

# Kubernetes Advanced



**kubernetes**

# Session #02

## Other Workloads



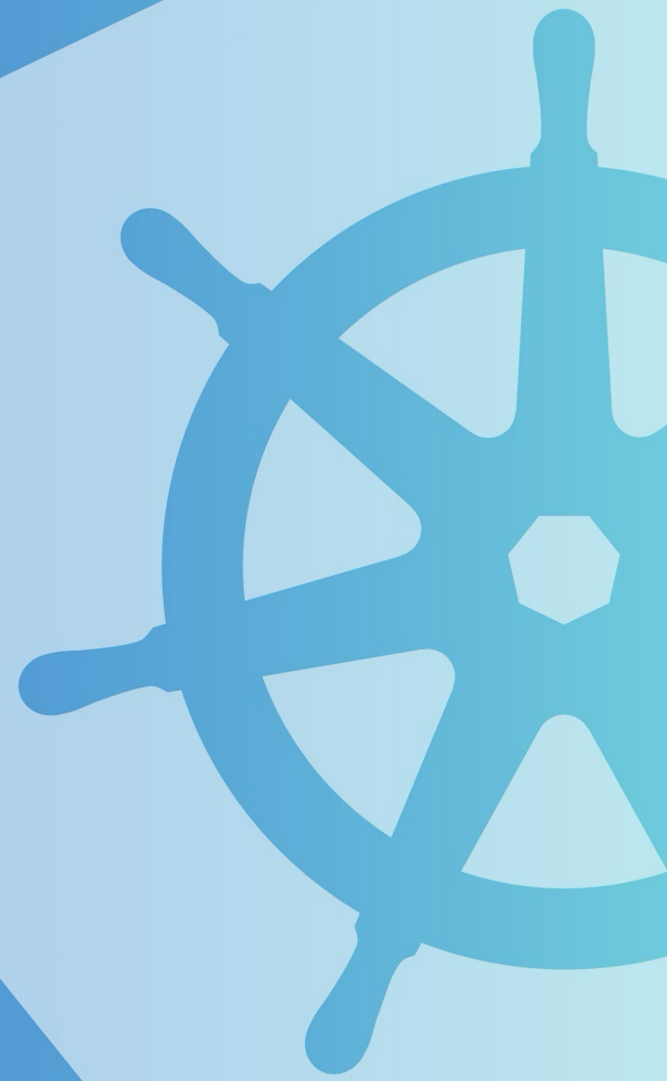
**kubernetes**

# Session Contents



- DaemonSets
- Jobs & CronJobs
- StatefulSets

# DaemonSets



# DaemonSets



- Daemonset is another controller that manages pods like Deployments or ReplicaSets
- 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.

# DaemonSets: Typical Uses

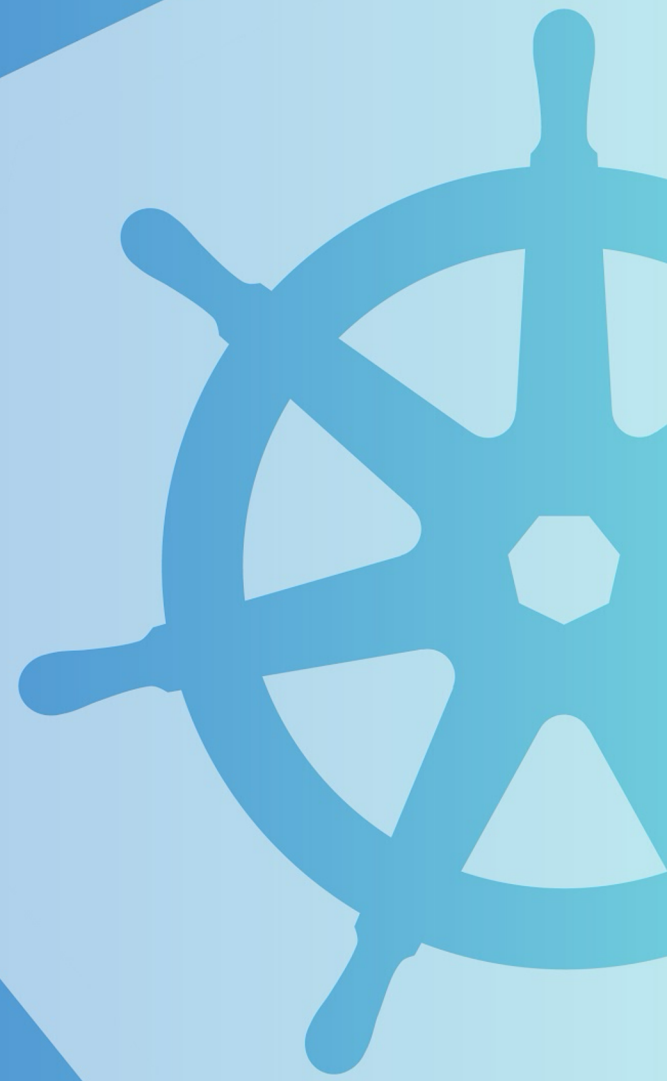


- Running a cluster storage daemon on every node
- Running a logs collection daemon on every node
- Running a node monitoring daemon on every node

# Demo | DaemonSets



# Jobs & CronJobs





# Motivation



- When Kubernetes controller detects a container has failed, it will attempt to restart it based on the container's restartPolicy: Always, Never or OnFailure
- When controlled by a Deployment, containers in Pods are expected to run continuously. Thus, the only restart policy a Deployment supports is Always.
- However, what if you want to run a container that's expected to perform a finite task and then just stop? Just like an Init Container, but without creating any other container afterwards?

# Job



- A Job is a Kubernetes controller that creates and run task-based workload.
- A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate.
- As Pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (i.e., Job) is complete.
- Deleting a Job will clean up the Pods it created.
- Containers defined as part of a Job only support Never and OnFailure restart policies.
- To make changes you need to delete and recreate

# Job Types



- Multiple Parallel Jobs: Run multiple jobs (pods) in parallel to make it faster (when possible) to process. This can be set using **parallelism** property
- Parallel Jobs with Fixed Completion Count: These jobs occur concurrently but run a set number of times before terminating successfully. By setting **completions** to a value greater than one, you trigger the formation of successful pods.
- Non-parallel Jobs: This specifies a job that executes single-handedly or independently. Only one successful pod is started, with additional pods forming in response to any startup failures. Once a pod terminates successfully, that specific job is complete.

# CronJob

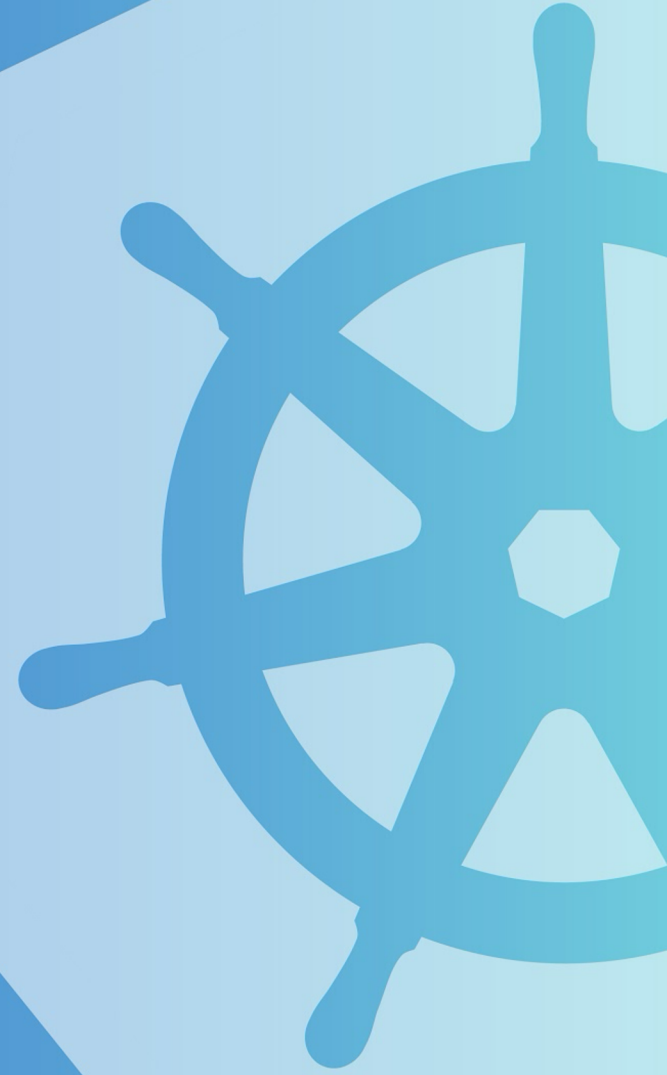


- A CronJob is a Kubernetes controller that creates and manages Jobs according to a cron schedule.
- CronJobs are useful for creating periodic and recurring tasks, like running backups or sending emails.
- CronJobs maintain a certain number of successful and failed jobs, so their logs can be examined.
- When a CronJob deletes a Job, that Job's Pods are deleted.
- When a CronJob is deleted, the Jobs it created (and their Pods) are deleted.

# Demo | Jobs & CronJobs



# StatefulSets



# Motivation



- Using Deployments you need to build your application (or components) on a stateless approach
- Your pods can be deleted and recreated without having a consistency on names and IPs
- Additionally, some solutions to have an associated volume to be always available when running and not depending from a PVC defined on another resource
- For these needs, you should use a StatefulSet

# StatefulSet



- StatefulSet is a pod controller designed to manage stateful applications
- Unlike Deployments, a StatefulSet maintain a sticky identity for each of its Pods
- Pods are created from the same spec but are not interchangeable: each Pod has a persistent identifier that it maintains across rescheduling
- Pods are created in order and are assigned sequential numbers, starting with 0...N-1.



# StatefulSet



- StatefulSets define a Volume Claim Templates, which define parameters to use when dynamically creating PersistentVolumeClaims and PersistentVolumes for each Pod.
- Pod order and persistent volume claims are maintained when Pods are deleted.
- Replacement Pods are linked to the same PVCs.
- Deleting Pods in a StatefulSet (or the StatefulSet itself) will use by default retain reclaim policy
- Can be configured delete reclaim policy when scale down or delete the StatefulSet

# Persistent Volumes



- Persistent volumes exist beyond the lifetime of a pod
- PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed using PersistentVolume and PersistentVolumeClaim
- PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes
- PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources
- To have a match between PVC and PV, a PV needs to match all PVC requests

# Persistent Volume Type



- PersistentVolume types are implemented as plugins. Kubernetes currently supports the following plugins:
  - [cephfs](#) - CephFS volume
  - [csi](#) - Container Storage Interface (CSI)
  - [fc](#) - Fibre Channel (FC) storage
  - [hostPath](#) - HostPath volume (for single node testing only; **WILL NOT WORK** in a multi-node cluster; consider using local volume instead)
  - [iscsi](#) - iSCSI (SCSI over IP) storage
  - [local](#) - local storage devices mounted on nodes.
  - [nfs](#) - Network File System (NFS) storage
  - [rbd](#) - Rados Block Device (RBD) volume

# Container Storage Interface (CSI)



- Defines a standard interface for Kubernetes to expose arbitrary storage systems to their container workloads
- Once a CSI volume driver is deployed on a Kubernetes cluster, users may use the csi volume type to attach or mount the volumes exposed by the CSI driver
  - Through a reference to a **PersistentVolumeClaim**
  - With a generic ephemeral volume
- Popular CSI drivers
  - [Azure File CSI Driver for Kubernetes](#)
  - [Azure Disk CSI driver for Kubernetes](#)
  - [Amazon Elastic Block Store \(EBS\) CSI driver](#)
  - [Google Compute Engine Persistent Disk CSI Driver](#)

# Storage Classes



- Provides a way for administrators to describe the "classes" of storage they offer
- Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators
- Storage classes are used to allow dynamic allocation of PVs
- Uses 3 concepts: provisioner, parameters and reclaimPolicy
- Provisioner: What volume plugin to be used (PV Type)
- Parameters: How to config/setup the provisioner
- ReclaimPolicy: What to do when volume is no longer needed

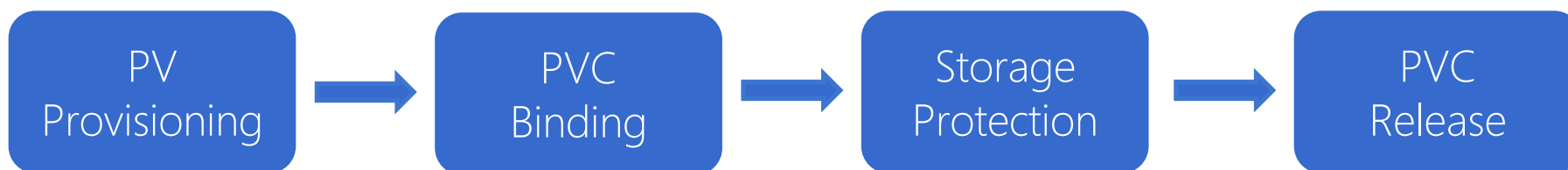


# Storage Classes: Reclaim Policy



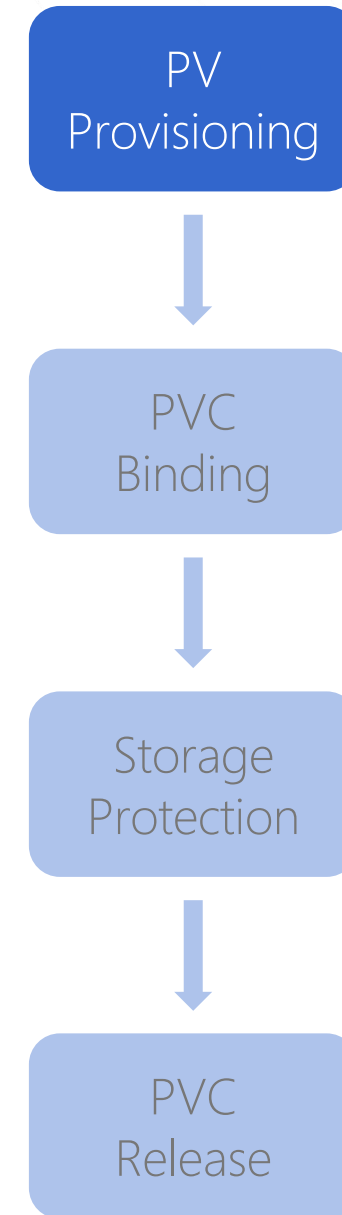
- Reclaim Policy can be Delete or Retain
- Delete means that the storage volume is deleted when it is no longer required by the pod
- Retain means that the storage volume is retained when no longer required by the pod and can be reused by other pods
- If no reclaimPolicy is specified when a StorageClass object is created, it will default to Delete.

# Persistent Volumes Lifecycle



# PV Provisioning

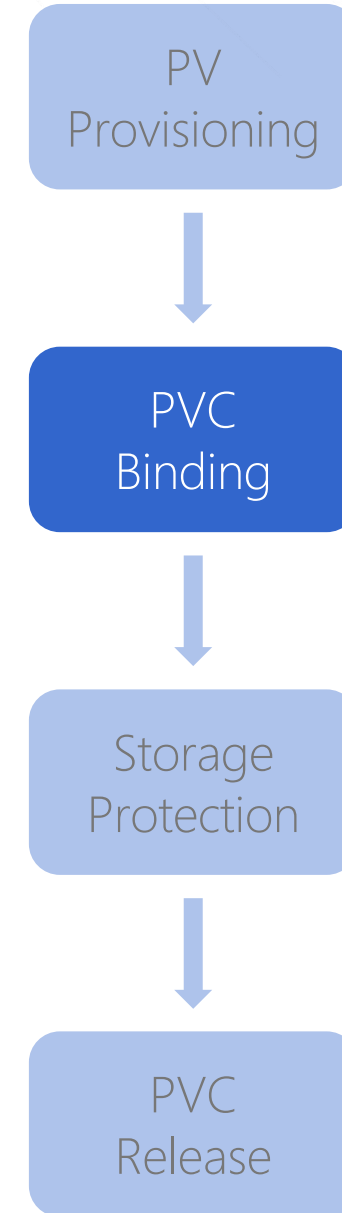
- There are two ways PVs may be provisioned: static or dynamic.
- Static: A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users
- Dynamic: When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on StorageClasses





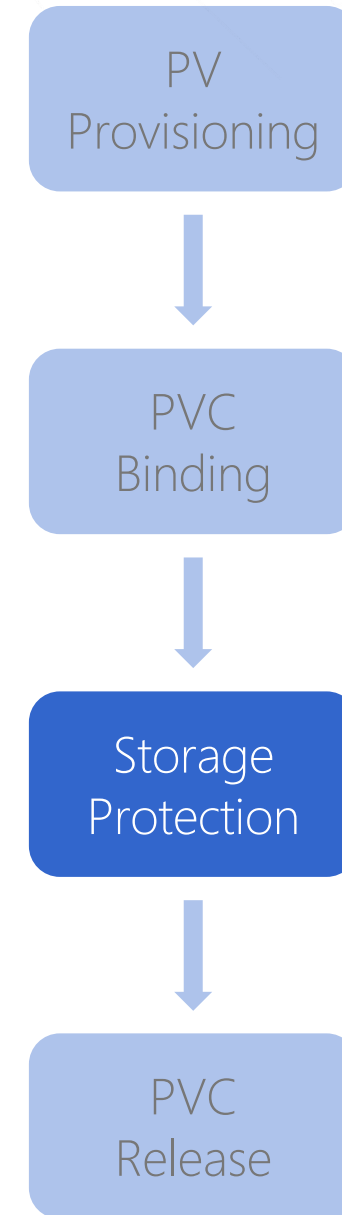
# PVC Binding

- A user creates a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes
- Claims will remain unbound indefinitely if a matching volume does not exist
- Access Modes
  - ReadWriteOnce (RWO): RW on single node for all pods on node
  - ReadOnlyMany (ROX): RO on multiple nodes
  - ReadWriteMany (RWX): RW on multiple nodes and pods
  - ReadWriteOncePod (RWOP): RW to a single pod



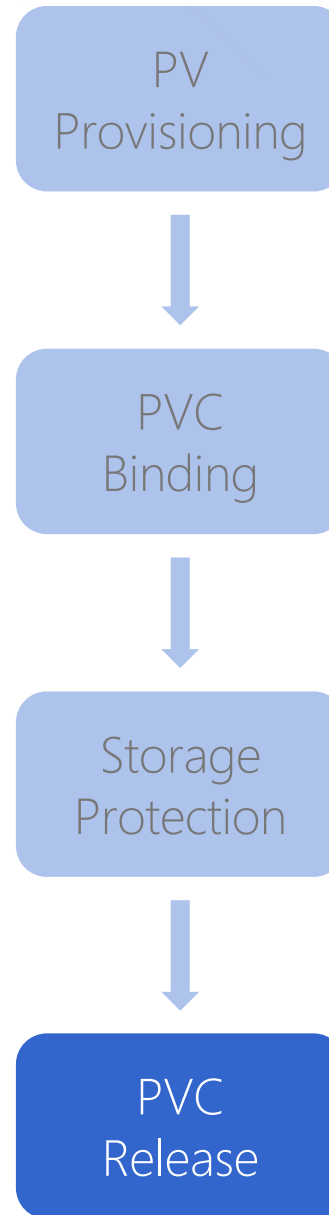
# Storage Protection

- The purpose is to ensure that PersistentVolumeClaims (PVCs) in active use by a Pod and PersistentVolume (PVs) that are bound to PVCs are not removed from the system
- If a user deletes an active PVC, is not removed immediately. PVC removal is postponed until the PVC is no longer actively used by any Pods
- If an admin deletes a PV that is bound to a PVC, the PV is not removed immediately. PV removal is postponed until the PV is no longer bound to a PVC



# PVC Release

- When a user is done with their volume, they can delete the PVC objects from the API that allows reclamation of the resource
- The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim: Retain or Delete
- Retain: Volume is available but data written remains on storage
- Delete: Volume is deleted. Depends on Storage Class



# Headless Service

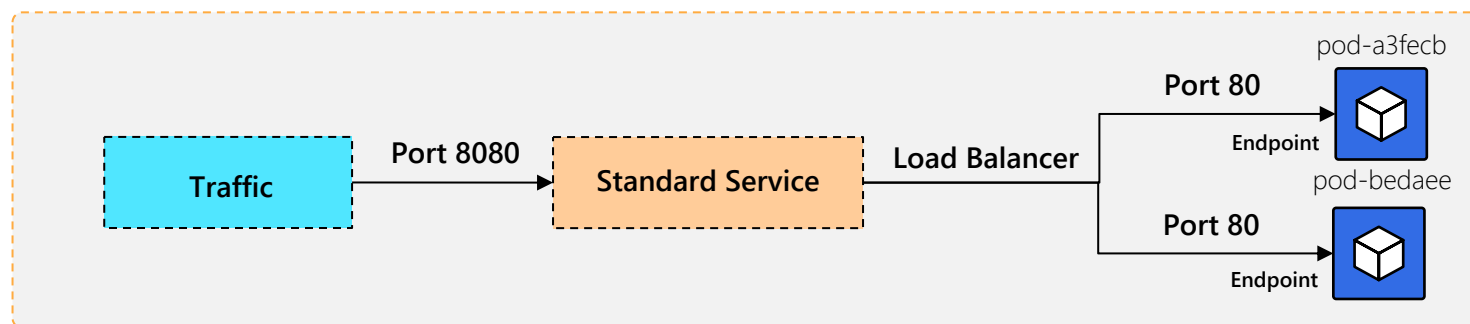


- A “headless” service (`clusterIP=None`) is required for StatefulSets.
- The service is referenced by the Pod definition and allows direct access to each without load balancing.
- Creates a DNS entry for each Pod instead of just for the Service.
- Pods is accessed individually through the headless service using DNS and the FQDN address.
  - Fully Qualified domain name:  
`{pod}.{service}.{namespace}.svc.cluster.local`
  - Example: `mysql.connect("mysql-02.db-service.micro.svc.cluster.local")`

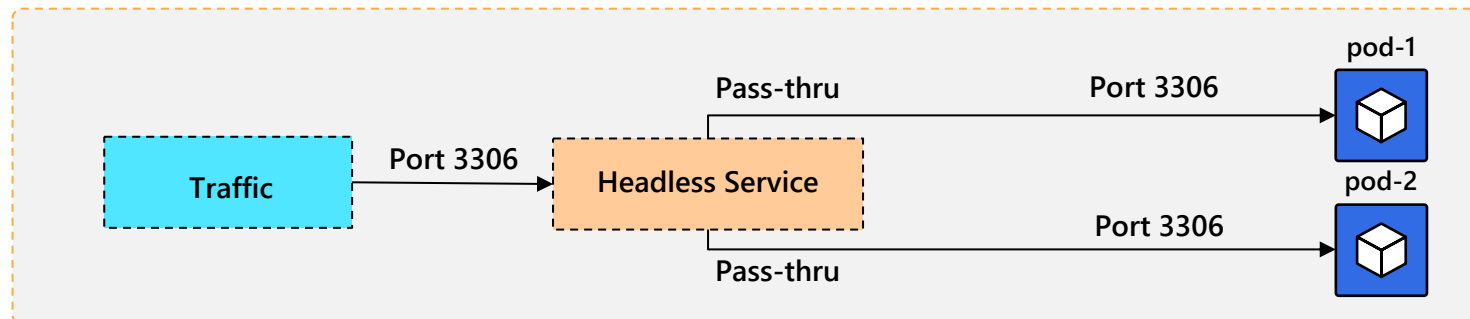
# Headless Service



A standard Service load balances traffic across all Pods listed as Endpoints



A Headless Service doesn't have Endpoints. It routes traffic directly to specific Pods



# Why to use?



- Persistent Storage: Stable and unique network identities for pods and stable persistent storage. This is particularly important for stateful applications like databases, caching systems, and message brokers.
- Ordered Deployment and Scaling: Provide a guaranteed ordering for deployment, scaling, and deletion of pods. This helps you avoid potential data loss or corruption that could occur if pods are deleted or recreated in an incorrect order.
- Ordered Rolling Updates: StatefulSets provide an ordered rolling update mechanism, which enables you to update your stateful application one pod at a time, rather than all at once. This helps minimize downtime and reduces the risk of data loss or corruption during an update.

# Demo | StatefulSets



# Questions?



**kubernetes**



# Lab #06: Other workloads



**kubernetes**