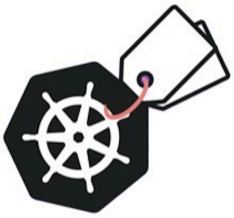
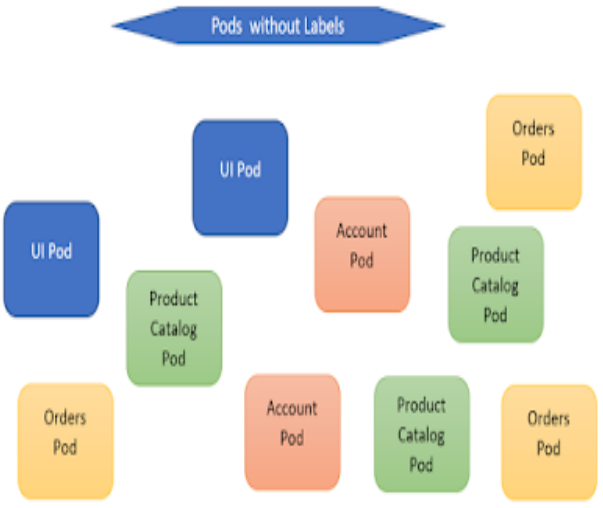
KUBERNETES ESSENTIALS

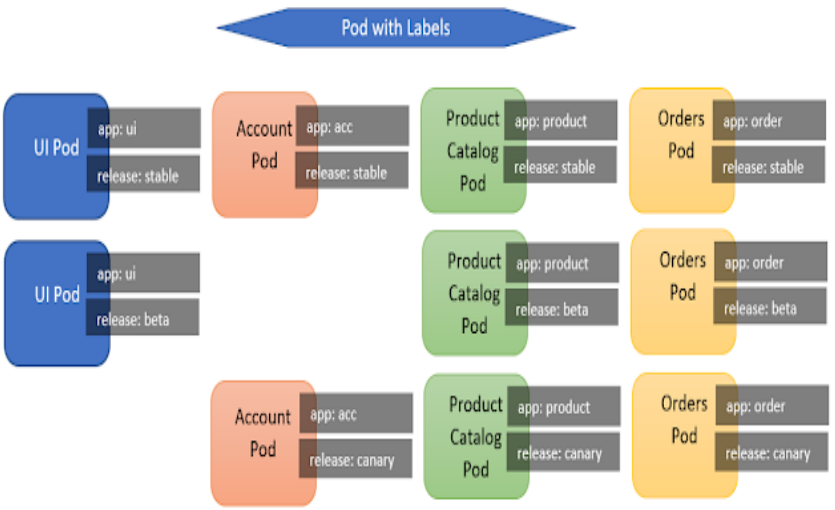


LABELS are key/value pairs that are attached to objects such as Pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object.

K8s selectors allow users to filter objects based on labels, and can be used by the Kubernetes platform as well. From a user perspective, K8s labels and selectors can manipulate objects such as:

- Pods
- Nodes
- Services
- Secrets
- Ingress Resources
- Deployments
- Namespaces





Lets create a pod with a label of key app and its value is alpha.

kubectl run nghw-alpha1 --image=raghuramanbtech/nghw --port=80 --labels="app=alpha"

```
raghuraman@Raghus-Laptop ~ % kubectl run nghw-alpha1  --image=raghuramanbtech/nghw --port=80 --labels="app=alpha"
pod/nghw-alpha1 created
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME                          READY   STATUS    RESTARTS   AGE     LABELS
nghw-alpha1                   1/1     Running   0          14s     app=alpha
nghw-pod-3-7c7c5d7cd9-2l94m   1/1     Running   0          178m    app=nghw-pod-3,pod-template-hash=7c7c5d7cd9
nghw-pod-3-7c7c5d7cd9-68w8h   1/1     Running   0          178m    app=nghw-pod-3,pod-template-hash=7c7c5d7cd9
nghw-pod-4                    1/1     Running   0          3h      run=nghw-pod-4
rc1-cr6r8                     1/1     Running   0          125m    app=kubia
rc1-lcqr6                     1/1     Running   0          128m    app=kubia
rc3-bl7v5                     1/1     Running   0          96m     app=changed,env=db
raghuraman@Raghus-Laptop ~ %
```

We can any number of labels for a pod. To add a label to a existing pod ,

kubectl label pod nghw-alpha1 db=mysql

```
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME                          READY   STATUS    RESTARTS   AGE     LABELS
nghw-alpha1                   1/1     Running   0          60m     app=alpha,db=mysql
nghw-pod-3-7c7c5d7cd9-2l94m   1/1     Running   0          3h58m   app=nghw-pod-3,pod-template-hash=7c7c5d7cd9
nghw-pod-3-7c7c5d7cd9-68w8h   1/1     Running   0          3h58m   app=nghw-pod-3,pod-template-hash=7c7c5d7cd9
nghw-pod-4                    1/1     Running   0          4h1m    run=nghw-pod-4
rc1-cr6r8                     1/1     Running   0          3h5m    app=kubia
rc1-lcqr6                     1/1     Running   0          3h8m    app=kubia
rc3-bl7v5                     1/1     Running   0          156m    app=changed,env=db
raghuraman@Raghus-Laptop ~ %
```

Now , lets edit a existing label value .

kubectl label pod nghw-alpha1 db=mangodb --overwrite

```
raghuraman@Raghus-Laptop ~ % kubectl label pod nghw-alpha1 db=mangodb --overwrite
pod/nghw-alpha1 labeled
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME                          READY   STATUS    RESTARTS   AGE     LABELS
nghw-alpha1                   1/1     Running   0          64m     app=alpha,db=mangodb
nghw-pod-3-7c7c5d7cd9-2l94m   1/1     Running   0          4h3m    app=nghw-pod-3,pod-template-hash=7c7c5d7cd9
nghw-pod-3-7c7c5d7cd9-68w8h   1/1     Running   0          4h3m    app=nghw-pod-3,pod-template-hash=7c7c5d7cd9
nghw-pod-4                    1/1     Running   0          4h5m    run=nghw-pod-4
rc1-cr6r8                     1/1     Running   0          3h10m   app=kubia
rc1-lcqr6                     1/1     Running   0          3h13m   app=kubia
rc3-bl7v5                     1/1     Running   0          160m    app=changed,env=db
raghuraman@Raghus-Laptop ~ %
```

Now , lets select pods based on label.

kubectl get pod -l db=mangodb

```
raghuraman@Raghus-Laptop ~ %
raghuraman@Raghus-Laptop ~ % kubectl get pod -l db=mangodb
NAME          READY   STATUS    RESTARTS   AGE
nghw-alpha1   1/1     Running   0          71m
raghuraman@Raghus-Laptop ~ %
```

Scheduling pods to specific nodes :

Now imagine you want to deploy a new pod that needs a GPU to perform its work. To ask the scheduler to only choose among the nodes that provide a GPU, you'll add a node selector to the pod's YAML. Create a file called nghw-mango.yaml with the following listing's contents and then use the below command to create the pod.

nano nghw-gpu.yaml (if you're using mac)

```
apiVersion: v1
kind: Pod
metadata:
  name: alpha-mango
spec:
  nodeSelector:
   gpu : "true"
  containers:
  - image: raghuramanbtech/nghw
    name: containername
```

kubectl create -f nghw-gpu.yaml

When you create the pod, the scheduler will only choose among the nodes that contain the gpu=true label (which is only a single node in your case).

NAMESPACE :

Namespaces enable you to separate resources that don't belong together into non- overlapping groups. If several users or groups of users are using the same Kubernetes cluster, and they each manage their own distinct set of resources, they should each use their own namespace. This way, they don't need to take any special care not to inadvertently modify or delete the other users' resources and don't need to concern them- selves with name conflicts, because namespaces provide a scope for resource names, as has already been mentioned.

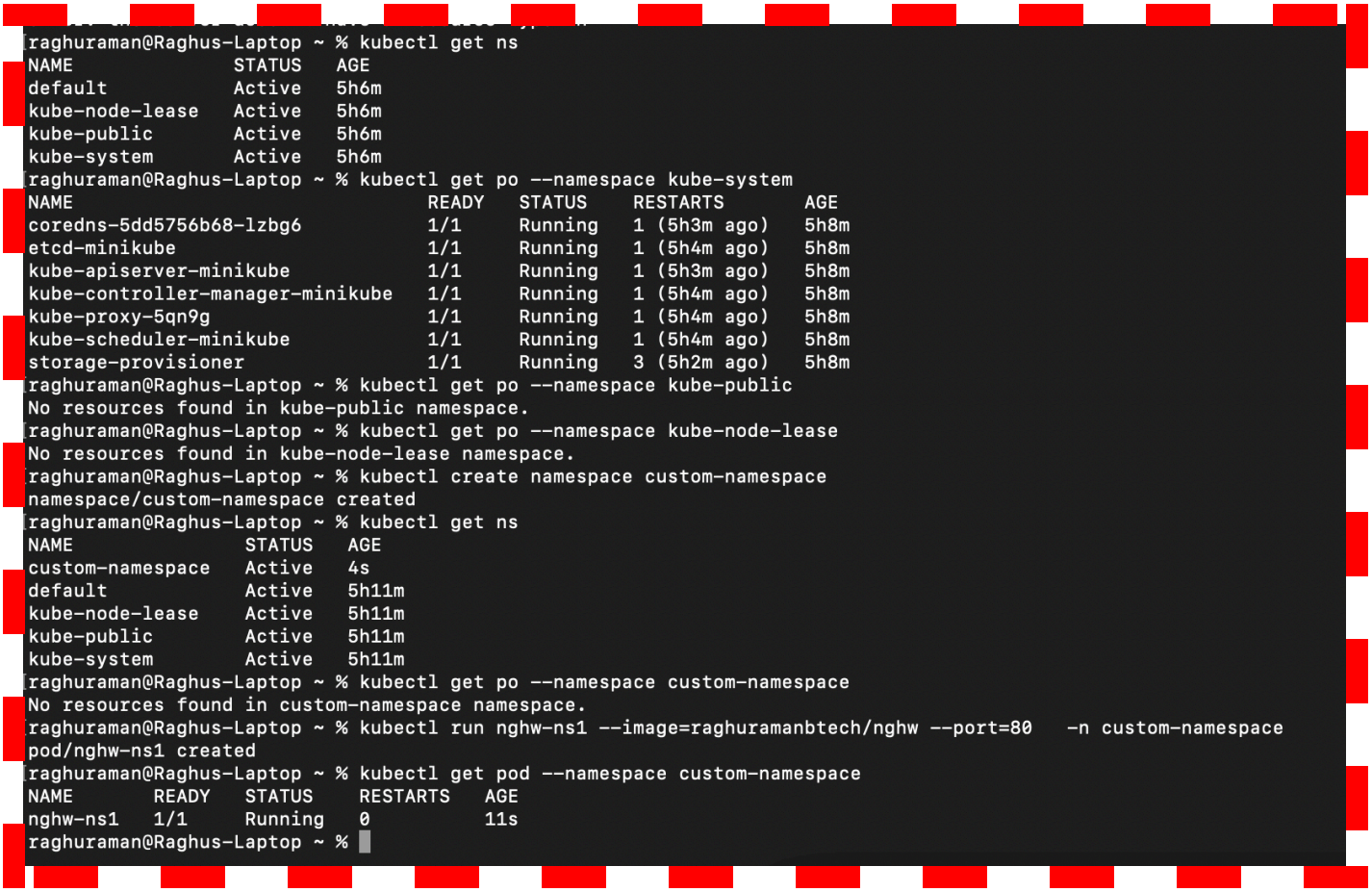
Besides isolating resources, namespaces are also used for allowing only certain users access to particular resources and even for limiting the amount of computational resources available to individual users

Lets create a custom namespace :

kubectl create namespace custom-namespace

Lets create a pod in this custom-namespace :

kubectl run nghw-ns1 --image=raghuramanbtech/nghw --port=80 -n custom-namespace



Understanding the isolation provided by namespaces :

Although namespaces allow you to isolate objects into distinct groups, which allows you to operate only on those belonging to the specified namespace, they don't provide any kind of isolation of running objects.

For example, you may think that when different users deploy pods across different namespaces, those pods are isolated from each other and can't communicate, but that's not necessarily the case. Whether namespaces provide network isolation depends on which networking solution is deployed with Kubernetes. When the solution doesn't provide inter-namespace network isolation, if a pod in namespace foo knows the IP address of a pod in namespace bar, there is nothing preventing it from sending traffic,such as HTTP requests, to the other pod.

DELETING PODS :

We can delete a pod with a pod name directly,

kubectl delete pod rc1-cr6r8

We can bulk delete pods with label,

kubectl delete pod -l app=web

We can delete a namespace, so that all pods under the namespace get deleted .

kubectl delete ns custom-namespace

We can delete all pods in current namespace, while keeping the namespace by,

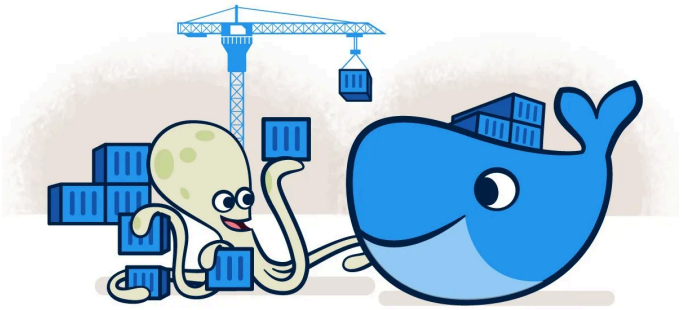
kubectl delete po --all

If the pods are created by replica controller/sets , even if the pod is deleted , replica controller will create a new pod with specified image, since its job is to make sure 'n' instances of pods are always running .

You can delete the ReplicationController and the pods, as well as all the Services you've created, by deleting all resources in the current namespace with a single command:

kubectl delete all --all

The first all in the command specifies that you're deleting resources of all types, and the --all option specifies that you're deleting all resource instances.



REPLICA CONTROLLER

A ReplicationController ensures that a specified number of pod replicas are running at any one time. In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

If there are too many pods, the ReplicationController terminates the extra pods. If there are too few, the ReplicationController starts more pods. Unlike manually created pods, 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.

Lets Create a replica controller with 4 replicas .

nano rc01.yaml (if youre using mac)

~ — nano rc01.yaml ~ — redis-cli

UW PICO 5.09

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: rc3
spec:
  replicas: 4
  selector:
    app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: kubia
        image: raghuramanbtech/nghw
        ports:
        - containerPort: 8080
```

From the yaml file , we are creating a replica controller with replicas 4 . The selector key checks for label with key “app” and value “web”. Therefore , the pods with label app=web will be maintained in count 4.

Template key describes how to create a new pod.

To create the ReplicationController, use the kubectl create command, which you already know:

kubectl create -f kubia-rc.yaml

```
~ — -zsh                                                    ~ — redi
raghuraman@Raghus-Laptop ~ % nano rc01.yaml
raghuraman@Raghus-Laptop ~ % kubectl get pods
No resources found in default namespace.
raghuraman@Raghus-Laptop ~ % kubectl create -f rco1.yaml
error: the path "rco1.yaml" does not exist
raghuraman@Raghus-Laptop ~ % kubectl create -f rc01.yaml
replicationcontroller/rc3 created
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS              RESTARTS   AGE   LABELS
rc3-27n7q   1/1     Running             0          7s    app=web
rc3-4fqg7   1/1     Running             0          7s    app=web
rc3-87ffk   0/1     ContainerCreating   0          7s    app=web
rc3-d5cpw   0/1     ContainerCreating   0          7s    app=web
raghuraman@Raghus-Laptop ~ %
```

What will happen if the pod gets terminated / deleted ?

Replica controller will create a new pod with the template specified on configuring rc01.yaml . let's try deleting a pod with its name .

```
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS              RESTARTS   AGE   LABELS
rc3-27n7q   1/1     Running             0          7s    app=web
rc3-4fqg7   1/1     Running             0          7s    app=web
rc3-87ffk   0/1     ContainerCreating   0          7s    app=web
rc3-d5cpw   0/1     ContainerCreating   0          7s    app=web
raghuraman@Raghus-Laptop ~ % kubectl delete pod rc3-27n7q
pod "rc3-27n7q" deleted
^[[A^[[A%
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS              RESTARTS   AGE     LABELS
rc3-4fqg7   1/1     Running             0          5m31s   app=web
rc3-87ffk   1/1     Running             0          5m31s   app=web
rc3-d5cpw   1/1     Running             0          5m31s   app=web
rc3-vgp2h   0/1     ContainerCreating   0          1s      app=web
raghuraman@Raghus-Laptop ~ %
```

What will happen if we update the label name ?

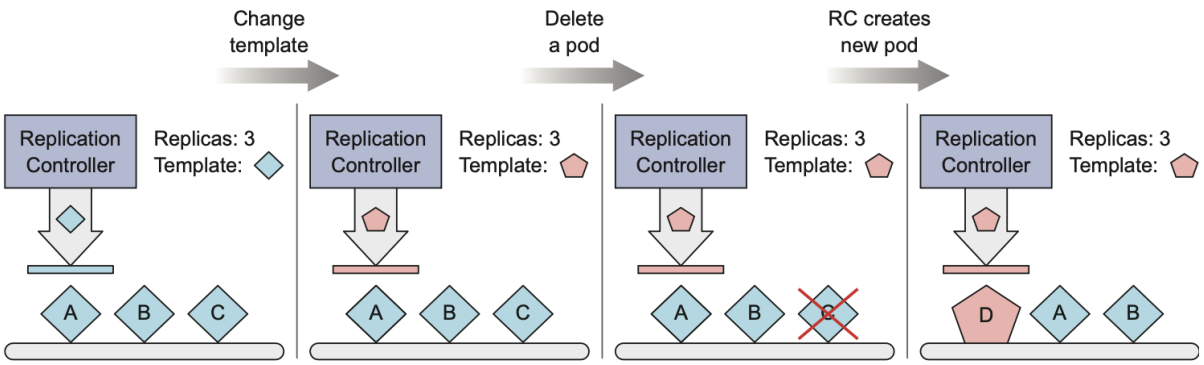
The updated pod will be moved out of the scope of rc , a new pod gets created. The updated label pod just moves out of scope and not deleted, remains unmanaged by rc.

```
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS              RESTARTS   AGE     LABELS
rc3-4fqg7   1/1     Running             0          5m31s   app=web
rc3-87ffk   1/1     Running             0          5m31s   app=web
rc3-d5cpw   1/1     Running             0          5m31s   app=web
rc3-vgp2h   0/1     ContainerCreating   0          1s      app=web
raghuraman@Raghus-Laptop ~ % kubectl label pod rc3-87ffk app=nodejs --overwrite
pod/rc3-87ffk labeled
raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS              RESTARTS   AGE   LABELS
rc3-4fqg7   1/1     Running             0          15m   app=web
rc3-87ffk   1/1     Running             0          15m   app=nodejs
rc3-d5cpw   1/1     Running             0          15m   app=web
rc3-vgp2h   1/1     Running             0          10m   app=web
rc3-vtm86   0/1     ContainerCreating   0          1s    app=web
raghuraman@Raghus-Laptop ~ % †
```

Note : If we add another label , it should not affect the rc , unless the selector (in our case label:app) is modified.

Changing the pod template :

A ReplicationController's pod template can be modified at any time. Changing the pod template is like replacing a cookie cutter with another one. It will only affect the cookies you cut out afterward and will have no effect on the ones you've already cut (see figure 4.6). To modify the old pods, you'd need to delete them and let the Replication- Controller replace them with new ones based on the new template.



Deleting a rc will not delete the managed pods , unless we use cascade = true.

kubectl delete rc rc3 --cascade=false

```
pod/rc3-87ffk labeled
[raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS              RESTARTS   AGE   LABELS
rc3-4fqg7   1/1     Running             0          15m   app=web
rc3-87ffk   1/1     Running             0          15m   app=nodejs
rc3-d5cpw   1/1     Running             0          15m   app=web
rc3-vgp2h   1/1     Running             0          10m   app=web
rc3-vtm86   0/1     ContainerCreating   0          1s    app=web
[raghuraman@Raghus-Laptop ~ % kubectl get rc
NAME   DESIRED   CURRENT   READY   AGE
rc3    4         4         4       58m
[raghuraman@Raghus-Laptop ~ % kubectl delete rc rc3 --cascade=false
warning: --cascade=false is deprecated (boolean value) and can be replaced with --cascade=orphan.
replicationcontroller "rc3" deleted
[raghuraman@Raghus-Laptop ~ % kubectl get pods --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
rc3-4fqg7   1/1     Running   0          59m   app=web
rc3-87ffk   1/1     Running   0          59m   app=nodejs
rc3-d5cpw   1/1     Running   0          59m   app=web
rc3-vgp2h   1/1     Running   0          53m   app=web
rc3-vtm86   1/1     Running   0          43m   app=web
raghuraman@Raghus-Laptop ~ %
```

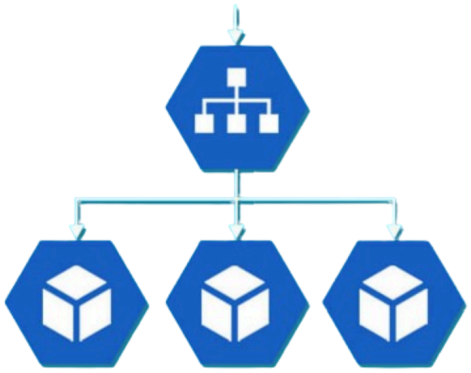
REPLICA SET :

A ReplicaSet behaves exactly like a ReplicationController, but it has more expressive pod selectors. Whereas a ReplicationController's label selector only allows matching pods that include a certain label, a ReplicaSet's selector also allows matching pods that lack a certain label or pods that include a certain label key, regardless of its value.

Also, for example, a single ReplicationController can't match pods with the label env=production and those with the label env=devel at the same time. It can only match either pods with the env=production label or pods with the env=devel label. But a sin- gle ReplicaSet can match both sets of pods and treat them as a single group.

Similarly, a ReplicationController can't match pods based merely on the presence of a label key, regardless of its value, whereas a ReplicaSet can. For example, a Replica- Set can match all pods that include a label with the key env, whatever its actual value is (you can think of it as env=*).

We will see more about replica sets later.



SERVICES

In a kubernetes cluster , each pod gets its own ip address .But, pods in k8 are ephemeral, meaning that they come and go very frequently. When a pod restarts , or deleted or replaced by another new pod , it get its own new ip address . so , it doesnt make sense to use pod ip address directly . But we have a solution for a static/stable ip address . Basically in front of each pods , we set a service which representa a persistent stable ip address .

There are four types of services :

1. CLUSTER IP :

This is the default type . when you didn't specify a type , service of type cluster ip is created. It provides internal connectivity between different components of our application. **Kubernetes assigns a virtual IP address to a ClusterIP service that can solely be accessed from within the cluster during its creation**. This IP address is stable and doesn't change even if the pods behind the service are rescheduled or replaced.ClusterIP services are an excellent choice for internal communication between different components of our application that don't need to be exposed to the outside world.

Lets create a service with service1.yaml :

```
apiVersion: v1
kind: Service
metadata:
  name: webappservice1
spec:
  selector:
    app: web
  ports:
  - name: http
    port: 80
    targetPort: 80
```

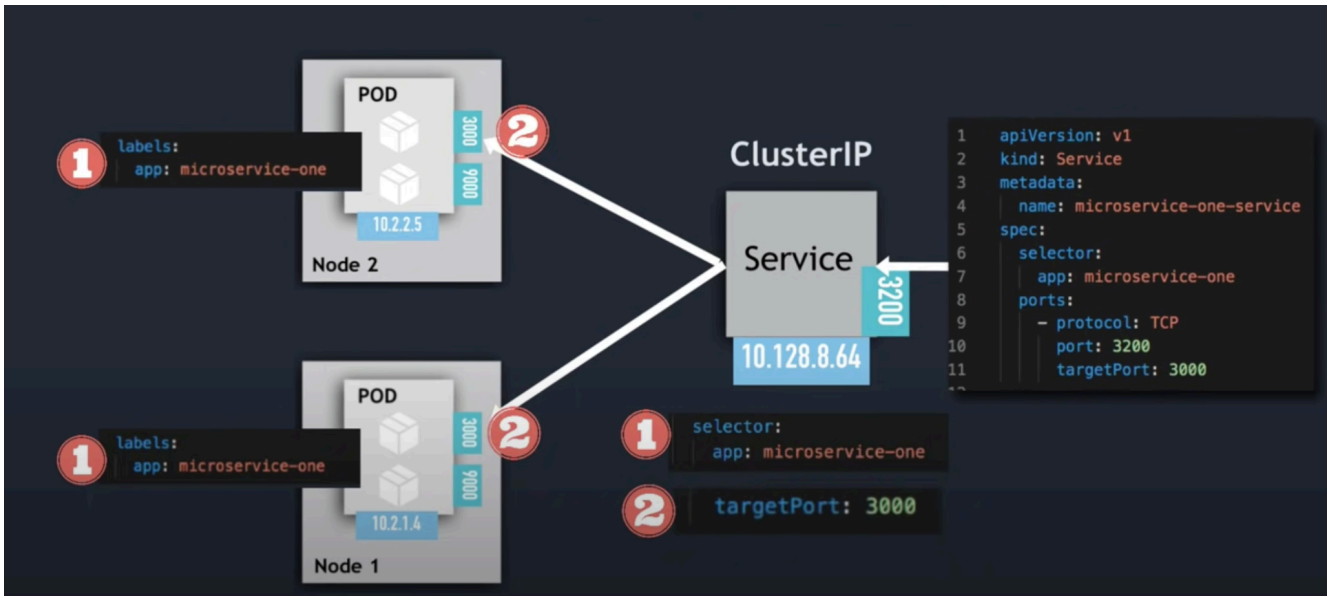
Let's create the service with kubectl create -f service1.yaml command . Now , lets breakdown this yaml .

How does a service knows which pod to communicate with ?

Thats why we have “selector” which specifies the labels to include in this service.

How does a service knows which port of a pod to communicate with ?

That's why we have a target port . If a pod has multiple applications listening inside a container in different ports , each will be listening at a unique port and that should be our target port .



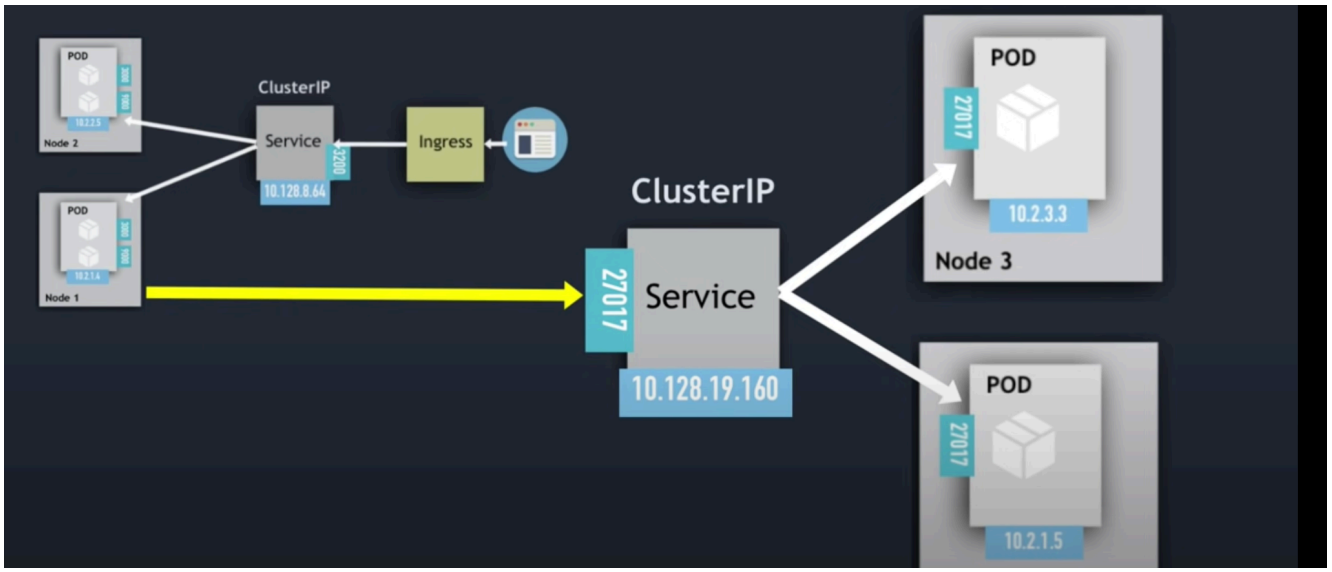
To keep track of members(pods) of a service, K8 will maintal these infos in name of endpoints , which has the same name of service name . The ip address of members are shown below.

Kubectl get endpoints

```
raghuraman@Raghus-Laptop ~ % kubectl get endpoints
NAME             ENDPOINTS                                                  AGE
kubernetes       192.168.49.2:8443                                          23h
webappservice1   10.244.0.22:80,10.244.0.24:80,10.244.0.27:80 + 1 more...   8m29s
raghuraman@Raghus-Laptop ~ %
```

Try Yourself : Now , try the command **minikube service <serviceName>** , which will automatically opens default browser and as per our service1.yaml , we selected app=web, so pods with this label and its specified image should be seen as output (in my case , i have used the same nghw image everywhere, so it should show “from minikube” like shown in “hello minikube” chapter.

Note : A microService application running inside the pod , can talk with another service through service’s port and that service , say serviceB, and that serviceB knows which is the target port as well. How how service works for internal communication .



2 . HeadLess Service:

A Headless Service is a variation of the ClusterIP Service, where the clusterIP field is set to None. Unlike traditional services, Headless Services do not use a single Service IP to proxy

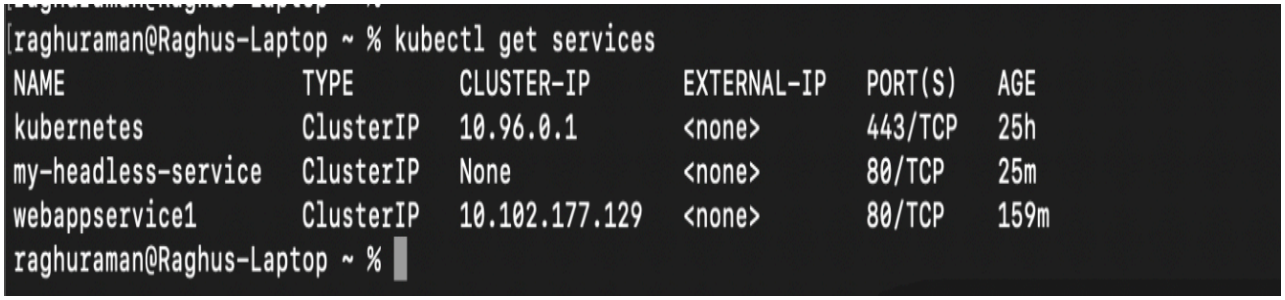
connections to the Pods. Instead, they allow you to directly connect to Pods without any load balancing intermediary.

The primary purpose of Headless Services is to enable service discovery mechanisms that require direct Pod access. This type of service is particularly useful for stateful applications, such as databases or distributed storage systems, where individual Pods need to be directly addressed. Additionally, Headless Services can be employed when you need to implement custom load balancing logic or when you want to leverage a specific load balancing mechanism that operates at the Pod level.

By removing the load balancing layer, Headless Services provide a straightforward way to access individual Pods based on their DNS names or IP addresses.

```
apiVersion: v1
kind: Service
metadata:
  name: my-headless-service
spec:
  clusterIP: None
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
```

When we execute this yaml ,we get service without cluster ip .



3 .NODE PORT :

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

Nodeport is internally implemented on top of ClusterIP . The port in the Kubernetes Service definition is used to specify the port on which the service will listen for traffic within the Kubernetes cluster. This is the port that will be exposed to other pods in the cluster as an endpoint for the service. When a request is made to this port by a client within the cluster, the traffic will be routed to one of the pods selected by the Service based on its load balancing algorithm.

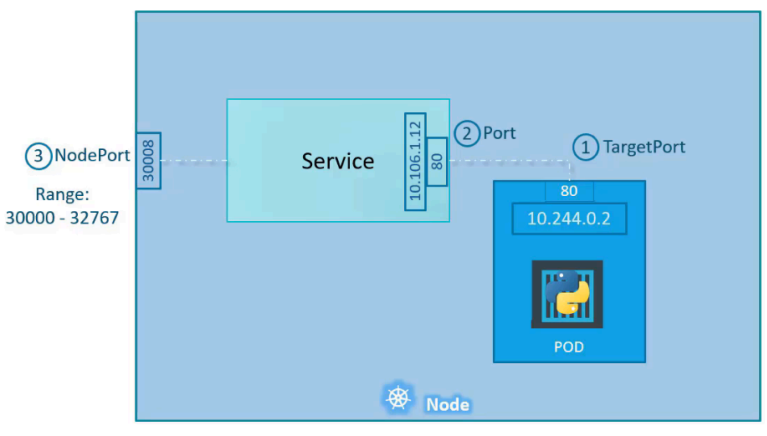
The nodePort is used to expose the service on a port on the node itself, which allows the service to be accessed from outside the cluster.

Nodeportservice.yaml would look like :

```
apiVersion: v1
kind: Service
metadata:
  name: nodeportsrc1
spec:
  type: NodePort
  selector:
    app: web
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30008
```

```
[raghuraman@Raghus-Laptop ~ % kubectl get services
NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes           ClusterIP   10.96.0.1        <none>        443/TCP        25h
my-headless-service  ClusterIP   None             <none>        80/TCP         39m
nodeportsrc1         NodePort    10.101.61.55     <none>        80:30008/TCP   4m53s
webappservice1       ClusterIP   10.102.177.129   <none>        80/TCP         173m
[raghuraman@Raghus-Laptop ~ %  kubectl proxy --port=8088
Starting to serve on 127.0.0.1:8088
```

For more understanding ,



service-definition.yml

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

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
```

4 . LOAD BALANCER :

When creating a Service, you have the option of automatically creating a cloud load balancer. This provides an externally-accessible IP address that sends traffic to the correct port on your cluster nodes, *provided your cluster runs in a supported environment and is configured with the correct cloud load balancer provider package*.

LoadBalancer Service this time builds on top of the NodePort Service, with an added benefit: It adds load balancing functionality to distribute traffic between nodes. This reduces the negative effects of any one node failing or becoming overloaded with requests.

The traffic coming from external clients goes through a path like this: External client -> Loadbalancer -> Worker node IP -> NodePort (as specified in nodePort key below) -> ClusterIP Service -> Pod.

The LoadBalancer Service can only be used when the Kubernetes cluster is provisioned on a cloud platform that supports this Service type.

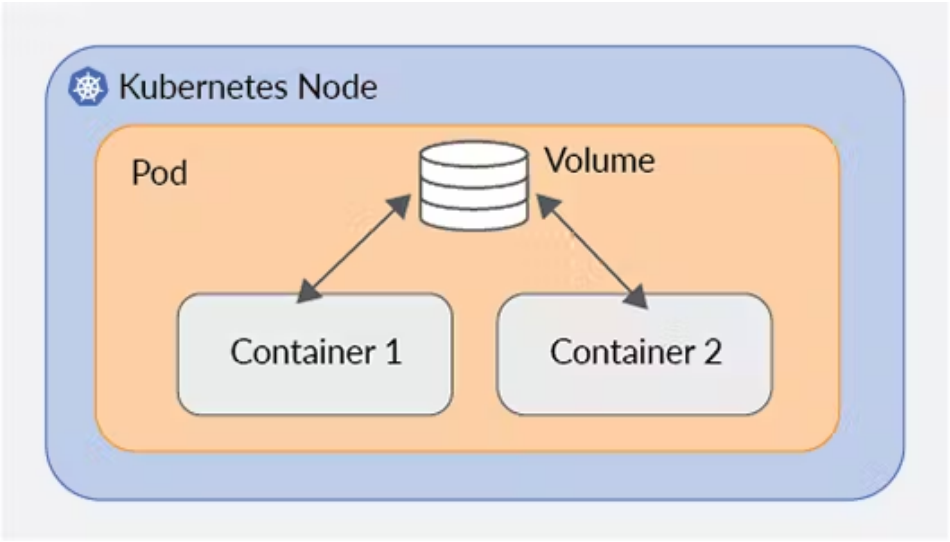
```
apiVersion: v1
kind: Service
metadata:
  name: nodeportsrc1
spec:
  type: LoadBalancer
  selector:
    app: web
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30009
```

```
raghuraman@Raghus-Laptop ~ % kubectl get services
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes           ClusterIP      10.96.0.1        <none>        443/TCP        26h
loadbalancersrc1     LoadBalancer   10.108.229.228   <pending>     80:30009/TCP   20s
my-headless-service  ClusterIP      None             <none>        80/TCP         85m
nodeportsrc1         NodePort       10.101.61.55     <none>        80:30008/TCP   51m
webappservice1       ClusterIP      10.102.177.129   <none>        80/TCP         3h39m
raghuraman@Raghus-Laptop ~ % kubectl get services
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes           ClusterIP      10.96.0.1        <none>        443/TCP        26h
loadbalancersrc1     LoadBalancer   10.108.229.228   <pending>     80:30009/TCP   26s
my-headless-service  ClusterIP      None             <none>        80/TCP         85m
nodeportsrc1         NodePort       10.101.61.55     <none>        80:30008/TCP   51m
webappservice1       ClusterIP      10.102.177.129   <none>        80/TCP         3h39m
raghuraman@Raghus-Laptop ~ % minikube service loadbalancersrc1 --url
http://127.0.0.1:63964
❗  Because you are using a Docker driver on darwin, the terminal needs to be open to run it.
```

VOLUME

A Container's file system lives only as long as the Container does. So when a Container terminates and restarts, filesystem changes are lost. For more consistent storage that is independent of the Container, you can use a Volume. This is especially important for stateful applications.

Containers run in an isolated environment. Let's assume there are two containers inside a pod , one container writes log in /bin/logs and another container reads logs from /bin/logs . In such cases, creating a pod without a sharing disk is useless because containers which read logs cannot access the directory of containers which write logs. That's where volumes come into picture. Linux allows you to mount a filesystem at arbitrary locations in the file tree. When you do that, the contents of the mounted filesystem are accessible in the directory it's mounted into. By mounting the same volume into two containers, they can operate on the same files .



There are many types of volumes. We will see basic types like ,emptyDir , hostPath , gitRepo , Persistent volume and persistent volume claim

EMPTY DIR :

For a Pod that defines an emptyDir volume, the volume is created when the Pod is assigned to a 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.
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

The emptyDir.medium field controls where emptyDir volumes are stored. By default emptyDir volumes are stored on whatever medium that backs the node such as disk, SSD, or network storage, depending on your environment. 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, files you write count against the memory limit of the container that wrote them.

A size limit can be specified for the default medium, which limits the capacity of the emptyDir volume. The storage is allocated from node ephemeral storage. If that is filled up from another source (for example, log files or image overlays), the emptyDir may run out of capacity before this limit.

We can have a empty dir volume like :

```
apiVersion: v1
kind: Pod
metadata:
  name: volumepod
spec:
- image: <img>
  name: html-generator
  volumeMounts:
  - name: html
    mountPath: /var/htdocs
- image: <img>
  name: web-server
  volumeMounts:
  - name: html
```

```
    mountPath: /usr/share/nginx/html
    readOnly: true
  ports:
  - containerPort: 80
    protocol: TCP
Volumes:
- name: html
  emptyDir: {}
```

From the above yaml file , we can see a single emptyDir volume called html that's mounted in the two containers .

SPECIFYING THE MEDIUM TO USE FOR THE EMPTYDIR :

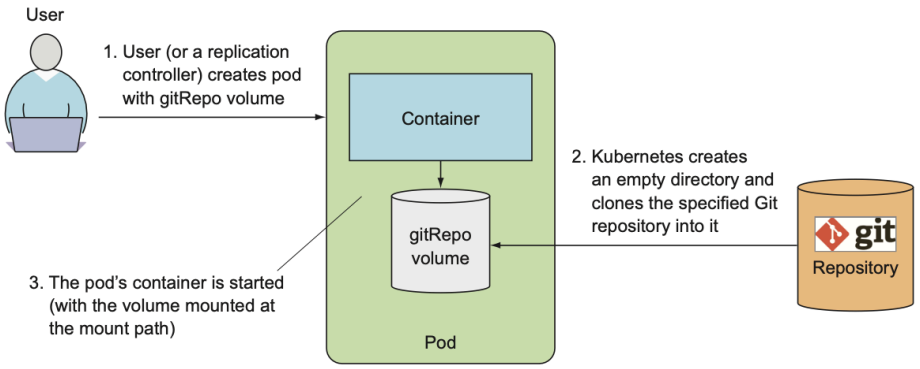
We can tell Kubernetes to create the emptyDir on a tmpfs filesystem (in memory instead of on disk). To do this, set the emptyDir's medium to Memory like this:

```
apiVersion: v1
kind: Pod
metadata:
 name: test-pd
spec:
 containers:
 - image: registry.k8s.io/test-webserver
   name: test-container
   volumeMounts:
   - mountPath: /cache
     name: cache-volume
 volumes:
 - name: cache-volume
   emptyDir:
     sizeLimit: 500Mi
```

GIT REPO :

A gitRepo volume is basically an emptyDir volume that gets populated by cloning a Git repository and checking out a specific revision when the pod is starting up (but before its containers are created) . After the gitRepo volume is created, it isn't kept in sync with the repo it's referencing. The files in the volume will not be updated when you push additional

commits to the Git repository. However, if your pod is managed by a ReplicationController, deleting the pod will result in a new pod being cre- ated and this new pod's volume will then contain the latest commits .

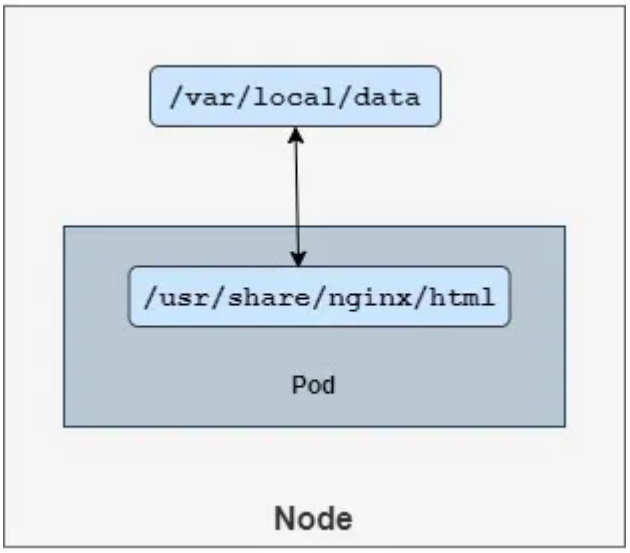



```
apiVersion: v1
kind: Pod
metadata:
  name: gitrepovolume
spec:
  containers:
  - image: nginx:alpine
    name: web-server
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
readOnly: true
  ports:
  - containerPort: 80
    protocol: TCP
volumes:
- name: html
gitRepo:
  repository: https://github.com/raghu-raman-btech/sudo_facebook
  revision: master
  directory: .
```

We'll see about "git sync" in later sesions.

HOST PATH VOLUME :

A hostPath volume points to a specific file or directory on the node's filesystem Pods running on the same node and using the same path in their host- Path volume see the same files.



Both the gitRepo and emptyDir volumes' contents get deleted when a pod is torn down, whereas a hostPath volume's contents don't. If a pod is deleted and the next pod uses a hostPath volume pointing to the same path on the host, the new pod will see whatever was left behind by the previous pod, but only if it's scheduled to the same node as the first pod. volume's contents are stored on a specific node's filesystem, **when the database pod gets rescheduled to another node, it will no longer see the data**. This explains why it's not a good idea to use a hostPath volume for regular pods, because it makes the pod sensitive to what node it's scheduled to.

While creating pod ,we can defining hostpath like below . This manifest mounts /data/foo on the host as /foo inside the single container that runs within the hostpath-example-linux Pod.

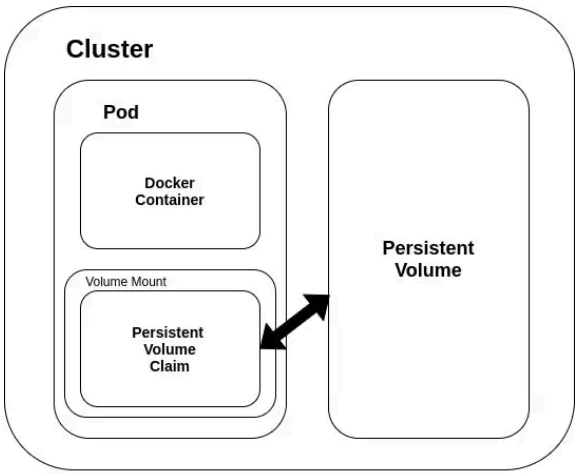
```
containers:
- name: example-container
  image: registry.k8s.io/test-webserver
  volumeMounts:
  - mountPath: /foo
    name: example-volume
```

```
    readOnly: true
  volumes:
  - name: example-volume
    # mount /data/foo, but only if that directory already exists
    hostPath:
      path: /data/foo # directory location on host
```

PERSISTANT VOLUME :

A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV.

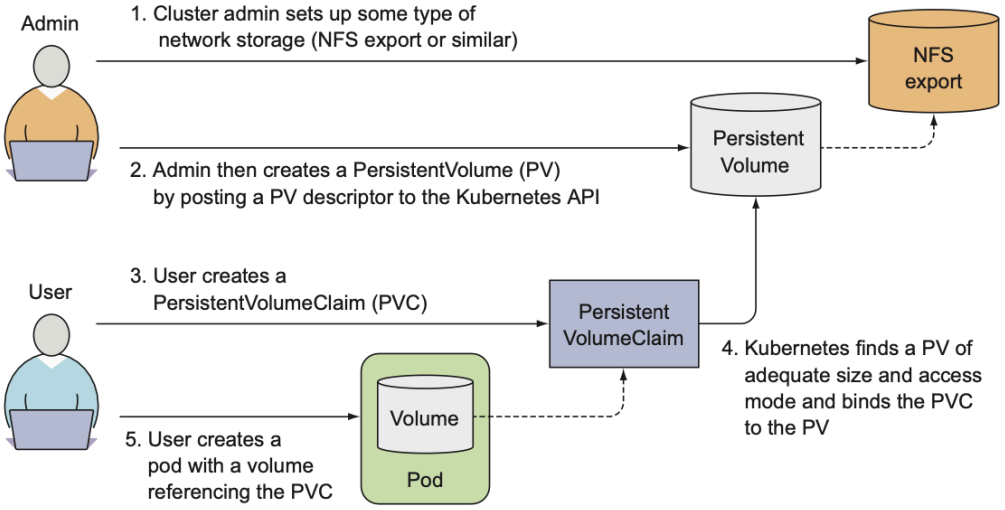
When an application running in a pod needs to persist data to disk and have that same data available even when the pod is rescheduled to another node, you can't use any of the volume types we've mentioned so far. Because this data needs to be accessible from any cluster node, it must be stored on some type of network-attached stor- age (NAS).



This is a 3 step process:

- You or the Kubernetes administrator defines a PersistentVolume (Disk space available for use)
- You define a PersistentVolumeClaim - you claim usage of a part of that PersistentVolume disk space.

- You create a Pod that refers to your PersistentVolumeClaim



Instead of the developer adding a technology-specific volume to their pod, it's the cluster administrator who sets up the underlying storage and then registers it in Kubernetes by creating a PersistentVolume resource through the Kubernetes API server. When creating the PersistentVolume, the admin specifies its size and the access modes it supports.

When a cluster user needs to use persistent storage in one of their pods, they first create a PersistentVolumeClaim manifest, specifying the minimum size and the access mode they require. The user then submits the PersistentVolumeClaim manifest to the Kubernetes API server, and Kubernetes finds the appropriate PersistentVolume and binds the volume to the claim.

The PersistentVolumeClaim can then be used as one of the volumes inside a pod. Other users cannot use the same PersistentVolume until it has been released by delet- ing the bound PersistentVolumeClaim.

Lets create a persistent volume with hostPath :

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-volume01
  labels:
    storage-tier: standard
spec:
  capacity:
    storage: 50Mi
```

```
  accessModes:
    - ReadWriteOnce
- ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  storageClassName: ""
  hostPath:
     path: /var/local/data
```

Capacity defines the storage size . accessModes says it can either be mounted by a single client for reading and writing or by multiple clients for reading only .persistentVolumeReclaimPolicy says After the claim is released , it should be retained , not erased or deleted .

PERSISTENT CLAIM :

A persistentVolumeClaim volume is used to mount a PersistentVolume into a Pod. PersistentVolumeClaims are a way for users to "claim" durable storage (such as an iSCSI volume) without knowing the details of the particular cloud environment.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-persistent-volumeclaim
spec:
    accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Mi
```

Now lets create a pod with claimName as previously created PersistentVolumeClaim.

```
kind: Pod
.
.
 volumes:
    - name: my-persistent-volumeclaim-name
      persistentVolumeClaim:
       claimName: my-persistent-volumeclaim
```

Types of accessModes :

RWO—ReadWriteOnce—Only a single node can mount the volume for reading and writing.

ROX—ReadOnlyMany—Multiple nodes can mount the volume for reading.
RWX—ReadWriteMany—Multiple nodes can mount the volume for both reading and writing.

RWO, ROX, and RWX pertain to the number of worker nodes that can use the volume at the same time, not to the number of pods!

PersistentVolumeReclaimPolicy :

Reclaim policies include : reclaim , recycle , delete . when a pod gets deleted , based on this policy given in pvc spec , claim will either

Stay alive for next pod to get claim/use it (retain) ,the contents in memory wont be erased , it can be accessed even from pods which are outside the namespace,

Volume will still be there without any data ,

volume also gets deleted along with pod.