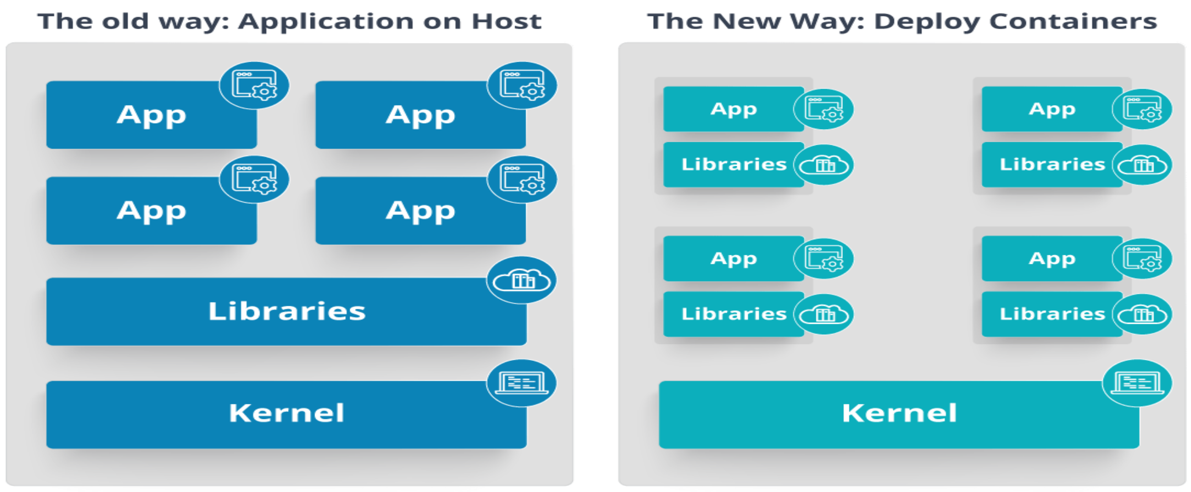
Kubernetes Simple Notes

How is Kubernetes related to Docker?

- Docker is a containerization platform and Kubernetes is a container management (orchestrator) tool for container platforms like Docker.
- Kubernetes uses a Container Runtime Interface (CRI) plugin interface which enables kubelet to use a wide variety of container runtimes, without the need to recompile.
- Kubernetes could use any container runtime that implements CRI to manage pods, containers and container images
- From docker images we are creating containers and Kubernetes is used to manages these containers

What is the difference between deploying applications on hosts and containers?



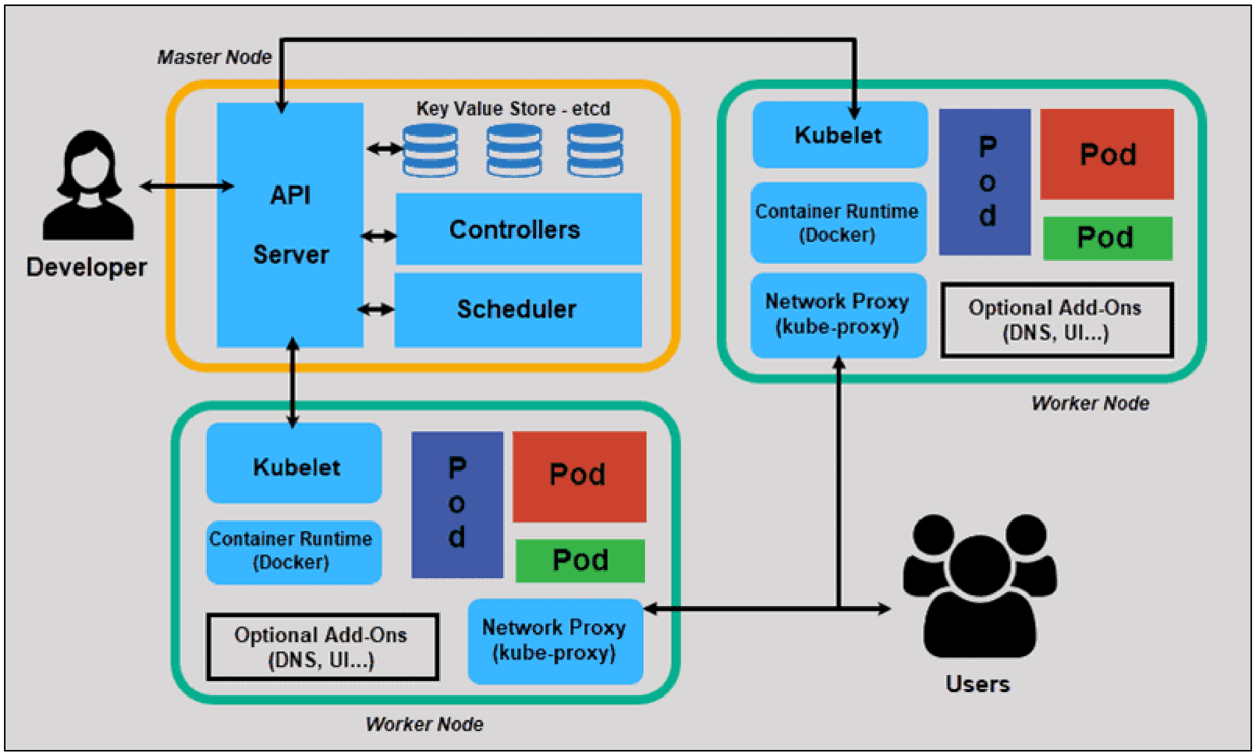
Deploying application on Host	**Deploying application on containers**
Deploying application on Host architecture will have operating system and then the operating system will have a kernel that will have various libraries installed on the operating system needed for the application. So, in this kind of framework you can have n number of applications and all the applications will share the libraries present in that operating system	deploying applications in containers the architecture will have a kernel and that is the only thing that's going to be the only thing common between all the applications. So, if there's a particular application that needs Java then that particular application we'll get access to Java and if there's another application that needs Python then only that particular application will have access to Python
One application depends on other application need of re-deploying full OS for small changes	individual blocks are containerized and these are isolated from other applications. So, the applications have the necessary libraries and binaries isolated from the rest of the system and cannot be encroached by any other application.
Allocates required memory	Requires less memory space
Hardware-level virtualization	OS virtualization
Fully isolated and hence more secure	Process-level isolation, possibly less secure

Architecture of Kubernetes

In Kubernetes, various sub-components can be grouped into two main components. The main components are:

Master nodes

Worker nodes



Master Node

- The management of a cluster is the responsibility of the master node as it is the first point of contact for almost all administrative tasks for the cluster.
- Depending upon the setup, there will be one or more master nodes in a cluster. This is done to keep an eye on the failure tolerance.
- master node comprises different components such as Controller-manager, ETCD, Scheduler, and API Server.
  1. **API Server**: It is the first point of contact for the entirety of the REST commands, which are used to manage and manipulate the cluster.
  2. **Scheduler**: The scheduler, as its name suggests, is responsible for scheduling tasks to the worker nodes. It also keeps the resource utilization data for each of the slave nodes.
  3. **ETCD**: It is majorly employed for shared configuration, as well as for service discovery. It is basically a distributed key-value store.
  - Kubernetes uses etcd as a key-value database store. It stores the configuration of the Kubernetes cluster in etcd.
  - It also stores the actual state of the system and the desired state of the system in etcd.
  - It then uses etcd’s watch functionality to monitor changes to either of these two things. If they diverge, Kubernetes makes changes to reconcile the actual state and the desired state.

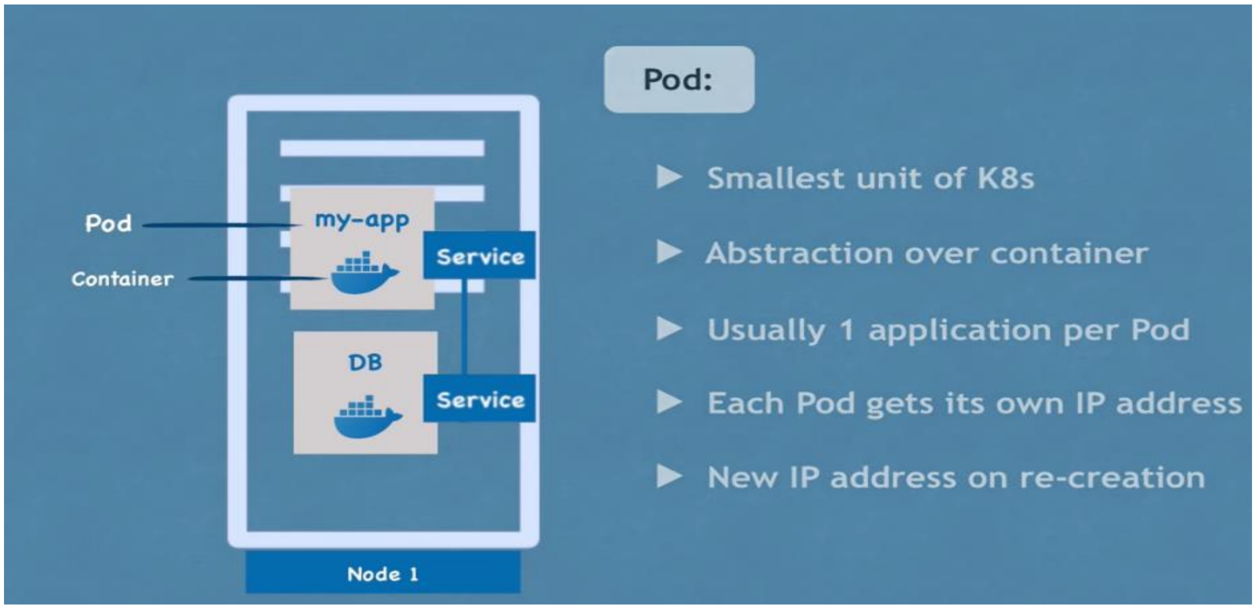
4. **Controller-manager**: It is a daemon that is responsible for regulating the cluster in Kubernetes, and it also manages various other control loops that are non-terminating.
   - ❖ **Replication controller:** Manages pod replication
   - ❖ **Node controller**: Responsible for noticing and responding when nodes go down.
   - ❖ **Job controller**: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
   - ❖ **Endpoints controller**: Populates the Endpoints object (that is, joins Services & Pods).
   - ❖ **Service Account & Token controllers**: Create default accounts and API access tokens for new namespaces

Worker/Slave Nodes

- Worker or slave nodes consist of all the needed services that are required to manage networking among containers.
- The services communicate with the master node and allocate resources to scheduled containers.
- worker nodes have the following components:

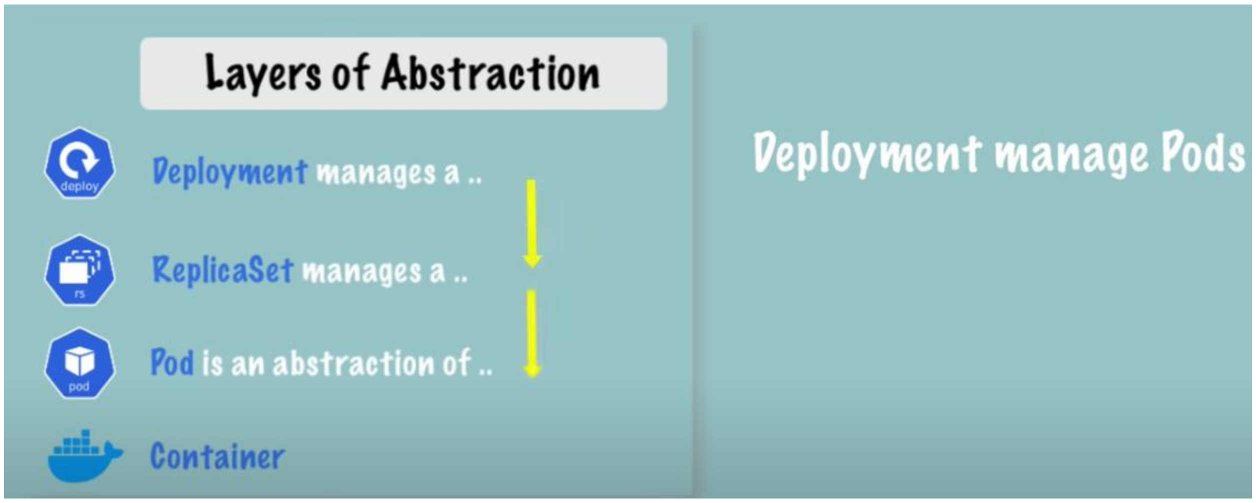
1. **Docker container**
   - Docker must be initialized and run on each worker node in a cluster.
   - Docker containers run on each worker node, and they also run the pods that are configured.
2. **Kubelet**
   - The job of kubelet is to get the configuration of pods from the API server.
   - It is also used to ensure that the mentioned containers are ready and running.
3. **Kube-proxy**
   - Kube-proxy behaves like a network proxy and act as a load balancer for a service on any single worker node for pods running on the node by implementing east/west load-balancing using NAT in iptables
   - The kube-proxy handles network communications inside or outside the cluster
4. **CAdvisor:**
   - Used for monitoring resource usage and performance
5. **Label:**
   - Used to identify pods
6. **Pods**
   - A pod can be thought of as one or more containers, which can logically run on nodes together.

What is a pod?



Pod:

- Smallest unit of K8s
- Abstraction over container
- Usually 1 application per Pod
- Each Pod gets its own IP address
- New IP address on re-creation



Layers of Abstraction

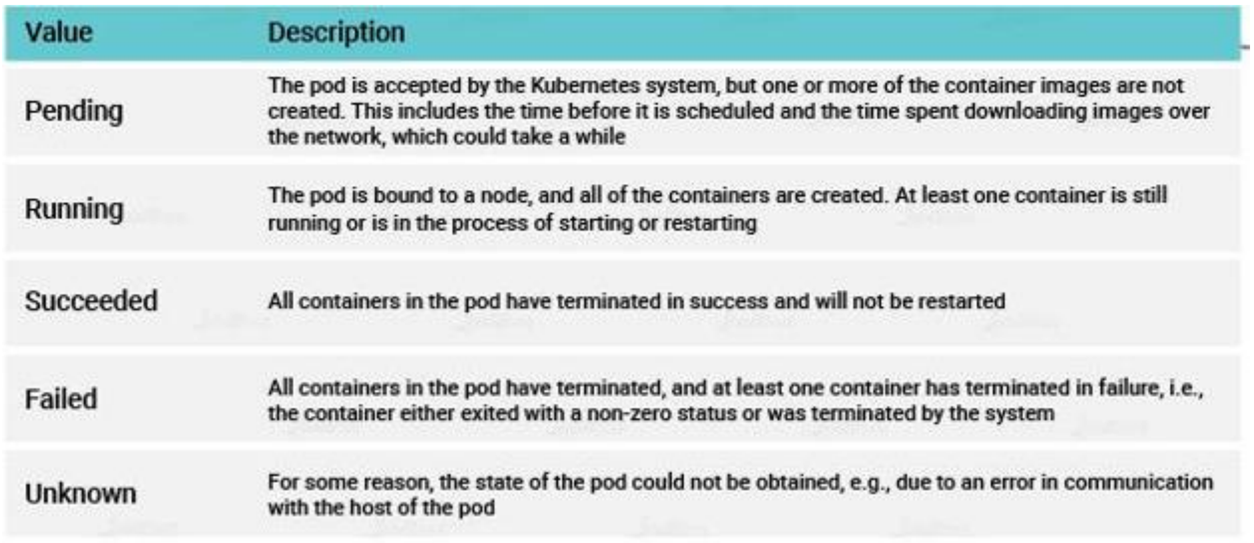
- Deployment manages a ..
- ReplicaSet manages a ..
- Pod is an abstraction of ..
- Container

Deployment manage Pods

- A group of one or more containers is called a Pod.
- Pods are also the simplest unit in the Kubernetes object model, which we can create and deploy.
- Pods life is short in nature, and they do not have the capability to self-heal by themselves. That is why we use them with controllers, which can handle a Pod's replication, fault tolerance, self-heal, etc.
- Examples of controllers are Deployments, Replica Sets, Replication Controllers, etc. We attach the Pod's specification to other objects using Pod Templates

  Limitations of POD
- A pod is a single entity, and if it fails, it cannot restart itself. This won't suit most use cases, as we want our applications to be highly available. But Kubernetes has this issue solved by using with controllers

Various phases of a pod are shown in the image below:



Value	Description
Pending	The pod is accepted by the Kubernetes system, but one or more of the container images are not created. This includes the time before it is scheduled and the time spent downloading images over the network, which could take a while
Running	The pod is bound to a node, and all of the containers are created. At least one container is still running or is in the process of starting or restarting
Succeeded	All containers in the pod have terminated in success and will not be restarted
Failed	All containers in the pod have terminated, and at least one container has terminated in failure, i.e., the container either exited with a non-zero status or was terminated by the system
Unknown	For some reason, the state of the pod could not be obtained, e.g., due to an error in communication with the host of the pod

Pod manifest file:

```
$ vi hello-pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: hello-pod
  labels:
     zones: prod
     version: v1
spec:
  containers:
  - name: hello-ctr
    image: nigelpoulton/pluralsight-docker-ci:latest
    ports:
    - containerPort: 8080

$ kubectl create -f hello-pod.yml
pod/hello-pod created
```

To access our hello-pod/ Replication Controller /Deployment we need to expose the pods though a service:

kubectl expose

When a pod is created, without a service, we cannot access to the app running within container in the pod. The most obvious way is to create a service for the pod either via Load Balancer or NodePort.

```
$ kubectl expose pod hello-pod --type=NodePort --target-port=80 -o yaml
```

```
$ kubectl describe pod hello-pod | grep -i ip
IP:            10.244.0.83
```

```
$ curl http://localhost:30779
http://10.244.0.83:30779/
```

```
$ kubectl get svc hello-pod -o wide
NAME    TYPE    CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE  SELECTOR
hello-pode NodePort 10.107.83.213 <none> 8080:30779/TCP 14m  version=v1,zones=prod
```

Implicit way of defining POD, RC, Deployment, Service

POD	Replication Controller	Deployment	Service
apiVersion=v1 **Kind=Pod** labels: zones: prod version: v1	apiVersion=v1 Kind= **ReplicationController** labels: zones: prod version: v1	apiVersion=apps/v1 Kind= **Deployment** labels: zones: prod version: v1	apiVersion=v1 Kind= Service Selectors: zones: prod version: v1

Explicit way of defining

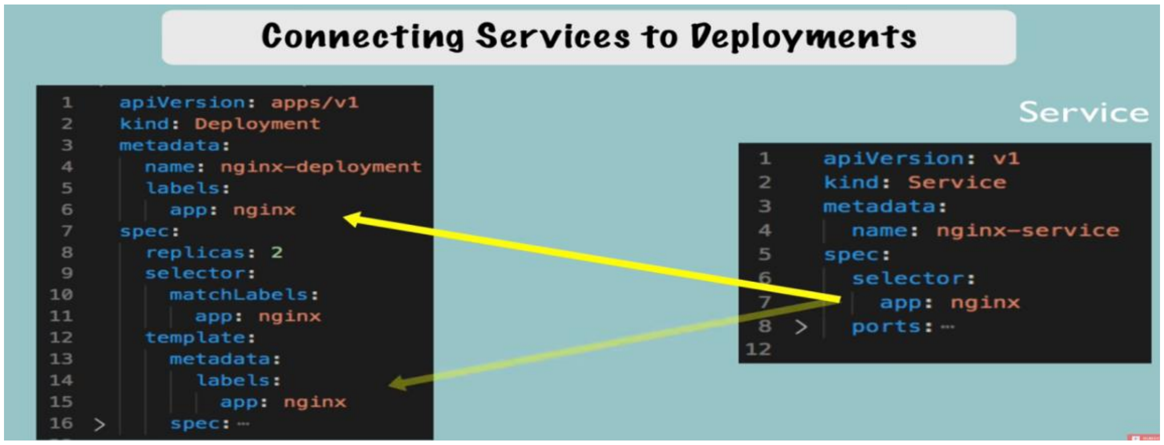
kubectl create Pod new-nginx --image=nginx:latest --- generally not recommended (pods are usually created by deployment)

kubectl create ReplicationController new-nginx --image=nginx:latest --- generally not recommended

kubectl create deployment new-nginx --image=nginx:latest—we can create it (it will create deployment,pod)

kubectl create Service new-nginx --image=nginx:latest –we can create it

Labels



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec: ...
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports: ...
```

- Labels are key-value pairs which are attached to pods, replication controller and services.
- They are used as identifying attributes for objects such as pods and replication controller.
- They can be added to an object at creation time and can be added or modified at the run time.
- With **Label Selectors** we can select a subset of objects. Kubernetes supports two types of Selectors:

Equality-Based Selectors

- Equality-Based Selectors allow filtering of objects based on label keys and values. With this type of Selectors, we can use the **=**, **==**, or **!=**operators. For example, with **env==dev** we are selecting the objects where the **env** label is set to **dev**.

Equality Based Requirement

- Equality based requirement will match for the specified label and filter the resources. The supported operators are =, ==, !=

```
adpspl-mac36:~ vbirada$ kubectl get pod --show-labels
NAME                READY   STATUS    RESTARTS   AGE    LABELS
example-pod           1/1     Running   0        17h
env=prod,owner=Ashutosh,status=online,tier=backend

example-pod1          1/1     Running   0        21m
env=prod,owner=Shovan,status=offline,tier=frontend

example-pod2          1/1     Running   0        8m
env=dev,owner=Abhishek,status=online,tier=backend

example-pod3          1/1     Running   0        7m
env=dev,owner=Abhishek,status=online,tier=frontend
```

Now, I want to see all the pods with online status:

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l status=online
NAME         READY    STATUS    RESTARTS   AGE
example-pod    1/1      Running   0         17h
example-pod2   1/1      Running   0         9m
example-pod3   1/1      Running   0         9m
```

Similarly, go through following commands

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l status!=online
NAME                   READY   STATUS    RESTARTS   AGE
example-pod1           1/1     Running   0          25m
example-pod4           1/1     Running   0          11m
```

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l status==offline
NAME           READY   STATUS    RESTARTS   AGE
example-pod1   1/1     Running   0          26m
example-pod4   1/1     Running   0          11m
```

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l status==offline,status=online
No resources found.
```

```
adpspl-mac36:~ vbirada$  kubectl get pods  -l status==offline,env=prod
NAME           READY   STATUS    RESTARTS   AGE
example-pod1   1/1     Running   0          28m
```

```
adpspl-mac36:~ vbirada$ kubectl get pods  -l owner=Abhishek
NAME           READY   STATUS    RESTARTS   AGE
example-pod2   1/1     Running   0          15m
example-pod3   1/1     Running   0          14m
```

In above commands, labels separated by comma is a kind of **AND** satisfy operation.

Set-Based Selectors

- Set-Based Selectors allow filtering of objects based on a set of values. With this type of Selectors, we can use the **in**, **notin**, and **exist** operators. For example, with **env in (dev,qa)**, we are selecting objects where the **env** label is set to **dev** or **qa**.

Set Based Requirement

- Label selectors also support set based requirements. In other words, label selectors can be used to specify a set of resources.
- The supported operators here are in , notin and exists .

Let us walk through kubectl commands for filtering resources using set based requirements.

```
adpspl-mac36:~ vbirada$ kubectl get pod -l 'env in (prod)'
NAME           READY   STATUS    RESTARTS   AGE
example-pod    1/1     Running   0          18h
example-pod1   1/1     Running   0          41m
```

```
adpspl-mac36:~ vbirada$  kubectl get pod -l 'env in (prod,dev)'
NAME           READY   STATUS    RESTARTS   AGE
example-pod    1/1     Running   0          18h
example-pod1   1/1     Running   0          41m
example-pod2   1/1     Running   0          27m
example-pod3   1/1     Running   0          27m
```

Here env in (prod,dev) the comma operator acts as a **OR** operator. That is it will list pods which are in prod or dev.

```
adpspl-mac36:~ vbirada$ kubectl get pod -l 'env in (prod),tier in (backend)'
NAME         READY   STATUS    RESTARTS   AGE
example-pod   1/1     Running   0          18h
```

```
adpspl-mac36:~ vbirada$  kubectl get pod -l 'env in (qa),tier in (frontend)'
No resources found.
```

Here the comma operator separating env in (qa) and tier in (frontend)will act as an AND operator.

To understand the exists operator let us add label region=central to example-pod and example-pod1 and region=northern to example-pod2 .

```
adpspl-mac36:~ vbirada$ kubectl get  pod --show-labels
NAME                  READY   STATUS    RESTARTS   AGE      LABELS
example-pod            1/1     Running   0          18h
env=prod,owner=Ashutosh,region=central,status=online,tier=backend

example-pod1           1/1     Running   0          54m
env=prod,owner=Shovan,region=central,status=offline,tier=frontend

example-pod2           1/1     Running   0          40m
env=dev,owner=Abhishek,region=northern,status=online,tier=backend

example-pod3           1/1     Running   0          40m
env=dev,owner=Abhishek,status=online,tier=frontend

example-pod4           1/1     Running   0          40m      env=qa,owner=Atul,status=offline,tier=backend
```

Now, I want to view pods that is not in central region:

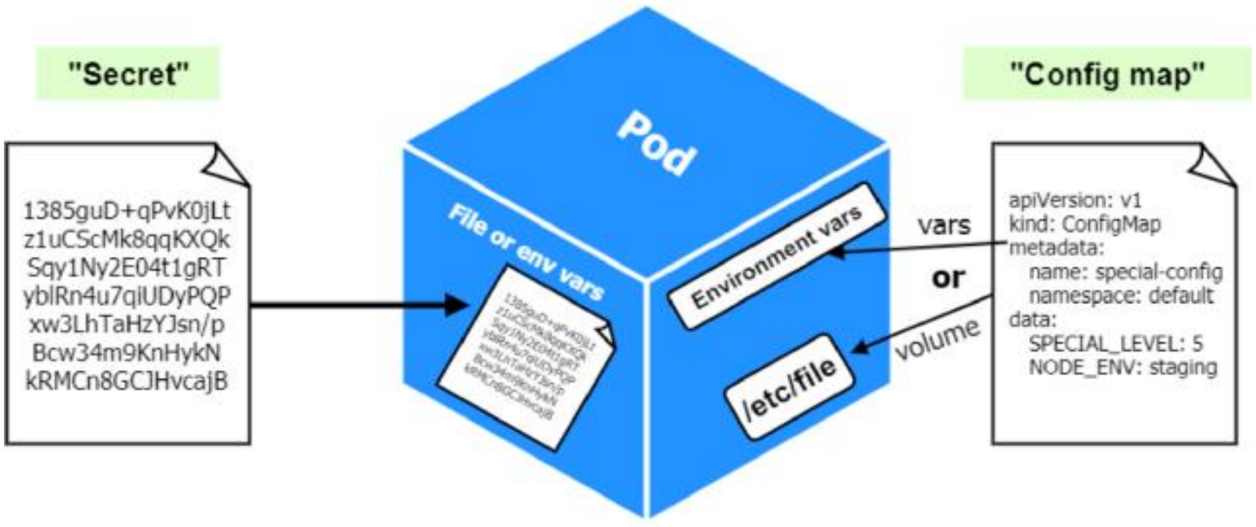
```
adpspl-mac36:~ vbirada$  kubectl get pods -l 'region notin (central)'
NAME                  READY   STATUS    RESTARTS   AGE
example-pod2           1/1     Running   0          42m
example-pod3           1/1     Running   0          42m
example-pod4           1/1     Running   0          41m
```

You can realise here that example-pod2 is having a region key with value northern and hence appears in result. But one point to note is that other two pods in result is not having any region field and will satisfy the condition to appear in result.

If we want that pods having region key should only be the set of resources over which filtering should be done we can restrict via the existsoperator.We do not specifically write exists like we do write in and notinin command.

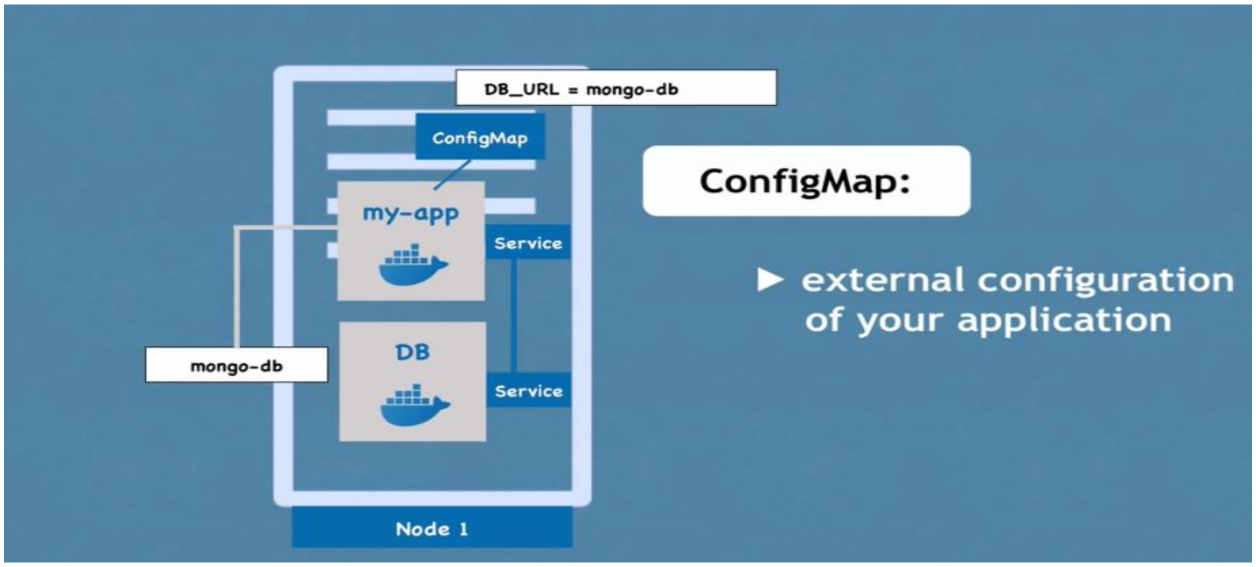
```
adpspl-mac36:~ vbirada$ kubectl get pods -l 'region,region notin (central)'
NAME          READY   STATUS    RESTARTS   AGE
example-pod2   1/1     Running   0          46m
```

Handling Sensitive Information and Container Configurations



```
1385guD+qPvK0jLt
z1uCScMk8qqKXQk
Sqy1Ny2E04t1gRT
yblRn4u7qlUDyPQP
xw3LhTaHzYJsn/p
Bcw34m9KnHykN
kRMCn8GCJHvcajB
```

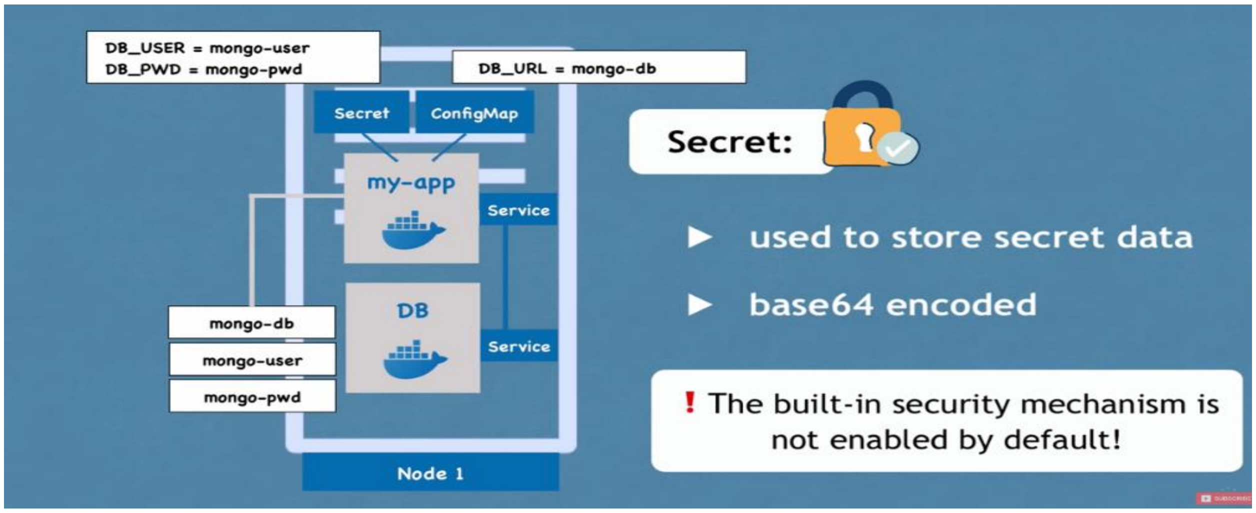
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: 5
  NODE_ENV: staging
```



ConfigMap:

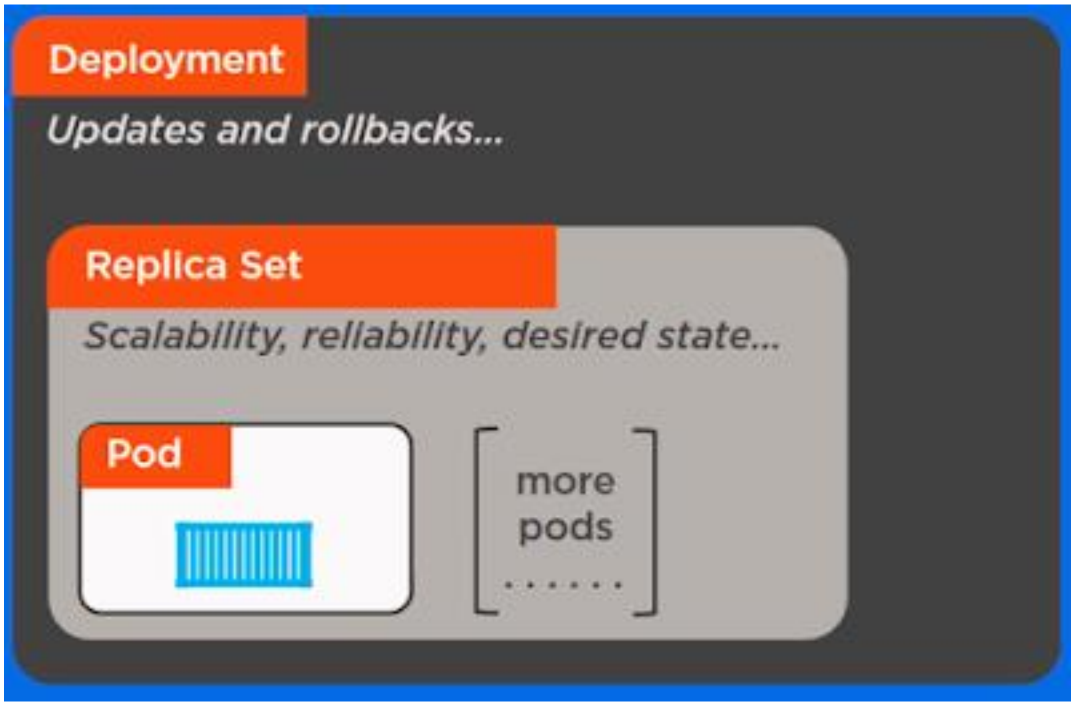
► external configuration of your application

- A ConfigMap is an API object that lets you store configuration for other objects to use.
- Unlike most Kubernetes objects that have a spec , a ConfigMap has data and binaryData fields.
- These fields accept key-value pairs as their values.



- Similar to config maps, secrets can be mounted into a pod as a volume to expose needed information or can be injected as environment variables.
- Secrets are intended to store credentials to other services that a container might need or to store any sensitive information.

What is a deployment in Kubernetes?



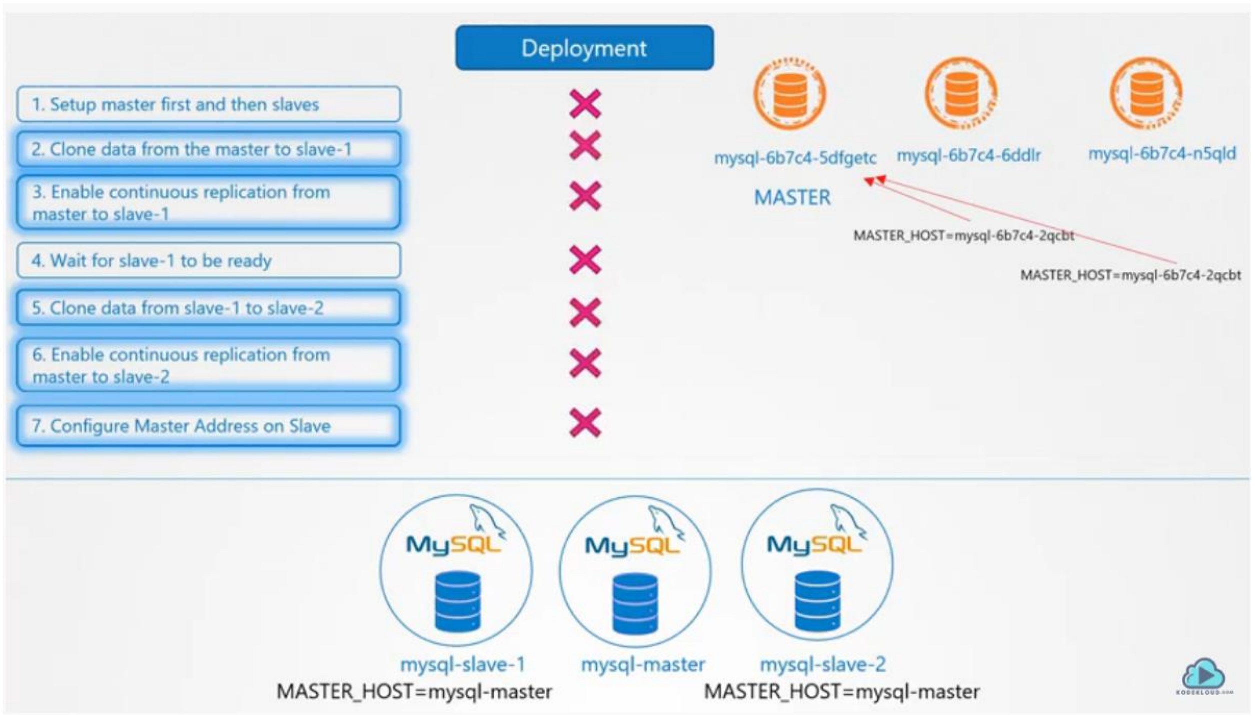
- Deployments in Kubernetes are a set of multiple identical pods.
- A deployment is responsible for running multiple replicas of the application.

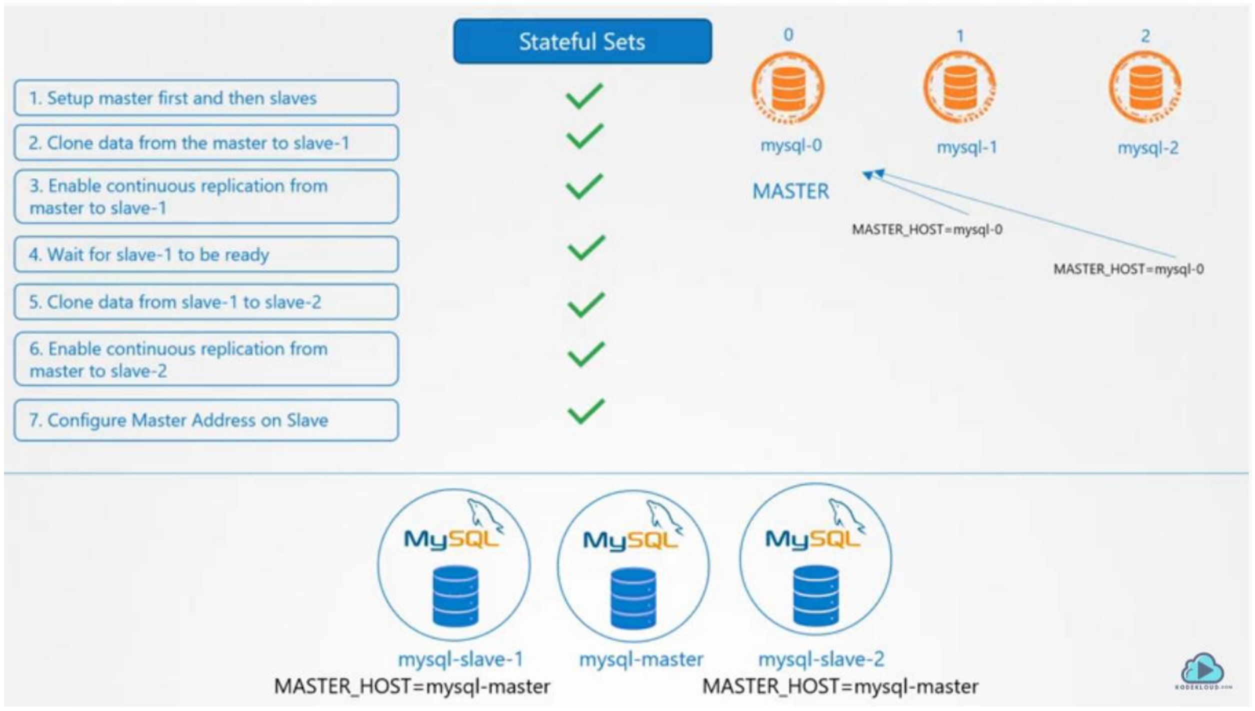
- If one of the instances fails, crashes, or becomes unresponsive, the deployment replaces the instance. This amazing feature makes sure that one of the instances of your application is always available.
- Kubernetes deployment controller manages all of the deployments. To run these replicas, deployments use pod templates.
- These pod templates have specifications on how the pod should look and behave like, e.g., which volumes the pod mounts, labels, taints, etc.
- When you change a deployment's pod template, new pods are created automatically one by one.

Without a deployment

- Running a pod without a deployment can be done, but it is generally not recommended.
- For very simple testing this may be an effective method to increase velocity but for anything of importance this approach has a number of flaws.
- Without a deployment, Pods can still be created and run through unmanaged ReplicaSets.
- While you will still be able to scale your application you lose out on a lot of base functionality deployments provide and drastically increase your maintenance burden.
- Kubernetes now recommends running almost all Pods in Deployments instead of using custom ReplicaSets.

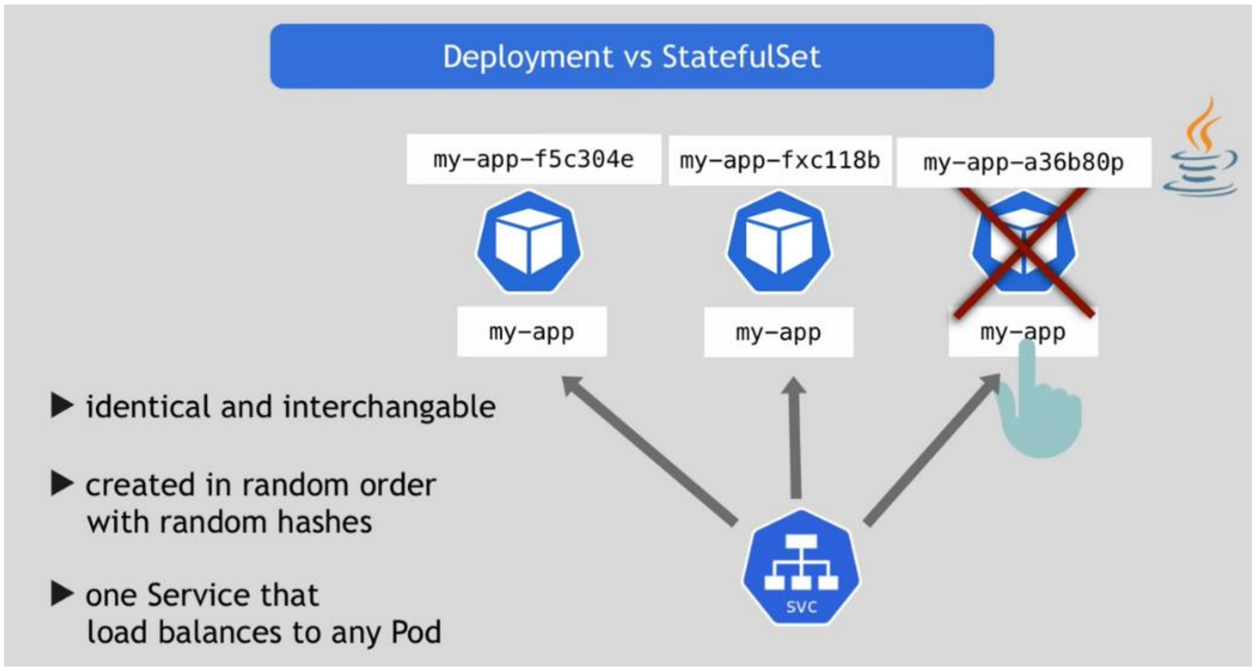
Deployment for Stateless Apps and Statefulset for StateFUL Databases


Deployment
1. Setup master first and then slaves
2. Clone data from the master to slave-1
3. Enable continuous replication from master to slave-1
4. Wait for slave-1 to be ready
5. Clone data from slave-1 to slave-2
6. Enable continuous replication from master to slave-2
7. Configure Master Address on Slave
mysql-6b7c4-5dfgetc
mysql-6b7c4-6ddlr
mysql-6b7c4-n5qld
MASTER
MASTER_HOST=mysql-6b7c4-2qcbt
MASTER_HOST=mysql-6b7c4-2qcbt
MySQL
mysql-slave-1
mysql-master
mysql-slave-2
MASTER_HOST=mysql-master
MASTER_HOST=mysql-master



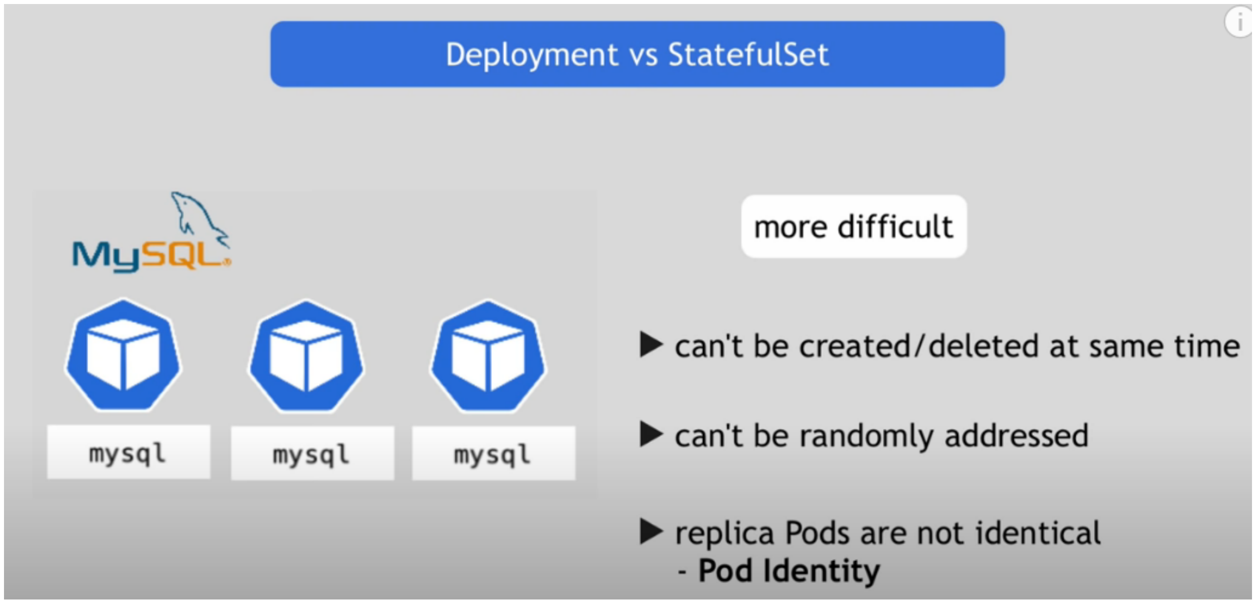
Stateful Sets

1. Setup master first and then slaves ✓
2. Clone data from the master to slave-1 ✓
3. Enable continuous replication from master to slave-1 ✓
4. Wait for slave-1 to be ready ✓
5. Clone data from slave-1 to slave-2 ✓
6. Enable continuous replication from master to slave-2 ✓
7. Configure Master Address on Slave ✓



Deployment vs StatefulSet

- ▶ identical and interchangable
- ▶ created in random order with random hashes
- ▶ one Service that load balances to any Pod

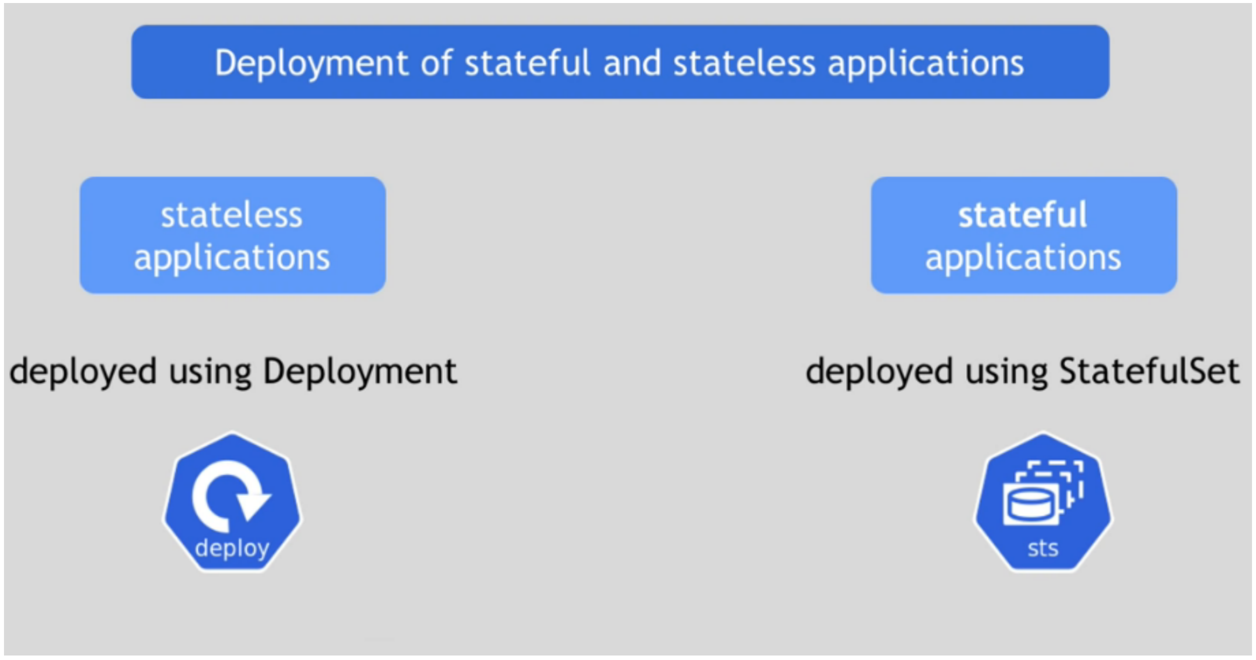


Deployment vs StatefulSet

MySQL
mysql
mysql
mysql

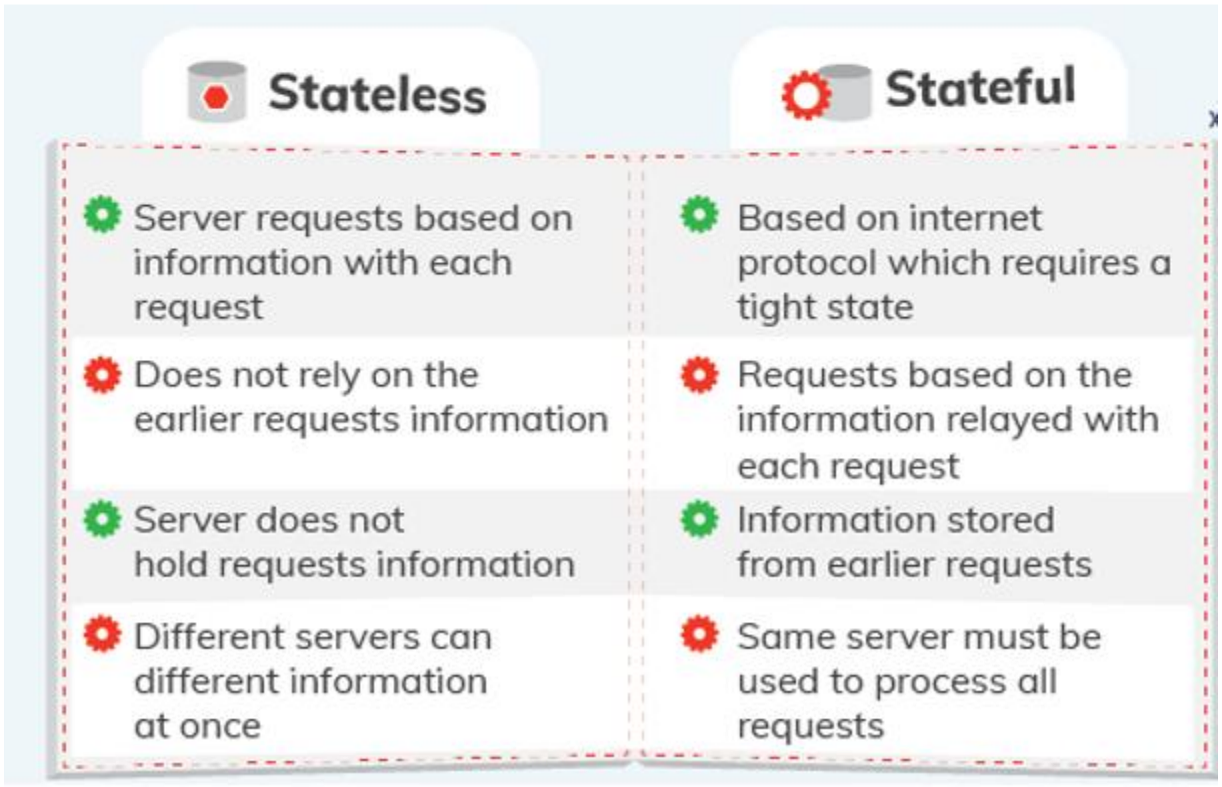
more difficult

- can't be created/deleted at same time
- can't be randomly addressed
- replica Pods are not identical
  - **Pod Identity**



Deployment of stateful and stateless applications

stateless applications	**stateful** applications
deployed using Deployment	deployed using StatefulSet



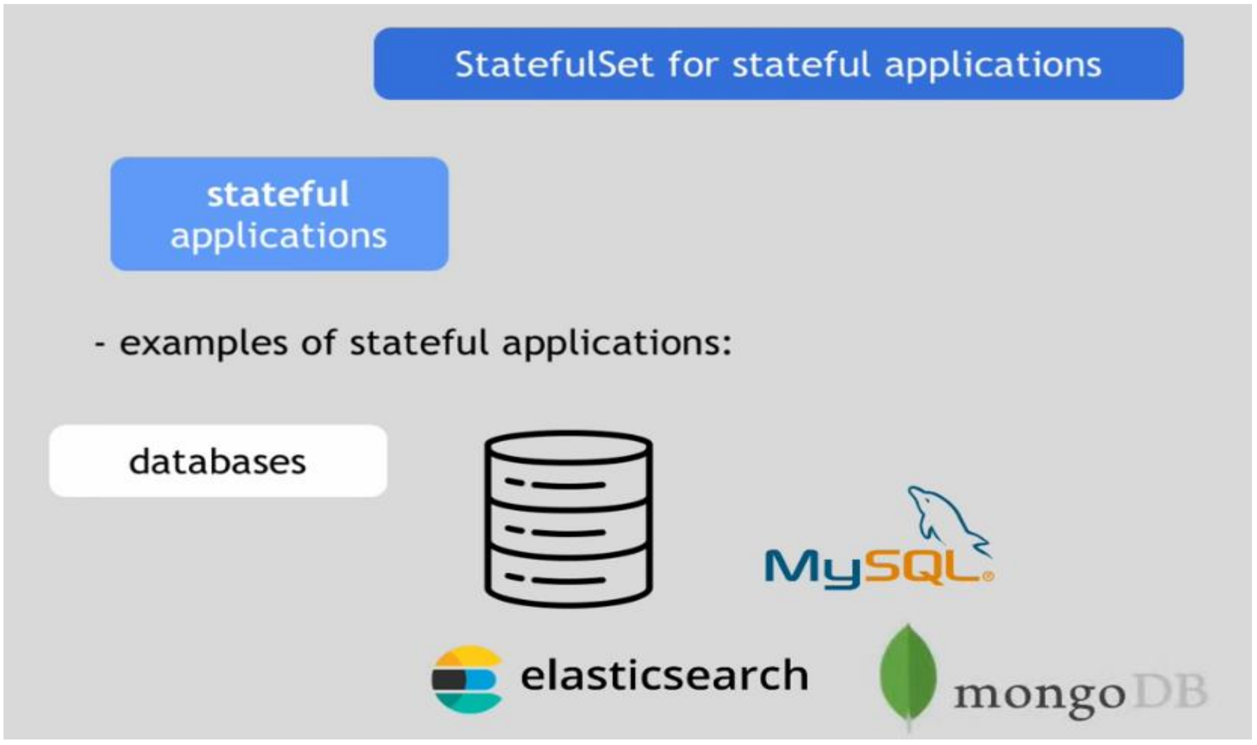
Stateless	Stateful
Server requests based on information with each request	Based on internet protocol which requires a tight state
Does not rely on the earlier requests information	Requests based on the information relayed with each request
Server does not hold requests information	Information stored from earlier requests
Different servers can different information at once	Same server must be used to process all requests

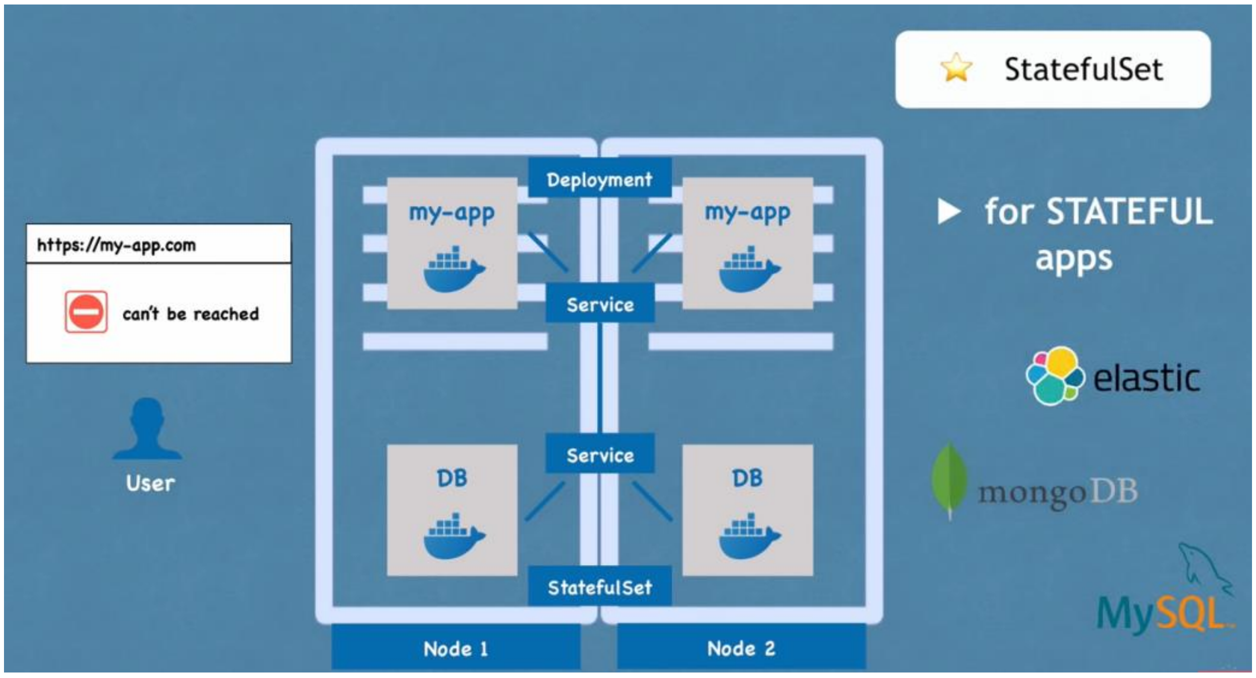
What is a StatefulSet in Kubernetes?

- 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 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.

What is a stateful application?

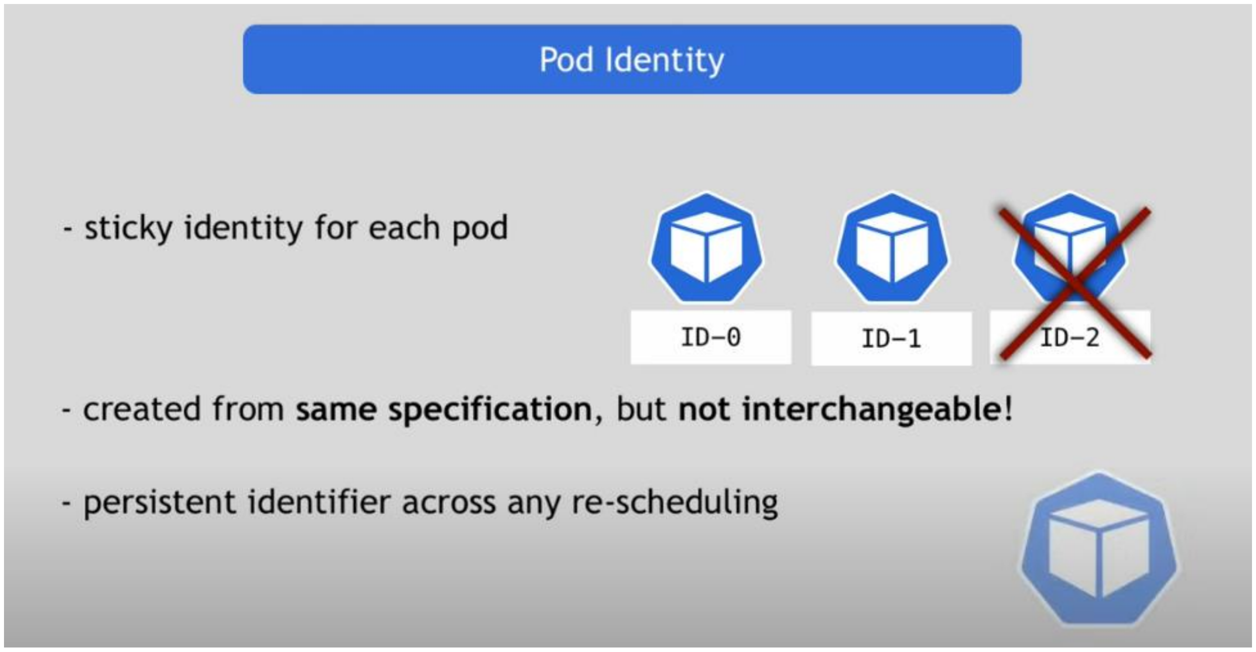
- A Stateful application is the one which saves all the data to a persistent disk storage which will be used by the application clients, other dependent applications or by the server itself.
- For example A database is a stateful application or you can say any key-value store to which data is saved and retrieved by other applications.
- Some popular examples of stateful applications are MongoDB, Cassandra, and MySQL etc.


StatefulSet for stateful applications
stateful applications
- examples of stateful applications:
databases
MySQL
elasticsearch
mongoDB



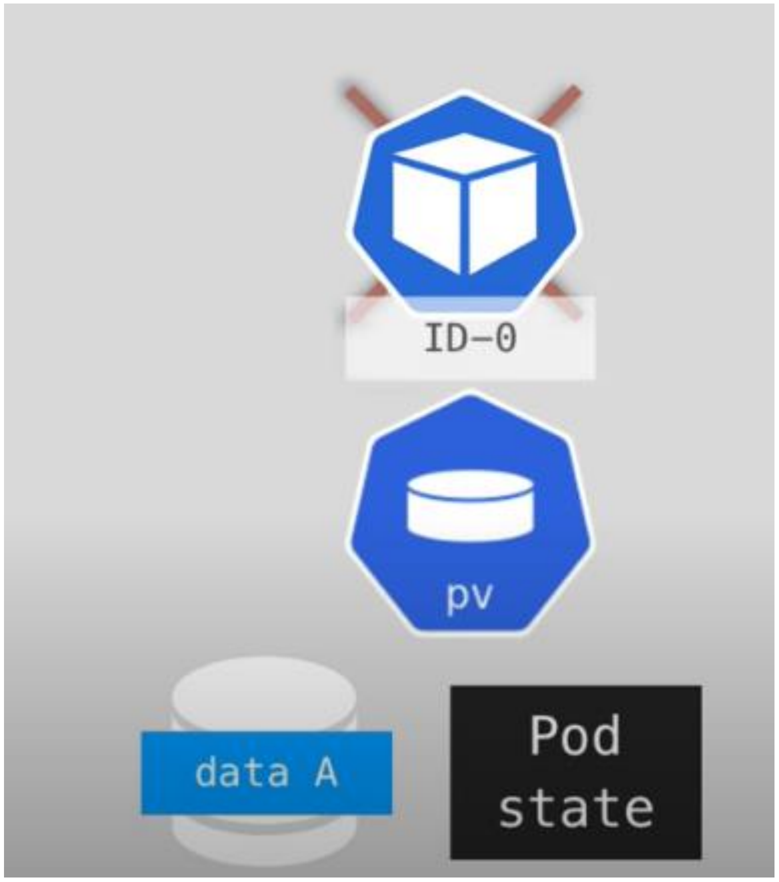
Pod Identity

- StatefulSet Pods have a unique identity that is comprised of an ordinal, a stable network identity, and stable storage. The identity sticks to the Pod, regardless of which node it's (re)scheduled on.

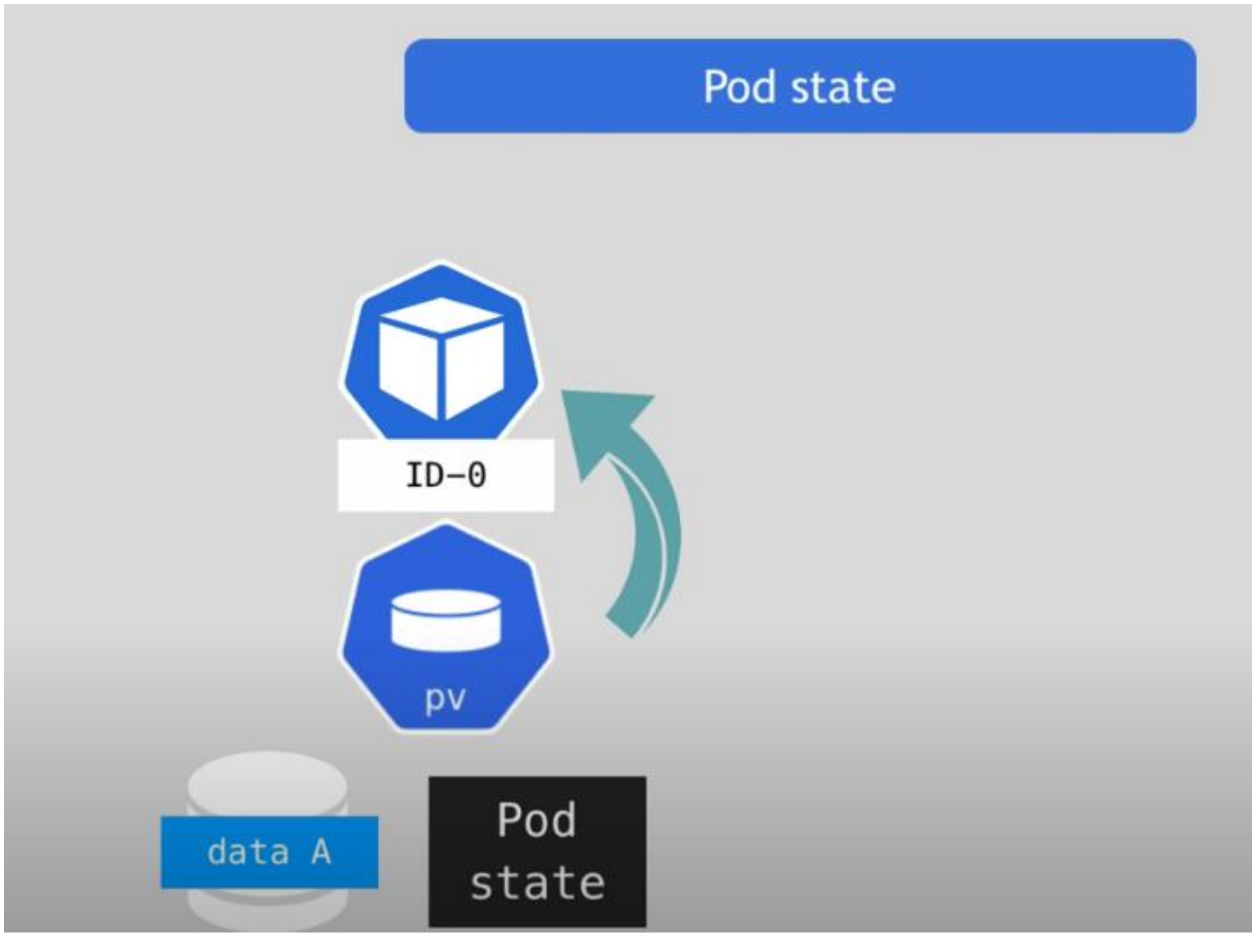


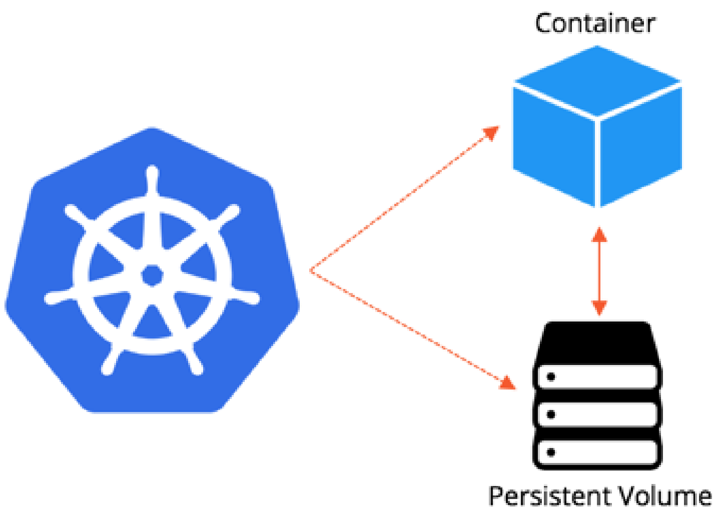
Pod Identity

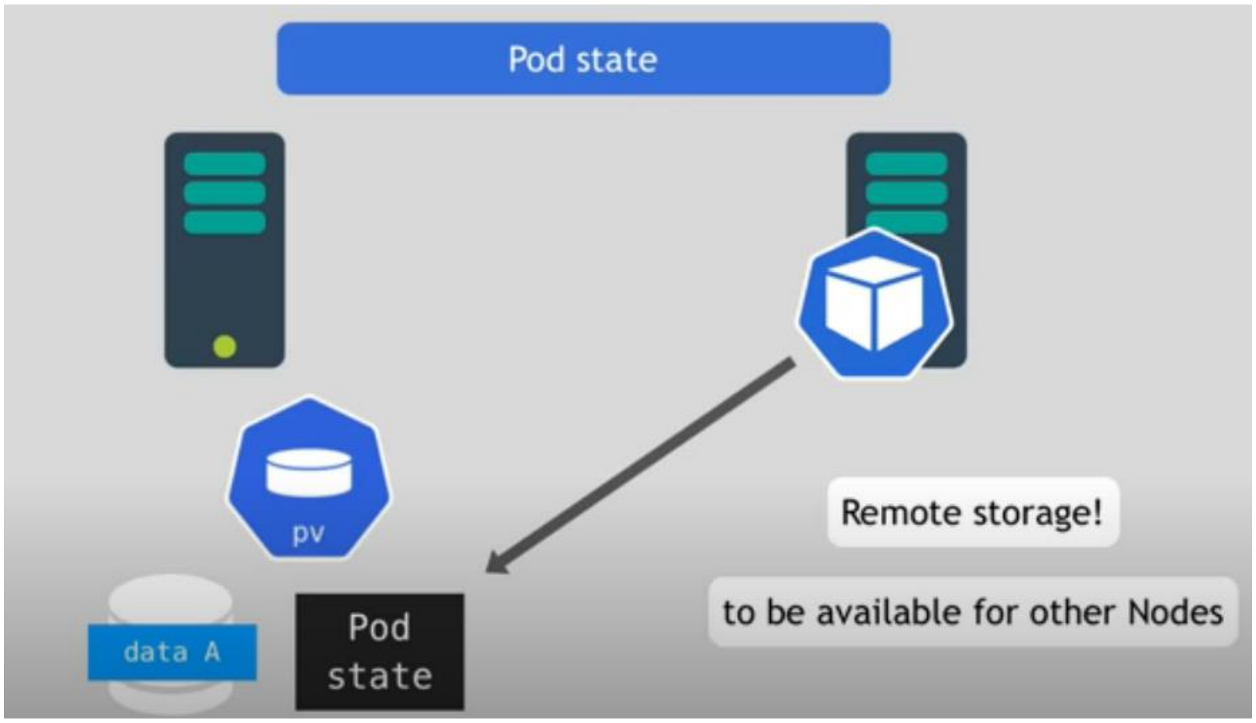
- sticky identity for each pod
- created from **same specification**, but **not interchangeable**!
- persistent identifier across any re-scheduling

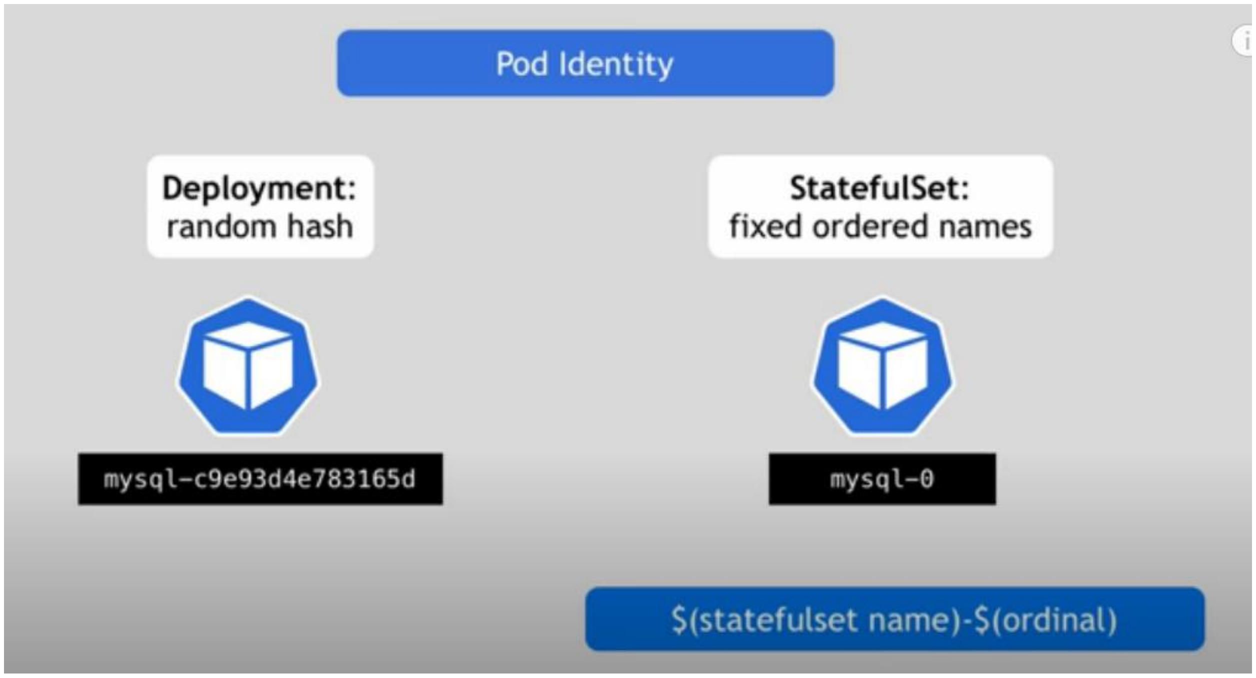


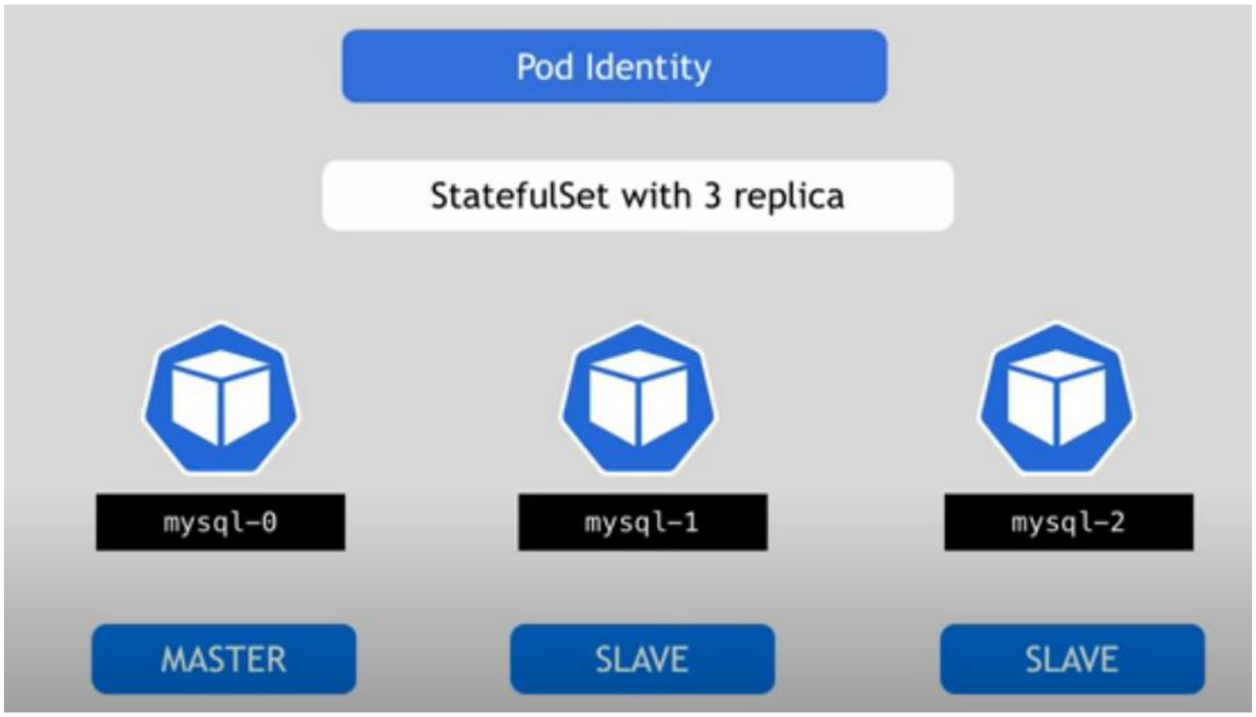
Pod state

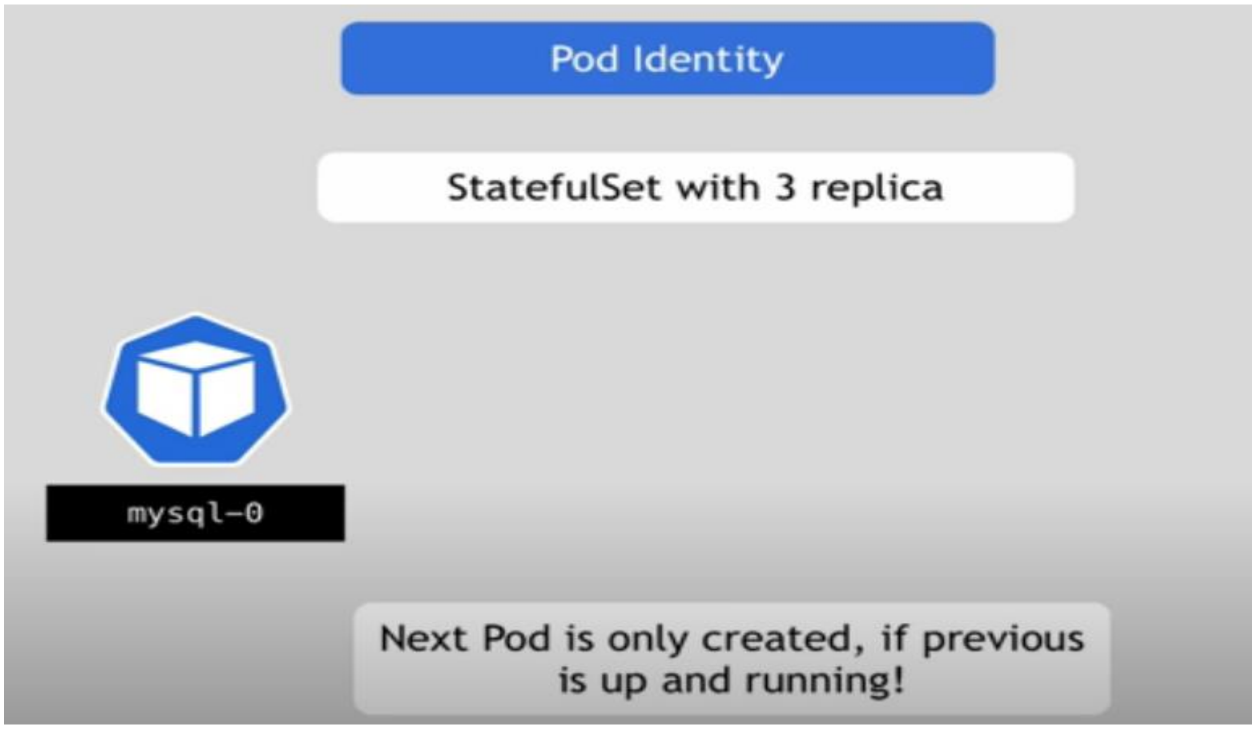



Container
Persistent Volume


Pod state
pv
data A
Pod state
Remote storage!
to be available for other Nodes


Pod Identity
Deployment:
random hash
StatefulSet:
fixed ordered names
mysql-c9e93d4e783165d
mysql-0
$(statefulset name)-$(ordinal)

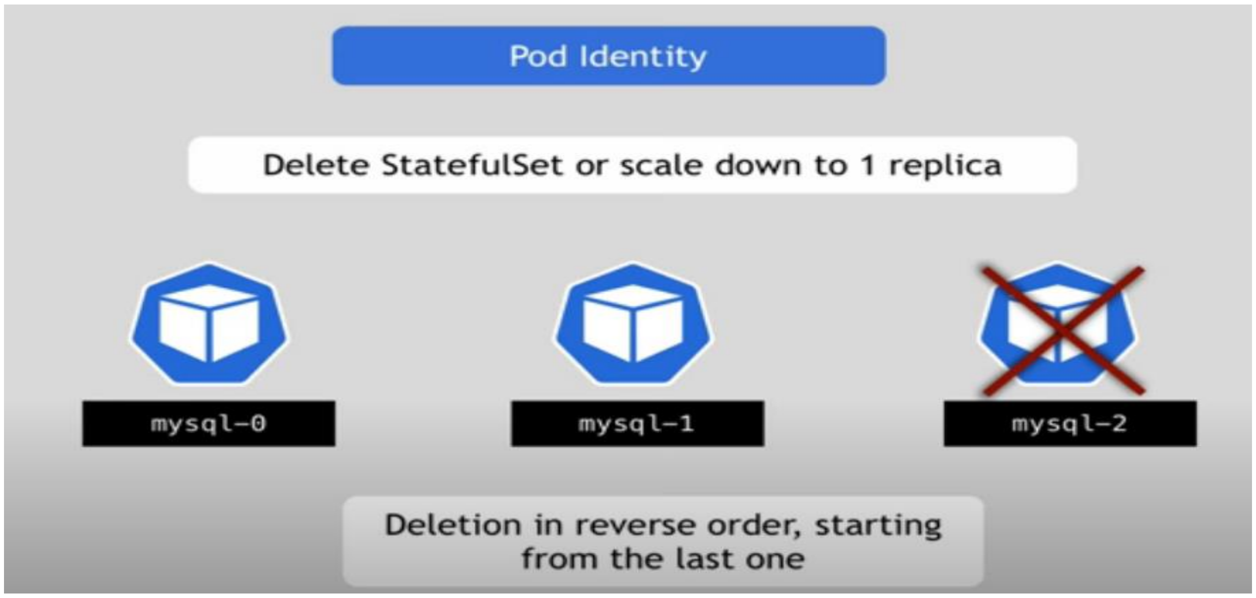

Pod Identity
StatefulSet with 3 replica
mysql-0
mysql-1
mysql-2
MASTER
SLAVE
SLAVE



Pod Identity

StatefulSet with 3 replica

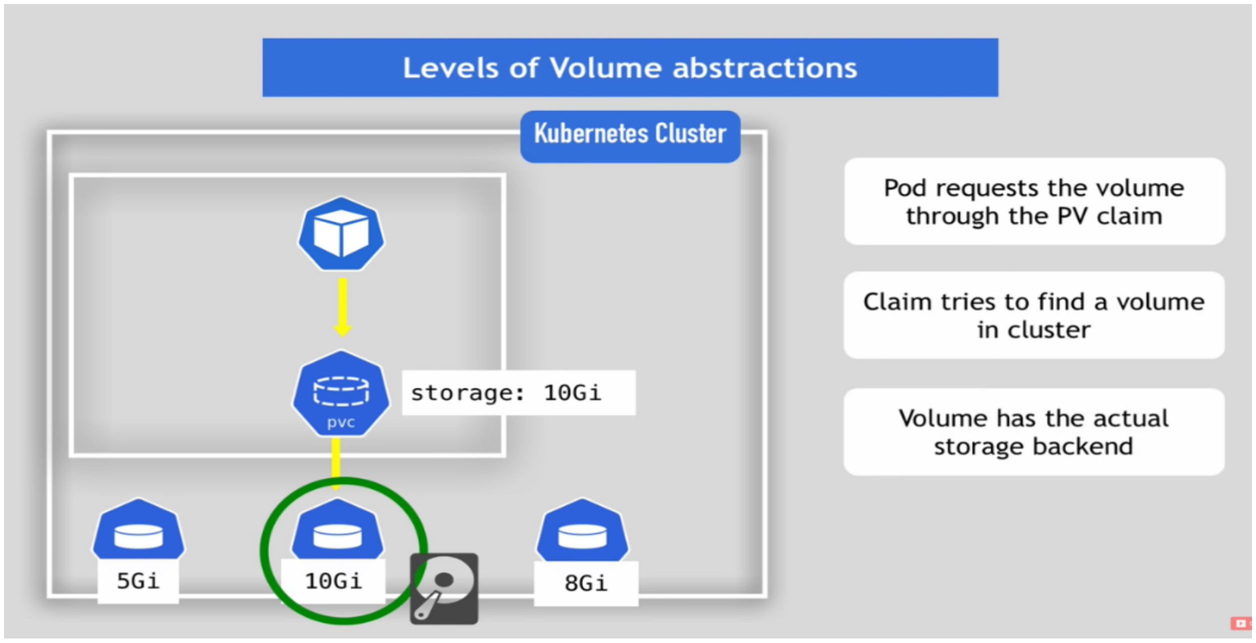
Next Pod is only created, if previous is up and running!

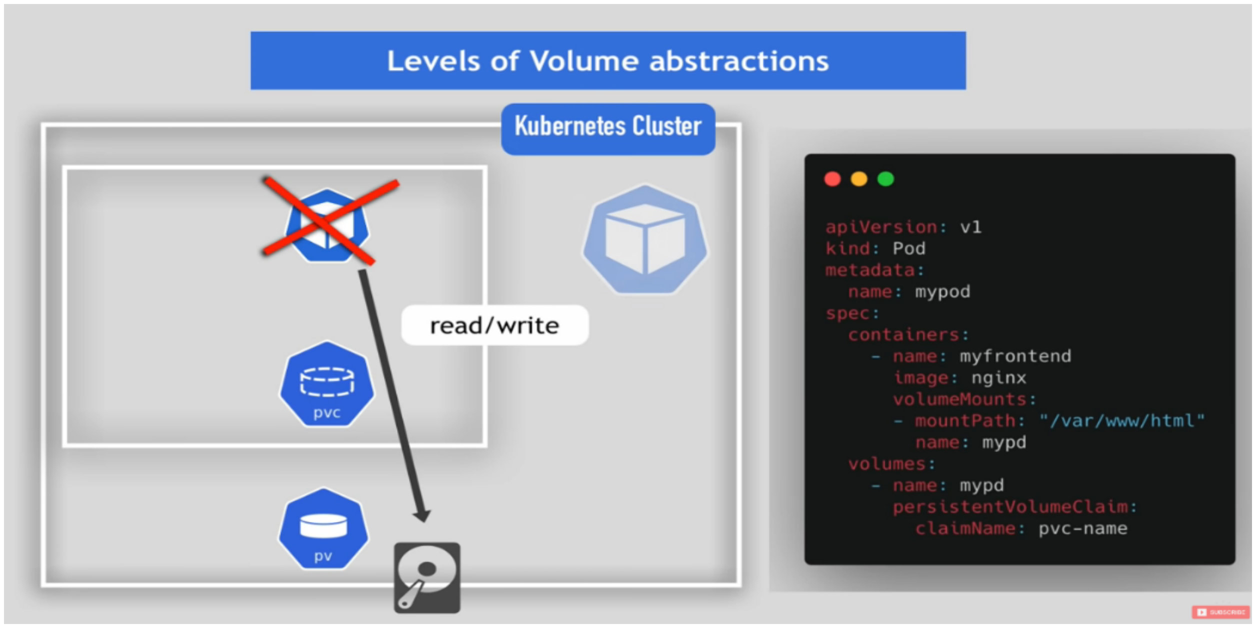


Pod Identity

Delete StatefulSet or scale down to 1 replica

Deletion in reverse order, starting from the last one


Levels of Volume abstractions
Kubernetes Cluster
storage: 10Gi
pvc
5Gi
10Gi
8Gi
Pod requests the volume through the PV claim
Claim tries to find a volume in cluster
Volume has the actual storage backend

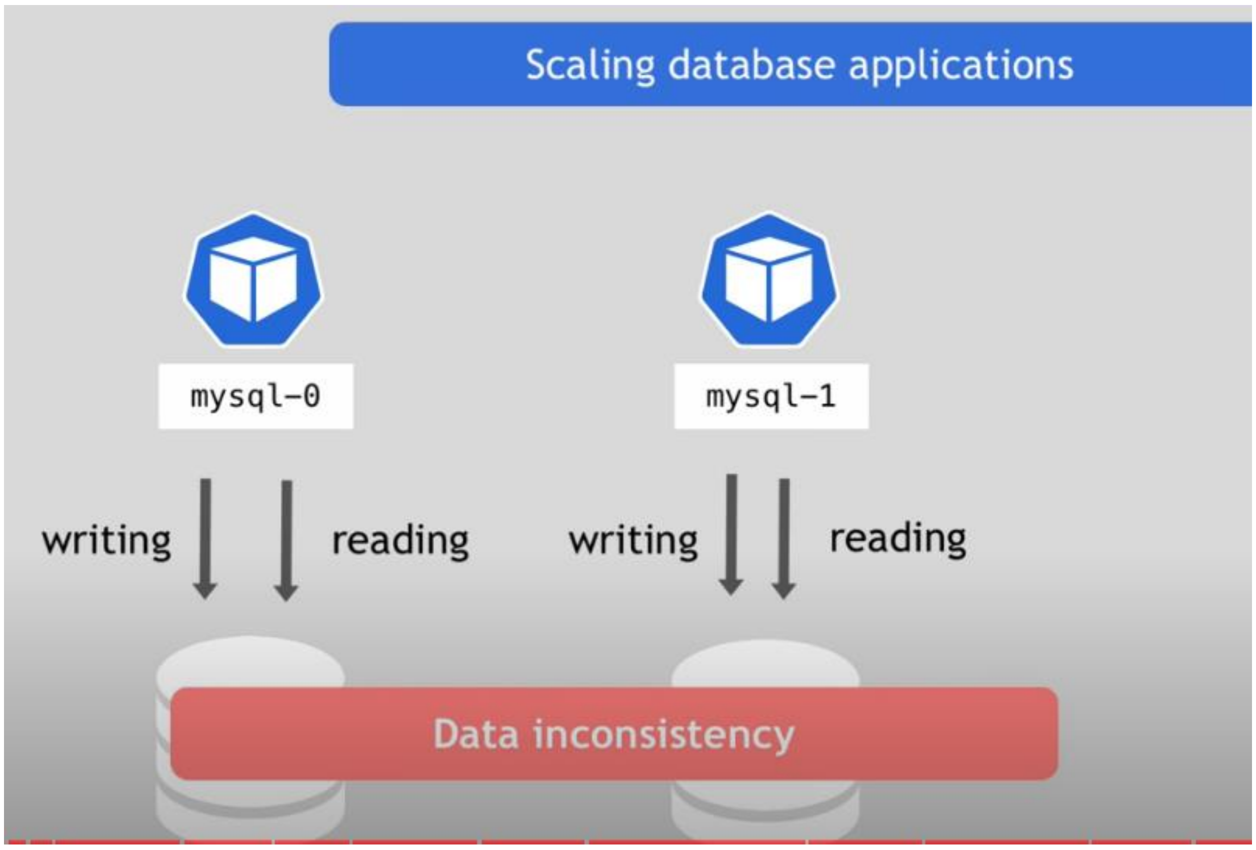


```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-name
```

Scaling a StatefulSet

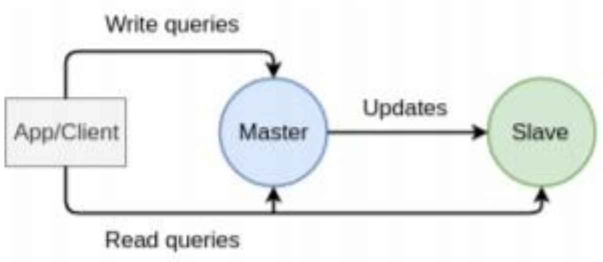
- When we scale down a stateful application, one of the instances terminates and its data should be redistributed to the remaining pods.
- If you do not reassign the data, it will remain inaccessible until it is extended again.
- When the controller detects a scaled down statefulset, it creates a new drain container based on the specified template and ensures that it is bound to the persistentvolumeclaim, which was previously bound to the stateful container deleted due to scaling down

- The drain container obtains the same identity (that is, name and host name) as the deleted stateful container.
- There are two reasons for this:
- Some stateful applications require stable identities – which may also apply during data redistribution.
- If statefulset is expanded again while executing the drain procedure, this prevents the statefulset controller from creating duplicate containers and attaching them to the same PVC.
- If the drain pod or its host node crashes, the drain pod will be rescheduled to another node where its operation can be retried / resumed. After drain pod is completed, pod and PVC will be deleted. When the statefulset is backed up, a new PVC is created.

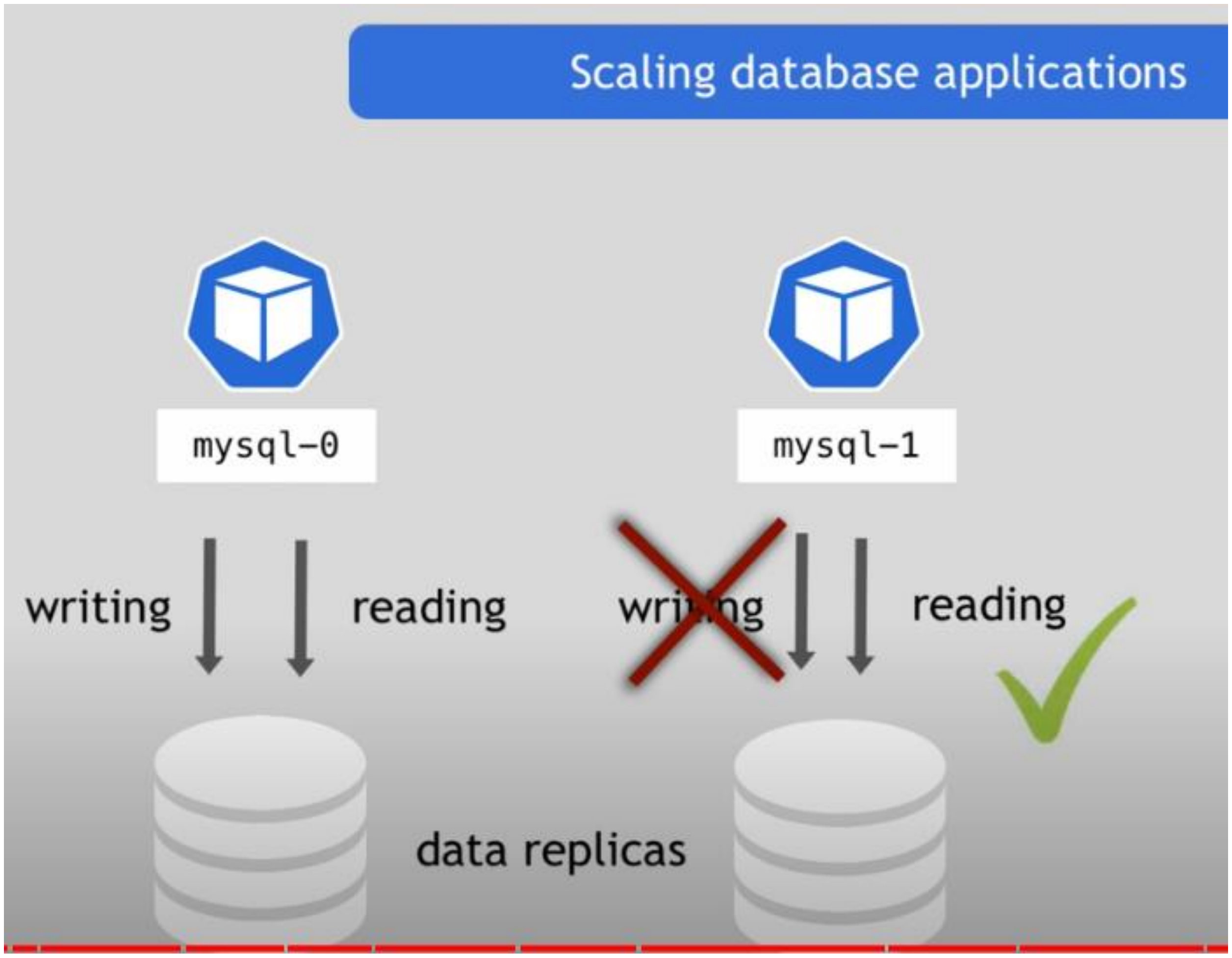


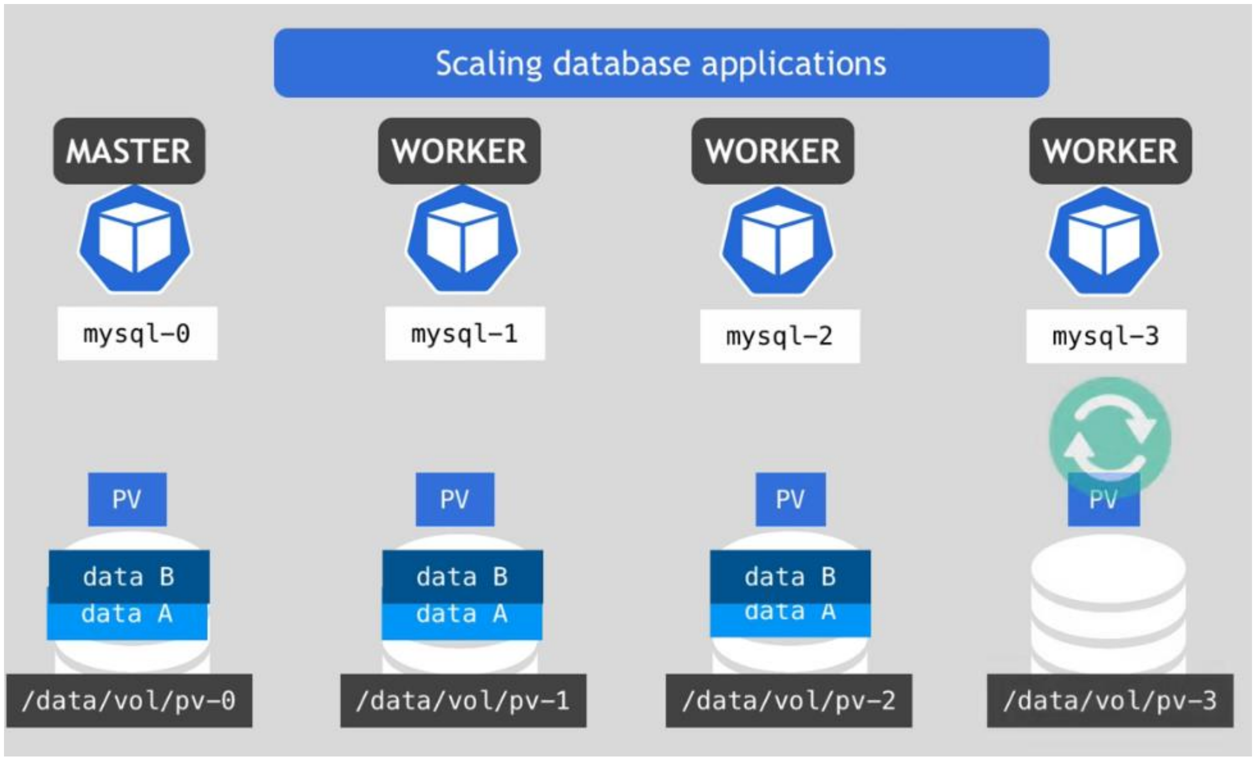
Stable Network Identities for a StatefulSet

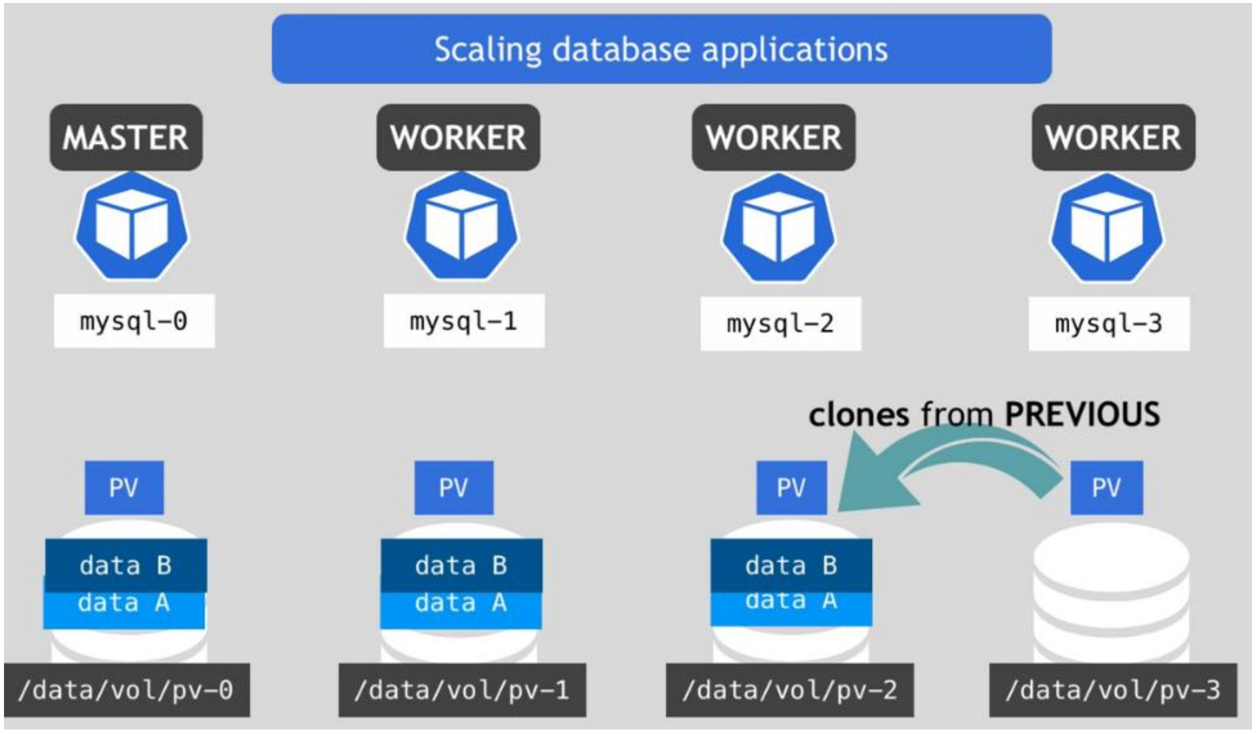
- There is a need for a stable network ID so that the special roles typically observed in stateful apps can be preserved.
- E.g., in a monolithic RDBMS such as PostgreSQL or MySQL deployment, there is usually 1 master and multiple read-only slaves.
- Each slave communicates with the master letting it know the state to which it has synced the master's data. So, the slaves know that "instance-0" is the master and the master knows that "instance-1" is a slave.

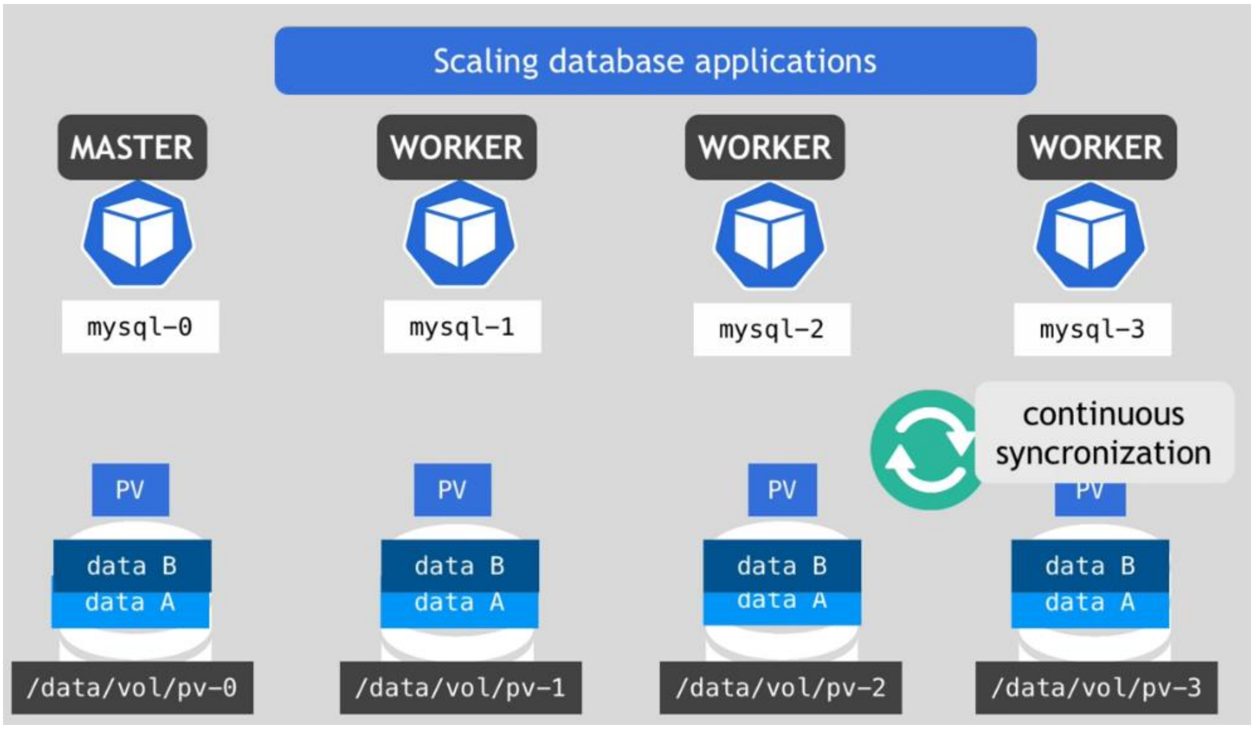


A simple illustration of a master-slave replication setup. Write queries are directed to the master and the read queries can be handles by slaves as well.

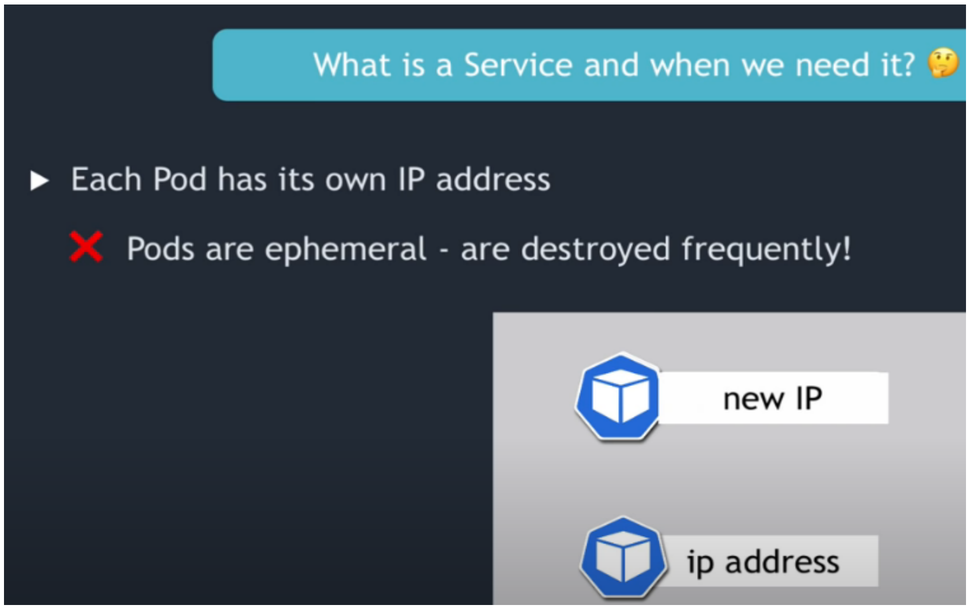



Scaling database applications
MASTER
WORKER
WORKER
WORKER
mysql-0
mysql-1
mysql-2
mysql-3
PV
PV
PV
PV
data B
data A
data B
data A
data B
data A
/data/vol/pv-0
/data/vol/pv-1
/data/vol/pv-2
/data/vol/pv-3


Scaling database applications
MASTER
WORKER
WORKER
WORKER
mysql-0
mysql-1
mysql-2
mysql-3
clones from PREVIOUS
PV
PV
PV
PV
data B
data A
data B
data A
data B
data A
/data/vol/pv-0
/data/vol/pv-1
/data/vol/pv-2
/data/vol/pv-3



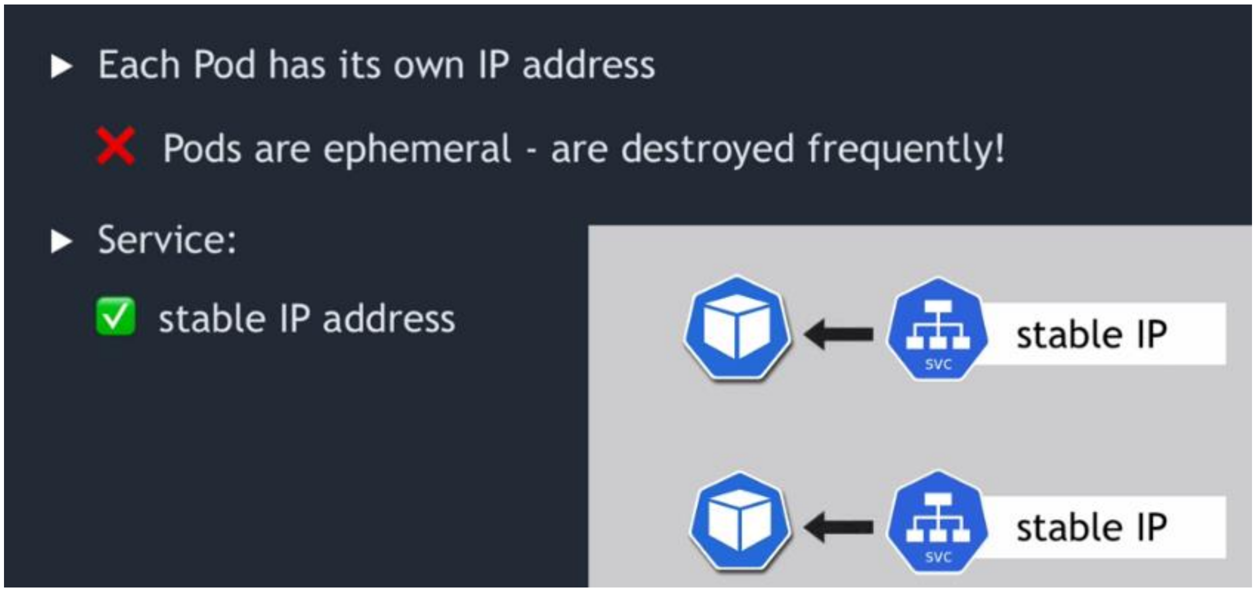
Services:
Collection of pods exposed as an endpoint



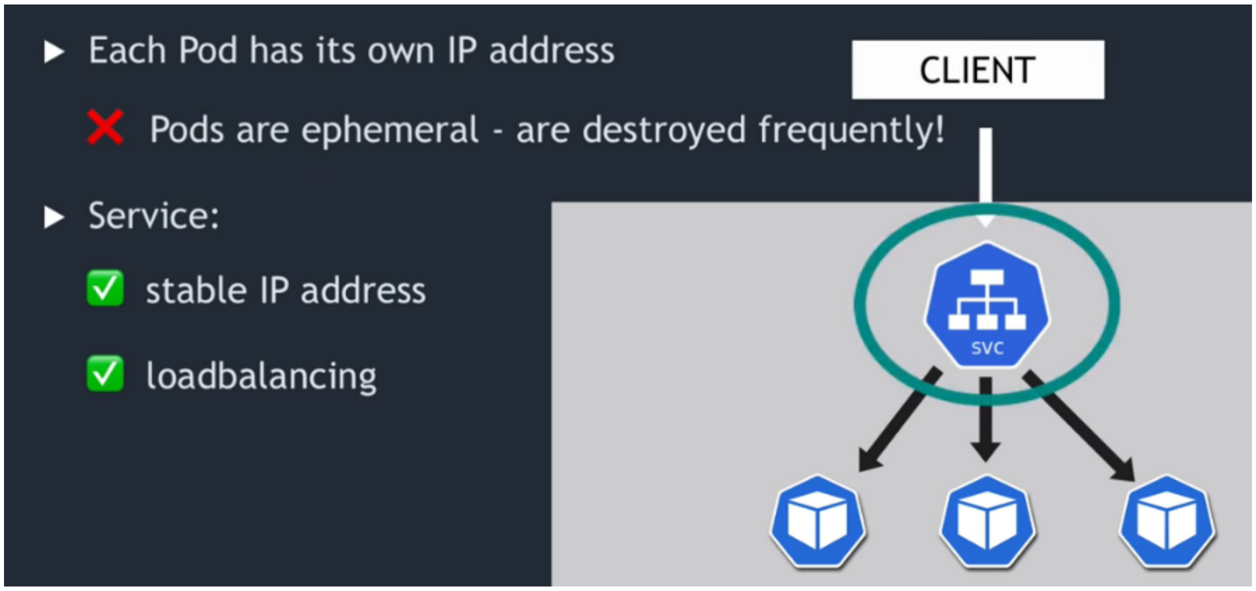
What is a Service and when we need it? 🤔

- ▶ Each Pod has its own IP address

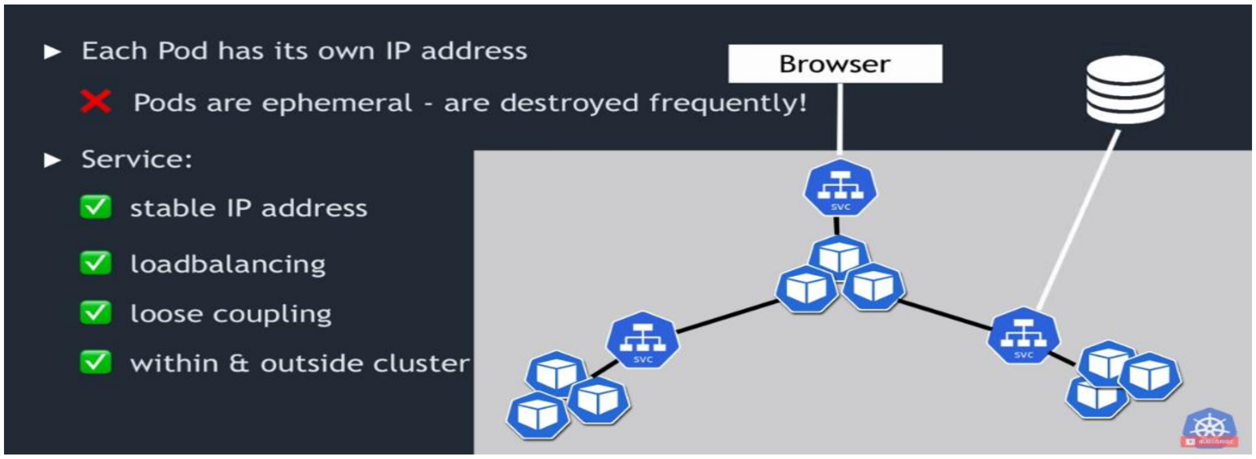
❌ Pods are ephemeral - are destroyed frequently!



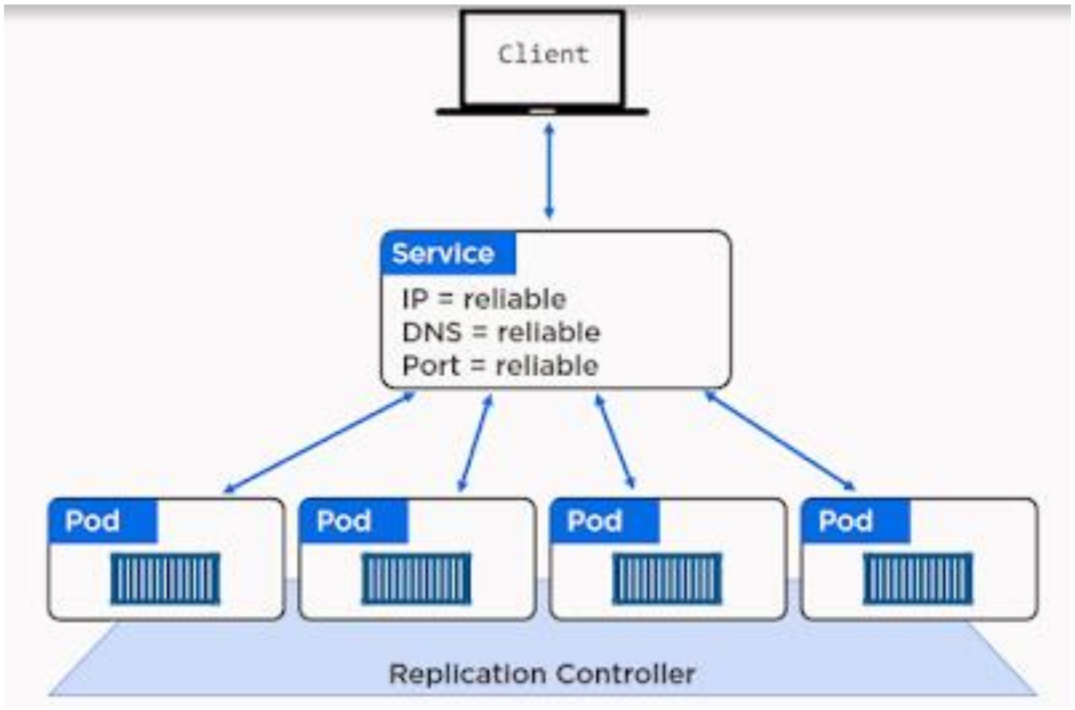
- ▶ Each Pod has its own IP address
  - ❌ Pods are ephemeral - are destroyed frequently!
- ▶ Service:
  - ✅ stable IP address

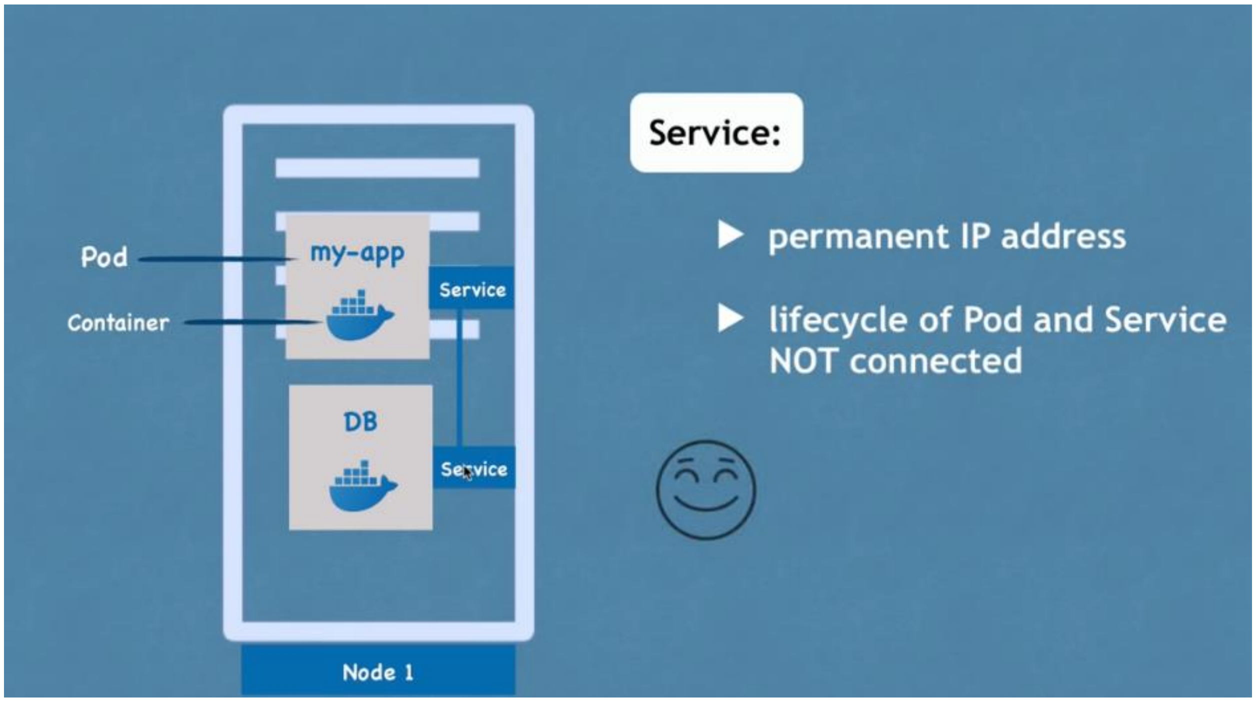


- ▶ Each Pod has its own IP address
  - ❌ Pods are ephemeral - are destroyed frequently!
- ▶ Service:
  - ✅ stable IP address
  - ✅ loadbalancing



- ▶ Each Pod has its own IP address
  - ❌ Pods are ephemeral - are destroyed frequently!
- ▶ Service:
  - ✅ stable IP address
  - ✅ loadbalancing
  - ✅ loose coupling
  - ✅ within & outside cluster

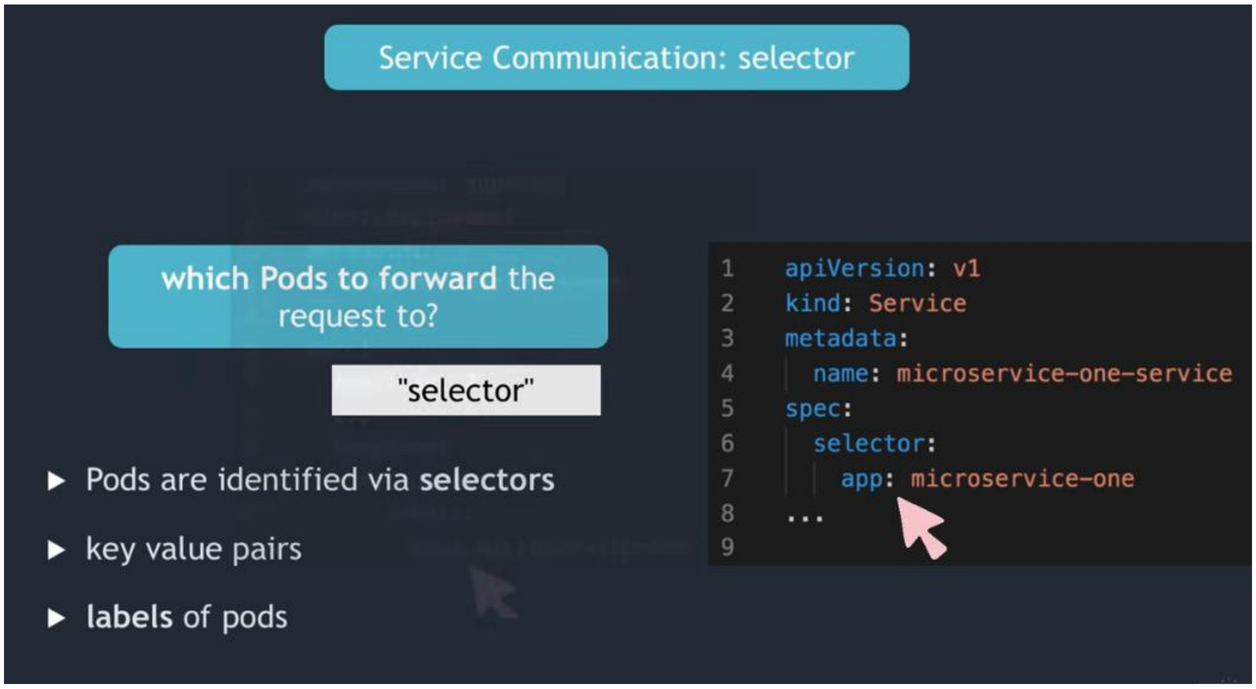




Without a service

- Running a pod or deployment without a service is very possible, and in some cases it will be perfectly fine.
- If your workloads do not require communication with other resources either within or outside of the cluster there is no need to use a service.
- However, for anything that will need to communicate with other resources, a service should be strongly considered.

- Without a service, Pods are assigned an IP address which allows access from within the cluster.
- Other pods within the cluster can hit that IP address and communication happens as normal.
- However, if that pod dies, a new pod will be created that comes online with a new IP address and anything trying to communicate with the dead pod somehow needs to know about this new address.
- While there are ways of circumventing this issue without using services they require a lot of manual configuration and management which will only cause more problems as the number of pods increases.
- Keep in mind services can also be used to abstract things like database connections, which can make them invaluable when trying to figure out how to layout the networking within your kubernetes cluster.



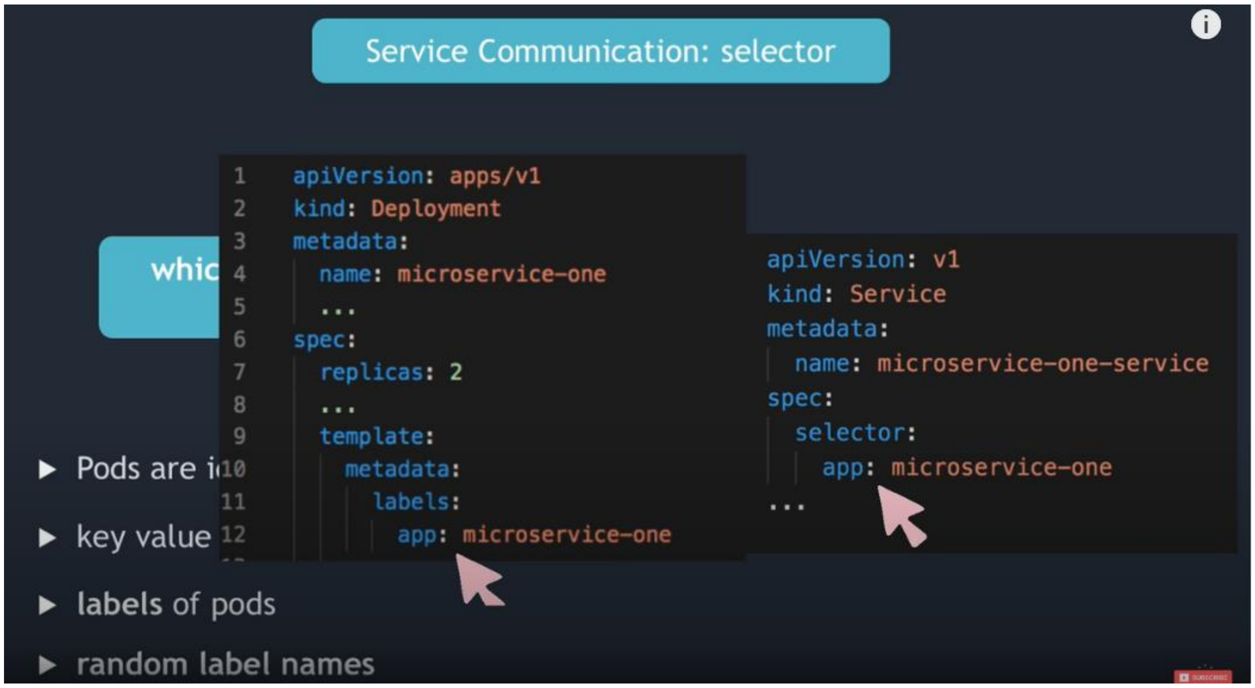
Service Communication: selector

which Pods to forward the request to?

"selector"

- Pods are identified via **selectors**
- key value pairs
- **labels** of pods

```yaml
apiVersion: v1
kind: Service
metadata:
  name: microservice-one-service
spec:
  selector:
    app: microservice-one
...
```



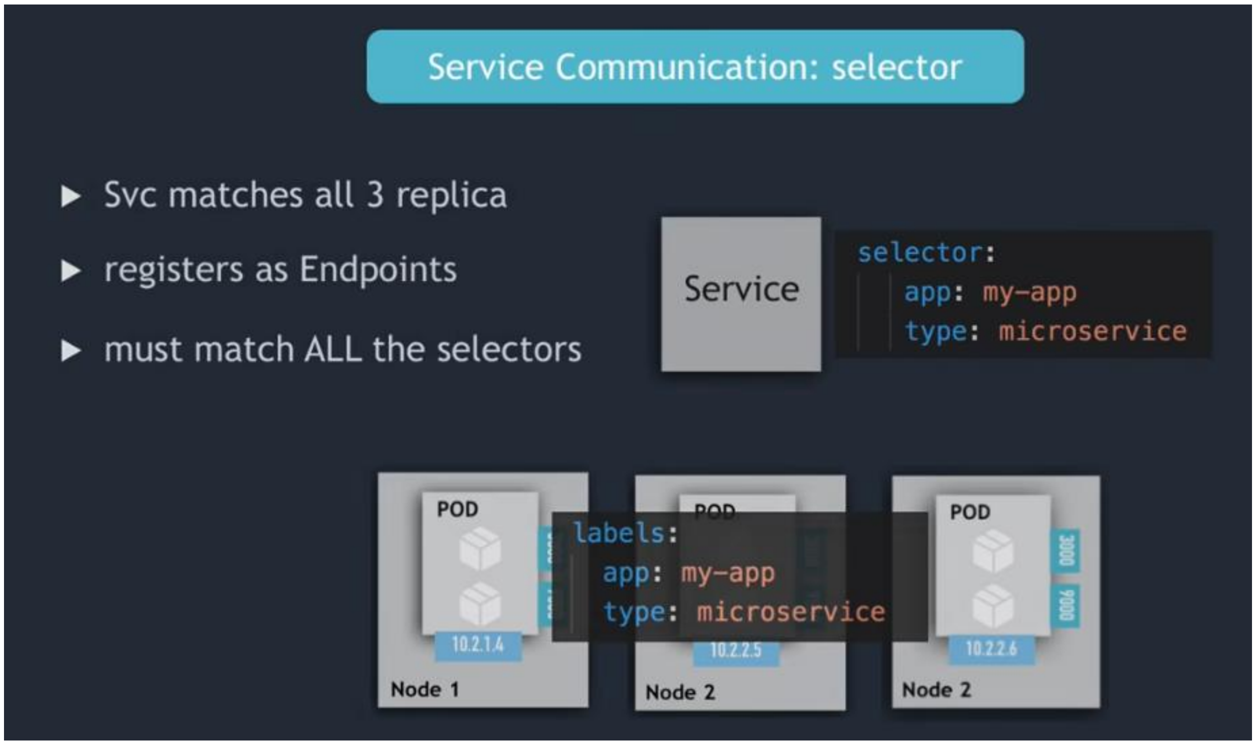
Service Communication: selector

whic

- Pods are i
- key value
- labels of pods
- random label names

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-one
  ...
spec:
  replicas: 2
  ...
  template:
    metadata:
      labels:
        app: microservice-one
```

```
apiVersion: v1
kind: Service
metadata:
  name: microservice-one-service
spec:
  selector:
    app: microservice-one
...
```

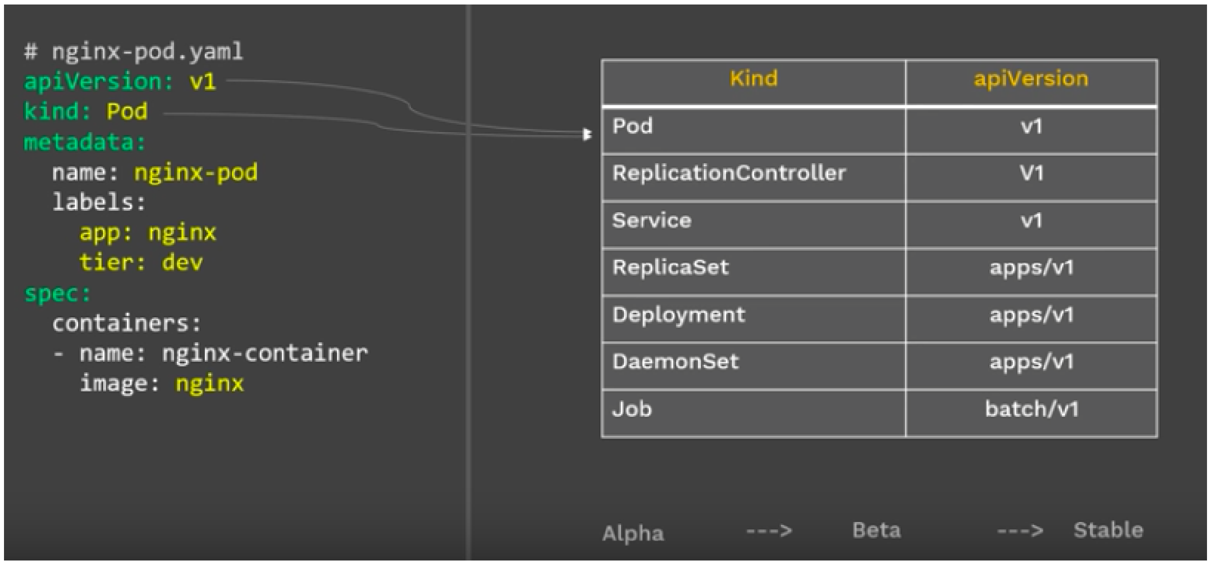


Service Communication: selector

- Svc matches all 3 replica
- registers as Endpoints
- must match ALL the selectors

```
selector:
  app: my-app
  type: microservice
```

```
labels:
  app: my-app
  type: microservice
```



```
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
    tier: dev
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Kind	apiVersion
Pod	v1
ReplicationController	V1
Service	v1
ReplicaSet	apps/v1
Deployment	apps/v1
DaemonSet	apps/v1
Job	batch/v1

Alpha ---> Beta ---> Stable

Types of Kubernetes Health check:

- **Readiness=when Pod is ready to accept the traffic. (health)**
- **Liveness=when to Restart the pod (ping)**

Replication Controller (rc)

RC is a controller that is part of the Master Node's Controller Manager.

It makes sure the specified number of replicas for a Pod is running at any given point in time.

Available in master

ReplicaSet

A **ReplicaSet** (rs) is the next-generation ReplicationController.

Replica Sets support both equality- and set-based Selectors, whereas Replication Controllers only support equality-based Selectors.

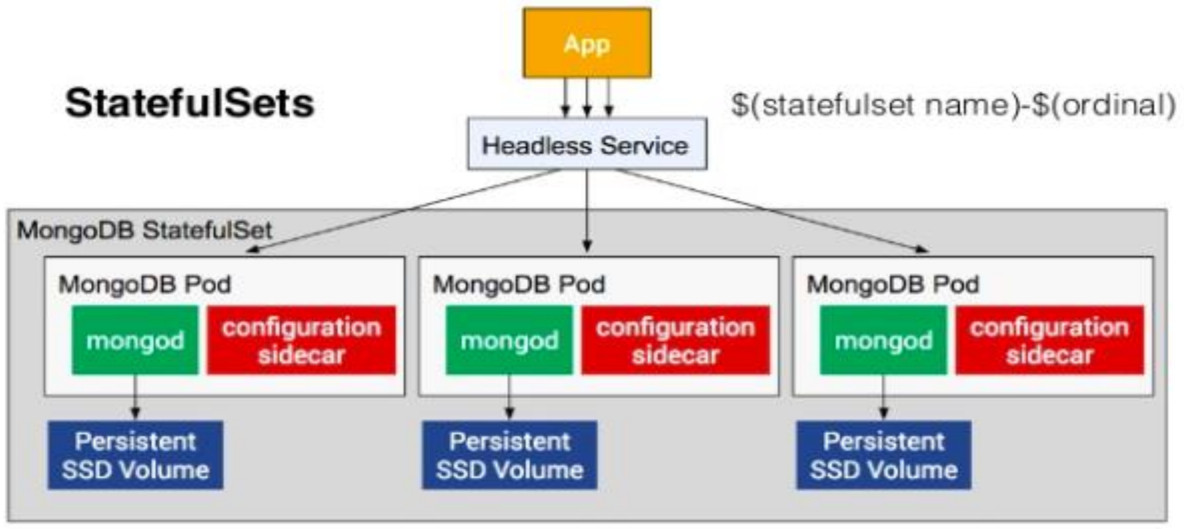
Available in deployment

DaemonSet

DaemonSets manage groups of replicated Pods

- DaemonSets works on one-Pod-per-node model

Headless service



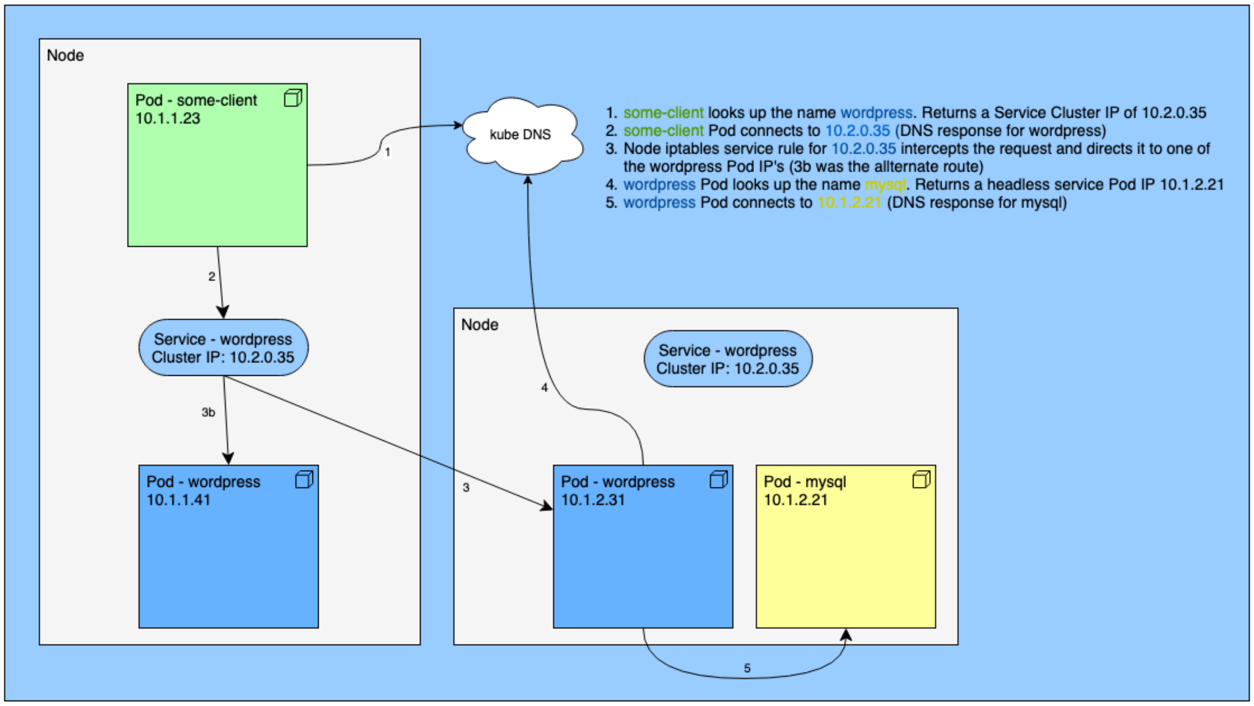
- A Headless Service is used, when a Pod or a client wants to communicate directly with another specific Pod.
- We can create **headless service** which is **a service that does not get a ClusterIP** and is created when we specify a Service with **ClusterIP** set to **None**.
- This is often used when a deployer wants to decouple their service from Kubernetes and **use an alternative method of service discovery and load balancing**.
- For headless Services, a cluster IP is not allocated, kube-proxy does not handle these Services, and there is no load balancing or proxying done by the platform for them. How DNS is automatically configured depends on whether the Service has selectors defined:

With selectors

- For headless Services that define selectors, the endpoints controller creates Endpoints records in the API, and modifies the DNS configuration to return records (addresses) that point directly to the Pods backing the Service.

Without selectors

- For headless Services that do not define selectors, the endpoints controller does not create Endpoints records. However, the DNS system looks for and configures either:
- CNAME records for ExternalName-type Services.
- A records for any Endpoints that share a name with the Service, for all other types



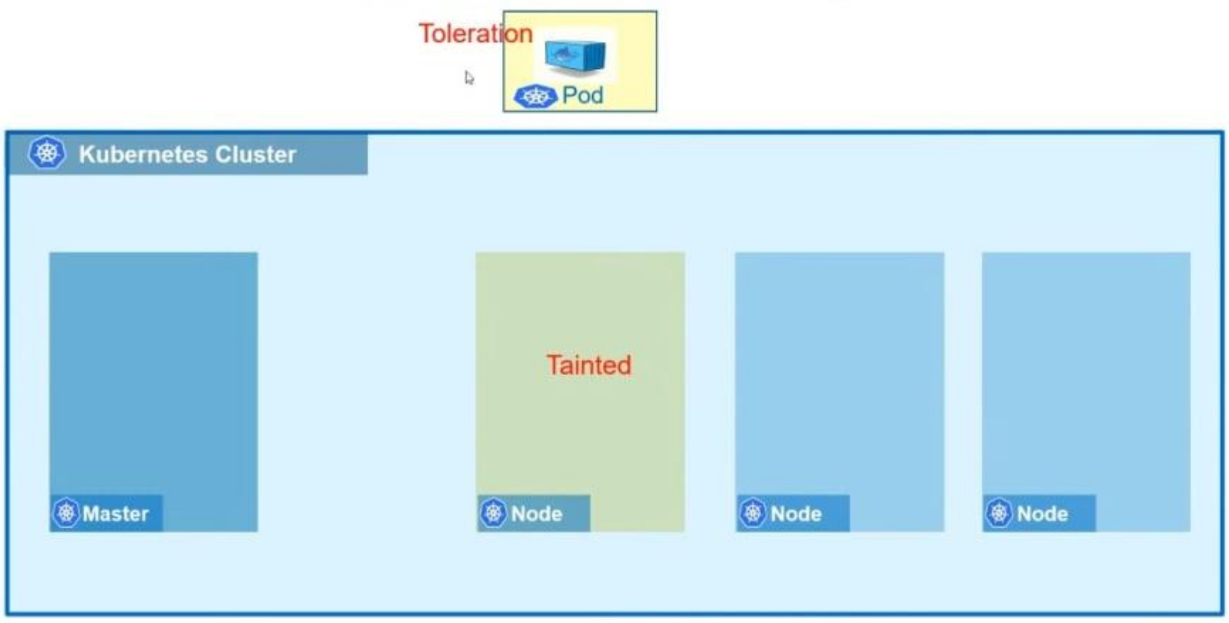
1. some-client looks up the name wordpress. Returns a Service Cluster IP of 10.2.0.35
2. some-client Pod connects to 10.2.0.35 (DNS response for wordpress)
3. Node iptables service rule for 10.2.0.35 intercepts the request and directs it to one of the wordpress Pod IP's (3b was the allternate route)
4. wordpress Pod looks up the name mysql. Returns a headless service Pod IP 10.1.2.21
5. wordpress Pod connects to 10.1.2.21 (DNS response for mysql)

$ kubectl expose deployment **wordpress-mysql** deployment **--cluster-ip=None** --name= wordpress-mysql/ wordpress-mysql - exposed

Multi-Port Service

You can configure multiple ports on a Service. This would be a Multi-Port Service.

Taints and tolerations



Taint Effects

- Taints and Tolerations are used to set restrictions on what pods can be shared on that node.

- When the PODS are created, kubernetes scheduler tries to place these pods on the available worker nodes.As of now, there are no restrictions or limitations and so scheduler places the PODs across all of the nodes to balance them out equally.
- By default, PODs won't have any tolerations i.e., unless specified otherwise none of the PODs can tolerate any taint.

Each taint has one of the following effects:

NoSchedule - this means that no pod will be able to schedule onto node unless it has a matching toleration.

PreferNoSchedule - this is a “preference” or “soft” version of NoSchedule – the system will try to avoid placing a pod that does not tolerate the taint on the node, but it is not required.

NoExecute - the pod will be evicted from the node (if it is already running on the node) and will not be scheduled onto the node (if it is not yet running on the node).

How to Taint the Node

Imagine, we have a node named `node1`, and we want to add a tainting effect to it:

```
$ kubectl taint nodes node1  key:=NoSchedule
```

```
$ kubectl describe node node1 | grep -i taint
```

```
Taints:            node-role.kubernetes.io/master:NoSchedule
```

How to Remove Taint from the Node

To remove the taint from the node run:

```
$ kubectl taint nodes  key:NoSchedule
```

```
node "node1" untainted
```

```
$ kubectl describe node node1 | grep -i taint
```

```
Taints:            <none>
```

Tolerations

In order to schedule to the **“tainted” node** pod should have some special tolerations, let’s take a look on system pods in kubeadm, for example, etcd pod:

```
$ kubectl describe pod etcd-node1 -n kube-system | grep  -i toleration
```

```
Tolerations:       :NoExecute
```

As you can see it has toleration to `:NoExecute` taint, let’s see where this pod has been deployed:

```
$ kubectl get pod etcd-node1 -n kube-system -o wide
```

```
NAME                                    READY     STATUS    RESTARTS   AGE      IP           NODE
```

```
etcd-node1                          1/1     Running   0          22h     192.168.1.212   node1
```

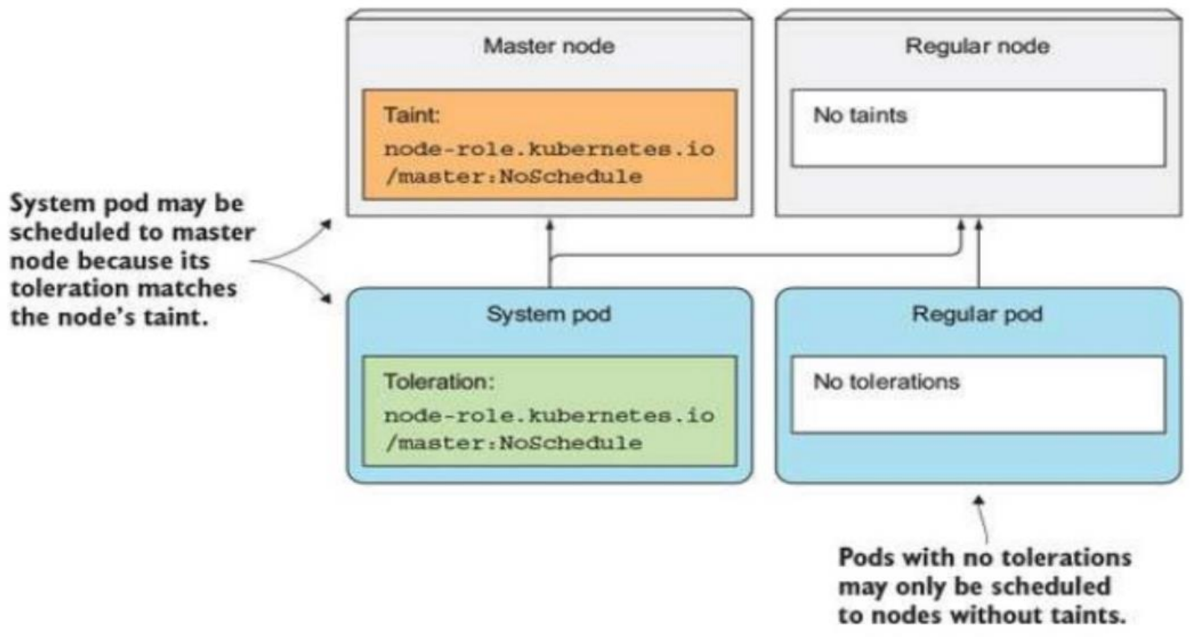
The pod was indeed deployed on the “tainted” node because it “tolerates” `NoSchedule` effect.

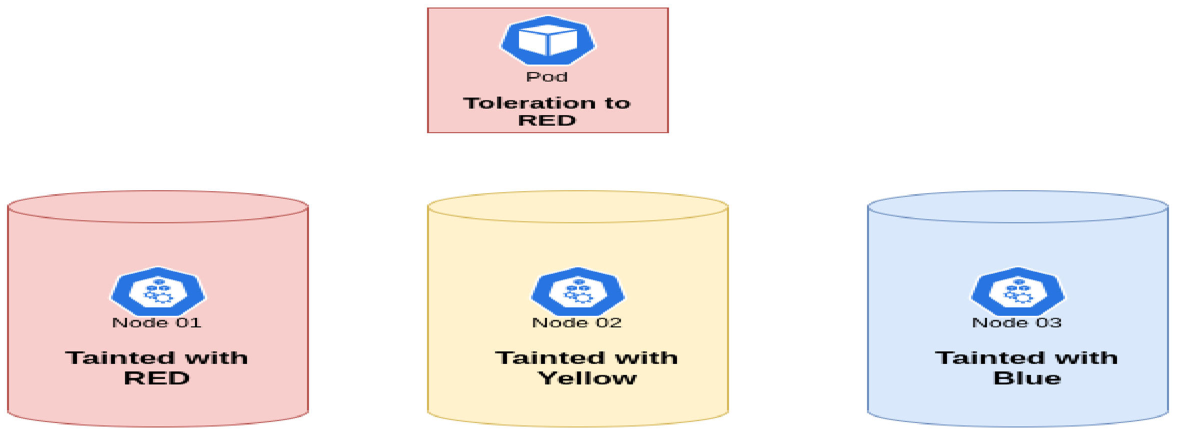
Now let’s have some practice: let’s create our own taints and try to deploy the pods on the “tainted” nodes with and without tolerations.

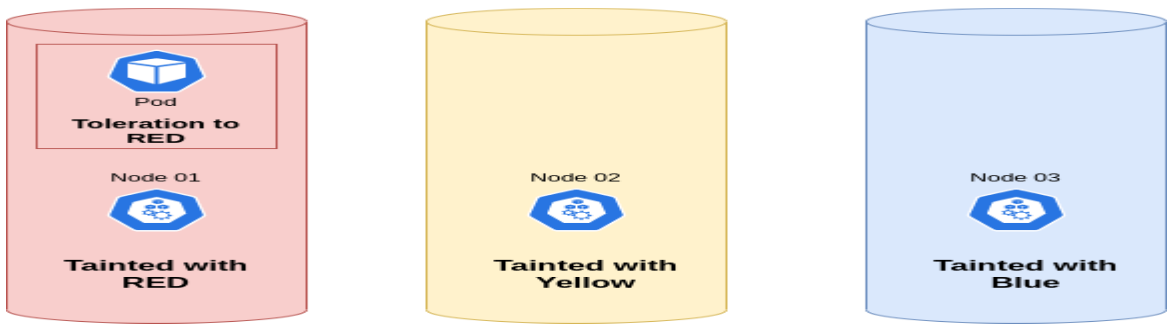
real use cases of Taints and Tolerations

- When we need Dedicated Nodes
- Nodes with Special Hardware

Taint and toleration







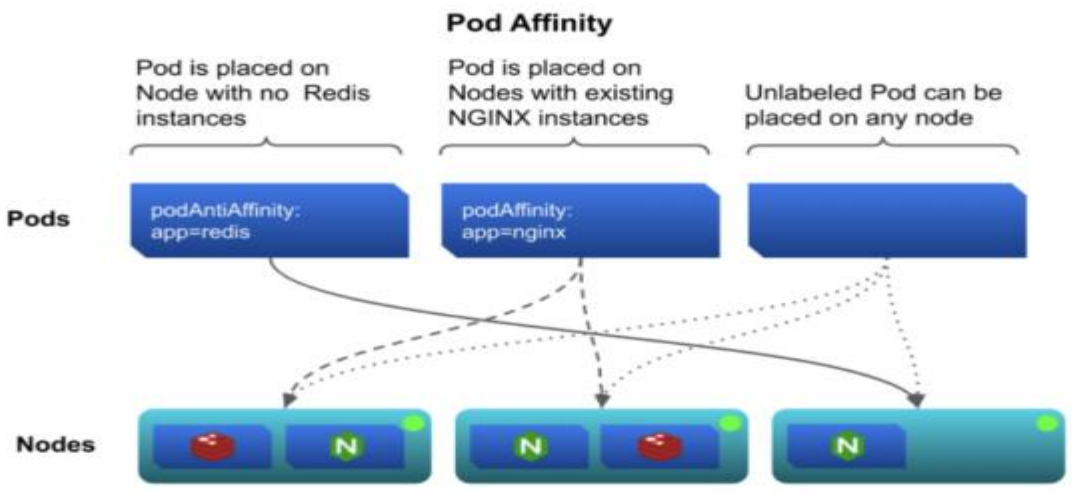
NODE AFFINITY

Since node affinity identifies the nodes on which to place pods via labels, we first need to add a label to our node.

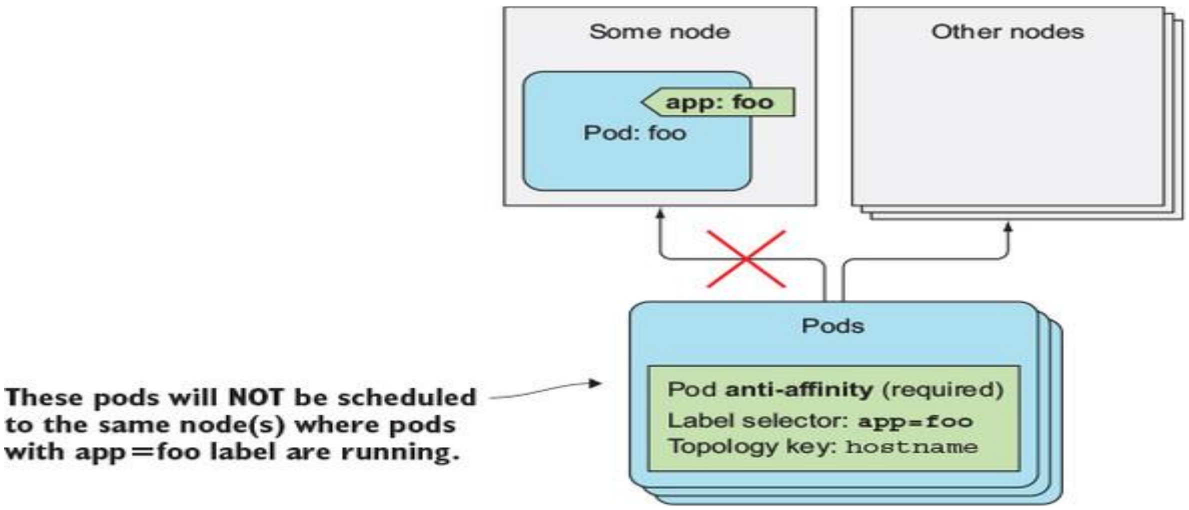
```
$ kubectl edit node node2

labels:

   ...

   test-node-affinity: test

...
```

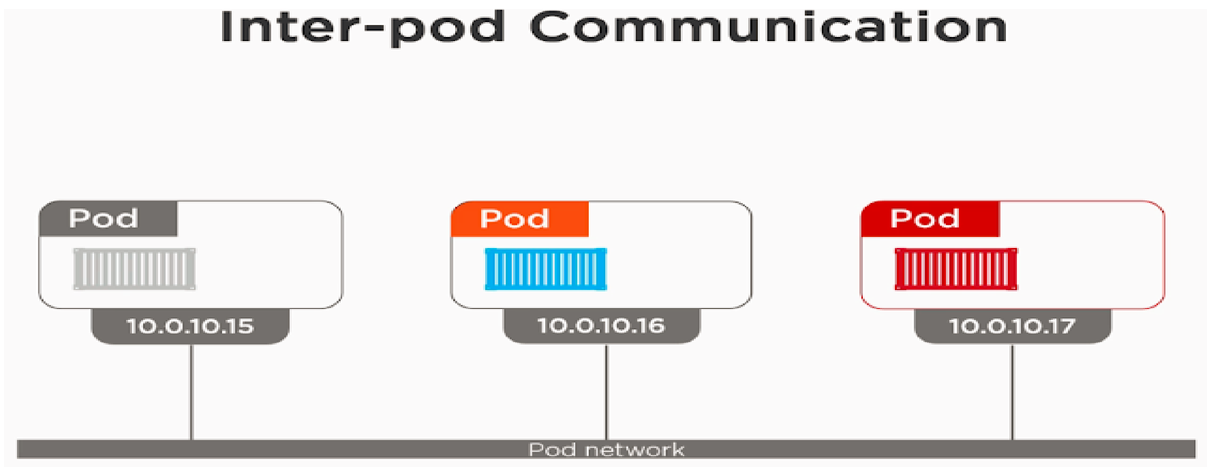
Pod affinity and anti-affinity

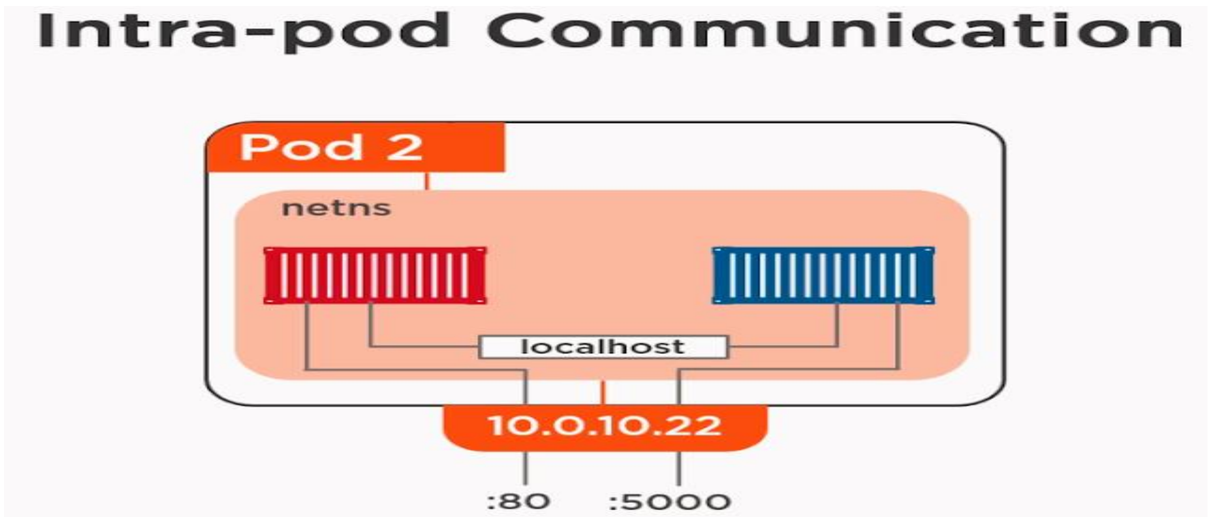
- Pod affinity allows placing pods to same node as other pods if the label selector on the new pod matches the label on the current pod
- Pod anti-affinity releases/prevent placing pods to same node as other pods if the label selector on the new pod matches the label on the current pod


Pod Affinity
Pod is placed on Node with no Redis instances
Pod is placed on Nodes with existing NGINX instances
Unlabeled Pod can be placed on any node
Pods
podAntiAffinity: app=redis
podAffinity: app=nginx
Nodes
N
N
N

POD anti-affinity

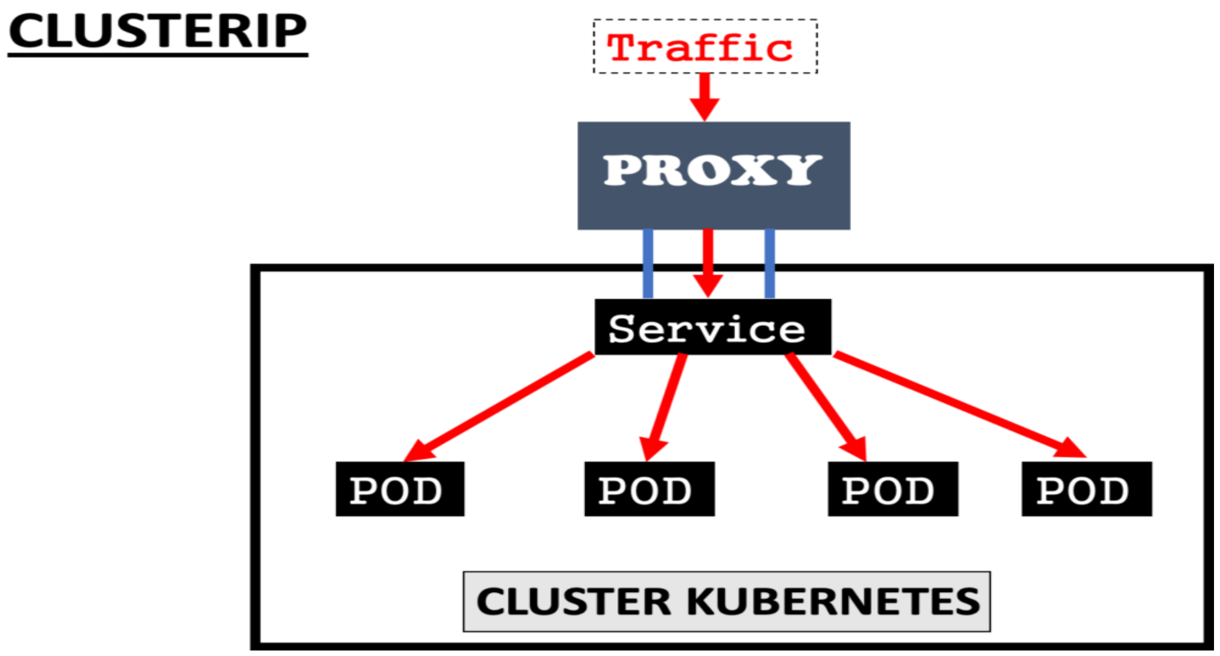

Some node
app: foo
Pod: foo
Other nodes
Pods
Pod anti-affinity (required)
Label selector: app=foo
Topology key: hostname
These pods will NOT be scheduled to the same node(s) where pods with app=foo label are running.





The followings are the different types of services being provided by the Kubernetes:

ClusterIP:



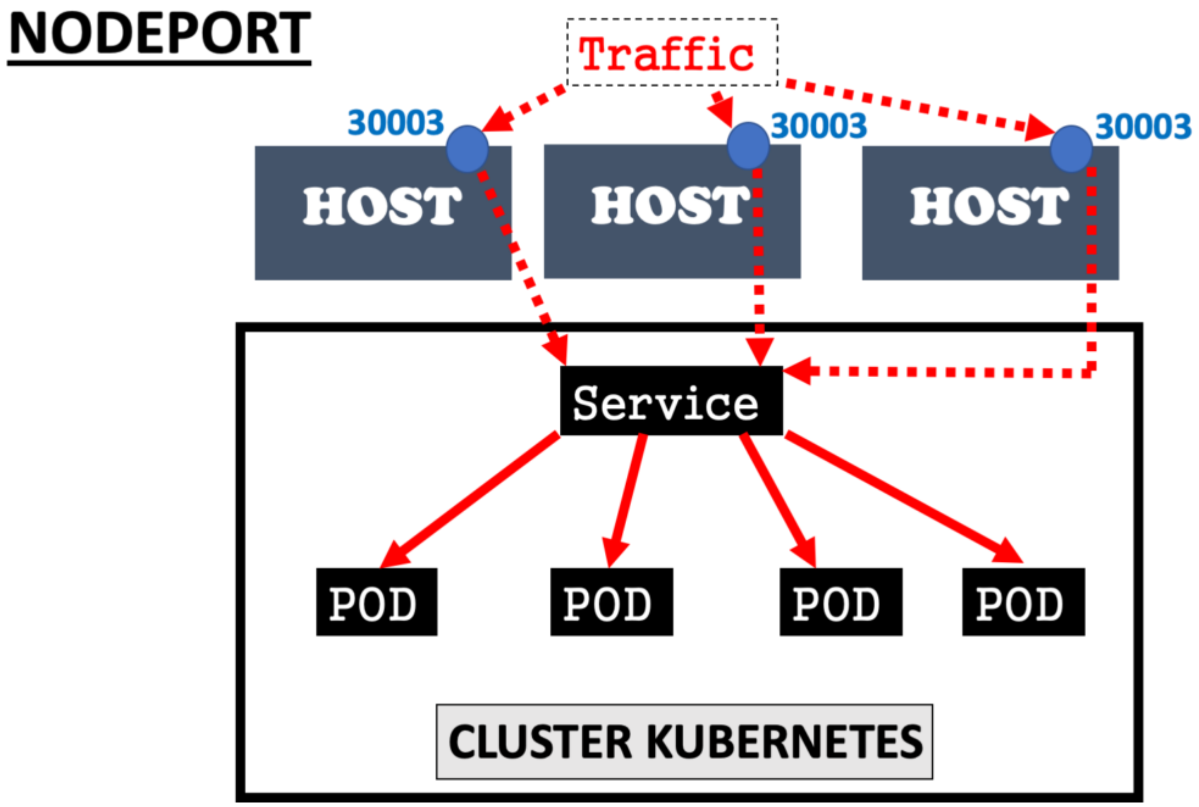
- This is the default service type which exposes the service on a cluster-internal IP by making the service only reachable within the cluster.
- A ClusterIP service is the **default Kubernetes service**. It gives you a service **inside** your cluster that other apps inside your cluster can access.
- There is no external access.

When would you use this?

- There are a few scenarios where you would use the Kubernetes proxy to access your services.
- Debugging your services, or connecting to them directly from your laptop for some reason
- Allowing internal traffic, displaying internal dashboards, etc.
- Because this method requires you to run kubectl as an authenticated user, you should NOT use this to expose your service to the internet or use it for production services.

NodePort

- This exposes the service on each Node's IP at a static port. Since a **ClusterIP** service, to which the NodePort service will route, is automatically created. We can contact the NodePort service outside the cluster.



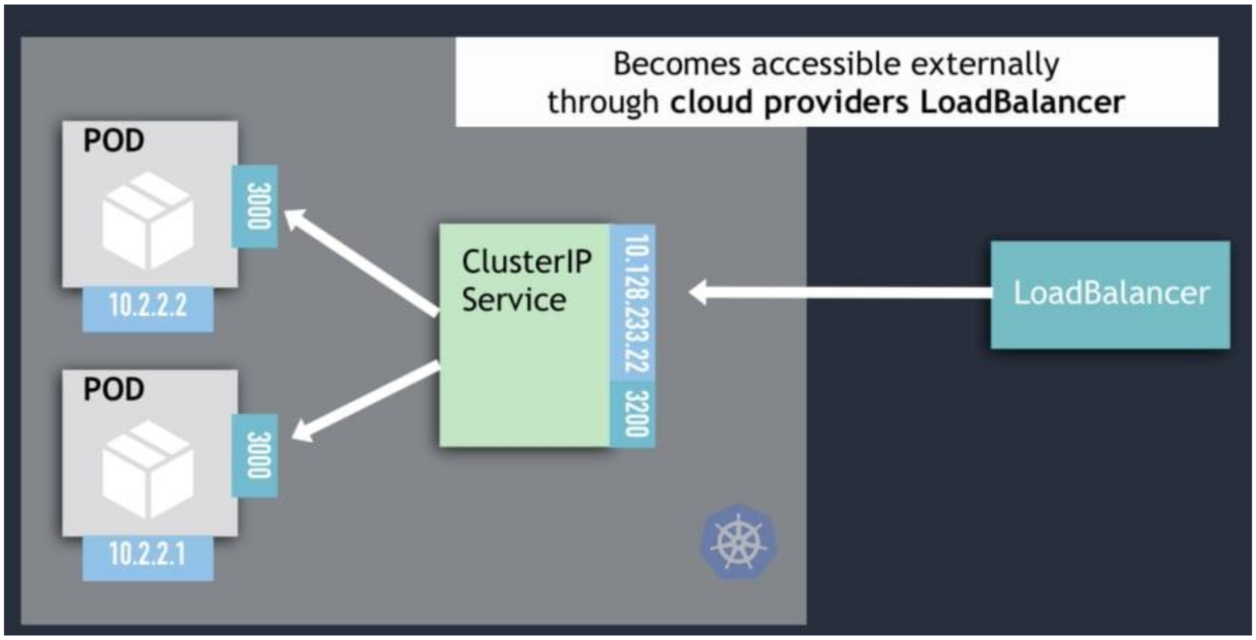
- A NodePort service is the most primitive way to get external traffic directly to your service.
- NodePort, as the name implies, opens a specific port on all the Nodes (the VMs), and any traffic that is sent to this port is forwarded to the service

When would you use this?

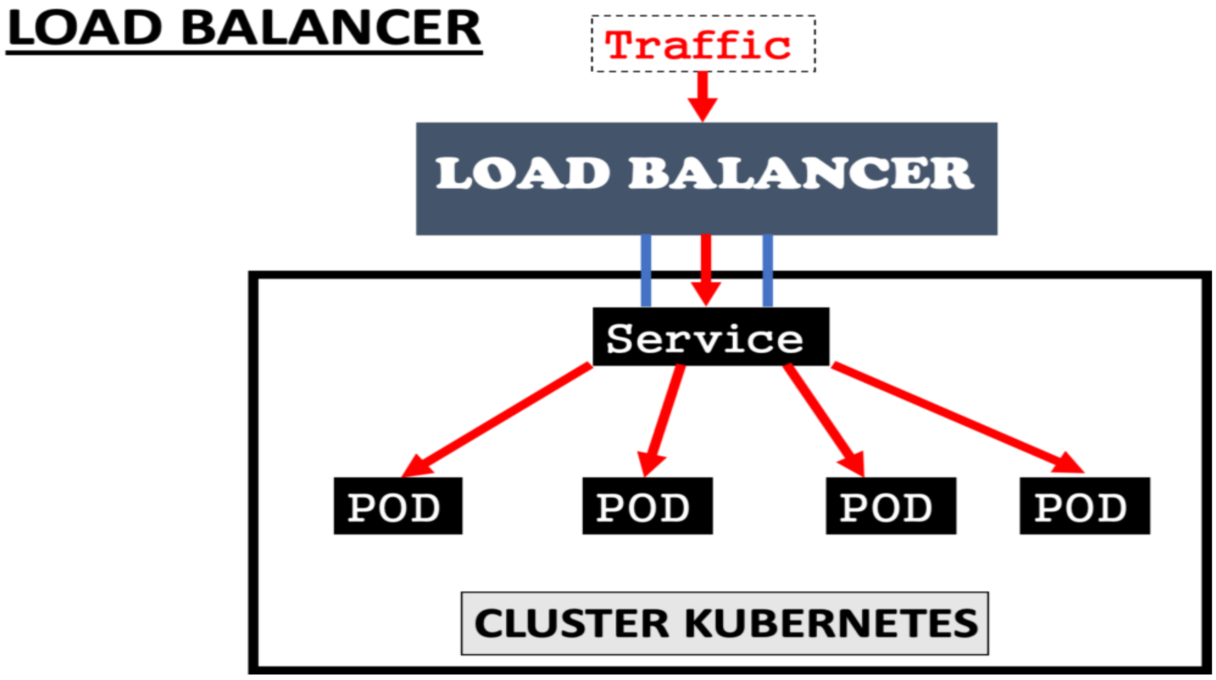
- There are many downsides to this method:
- You can only have once service per port
- You can only use ports 30,000–32,767
- If your Node/VM IP address change, you need to deal with that
- For these reasons, I don’t recommend using this method in production to directly expose your service. If you are running a service that doesn’t have to be always available, or you are very cost sensitive, this method will work for you. A good example of such an application is a demo app or something temporary.

LoadBalancer

With LoadBalancer type, the Service becomes accessible through a cloud provider's load balancer. Each cloud provider (AWS, Azure, Google Cloud, Linode etc) has its own native load balancer implementation.

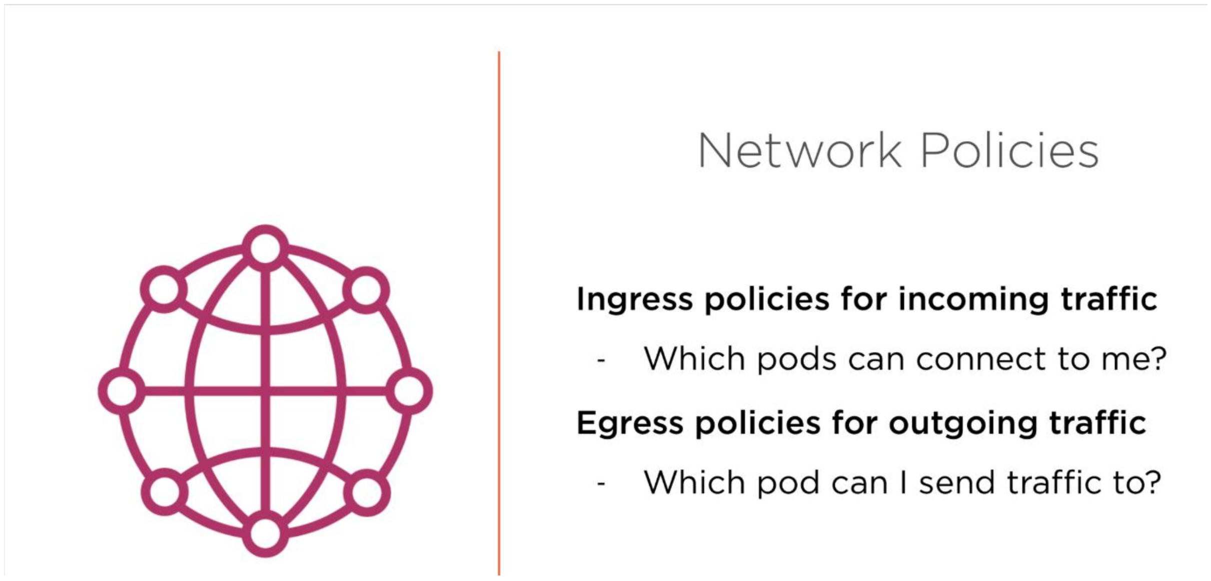


You can also use Ingress to make your Service accessible from outside. It will act as the entry point for your cluster, but Ingress is not a Service type. [K8s Ingress explained here]



ExternalName: This service type maps the service to the contents of the **externalName** field by returning a **CNAME** record with its value and it done by **Ingress Network**.

Ingress Network



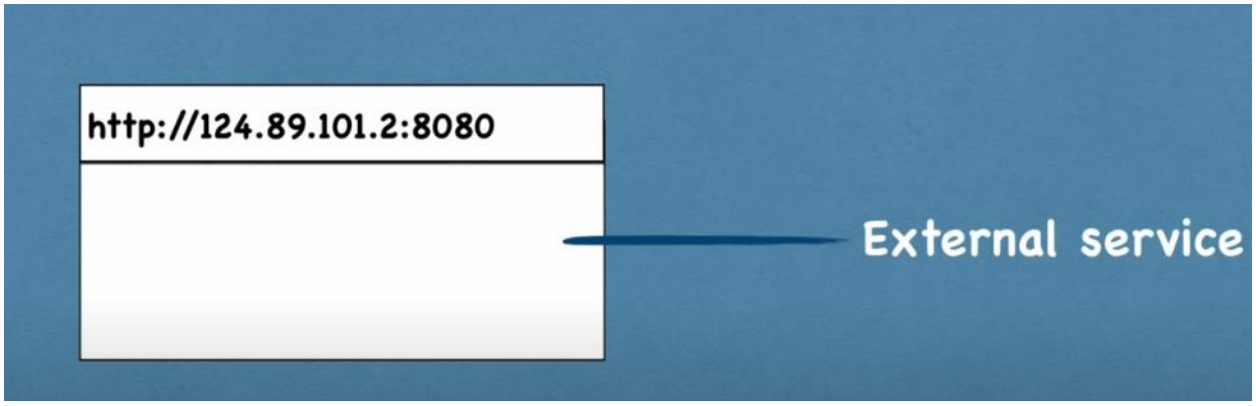
Network Policies

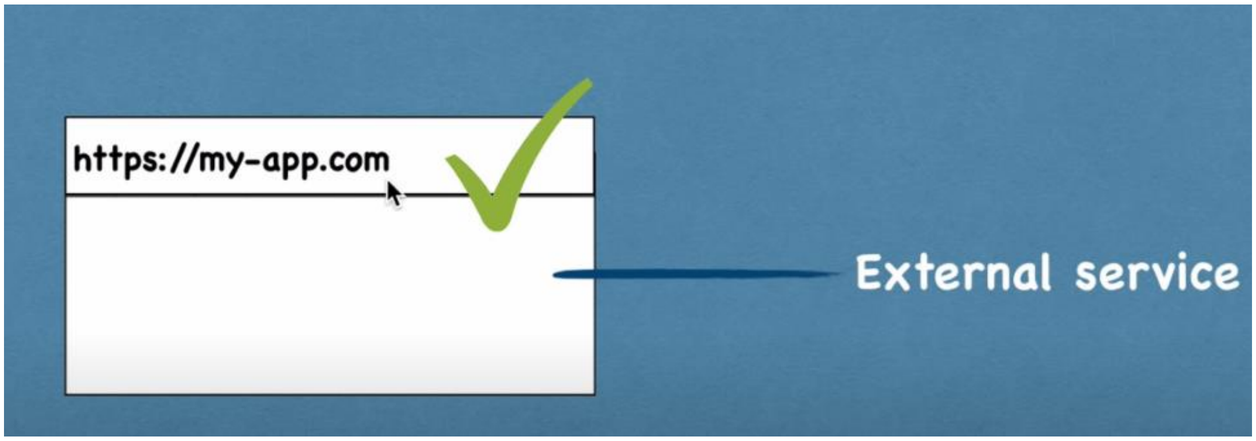
Ingress policies for incoming traffic

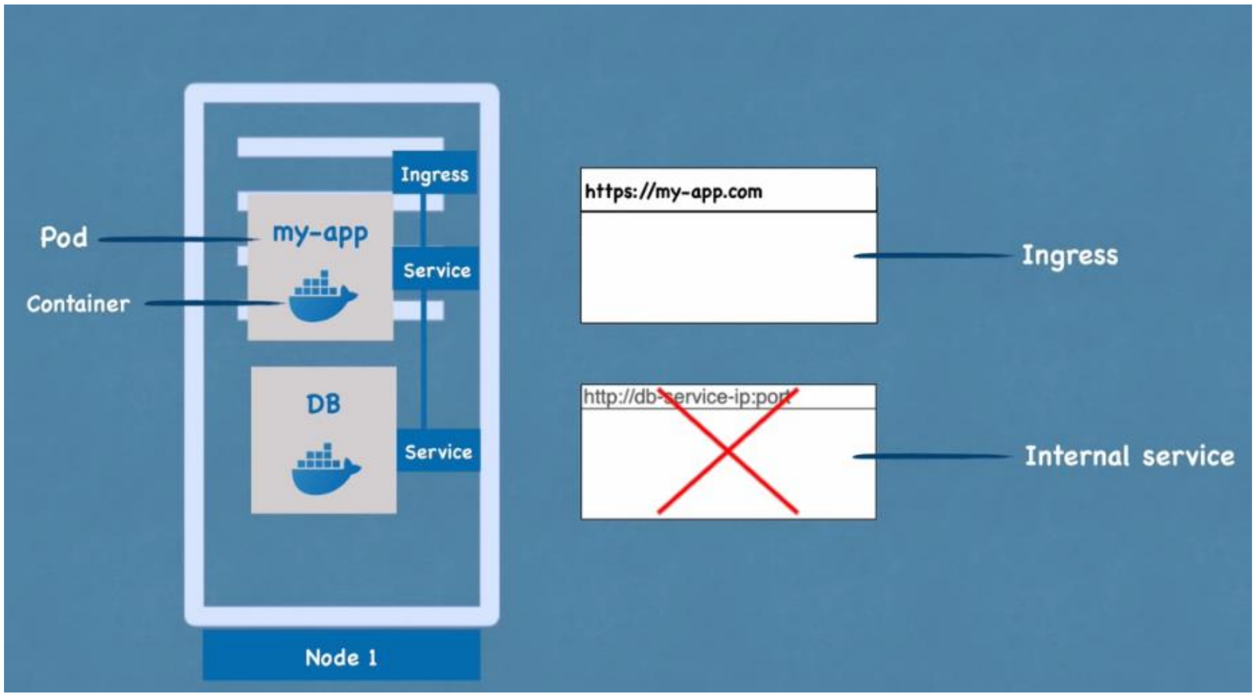
- Which pods can connect to me?

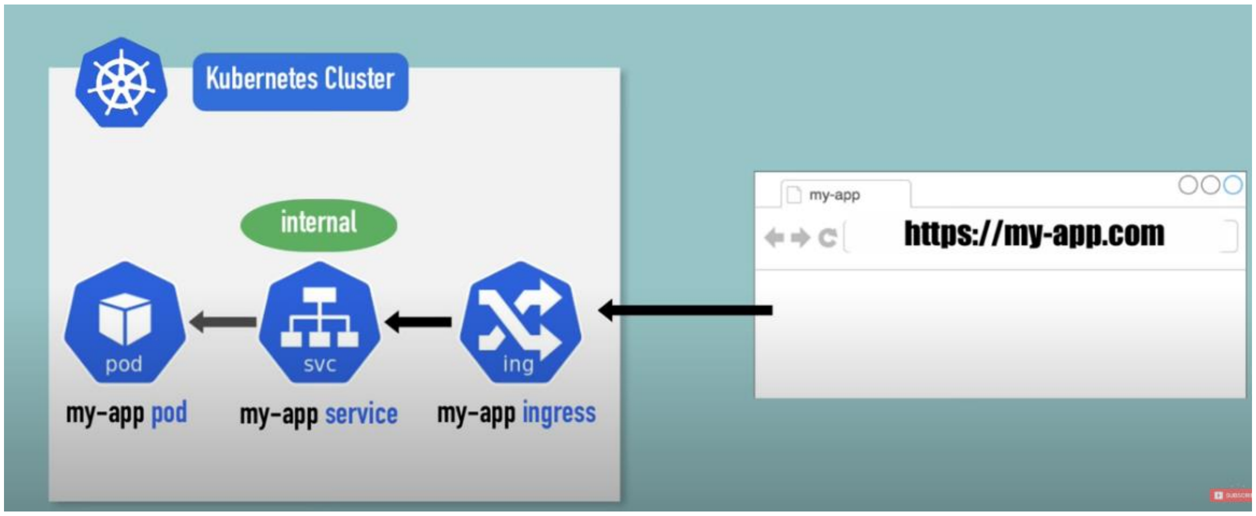
Egress policies for outgoing traffic

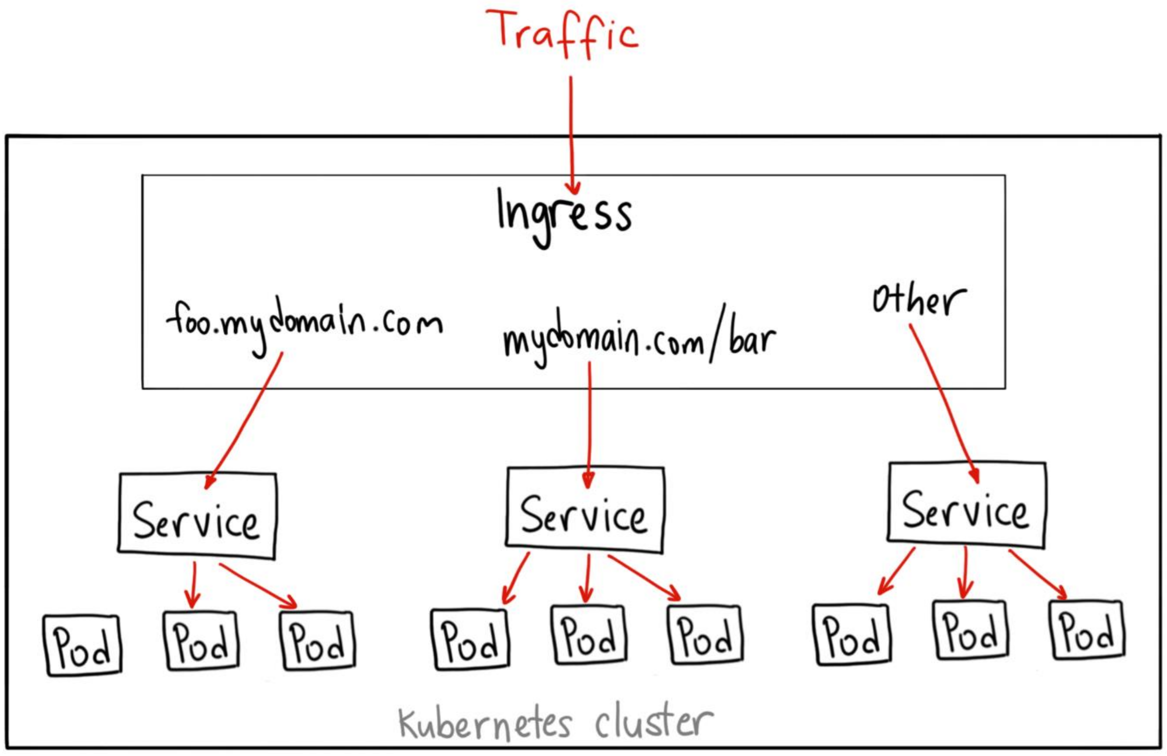
- Which pod can I send traffic to?






Pod
Container
my-app
Ingress
Service
DB
Service
Node 1
https://my-app.com
Ingress
http://db-service-ip:port
Internal service


Kubernetes Cluster
internal
pod
svc
ing
my-app pod
my-app service
my-app ingress
my-app
https://my-app.com

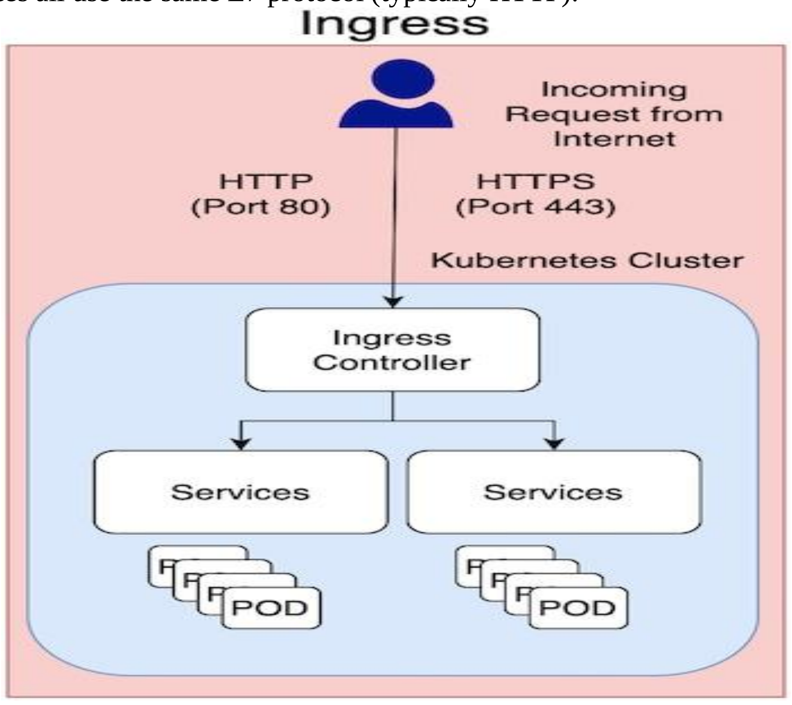


```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  backend:
    serviceName: other
    servicePort: 8080
  rules:
  - host: foo.mydomain.com
    http:
      paths:
      - backend:
          serviceName: foo
          servicePort: 8080
  - host: mydomain.com
    http:
      paths:
      - path: /bar/*
        backend:
          serviceName: bar
          servicePort: 8080
```

- Unlike all the above examples, Ingress is actually NOT a type of service. Instead, it sits in front of multiple services and act as a “smart router” or entry point into your cluster.
- You can do a lot of different things with an Ingress, and there are many types of Ingress controllers that have different capabilities.
- The default GKE ingress controller will spin up a HTTP(S) Load Balancer for you. This will let you do both paths based and subdomain based routing to backend services. For example, you can send everything on foo.yourdomain.com to the foo service, and everything under the yourdomain.com/bar/ path to the bar service.

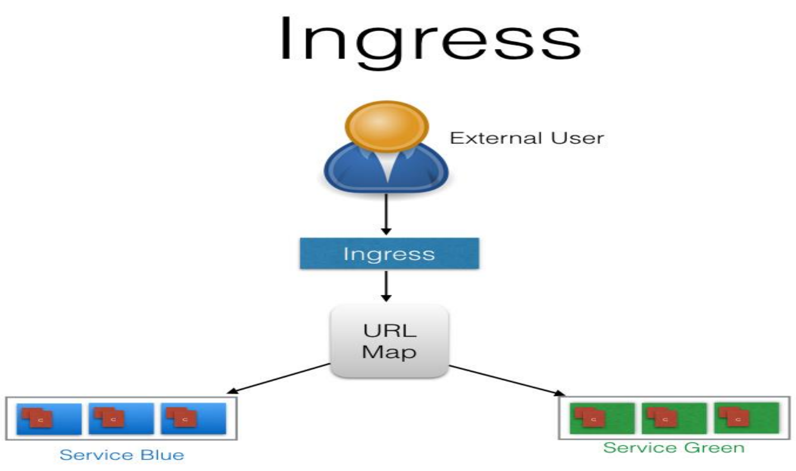
When would you use this?

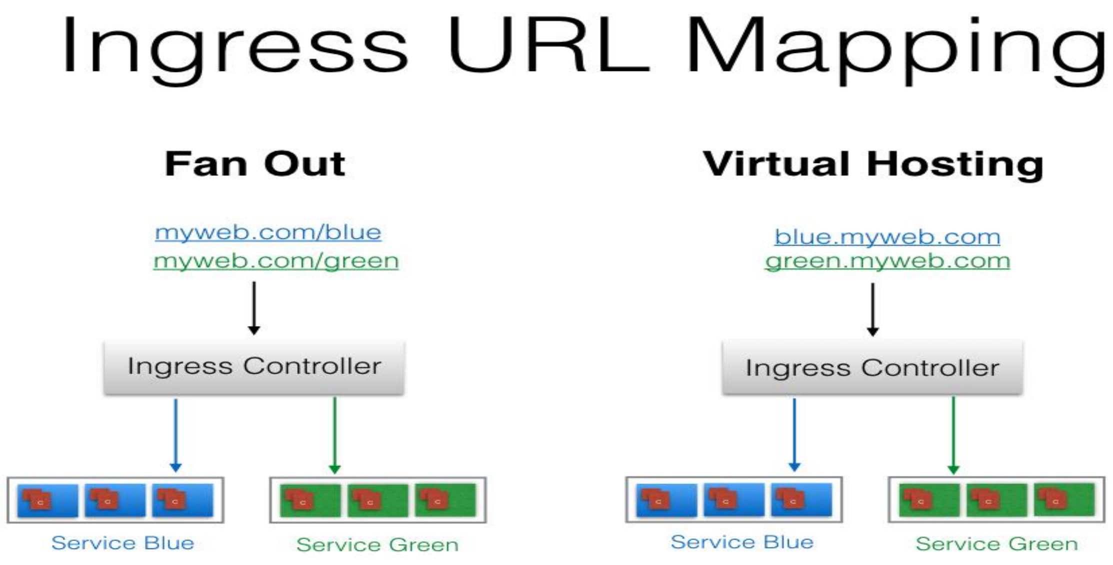
- Ingress is probably the most powerful way to expose your services but can also be the most complicated.
- There are many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more.
- There are also plugins for Ingress controllers, like the cert-manager, that can automatically provision SSL certificates for your services.
- Ingress is the most useful if you want to expose multiple services under the same IP address, and these services all use the same L7 protocol (typically HTTP).



- You only pay for one load balancer if you are using the native GCP integration, and because Ingress is “smart” you can get a lot of features out of the box (like SSL, Auth, Routing, etc)
- Ingress enables you to consolidate the traffic-routing rules into a single resource and runs as part of a Kubernetes cluster. Some reasons Kubernetes Ingress is the preferred option for exposing a service in a production environment include the following:
- Traffic routing is controlled by rules defined on the Ingress Resource.
- Ingress is part of the Kubernetes cluster and runs as pods.
- An external Load Balancer is expensive, and you need to manage this outside the Kubernetes cluster. Kubernetes Ingress is managed from inside the cluster.

- In production environments, you typically use Ingress to expose applications to the Internet. An application is accessed from the Internet via Port 80 (HTTP) or Port 443 (HTTPS), and Ingress is an object that allows access to your Kubernetes services from outside the Kubernetes cluster.

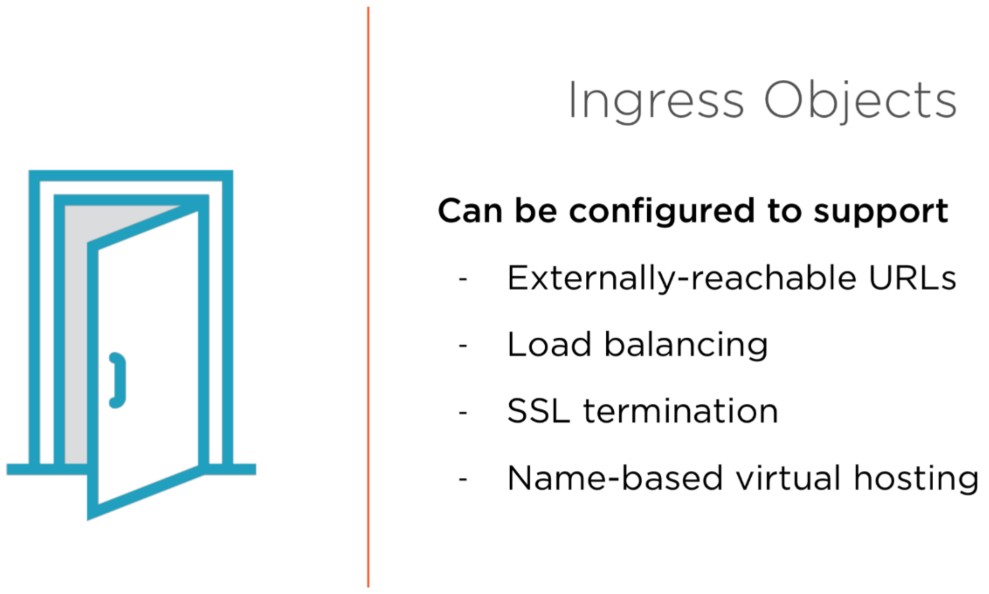




Ingress is split into two main parts – Ingress resources and ingress controller

Ingress Resources

- Ingress Resources defines how you want the requests to the services to be routed.
- It contains the main routing rules.

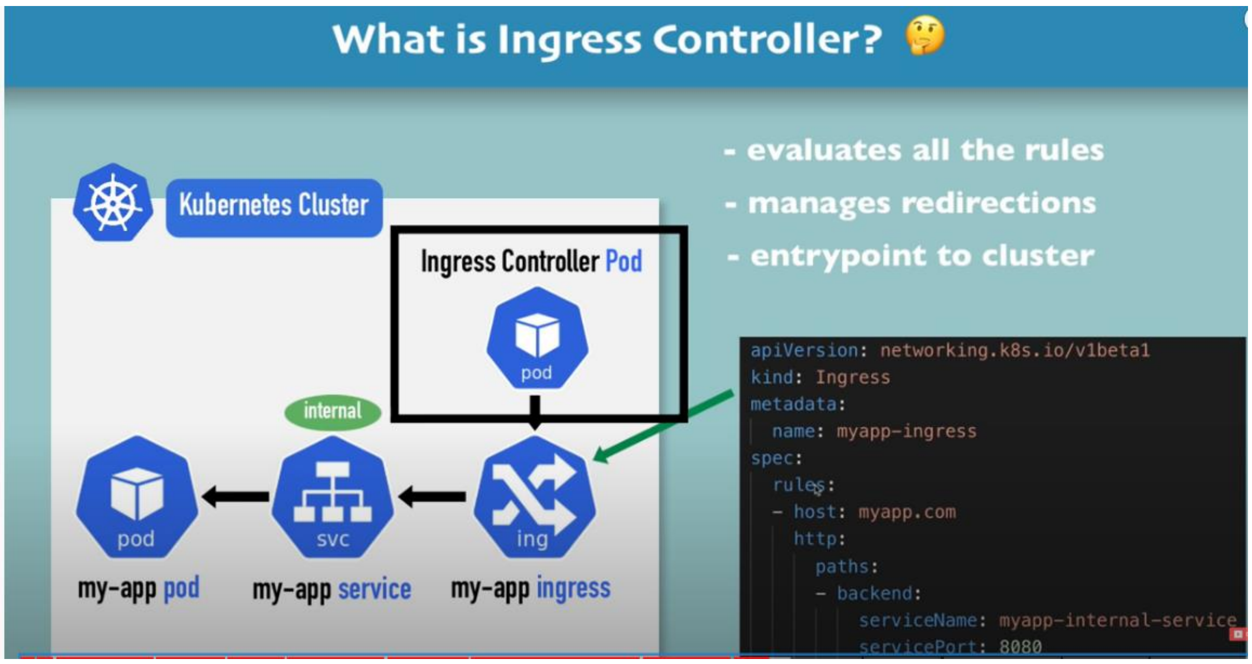


Ingress Objects

Can be configured to support

- Externally-reachable URLs
- Load balancing
- SSL termination
- Name-based virtual hosting

Ingress controller



What is Ingress Controller? 🤔

- evaluates all the rules
- manages redirections
- entrypoint to cluster

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - backend:
          serviceName: myapp-internal-service
          servicePort: 8080
```

- Ingress controller reads the ingress resource's information and process the data accordingly. So basically, ingress resources contain the rules to route the traffic and ingress controller routes the traffic.
- Traffic routing is done by an ingress controller.
- There are many types of Ingress controllers, from the Google Cloud Load Balancer, Nginx, Contour, Istio, and more.