret created by providing files as data values:



The second example describes the secret created using string literals. Notice the change in the Data section, which now shows names of the keys instead of filenames:



### Decode Secrets

1. To decode the values in a secret, access them by typing the following command:

kubectl get secret [secret] -o jsonpath='{.data}'

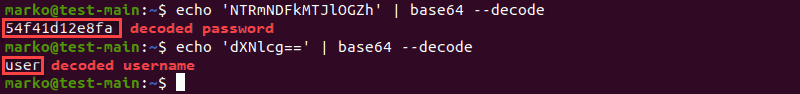
The output shows the encoded key-value pairs stored in the data section:

Using kubectl get to see keys and values in a json file

2. [Use the echo command](https://phoenixnap.com/kb/echo-command-linux) to type the encoded string and pipe the output to the base64 command:

echo '[encoded-value]' | base64 --decode

The decoded strings appear as the output:



### Access Secrets Loaded in a Volume

1. To access secrets mounted to a pod in a [separate volume](https://phoenixnap.com/kb/kubernetes-persistent-volumes), modify the definition of the pod to include a new volume. Choose any volume name you want, but make sure that it is the same as the name of the secret object.

2. Be sure to specify readOnly as true. For example, the pod definition may look like this:

apiVersion: v1

kind: Pod

metadata:

name: test-pod

spec:

containers:

- name: test-pod

image: redis

volumeMounts:

- name: newsecret

mountPath: “/etc/newsecret”

readOnly: true

volumes:

- name: newsecret

secret:

secretName: newsecret

2. Open another terminal instance and use the kubectl exec command to access the pod’s bash shell:

kubectl exec -it [pod] -- /bin/bash

Bashing into a pod by using kubectl exec

3. cd into /etc/newsecret, and find the files contained in the secret:

cd /etc/newsecret



### Project Secrets into a Container Using Environment Variables

1. Another way to access secrets in a Kubernetes pod is to import them as [environment variables](https://phoenixnap.com/kb/helm-environment-variables) by modifying the pod definition to include references to them. For example:

apiVersion: v1

kind: Pod

metadata:

name: secret-env-pod

spec:

containers:

- name: secret-env-pod

image: redis

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: newsecret

key: username

- name: SECRET\_PASSWORD

valueFrom:

secretKeyRef:

name: newsecret

key: password

restartPolicy: Never

2. Use kubectl exec again to bash into a pod.

3. Test the environment variable using the echo command:

echo $[VARIABLE]

The command output shows the value mapped to the variable:

Exposing the value of an environment variable using echo

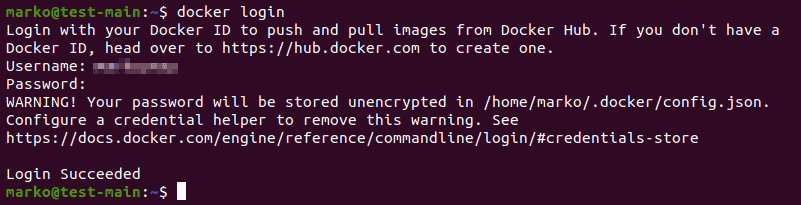
**Note:** Check out our guide to learn [how to deploy a Redis Cluster on Kubernetes](https://phoenixnap.com/kb/kubernetes-redis) using either Helm or ConfigMap.

### Use Secrets to Pull Docker Images from Private Docker Registries

1. To use private Docker registries, first, you need to log in to Docker:

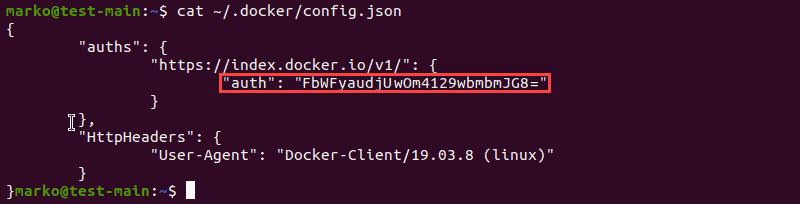
docker login

2. When prompted, provide your login credentials:



3. If the login is successful, Docker updates the config.json file with your data. Use the [cat command](https://phoenixnap.com/kb/linux-cat-command) to view the file:

cat ~/.docker/config.json



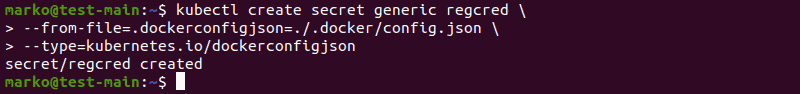
The auths section contains the auth key, which is an encoded version of the Docker credentials.

4. Use kubectl to create a secret, providing the location of the config.json file and the type of the secret:

kubectl create secret generic [secret] \

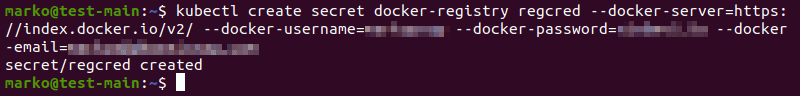
--from-file=.dockerconfigjson=./.docker/config.json \

--type=kubernetes.io/dockerconfigjson



Alternatively, perform all the steps above, including logging in to Docker, on the same line:

kubectl create secret docker-registry [secret] --docker-server:[address] --docker-username=[username] --docker-password=[password] --docker-email=[email]



5. To create a pod that has access to this secret, create a yaml file that defines it. The file should look like this:

apiVersion: v1

kind: Pod

metadata:

name: private-reg

spec:

containers:

- name: private-reg-container

image:

imagePullSecrets:

- name: regcred

6. Finish creating the pod by activating it with kubectl apply:

kubectl apply -f [file]

Creating a pod with access to the Docker registry secret

## **Kubernetes Secrets Considerations**

**Kubernetes secrets** are a secure way to store sensitive information. However, before you decide on the best method for your usage scenario, you should consider the following points:

* Usernames and passwords in Secrets are encoded with base-64. This text-encoding technique obscures data and prevents accidental exposure, but it is not secure against malicious [cyber attacks](https://phoenixnap.com/blog/cyber-security-attack-types).
* Secrets are only available in the cluster in which they are located.
* Secrets usually rely on a master key which is used to unlock them all. While there are methods to secure the master key, using them only creates another master key scenario.

To mitigate these problems, apply some of the solutions below:

* Integrate a secrets management tool that uses the Kubernetes Service account to authenticate users who need access to the secret vault.
* Integrate an IAM ([Identity and Access Management](https://phoenixnap.com/glossary/identity-and-access-management/)) tool to allow the system to use tokens from a Secure Token Service.
* Integrate a third-party secrets manager into pods.

**LoadBalancer**

Load Balancer Service, IP: NodePort

IP1: NodePort IP2: NodePort IP3: NodePort

Pod01

Pod03

Pod02

Worker Node02

Worker Node01

Mater Node01

Service : NodePort

**What is Kubernetes Load Balancing?**

Websites and business applications have to deal with a vast number of queries at peak times. Moreover, to meet this demand, enterprises **spread the workload** over many servers. Besides **cost savings**, it also enables an even distribution of load across all servers. This [load balancing](https://techgenix.com/exchange-server-load-balancer/) prevents a single server from crashing. Due to business demands and user needs all modern applications cannot run without it.

Servers can be on-premises, in the cloud, or in a data center. They can be virtual, physical, or part of hybrid solutions. To that end, load balancing needs to **work across many different platforms**. In any case, you need to achieve the greatest output with the lowest response time.

One way of looking at [load balancing](https://techgenix.com/free-fault-tolerant-load-balancing-using-citrix-netscaler-express-part1/) is to consider the process as a **“traffic cop”**. In this analogy, the “traffic cop” routes incoming requests and data across all servers. Here, the “cop” ensures no single server is overworked and brings order to a chaotic situation. If, for some reason a server does go down, the load balancer **redirects traffic automatically**. When you add a new server to the server pool, the load balancer automatically allocates its resources. To that end, automatic load balancers ensure your system maintains high availability during upgrades and system maintenance tasks.

Now you’re up to speed with the load balancing basics, let’s look at the benefits.

## **Benefits of Load Balancing**

Load balancing provides high availability to your business along with many other value-adding advantages:

* **Support for traffic during peak hours:** as transactions spike load balancing provides a high quality and rapid response to demand.
* **Traffic shifting during** [**canary releases**](https://techgenix.com/vamp-io-uses-istio/): as new developments are released traffic is redirected through the network to compensate for specific resource bottlenecks.
* **Blue or green releases:** as you run different versions of an application in two separate environments load balancing helps reduce a systemwide slowdown.
* **Infrastructure migration:**as platform transitions occur, load balancing helps ensure high availability is achieved.
* **Predictive analytics:** as user traffic changes it can be monitored and proactive changes to routine made to accommodate it.
* **Maintenance task flexibility:** as maintenance occurs outages are reduced by routing users to online servers.

I’ve now explained how you can use load balancing to help your business. Now, you’ll learn how load balancing can be applied to the Kubernetes network model.

## Understanding the Kubernetes Network Model

In this section, I’ll help you understand how load balancing works with a Kubernetes solution. You’ll learn all the main terms used in **Kubernetes networking**to help you understand how everything works.

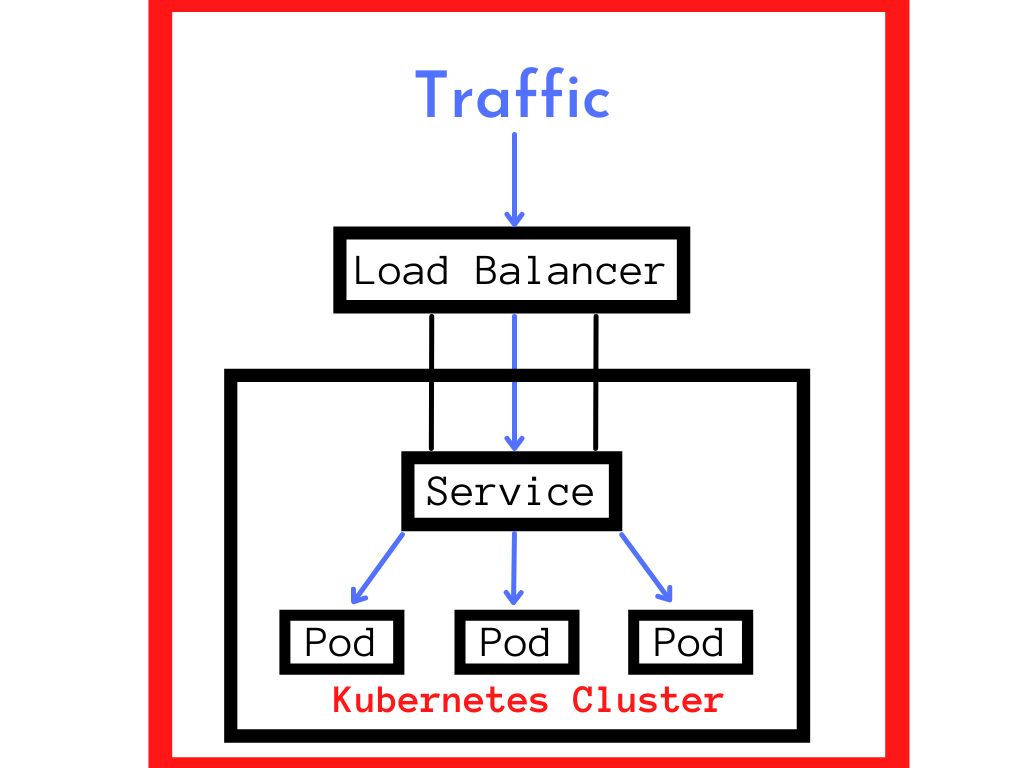
### Load Balancing in Kubernetes

Something called **“**[**services**](https://techgenix.com/?s=Kubernetes+services)**”** plays a big role in load balancing. In short, a service is a group of pods under a common name. By grouping objects like this, the system can easily identify the pod groups and enables selection by other components.

Kubernetes assigns a **ClusterIP** to each service you create. This is an IP address accessible only within a K8s cluster. It enables you to link other containers within the cluster with these pods. Services can go beyond internal usage. In fact, services can become accessible to external clients if needed.

### Load Balancing in External Connections

LoadBalancer, Ingress, and NodePorts are different ways of exposing services to the world. In effect, they enable you to bring **external traffic** into your [cluster](https://techgenix.com/local-kubernetes-cluster-setup/). The standard way to directly expose a service to the internet is via a LoadBalancer service. In this case, traffic on the specified node is forwarded to the service without any filtering or routing. This enables you to send any kind of traffic, like **TCP**, **gRPC**, **HTTP**, **UDP**, etc., to it. In other words, LoadBalancer provides a stable endpoint for external traffic to access.



**It’s a balancing act.**

### Ingress Smart Routing

Ingress, unlike the **LoadBalancer**, isn’t a type of service. Instead, Ingress acts as a **“smart router”** in front of multiple routers. This smart router operates using a controller, which includes an **Ingress resource** and a **daemon**. Both resource and daemon use a specialized [Kubernetes pod](https://techgenix.com/open-source-storage-projects-kubernetes/) that helps the project pods to communicate to external clients. In essence, the Ingress resource is a set of rules that **govern traffic**. This is to say, ingress decides which inbound connections can reach which services. The **daemon** applies these rules during runtime and specifies **connection requirements**.

### Ingress Customized Rules and Compliance

The ingress resource enables you to add more detailed **load balancing rules** if needed. Doing so helps you accommodate specific **vendors or regulatory requirements**. Customizable rules allow you to control load balancing specifically for your needs. In this case, it means you can cater to application needs and runtime conditions.

Now that you have a good understanding of how load balancing works in Kubernetes, let’s go through key terms used in Kubernetes networking.

## **Key Terms in Kubernetes Networking**

When working with a complex system like K8s, it can get overwhelming fast! To help you understand and keep up with unfamiliar phrases and terms I’ve added here key terms you need to know. You’ll see many of these terms when discussing **K8s network** **structures** so ensure you know each one well!

### Service

**Kubernetes services** are **groups of pods** under a **common name**. Services have **stable IP addresses** and act as the point of access for external clients. Services are like conventional load balancers as they’re designed to distribute the traffic to a set of pods.

### Pods

Peas (containers) in a pod.

Pods are objects that consist of a **set of containers**. Projects that use these spaces are closely related to the service they provide. In essence, pods exist so you can simulate **application-specific** environments reflecting the use-case. Alternatively, you often use them as a **self-contained “logical host”**. This makes pods perfect for software development and enables teams to migrate projects quickly between sites or companies in dynamic working environments.

In general, pods help you form a single cohesive service building block. You often **create Pods** for projects and similarly **destroy** or **achieve** them to meet business needs. You can think of pods as non-persistent entities that can be scaled, modified and transferred as required. Each pod you create has its own **UID**and **IP address**. Pod can use these attributes to communicate with each other but not with containers of a different pod.

### Ingress

Ingress is a collection of **routing rules** that controls how external users access services. Each rule set is capable of**load balancing**, **SSL termination**, and name-based**virtual hosting** or **routing**. In essence, Ingress is capable of working at **layer 7**, this allows it to gather more information for intelligent routing through packet sniffing. For Ingress to function in a **Kubernetes cluster**, you need to have a component called an **ingress controller**. Some controller examples include; **NginX**, **HAProxy**, **Traefik**, etc. In any case they don’t start with the cluster so you’ll need to activate them.

So far, you’ve come to understand the basics and benefits of load balancing and how it works in a K8s environment. Now, I’ll show you how a service mesh works and handles load balancing.

## **How a Service Mesh Handles Load Balancing**

A [service mesh](https://techgenix.com/service-mesh/) helps manage all **service-to-service communications**. In your own case, you can use it to observe data as it resides on the critical path for every request being handled.

**Service meshes** can help you manage the traffic inside your cluster. In other words, it amplifies your application with a new process that handles **load balance requests**. In terms of service meshes, they look at protocols and discover IP address services automatically. Service meshes inspect connections like **gRPC**.

Load balancing in a **service mesh** utilizes algorithms to identify data throughput and **route traffic** across your network. In essence, it holds back or routes traffic around unhealthy resources to provide a robust service. Moreover, a service mesh helps reduce the risk of system downs from unbalanced network loading.

Now, I’ll go through a few policy types used for load balancing you can use to optimize your system.

## **Policies for Load Balancing**

Various methods or **policies** exist for load balancing that can **control traffic distribution**. To that end, they help carry out intelligent and efficient load balancing. Check out the **3 key policy types**you can use.

### 1. Round Robin

Round robin is the default load balancer [policy](https://techgenix.com/kubernetes-security-and-policy-management/) that distributes incoming traffic. In this case, the **server list rotates server connections** in the same order as the list. For most companies Round robin works the best in situations with minimal performance variation.

### 2. Least Connection

This is a **dynamic policy** where incoming traffic is routed to the backend server with the fewest active connections. It ensures that connections are **equally distributed** between backend servers.

### 3. IP Hash

This policy utilizes the **source IP address** of an incoming request as a **hashing key** to route client requests to the same backend server. It ensures that requests from a particular client are always routed to the same backend server as long as it’s available.

## Final Thoughts

Load balancing has progressed from simple **traffic management systems**to managing **complex systems**. Moreover, it can deliver a lot of value to a business that hosts demanding platforms like Kubernetes. This is because Kubernetes needs [dynamic resource allocation](https://techgenix.com/dynamic-memory-debate/) across many platforms for projects.

Load balancing is an essential part of keeping your **Kubernetes clusters** operational. Above all else, remember to configure your Kubernetes infrastructure to meet your needs. That is to say, you don’t have to stick to **default traffic management rules**. When you optimize your system you’ll have a hardy solution that’ll have less ‘server downs’ and is easier to maintain.

**What is a StatefulSet in Kubernetes?**

A StatefulSet is a controller that helps you deploy and scale groups of Kubernetes pods.

When using Kubernetes, most of the time you don’t care how your pods are scheduled, but sometimes you care that pods are deployed in order, that they have a persistent storage volume, or that they have a unique, stable network identifier across restarts and reschedules. In those cases, StatefulSets can help you accomplish your objective.

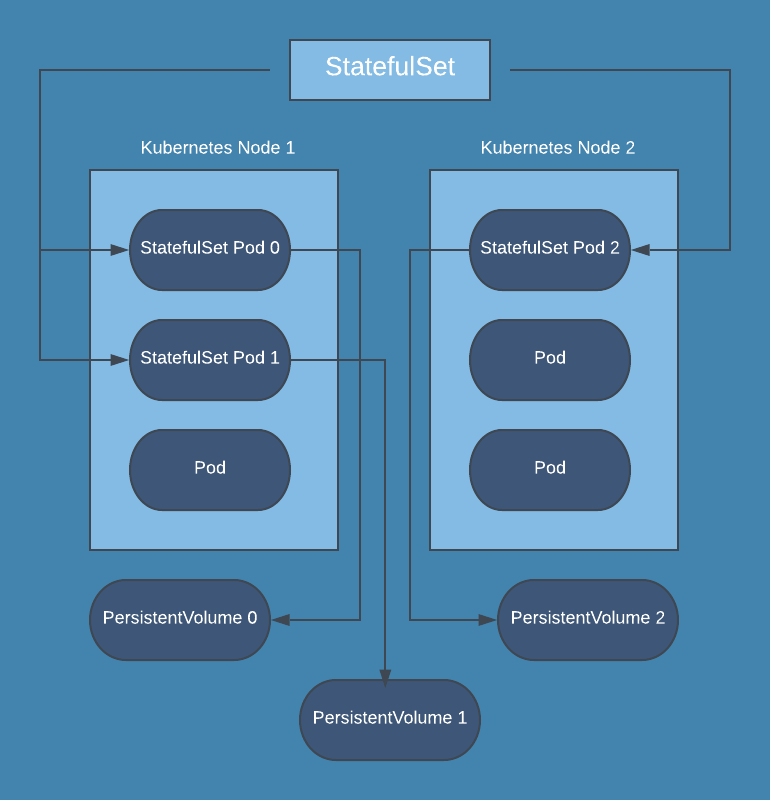
Each pod created by the StatefulSet has an ordinal value (0 through # replicas - 1) and a stable network ID (which is statefulsetname-ordinal) assigned to it. You can also create a **VolumeClaimTemplate** in the manifest file that will create a [persistent volume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) for each pod. When pods are deployed by a StatefulSet, they will go in order from 0 to the final pod and require that each pod is Running and Ready before creating the next pod.

## **When to use StatefulSet**

Some examples of reasons you’d use a StatefulSet include:

* A Redis pod that has access to a volume, but you want it to maintain access to the same volume even if it is redeployed or restarted
* A Cassandra cluster and have each node maintain access to its data
* A webapp that needs to communicate with its replicas using known predefined network identifiers

StatefulSets are available in Kubernetes 1.9 and later.



**How to create a StatefulSet**

Like everything else in Kubernetes, StatefulSets can be configured using a manifest file. StatefulSets require a [headless service](https://kubernetes.io/docs/concepts/services-networking/service/#headless-services), which can be created in the same manifest file.

apiVersion: v1

kind: Service

metadata:

name: redis

namespace: default

labels:

app: redis

spec:

ports:

- port: 6379

protocol: TCP

selector:

app: redis

type: ClusterIP

clusterIP: None

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: redis

spec:

selector:

matchLabels:

app: redis

serviceName: "redis"

replicas: 1

template:

metadata:

labels:

app: redis

spec:

containers:

- name: redis

image: redis:5.0.4

command: ["redis-server", "--appendonly", "yes"]

ports:

- containerPort: 6379

name: web

volumeMounts:

- name: redis-aof

mountPath: /data

volumeClaimTemplates:

- metadata:

name: redis-aof

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "gp2"

resources:

requests:

storage: 1Gi

This example first creates a headless service that can be used by the StatefulSet, and then creates a StatefulSet that maintains one Redis pod connected to a volume. To actually create these resources, you’ll run the following command:

kubectl apply -f statefulset.yaml

**How to update a Kubernetes StatefulSet**

Updating a StatefulSet is easy. Just make changes to your manifest file, and then run the same command you used to create the StatefulSet:

kubectl apply -f statefulset.yam

What happens when you run this command depends on the update strategy you’ve selected for your StatefulSet. There are two update strategies:

**OnDelete**

If you’ve chosen  OnDelete  as the value of  .spec.updateStrategy.type  pods will not be replaced when you apply the manifest. Instead, you will have to manually delete existing StatefulSet pods before the new version will be created.

**RollingUpdate**

If you’ve chosen  RollingUpdate , when you apply the manifest, StatefulSet pods will be removed and then be replaced in reverse ordinal order. With this update strategy, you can also specify  .spec.updateStrategy.rollingUpdate.partition  to an ordinal value, and all pods with a higher ordinal value will be replaced with the new version, while old ones will be retained. This allows you to perform phased rollouts. Check out our blog on [how to configure rolling updates on Kubernetes](https://www.bluematador.com/blog/kubernetes-deployments-rolling-update-configuration).

[**What is a Headless Service?**](https://blog.knoldus.com/what-is-headless-service-setup-a-service-in-kubernetes/#what-is-a-headless-service)

In the default behaviour of Kubernetes we assign as Internal IP address to the service.And with this IP address the service will proxy and load-balance the requests to the pods.We can choose which kind of service type we need while deploying it. These service types are-

* ClusterIP- For exposing the server on cluster-internal IP address
* NodePort- For exposing the service through a static port on the node
* LoadBalancer- to expose the service using an external load-balancer

**What is a Headless Service?**

When there is no need of load balancing or single-service IP addresses.We create a headless service which is used for creating a service grouping. That does not allocate an IP address or forward traffic.So you can do this by explicitly setting ClusterIP to “None” in the mainfest file, which means no cluster IP is allocated.

For example, if you host MongoDB on a single pod. And you need a service definition on top of it for taking care of the pod restart.And also for acquiring a new IP address. But you don’t want any load balancing or routing. You just need the service to patch the request to the back-end pod. So then you use Headless Service since it does not have an IP.

Kubernetes allows clients to discover pod IPs through DNS lookups. Usually, when you perform a DNS lookup for a service, the DNS server returns a single IP which is the service’s cluster IP. But if you don’t need the cluster IP for your service, you can set ClusterIP to None , then the DNS server will return the individual pod IPs instead of the service IP.Then client can connect to any of them.

### Use Cases of Headless Service-

* Create Stateful service
* Deploying RabbitMQ to Kubernetes requires a stateful set for RabbitMQ cluster nodes.
* Deployment of Relational databases

#### **Deployment manifest file:-**

apiVersion: apps/v1

kind: Deployment

metadata:

name: app

labels:

app: server

spec:

replicas: 3

selector:

matchLabels:

app: web

template:

metadata:

labels:

app: web

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

#### **Regular Service:-**

apiVersion: v1

kind: Service

metadata:

name: regular-service

spec:

selector:

app: web

ports:

- protocol: TCP

port: 80

targetPort: 8080

#### **Headless Service:-**

apiVersion: v1

kind: Service

metadata:

name: headless-svc

spec:

clusterIP: None

selector:

app: web

ports:

- protocol: TCP

port: 80

targetPort: 8080

#### **Create all resources and run the pod**

kubectl create -f deployment.yaml

kubectl create -f regular-service.yaml

kubectl create -f headless-service.yaml

kubectl run temporary --image=radial/busyboxplus:curl -i --tty

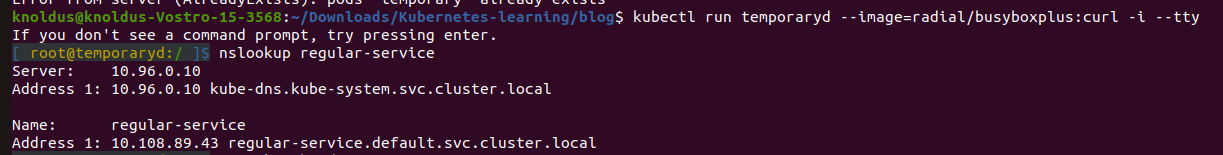
#### **Cleanup the used resources**

kubectl delete svc regular-service

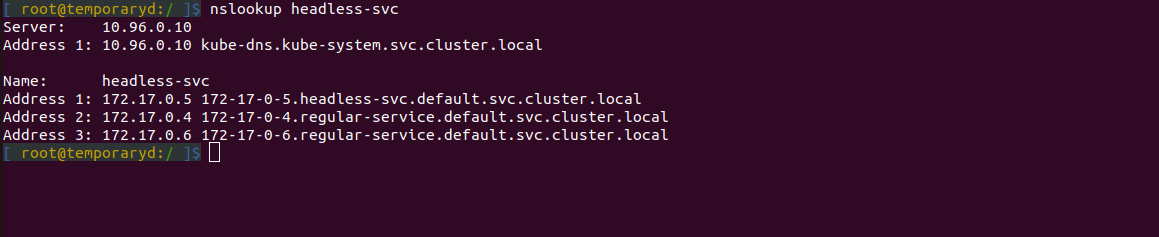
kubectl delete svc headless-svc

kubectl delete deployment app

kubectl delete po temporary



##### **The DNS server returns three different IPs for the pods**



### Conclusions

With a Headless Service, clients can connect to it’s pods by connecting to the service’s DNS name. But using headless services, DNS returns the pod’s IPs and client can connect directly to the pods instead via the service proxy.

### Security

### Authentication, Authorization, and Admission Control

Authentication and authorization play a very vital role in securing applications. These two terms are often used interchangeably but are very different. Authentication validates the identity of a user. Once the identity is validated, authorization is used to check whether the user has the privileges to perform the desired action. Authentication uses something the user knows to verify their identity; in the simplest form, this is a username and password. Once the application verifies the user's identity, it checks what resources the user has access to. In most cases, this is a variation of an access control list. Access control lists for the user are compared with the request attributes

### Understanding Authentication & Authorization in kubernetes

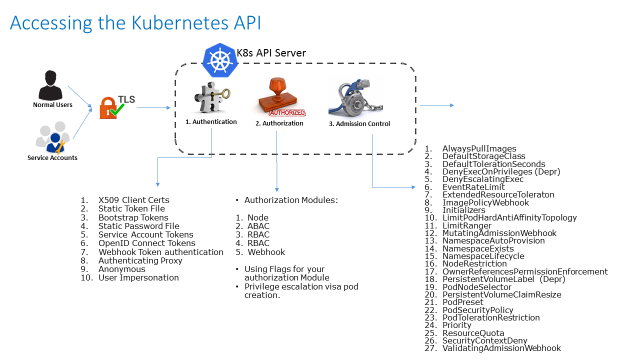
**Authentication**– How User’s access should be allowed? The process or action of verifying the identity of a user or process.  
**Authorization** – What Access and till what extent should be accessible to user

## **Official ref for Authentication**

* https://kubernetes.io/docs/reference/access-authn-authz/authentication/

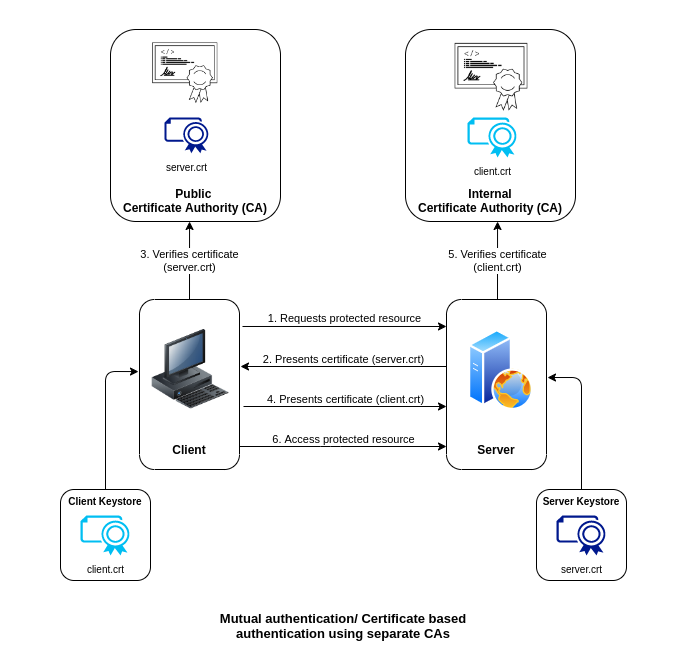
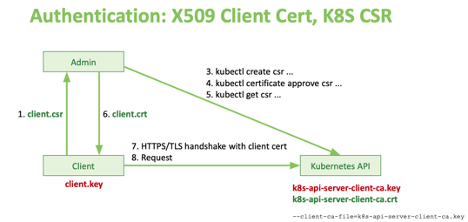
## **Method of Authentication in kubernetes**

* Certificate
* Token
* OpenID
* Web Hook

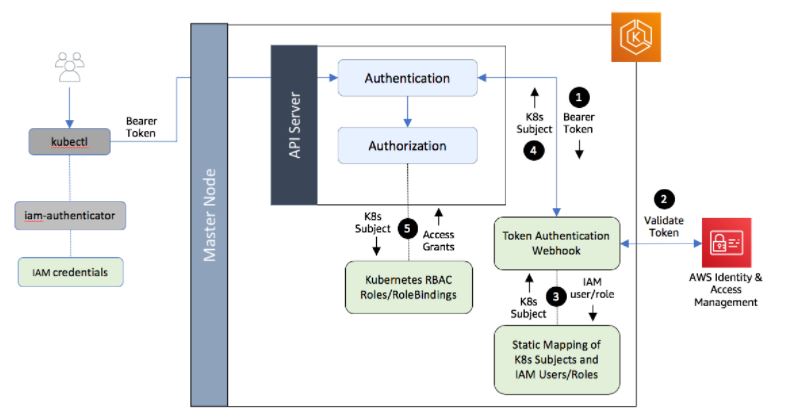
[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/Authentication-Authorization-kubernetes-tutorials-2.png)

## **How Certificate Based Auth Works in kubernetes?**

* User (or administrator on behalf of user) creates a private key.
* User/administrator generates a certificate signing request (CSR).
* Administrator approves the request and signs it with their CA.
* Administrator provides the resulting certificate back to the user.

[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/Authentication-Authorization-kubernetes-tutorials-1.png)[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/Authentication-Authorization-kubernetes-tutorials-3.png)

## **How Token Based Auth Works in kubernetes?**



## **How to create user in kubernetes?**

# USER run these commands in Workstation

# Create a pvt key

$ openssl genrsa -out employee.key 2048

# Create CSR file

$ openssl req -new -key employee.key -out employee.csr -subj "/CN=employee/O=bitnami"

# How to send a CSR file to CA (Master Admin or K8s admin)

- Send via manual way eg. email

- csr api

# Admin run these commands in Workstation

$ openssl x509 -req -in employee.csr -CA /etc/kubernetes/pki/ca.crt -CAkey /etc/kubernetes/pki/ca.key -CAcreateserial -out employee.crt -days 500

# Admin would send employee.crt to USER.

- Send via manual way eg. email

- csr api - they can download self

# USER would set employee.key & employee.crt in CONFIG file.

$ kubectl config set-credentials employee --client-certificate=/root/employee.crt --client-key=/root/employee.key

$ kubectl config view

$ kubectl config set-context employee-context --cluster=kubernetes --namespace=office --user=employee

$ kubectl config view

$ kubectl create namespace office

$ kubectl --context=employee-context get pods

[root@rajesh ~]# kubectl --context=employee-context get pods

Error from server (Forbidden): pods is forbidden: User "employee" cannot list resource "pods" in API group "" in the namespace "office"

# Only we have enabled employee authentication. He has no rights on K8s.

## **What are the Methods of Authorization in kubernetes?**

* Node
* ABAC
* RBAC [ FOCUS ]
* Webhook

# **Official ref for Authorization**

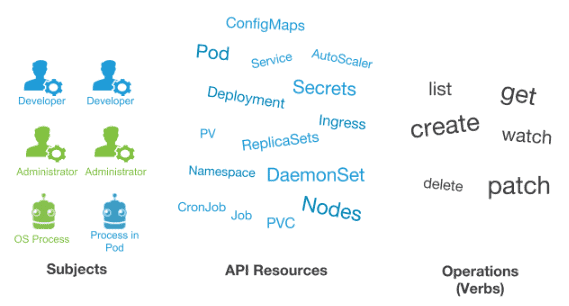
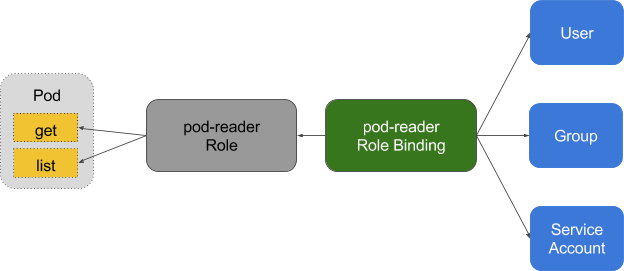
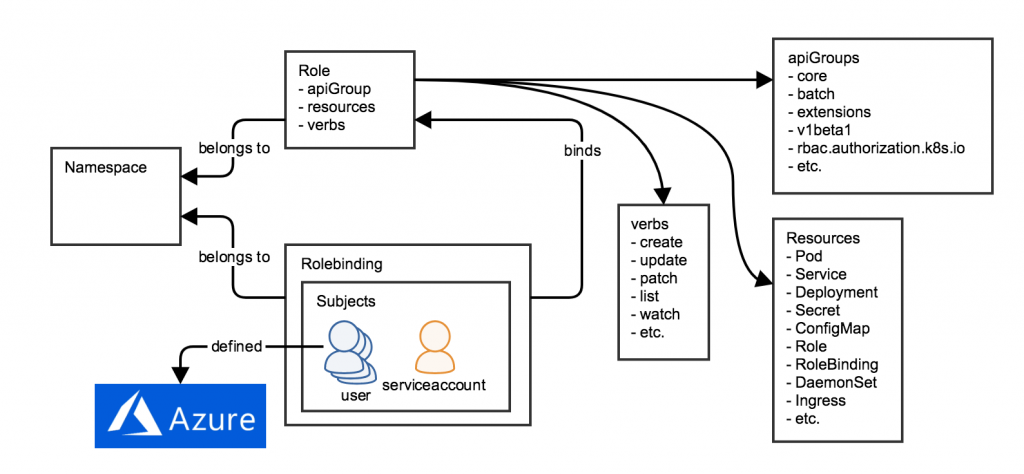
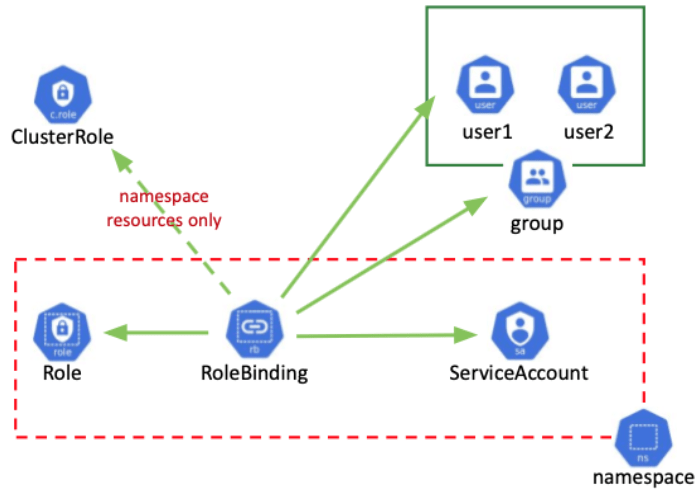
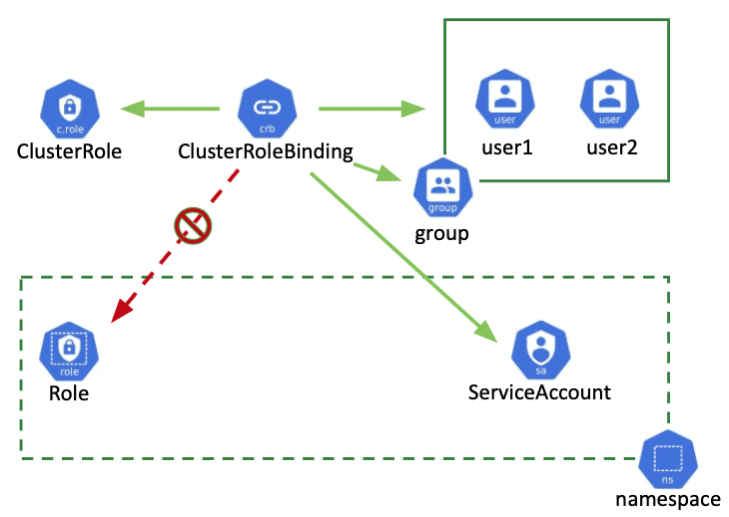
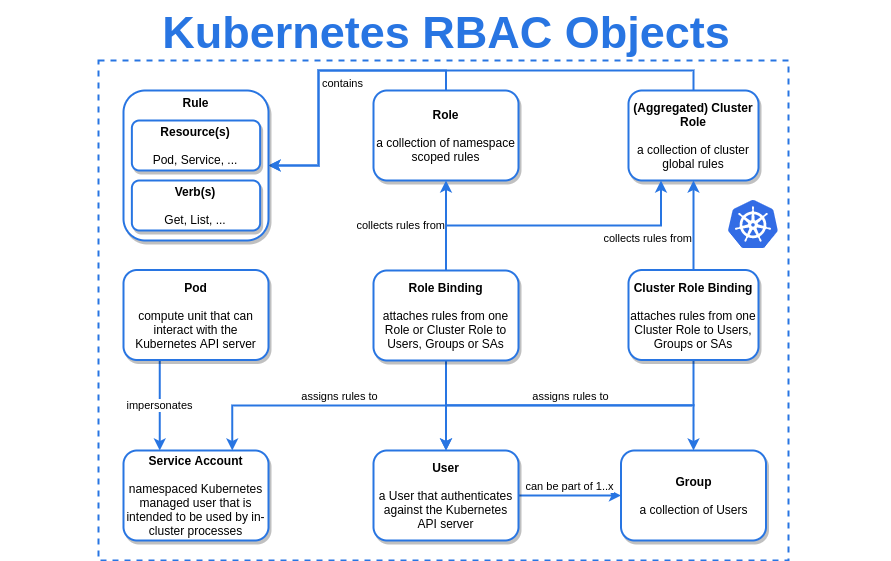
* https://kubernetes.io/docs/reference/access-authn-authz/authorization/

## **How to Authorized user in kubernetes clustor?**

WHOM – USER or GROUP  
WHAT – verbs: [“get”, “list”, “watch”, “create”, “update”, “patch”, “delete”] # You can also use [“\*”]  
WHERE – API Resources or API Group $ kubectl api-resources  
How???

* Node
* ABAC
* RBAC [ FOCUS ]
* Webhook

## **How RBAC works in kubernetes?**

[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/rbac-authorization-kubernetes-tutorials-1.png)[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/rbac-authorization-kubernetes-tutorials-2.png)[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/rbac-authorization-kubernetes-tutorials-5.png)[](https://www.devopsschool.com/blog/wp-content/uploads/2021/07/rbac-authorization-kubernetes-tutorials-6.png)

### - WHOM - USER or GROUPWHAT - verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # You can also use ["\*"] WHERE - API Resources or API Group $ kubectl api-resourcesHow???- Node- ABAC - RBAC [ FOCUS ] - Webhook How RBAC works in kubernetes?ROlE BASED ACCESS CONTROL-----------------------------ROLE TYPE- role: This to be used for graning access to Namespace- clustorrole: This to be used for graning access to ClustorUser Type- User: like me.- Service Account: a user which is used by systemrole contains- WHAT - verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # You can also use ["\*"] - WHERE - API Resources or API Group $ kubectl api-resources &WHOM - USER OR& GROUP or& Service Account #using ROLE BINDINGclustorrole contains- WHAT - verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # You can also use ["\*"] - WHERE - API Resources or API Group $ kubectl api-resources &WHOM - USER OR& GROUP or& Service Account #using CLUSTORROLE BINDING API resources for working with RBAC- serviceaccounts- roles- clusterroles- rolebindings- clusterrolebindingskind: RoleapiVersion: rbac.authorization.k8s.io/v1beta1metadata: namespace: office name: deployment-managerrules:- apiGroups: ["", "extensions", "apps"] resources: ["deployments", "replicasets", "pods"] verbs: ["get", "list", "watch", "create", "update", "patch", "delete"] # You can also use ["\*"]kind: RoleBindingapiVersion: rbac.authorization.k8s.io/v1beta1metadata: name: deployment-manager-binding namespace: officesubjects:- kind: User name: employee apiGroup: ""roleRef: kind: Role name: deployment-manager apiGroup: ""$ kubectl --context=employee-context run nginx --image=nginx$ kubectl --context=employee-context get pods[root@rajesh ~]# kubectl --context=employee-context run nginx --image=nginxpod/nginx created[root@rajesh ~]# kubectl get pod -n=officeNAME READY STATUS RESTARTS AGEnginx 1/1 Running 0 18s[root@rajesh ~]# ubectl --context=employee-context run nginx --image=nginx^C[root@rajesh ~]# kubectl --context=employee-context get podsNAME READY STATUS RESTARTS AGEnginx 1/1 Running 0 54s[root@rajesh ~]# kubectl --context=employee-context get svcError from server (Forbidden): services is forbidden: User "employee" cannot list resource "services" in API group "" in the namespace "office"[root@rajesh ~]#apiVersion: rbac.authorization.k8s.io/v1kind: ClusterRolemetadata: # "namespace" omitted since ClusterRoles are not namespaced name: secret-readerrules:- apiGroups: [""] # # at the HTTP level, the name of the resource for accessing Secret # objects is "secrets" resources: ["secrets"] verbs: ["get", "watch", "list"]apiVersion: rbac.authorization.k8s.io/v1# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.kind: ClusterRoleBindingmetadata: name: read-secrets-globalsubjects:- kind: Group name: manager # Name is case sensitive apiGroup: rbac.authorization.k8s.ioroleRef: kind: ClusterRole name: secret-reader apiGroup: rbac.authorization.k8s.io

### Security-Contexts

### Basic Networking

### 

### CNI 🡪 container network manager

### Overlay  network 🡪 The overlay network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely when encryption is enabled. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

## **Kubernetes Networking**

**Since a Kubernetes cluster consists of various nodes and pods, understanding how they communicate between them is essential. The Kubernetes networking model supports different types of open source implementations.**

The Kubernetes networking model, on the other hand, natively supports multi-host networking in which pods are able to communicate with each other by default, regardless of which host they live in.

Kubernetes does not provide a default network implementation, it only enforces a model for third-party tools to implement. There is a variety of implementations nowadays, below we list some popular ones.

1. Flannel - a very simple overlay network that satisfies the Kubernetes requirements. Flannel runs an agent on each host and allocates a subnet lease to each of them out of a larger, preconfigured address space. Flannel creates a flat network called as overlay network which runs above the host network.
2. Project Calico - an open source container networking provider and network policy engine. Calico provides a highly scalable networking and network policy solution for connecting Kubernetes pods based on the same IP networking principles as the internet. Calico can be deployed without encapsulation or overlays to provide high-performance, high-scale data center networking.
3. Weave Net - a cloud native networking toolkit which provides a resilient and simple to use (does not require any configuration) network for Kubernetes and its hosted applications. It provides various functionalities like scaling, service discovery, performance without complexity and secure networking.
4. Other options include Cisco ACI , Cilium , Contiv , Contrail , Kube-router , Nuage , OVN , Romana , VMWare NSX-T with NSX-T Container Plug-in (NCP) . Some tools even support using multiple implementations, such as Huawei CNI-Genie and Multus .

## Pod to Pod Communication

Kubernetes follows an “IP-per-pod” model where each pod get assigned an IP address and all containers in a single pod share the same network namespaces and IP address. Containers in the same pod can therefore reach each other’s ports via localhost:.

However, it is not recommended to communicate directly with a pod via its IP address due to pod’s volatility (a pod can be killed and replaced at any moment).

Instead, use a Kubernetes service which represents a group of pods acting as a single entity to the outside. Services get allocated their own IP address in the cluster and provide a reliable entry point.

**Kubernetes services allow grouping pods under a common access policy (for example, load-balanced). The service gets assigned a virtual IP which pods outside the service can communicate with. Those requests are then transparently proxied (via the kube-proxy component that runs on each node) to the pods inside the service. Different proxy-modes are supported:**

* iptables: kube-proxy installs iptables rules trap access to service IP addresses and redirect them to the correct pods.
* userspace: kube-proxy opens a port (randomly chosen) on the local node. Requests on this “proxy port” get proxied to one of the service’s pods (as retrieved from Endpoints API).
* ipvs (from Kubernetes 1.9): calls netlink interface to create ipvs rules and regularly synchronizes them with the Endpoints API.

## **DNS for Services and Pods**

Kubernetes provides its own DNS service to resolve domain names inside the cluster in order for pods to communicate with each other. This is implemented by deploying a regular Kubernetes service which does name resolution inside the cluster, and configuring individual containers to contact the DNS service to resolve domain names. Note that this “internal DNS” is compatible and expected to run along with the cloud provider’s DNS service.

Every service gets assigned a DNS name which resolves to the cluster IP of the service. The naming convention includes the service name and its namespace. For example:

* my-service.my-namespace.svc.cluster.local

A pod inside the same namespace as the service does not need to fully qualify its name, for example a pod in my-namespace could lookup this service with a DNS query for my-service , while a pod outside my-namespace would have to query for my-service.my-namespace .

For headless services (without a cluster IP), the DNS name resolves to the set of IPs of the pods which are part of the service. The caller can then use the set of IPs as it sees fit (for example round-robin).

By default pods get assigned a DNS name which includes the pod’s IP address and namespace. In order to assign a more meaningful DNS name, the pod’s specification can specify a hostname and subdomain:

### Ingress / Egress

The bulk of securing network traffic typically revolves around defining egress and ingress rules. From the point of view of a Kubernetes pod, **ingress** is incoming traffic to the pod, and **egress** is outgoing traffic from the pod. In Kubernetes network policy, you create ingress and egress “allow” rules independently (egress, ingress, or both).

#### Default deny/allow behavior

**Default allow** means all traffic is allowed by default, unless otherwise specified. **Default deny** means all traffic is denied by default, unless explicitly allowed.

### HPA

## Kubernetes autoscaling basics

Before we go in-depth on HPA, we need to review Kubernetes autoscaling in general. Autoscaling is a method of automatically scaling K8s workloads up or down based on historical resource usage. Autoscaling in Kubernetes has three dimensions:

1. Horizontal Pod Autoscaler (HPA):adjusts the number of replicas of an application.
2. Cluster Autoscaler:adjusts the number of nodes of a cluster.
3. Vertical Pod Autoscaler (VPA):adjusts the resource requests and limits of a container.

The different autoscalers work at one of two Kubernetes layers

* Pod level:The HPA and VPA methods take place at the pod level. Both HPA and VPA will scale the available resources or instances of the container.
* Cluster level:The Cluster Autoscaler falls under the Cluster level, where it scales up or down the number of nodes inside your cluster.

Now that we have summarized the basics, let’s take a closer look at HPA.

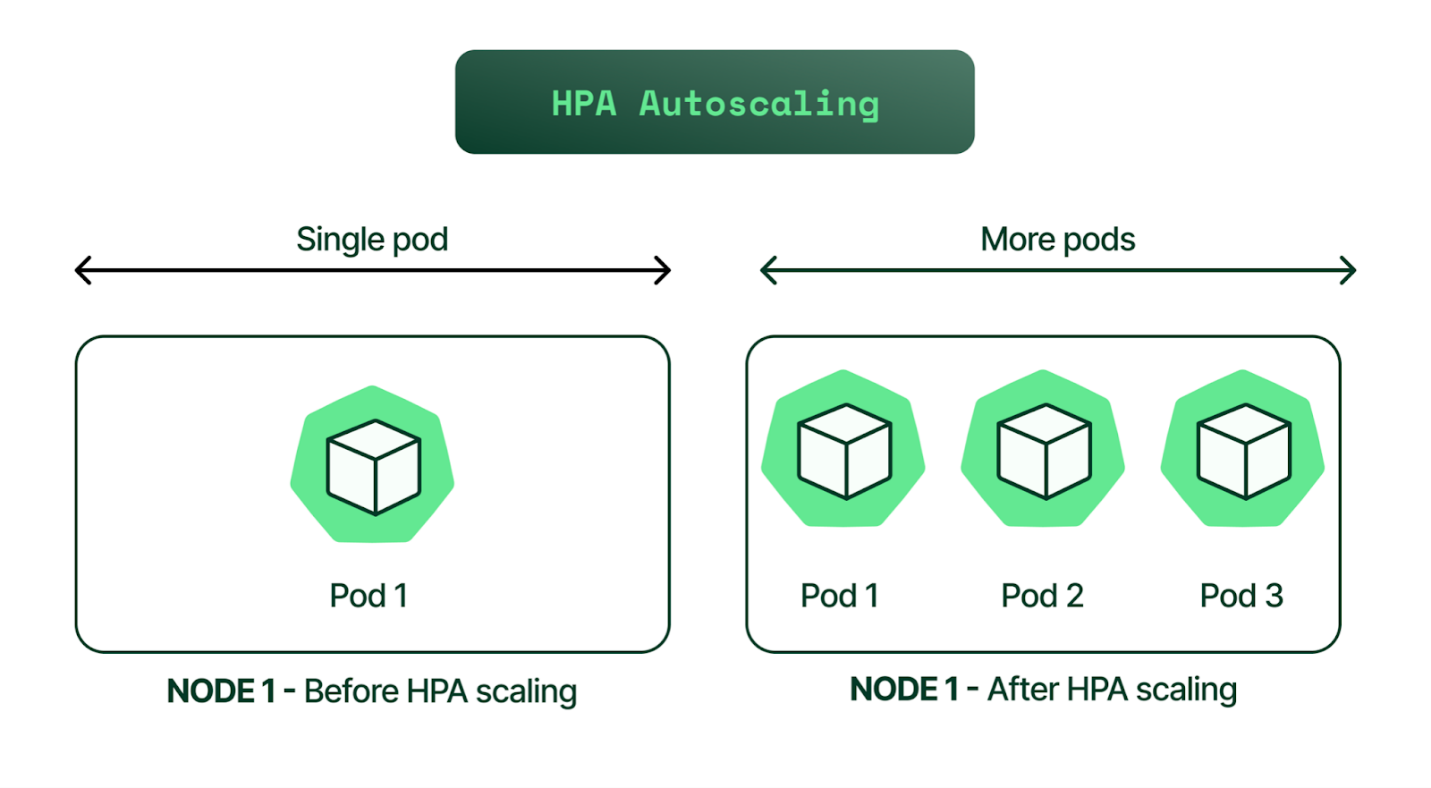
## What is HPA?

HPA is a form of autoscaling that increases or decreases the number of pods in a replication controller, deployment, replica set, or stateful set based on CPU utilization—the scaling is horizontal because it affects the number of instances rather than the resources allocated to a single container.

HPA can make scaling decisions based on custom or externally provided metrics and works automatically after initial configuration. All you need to do is define the MIN and MAX number of replicas.

Once configured, the Horizontal Pod Autoscaler controller is in charge of checking the metrics and then scaling your replicas up or down accordingly. By default, HPA checks metrics every 15 seconds.

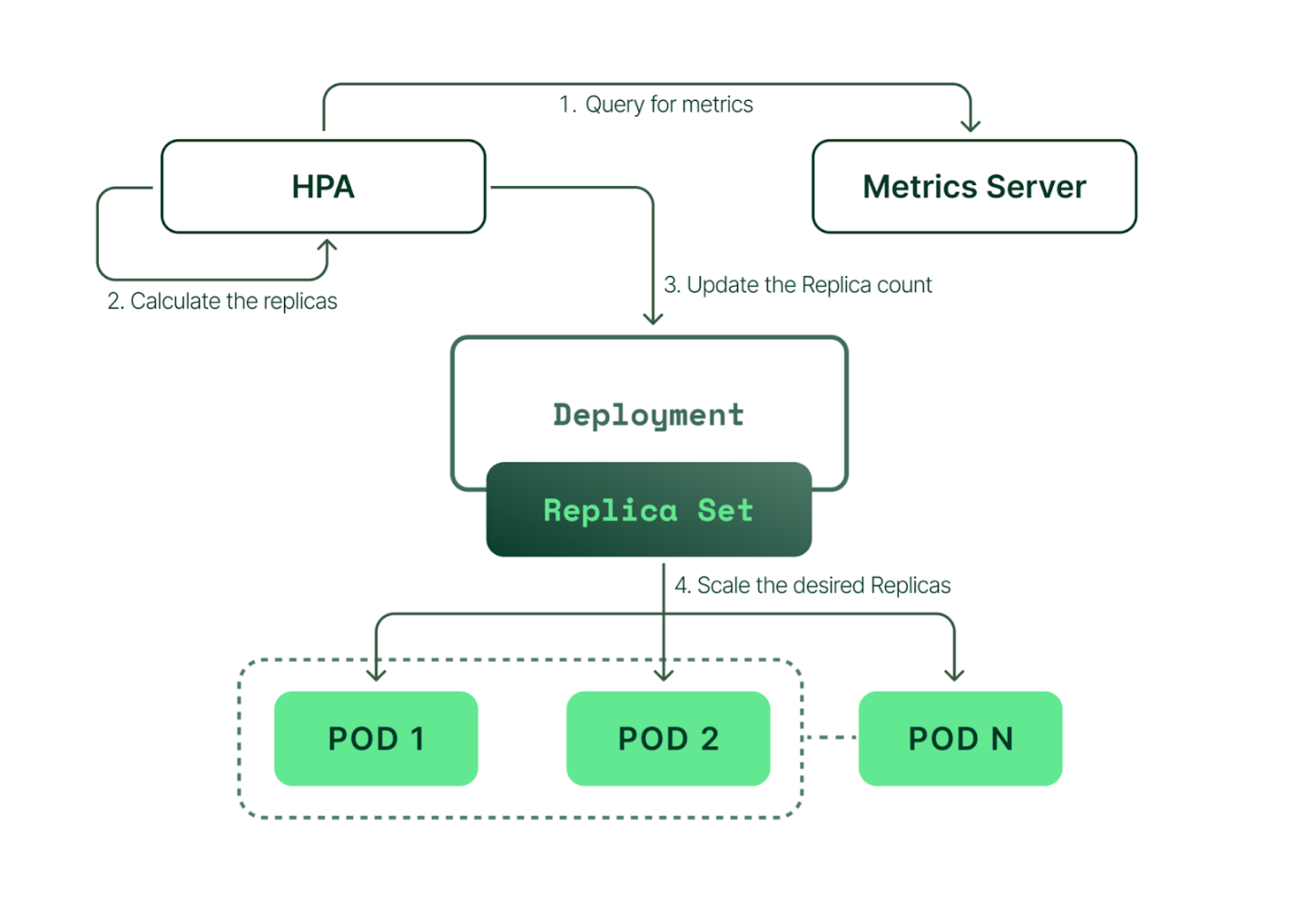
To check metrics, HPA depends on another Kubernetes resource known as the Metrics Server. The Metrics Server provides standard resource usage measurement data by capturing data from “kubernetes.summary\_api” such as CPU and memory usage for nodes and pods. It can also provide access to custom metrics (that can be collected from an external source) like the number of active sessions on a load balancer indicating traffic volume.



While the HPA scaling process is automatic, you can also help account for predictable load fluctuations in some cases. For example, you can:

* Adjust replica count based on the time of the day.
* Set different capacity requirements for weekends or off-peak hours.
* Implement an event-based replica capacity schedule (such as increasing capacity upon a code release).

## How does HPA work?



Overview of HPA

In simple terms, HPA works in a “check, update, check again” style loop. Here’s how each of the steps in that loop work.

1. HPA continuously monitors the metrics server for resource usage.
2. Based on the collected resource usage, HPA will calculate the desired number of replicas required.
3. Then, HPA decides to scale up the application to the desired number of replicas.
4. Finally, HPA changes the desired number of replicas.
5. Since HPA is continuously monitoring, the process repeats from Step 1.

## Limitations of HPA

While HPA is a powerful tool, it’s not ideal for every use case and can’t address every cluster resource issue. Here are the most common examples:

* One of HPA’s most well-known limitations is that it does not work with DaemonSets.
* If you don’t [efficiently set CPU and memory limits on pods](https://blog.kubecost.com/blog/requests-and-limits/), your pods may terminate frequently or, on the other end of the spectrum, you’ll waste resources.
* If the cluster is out of capacity, HPA can’t scale up until new nodes are added to the cluster. Cluster Autoscaler (CA) can automate this process. We have an article dedicated to CA; however, below is a quick contextual explanation.

Cluster Autoscaler (CA) automatically adds or removes nodes in a cluster based on resource requests from pods. Unlike HPA, Cluster Autoscaler doesn't look at memory or CPU available when it triggers the autoscaling. Instead, Cluster Autoscaler reacts to events and checks for any unscheduled pods every 10 seconds.

## EKS Example: How to Implement HPA

To help you get started with HPA, let’s walk through some practical examples. We’ll work through the following steps one-by-one:

1. Create an EKS cluster
2. Install the Metrics Server
3. Deploy a sample application
4. Install Horizontal Pod Autoscaler
5. Monitor HPA events
6. Decrease the load

### Step 1: create an EKS cluster

For this step, we will use [AWS EKS](https://aws.amazon.com/eks/) (Amazon's managed Kubernetes service), so make sure you have access to your AWS account. We’ll use eksctl, a simple CLI tool for creating and managing clusters on EKS. It is written in Go and uses CloudFormation in the background.

The EKS cluster kubeconfig file will be stored in the local directory (of your workstation or laptop), and if the command is successful, you will see a ready status. To begin, we’ll run the eksctl create cluster command below (using Kubernetes version 1.20 in this example).

$ eksctl create cluster --name example-hpa-autoscaling --version 1.20 --region us-west-2 --nodegroup-name hpa-worker-instances --node-type c5.large --nodes 1

2021-08-30 12:52:24 [ℹ] eksctl version 0.60.0

2021-08-30 12:52:24 [ℹ] using region us-west-2

2021-08-30 12:52:26 [ℹ] setting availability zones to [us-west-2a us-west-2b us-west-2d]

2021-08-30 12:52:26 [ℹ] subnets for us-west-2a - public:192.168.0.0/19 private:192.168.96.0/19

2021-08-30 12:52:26 [ℹ] subnets for us-west-2b - public:192.168.32.0/19 private:192.168.128.0/19

2021-08-30 12:52:26 [ℹ] subnets for us-west-2d - public:192.168.64.0/19 private:192.168.160.0/19

2021-08-30 12:52:26 [ℹ] nodegroup "hpa-worker-instances" will use "" [AmazonLinux2/1.20]

2021-08-30 12:52:26 [ℹ] using Kubernetes version 1.20

2021-08-30 12:52:26 [ℹ] creating EKS cluster "example-hpa-autoscaling" in "us-west-2" region with managed nodes

...

...

2021-08-30 12:53:29 [ℹ] waiting for CloudFormation stack

2021-08-30 13:09:00 [ℹ] deploying stack "eksctl-example-hpa-autoscaling-nodegroup-hpa-worker-instances"

2021-08-30 13:09:00 [ℹ] waiting for CloudFormation stack

2021-08-30 13:12:11 [ℹ] waiting for the control plane availability...

2021-08-30 13:12:11 [✔] saved kubeconfig as "/Users/karthikeyan/.kube/config"

2021-08-30 13:12:11 [ℹ] no tasks

2021-08-30 13:12:11 [✔] all EKS cluster resources for "example-hpa-autoscaling" have been created

2021-08-30 13:12:13 [ℹ] nodegroup "hpa-worker-instances" has 1 node(s)

2021-08-30 13:12:13 [ℹ] node "ip-192-168-94-150.us-west-2.compute.internal" is ready

2021-08-30 13:12:13 [ℹ] waiting for at least 1 node(s) to become ready in "hpa-worker-instances"

2021-08-30 13:12:13 [ℹ] nodegroup "hpa-worker-instances" has 1 node(s)

2021-08-30 13:12:13 [ℹ] node "ip-192-168-94-150.us-west-2.compute.internal" is ready

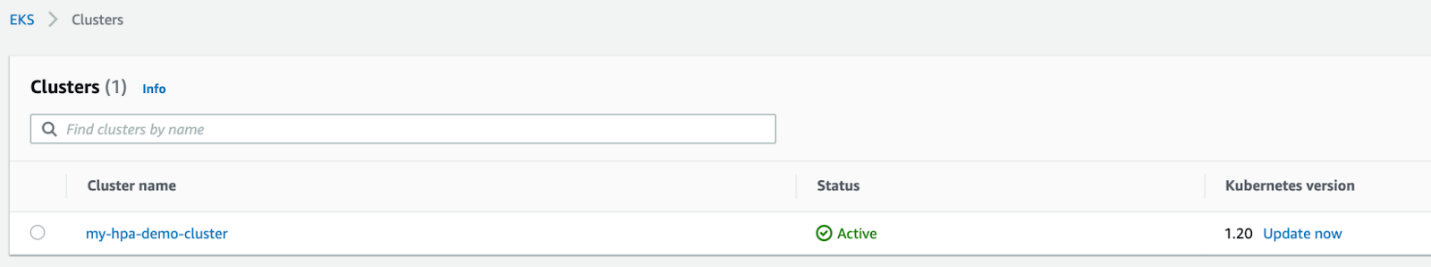
2021-08-30 13:14:20 [ℹ] kubectl command should work with "/Users/karthikeyan/.kube/config", try 'kubectl get nodes'

2021-08-30 13:14:20 [✔] EKS cluster "example-hpa-autoscaling" in "us-west-2" region is ready

Next, verify the cluster:

$ aws eks describe-cluster --name my-hpa-demo-cluster --region us-west-2

You can also check by logging into the AWS console:



To get the cluster context to log in, read your local kubeconfig like below:

$ cat ~/.kube/config |grep "current-context"

current-context: bob@my-hpa-demo-cluster.us-west-2.eksctl.io

List the nodes and pods:

$ kubectx bob@example-hpa-autoscaling.us-west-2.eksctl.io

Switched to context "bob@example-hpa-autoscaling.us-west-2.eksctl.io".

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

ip-192-168-94-150.us-west-2.compute.internal Ready 15m v1.20.4-eks-6b7464

$ kubectl get pods --all-namespaces

NAMESPACE NAME READY STATUS RESTARTS AGE

kube-system aws-node-f45pg 1/1 Running 0 15m

kube-system coredns-86d9946576-2h2zk 1/1 Running 0 24m

kube-system coredns-86d9946576-4cvgk 1/1 Running 0 24m

kube-system kube-proxy-864g6 1/1 Running 0 15m

kubectx is a tool used to switch between various Kubernetes clusters. Now that we have the EKS cluster up and running, next, we need to deploy the Metrics Server.

### Install the Metrics Server

We can check if we have set up the Metrics Server in our EKS cluster by using the following command:

$ kubectl top pods -n kube-system

error: Metrics API not available

📝 Note: For this process, we have created a directory named “/Users/bob/hpa/” on our local laptop and saved all configuration files used in this article to that location. We recommend creating a similar directory on your local workstation to download all required files (mentioned below).

Let’s install the Metrics Server. Download the YAML files from: <https://github.com/nonai/k8s-example-files/tree/main/metrics-server>

$ cd /Users/bob/hpa/metrics-server && ls -l

total 56

-rw-r--r-- 1 bob 1437157072 136 Aug 30 13:48 0-service-account.yaml

-rw-r--r-- 1 bob 1437157072 710 Aug 30 13:48 1-cluster-roles.yaml

-rw-r--r-- 1 bob 1437157072 362 Aug 30 13:48 2-role-binding.yaml

-rw-r--r-- 1 bob 1437157072 667 Aug 30 13:48 3-cluster-role-bindings.yaml

-rw-r--r-- 1 bob 1437157072 254 Aug 30 13:48 4-service.yaml

-rw-r--r-- 1 bob 1437157072 1659 Aug 30 13:48 5-deployment.yaml

-rw-r--r-- 1 bob 1437157072 331 Aug 30 13:48 6-api-service.yaml

Once you have downloaded the file, run the below command to create all of the resources:

$ kubectl apply -f .

serviceaccount/metrics-server created

clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader created

clusterrole.rbac.authorization.k8s.io/system:metrics-server created

rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created

clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-delegator created

clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created

service/metrics-server created

deployment.apps/metrics-server created

apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created

Verify the Metrics Server deployment:

$ kubectl get pods --all-namespaces

NAMESPACE NAME READY STATUS RESTARTS AGE

kube-system aws-node-982kv 1/1 Running 0 14m

kube-system aws-node-rqbg9 1/1 Running 0 13m

kube-system coredns-86d9946576-9k6gx 1/1 Running 0 25m

kube-system coredns-86d9946576-m67h6 1/1 Running 0 25m

kube-system kube-proxy-lcklc 1/1 Running 0 13m

kube-system kube-proxy-tk96q 1/1 Running 0 14m

kube-system metrics-server-9f459d97b-q5989 1/1 Running 0 41s

List services in the kube-system namespace:

$ kubectl get svc -n kube-system

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

kube-dns ClusterIP 10.100.0.10 53/UDP,53/TCP 26m

metrics-server ClusterIP 10.100.66.231 443/TCP 82s

Use kubectl to see CPU and memory metrics:

$ kubectl top pods -n kube-system

NAME CPU(cores) MEMORY(bytes)

aws-node-982kv 4m 40Mi

aws-node-rqbg9 5m 39Mi

coredns-86d9946576-9k6gx 2m 8Mi

coredns-86d9946576-m67h6 2m 8Mi

kube-proxy-lcklc 1m 11Mi

kube-proxy-tk96q 1m 11Mi

metrics-server-9f459d97b-q5989 3m 15Mi

## Deploy a sample application

Now, we are going to use a custom Docker image that runs on Apache and PHP to… . The Docker image is publicly accessible, so we can refer to it directly from our Kubernetes deployment.

Let’s deploy the application as part of our Kubernetes cluster, maintaining a minimum of 1 replica and a maximum of 10 replicas. Below is the configuration, which you can save as “deployment.yml”.

$ cd /Users/bob/hpa/

$ cat deployment.yml

apiVersion: apps/v1

kind: Deployment

metadata:

name: hpa-demo-deployment

spec:

selector:

matchLabels:

run: hpa-demo-deployment

replicas: 1

template:

metadata:

labels:

run: hpa-demo-deployment

spec:

containers:

- name: hpa-demo-deployment

image: k8s.gcr.io/hpa-example

ports:

- containerPort: 80

resources:

limits:

cpu: 500m

requests:

cpu: 200m

Apply it by running:

$ kubectl apply -f deployment.yml

deployment.apps/hpa-demo-deployment created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

hpa-demo-deployment-6b988776b4-b2hkb 1/1 Running 0 20s

We created the deployment successfully. Next, let’s get the list of deployment status:

$ kubectl get deploy

NAME READY UP-TO-DATE AVAILABLE AGE

hpa-demo-deployment 1/1 1 1 9s

### Comprehensive Kubernetes cost monitoring & optimization

[> Install in 5 mins or less.Get started](https://www.kubecost.com/install.html?utm_source=guide&utm_medium=kubernetes-hpa)

* [Cost Allocationfor cloud-based & self-hosted k8s](https://www.kubecost.com/features?utm_source=guide&utm_medium=kubernetes-hpa)
* [Insightsinto efficiency, savings & health](https://www.kubecost.com/features?utm_source=guide&utm_medium=kubernetes-hpa)
* [Free foreverfor a single cluster](https://www.kubecost.com/pricing?utm_source=guide&utm_medium=kubernetes-hpa)

### Create the Kubernetes service

For our next step, we have to create a service. The sample application will listen to the public endpoint by using this service. Create a service configuration file with the following content:

$ cd /Users/bob/hpa/

$ cat service.yaml

apiVersion: v1

kind: Service

metadata:

name: hpa-demo-deployment

labels:

run: hpa-demo-deployment

spec:

ports:

- port: 80

selector:

run: hpa-demo-deployment

This service will be a front-end to the deployment we created above, which we can access via port 80.

Apply the changes:

$ kubectl apply -f service.yaml

service/hpa-demo-deployment created

We have created the service. Next, let’s list the service and see the status:

$ kubectl get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

hpa-demo-deployment ClusterIP 10.100.124.139 80/TCP 7s

kubernetes ClusterIP 10.100.0.1 443/TCP 172m

Here, we can see:

* hpa-demo-deployment = Service Name
* 10.100.124.139 = IP address of the service, and it is open on Port 80/TCP

### Install the Horizontal Pod Autoscaler

We now have the sample application as part of our deployment, and the service is accessible on port 80. To scale our resources, we will use HPA to scale up when traffic increases and scale down the resources when traffic decreases.

Let’s create the HPA configuration file as shown below:

$ cd /Users/bob/hpa/

$ cat hpa.yaml

apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

metadata:

name: hpa-demo-deployment

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: hpa-demo-deployment

minReplicas: 1

maxReplicas: 10

targetCPUUtilizationPercentage: 50

Apply the changes:

$ kubectl apply -f hpa.yaml

horizontalpodautoscaler.autoscaling/hpa-demo-deployment created

Verify the HPA deployment:

$ kubectl get hpa

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

hpa-demo-deployment Deployment/hpa-demo-deployment 0%/50% 1 10 0 8s

The above output shows that the HPA maintains between 1 and 10 replicas of the pods controlled by the hpa-demo-deployment. In the example shown above (see the column titled “TARGETS”), the target of 50% is the average CPU utilization that the HPA needs to maintain, whereas the target of 0% is the current usage.

If we want to change the MIN and MAX values, we can use this command:

📝Note: Since we already have the same MIN/MAX values, the output throws an error that says it already exists.

### Increase the load

So far, we have set up our EKS cluster, installed the Metrics Server, deployed a sample application, and created an associated Kubernetes service for the application. We also deployed HPA, which will monitor and adjust our resources.

To test HPA in real-time, let’s increase the load on the cluster and check how HPA responds in managing the resources.

First, let’s check the current status of the deployment:

$ kubectl get deploy

NAME READY UP-TO-DATE AVAILABLE AGE

hpa-demo-deployment 1/1 1 1 23s

Next, we will start a container and send an infinite loop of queries to the ‘php-apache’ service, listening on port 8080. Open a new terminal and execute the below command:

# kubectl run -i --tty load-generator --rm --image=busybox --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://hpa-demo-deployment; done"

📝Note: If you do not have DNS entries for the service, use the service name.

To view the service name:

$ kubectl get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

hpa-demo-deployment ClusterIP 10.100.95.188 80/TCP 10m

Before we increase the load, the HPA status will look like this:

$ kubectl get hpa

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

hpa-demo-deployment Deployment/hpa-demo-deployment 0%/50% 1 10 1 12m

Once you triggered the load test, use the below command, which will show the status of the HPA every 30 seconds:

$ kubectl get hpa -w

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

hpa-demo-deployment Deployment/hpa-demo-deployment 0%/50% 1 10 1 15m

...

...

hpa-demo-deployment Deployment/hpa-demo-deployment 38%/50% 1 10 8 25m

Here, you can see that as our usage went up, the number of pods scaled from 1 to 7:

$ kubectl get deployment php-apache

NAME READY UP-TO-DATE AVAILABLE AGE

hpa-demo-deployment 7/7 7 7 21m

You can also see pod usage metrics. The load-generator pod generates the load for this example:

$ kubectl top pods --all-namespaces

NAMESPACE NAME CPU(cores) MEMORY(bytes)

default hpa-demo-deployment-6b988776b4-b2hkb 1m 10Mi

default load-generator 10m 1Mi

default hpa-demo-deployment-d4cf67d68-2x89h 97m 12Mi

default hpa-demo-deployment-d4cf67d68-5qxgm 86m 12Mi

default hpa-demo-deployment-d4cf67d68-ddm54 131m 12Mi

default hpa-demo-deployment-d4cf67d68-g6hhw 72m 12Mi

default hpa-demo-deployment-d4cf67d68-pg67w 123m 12Mi

default hpa-demo-deployment-d4cf67d68-rjp77 75m 12Mi

default hpa-demo-deployment-d4cf67d68-vnd8k 102m 12Mi

kube-system aws-node-982kv 4m 41Mi

kube-system aws-node-rqbg9 4m 40Mi

kube-system coredns-86d9946576-9k6gx 4m 9Mi

kube-system coredns-86d9946576-m67h6 4m 9Mi

kube-system kube-proxy-lcklc 1m 11Mi

kube-system kube-proxy-tk96q 1m 11Mi

kube-system metrics-server-9f459d97b-q5989 4m 17Mi

### Monitor HPA events

If you want to see what steps HPA is performing while scaling, use this command and check for the events section:

$ kubectl describe deploy hpa-demo-deployment

Name: hpa-demo-deployment

Namespace: default

CreationTimestamp: Mon, 30 Aug 2021 17:15:34 +0530

Labels:

Annotations: deployment.kubernetes.io/revision: 1

Selector: run=php-apache

Replicas: 7 desired | 7 updated | 7 total | 7 available | 0 NewReplicaSet: hpa-demo-deployment-d4cf67d68 (7/7 replicas created)

...

...

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal ScalingReplicaSet 12m deployment-controller Scaled up replica set hpa-demo-deployment-d4cf67d68 to 1

Normal ScalingReplicaSet 5m39s deployment-controller Scaled up replica set hpa-demo-deployment-d4cf67d68 to 4

Normal ScalingReplicaSet 5m24s deployment-controller Scaled up replica set hpa-demo-deployment-d4cf67d68 to 5

Normal ScalingReplicaSet 4m38s deployment-controller Scaled up replica set hpa-demo-deployment-d4cf67d68 to 7

We can see that the pods were scaled up from 1 to 4, then to 5, and finally to 7.

**Decrease the load**

Next, we’ll decrease the load. Navigate to the terminal where you executed the load test and stop the load generation by entering + C.

Then, verify the status of your resource usage:

$ kubectl get hpa

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

hpa-demo-deployment Deployment/hpa-demo-deployment 0%/50% 1 10 1 25m

$ kubectl get deployment hpa-demo-deployment

NAME READY UP-TO-DATE AVAILABLE AGE

hpa-demo-deployment 1/1 1 1 25m

Another way to verify the status is:

$ kubectl get events

51m Normal SuccessfulCreate replicaset/hpa-demo-deployment-cf6477c46 Created pod: hpa-demo-deployment-cf6477c46-b56vr

52m Normal SuccessfulRescale horizontalpodautoscaler/hpa-demo-deployment New size: 4; reason: cpu resource utilization (percentage of request) above target

52m Normal ScalingReplicaSet deployment/hpa-demo-deployment Scaled up replica set hpa-demo-deployment-cf6477c46 to 4

52m Normal SuccessfulRescale horizontalpodautoscaler/hpa-demo-deployment New size: 6; reason: cpu resource utilization (percentage of request) above target

52m Normal ScalingReplicaSet deployment/hpa-demo-deployment Scaled up replica set hpa-demo-deployment-cf6477c46 to 6

51m Normal SuccessfulRescale horizontalpodautoscaler/hpa-demo-deployment New size: 7; reason: cpu resource utilization (percentage of request) above target

51m Normal ScalingReplicaSet deployment/hpa-demo-deployment Scaled up replica set hpa-demo-deployment-cf6477c46 to 7

53m Normal Scheduled pod/load-generator Successfully assigned default/load-generator to ip-192-168-74-193.us-west-2.compute.internal

53m Normal Pulling pod/load-generator Pulling image "busybox"

52m Normal Pulled pod/load-generator Successfully pulled image "busybox" in 1.223993555s

52m Normal Created pod/load-generator Created container load-generator

52m Normal Started pod/load-generator Started container load-generator

### Destroy the cluster

Finally, we’ll destroy the demo EKS cluster with this command:

$ eksctl delete cluster --name my-hpa-demo-cluster --region us-west-2

2021-08-30 20:10:09 [ℹ] eksctl version 0.60.0

2021-08-30 20:10:09 [ℹ] using region us-west-2

2021-08-30 20:10:09 [ℹ] deleting EKS cluster "my-hpa-demo-cluster"

-08-30 20:12:40 [ℹ] waiting for CloudFormation stack "eksctl-my-hpa-demo-cluster-nodegroup-ng-1"

2021-08-30 20:12:41 [ℹ] will delete stack "eksctl-my-hpa-demo-cluster-cluster"

2021-08-30 20:12:42 [✔] all cluster resources were deleted

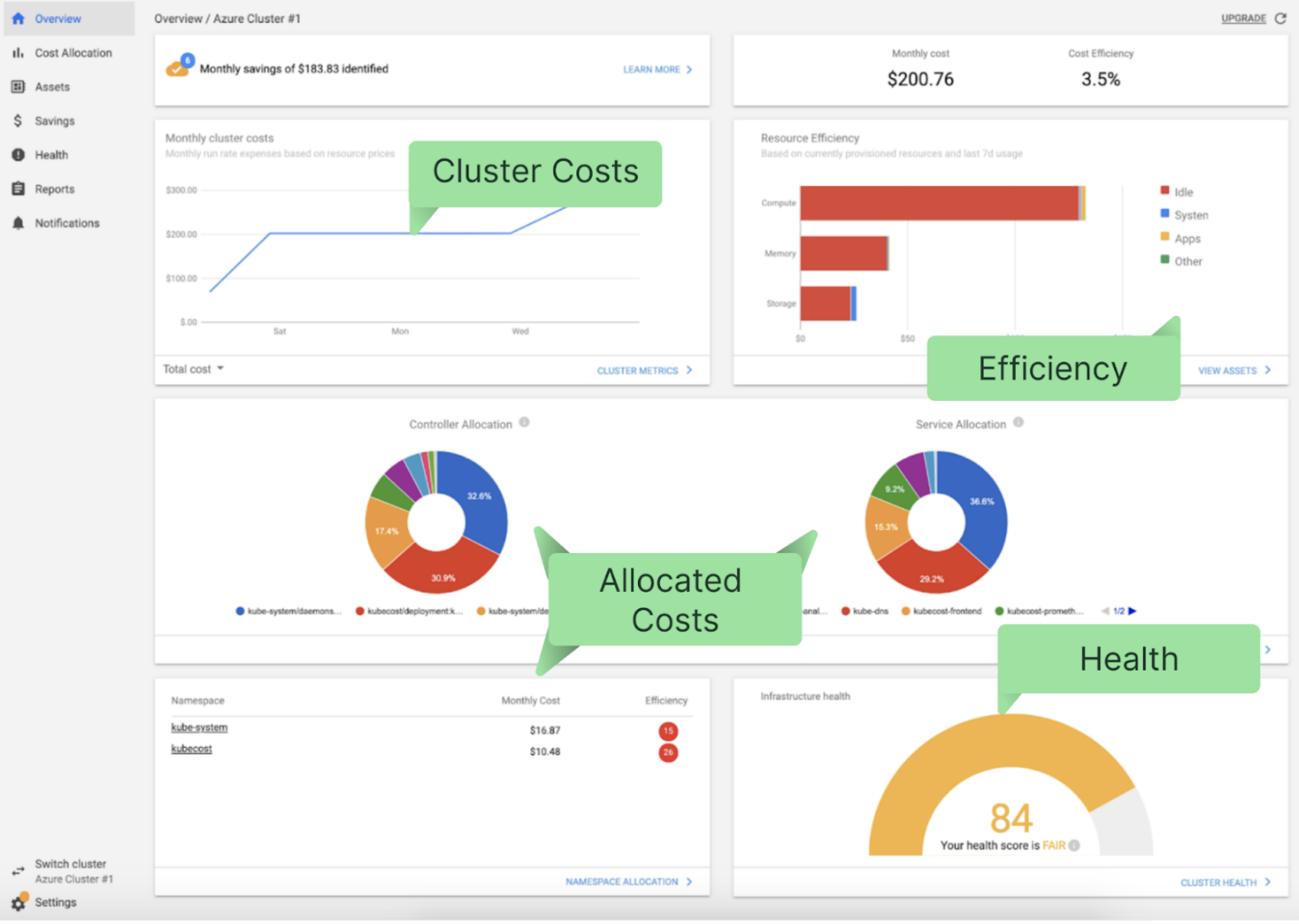
## Usage and cost reporting with HPA

With increased scalability comes increased complexity—horizontal autoscaling complicates usage and cost reporting by introducing variability into the equation. Usage is easier to measure and attribute to each tenant in a Kubernetes cluster when capacity is absolutely static and the cluster has no more than a couple of tenants. However, as you add more tenants and constantly adjust the capacity (as HPA allows us to do), specialized tooling becomes a must-have for allocating costs to tenants.

The open-source Kubecost tool solves this problem by measuring detailed usage by Kubernetes concept, and correlating granular usage data with billing information from your cloud provider or cost estimates from your on-prem environment.

It’s easier to measure usage and calculate costs in a static Kubernetes cluster. It’s much harder to do it when the resources allocated to pods routinely change.

The main Kubecost dashboard shown below summarizes cluster costs, efficiency, and the health of your clusters:



A single Helm command installs Kubecost. You can start [here](https://www.kubecost.com/install.html) and try it out for free.