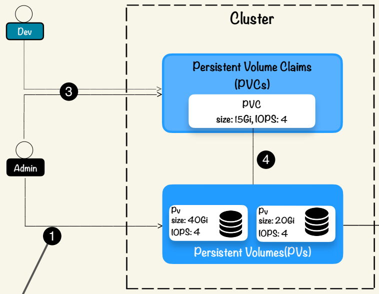
PVs can be provisioned statically or dynamically

Static provisioning involves manually creating PVs and configuring their properties, such as storage capacity, access modes, and claimPolicy.(provisioned by an Administrator)

Dynamic provisioning allows Kubernetes to automatically create PV when a PVC is created. Dynamic provisioning can be implemented using StorageClasses



When you create a PV that uses an NFS volume, the PV connects to the NFS server and uses it as the backend storage for the PV

File storage

0 To create a Persistent Volume (PV) in Kubernetes from an existing NFS volume, you first need to create an NFS export on the NFS server that will make the volume available to the Kubernetes cluster

```
exportfs -o rw,sync,no_subtree_check,no_root_squash,size=40g,fsid=0,
sec=sys,anonuid=65534,anongid=65534 k8s-cluster-ip:/mnt/nfs_share/pv1-40g-rw
```

```
exportfs -o rw,sync,no_subtree_check,no_root_squash,size=20g,fsid=0,
sec=sys,anonuid=65534,anongid=65534 k8s-cluster-ip:/mnt/nfs_share/pv2-20g-rw
```

1 Persistent Volumes (PVs) that fit NFS volumes are created by the cluster administrator on the Kubernetes cluster.

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nfs-pv1-40g-rw
spec:
  capacity:
    storage: 40Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs_server_ip
    path: /mnt/nfs_share/pv1-40g-rw
  persistentVolumeReclaimPolicy: Retain
```

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: nfs-pv1-20g-rw
spec:
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: nfs_server_ip
    path: /mnt/nfs_share/pv2-20g-rw
  persistentVolumeReclaimPolicy: Retain
```

3 To use persistent storage in their Pod, the user can run the kubectl get pv command to view the available PV

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
nfs-pv1-20g-rw	20Gi	RWX	Retain	Available		nfs		1d
nfs-pv1-40g-rw	40Gi	RWX	Retain	Available		nfs		1d

In order to use one of these PVs for persistent storage in a Pod, we can create a Persistent Volume Claim (PVC) that requests storage from the desired PV

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nfs-pvc-20g
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 15Gi
```

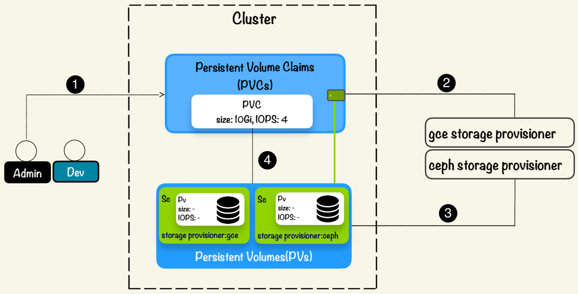
4 Once the PVC is bound to the PV, we can mount the PV to the Pod by including it as a volume in the Pod definition file.

Each PV can be bound to only one PVC at a time , because when a PVC is created, Kubernetes will try to find an available PV that matches the PVC's requirements based on capacity, access mode, and storage class. If a suitable PV is found, the PVC is bound to that PV, and the PV becomes unavailable for other PVCs to

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: my-image
      volumeMounts:
        - name: my-volume
          mountPath: /data
  volumes:
    - name: my-volume
      persistentVolumeClaim:
        claimName: nfs-pvc-20g
```

When the Pod is created, Kubernetes will use the bound PVC named nfs-pvc-20g as a volume mount point to access the persistent storage associated with the bound PV. The volumeMounts section in the Pod specification specifies that the volume should be mounted at the path /data within the container, so any data written to that path will be stored persistently in the PV.

If Pod is deleted and then rebuilt with the same PVC, it will be connected to the same persistent volume, and any data that was previously stored in the volume will still be accessible However if the PVC is deleted, any data stored in the associated persistent volume will be lost and the Pod that was using that PVC will no longer be able to access the data. This is because deleting a PVC deletes the binding between the PVC and the PV, which causes the PV to be released and potentially recycled for use by other PVC



storage classes act as an abstraction layer on top of PVs, allowing you to define a set of default parameters and policies that are used when dynamically provisioning new PVs based on PVC requests

1 When a PVC is created, it specify a StorageClass to use, which will dictate how the PV is provisioned. If no StorageClass is specified, the default StorageClass will be used (if one is defined)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: gce-pd-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gce-pd-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: none
reclaimPolicy: Retain
allowVolumeExpansion: true
volumeBindingMode: Immediate
```

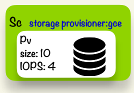
The provisioner field in a StorageClass specifies the name of the provisioner that should be used to provision the storage. There are many different provisioners available for different types of storage, including those for cloud providers like GCE, AWS, and Azure.

The storageClassName field in the PVC specification is used to specify the name of the StorageClass that should be used to provision the requested storage

This accessMode don't allow multiple pods to read and write to the same PVC simultaneously.

2 Kubernetes first tries to find an existing PV that matches the criteria specified in the PVC. If no suitable PV is found , Kubernetes requests the provisioner mapped to the PVC's storage class to create a new volume. The storage provisioner can be a plugin or a driver that interfaces with the underlying storage system

3 The storage provisioner creates a new PV that matches the PVC's requirements, such as size, access mode, and storage class



4 Once the new PV is created, Kubernetes binds it to the PVC and the PVC is ready to be used by a Kubernetes pod. The pod can then mount the volume and use it to store and retrieve data

Finalizers

finalizers are markers attached to resources (such as pods, services, or deployments) to indicate that some additional cleanup or finalization steps need to be performed before the resource can be fully deleted. Finalizers are represented as strings and are stored in the metadata of the resource

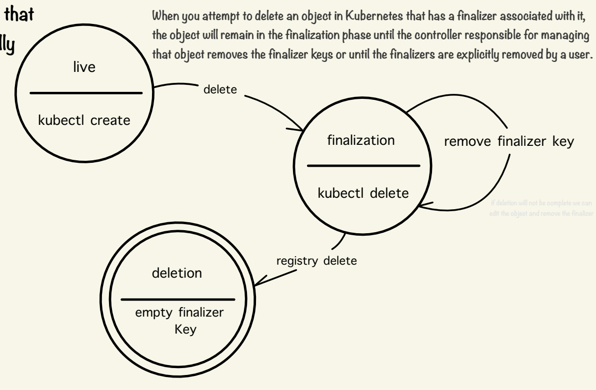
Some common finalizers you've likely encountered are:

kubernetes.io/pv-protection

kubernetes.io/pvc-protection

The finalizers above are used on volumes to prevent accidental deletion

When you attempt to delete an object in Kubernetes that has a finalizer associated with it, the object will remain in the finalization phase until the controller responsible for managing that object removes the finalizer keys or until the finalizers are explicitly removed by a user.

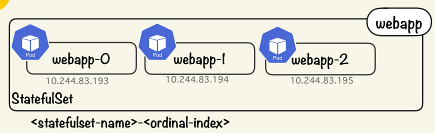


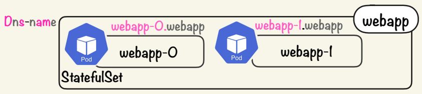
StatefulSet

StatefulSets are a type of workload object in Kubernetes that are used to manage stateful applications. They are designed to handle applications that require unique identities, stable network addresses, persistent storage ,ordered deployment and scaling, and graceful deletion. Such as databases, message queues, etc. StatefulSets maintain a sticky identity for each pod, so even if a pod gets rescheduled, it still maintains the same identity/name. The pods are created from the same spec, but are not interchangeable - each has a unique persistent ID.

Predictable pod name:

In a StatefulSet, each Pod is assigned a predictable name based on the name of the StatefulSet and its index. For example, if the StatefulSet is named "webapp" and has three replicas, the Pods will be named "webapp-0," "webapp-1," and "webapp-2." This allows for easy identification and reference to specific Pods within the Set





Each Pod has a stable hostname based on its ordinal index

Fixed individual DNS name:

StatefulSets also provide a fixed individual DNS name for each Pod, based on the predictable name assigned to it. This allows applications to refer to each Pod by a consistent DNS name, even if the Pod is rescheduled to a different node. For example, if the StatefulSet is named "webapp," and the Pod is named "webapp-0," the DNS name for that Pod will be "webapp-0.webapp

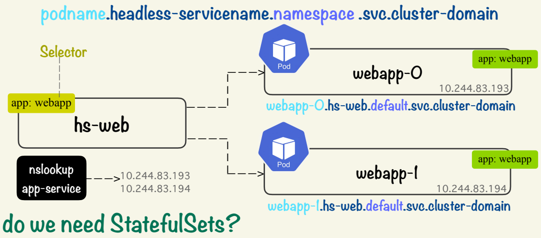
Headless service:

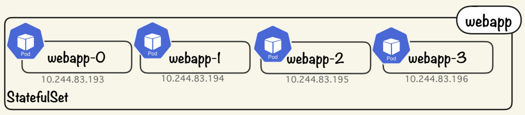
StatefulSets are accompanied by a headless service, which allows for direct communication with individual Pods rather than the Service as a whole. This is useful for stateful applications that require direct communication between Pods, such as database clusters.

Ordered Pod creation:

StatefulSets ensure that Pods are created in a specific order, with each Pod waiting for the previous one to be ready before starting. This is particularly important for stateful applications that require specific sequencing of events, such as database clusters

podname.headless-servicename.namespace .svc.cluster-domain





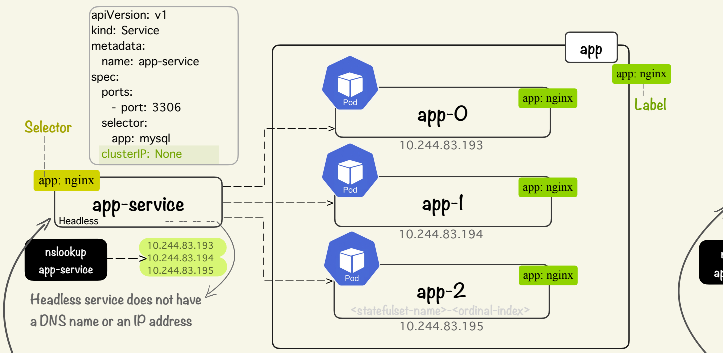
Pods are deployed in order from 0 to N-1, and terminated in reverse order from N-1 to 0.

Why do we need StatefulSets?

Consider an example of a stateful application - a database. Databases are typically stateful, meaning they require persistent storage to store their data. They also require stable network identities to ensure that client applications can consistently connect to the same instance of the database, If you deploy a database using a regular Deployment or RS, Kubernetes will create multiple replicas of the database, each with its own randomly assigned hostname and IP address. This can cause problems for the database, as the client applications may not be able to connect to the correct instance of the database, or data may be lost when pods are deleted or recreated. To solve these problems, you can use a StatefulSet to manage the deployment and scaling of the database.

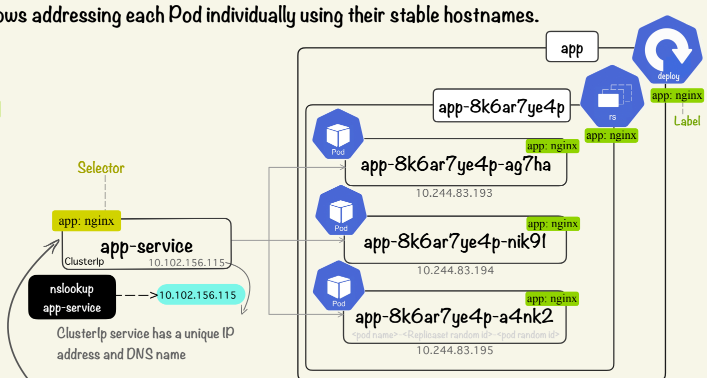
Headless service

A Headless Service is a type of Kubernetes service that does not have a ClusterIP assigned to it. Instead, it manages the Domain Name System (DNS) records directly. This means that when a client tries to connect to a Pod that is part of the Headless Service, it can use the DNS name associated with the Pod's IP address to directly communicate with the Pod. When used with StatefulSets, it allows addressing each Pod individually using their stable hostnames.



Headless-StatefulSet

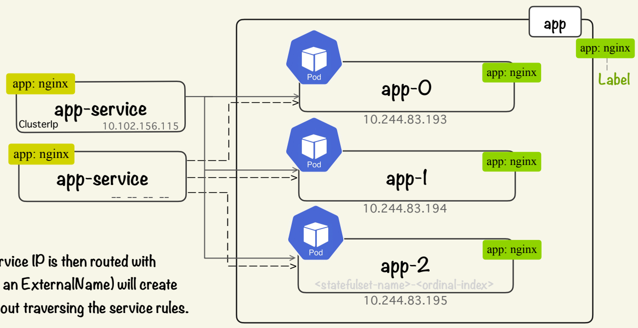
When a client sends traffic to a Headless Service, Kubernetes returns the IP addresses of all the Pods that are backing the service, regardless of their status. This means that the client may receive IP addresses for Pods that are not running or are in a failed state. The client is then responsible for load-balancing the traffic across the individual Pod IP addresses that are returned



ClusterIP-Deployment

When a client sends traffic to the service, Kubernetes chooses one of the Pods based on a load-balancing algorithm. Regular services use a ClusterIP address to load-balance traffic across the Pods that are backing the service

- Regular service provides a single IP address that represents a group of Pods, while a Headless Service provides individual DNS names and IP addresses for each Pod in the service
- Regular services are typically used for stateless applications that can handle traffic from multiple clients, while Headless Services are more commonly used for stateful applications that require direct access to individual Pods
- Headless services can be used in combination with regular services to provide both direct access to individual pods and load-balanced access to the service as a whole. For example, you might use a headless service to allow database nodes to communicate directly with each other, while also exposing a regular service for client applications to connect to
- Regular service has a virtual Service IP that exists as iptables or ipvs rules on each node. A new connection to this service IP is then routed with DNAT to one of the Pod endpoints, to support a form of load balancing across multiple pods.A headless service (that isn't an ExternalName) will create DNS A records for any endpoints with matching labels or name. Connections will go directly to a single pod/endpoint without traversing the service rules.



Headless-ClusterIP -StatefulSet