

# Container Orchestration with Kubernetes

## Key Takeaways

# Introduction to Kubernetes

# Official Definition

- Open source **container orchestration tool**
- Developed by **Google**
- **Automates** many processes involved in deploying, managing and scaling containerized applications

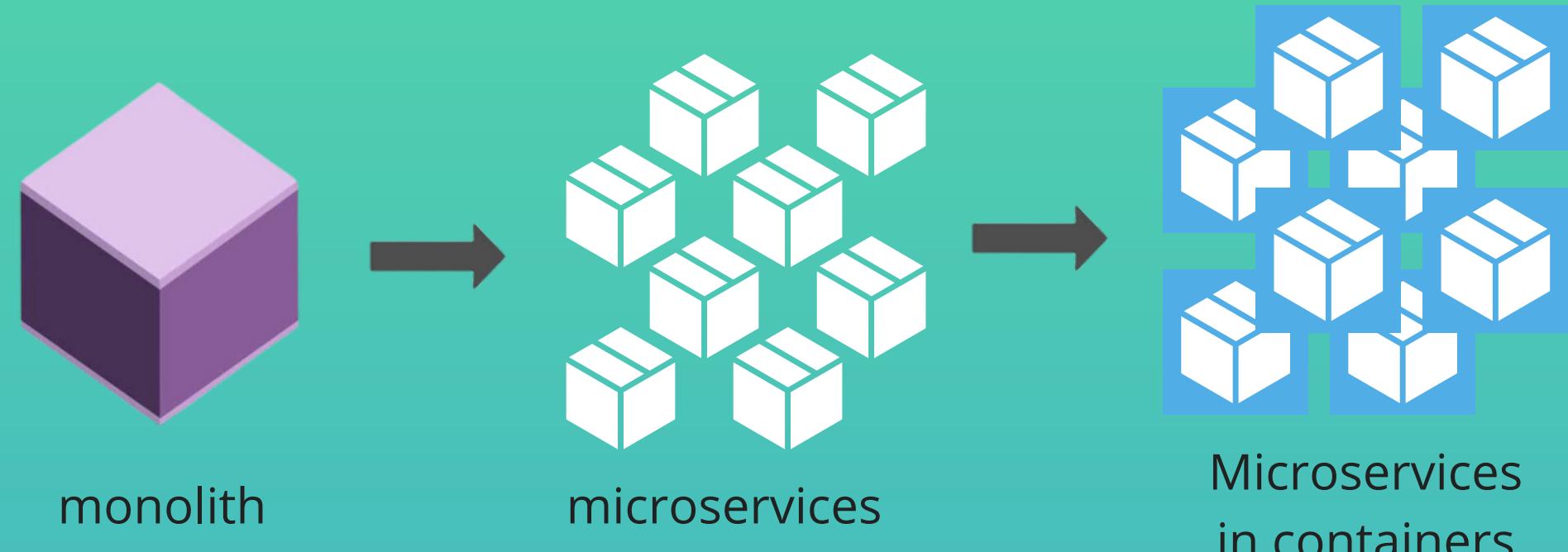


- **Most used** container orchestration platform
- Also known as "K8s" or "Kube"



# The need for Kubernetes

- Trend from **Monolith** to **Microservices**
- Containers are the perfect host for microservice applications
- Resulted in an **increased usage of containers**



- Manually **managing hundreds or hundreds of 1000s containers** is a lot of effort
- Kubernetes automates many of those manual tasks and provides
  - **High Availability** or no downtime
  - Automatic **Scaling**
  - **Disaster Recovery** - Backup and Restore
  - **Self-Healing**



# Core Kubernetes Components - 1

Kubernetes has many components, but these are the **main ones you need to know:**

**POD**

**SERVICE**

**CONFIGMAP**

**INGRESS**

**SECRET**

**DEPLOYMENT**

**VOLUMES**

**STATEFULSET**

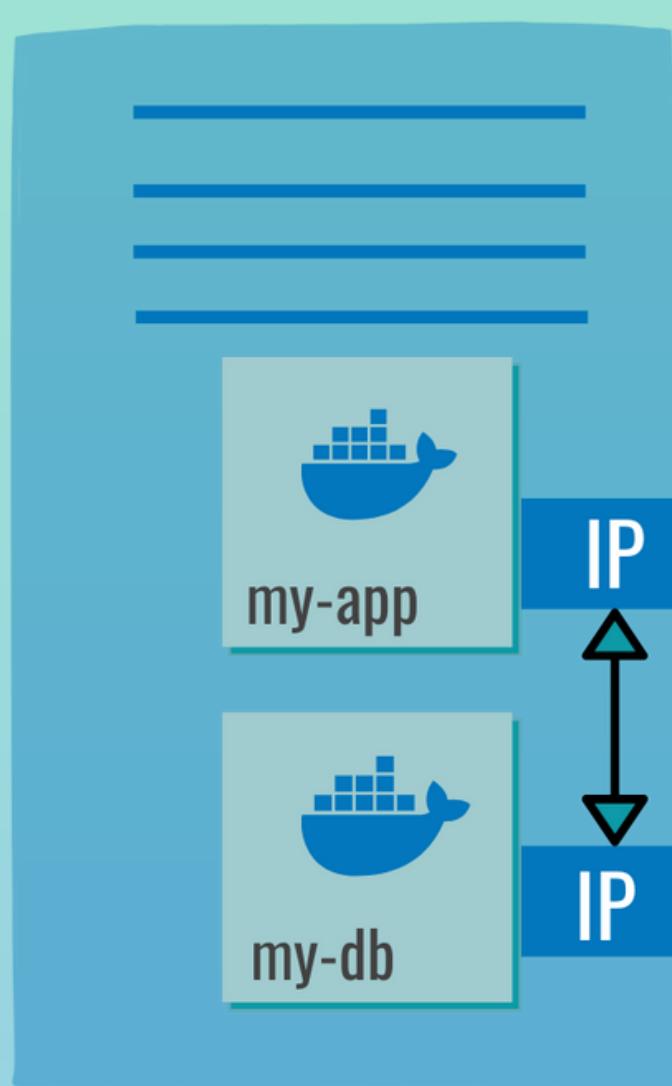
# Core Kubernetes Components - 2

## POD

- Group of 1 or more containers
- Smallest unit of K8s
- An abstraction over container
- Usually 1 application/container per Pod
- Pods are ephemeral



New IP address assigned on re-creation



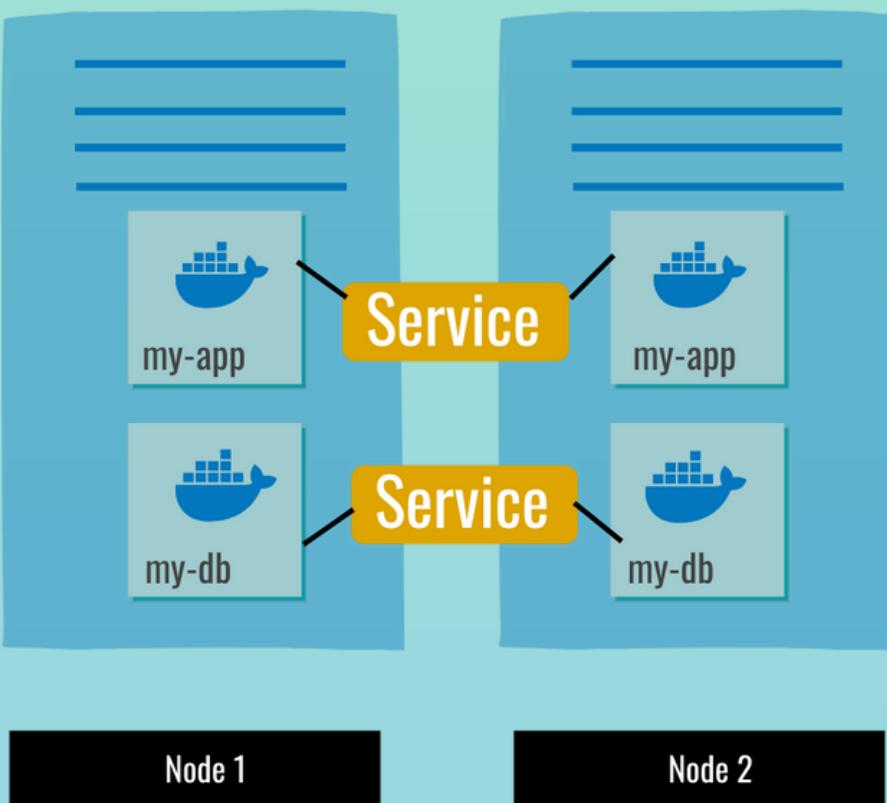
# Core Kubernetes Components - 3

## SERVICE

- Basically a static or **permanent IP address** that can be attached to each Pod
- Also serves as a **loadbalancer**
- **Lifecycles** of Service and Pod are **not connected**



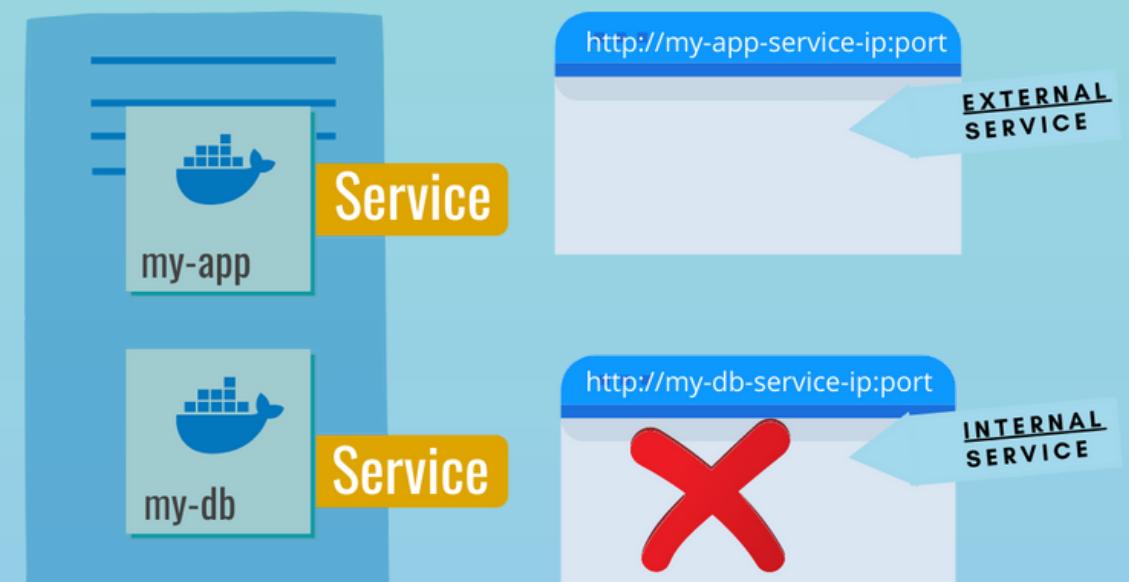
If Pod crashes, the Service and its IP address will be the same



## Internal vs External Service

When creating a service you can specify its **type**:

- **Internal Service:** By default, for example a database, which should not be accessible from outside
- **External Service:** Application accessible through browser



# Core Kubernetes Components - 4



URL of external Service:



http://124.89.101.2:8080



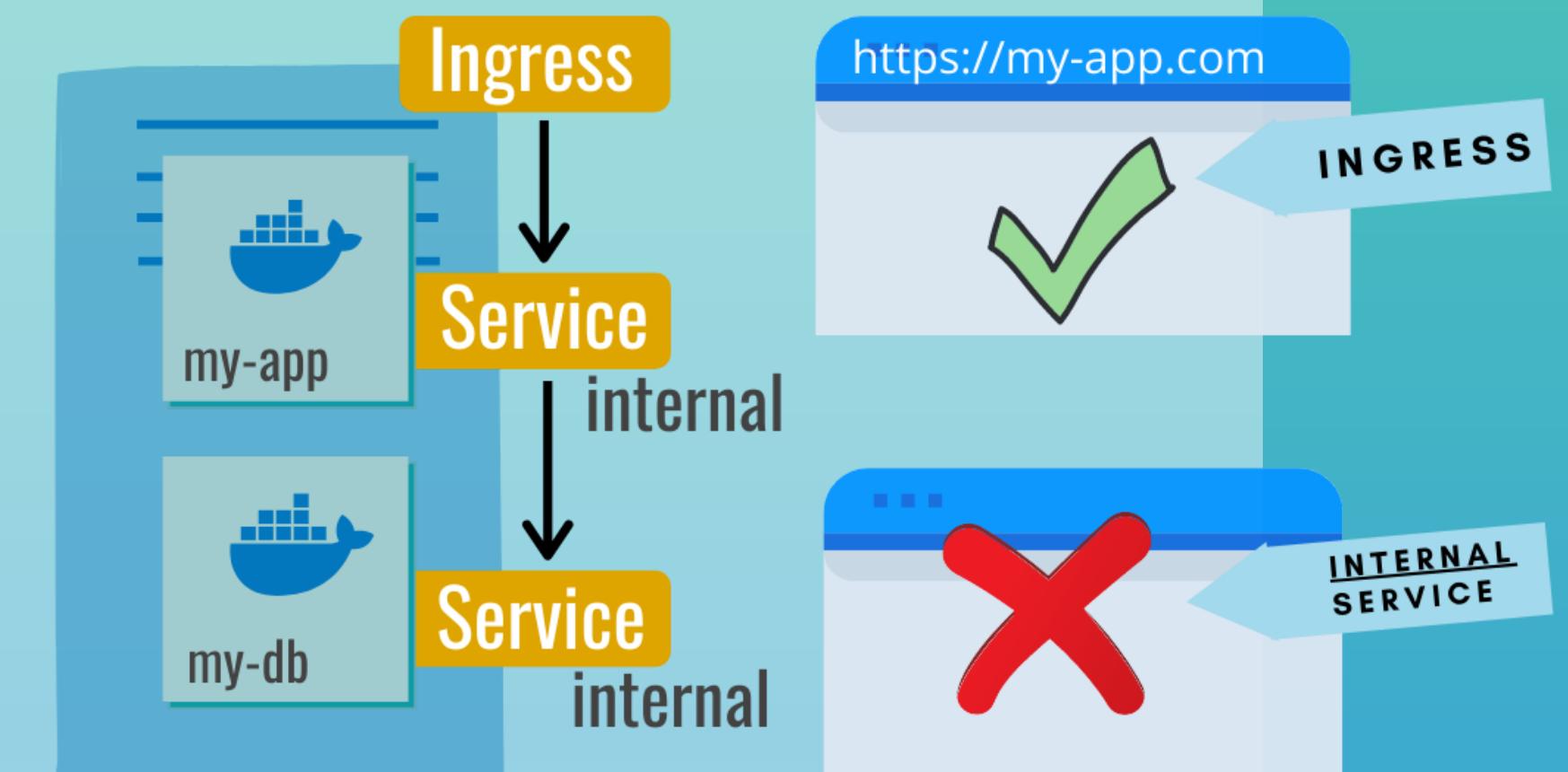
URL of Ingress Service:



https://my-app.com

## INGRESS

- Ingress is the **entrypoint to your K8s cluster**
- Request goes to Ingress first, which does the forwarding to the Service



# Core Kubernetes Components - 5

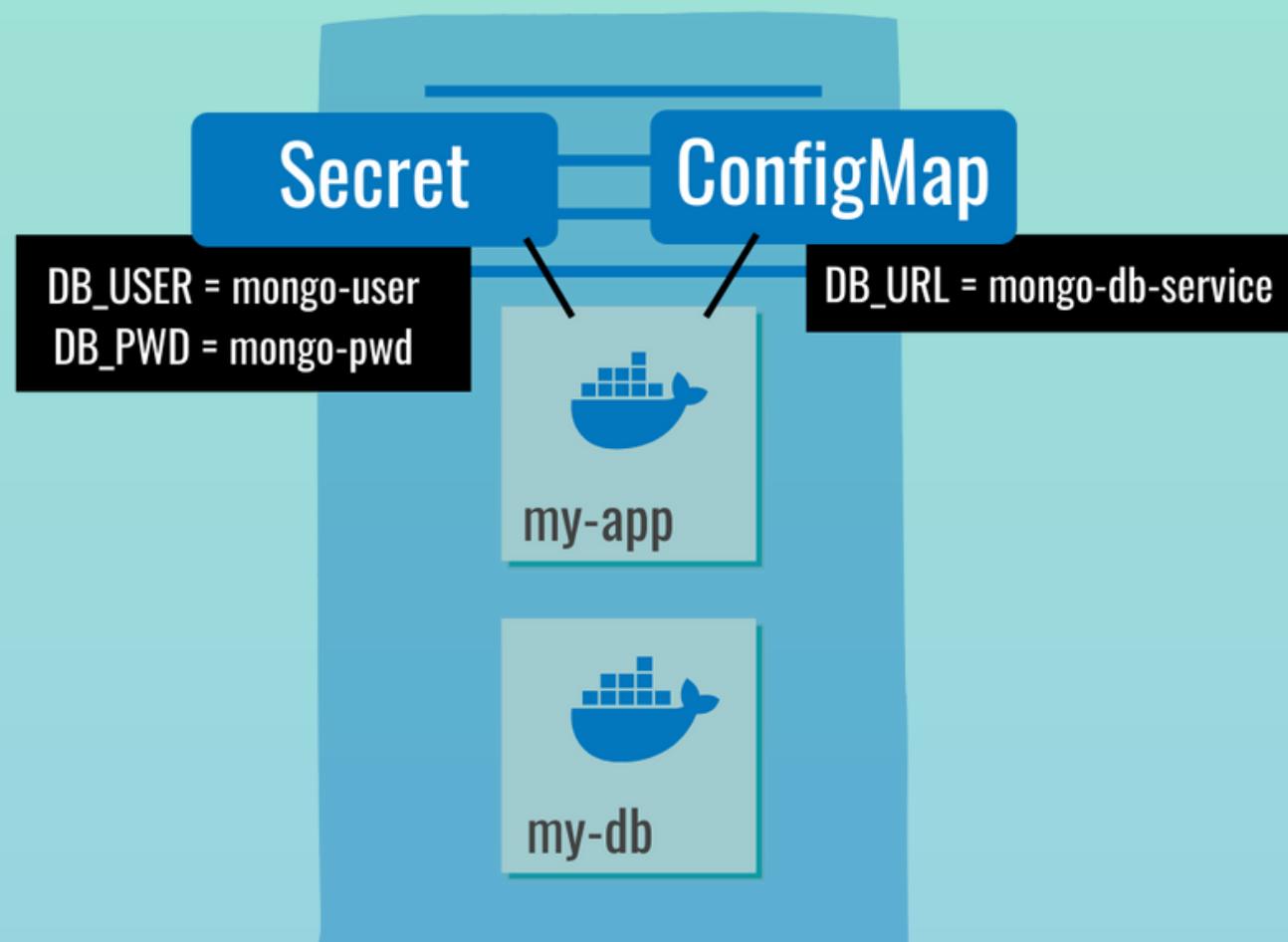
For **external configuration**, Kubernetes has these 2 components:

## CONFIGMAP

- To **store non-confidential data** in key-value pairs

## SECRET

- Similar to ConfigMap, but to **store sensitive data** such as passwords or tokens



- **Pods** can consume ConfigMaps and Secrets as environment variables, CLI arguments or as config files in a Volume



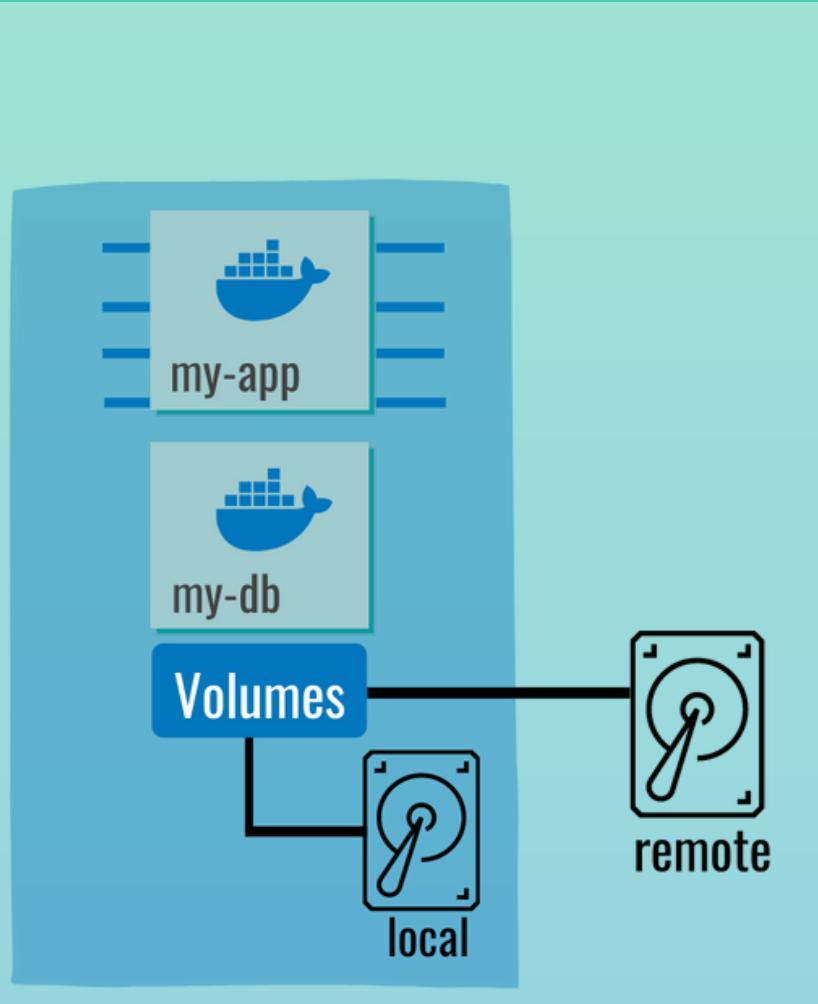
**Storing the data in a Secret component doesn't automatically make it secure.** There are built-in mechanisms (like encryption, defining authorization policies) for basic security, which are not enabled by default! It's recommended to use third-party secret management tools, because the provided capabilities by K8s are not enough for most companies

# Core Kubernetes Components - 6

When a container crashes, K8s restarts the container but with a clean state. Meaning your **data is lost!**

## VOLUME

- Volume component basically **attaches a physical storage** on a hard drive to your Pod
- Storage could be either on a local server or outside the K8s cluster



Kubernetes cluster



storage

Think of storage as an **external hard drive plugged in** to your K8s cluster



K8s doesn't manage any data persistence, meaning **you are responsible for backing up, replicating the data etc.**



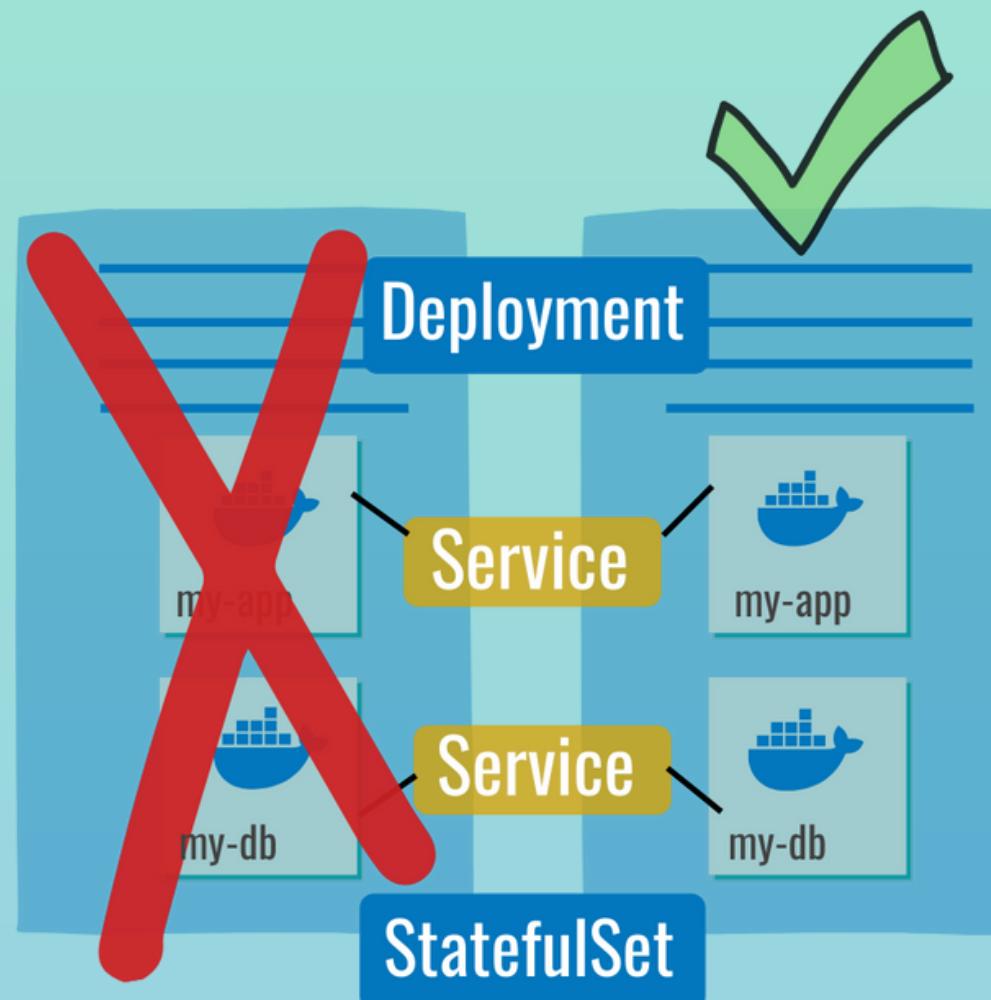
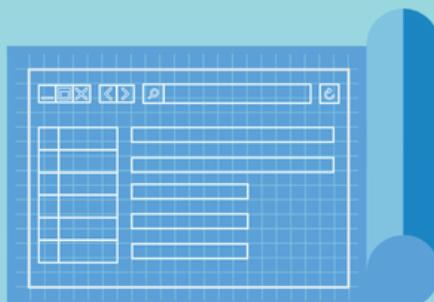
TECHWORLD  
WITH NANA

# Core Kubernetes Components - 7

Deployment and StatefulSet are an abstraction of Pods

## DEPLOYMENT

- Blueprint for Pods
- You work with Deployments and by defining the number of **replicas**, K8s creates Pods



## STATEFULSET

- Blueprint for stateful applications
- Like databases etc.
- In addition to **replicating features**, StatefulSet makes sure database reads and writes are synchronized to **avoid data inconsistencies**



Having load balanced replicas makes our setup much **more robust**

# Kubernetes Architecture

# 2 Types of Nodes

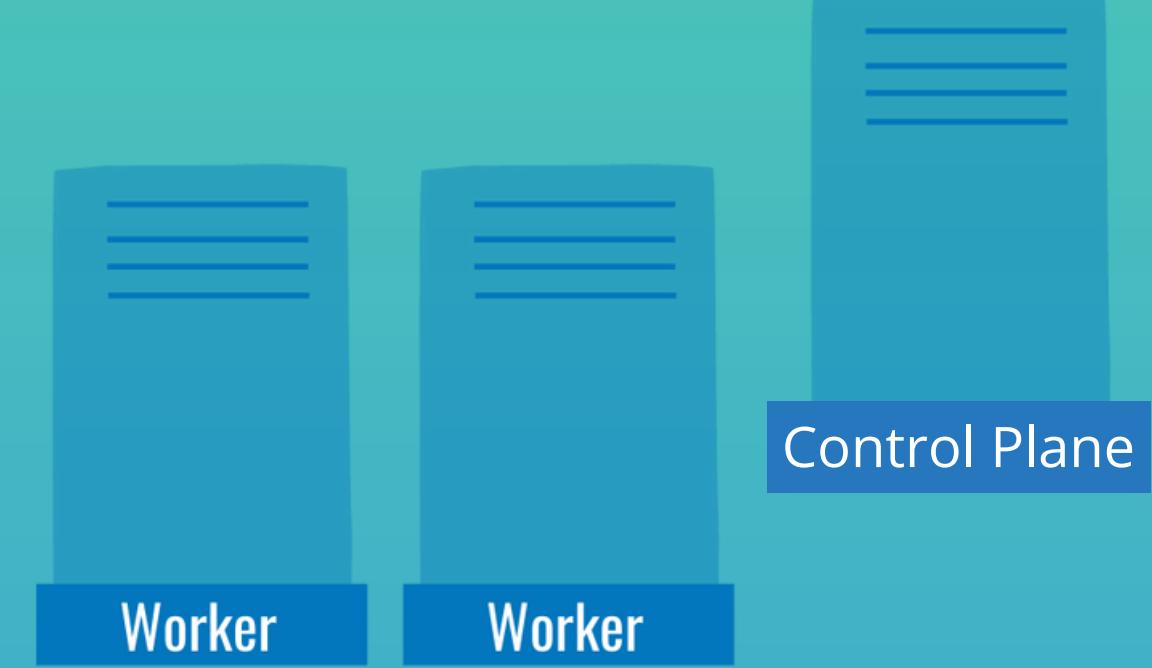
- A Kubernetes cluster consists of a set of worker machines, called "Nodes"

## Worker Node

- The **containerized applications run on the Worker Nodes**
- Each Node runs multiple Pods on it
- Much more compute resources needed, because the actual workload runs on them

## Control Plane

- **Manages** the Worker Nodes and the Pods in the cluster
- Much more important and needs to be replicated
- So in production, replicas run across multiple machines



# Worker Node Components

Each worker node needs to have **3 processes installed:**

## 1) Container Runtime

- Software responsible for running containers
- For example containerd, CRI-O or Docker

## 2) Kubelet

- Agent that makes sure containers are running in a Pod
- Talks to underlying server (to get resources for Pod) and container runtime (to start containers in Pod)

## 3) Kube-proxy

- A network proxy with **intelligent forwarding** of requests to the Pods

## 3) Kube-proxy

## 2) Kubelet

## 1)

## Worker

# Control Plane Components - 1

- Each control plane needs to have **4 processes installed:**

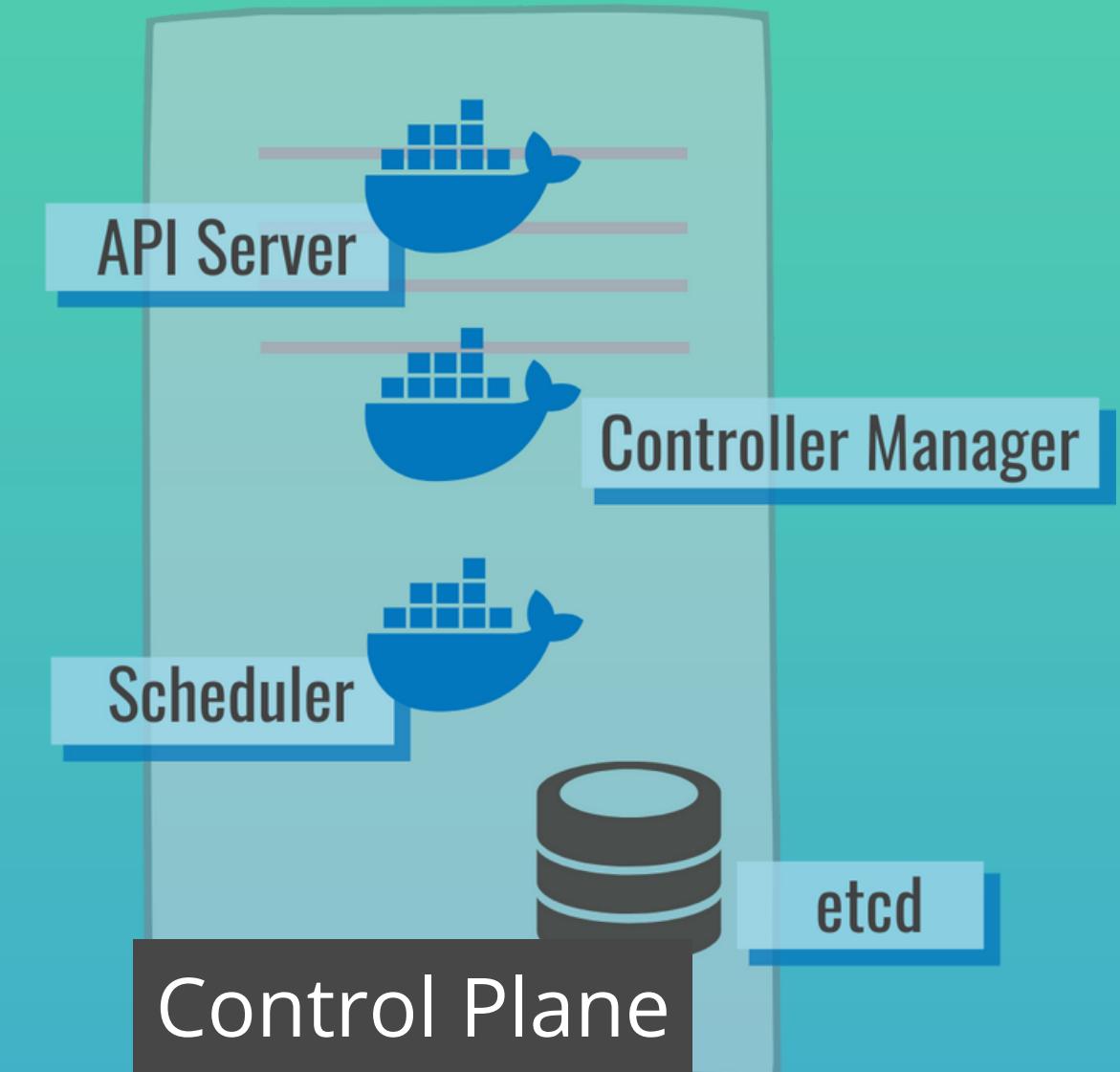
1) API Server

2) Scheduler

3) Controller Manager

4) etcd

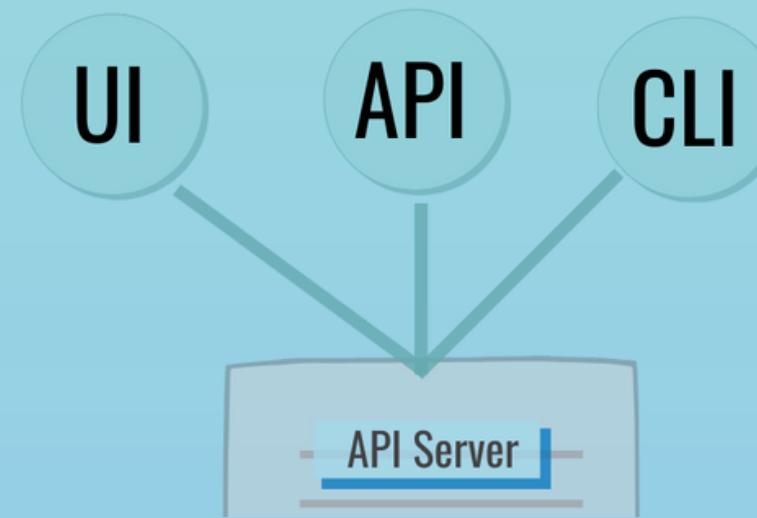
- Control Plane makes global decisions about the cluster
- Detects and responds to cluster events



# Control Plane Components - 2

## 1) API Server

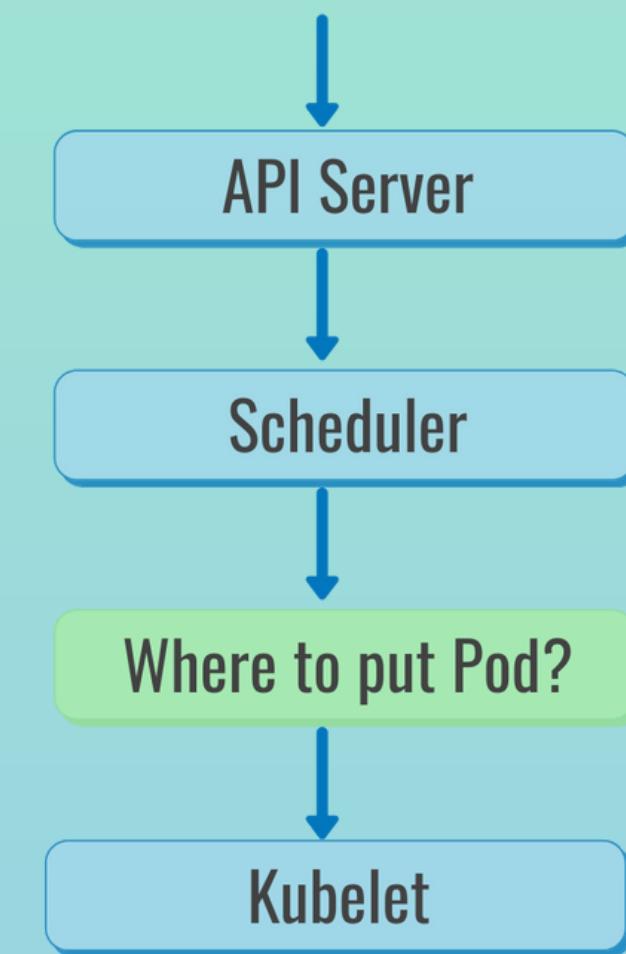
- The cluster gateway - **single entrypoint to the cluster**
- Acts as a gatekeeper for authentication, validating the request
- Clients to interact with the API server: UI, API or CLI



## 2) Scheduler

- **Decides** on which Node new Pod should be scheduled
- Factors taken into account for scheduling decisions: resource requirements, hardware/software/policy constraints, data locality, ...

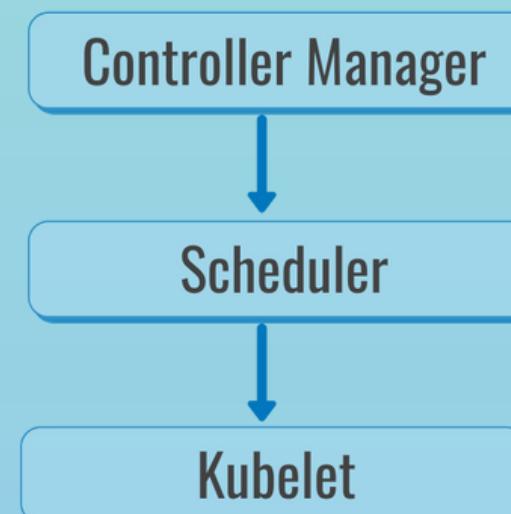
SCHEDULE NEW POD



# Control Plane Components - 3

## 3) Controller Manager

- Detects state changes, like crashing of Pods and tries to recover the cluster state as soon as possible
- For that it makes a **request to the Scheduler** to reschedule those Pods and the same cycle happens



## 4) etcd

- K8s' backing **store for all cluster data**. A consistent, high-available key-value store
- Think of it as a cluster brain, every change in the cluster gets saved or updated into it
- All other processes like Scheduler, Controller Manager etc **work based on the data in etcd** as well as communicate with each other through etcd store



The **actual application data is NOT stored** in the etcd store



# Increase Kubernetes Cluster Capacity

As your application grows and its demand for resources increases, you may actually **add more Nodes** to your cluster, thus forming a more powerful and robust cluster to meet your application resource requirements

## Add a Control Plane Node

1. Get a fresh new server
2. Install all control plane processes on it
3. Join it to the K8s cluster using a K8s command

## Add a Worker Node

1. Get a fresh new server
2. Install all the Worker Node processes, like container runtime, Kubelet and KubeProxy on it
3. Join it to the K8s cluster using a K8s command



# Deep Dive into Kubernetes

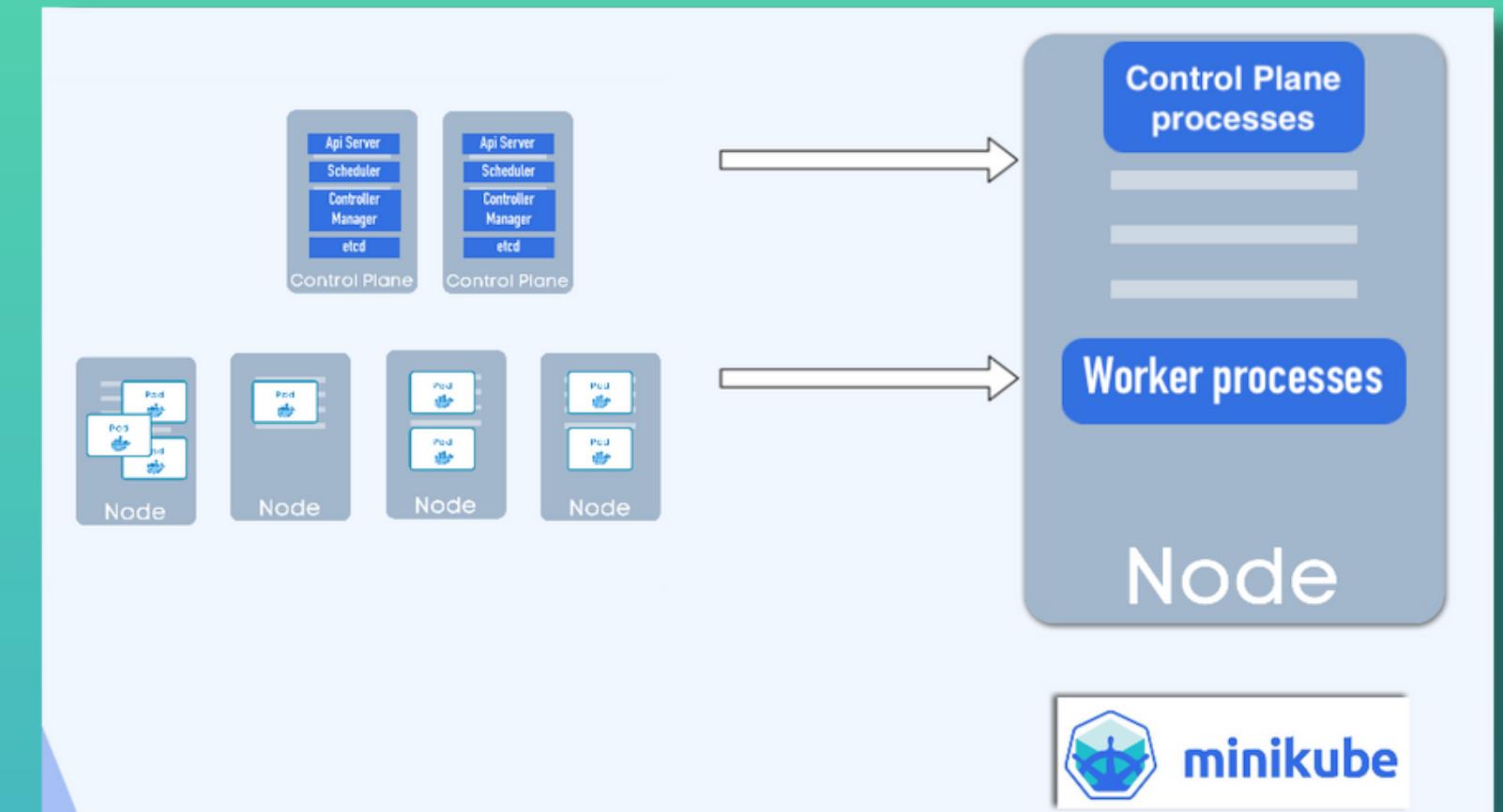
# Minikube - Local Cluster Setup

- Minikube implements a local K8s cluster
- Useful for local K8s application development, because running a test cluster would be complex

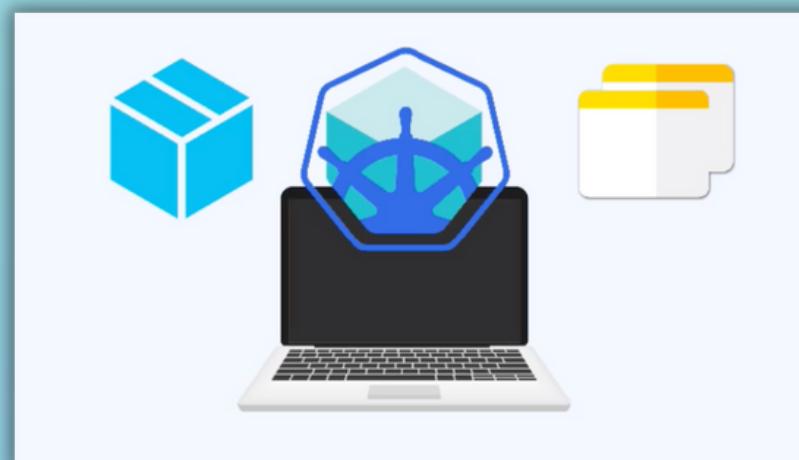


Control Plane and Worker

processes run on **ONE machine**



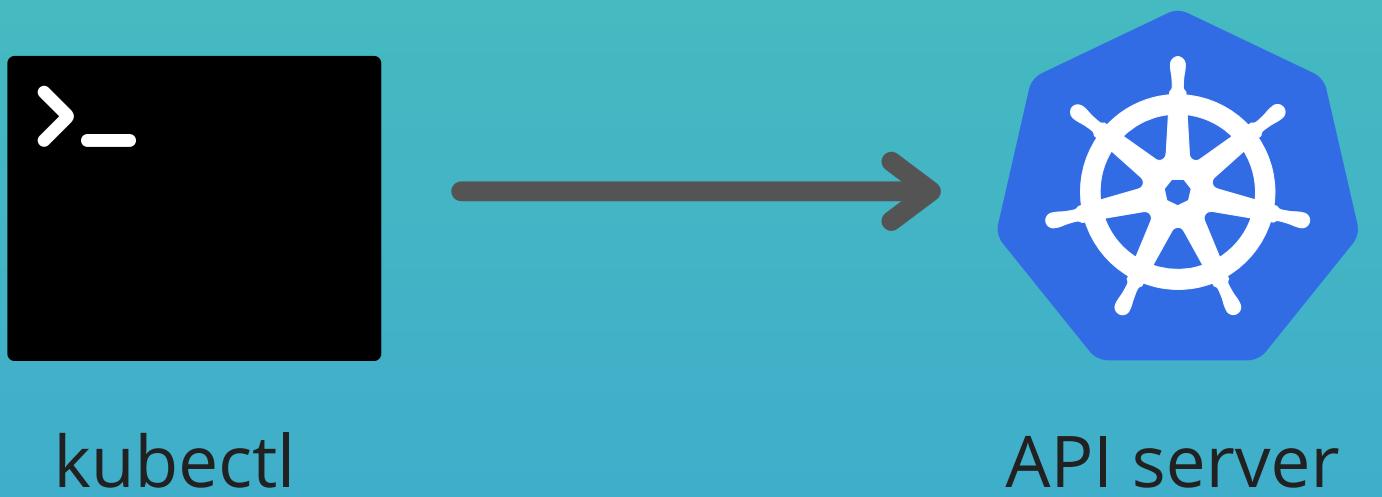
Run Minikube either as a **container** or **virtual machine** on your laptop



- So you need a container runtime or virtual machine manager on your laptop

# Kubectl - Command Line Tool

- CLI Tool to **interact with your K8s cluster**
- In order for kubectl to access a K8s cluster, it needs a **kubeconfig file**, which is created automatically when deploying your minikube cluster
- By default, config file is located at **~/.kube/config**



Basic kubectl commands:

```
● ● ●  
kubectl get {k8s-component}  
kubectl get nodes  
kubectl get pods  
kubectl get services  
kubectl get deployment
```

Get status of different components

```
kubectl create {k8s-component} {name} {options}  
kubectl create deployment my-nginx-depl --image=nginx
```

CRUD

```
kubectl edit {k8s-component} {name}  
kubectl delete {k8s-component} {name}
```

```
Debugging
```

```
kubectl logs {pod-name}  
kubectl describe {pod-name}
```

```
kubectl exec -it {pod-name} -- bash  
kubectl apply -f config-file.yaml
```

# K8s YAML Configuration File - 1

- Also called "Kubernetes manifest"
- Declarative: A manifest **specifies the desired state** of a K8s component
- Config files are in **YAML format**, which is user-friendly, but requires **strict indentation!**
- Config files should be stored in version control

Each configuration file has **3 parts**:

## 1) metadata

```
! nginx-deployment.yaml ✘
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  labels:...
6
7  spec:
8    replicas: 2
9    selector:...
10   template:...
```

```
! nginx-service.yaml ✘
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: nginx-service
5
6  spec:
7    selector:...
8    ports:...
```

## 2) specification

- Attributes of "spec" are specific to the kind

```
! nginx-deployment.yaml ✘
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  labels:...
6
7  spec:
8    replicas: 2
9    selector:...
10   template:...
```

```
! nginx-service.yaml ✘
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: nginx-service
5
6  spec:
7    selector:...
8    ports:...
```

# K8s YAML Configuration File - 2

Current status: 1 replica

```
status:  
  availableReplicas: 1  
  conditions:  
    - lastTransitionTime: "2020-01-24T10:54:59Z"  
      lastUpdateTime: "2020-01-24T10:54:59Z"  
      message: Deployment has minimum availability.  
      reason: MinimumReplicasAvailable  
      status: "True"  
      type: Available  
    - lastTransitionTime: "2020-01-24T10:54:56Z"  
      lastUpdateTime: "2020-01-24T10:54:59Z"  
      message: ReplicaSet "nginx-deployment-7d64f4b..."  
      reason: NewReplicaSetAvailable  
      status: "True"  
      type: Progressing  
  observedGeneration: 1  
  readyReplicas: 1  
  replicas: 1  
  updatedReplicas: 1
```



## 3) status

- **Automatically generated** and added by Kubernetes
- K8s gets this information from etcd, which holds the current status of any K8s component

Desired status: 2 replicas

```
! nginx-deployment.yaml ✘  
1  apiVersion: apps/v1  
2  kind: Deployment  
3  metadata:  
4    name: nginx-deployment  
5  +   labels: ...  
7  spec:  
8    replicas: 2  
9  +   selector: ...  
12 +   template: ...  
22
```

# Deployment Configuration File

- Deployment Configuration is a bit special

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels: ...
spec:
  replicas: 2
  selector: ...
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 8080
```

- Since it's an abstraction over Pod, we have
  - the **Pod configuration inside Deployment configuration**
- Own "metadata" and "spec" section
- Blueprint for Pod

# Labels & Selectors

## Labels

- Labels are **key/value pairs** that are attached to resources, such as Pods.
- Used to specify identifying attributes that are meaningful and relevant to users

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  labels:
6    app: nginx
7  spec:
8    replicas: 2
9    selector:
10   matchLabels:
11     app: nginx
12   template:
13     metadata:
14       labels:
15         app: nginx
16   
```

## Label Selectors

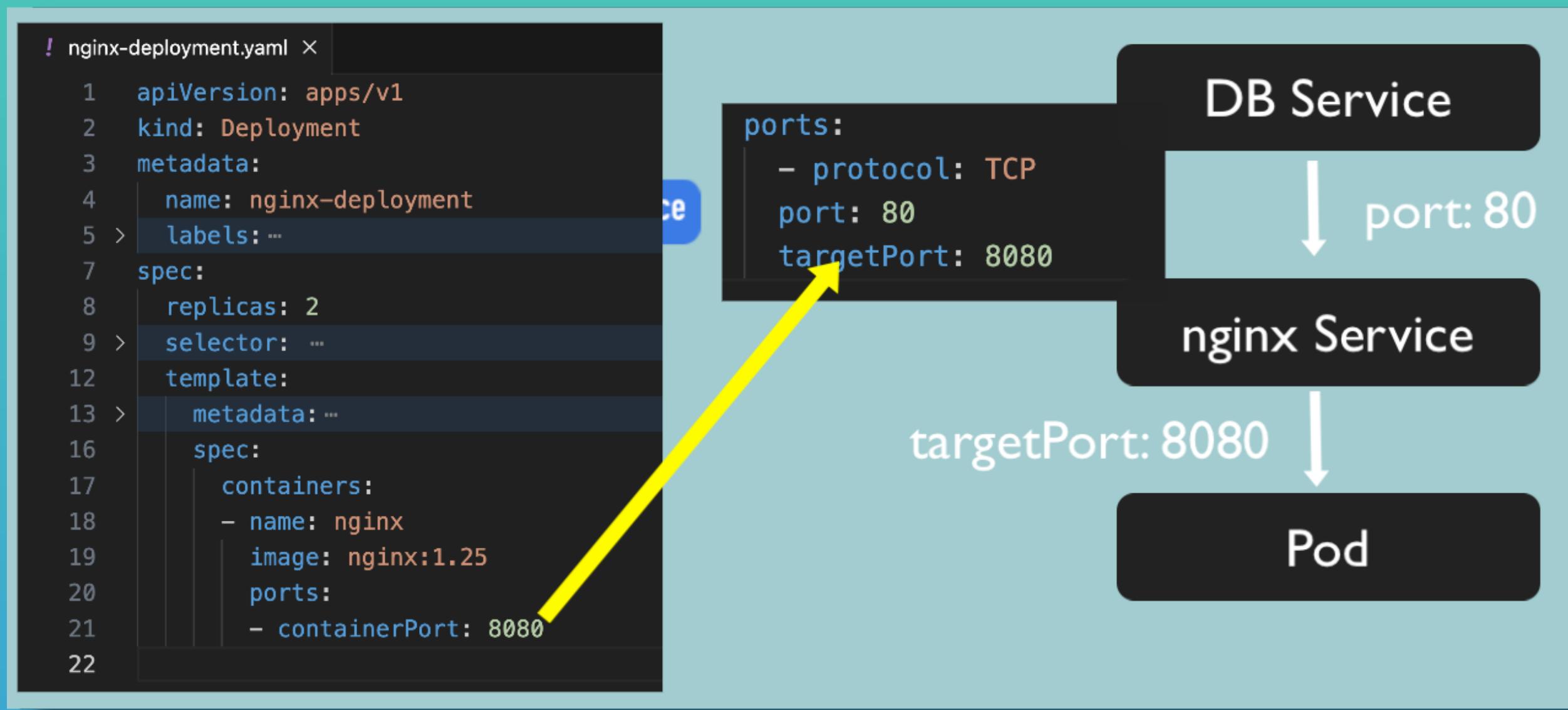
- Labels do not provide uniqueness
- Via **selector** the user can identify a set of resources

## Connecting Services to Deployments

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: nginx-service
5  spec:
6    selector:
7      app: nginx
8    ports: ...
```

# Ports in Service and Pod

- In Service component you need to specify:
  - **port** = the port where the service itself is accessible
  - **targetPort** = port, the container accepts traffic on
- In Deployment component:
  - **containerPort** = port, the container accepts traffic on



# Browser Request Flow through the K8s components

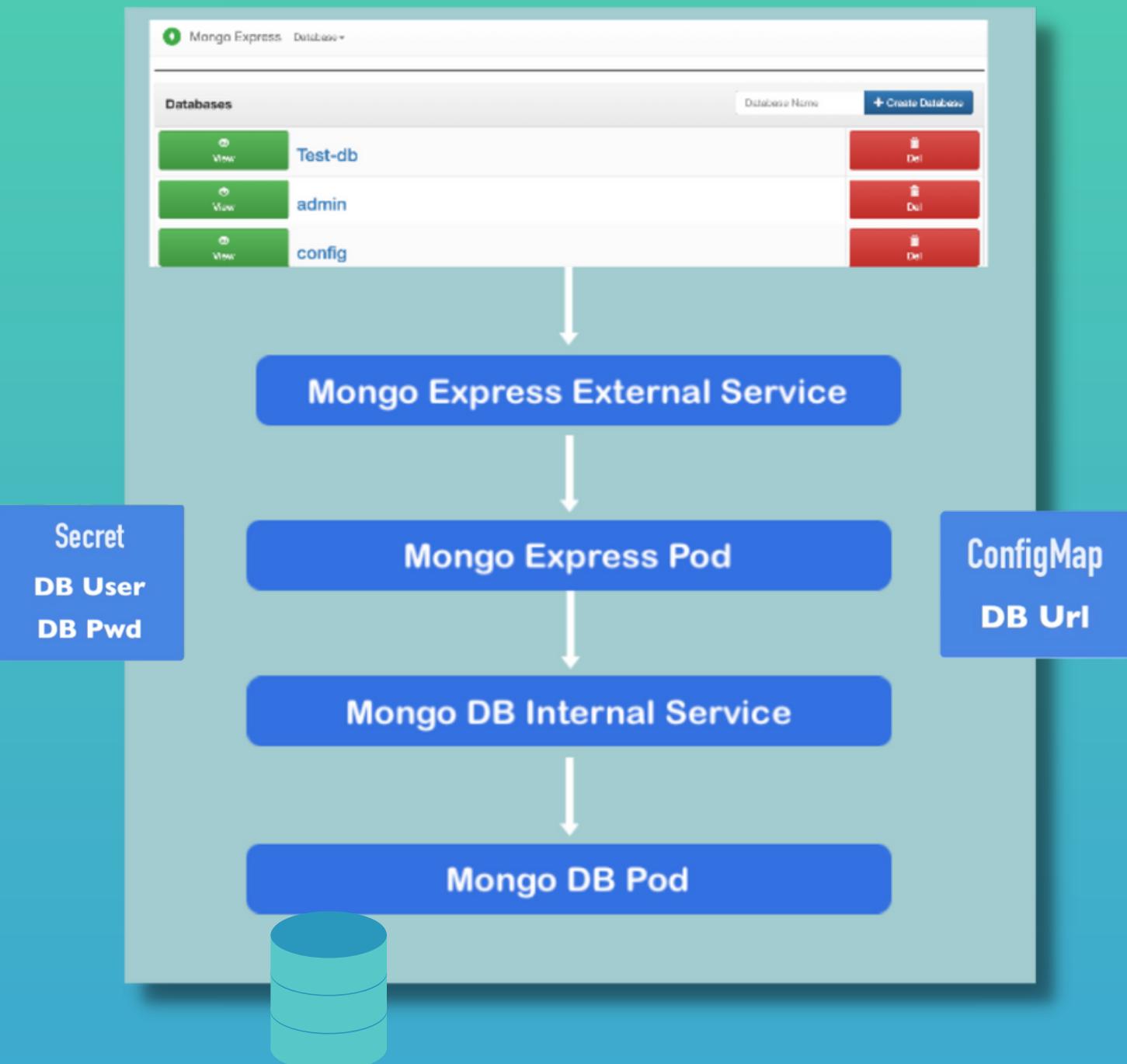
## Example Setup:

- Mongo Express as UI
- MongoDB as database
- User updates entries in database via browser
- ConfigMap and Secret holds the MongoDB endpoint (Service name of MongoDB) and credentials (user, pwd), which gets injected to MongoExpress Pod, so MongoExpress can connect to the DB

### K8s Components needed in this setup

- 2 Deployment / Pod
- 2 Services
- 1 ConfigMap
- 1 Secret

URL: IP address of Node + Port of external Service

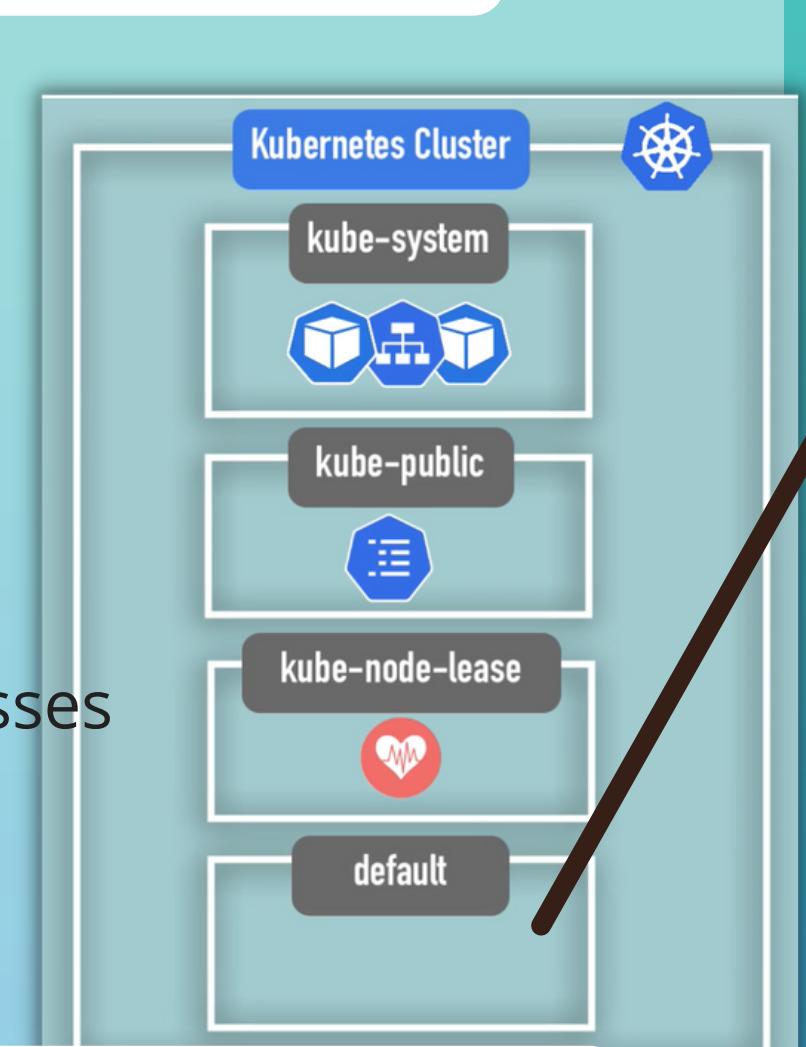


# Namespaces - 1

- Namespaces provide a **mechanism for isolating groups of resources** within a single cluster
- Names of resources need to be unique within a namespace, but not across namespaces
- Like a virtual cluster inside a cluster

## Namespaces per default

- Namespaces per default, when you install K8s
- "kube-system" has control plane processes running



DON'T modify kube-system!!

## "Default" namespace

- Start deploying your application in the default namespace called "default"

## Create a new namespace

- Via **kubectl** command:
- Via **configuration file**:

***kubectl create namespace my-ns***

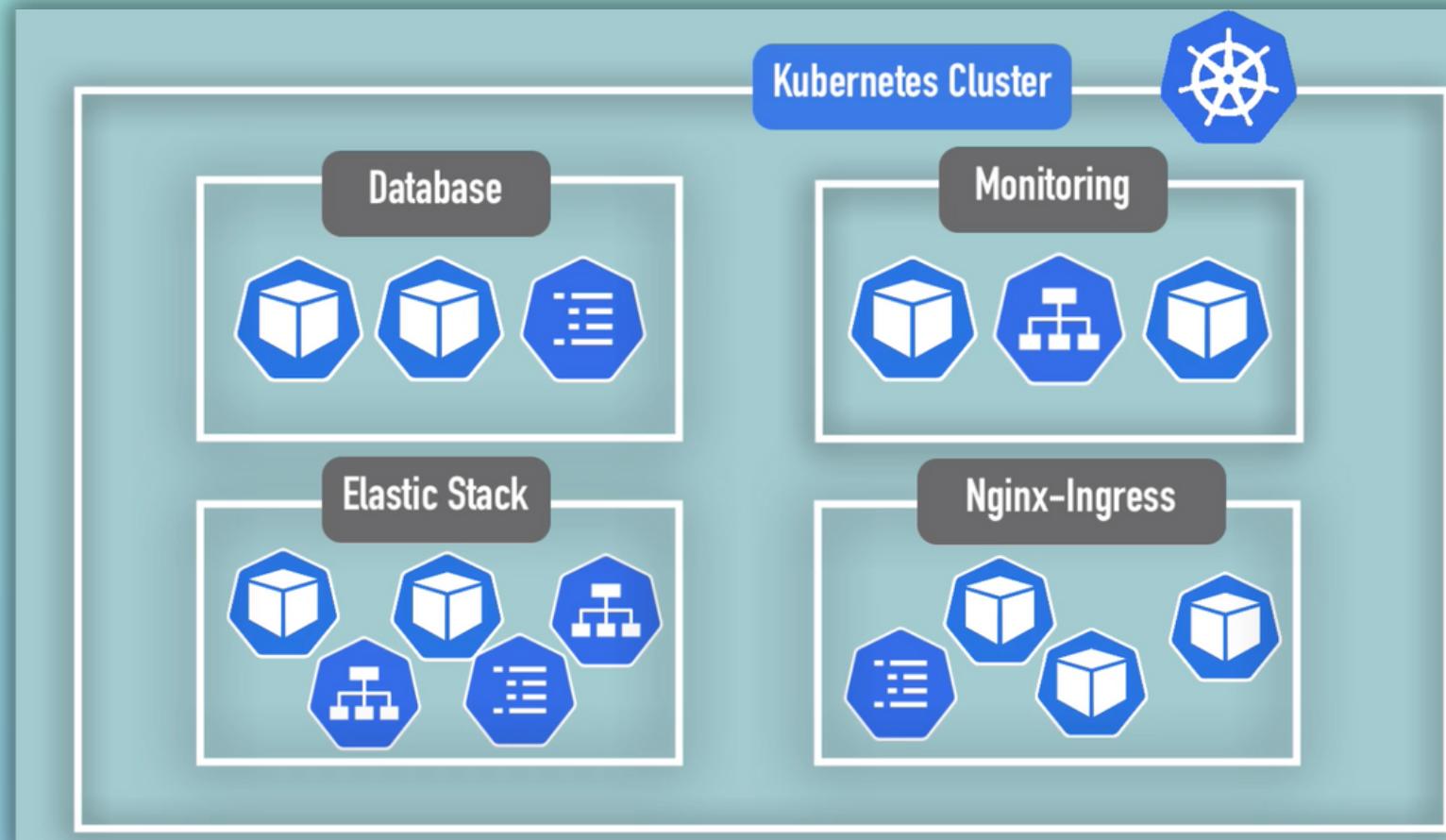
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-configmap
  namespace: my-namespace
data:
  db_url: mysql-service.database
```

# Namespaces - 2

**Use Cases** for when to use namespaces:

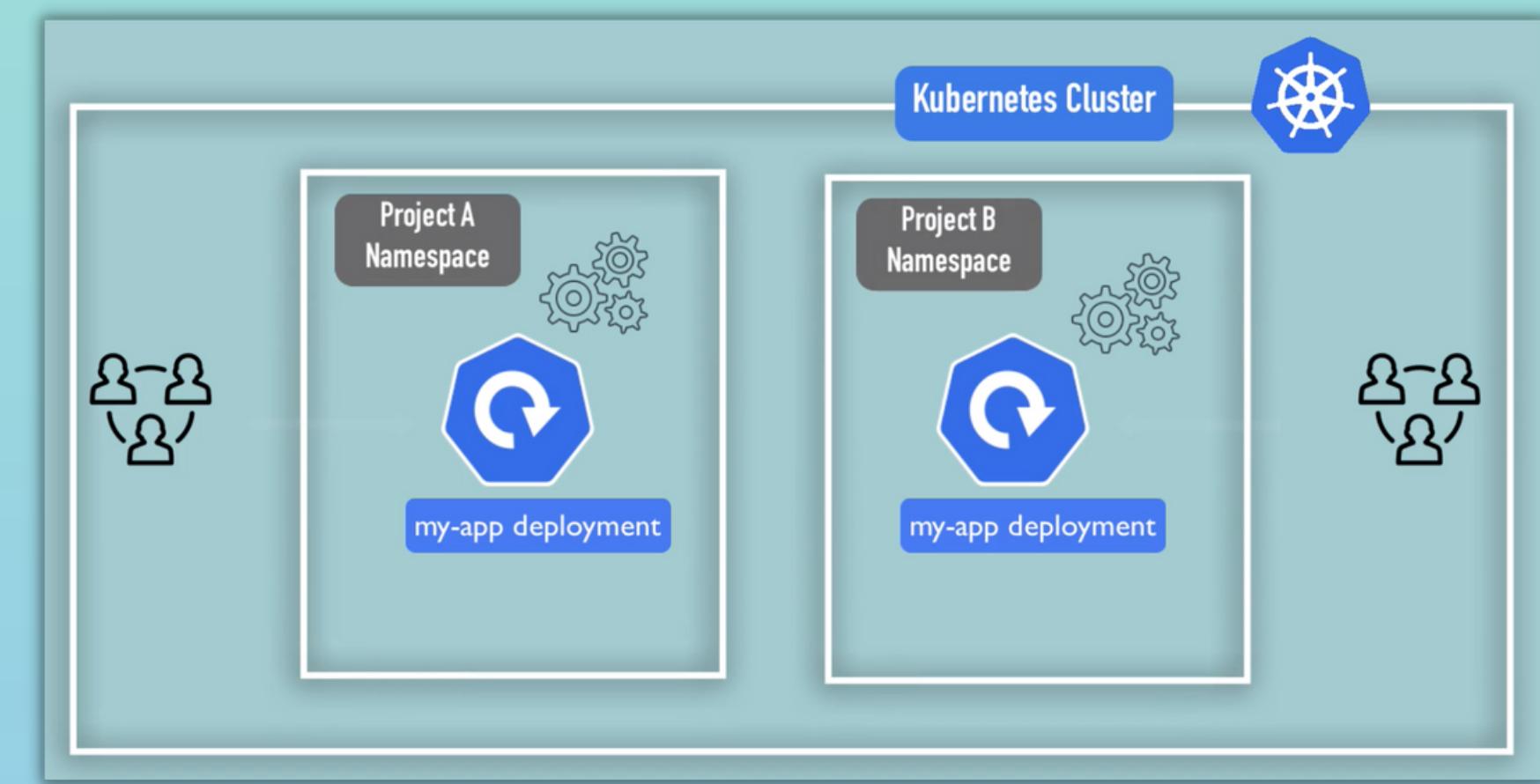
## 1 - Group resources logically

- Instead of having all in the "default" namespace



## 2 - Isolate team resources

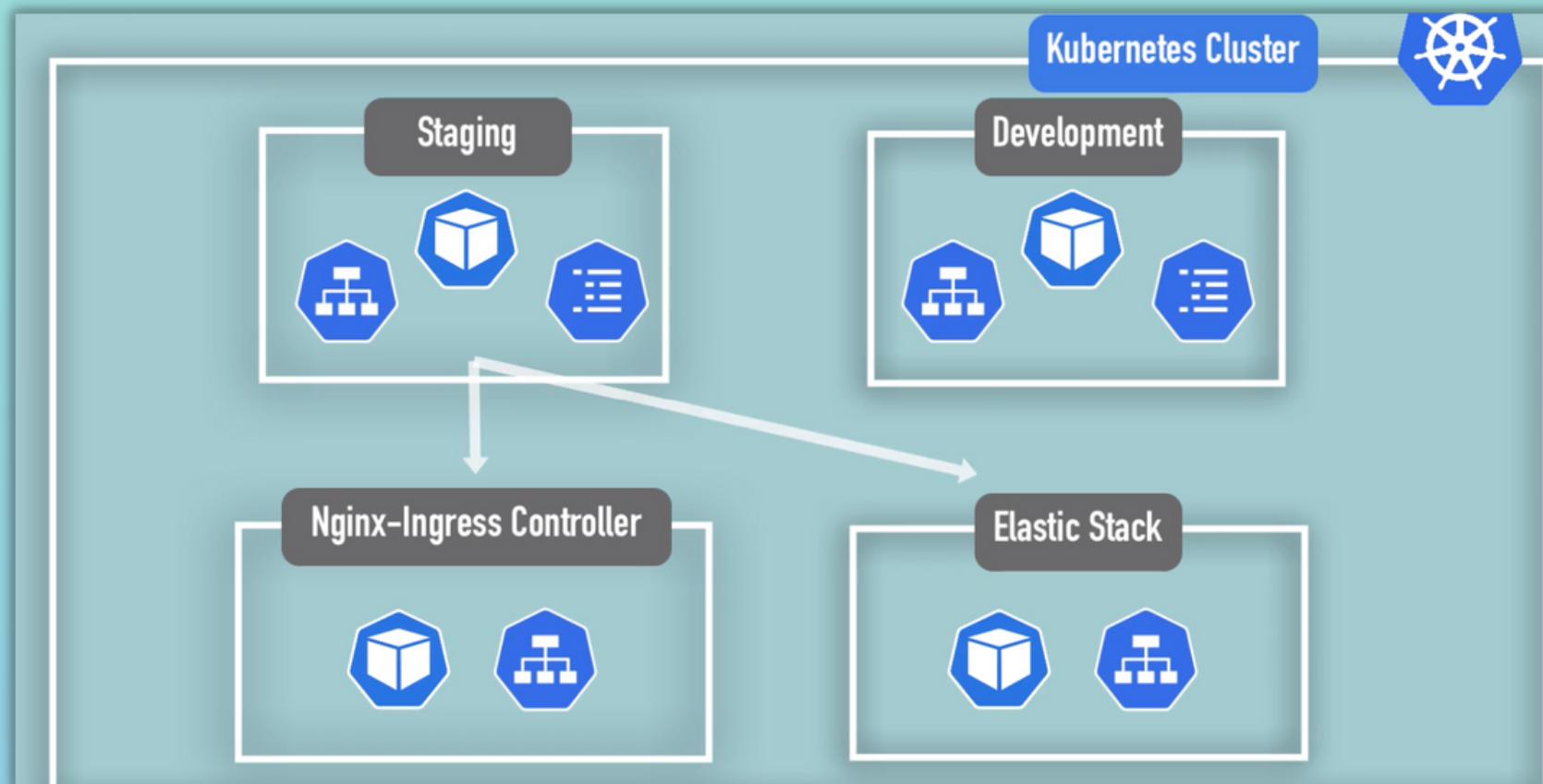
- To avoid conflicts



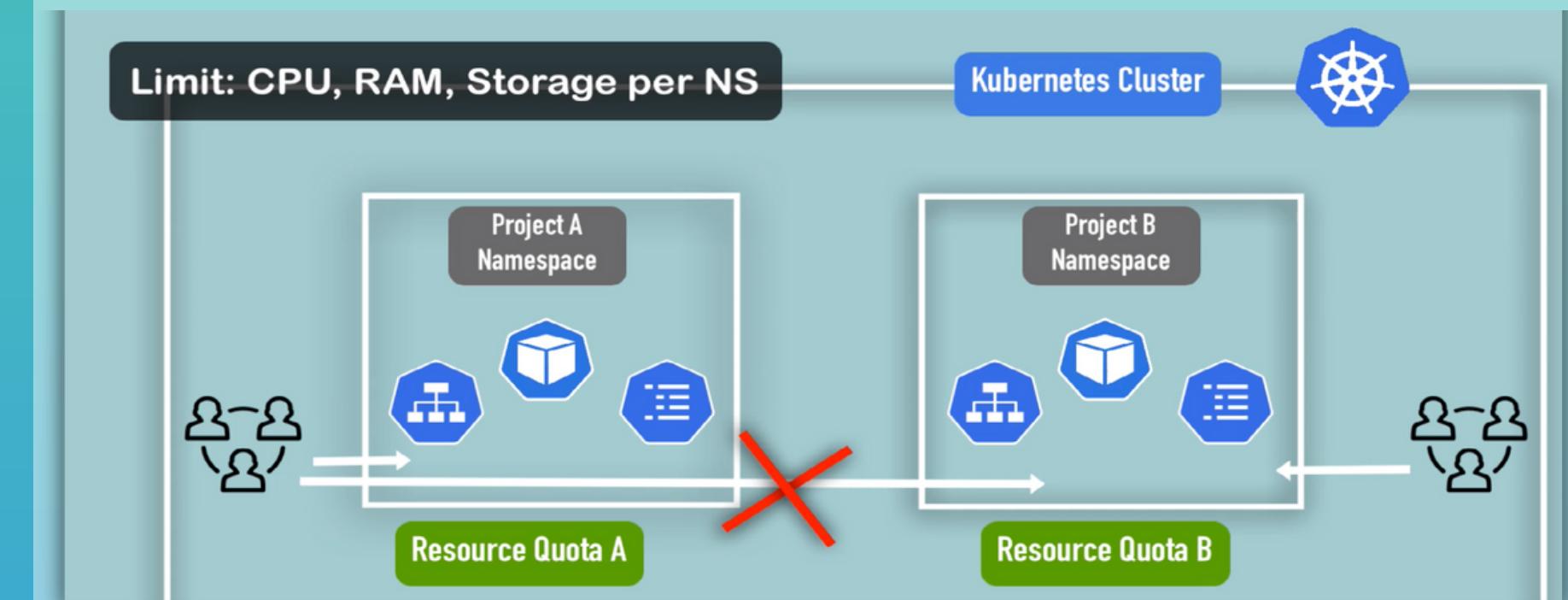
# Namespaces - 3

**Use Cases** for when to use namespaces:

3 - Share resources between different environments



4 - Limit permissions and compute resources for teams

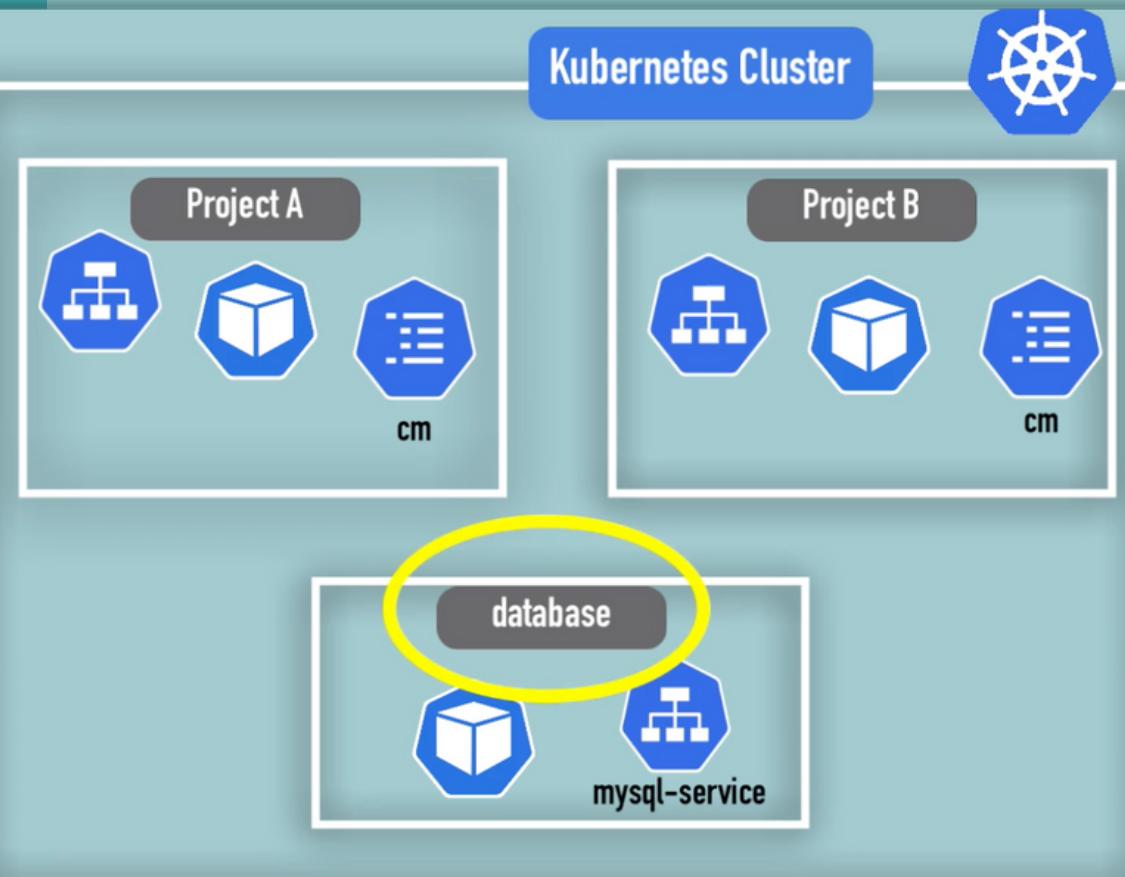


# Namespaces - 4

- There are resources, which can't be created within a namespace, called cluster-wide resources:

## Namespaced resources

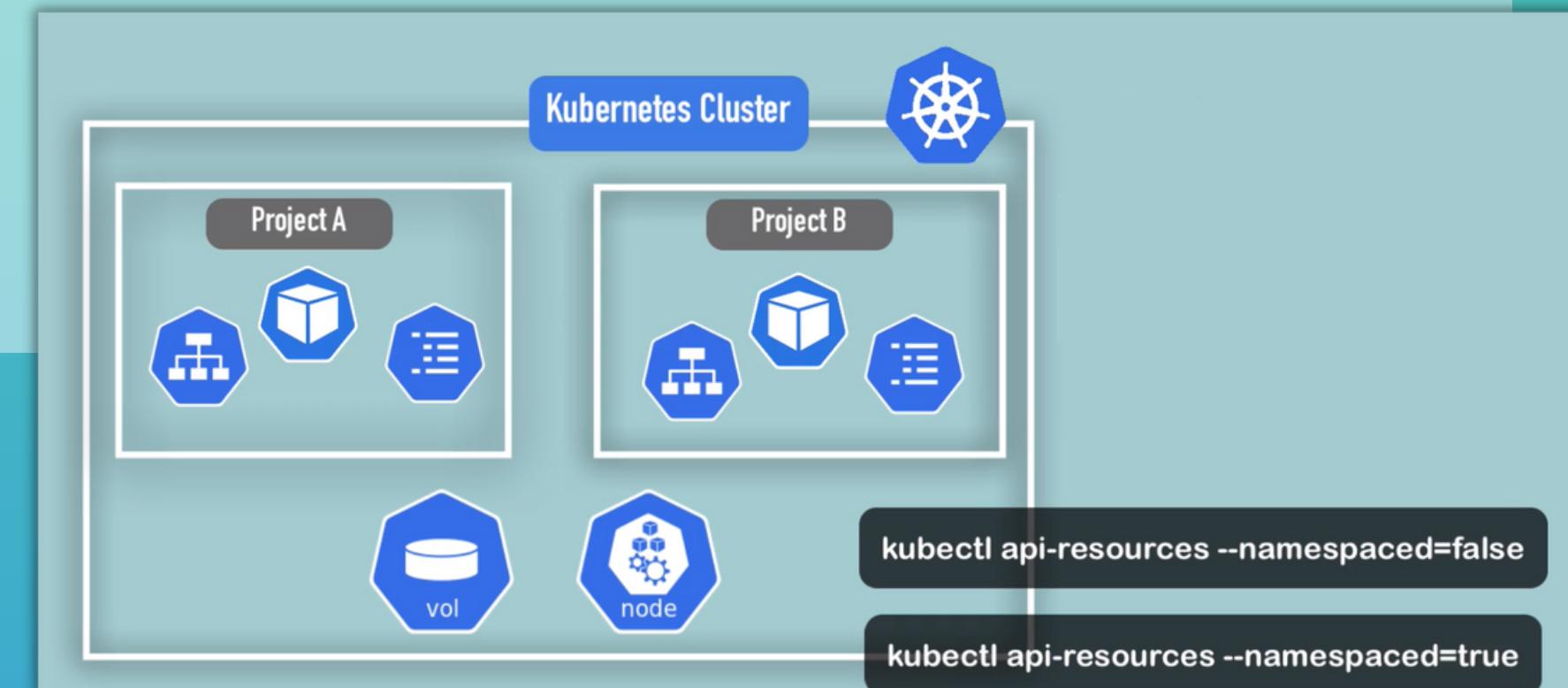
- Most K8s resources (e.g. pods, services, etc.) are in some namespaces
- Access service in another namespace:



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-configmap
data:
  db_url: mysql-service.database
```

## Cluster-wide resources

- Live globally in a cluster, you can't isolate them
- Low-level resources, like Volumes, Nodes

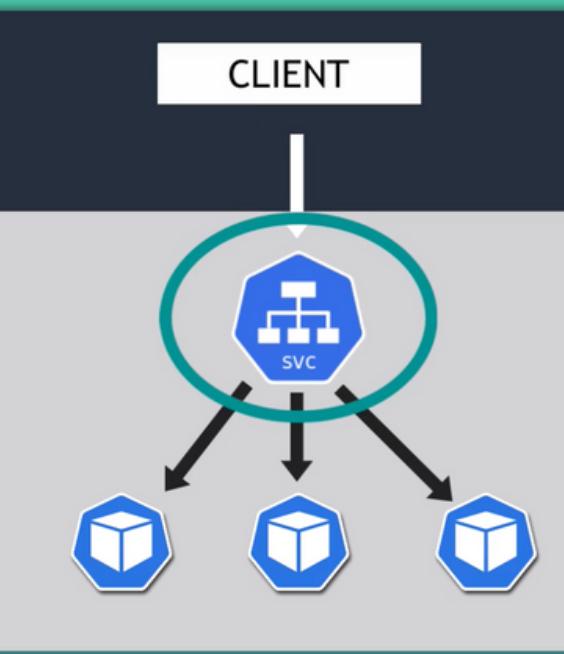


# Deep Dive into Kubernetes Services

# Kubernetes Services - 1

- Abstract way to expose an application running on a set of Pods
- Different types of services:

3 Service type attributes



## Why Service?

- ✓ Stable IP address
- ✓ Loadbalancing
- ✓ Loose coupling
- ✓ Within & Outside Cluster

ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ClusterIP
```

NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
```

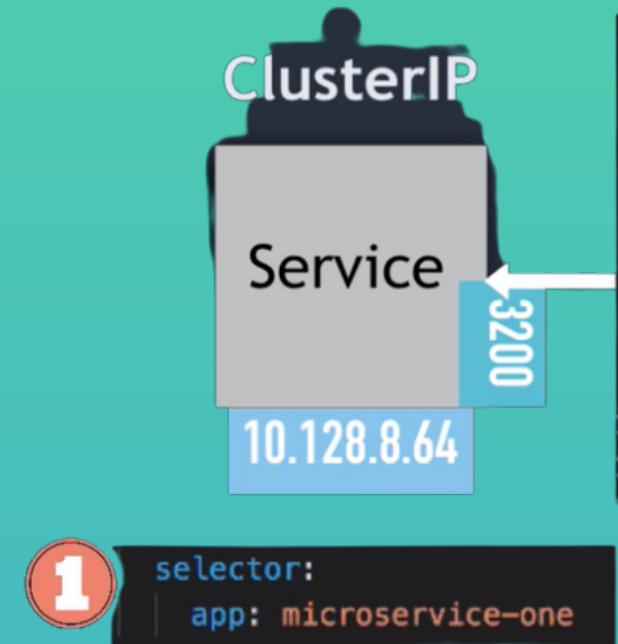
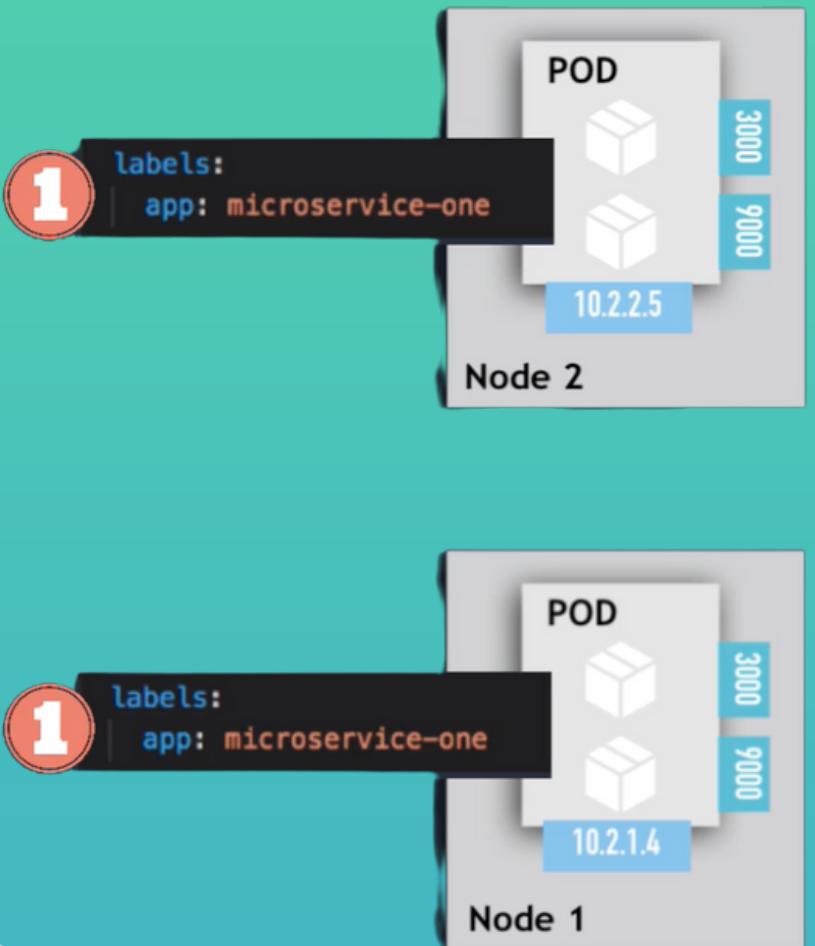
LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
```

- ClusterIP is the **default type**, when you don't specify a type

# Kubernetes Services - 2

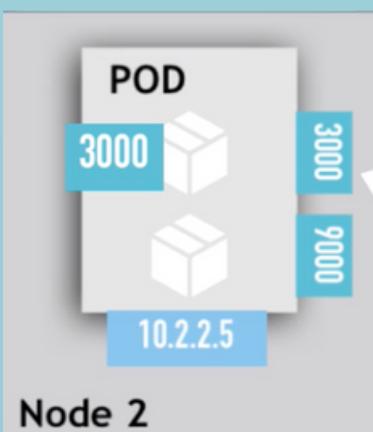
- Service defines a logical set of Pods
- The set of Pods targeted by a Service is **determined by a selector**



```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: microservice-one-service
5 spec:
6   selector:
7     app: microservice-one
8   ports:
9     - protocol: TCP
10    port: 3200
11    targetPort: 3000
```

This creates a new Service named "miroservice-one-service", which **targets port 3000 on any Pod with the *app=microservice-one* label.**

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: microservice-one-service
5 spec:
6   selector:
7     app: microservice-one
8   ports:
9     - protocol: TCP
10    port: 3200
11    targetPort: 3000
```



## PORTS:

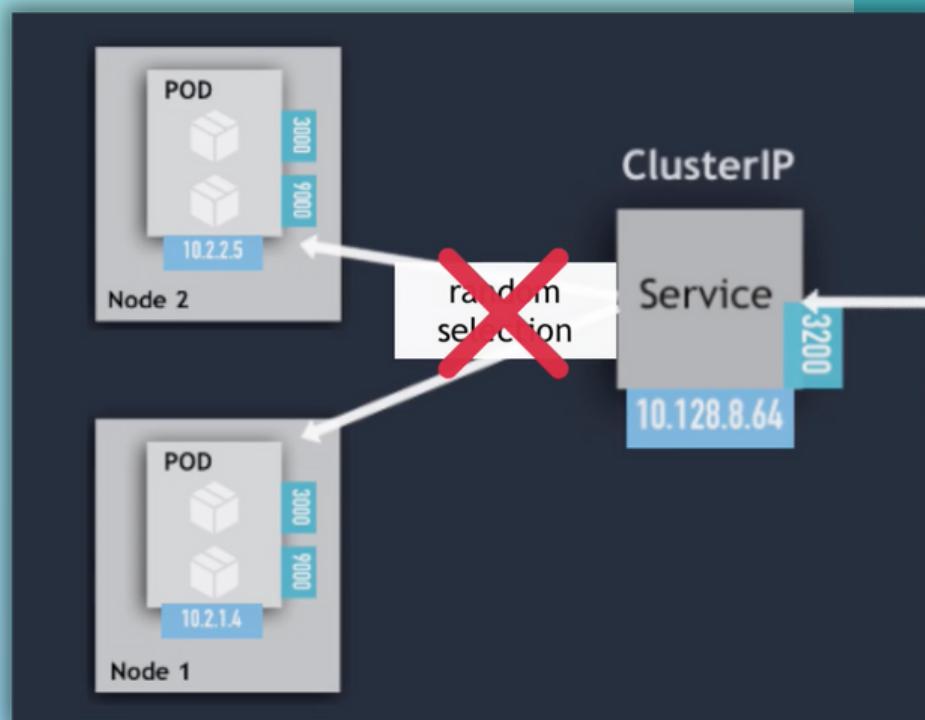
- The service port is arbitrary
- targetPort must **match the port the container is listening at**

# ClusterIP Service & its subtypes

- ClusterIP is an internal service, not accessible from outside the cluster
- All Pods in the cluster can talk to this internal service

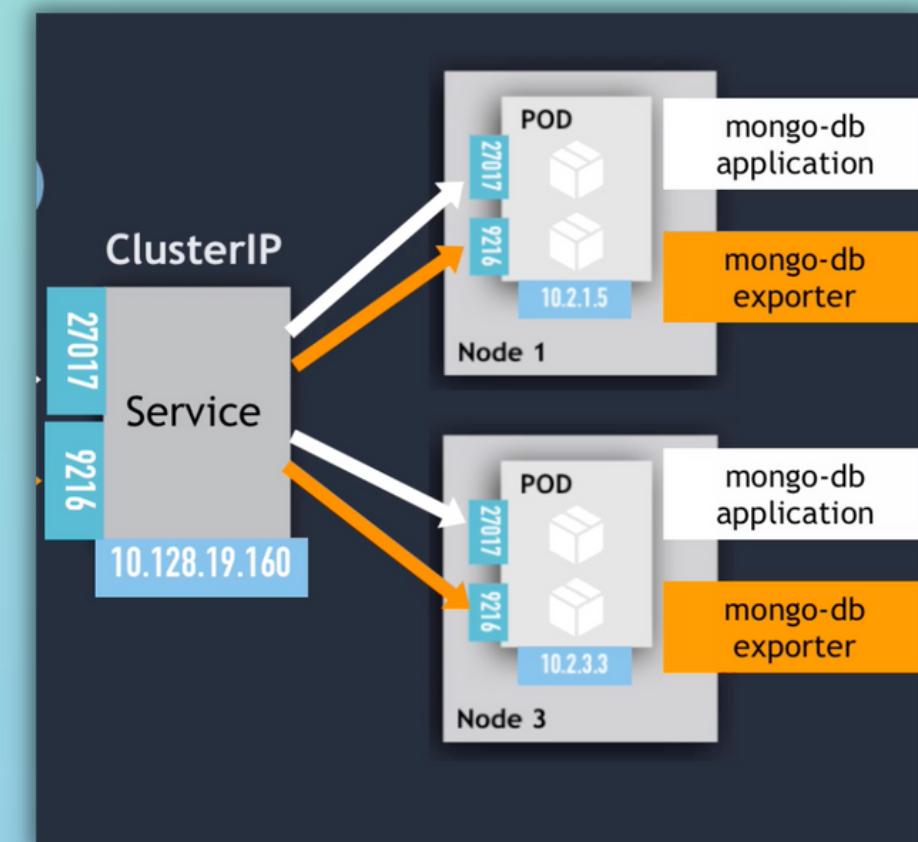
## Headless Internal Service

- When client needs to **communicate with 1 specific Pod directly, instead of randomly selected**
- Use Case: When Pod **replicas are not identical**. For example stateful apps, like when only master is allowed to write to database



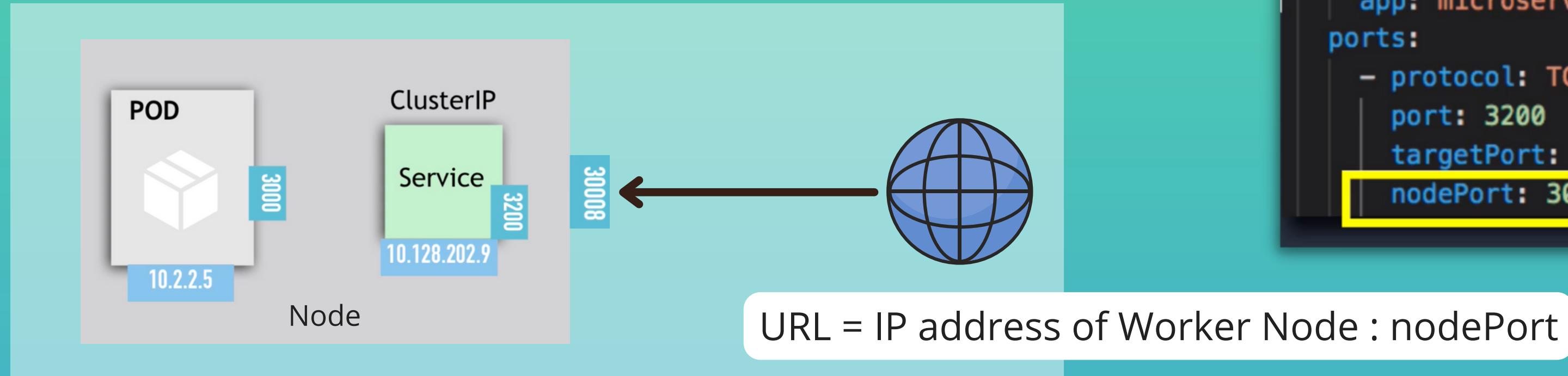
## Multi-Port Internal Service

- When you need to expose more than 1 port
- K8s lets you configure multiple port definitions on a Service
- In that case, you must give all of your ports names so that these are unambiguous



# NodePort Service

- Unlike internal Service, it is accessible directly from outside cluster
- **Exposes the Service on each Node's IP at a static port**



- A ClusterIP Service, to which the NodePort Service routes, is automatically created

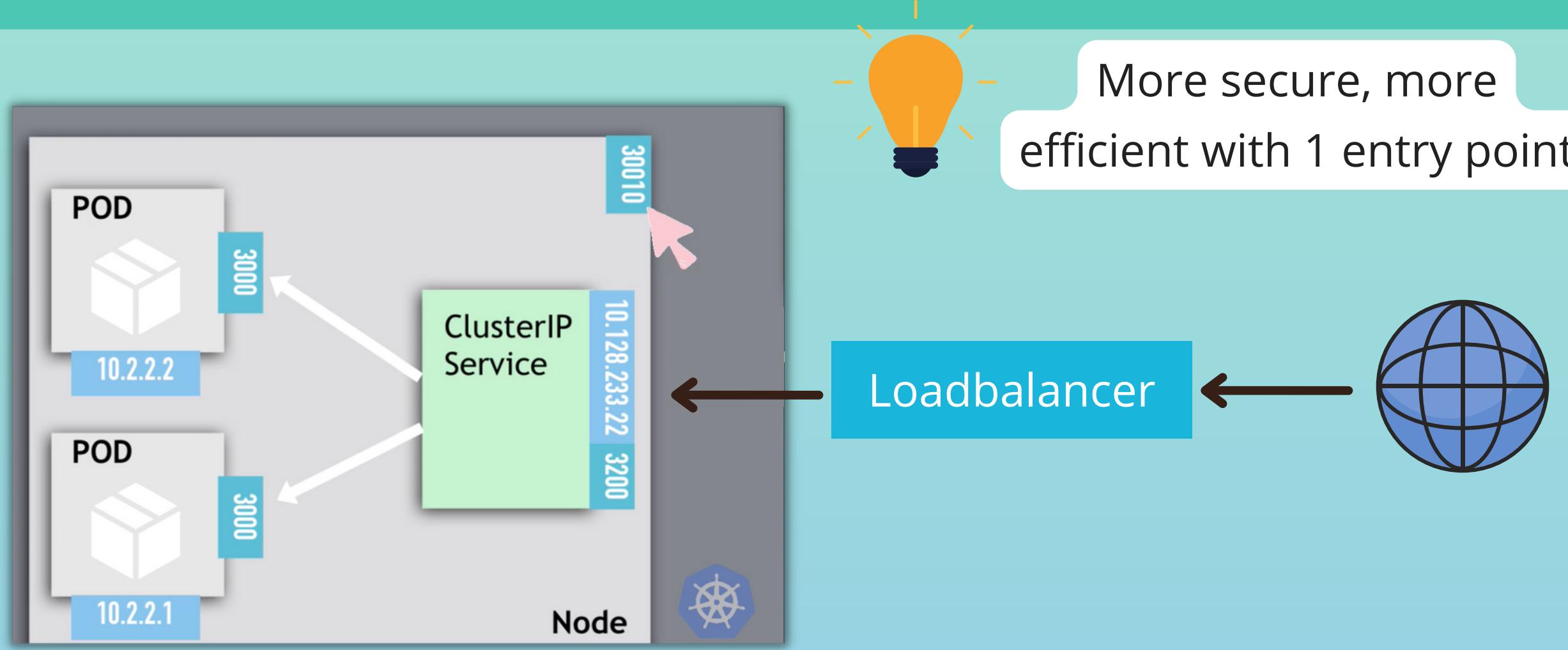
[k8s-services]\$ kubectl get svc	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
	kubernetes	ClusterIP	10.128.0.1	<none>	443/TCP	20m
	mongodb-service	ClusterIP	10.128.204.105	<none>	27017/TCP	10m
	mongodb-service-headless	ClusterIP	None	<none>	27017/TCP	2m8s
	ms-service-nodeport	NodePort	10.128.202.9	<none>	3200:30008/TCP	8s



**Not Secure:** External traffic has access to fixed port on each Worker Node

# Loadbalancer Service

- Exposes the Service **externally using a cloud provider's load balancer**
- NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created

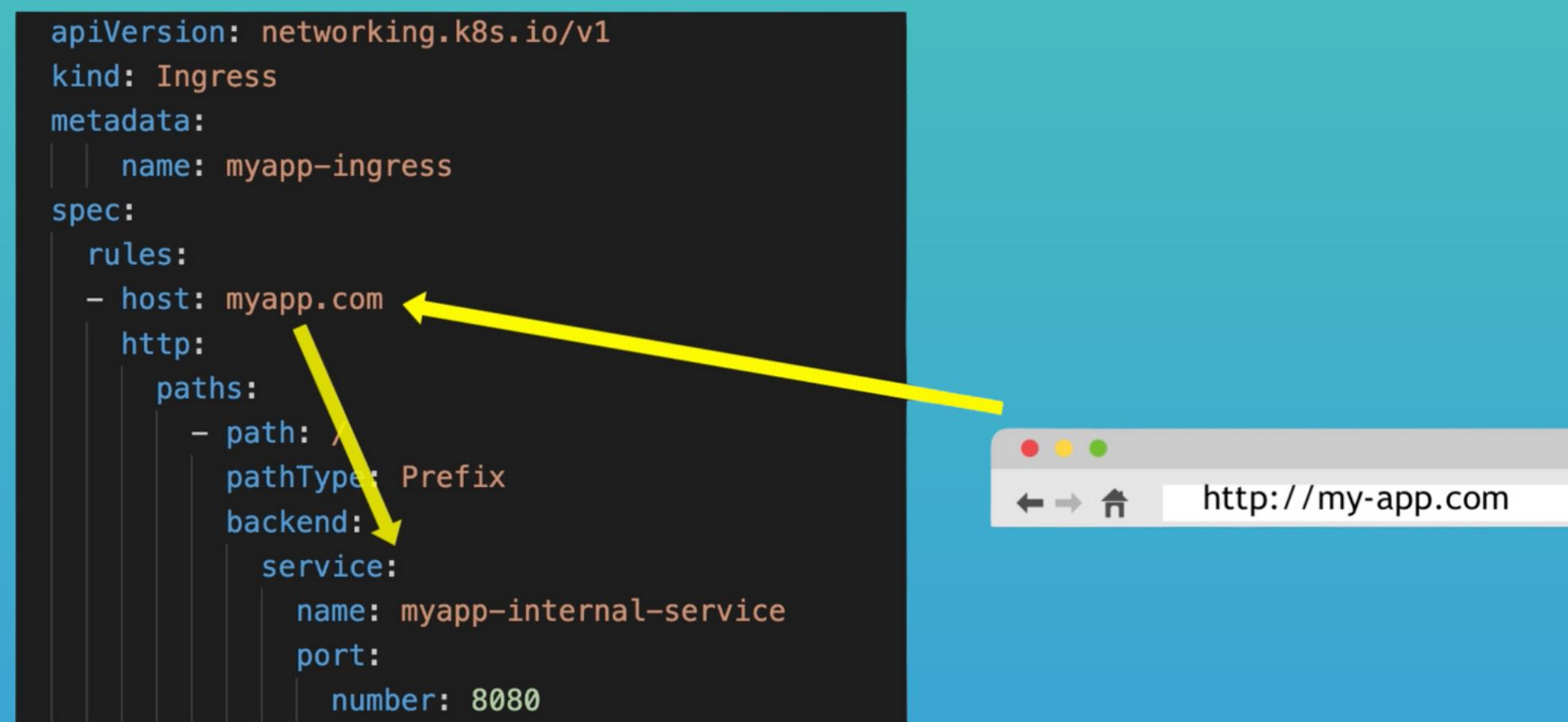


```
spec:  
  type: LoadBalancer  
  selector:  
    app: microservice-one  
  ports:  
    - protocol: TCP  
      port: 3200  
      targetPort: 3000  
      nodePort: 30010
```

[k8s-services]\$ kubectl get svc	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
	kubernetes	ClusterIP	10.128.0.1	<none>	443/TCP
	mongodb-service	ClusterIP	10.128.204.105	<none>	27017/TCP
	monaodb-service-headless	ClusterIP	None	<none>	27017/TCP
	ms-service-loadbalancer	LoadBalancer	10.128.233.22	172.104.255.5	3200:30010/TCP
	ms-service-nodeport	NodePort	10.128.202.9	<none>	3200:30008/TCP

# Ingress - 1

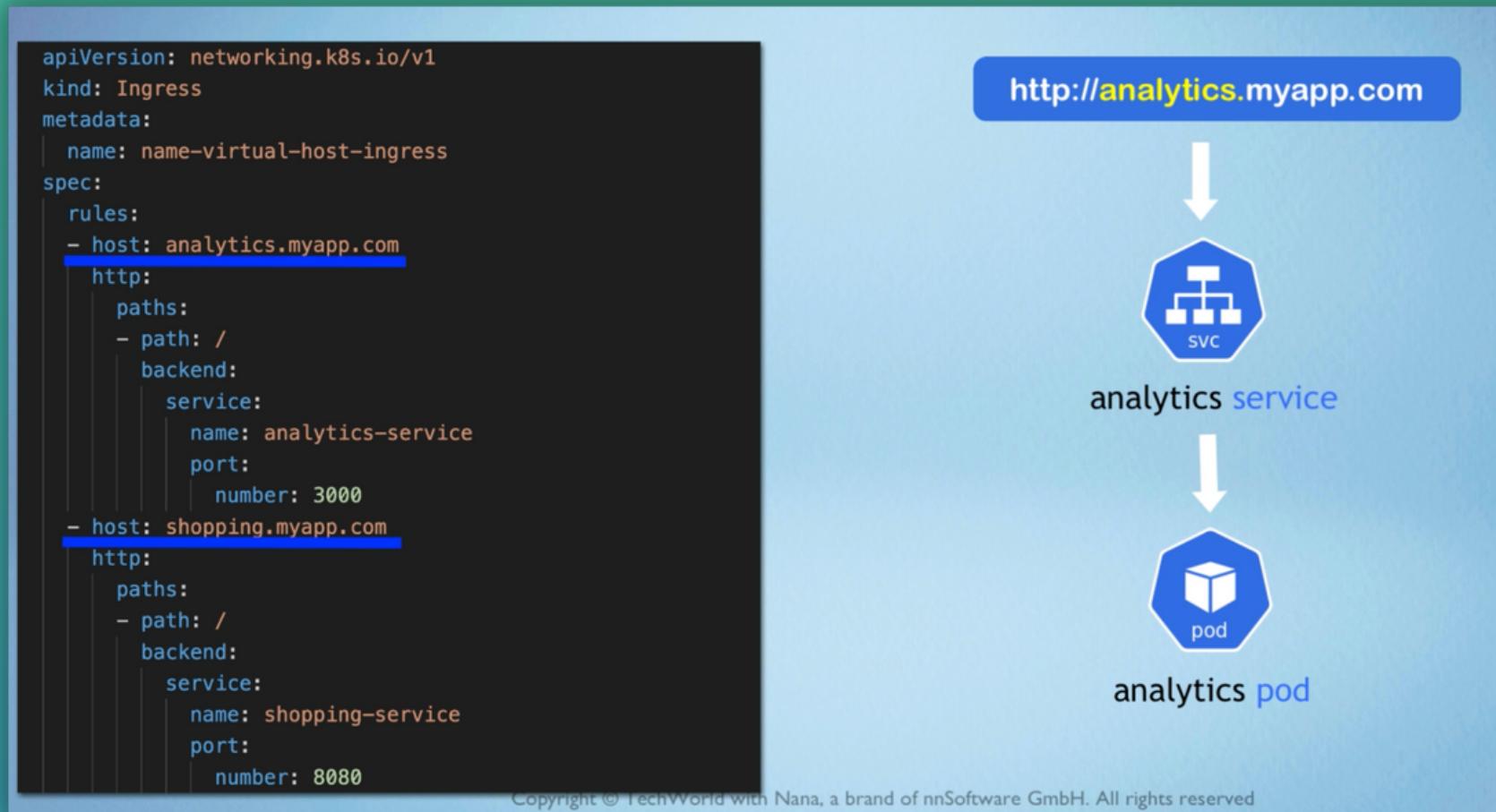
- External Services are a way to access applications in K8s from outside
- **In production a better alternative is Ingress!**
- Not a Service type, but acts as the entry point for your cluster
- **More intelligent and flexible:** Let's you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address



# Ingress - 2

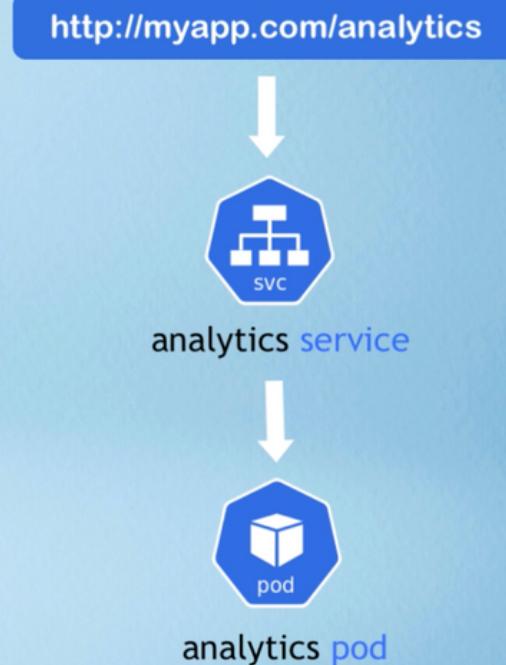
- Configure multiple paths for same host:

- Configure multiple sub-domains or domains:



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: myapp.com
      http:
        paths:
          - path: /analytics
            backend:
              service:
                name: analytics-service
                port:
                  number: 3000
          - path: /shopping
            backend:
              service:
                name: shopping-service
                port:
                  number: 8080
```

Copyright © TechWorld with Nana, a brand of nnSoftware GmbH. All rights reserved.



- Configure TLS

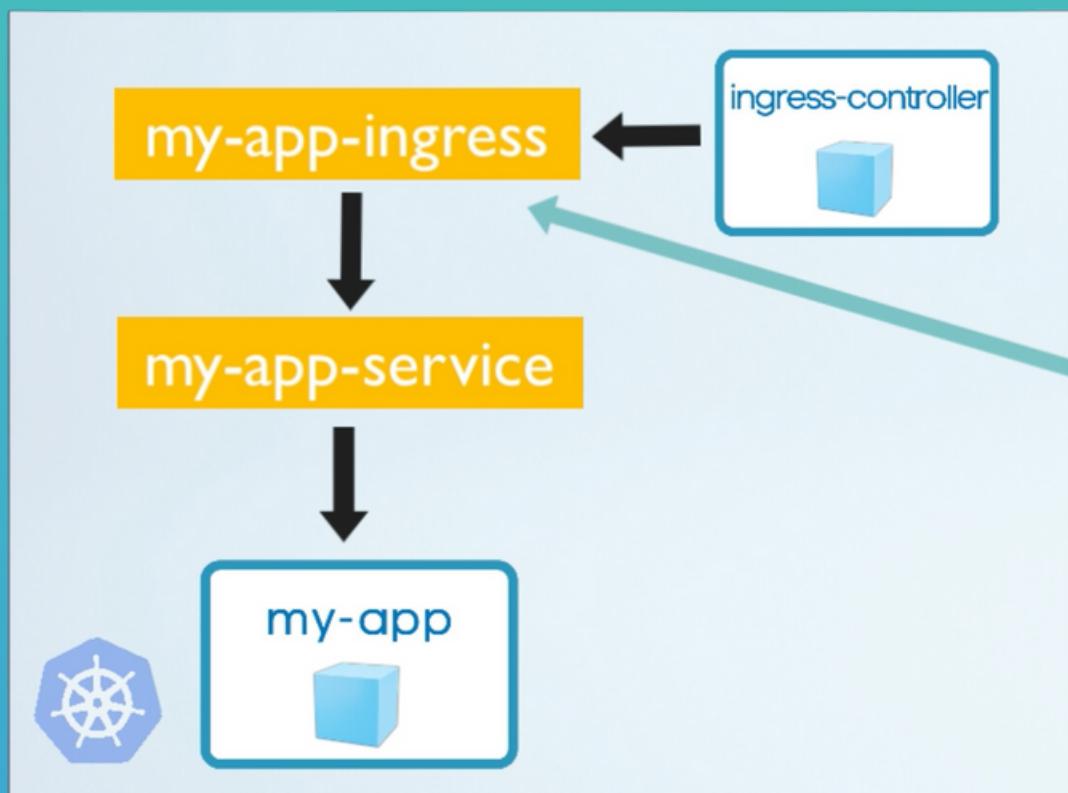
Certificate - https://

```
spec:
  tls:
    - hosts:
      - myapp.com
      secretName: myapp-secret-tls
    rules:
      - host: myapp.com
        http:
          paths:
            - path: /analytics
              backend:
                service:
                  name: myapp-internal-service
                  port:
                    number: 8080
```

# Ingress - 3

## How to configure Ingress in your cluster

- You need an **implementation for Ingress**
- Which is Ingress Controller



```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myapp-internal-service
            port:
              number: 8080
```

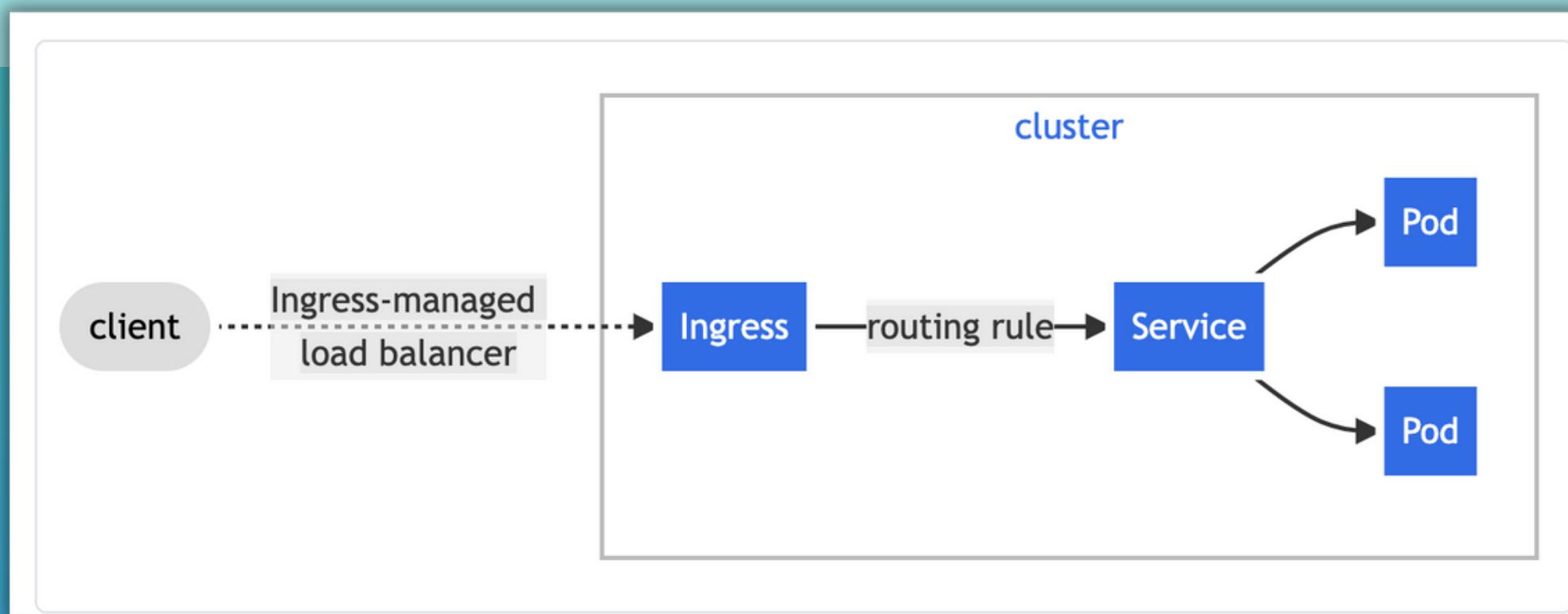
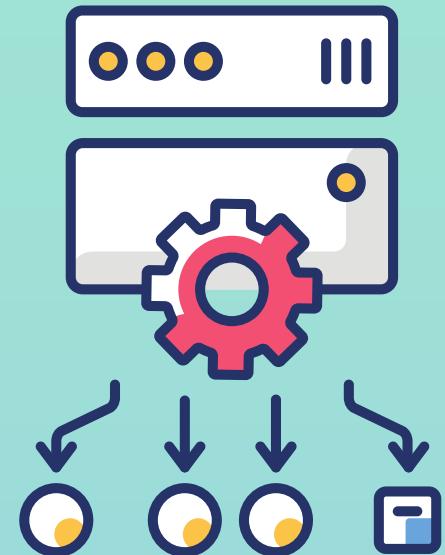
## Ingress Controller

- Evaluates all the rules
- Manages redirections
- Entry point to the cluster
- Many third-party implementations
- K8s Nginx Ingress Controller

# Ingress - 4

Before Ingress Controller you still need 1 load balancer

- **Option 1 - Cloud service provider:** have out-of-the-box K8s solutions
- **Option 2 - Bare Metal:** You need to configure some kind of entry point. Either inside the cluster or outside as separate server



Source: <https://kubernetes.io/docs/concepts/services-networking/ingress/>

# Deep Dive into Kubernetes Volumes

# Volumes - 1

- K8s offers no data persistence out of the box



- At its core, a **volume is a directory** (with some data in it), **which is accessible to the containers in a Pod**
- K8s supports **many types of volumes**
- Ephemeral volume types have a lifetime of a Pod, persistent volumes exist beyond the lifetime of a Pod

- In this lecture we talk about **persistent volumes**, with these storage requirements:

## Storage Requirements

- Storage that **doesn't depend on pod lifecycle**
- Storage must be **available on all Nodes**
- Storage needs to **survive even if cluster crashes**



# Volumes - 2

- The way to persist data in K8s using Volumes is with these 3 resources:



## Persistent Volume (PV)

- **Storage** in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes



## Storage Class (SC)

- SC provisions PV dynamically when PVC claims it



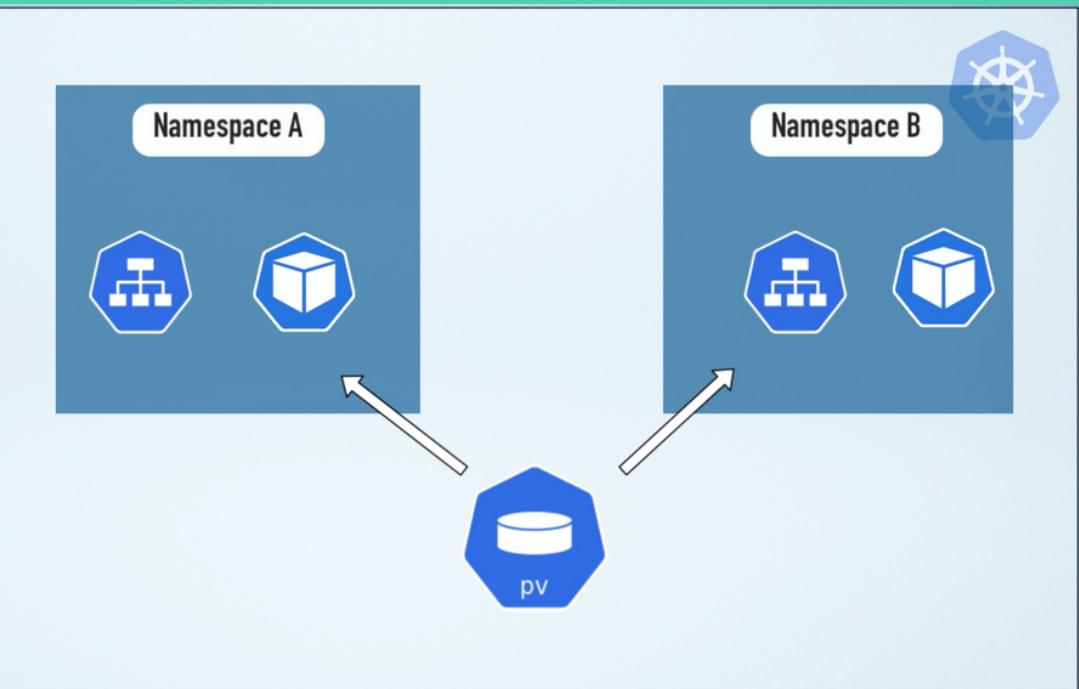
## Persistent Volume Claim (PVC)

- A **request for storage by a user**
- Similar to Pods. While Pods consume node resources, PVCs consume PV resources

# Persistent Volume - 1



- Persistent Volumes are **NOT namespaced**, so PV resource is accessible to the whole cluster:



- Depending on storage type, **spec attributes differ**
- In official documentation you can find a complete list of storage backends supported by K8s:

Types of volumes

- awsElasticBlockStore (removed)
- azureDisk (removed)
- azureFile (deprecated)
- cephfs
- cinder (removed)
- configMap
- downwardAPI
- emptyDir
- fc (fibre channel)
- gcePersistentDisk (deprecated)
- gitRepo (deprecated)
- glusterfs (removed)
- hostPath
- iscsi
- local

- Configuration Example

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-name
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.0
  nfs:
    path: /dir/path/on/nfs/server
    server: nfs-server-ip-address
```

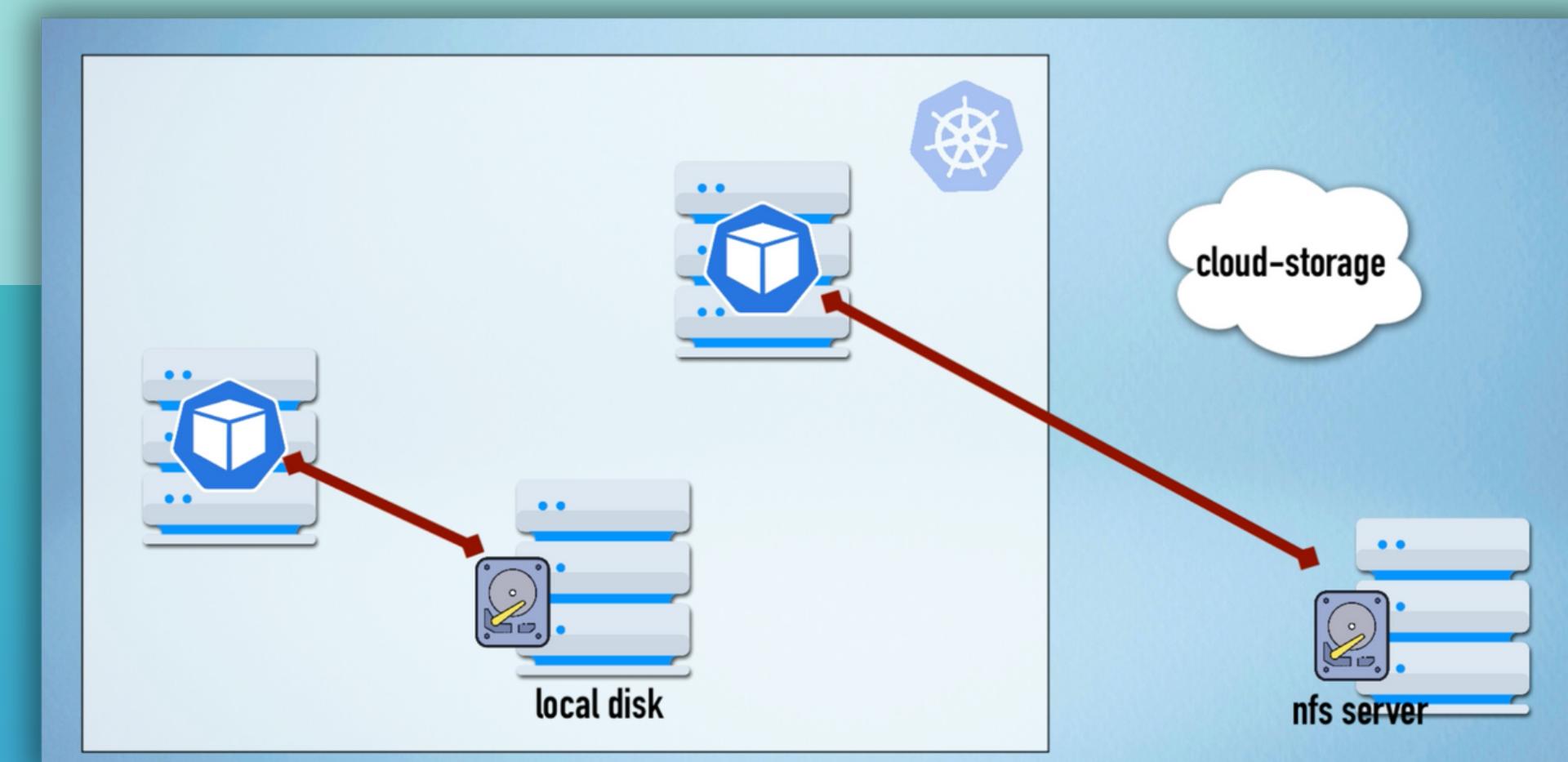
# Persistent Volume - 2

## Local vs Remote Volume Types

- Each volume type has its own use case!
- **Local volume types** violate 2nd and 3rd requirement for data persistence:
  - ✖ Being tied to 1 specific Node
  - ✖ Not surviving cluster crashes



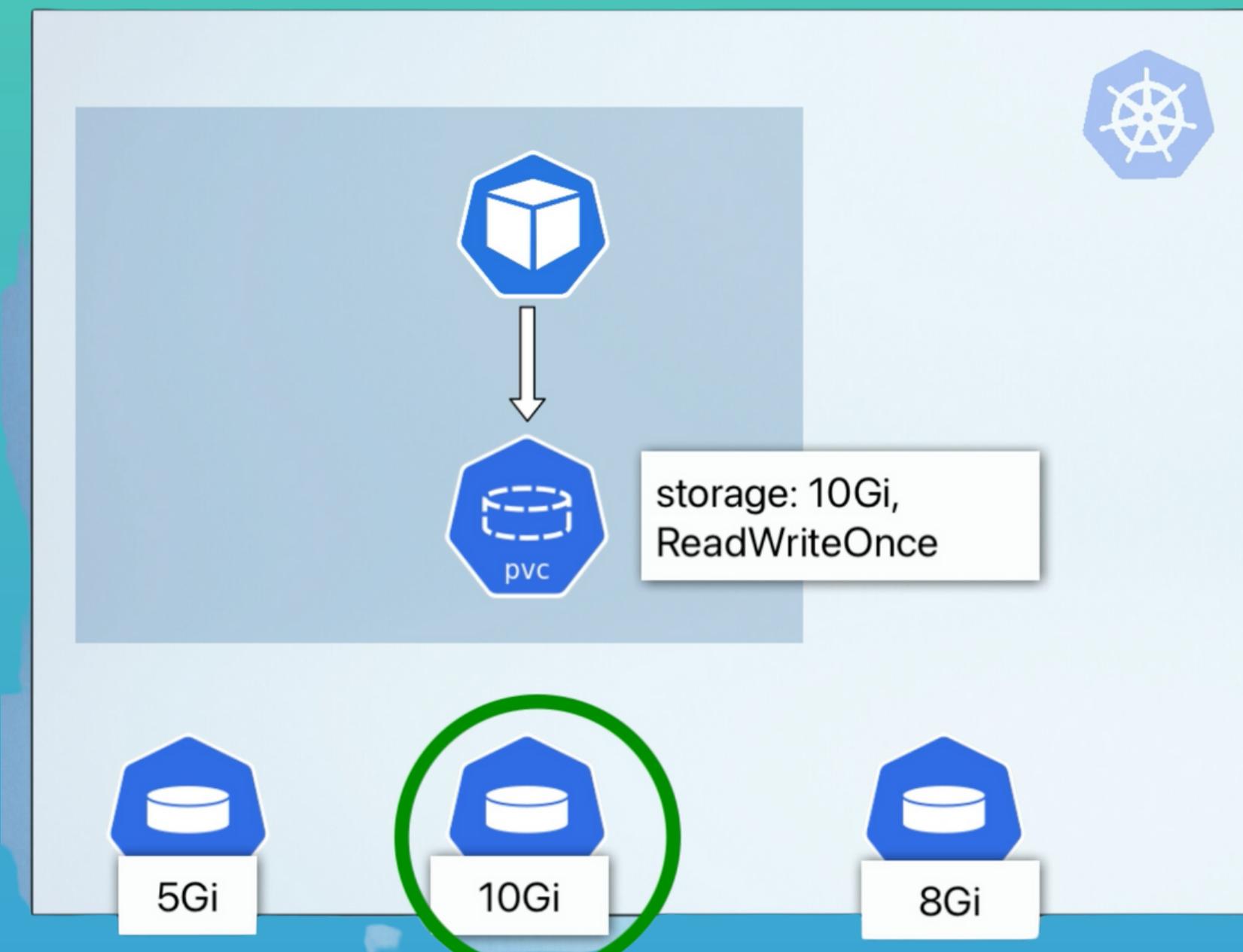
For DB persistence use **remote storage!**



# Persistent Volume Claim



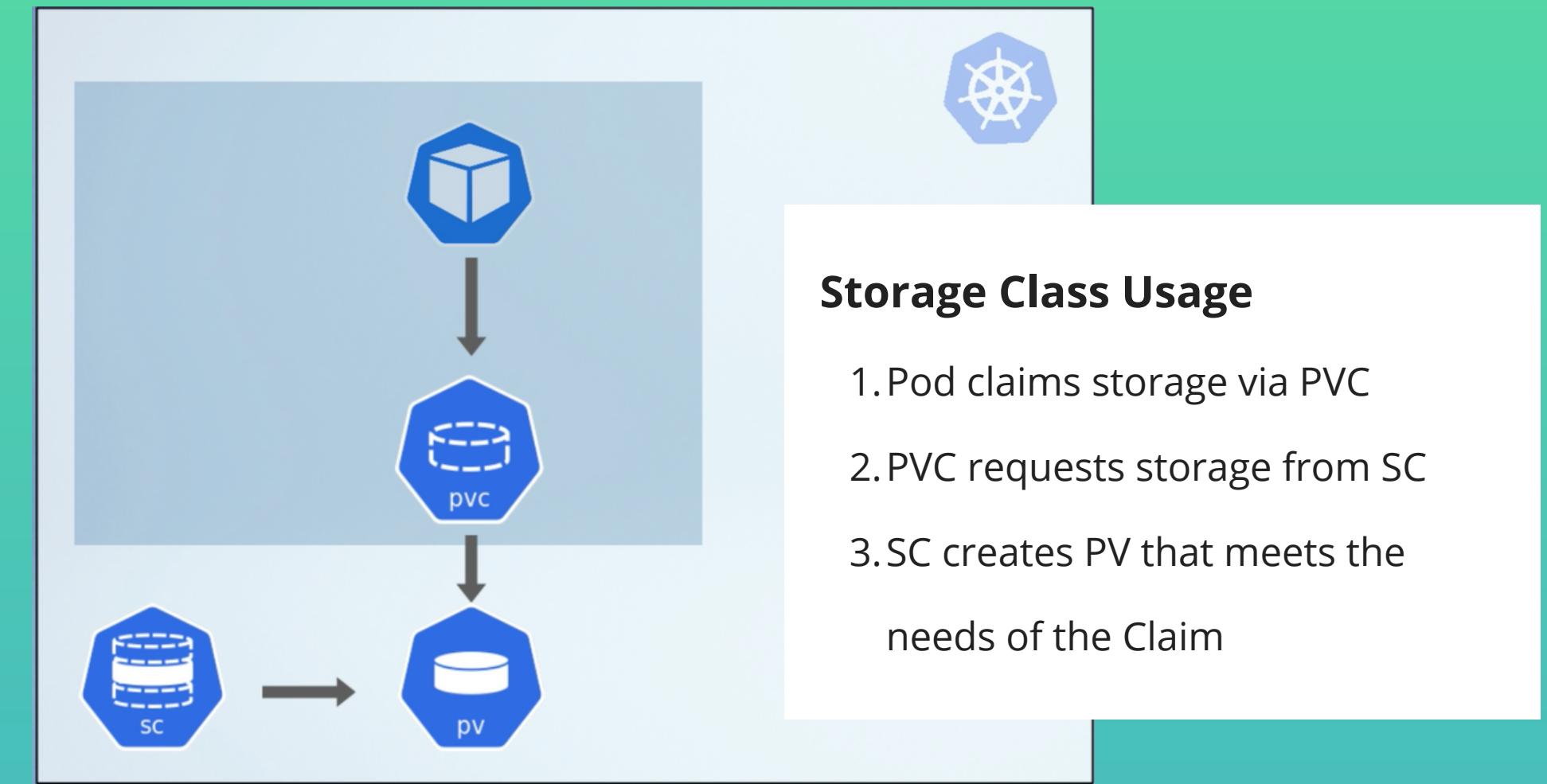
- Request for storage by a user
- Claims can **request specific size and access modes** (e.g. they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany).



```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-name
spec:
  storageClassName: manual
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

# Storage Class

- Provisions PV dynamically



- **StorageBackend is defined in the SC resource**
  - via "provisioner" attribute
  - each storage backend has own provisioner
  - internal provisioner - "kubernetes.io"
  - external provisioner
  - configure parameters for storage we want to request for PV



```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: storage-class-name
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```



# Deep Dive into StatefulSets

# StatefulSet

- Used to **manage stateful applications**, like databases
- Manages the deployment and scaling of a set of Pods and **provides guarantees about the ordering and uniqueness of these Pods**

## Stateless Applications

- Doesn't depend on previous data
- Deployed using Deployment



## Stateful Applications

- Update data based on previous data
- Query data
- Depends on most up-to-date data/state

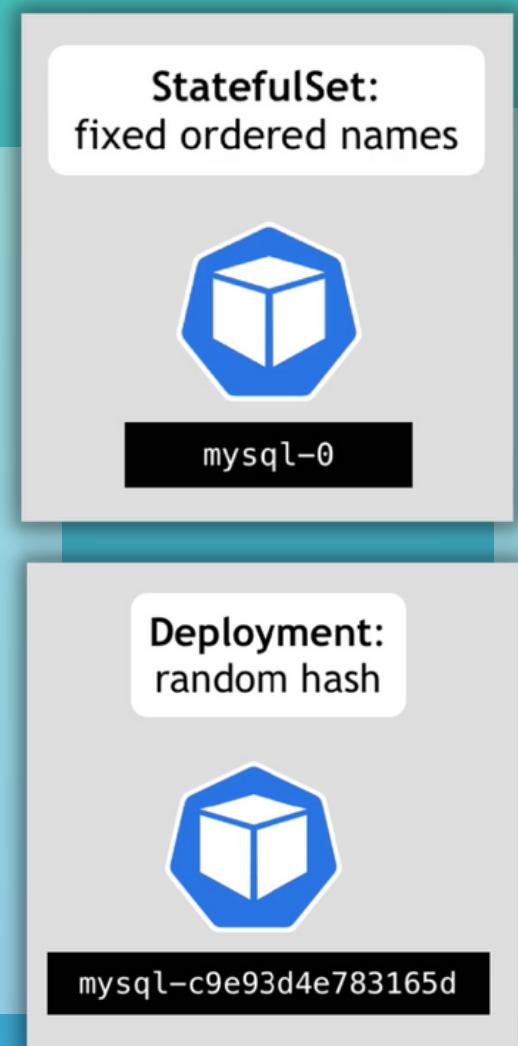
Both manage Pods based on container specification

# Deployment vs StatefulSet

- Unlike a Deployment, a StatefulSet maintains a **sticky identity** for each of their Pods
- These pods are created from the **same spec, but are not interchangeable**: each has a persistent identifier that it maintains across any rescheduling

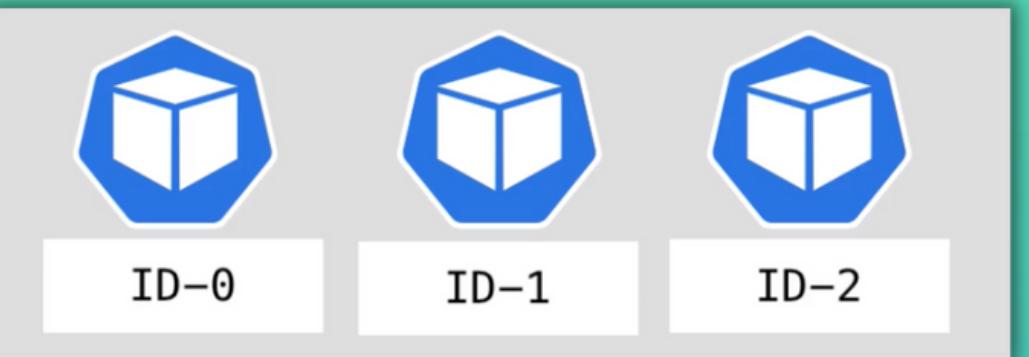
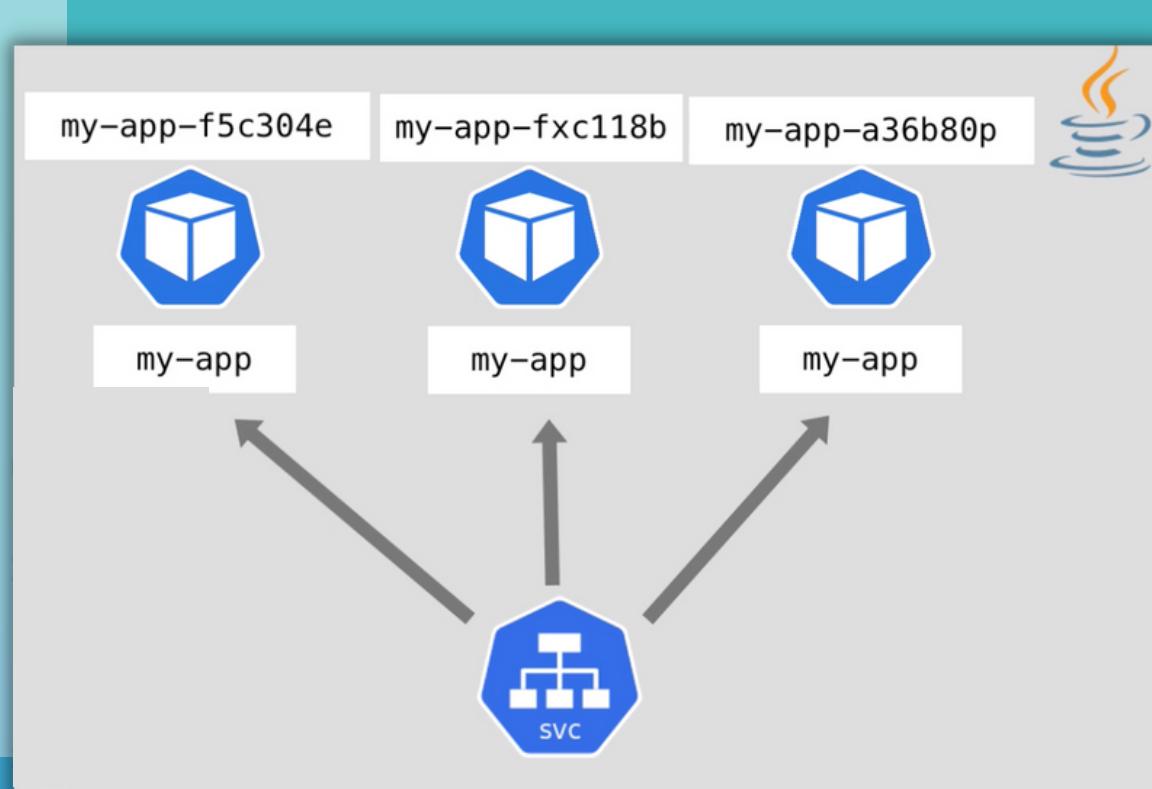
## Pods created from StatefulSet

- More difficult
- Can't be created/deleted at same time
- Can't be randomly addressed



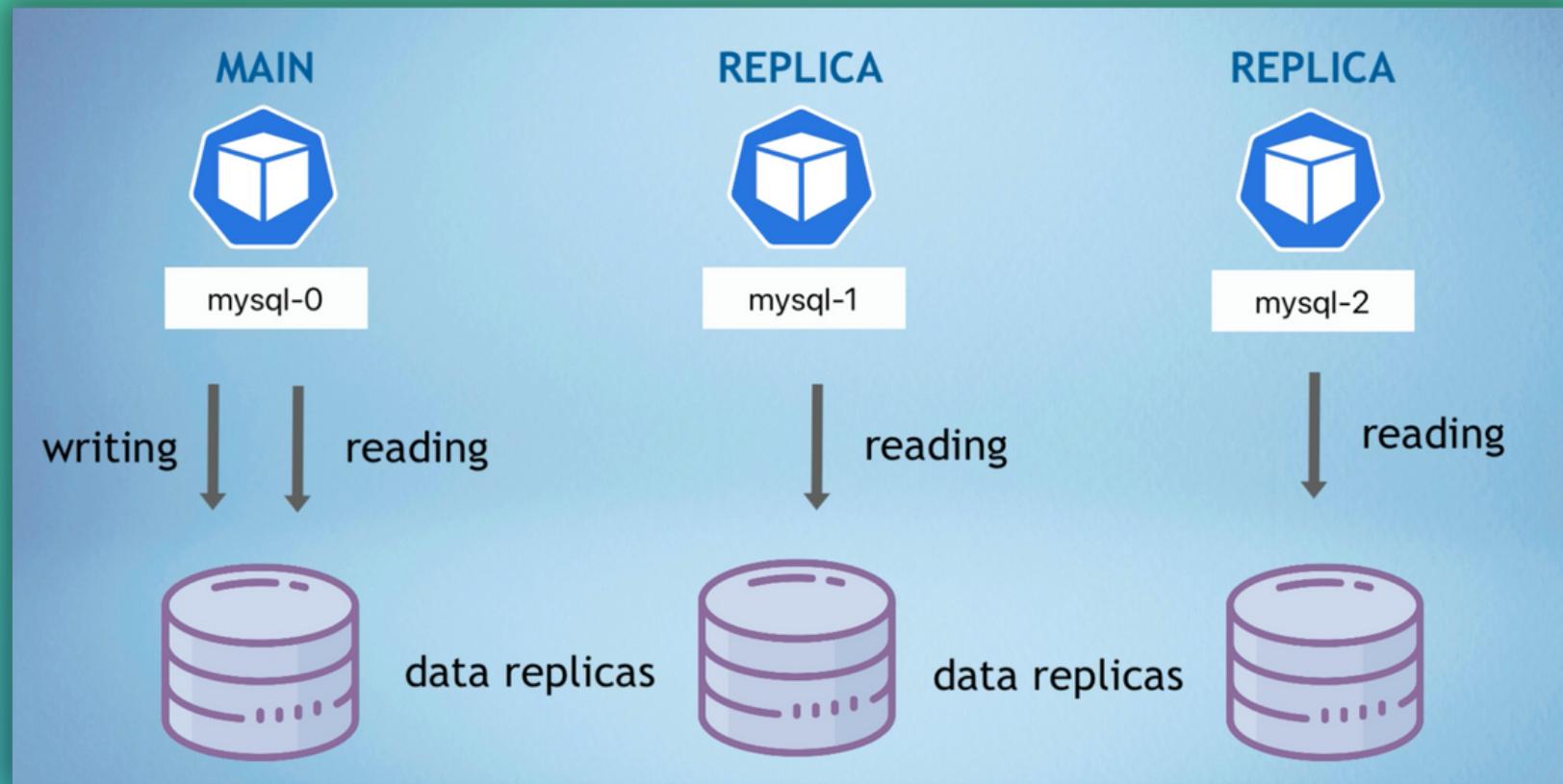
## Pods created from Deployment

- Identical and interchangeable
- Created in random order with random hashes
- 1 Service that load balances to any Pod



`$(statefulset name)-$(ordinal)`

# Scaling database applications



- Each replica has its own storage
- These storages are constantly synchronized

- Only 1 replica, can make changes
- So replicas are not identical



# Managed K8s Service

# Kubernetes on Cloud platform

**2 options** to create a Kubernetes cluster on a cloud platform

## Create own cluster from scratch

- ✖ You need to manage everything yourself
- ✖ Not practical, when you want to setup things fast and easy



## Use Managed K8s Service

- You only care about Worker Nodes
- Everything pre-installed
- Control Plane Nodes created and managed by cloud provider
- You only pay for the Worker Nodes
- Use cloud native load balancer for Ingress controller
- Use cloud storage
- Less effort and time



# Example Managed Kubernetes Services

- **AWS:** Elastic Kubernetes Service (EKS)
- **Azure:** Azure Kubernetes Service (AKS)
- **Google:** Google Kubernetes Engine (GKE)
- **Linode:** Linode Kubernetes Engine (LKE)



# Helm Package Manager

# Helm - Package Manager

- Helm is the **package manager for Kubernetes**. Think of it like apt/yum for Kubernetes
- Packages YAML files and distributes them in public and private repositories

## Helm

- Tool that installs and manages K8s applications
- This is done via Charts, so Helm manages these Charts



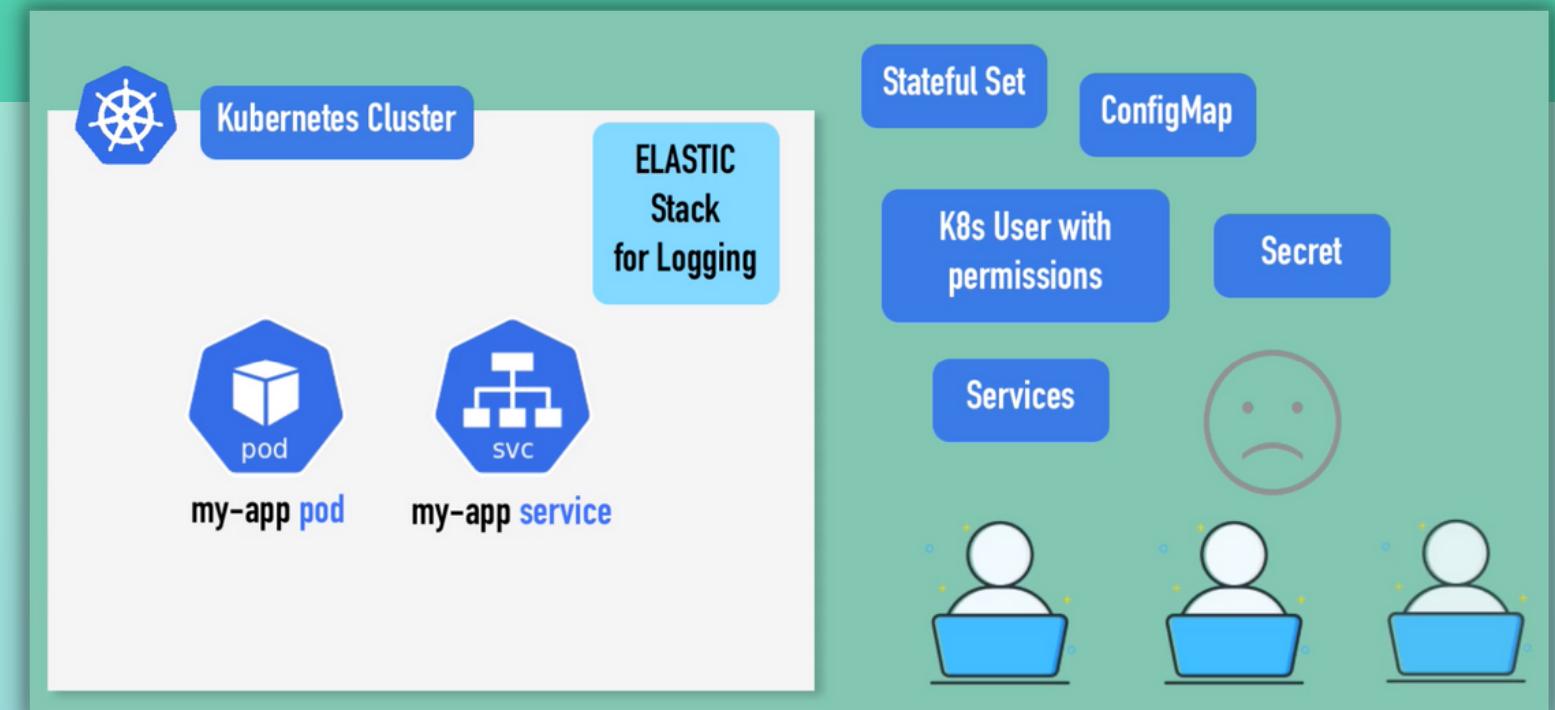
## Helm Chart

- **Helm package** that contains:
  - description of the package *Chart.yaml*
  - 1 or more templates, which contain K8s manifest files
- You can create your own Helm Charts with Helm, push to Helm Repository
- Download and use existing ones

# Helm Charts

## Why Helm Charts?

- Instead of everyone creating their own K8s config files, **1 bundle** of all needed K8s manifests



## Use existing official Charts:

- Many of these created by the official sources.
  - Mysql Chart from Mysql or ElasticSearch Chart from ElasticSearch

## Sharing Helm Charts:

- Charts are hosted on their own repositories
- There are public repos, but you can have your own private repos in your company. E.g. Nexus

`helm search <keyword>`

or Artifact Hub website

# Helm as Templating Engine - 1

- Helm **renders the templates** and communicates with the Kubernetes API

How to:

1. Define a common blueprint
2. Dynamic values are replaced by placeholders

## Helm Chart Structure

Top level ***mychart*** folder  
***Chart.yaml***  
***values.yaml***  
***charts*** folder  
***templates*** folder

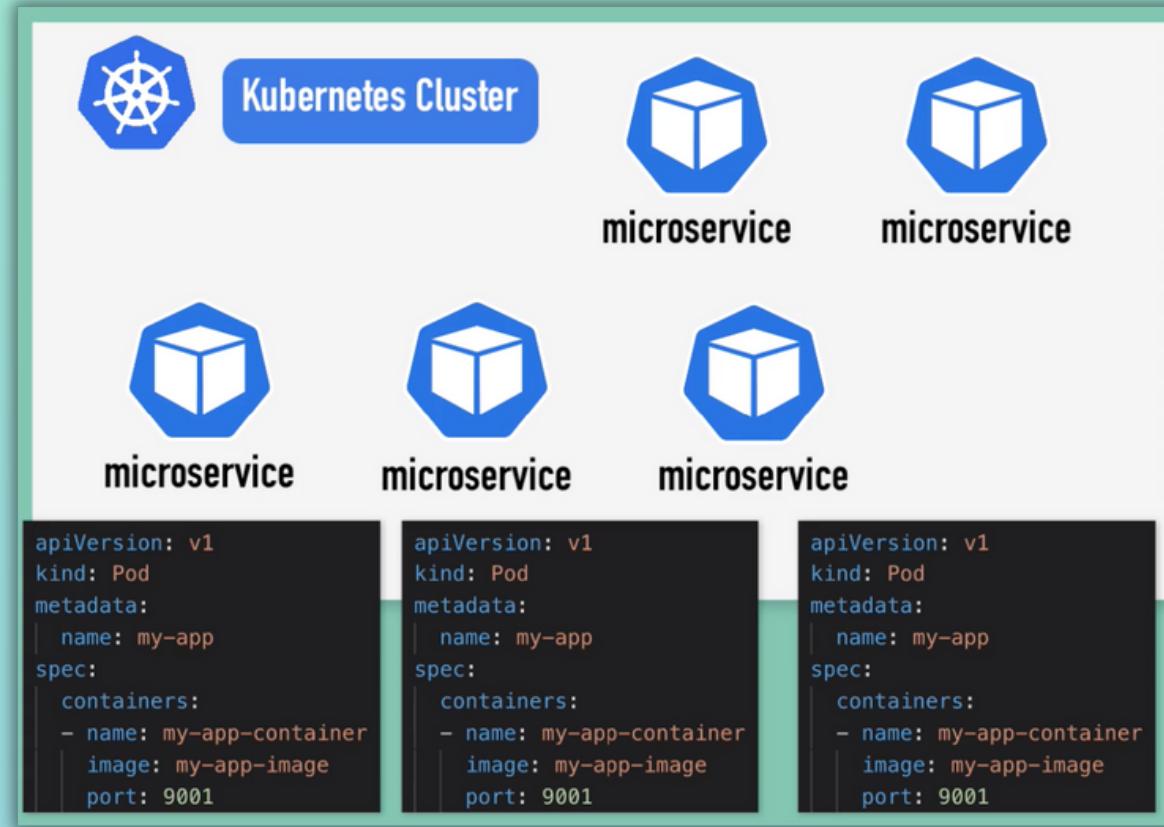


name of chart  
meta info about chart  
values for the template files  
chart dependencies  
the actual template files

```
mychart/  
  Chart.yaml  
  values.yaml  
  charts/  
  templates/  
  ...
```

# Helm as Templating Engine - 2

- Many values are the same!



✗ many YAML files

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app-container
      image: my-app-image
      port: 9001
```

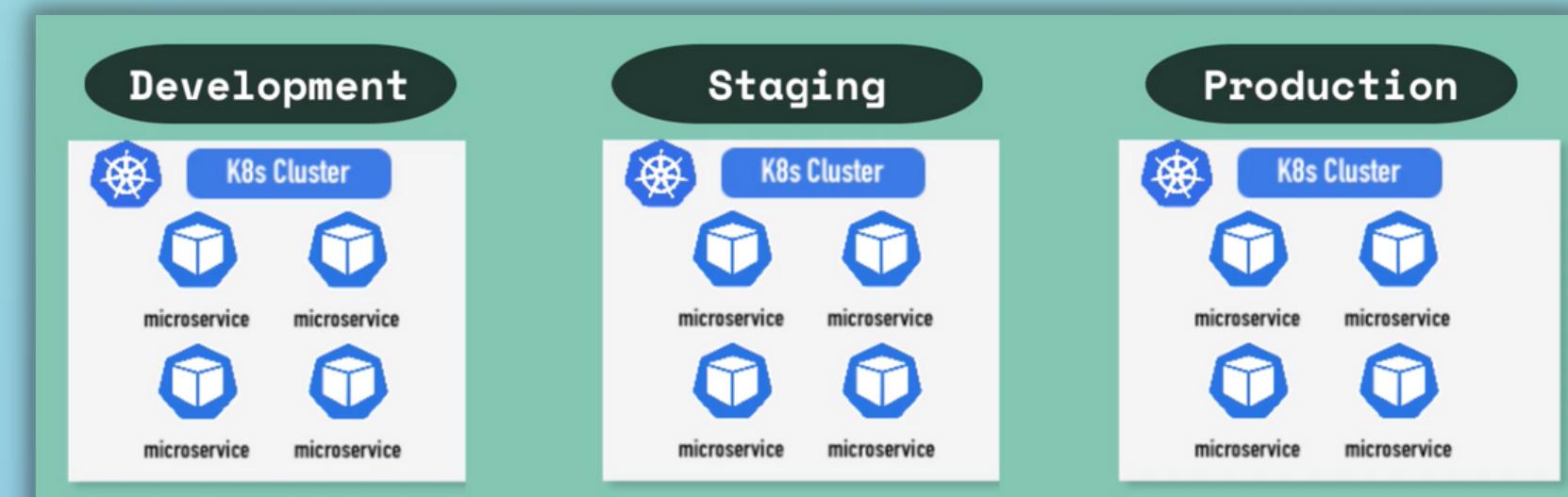
```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app-container
      image: my-app-image
      port: 9001
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: my-app-container
      image: my-app-image
      port: 9001
```

✓ just 1 YAML file

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
    - name: {{ .Values.container.name }}
      image: {{ .Values.container.image }}
      port: {{ .Values.container.port }}
```

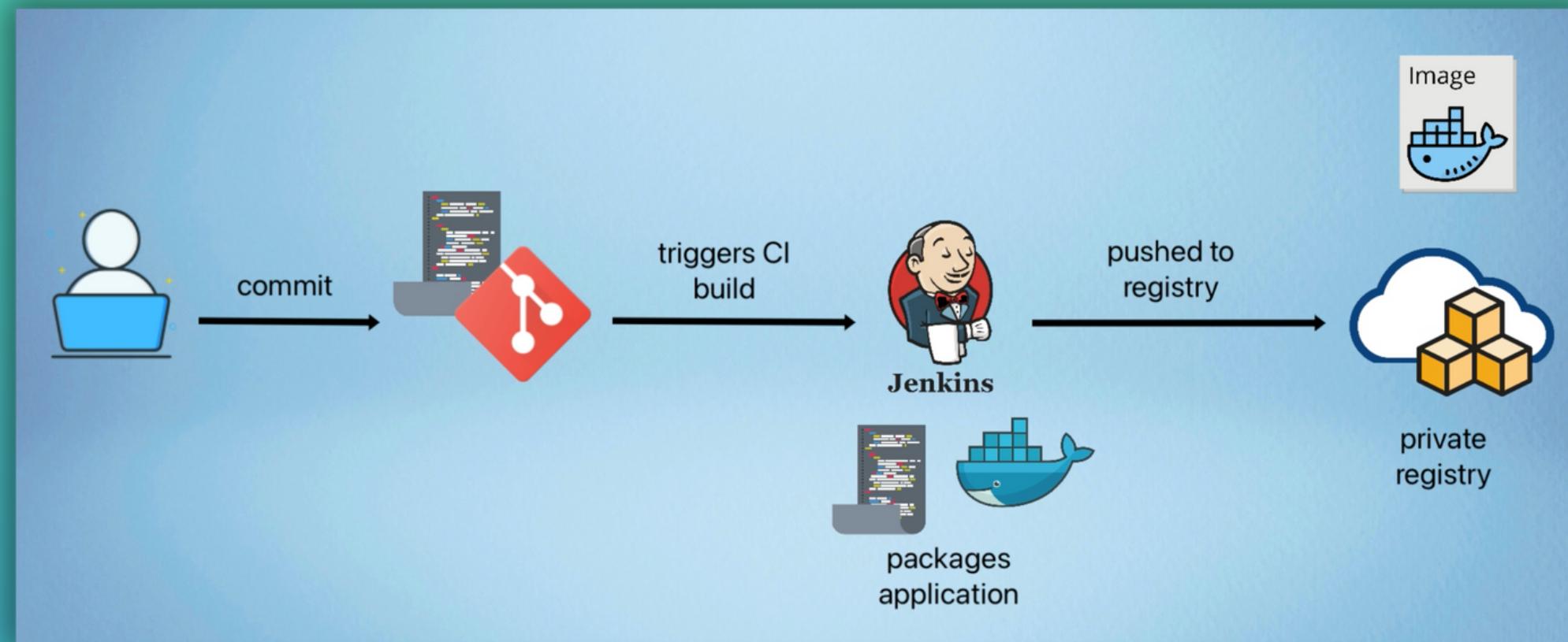
- Another use case: **Deploy** the same bundle of K8s YAML files **across multiple clusters**:



```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
```

# CD - Jenkins & Kubernetes

# Pull Docker Images into K8s cluster



- **PRIVATE REPO:** For K8s to fetch the Docker Image into K8s cluster, it **needs explicit access**



- **PUBLIC REPO:** For public images, like mongodb etc. pulled from public repositories **no credentials needed**



# Steps to pull image from Private Registry

## 1 - Create Secret Component

- Contains credentials for Docker registry



```
|my-registry-key      kubernetes.io/dockerconfigjson    1
```

- docker login token is stored in dockerconfig file

## 2 - Configure Deployment

- Use Secret using imagePullSecrets

```
6   |   app: my-app
7   spec:
8     replicas: 1
9     selector:
10    matchLabels:
11      app: my-app
12    template:
13      metadata:
14        labels:
15          app: my-app
16    spec:
17      containers:
18        - name: my-app
19          image: 664574038682.dkr.ecr.eu-central-1.amazonaws.com
20          imagePullPolicy: Always
21          ports:
22            - containerPort: 3000
```



# Kubernetes Operators

# Kubernetes Operators - 1

- Stateful applications need **constant management and syncing after deployment**. So stateful applications, like databases need to be operated
- Instead of a human operator, you have an **automated scripted operator**



Operators

- Stateless applications:

Is **managed by Kubernetes**

- Stateful applications:

**K8s can't automate** the process natively

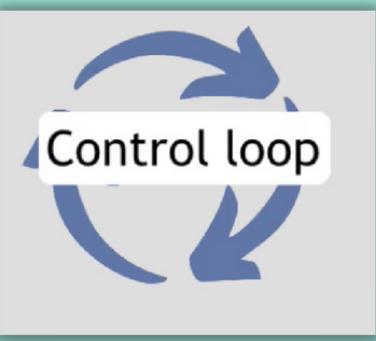
Operators are created  
by official maintainers



# Kubernetes Operators - 2

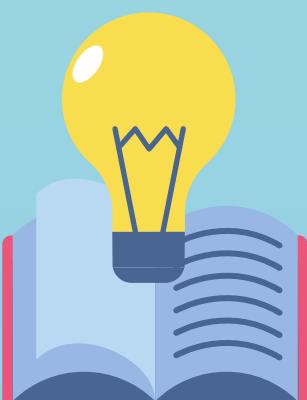
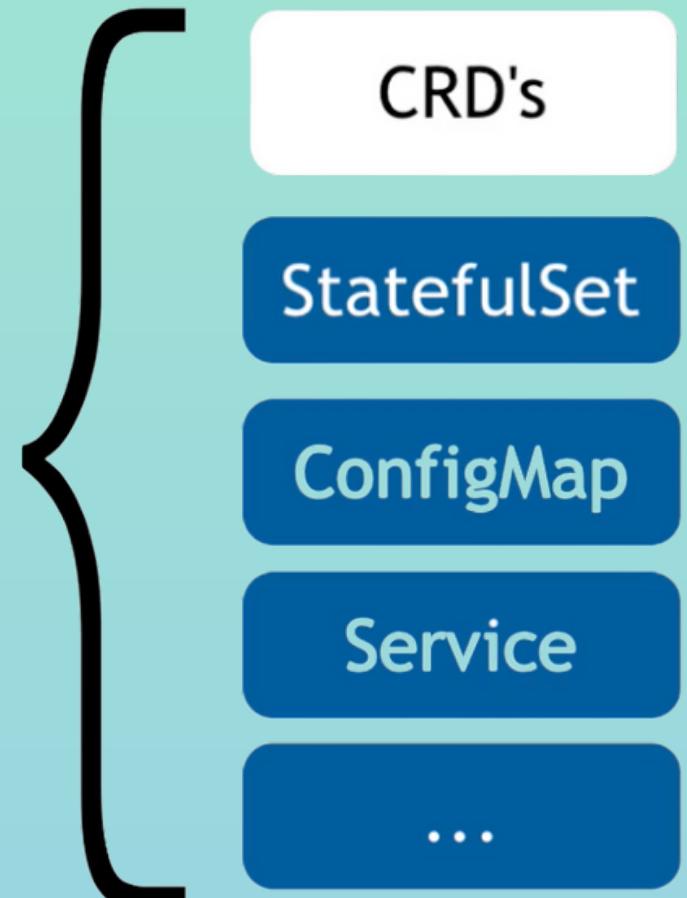
## How it works:

- Control loop mechanism
- Makes use of CRD's
  - Custom K8s component (extends the K8s API)
- Include domain/App-specific-knowledge, e.g. mysql:
  - How to create mysql cluster
  - How to run it
  - How to synchronize the data
  - How to update



### Custom Resource Definitions

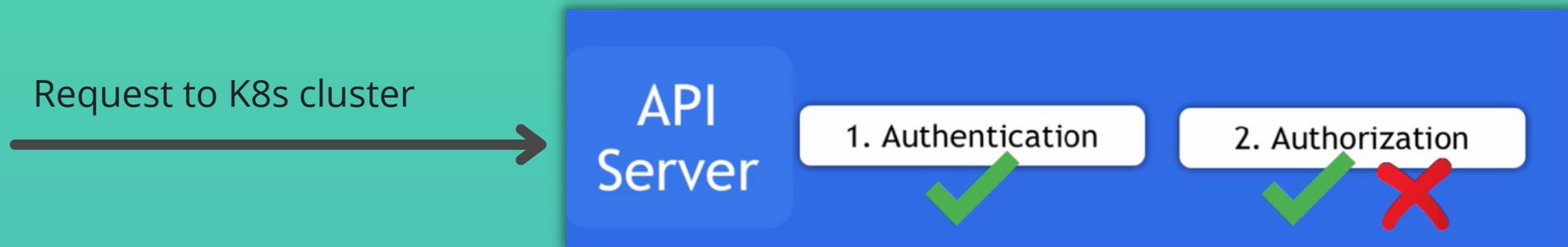
- Custom K8s component (extends the K8s API)



# Secure your Cluster - Authentication & Authorization

# Layers of Security

- In K8s, you must be **authenticated** (**logged in**) before your request can be **authorized** (**granted permission to access**)



## Authentication

- K8s doesn't manage Users natively
- No K8s resources exist for representing normal user accounts
- Admins can choose from different authentication strategies

## Authorization

- K8s supports multiple authorization modules, such as ABAC mode, RBAC Mode and Webhook mode
- On cluster creation, admins configure the authorization modules that should be used
- K8s checks each module, and if any module authorizes the request, then the request can proceed

# Authentication

- API server handles authentication of all the requests
- Available **authentication strategies**:



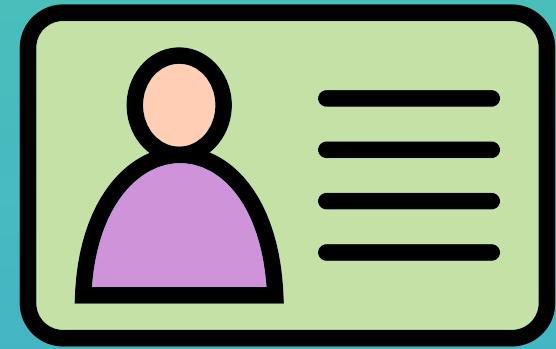
Client Certificates



Static Token File



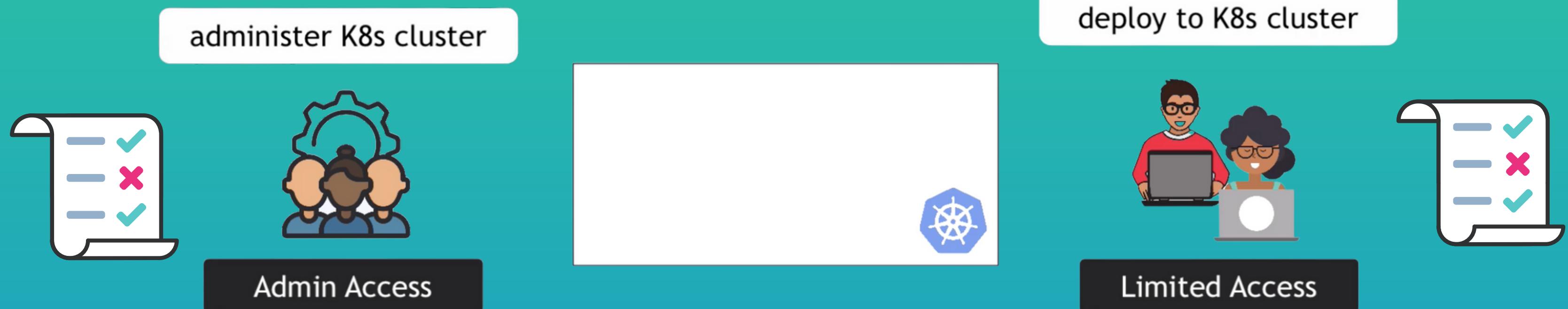
3rd Party Identity Service, like LDAP



- A csv file with a minimum of 3 columns: token, user name, user uid, followed by optional group names

# Authorization

- As a security best practice, we only want to give people or services **just enough permission** to do their tasks: **Least Privilege Rule**



- Admins need cluster-wide access to do tasks like configure namespaces etc.

- Developers need only limited access, for example to deploy applications to 1 specific namespace

# Authorization with RBAC - 1

## Role-Based Access Control (RBAC)

- A method of regulating access to resources based on the roles of users within the organization
- Enabling RBAC: `kube-apiserver --authorization-mode=RBAC --other-options`
- 4 kinds of K8s resources: ClusterRole, Role, ClusterRoleBinding, RoleBinding

### Role and ClusterRole

- Contains rules that represent a **set of permissions**
- Permissions are additive

### RoleBinding and ClusterRoleBinding

- **Link ("Bind")** a Role or ClusterRole to a **User or Group**

# Authorization with RBAC - 2

The diagram illustrates the RBAC components and their relationships:

- ClusterRole** (Orange box): Associated with **K8s Admins** (Icon: three people with gear) and a **ClusterRole** button.
- Role** (Green box): Associated with **Developer** (Icon: two people with laptops) and a **Role** button.
- ClusterRoleBinding**: A large orange arrow points from the ClusterRole box to the Role box, indicating the binding process.
- Namespace Context**: A central area shows two namespaces: **NS 1** and **NS 2**. **NS 2** is highlighted with a green border, representing a namespaced context.
- RoleBinding**: A green arrow points from the Role box to the Namespace context, indicating how a Role is bound to a specific namespace.

**ClusterRole** (Left side):

- Define permissions **cluster wide**
- For example: configure cluster-wide volumes, namespaces

**Role** (Right side):

- Define **namespaced** permissions through Role
- Bound to a specific namespace
- **What resources** in that namespace you can access
- **What action** you can do with this resource

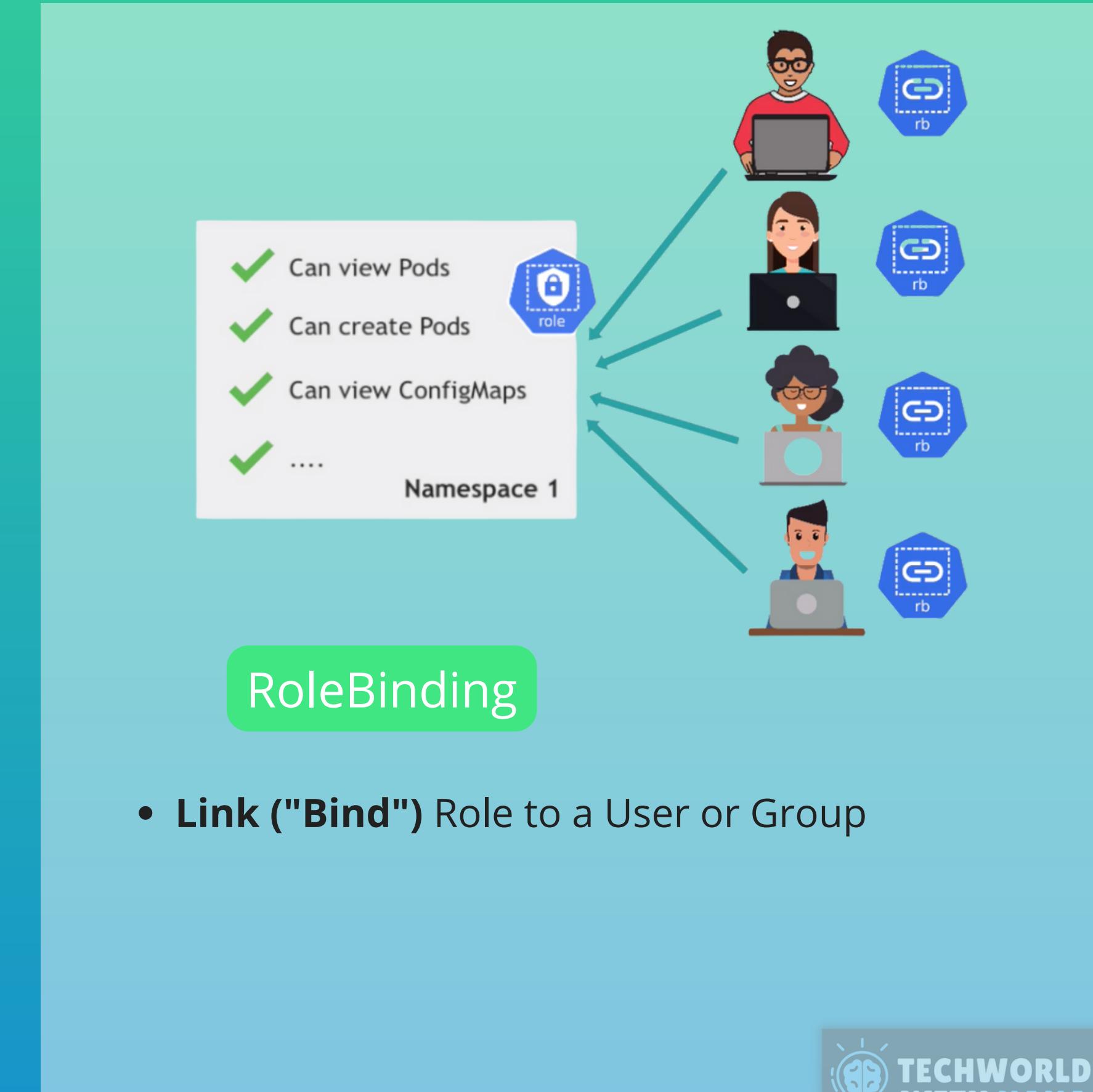
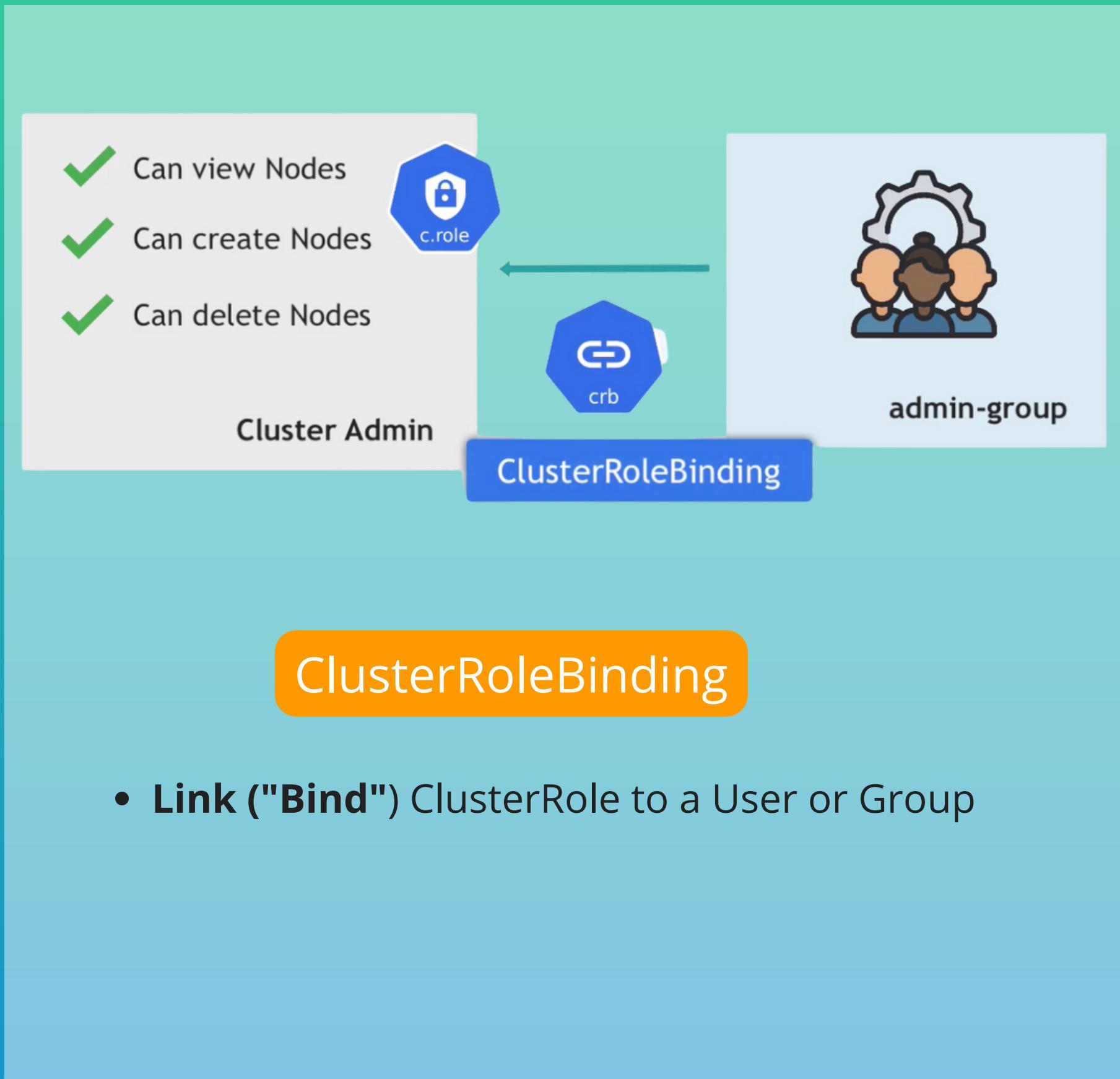
**ClusterRole YAML Example** (Bottom center):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: k8s-admins
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["*"]
```

**Role YAML Example** (Bottom right):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "create", "list"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
```

# Authorization with RBAC - 3

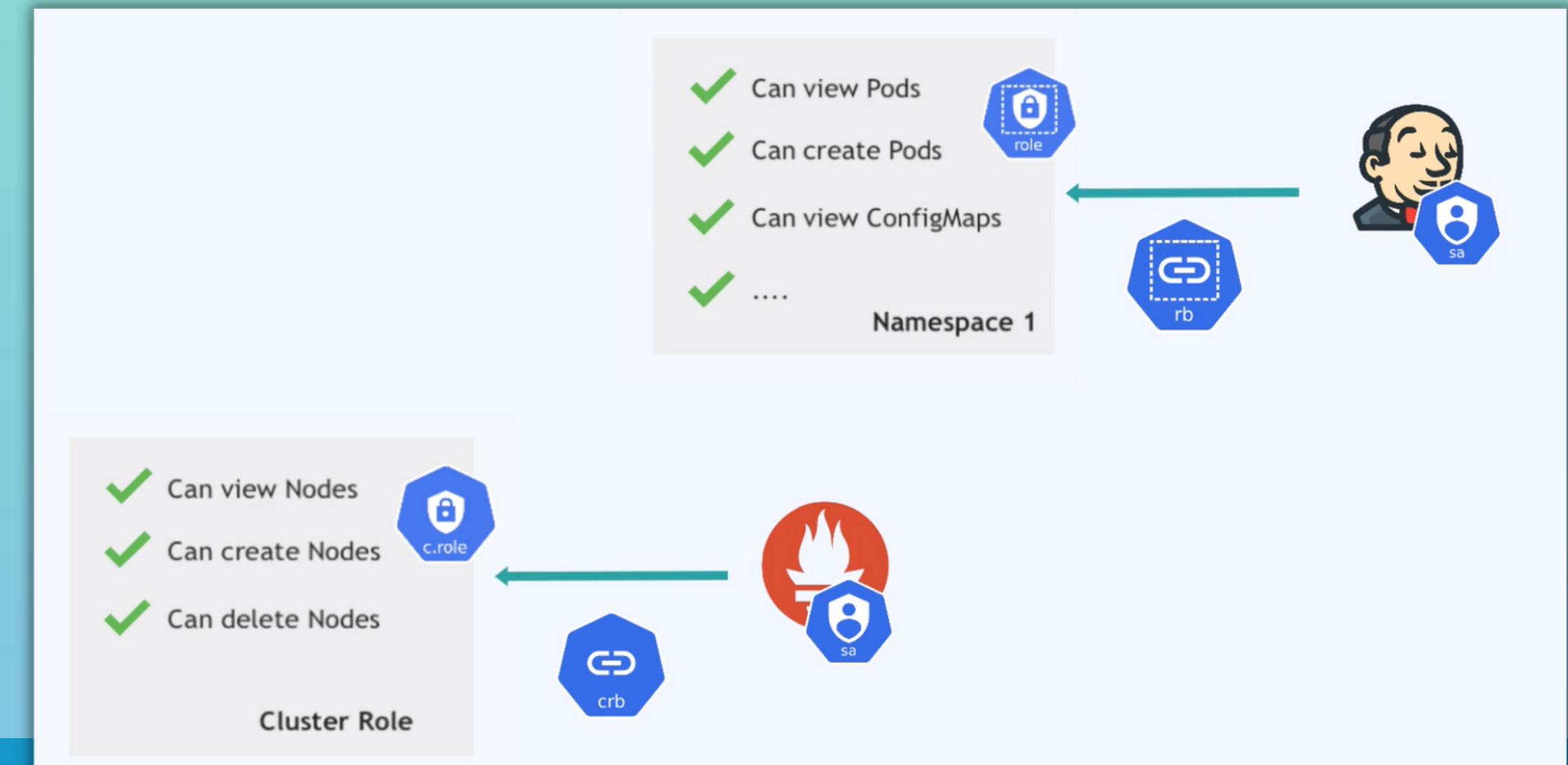


# Authorization for Applications

- **ServiceAccounts** provide an identity for processes that run in a Pod, for example Jenkins or Prometheus



- A RoleBinding or ClusterRoleBinding can also bind a role to a **ServiceAccount**



# Microservices in Kubernetes

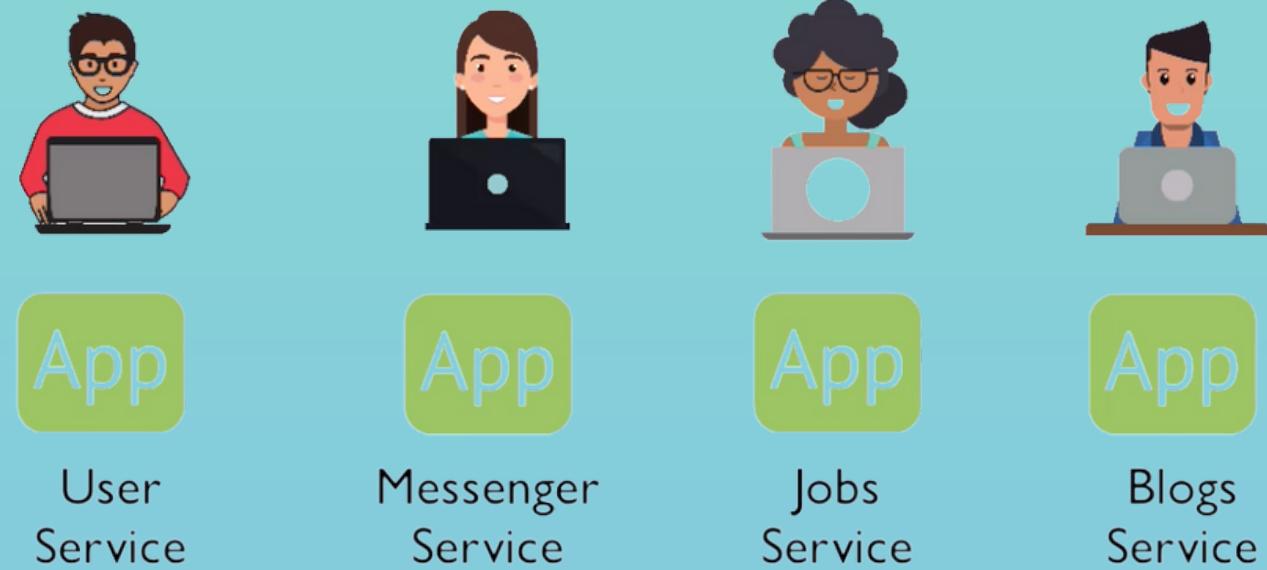
# Introduction to Microservices

- Microservices are an architectural approach to software development
- Software is **composed of small independent services** (instead of having a huge monolith)
- Each business functionality is **encapsulated** into own Microservice (MS)



## Benefits

- Each MS can be developed, packaged and released **independently**
- Changes in 1 MS doesn't affect other MS
- Less interconnected logic, **loosely coupled**
- Each MS can be developed by separate developer teams

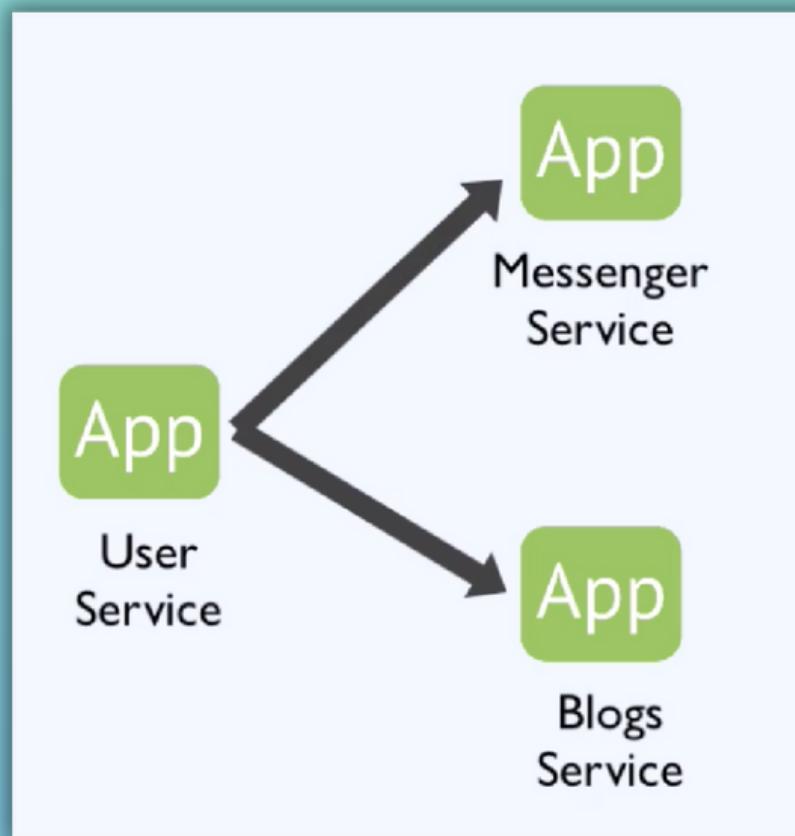


# Communication between Microservices

- Communication between those services can happen in different ways:

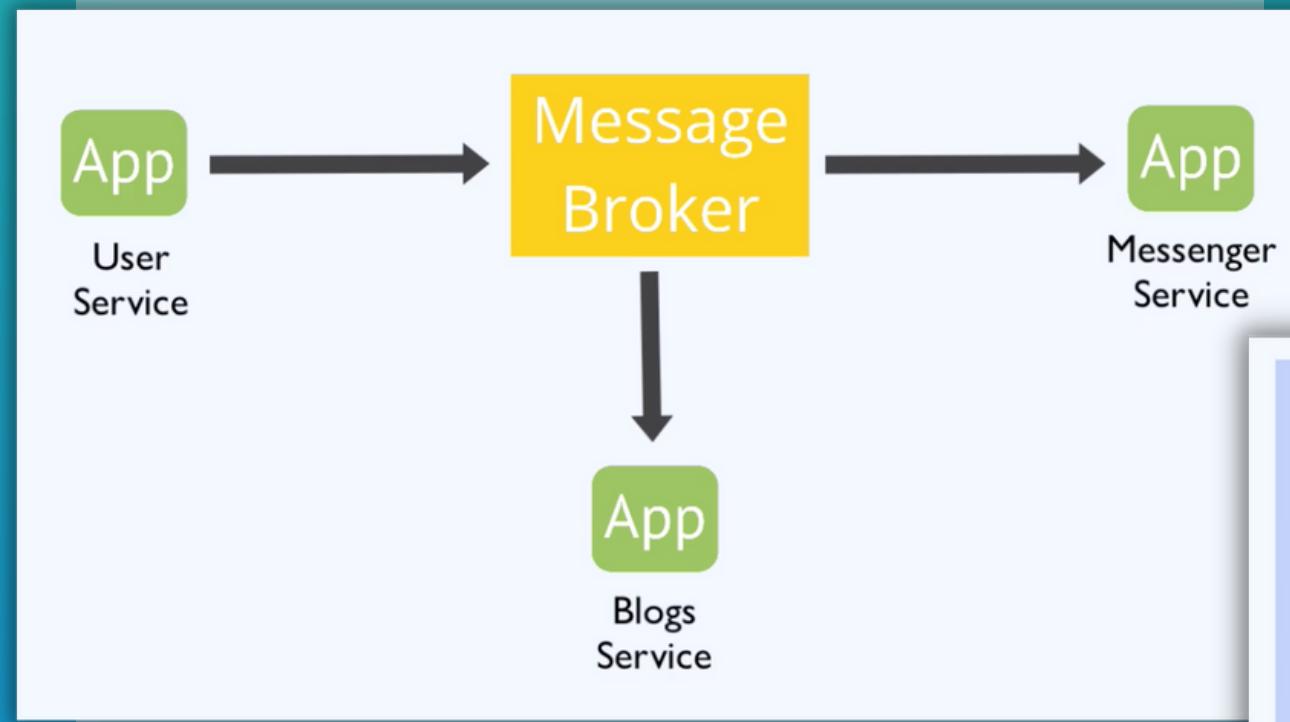
## Service-to-Service API Calls

- Direct communication



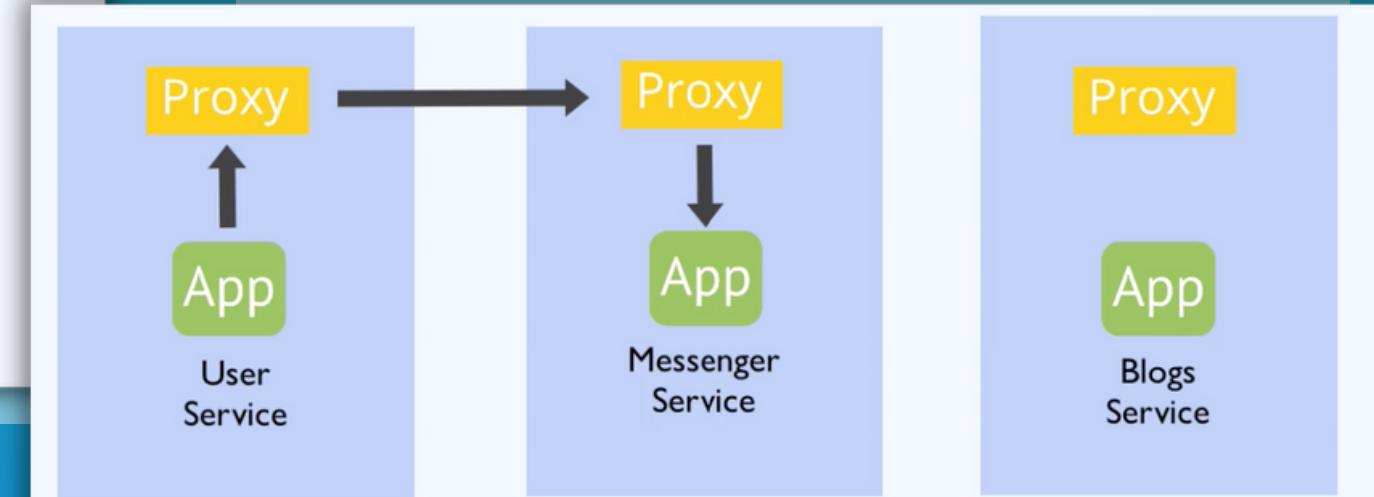
## Message-Based Communication

- Communication through a message broker, like Rabbitmq or Redis

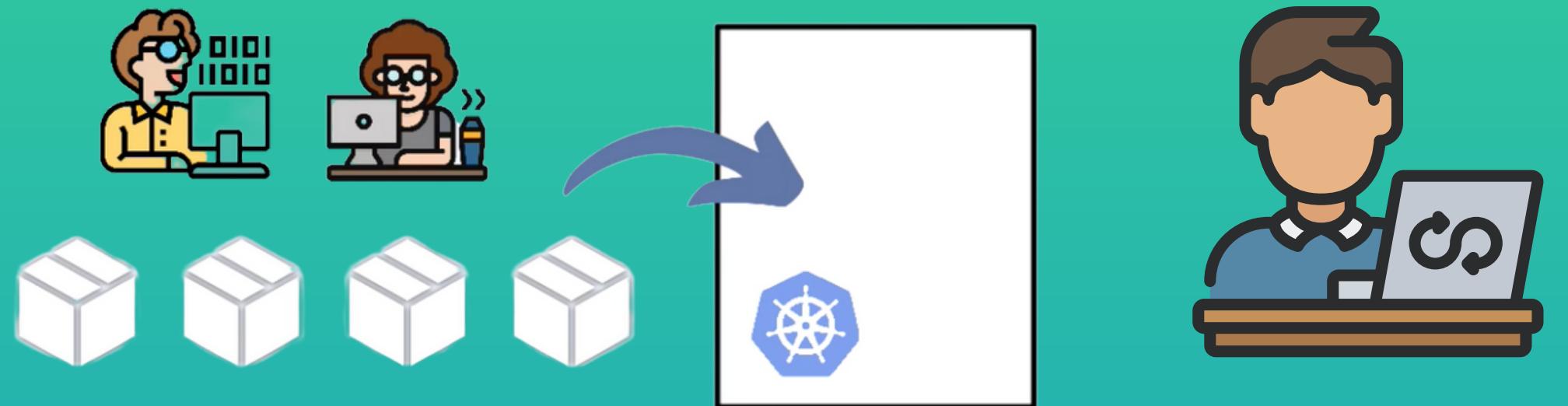


## Service Mesh Architecture

- Platform layer on top of the infrastructure layer, like Istio, Linkerd, HashiCorp Consul
- Enables managed, observable and secure communication



# DevOps Task: Deploy Microservices App - 1



- Developers develop the microservice applications
- As a DevOps engineer your task would be to **deploy the existing microservices application** in a K8s cluster

Information you need from developers:

1. Which services need to be deployed?
2. Which service is talking to which service?
3. How are they communicating?
4. Which database are they using? Other 3rd party services?
5. On which port does each service run?

# DevOps Task: Deploy Microservices App - 2

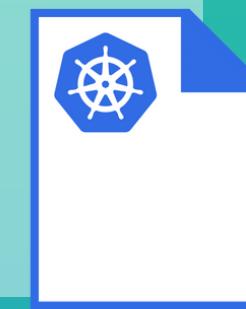


HOW TO

## 1. Prepare K8s environment

- a. Deploy any 3rd party apps
- b. Create Secrets and ConfigMaps for microservices

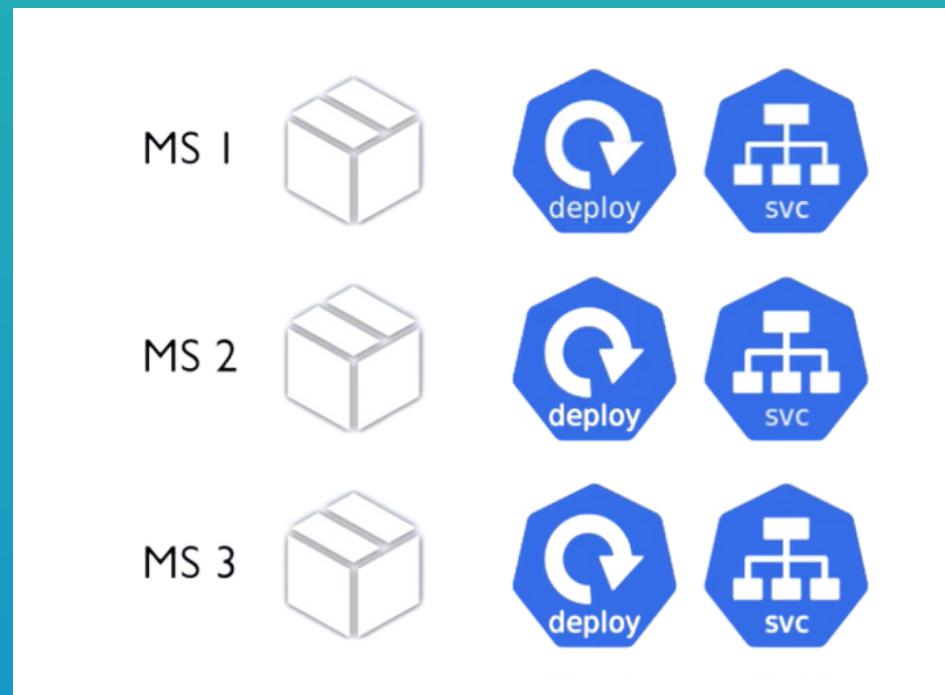
## 2. Create Deployment and Service for each microservices



1)



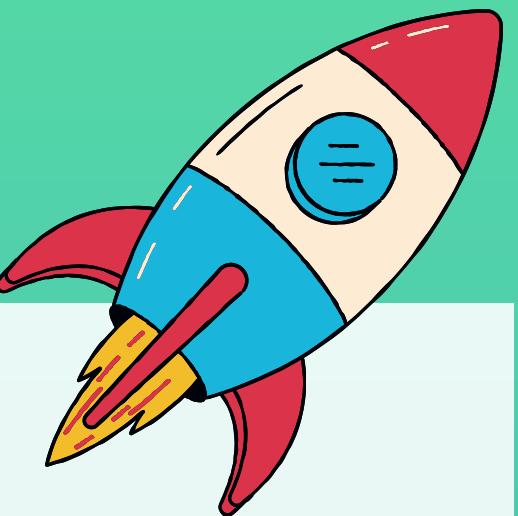
2)



When deploying multiple similar services to K8s, you can **use helm chart with 1 common template and replace specific values for each service on the fly** during deployment

# Kubernetes Production & Security Best Practices

# Best Practices - 1



- Specify a pinned version on each container image

**Why?** Otherwise, latest version is fetched, which makes it unpredictable and intransparent as to which versions are deployed in the cluster

- Configure a liveness probe on each container

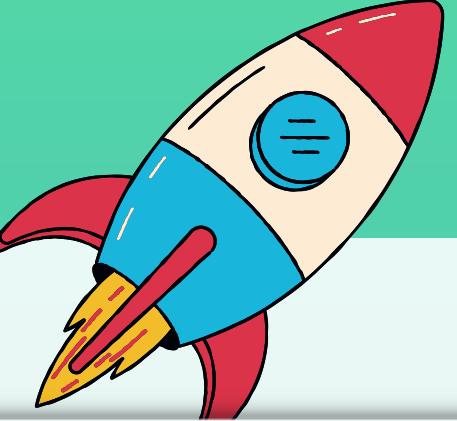
**Why?** K8s knows the Pod state, not the application state. Sometimes the pod is running, but the container inside has crashed. With a liveness probe we can let K8s know if it needs to restart the container

```
spec:  
  containers:  
    - name: service  
      image: gcr.io/google-samples/microservices-demo/emailserv  
      ports:  
        - containerPort: 8080  
      env:  
        - name: PORT  
          value: "8080"  
      livenessProbe:  
        periodSeconds: 5  
        exec:  
          command: ["/bin/grpc_health_probe", "-addr=:8080"]  
      readinessProbe:  
        periodSeconds: 5  
        exec:  
          command: ["/bin/grpc_health_probe", "-addr=:8080"]
```

- Configure a readiness probe on each container

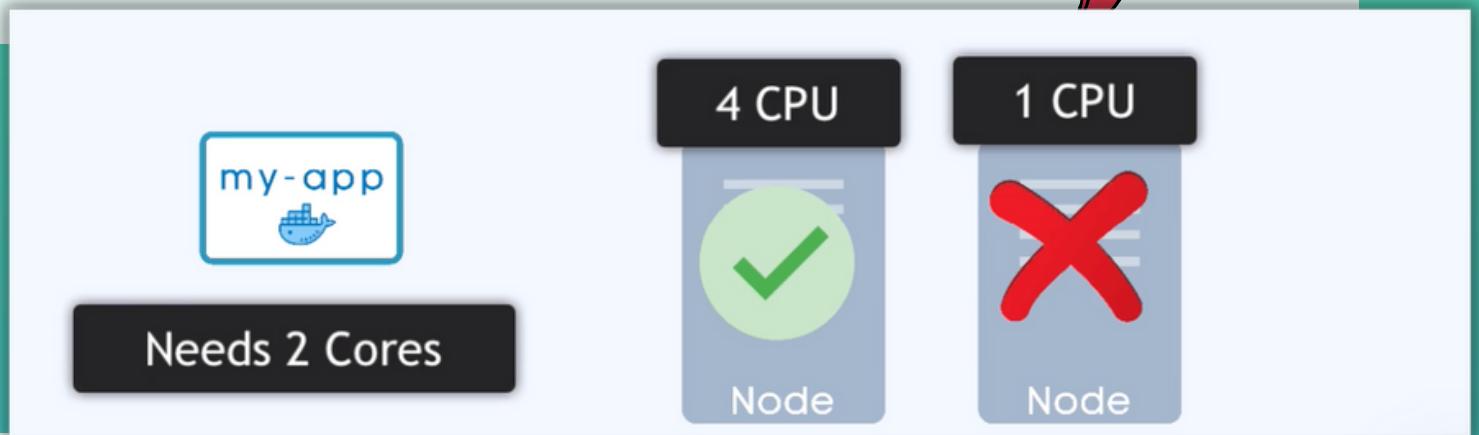
**Why?** Let's K8s know if application is ready to receive traffic

# Best Practices - 2



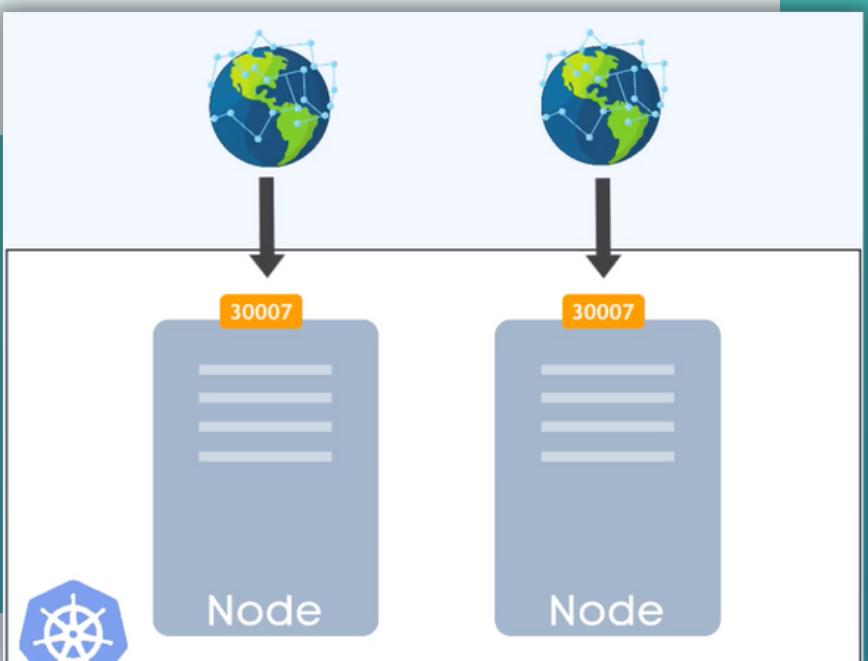
- Configure resource limits & requests for each container

**Why?** To make sure 1 buggy container doesn't eat up all resources, breaking the cluster



- Don't use NodePort in production

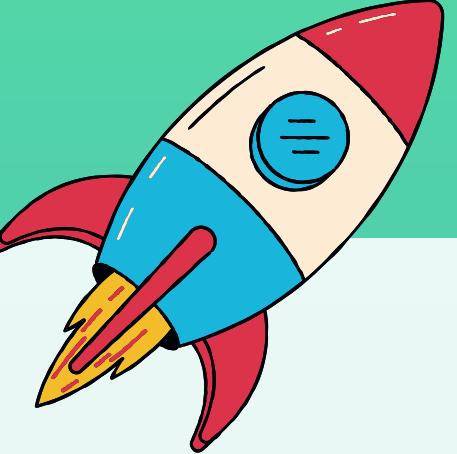
**Why?** NodePort exposes Worker Nodes directly, multiple points of entry to secure. Better alternative: Loadbalancer or Ingressss



- Always deploy more than 1 replica for each application

**Why?** To make sure your application is always available, no downtime for users!

# Best Practices - 3



- Always have more than 1 Worker Node

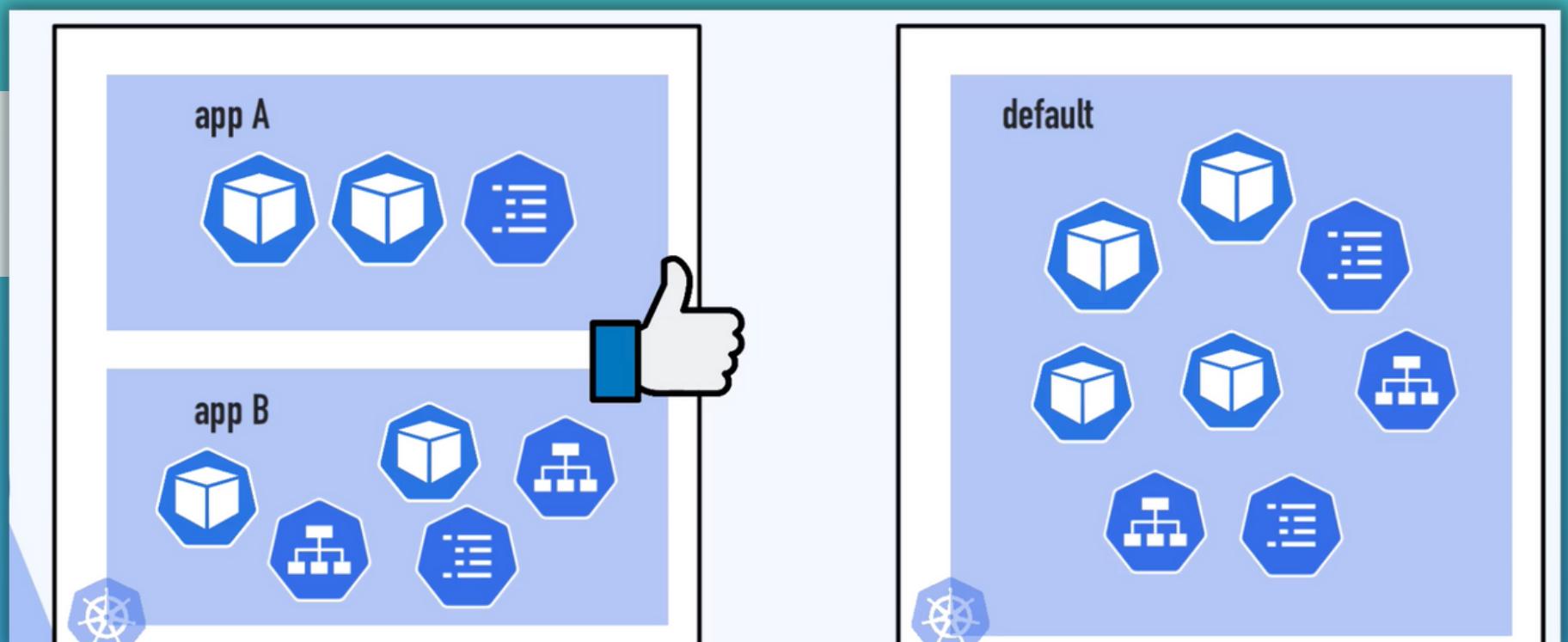
**Why?** Avoid single point of failure with just 1 Node

- Label all your K8s resources

**Why?** Have an identifier for your components to group pods and reference e.g. in Service

- Use namespaces to group your resources

**Why?** To organize resources and to define access rights based e.g. on namespaces



# Security Best Practices

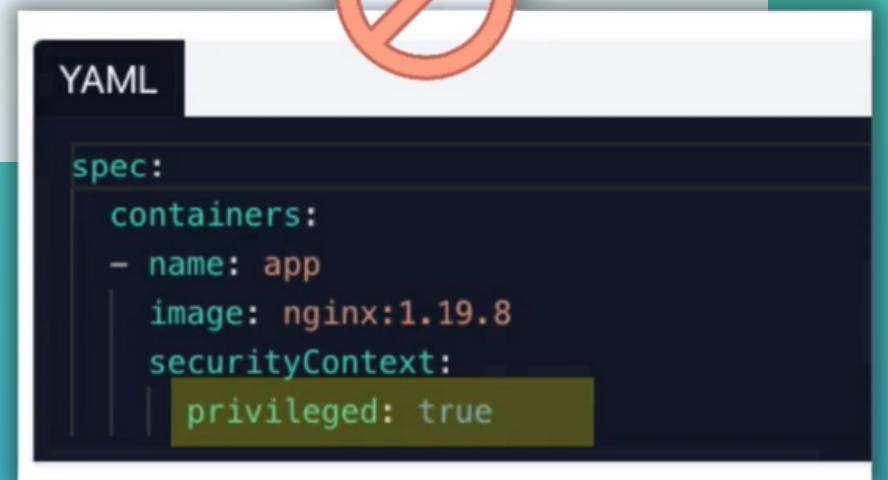
- Ensure Images are free of vulnerabilities

**Why?** Third-party libraries or base images can have known vulnerabilities. You can do manual vulnerability scans or better automated scans in CI/CD pipeline



- No root access for containers

**Why?** With root access they have access to host-level resources. Much more damage possible, if container gets **hacked!**



- Keep K8s version up to date

**Why & How?** Latest versions include patches to previous security issues etc.

Upgrade with zero downtime by having multiple nodes and pod replicas on different nodes