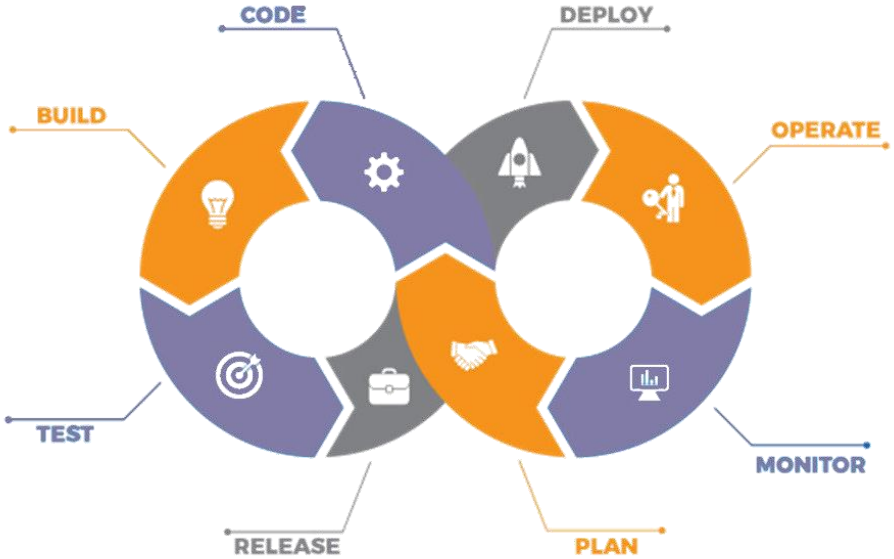
Introduction to Kubernetes



Agenda

01 Introduction to Kubernetes

02 Docker Swarm Vs. Kubernetes

03 Kubernetes Architecture

04 Kubernetes Installation

05 Working of Kubernetes

06 Deployments in Kubernetes

07 Services in Kubernetes

08 Ingress in Kubernetes

09 Kubernetes Dashboard

Introduction to Kubernetes

Introduction to Kubernetes

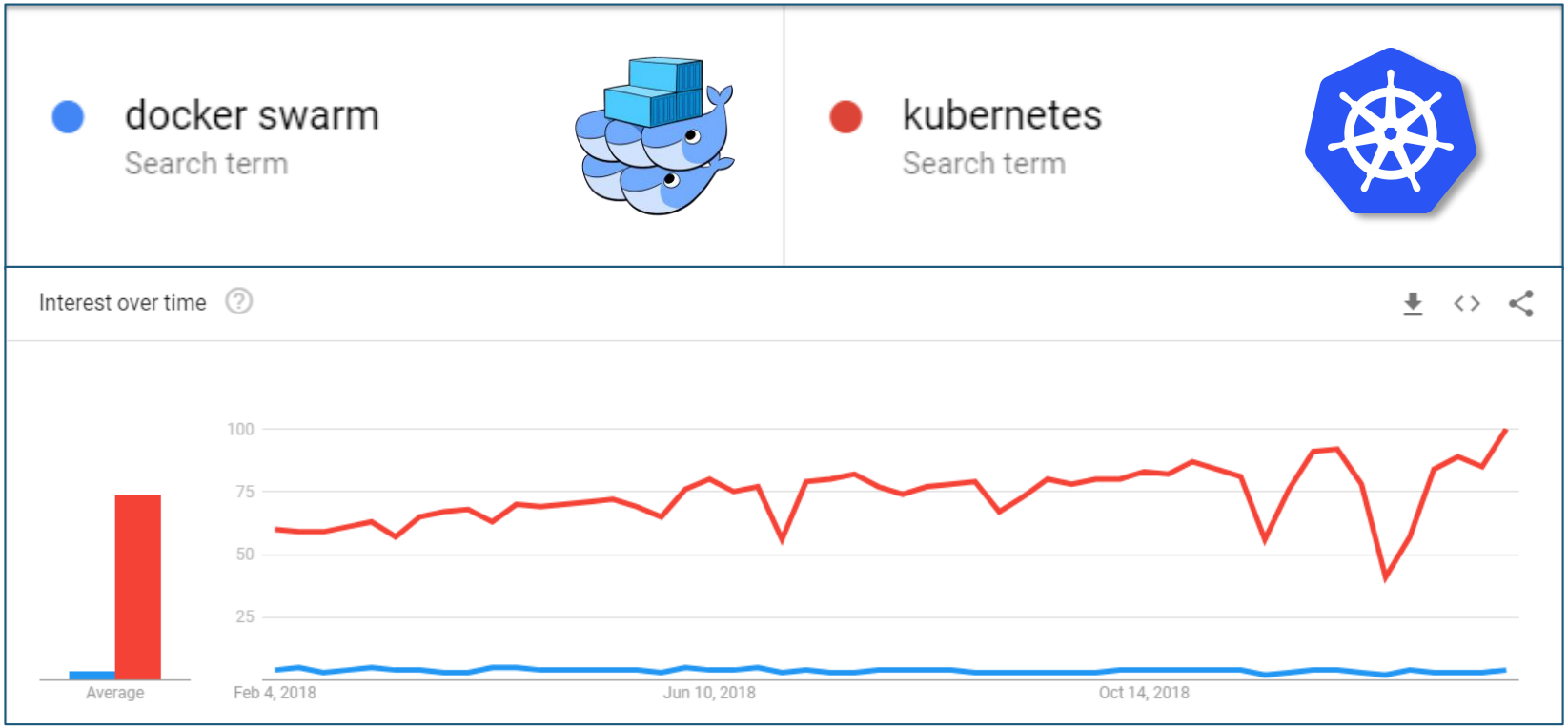
- Kubernetes is an open-source container orchestration software.
- It was originally developed by Google.
- It was first released on July 21, 2015.
- It is the ninth most active repository on GitHub in terms of number of commits.

Features of Kubernetes

- Pods
- Replication Controller
- Storage Management
- Resource Monitoring
- Health Checks
- Service Discovery
- Networking
- Secret Management
- Rolling Updates

Docker Swarm Vs. Kubernetes

Docker Swarm Vs. Kubernetes



Source: trends.google.com

Docker Swarm Vs. Kubernetes

Docker Swarm

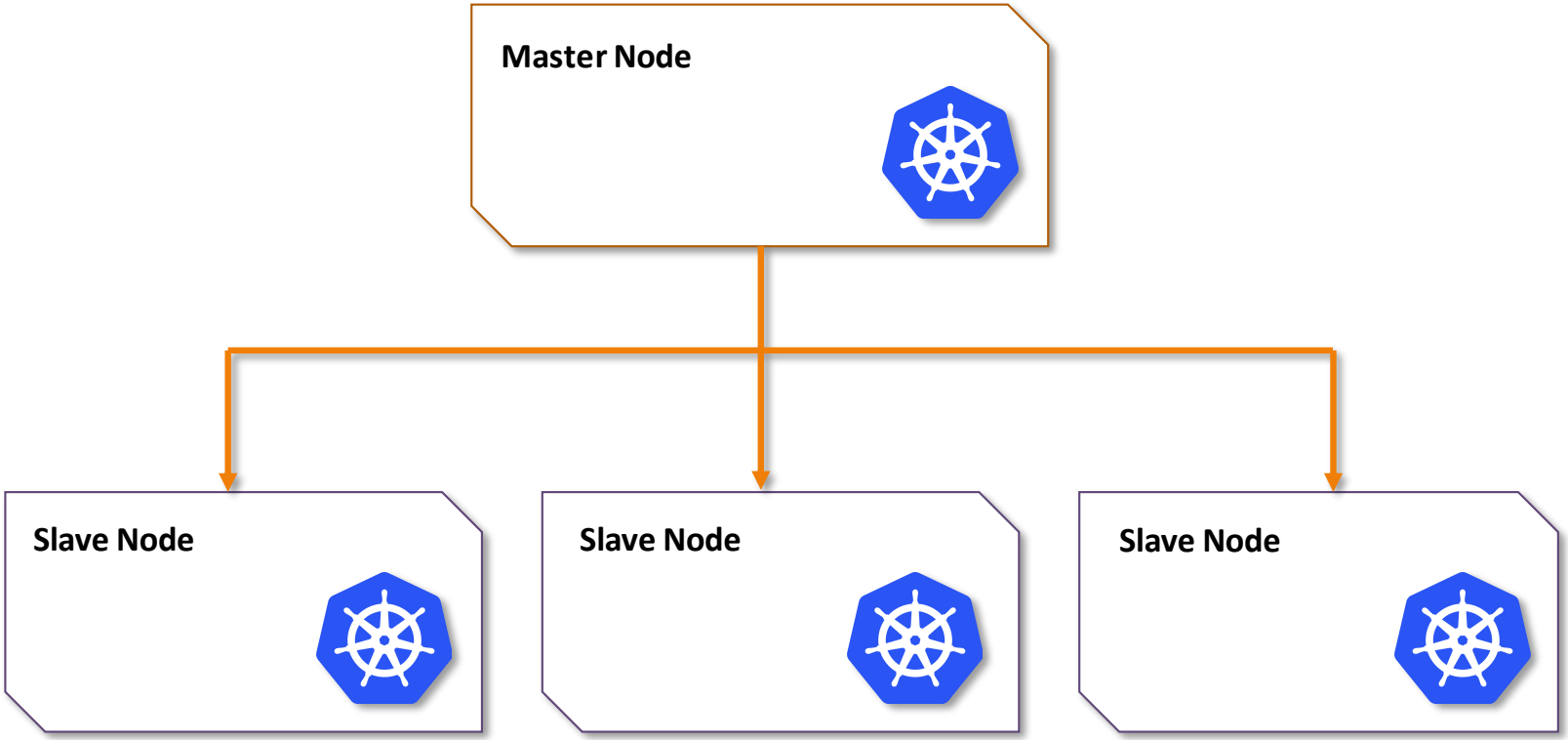
- Easy to install and initialize
- Faster when compared to Kubernetes
- Not reliable and has less features

Kubernetes

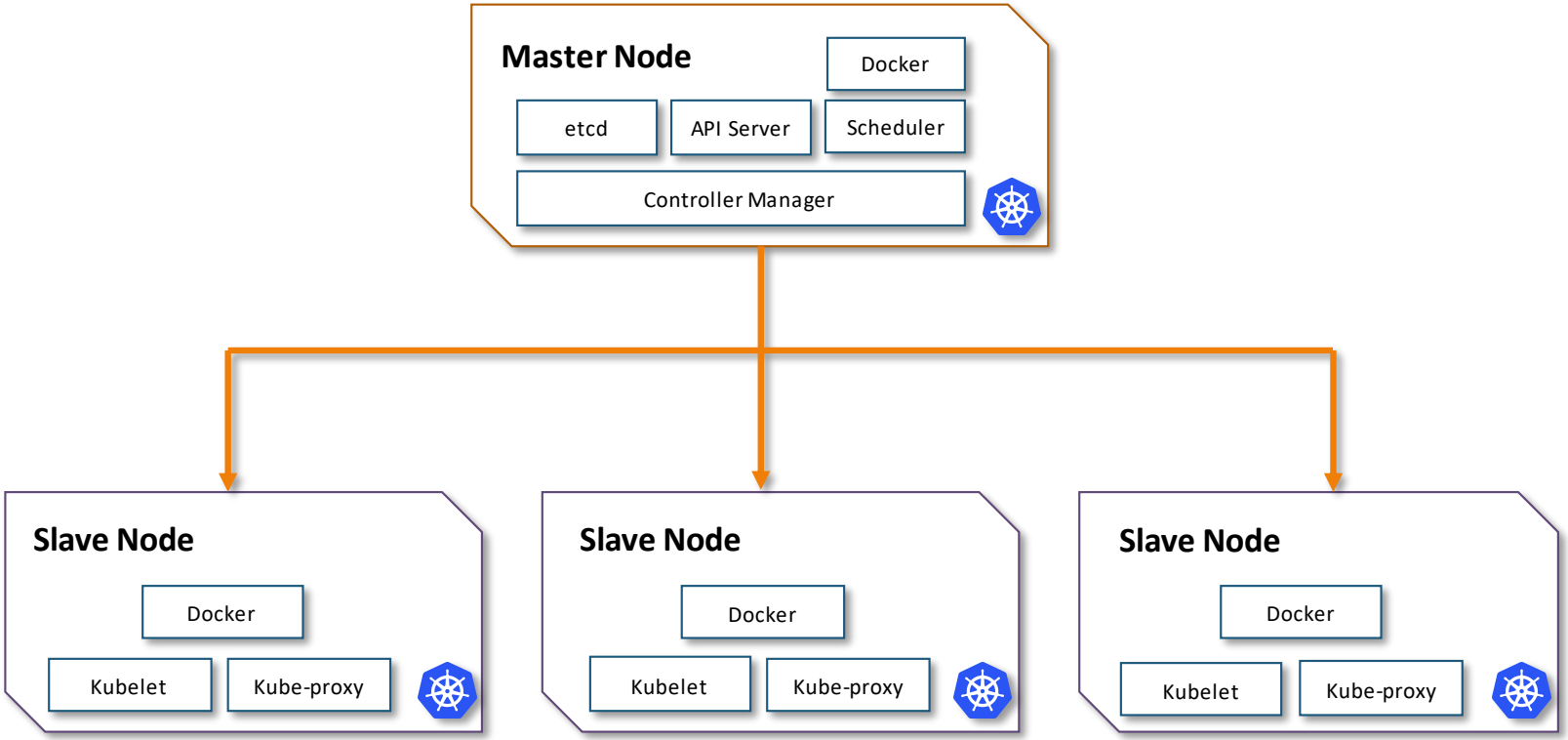
- Complex procedure to install
- Slower when compared to Docker Swarm
- More reliable and has more features

Kubernetes Architecture

Kubernetes Architecture



Kubernetes Architecture

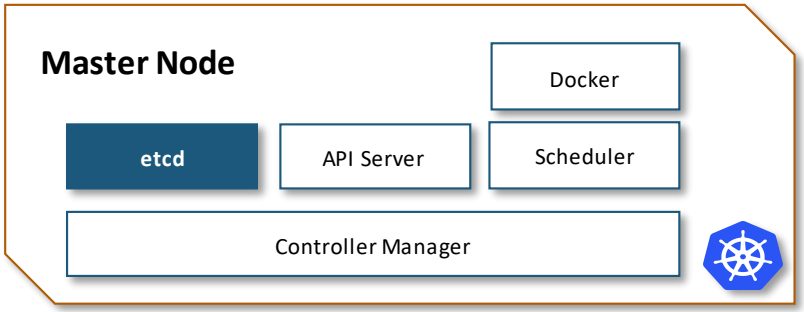


Kubernetes Architecture: Master Components

Kubernetes Architecture: Master Components

- **etcd**
- API Server
- Scheduler
- Controller Manager

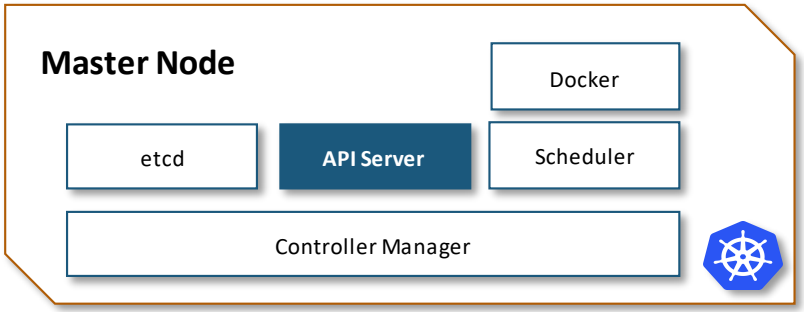
It is a highly available distributed key–value store, which is used to store cluster wide secrets. It is only accessible by the Kubernetes API server, as it has sensitive information.



Kubernetes Architecture: Master Components

- etcd
- **API Server**
- Scheduler
- Controller Manager

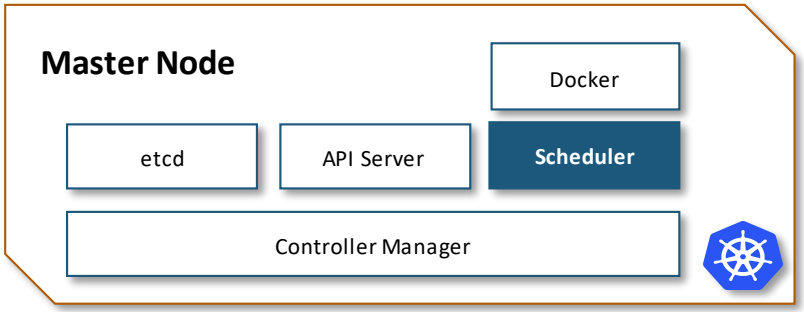
It exposes Kubernetes API. Kubernetes API is the front-end for the Kubernetes Control Plane and is used to deploy and execute all operations in Kubernetes.



Kubernetes Architecture: Master Components

- etcd
- API Server
- **Scheduler**
- Controller Manager

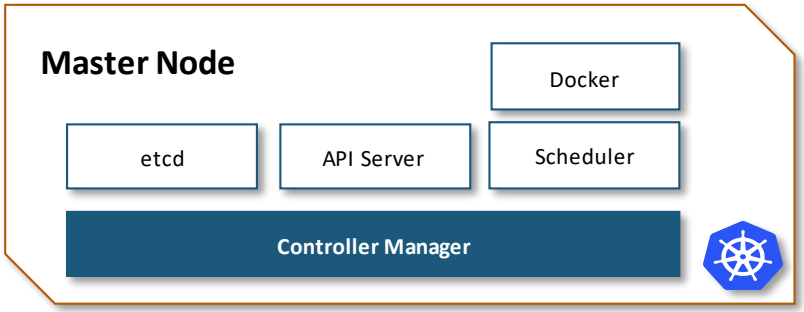
The scheduler takes care of scheduling of all processes and the dynamic resource management and manages present and future events on the cluster.



Kubernetes Architecture: Master Components

- etcd
- API Server
- Scheduler
- **Controller Manager**

The controller manager runs all controllers on the Kubernetes cluster. Although each controller is a separate process, to reduce complexity, all controllers are compiled into a single process. They are as follows: **Node Controller, Replication Controller, Endpoints Controller, Service Accounts and Token Controllers**.



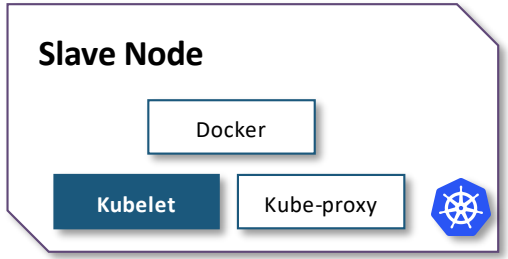
Kubernetes Architecture: Slave Components

Kubernetes Architecture: Slave Components

Kubelet

Kube-proxy

Kubelet takes the specification from the API server and ensures that the application is running according to the specifications which were mentioned. Each node has its own kubelet service.

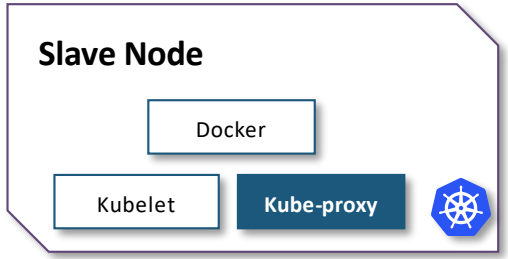


Kubernetes Architecture: Slave Components

Kubelet

Kube-proxy

This proxy service runs on each node and helps in making services available to the external host. It helps in connection forwarding to the correct resources. It is also capable of doing primitive load balancing.

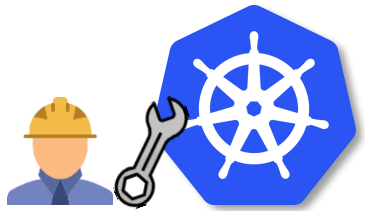


Kubernetes Installation

Kubernetes Installation

There are numerous ways to install Kubernetes. Following are some of the popular ways:

- **Kubeadm**: Bare Metal Installation
- **Minikube**: Virtualized Environment for Kubernetes
- **Kops**: Kubernetes on AWS
- **Kubernetes on GCP**: Kubernetes running on Google Cloud Platform



Hands-on: Installing Kubernetes Using Kubeadm

Working of Kubernetes

Working of Kubernetes

Pods can have one or more containers coupled together. They are the basic unit of Kubernetes. To increase high availability, we always prefer pods to be in replicas.

Working of Kubernetes

Services are used to load balance the traffic among the pods. It follows round-robin distribution among the healthy pods.

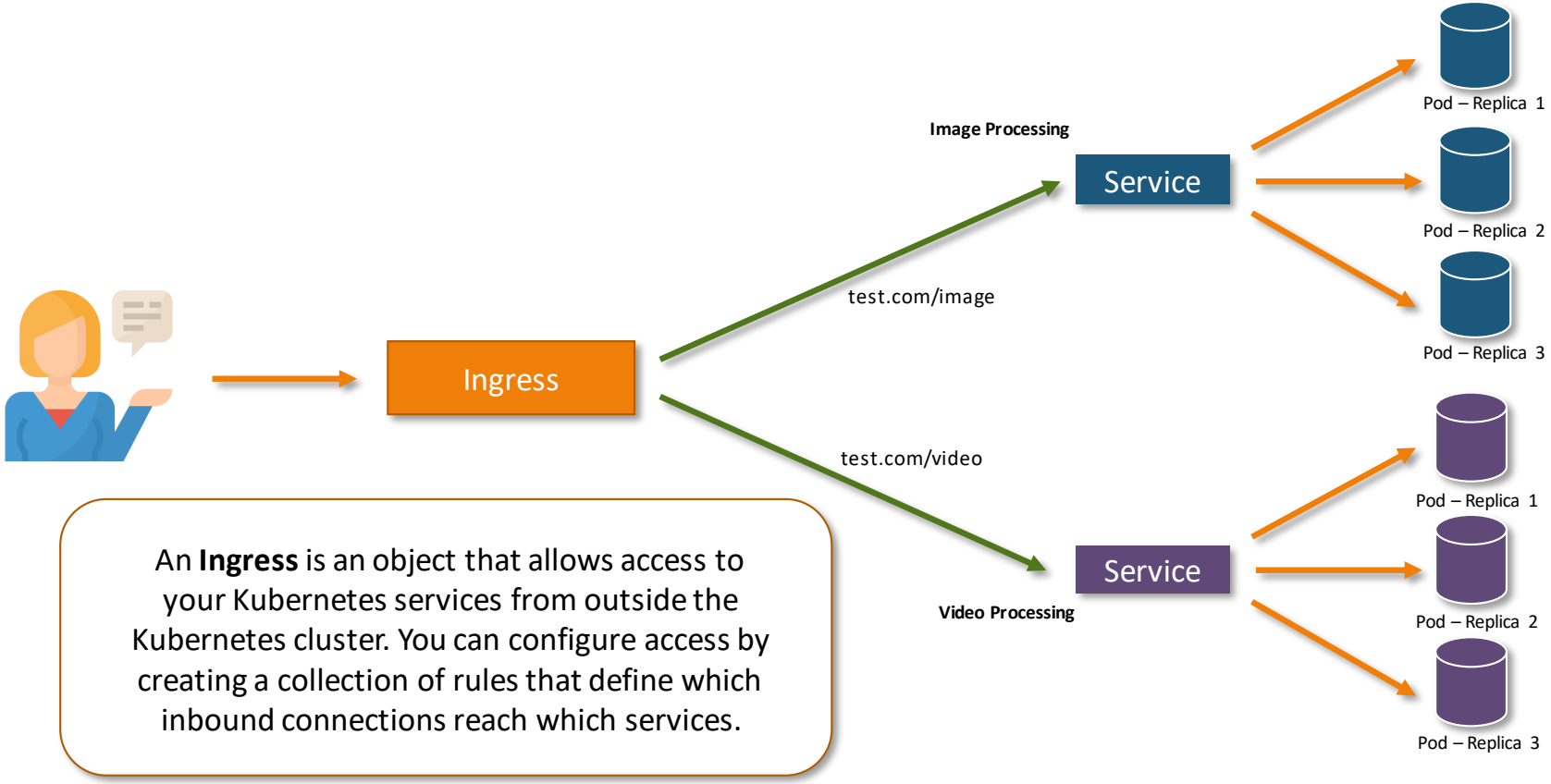
Service

Pod – Replica 1

Pod – Replica 2

Pod – Replica 3

Working of Kubernetes

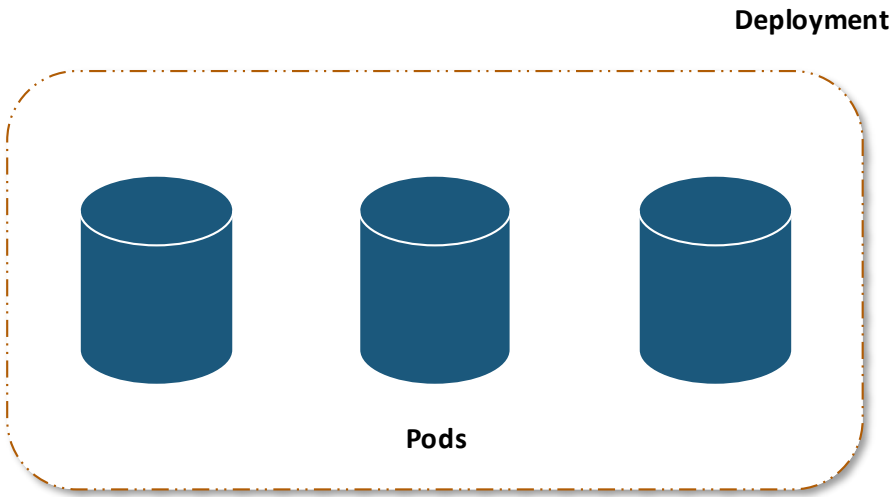


An **Ingress** is an object that allows access to your Kubernetes services from outside the Kubernetes cluster. You can configure access by creating a collection of rules that define which inbound connections reach which services.

Deployments in Kubernetes

Deployments in Kubernetes

Deployment in Kubernetes is a controller which helps your applications reach the desired state; the desired state is defined inside the deployment file.



YAML Syntax for Deployments

This YAML file will deploy 3 pods for nginx and will maintain the desired state, which is 3 pods, until this deployment is deleted.

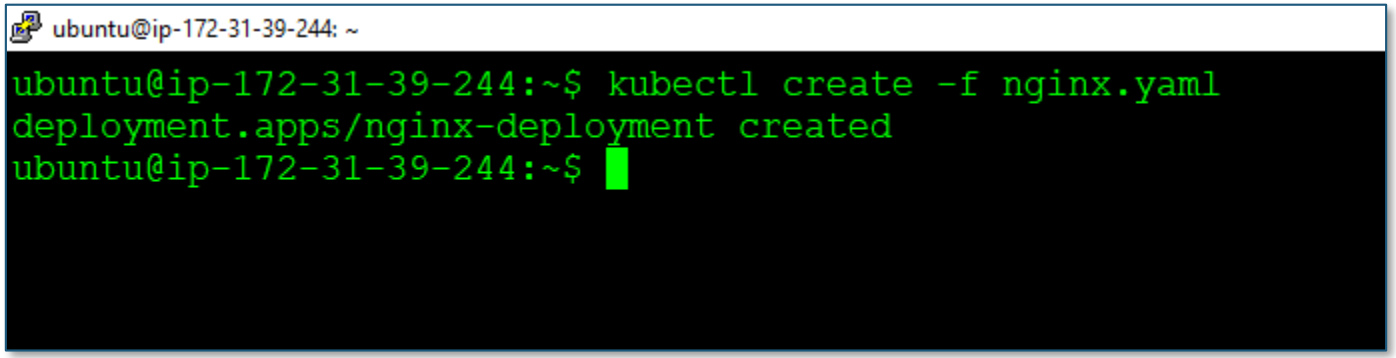
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Creating a Deployment

Once the file is created, to deploy this deployment use the following syntax:

Syntax

```
kubectl create –f nginx.yaml
```

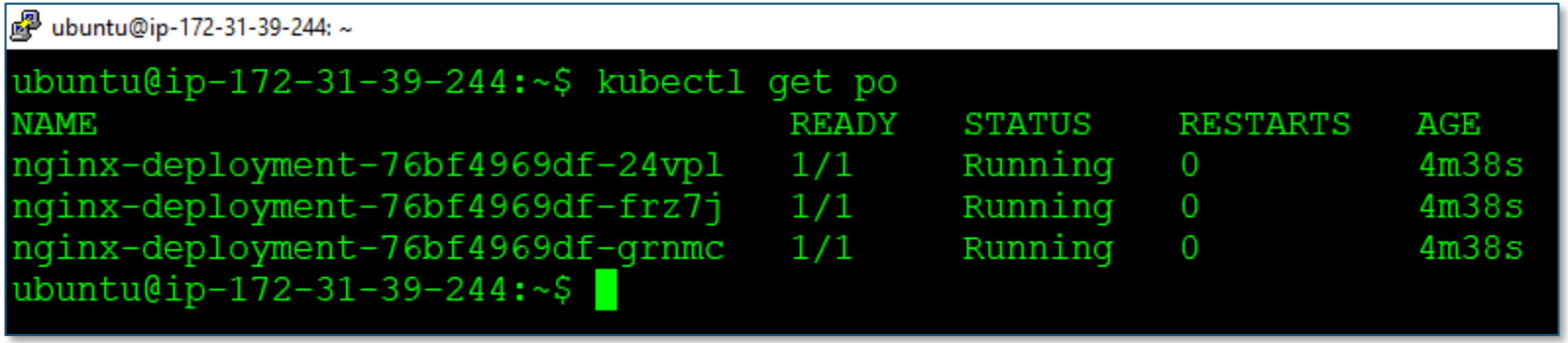


Listing the Pods

To view the pods, type the following command:

Syntax

```
kubectl get po
```

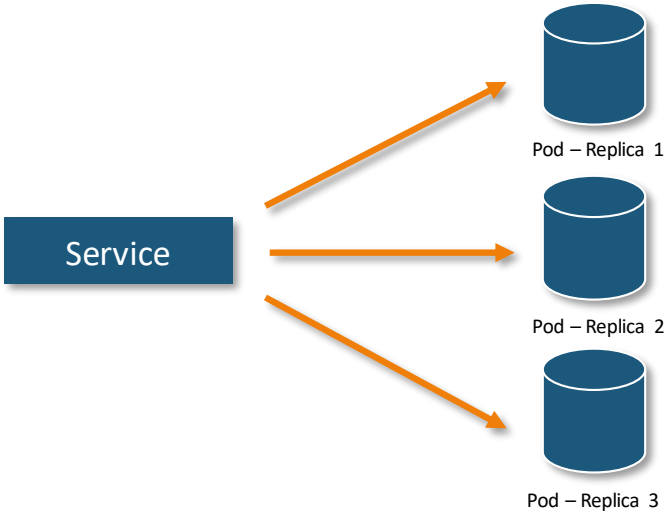


As you can see, the number of pods are matching with the number of replicas specified in the deployment file.

Creating a Service

Creating a Service

A Service is basically a round-robin load balancer for all pods, which matches with its name or selector. It constantly monitors the pods; in case a pod gets unhealthy, the service will start deploying the traffic to other healthy pods.



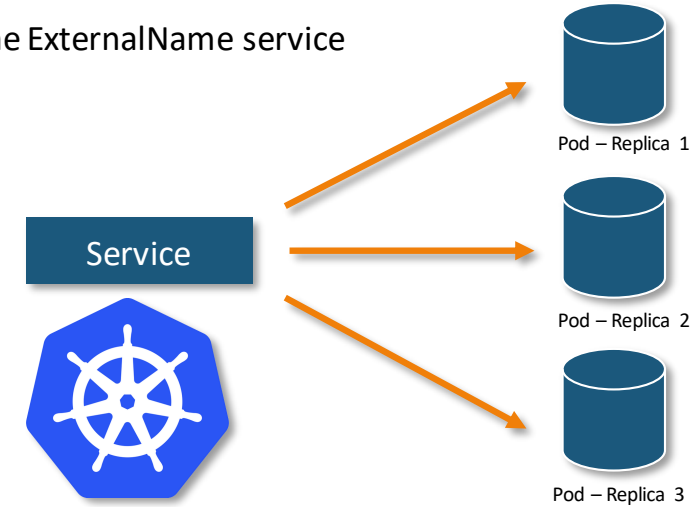
Service Types

ClusterIP: Exposes the service on cluster-internal IP

NodePort: Exposes the service on each Node's IP at a static port

LoadBalancer: Exposes the service externally using a cloud provider's load balancer

ExternalName: Maps the service to the DNS Name mentioned with the ExternalName service

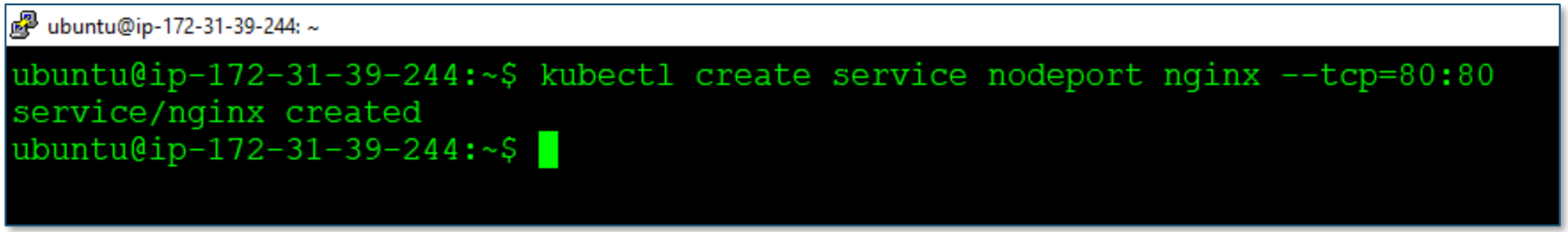


Creating a NodePort Service

We can create a NodePort service using the following syntax:

Syntax

```
kubectl create service nodeport <name-of-service> --tcp=<port-of-service>:<port-of-container>
```

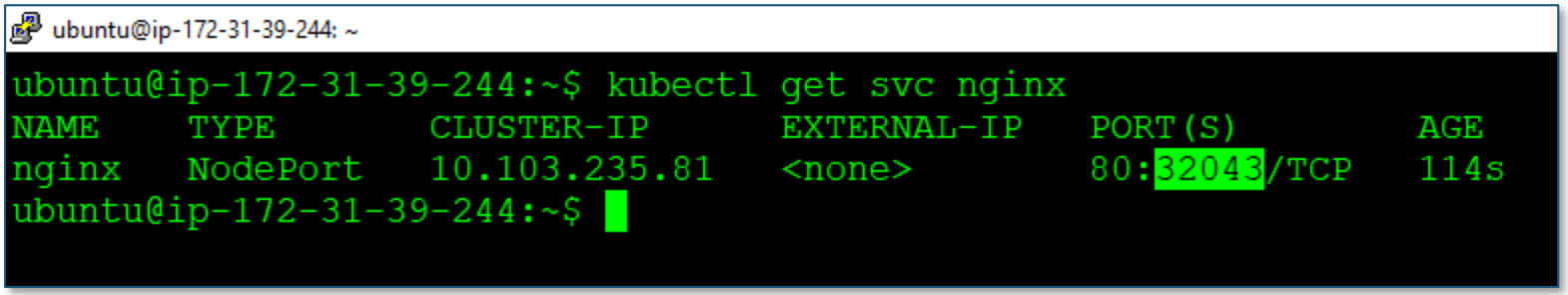


Creating a NodePort Service

To know the port, on which the service is being exposed, type the following command:

Syntax

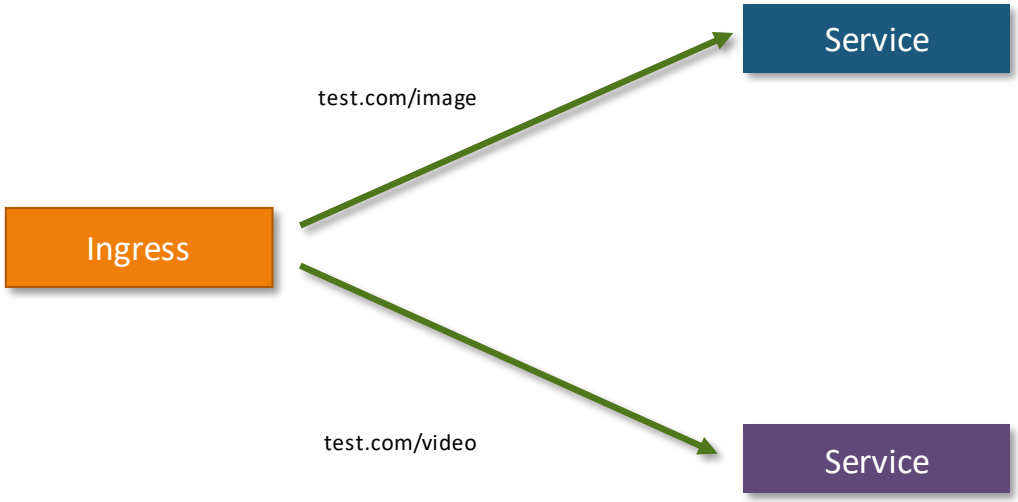
```
kubectl get svc nginx
```



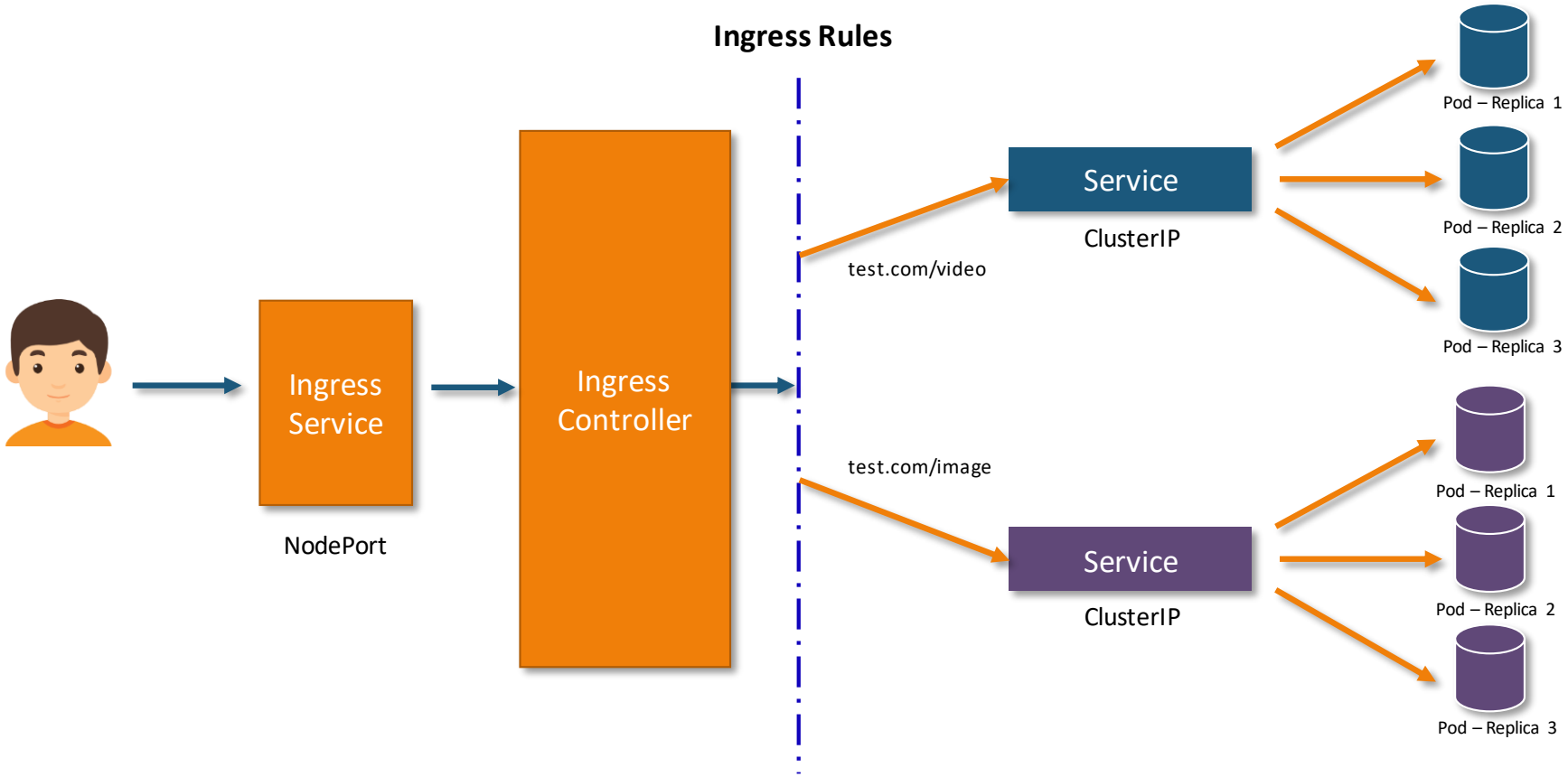
Creating an Ingress

What is an Ingress?

Kubernetes ingress is a collection of routing rules that govern how external users access services running in a Kubernetes cluster.



What is an Ingress?



Installing Ingress Controller

We will be using the nginx ingress controller for our demo. We can download it from the following link:

Link

https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md

Defining Ingress Rules

The following rule, will redirect traffic which asks for /foo to nginx service. All other requests will be redirected to ingress controller's default page.

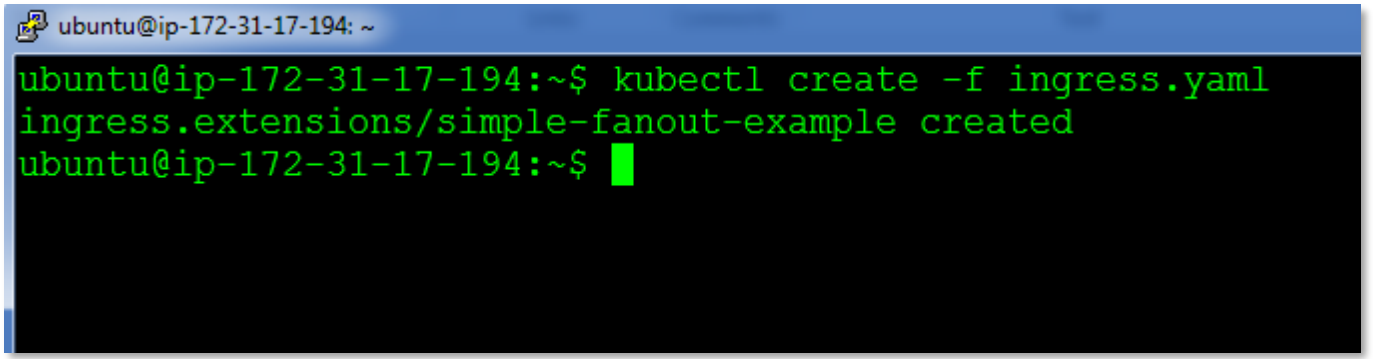
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /foo
        backend:
          serviceName: nginx
          servicePort: 80
```

Deploying Ingress Rules

To deploy ingress rules, we use the following syntax:

Syntax

```
kubectl create –f ingress.yaml
```

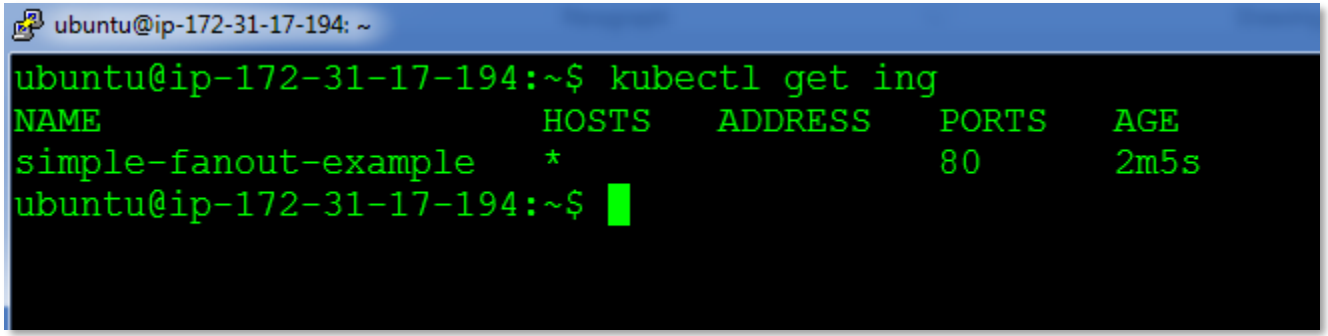


Viewing Ingress Rules

To list the ingress rules we use the following syntax:

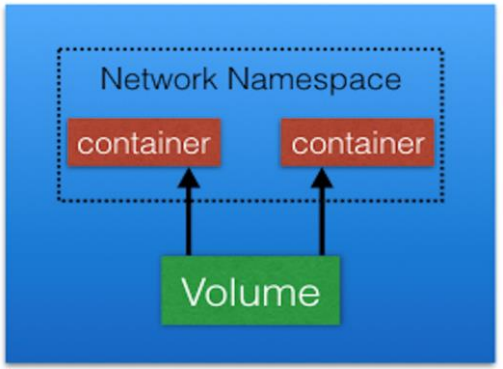
Syntax

```
kubectl get ing
```



Kubernetes: Building Blocks: Pod

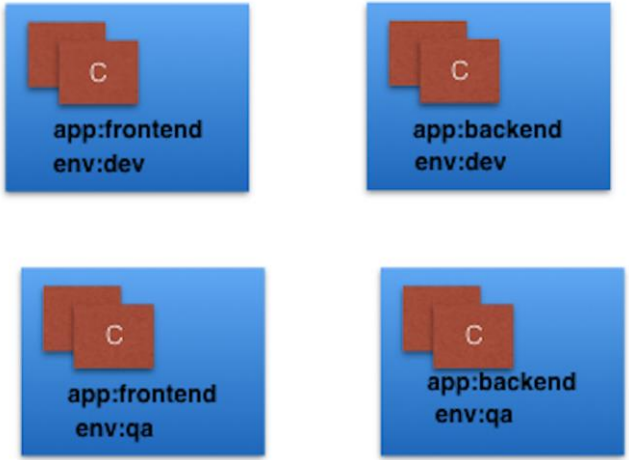
- Pod is the **smallest** and **simplest** Kubernetes **object**.
- It is the **unit** of **deployment** in Kubernetes, which represents a single instance of the application.
- A Pod is a **logical collection** of one or more **containers**, which:
    - Are **scheduled** together on the same host
    - Share the same **network** namespace
    - **Mount** the same external storage (**volumes**)
- Pods are **ephemeral** in nature, and they do not have the capability to **self-heal** by themselves.
- We use them with **controllers** like **Deployments, ReplicaSets**, which can handle a Pod's **replication**, **fault tolerance**, s**elf-heal**, etc



Kubernetes: Building Blocks: Labels

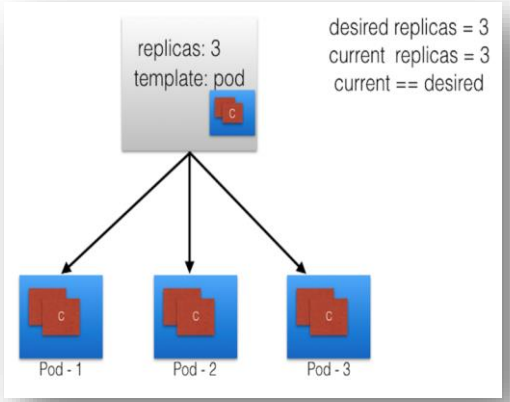
- **Labels** are **key-value** pairs that can be **attached** to any **Kubernetes objects** (e.g. Pods).
- Labels are used to **organize** and **select** a **subset** of **objects**, based on the **requirements** in place.
- Many objects can have the same Label(s).
- Labels do not provide **uniqueness** to objects.
- We have used two Labels: app and env.
- Based on our requirements, we have given different values to our four Pods

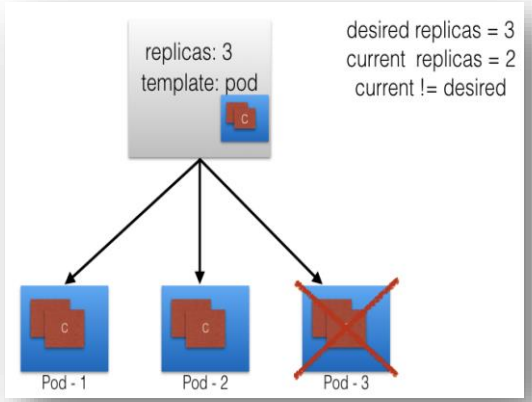
- **Label Selector**
- We can use Label Selectors, to select subset of objects
- For example**,**
- **Equity Based Selector**
  env==dev, env!=prod ,
- **Set Based Selector**
  env in (dev,qa), app notin (backend,midlleware)

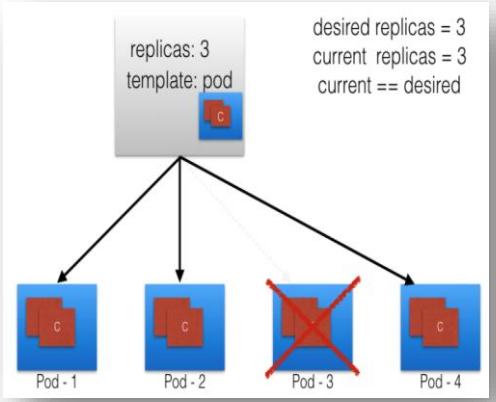


Kubernetes: Building Blocks: ReplicaSets

- **ReplicationController** 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.
- If there are more Pods than the **desired** count, the ReplicationController would **kill** the extra **Pods**, and, if there are less Pods, then the ReplicationController would **create more Pods** to match the **desired** count.
- **ReplicaSet** is the **next-generation ReplicationController**.
- **ReplicaSets** support both equality- and set-based selectors, whereas **ReplicationControllers** only support equality-based Selectors.



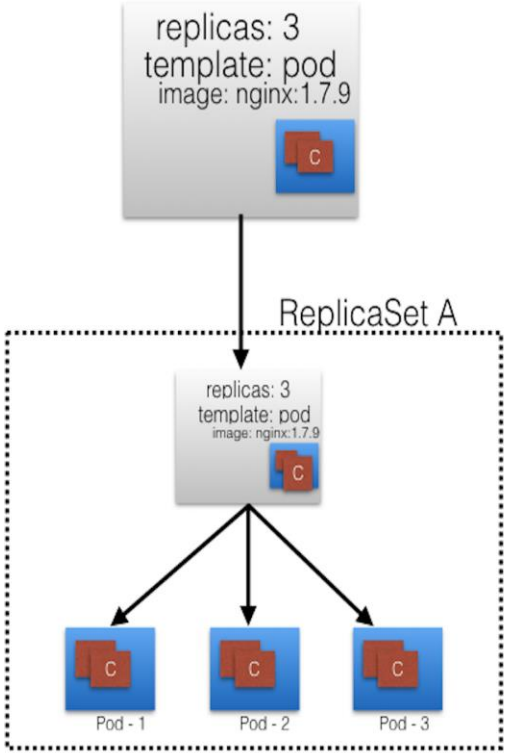




Kubernetes: Building Blocks: Deployments

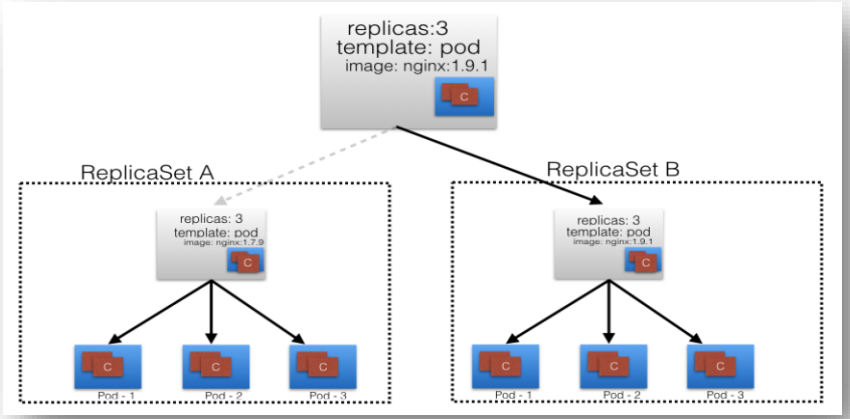
- **Deployment** objects provide **declarative** updates to Pods and **ReplicaSets**.
- The **DeploymentController** is part of the master node's **controller manager**, and it makes sure that the current state always matches the **desired state**.

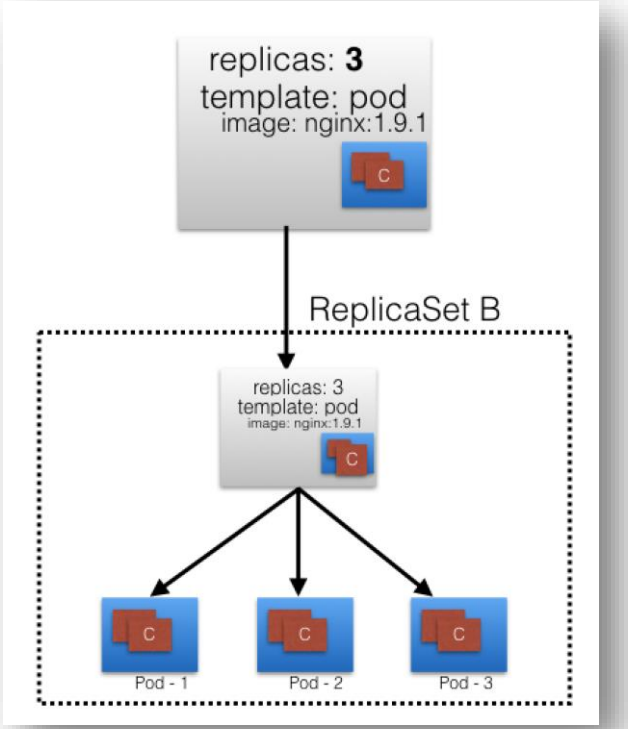
- In the following example, we have a **Deployment** which creates a **ReplicaSet** A.
- ReplicaSet A then creates 3 **Pods**.
- In each **Pod**, one of the **containers** uses the **nginx:1.7.9** image.



Kubernetes: Building Blocks: Deployment Rollout

- Now, in the **Deployment**, we change the **Pods Template** and we **update the image** for the nginx container from nginx:1.7.9 to nginx:1.9.1.
- As have **modified** the **Pods Template**, a new **ReplicaSet B** gets created.
- This process is referred to as a **Deployment rollout**.
- A **rollout** is only **triggered** when we **update** the **Pods Template** for a **deployment**.
- **Operations** like **scaling** the deployment **do not trigger** the deployment.
- Once **ReplicaSet B** is ready, the Deployment starts pointing to it.



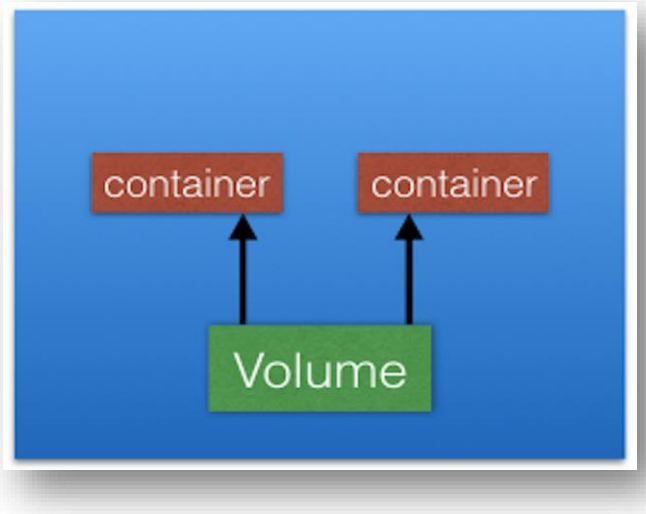


Kubernetes Networking

- In Kubernetes, each Pod gets a **unique** IP address.
- Kubernetes uses **Container Network Interface** (CNI), proposed by **CoreOS** to assign the IP address to each **Pod**.
- Inside a Pod, **containers** share the **network namespaces**, so that they can reach to each other via **localhost**.
- **Pod-to-Pod** communication across **Nodes** can be achieved via:
    - **Routable** Pods and nodes, using the **underlying physical infrastructure**, like **Google Kubernetes Engine**
    - Using **Software Defined Networking**, like Flannel, Weave, Calico, etc.
- We can access our **applications** from outside the **cluster** by exposing our **services** to the external world with **kube-proxy**

Kubernetes: Volumes

- Containers inside pods are **ephemeral** in nature
- All data stored inside a container is **deleted** if the container **crashes**
- kubelet will **restart** it with a **clean state**, which means that it will not have any of the **old data**
- To overcome this problem, Kubernetes uses **Volumes**
- A Volume is essentially a **directory** backed by a **storage** medium.
- The storage medium and its content are determined by the Volume Type
- Volume is attached to a **Pod** and shared among the **containers** of that Pod
- The **Volume** has the same **life span** as the **Pod**, and it outlives the containers of the Pod - this allows **data** to be **preserved** across container restarts.

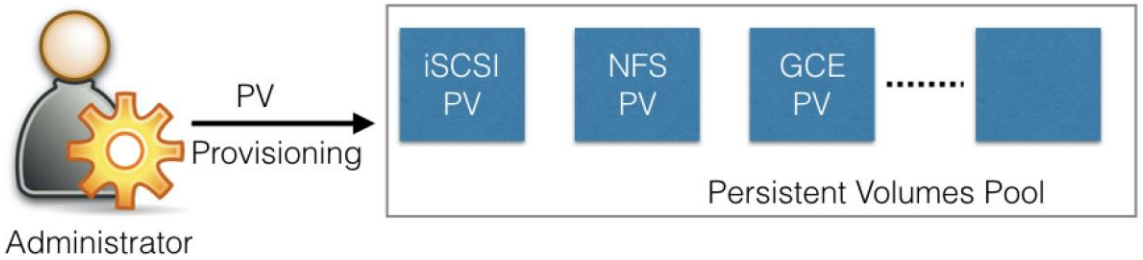


Kubernetes: Volumes Types

- A directory which is **mounted** inside a **Pod** is backed by the underlying **Volume Type**
- Volume Type decides the **properties** of the directory, like **size**, **content**, etc
- Some examples of Volume Types are:
    - emptyDir
    - hostPath
    - gcePersistentDisk
    - awsElasticBlockStore
    - nfs
    - iscsi
    - secret
    - persistentVolumeClaim

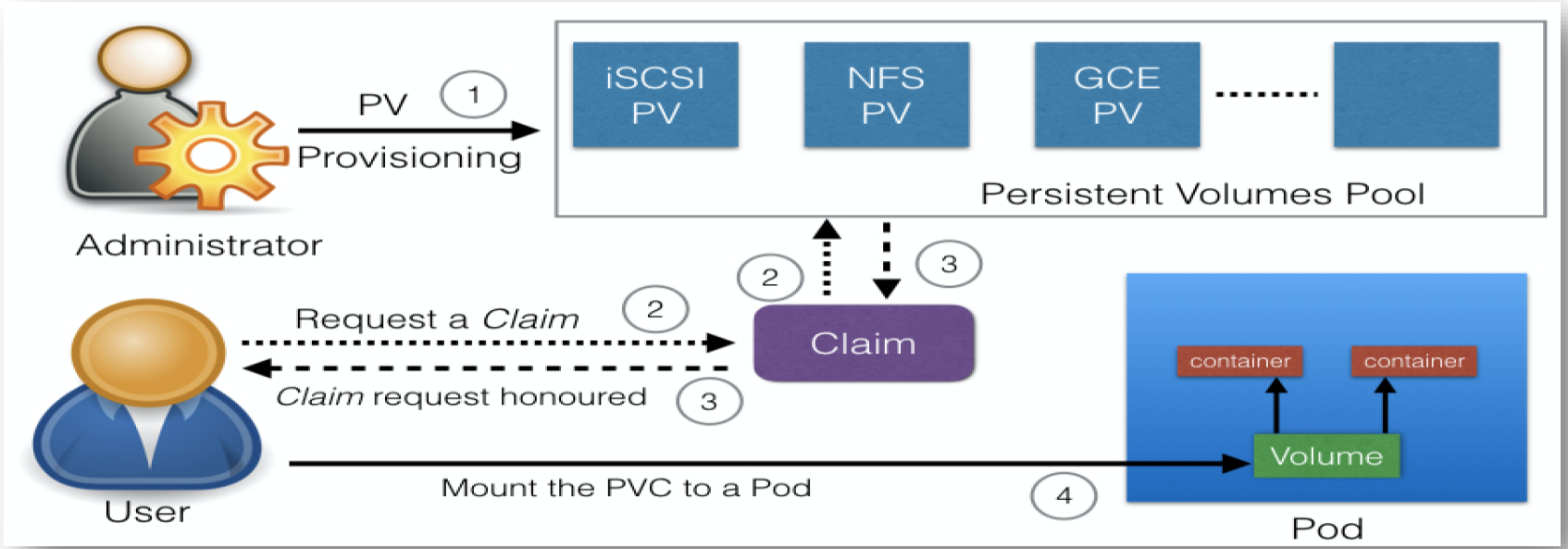
Kubernetes: PersistentVolumes

- **Persistent Volume** is a **network-attached storage** in the **cluster**, which is provisioned by the **administrator**
- PersistentVolumes can be **dynamically** provisioned based on the **StorageClass** resource
- A **StorageClass** contains pre-defined **provisioners** and parameters to create a **PersistentVolume**
- Using **PersistentVolumeClaims**, a user sends the request for dynamic PV **creation**, which gets wired to the **StorageClass** resource.
- Some of the Volume Types that support managing storage using PersistentVolumes are:
    - GCEPersistentDisk
    - AWSElasticBlockStore
    - AzureFile
    - NFS
    - iSCSI



Kubernetes: PersistentVolumeClaims

- **PersistentVolumeClaim** (PVC) is a request for **storage** by a **user**.
- Users **request** for **PersistentVolume** resources based on size, access modes, etc.
- Once a suitable **PersistentVolume** is found, it is bound to a **PersistentVolumeClaim**.
- Once a user finishes its work, the attached PersistentVolumes can be released.
- The underlying PersistentVolumes can then be reclaimed and recycled for future usage.



Kubernetes: Liveness Probe

- Liveness probe checks on an application's health, and, if for some reason, the health check fails, it restarts the affected container automatically

- Liveness Probes can be set by defining:
  - Liveness command
  - Liveness HTTP request
  - TCP Liveness Probe.
- **Liveness Command**

In this demo file liveness-exec.yaml ,
we will use existence of on file inside container as **Health Criteria .**

File: /tmp/heathy
Initial Probe: after 3 seconds
Probe Interval: 5 seconds
File deletion: 30 seconds after creation

- sudo kubectl create –f liveness-exec.yaml
- sudo kubectl get pods
- sudo kubectl describe pod liveness-exec

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep
600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
```

Kubernetes: Liveness Probe

- **Liveness HTTP**

In this demo file liveness-http.yaml ,
we will use one http response inside container
as **Health Criteria .**

- sudo kubectl create –f liveness-http.yaml
- sudo kubectl get pods
- sudo kubectl describe pod liveness-http

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: livenesshttp
  name: liveness-http
spec:
  containers:
  - name: livenesshttp
    image: httpd:latest
    ports:
      - containerPort: 80
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

Kubernetes: hostPath Volume Type

- In this Demo we will Create Nginx Service which will use hostPath Volume to get Content
- Create a Directory named “vol” on Node Machine
- Create a file named “index.html” with given content
- Create a Deployment from webserver-vol.yaml
- Create a Service from webserver-svc-vol.yaml
- Describe Service to get nodePort
- Visit Node’s IP_Address:nodePort in browser
- It will show you Nginx page with custom home page
- change index.html’s content and refresh tab in browser

```
<html>
<head>K8S Volume Demo</head>
<body>
<h1>
Welcome to Kubernetes Session
</h1>
</body>
</html>
```

index.html

```
apiVersion: v1
kind: Service
metadata:
  name: web-service
  labels:
    run: web-service
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: webserver
```

webserver-svc-vol.yaml

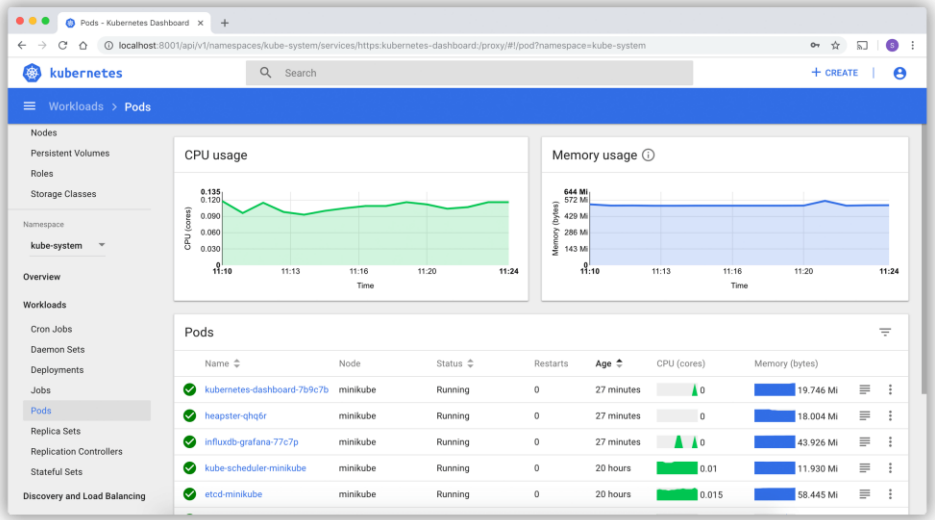
webserver-vol.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: webserver
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
        - name : hostvol
          mountPath:
/usr/share/nginx/html
      volumes:
      - name: hostvol
```

Kubernetes Dashboard

Kubernetes Dashboard

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application and manage cluster resources.

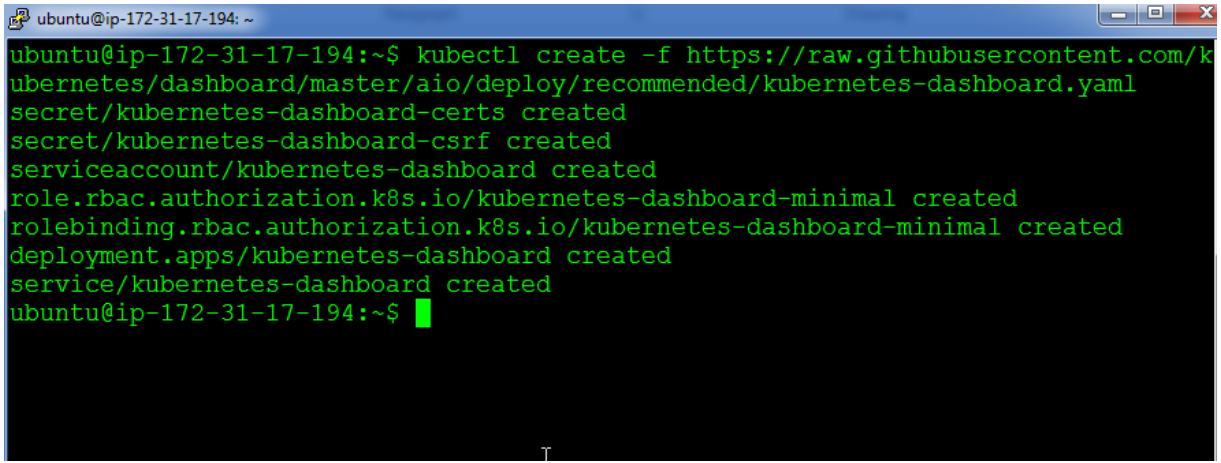


Installing Kubernetes Dashboard

To install Kubernetes Dashboard, execute the following command:

Syntax

```
kubectl create -f
https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy/recommended/kubernetes-dashboard.yaml
```

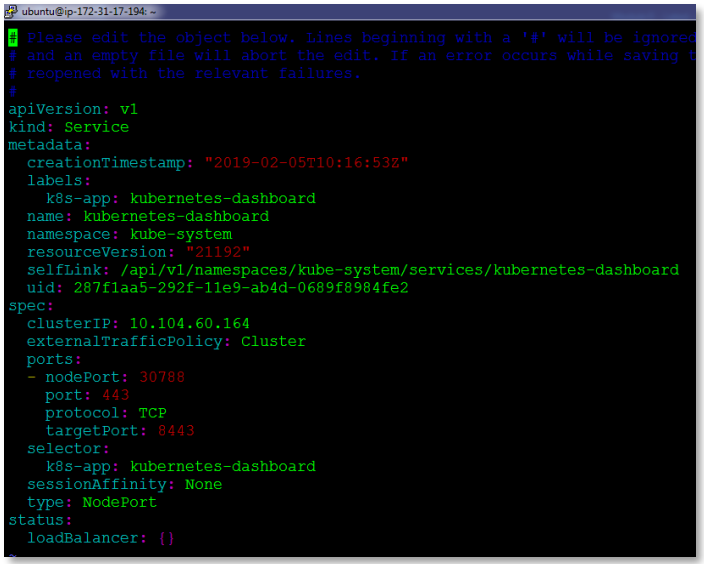


Accessing Kubernetes Dashboard

Change the service type for kubernetes-dashboard to NodePort

Syntax

```
kubectl -n kubernetes-dashboard edit service kubernetes-dashboard
```



Logging into Kubernetes Dashboard

1. Check the NodePort from the kubernetes-dashboard service
2. Browse to your cluster on the Internet browser, and enter the IP address
3. Click on Token, which will ask you for the token entry
4. Generate a token using the following command:

```
$ kubectl create serviceaccount cluster-admin-dashboard-sa
$ kubectl create clusterrolebinding cluster-admin-dashboard-sa \
  --clusterrole=cluster-admin \
  --serviceaccount=default:cluster-admin-dashboard-sa

$ TOKEN=$(kubectl describe secret $(kubectl -n kube-system get secret | awk '/^cluster-admin-dashboard-sa-token-/{print $1}') | awk '$1=="token:"{print $2}')

$ echo $TOKEN
```

5. Finally, enter the token and login to your dashboard

Hands-on: Deploying an App Using Dashboard