What is Kubernetes?

- K8s is an open-source container-orchestration system for automating deployment, scaling and management of containerized applications across clusters of hosts.
- At a high level, it takes multiple Containers running on different hosts and lets you use them together.
- Run containers anywhere on clusters of physical, virtual machines and cloud infrastructure.
- Start, stop, update, and manage a cluster of machines running containers in a consistent and maintainable way.
- A **declarative** language for launching containers.
- We will define the *desired state*, K8s will monitor the current *actual state* and synchronize it with the *desired state.*
- K8s provides the tooling to manage the **when**, **where**, and **how** many of the application stack and its components.
- K8s also allows finer control of resource usage, such as CPU, memory, and disk space across our infrastructure.

What is an API? (Application Programming Interface)

- **API** is a software intermediary that allows two applications to talk to each other. Each time you use an app like Facebook, send an instant message, or check the weather on your phone, you're using an API
- **RESTful API** is a method of allowing communication between a web-based client and server

Kubernetes Objects

- Kubernetes uses Objects to represent the state of your cluster
  - ✓What containerized applications are running (and on which nodes)
  - ✓The resources available to those applications
  - ✓The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance
- Once you create the object, the Kubernetes system will constantly work to ensure that object exists and maintain cluster's **desired state**
- Every Kubernetes object includes two nested fields that govern the object's configuration: the **object *spec*** and the **object *status***
- The ***spec***, which we provide, describes your *desired state* for the object–the characteristics that you want the object to have.
- The ***status*** describes the *actual state* of the object, and is supplied and updated by the Kubernetes system.
- All objects in the are identified by a Unique Name and a UID.

STATE OF THE OBJECT:

- ✓ Replicas (2/2)
- ✓ Image (JBOSS/MYIMAGE)
- ✓ Name
- ✓ Port
- ✓ Volume
- ✓ Startup cmd
- ✓ Detached (default)

MANAGE THE OBJECT:

- ✓ Create
- ✓ Maintain the state of your application deployment (containers)
- ✓ Delete

Kubernetes Objects

- The basic Kubernetes objects include:
  - ❑ Pod
  - ❑ Service
  - ❑ Volume
  - ❑ Namespace
- In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality:
  - ❑ReplicaSet
  - ❑Deployment
  - ❑StatefulSet
  - ❑DaemonSet
  - ❑Job

Kubernetes Objects Management

- The **kubectl** command-line tool supports several different ways to create and manage Kubernetes objects

Management technique	Operates on	Recommended environment
Imperative commands	Live objects	Development projects
Declarative object configuration	Individual files (yaml/json)	Production

- **Declarative** is about describing what you're trying to achieve, without instructing how to do it
- **Imperative** explicitly tells "how" to accomplish it

Kubernetes Configuration

All-in-One Single-Node Installation

- With all-in-one, all the master and worker components are installed on a single node. This is very useful for learning, development, and testing. This type should not be used in production. Minikube is one such example, and we are going to explore it in future chapters.

Single-Node etcd, Single-Master, and Multi-Worker Installation

- In this setup, we have a single master node, which also runs a single-node etcd instance. Multiple worker nodes are connected to the master node.

Single-Node etcd, Multi-Master, and Multi-Worker Installation

- In this setup, we have multiple master nodes, which work in an HA mode, but we have a single-node etcd instance. Multiple worker nodes are connected to the master nodes.

Infrastructure for Kubernetes Installation

- **Localhost Installation -** Minikube
- **On-Premise Installation** - Vagrant, Vmware, KVM
- **Cloud Installation**
  - ✓Google Kubernetes Engine (GKE)
  - ✓Azure Container Service (AKS)
  - ✓Amazon Elastic Container Service for Kubernetes (EKS)
  - ✓OpenShift
  - ✓Platform9
  - ✓IBM Cloud Container Service

Kubernetes Installation Tools/Resources:

- ❑ Kubeadm
- ❑ Kubespray
- ❑ kops

Setting Up a Single-Node Kubernetes Cluster with Minikube (EC2 Ubuntu)

- **Install Docker**

```
$ sudo apt update && apt -y install docker.io
```

- **Install kubectl**

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl && chmod +x ./kubectl && sudo mv ./kubectl /usr/local/bin/kubectl
```

- **Install Minikube**

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

- **Start Minikube**

```
$ apt install conntrack
$ minikube start --vm-driver=none
$ minikube status
```

POD

- A Pod is the smallest and simplest unit in the Kubernetes object model that you

create or deploy.

- A Pod represents a running process on your cluster.
- A Pod encapsulates an application container, storage resources, a unique network IP, and options that govern how the container(s) should run
- Kubernetes manages the Pods rather than the containers directly.
- Pods in a Kubernetes cluster can be used in two main ways:
  - ✓ Pods that run a single container - common usecase
  - ✓ Pods that run multiple containers that need to work together
- Each Pod is assigned a **unique IP** address.
- Every container in a Pod shares the IP address and network ports.
- Containers inside a Pod can communicate with one another using localhost.
- All containers in the Pod can access the shared volumes, allowing those containers to share data.
- Kubernetes scales pods and not containers

Understanding Pods

- When a Pod gets created, it is scheduled to run on a Node in your cluster.
- The Pod remains on that Node until the process is terminated, the pod object is deleted, the pod is evicted for lack of resources, or the Node fails.
- If a Pod is scheduled to a Node that fails, or if the scheduling operation itself fails, the Pod is deleted
- If a node dies, the pods scheduled to that node are scheduled for deletion, after a timeout period.
- A given pod (UID) is not "rescheduled" to a new node; instead, it will be replaced by an identical pod, with even the same name if desired, but with a new UID
- Volumes in a pod will exists as long as that pod (with that UID) exists. If that pod is deleted for any reason, volume is also destroyed and created as new on new pod
- Kubernetes uses a Controller, that handles the work of managing the Pod instances.
- A Controller can create and manage multiple Pods, handling replication, rollout and providing self-healing capabilities

Kubectl Usage

$ kubectl [COMMAND] [TYPE] [NAME] [flags]

COMMAND: Specifies the operation that you want to perform on one or more resources, for example create, apply, get, describe, delete, exec …

TYPE: Specifies the resource type i.e nodes, pods, services, rc, rs …

NAME: Specifies the name of the resource

flags: Optional arguments

$ kubectl version

$ kubectl get node

$ kubectl describe nodes <nodename>

Kubernetes YAML Basics

```
apiVersion: [ v1 | apps/v1 ]
Kind:  [ Pod | Deployment | Service | PersistentVolume ]          # Object we are trying to achieve
metadata:
  name: <Value>
  label: <Value>
spec:                                                            # Set of data which the defines the desired state for the resource
  containers:
     name: <Value>
     image: <Value>
     command: <Value>
     workingDir: <Value>
     ports: <Value>
     env: <Value>
     resources: <Value>
     volumeMounts: <Value>
     livenessProbe: <Value>
     readinessProbe: <Value>
```

Pod Single Container Pod - Example1: pod1.yml

```
kind: Pod                  # Object Type
apiVersion: v1             # API version
metadata:                  # Set of data which describes the Object
  name: testpod            # Name of the Object
spec:                      # Data which describes the state of the Object
  containers:              # Data which describes the Container details
    - name: c00            # Name of the Container
      image: ubuntu        # Base Image which is used to create Container
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
  restartPolicy: Never     # Defaults to Always
```

- **Always** means that the container will be restarted even if it exited with a zero exit code
- **OnFailure** means that the container will only be restarted if it exited with a non-zero exit code
- **Never** means that the container will not be restarted regardless of why it exited.

- Create or update an Object

$ **kubectl create -f pod1.yml** (or)

$ **kubectl apply -f pod1.yml**

- Get the list of the pod objects available

$ **kubectl get pods**

$ **kubectl get pods -o wide**

$ **kubectl get pods --all-namespaces**

- Get the details of the pod object

$ **kubectl describe pod <podname>**

- Get running logs from the container inside the pod object

$ **kubectl logs -f <podname>**

$ **kubectl logs -f <podname> -c <containername>**

Pod Annotate - Example2: podannotate.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: testpod2
  annotations:                    # comments to describe the Object usage
    description: our first testpod
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

$ **kubectl apply -f podannotate.yml**

- Run OS commands in an existing pod (1 container pod)

$ **kubectl exec <podname> -- <OScmd>**

$ **kubectl exec testpod2 -- hostname -i**

- Run OS commands in an existing container(multi container pod)

$ **kubectl exec <podname> -c <containername> -- <OScmd>**

- Attach to the running container interactively

$ **kubectl exec <podname> -i -t -- /bin/bash (or)**

$ **kubectl attach <podname> -i**

- Delete a Pod

$ **kubectl delete pods <podname> (or)**

$ **kubectl delete -f <YAML>**

Pod Multi-container Pod - Example3: podmulcont.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: testpod3
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    - name: c01
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Students; sleep 5 ; done"]
```

$ **kubectl apply -f podmulcont.xml**

Example 05: podwithports.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: testservice
  labels:
    myvalue: demo                    # Label for applying service later
spec:
  containers:
    - name: c00
      image: httpd
      ports:
        - containerPort: 80          # Expose port 80 from container
```

$ kubectl describe pod <podname>

Labels & Selectors

- **Labels** are the mechanism you use to organize Kubernetes objects
- A label is a key-value pair without any predefined meaning that can be attached to the Objects
- Labels are similar to Tags in AWS or Git where you use a name to quick reference
- So you're free to choose labels as you need it to refer an environment which is used for Dev or Testing or Production, refer an product group like DepartmentX, DepartmentY
- Multiple labels can be added to a single object

Pod Labels – Example6: podlabels.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: labelspod
  labels:                            # Specifies the Label details under it
    <Keyname1>: <value>
    <Keyname2>: <value>
spec:
    containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

$ kubectl apply -f podlabels.yml

- To see list of pods available with details of Labels if any attached to it

$ kubectl get pods --show-labels

- Add a label to an existing pod

$ kubectl label pods <podname> <labelkey>=<value>

- List pods matching a label

$ kubectl get pods -l <label>=<value>

- We can also delete pods based on label selection

$ kubectl delete pods -l <label>=<value>

Pod Labels – Example6: podlabels.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: labelspod
  labels:                            # Specifies the Label details under it
    <Keyname1>: <value>
    <Keyname2>: <value>
spec:
    containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

Label Selectors

- Unlike name/UIDs, Labels do not provide uniqueness, as in general, we can expect many objects to carry the same label
- Once labels are attached to an object, we would need filters to narrow down and these are called as Label **Selectors**
- The API currently supports two types of selectors: ***equality-based*** and ***set-based***
- A label selector can be made of multiple *requirements* which are comma-separated

Equality-based **requirement: (= , !=)**

environment = production
tier != frontend

Set-based **requirement: (in,notin and exists)**

environment in (production, qa)
tier notin (frontend, backend)

- K8s also support set-based selectors i.e match multiple values

$ kubectl get pods -l 'label in (value1, value2)'

$ kubectl get pods -l 'env in (development, testing)'

- List pods matching multiple values

$ kubectl get pods -l environment=testing, tier=frontend

Node Selectors

- One use case for selecting labels is to constrain the set of nodes onto which a pod can schedule i.e You can tell a pod to only be able to run on particular nodes
- Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement, but on certain circumstances we might need it
- We can use labels to tag **Nodes**
- You the nodes are tagged, you can use the Label Selectors to specify the pods run only of **specific nodes**
  - First we tag the node
  - Use nodeSelector to the pod configuration

Node Selectors – Example7: nodelabels.yml

```
kind: Pod
apiVersion: v1
metadata:
  name: nodelabels
  labels:
    env: development
spec:
    containers:
      - name: c00
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
    nodeSelector:                    # specifies which node to run the pod
      hardware: t2-medium
```

$ kubectl apply -f nodelabels.yml

$ kubectl describe node <nodename>

$ kubectl get pods

- you would see the status for this new pod is pending, as there are no nodes which has the label

$ kubectl describe pods <podname>

- Add a label to an node

$ kubectl label nodes <nodename> <labelkey>=<value>

$ kubectl label nodes minikube hardware=t2-medium

$ kubectl describe node <nodename>

$ kubectl get pods

Scaling & Replication

- Kubernetes was designed to orchestrate multiple containers and replication.
- Need for multiple containers/replication helps us with these:

Reliability: By having multiple versions of an application, you prevent problems if one or more fails.

Load balancing: Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node

Scaling: When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application, adding additional instances as needed

Rolling updates: updates to a service by replacing pods one-by-one

Replication Controller

- A Replication Controller is a object that enables you to easily create multiple pods, then make sure that that number of pods always exists.
- If a pod created using RC will be automatically replaced if they does crash,fail, deleted or terminated
- Using RC is recommended if you just want to make sure 1 pod is always running, even after system reboots
- You can run the RC with 1 replica & the RC will make sure the pod is always running

RC – Example8: rc.yml

```
kind: ReplicationController          # this defines to create the object of replication type
apiVersion: v1
metadata:
  name: replicationcontroller
spec:
  replicas: 2                        # this element defines the desired number of pods
  selector:                          # tells the controller which pods to watch/belong to this Replication Controller
    myname: adam                     # these must match the labels
  template:                          # template element defines a template to launch a new pod
    metadata:
      name: testpod6
      labels:                        # selector values need to match the labels values specified in the pod template
        myname: adam
    spec:
      containers:
        - name: c00
          image: ubuntu
          command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

```
$ kubectl apply -f rc.yml
$ kubectl get pods --show-labels
$ kubectl get pods -l myname=adam
$ kubectl get rc
```

- Delete the pod & RC will recreate it

```
$ kubectl delete pods <podname>
$ kubectl describe rc <replicationcontrollername>
```

- Scale Replicas:

```
$ kubectl scale --replicas=<num>  rc/<replicationcontrollername>   (OR)
$ kubectl scale --replicas=<num> <resourcetype> -l <key>=<value>
$ kubectl scale --replicas=2 rc -l myname=adam
```

- Delete Replicationcontroller:

```
$ kubectl delete rc <replicationcontrollername>
```

Replication Set

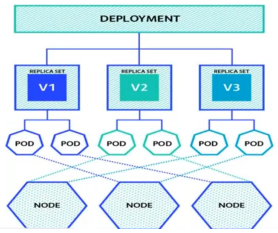
- ReplicaSet is the next generation Replication Controller
- The replication controller only supports equality-based selector whereas the replica set supports set-based selector i.e filtering according to set of values
- ReplicaSet rather than the Replication Controller is used by other objects like Deployment

RS – Example9: rc.yml

```
kind: ReplicaSet                     # Defines the object to be ReplicaSet
apiVersion: apps/v1                  # Replicaset is not available on v1
metadata:
  name: myreplicaset
spec:
  replicas: 2                        # this element defines the desired number of pods
  selector:                          # tells the controller which pods to watch/belong to this Replication Set
    matchExpressions:                # these must match the labels
      - {key: myname, operator: In, values: [adam, adamm, aadam]}
      - {key: env, operator: NotIn, values: [production]}
  template:                          # template element defines a template to launch a new pod
    metadata:
      name: testpod7
      labels:                        # selector values need to match the labels values specified in the pod template
        myname: adam
    spec:
      containers:
        - name: c00
          image: ubuntu
          command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

Deployments

- Just using RC & RS might be cumbersome to deploy apps, update or rollback apps in the cluster
- A **Deployment** object acts as a supervisor for pods, giving you fine-grained control over how and when a new pod is rolled out, updated or rolled back to a previous state
- When using Deployment object, we first define the **state** of the App, then k8s master schedules mentioned app instance onto specific individual Nodes
- K8s then monitors, if the Node hosting an instance goes down or pod is deleted, the Deployment controller replaces it.
- This provides a self-healing mechanism to address machine failure or maintenance.



The following are typical use cases for Deployments:

- **Create a Deployment to rollout a ReplicaSet**. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- **Declare the new state of the Pods** by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the **revision** of the Deployment.
- **Rollback to an earlier Deployment revision** if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- **Scale up the Deployment** to facilitate more load.
- **Pause the Deployment** to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- **Use the status of the Deployment** as an indicator that a rollout has stuck.
- **Clean up older ReplicaSets** that you don't need anymore

Deployments – Example10: deployments.yml

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: mydeployments
spec:
  replicas: 2
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: deployment
  template:
    metadata:
      name: testpod8
      labels:
        name: deployment
    spec:
      containers:
        - name: c00
          image: ubuntu
          command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5; done"]
```

- **Creating a Deployment**

$ kubectl apply -f pod8.xml
$ kubectl get deploy
$ kubectl describe deploy/<deploymentname>
$ kubectl get rs
$ kubectl get pods

- **Updating a Deployment**

Change the echo cmd & save the yml content as new file & apply it:
$ kubectl apply –f <yml>
$ kubectl get pods
✓ We will see the rollout of two new pods with the updated cmd, as well as old pods being terminated
✓ Also, a new replica set has been created by the deployment
$ kubectl get rs

- Recovering from crashes

$ kubectl delete pod <podname>

- **Scaling a Deployment**

$ kubectl scale --replicas=<num> deployment/<deploymentname>

Manage the rollout of a resource

Usage $ kubectl rollout SUBCOMMAND OBJECT

history	View rollout history
pause	Mark the provided resource as paused
resume	Resume a paused resource
status	Show the status of the rollout
undo	Undo a previous rollout

- **Deployment status**

$ kubectl rollout status deployment/<deploymentname>

- **Deployment history**

$ kubectl rollout history deploy/<deploymentname>

- **Rolling Back a Deployment**

If there are problems in the deployment Kubernetes will automatically roll back to the previous version, however you can also explicitly roll back to a specific revision, as in our case to revision 1 (the original pod version)

$ kubectl rollout undo deploy/<deploymentname> --to-revision=1
$ kubectl rollout undo deploy/<deploymentname>

- **Set the image of the deployment**

$ kubectl set image deployment/<deploymentname> <containername>=<image>

- **Remove the deployment, with it the replica sets and pods it supervises**

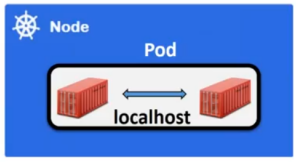
$ kubectl delete deploy <deploymentname>

- **Pausing and Resuming a Deployment**

$ kubectl rollout pause deploy/<deploymentname>
$ kubectl rollout resume deploy/<deploymentname>

- **Container to Container communication** on same Pod happens through **localhost** within the containers

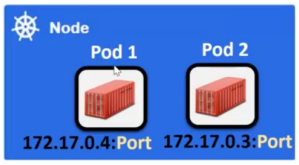
```
kind: Pod
apiVersion: v1
metadata:
  name: testpod
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while
true; do echo Hello-Adam; sleep 5 ; done"]
    - name: c01
      image: httpd
      ports:
       - containerPort: 80
```



$ kubectl exec testpod -it -c c00 -- /bin/bash
- curl localhost:80 # If curl is not there, run: apt update && apt install curl

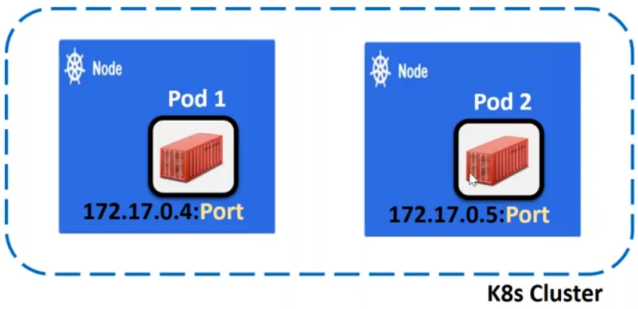
- **Pod to Pod communication** on same worker happens through **Pod IP**
- By default Pod's IP will not be **accessible outside** the **Node**

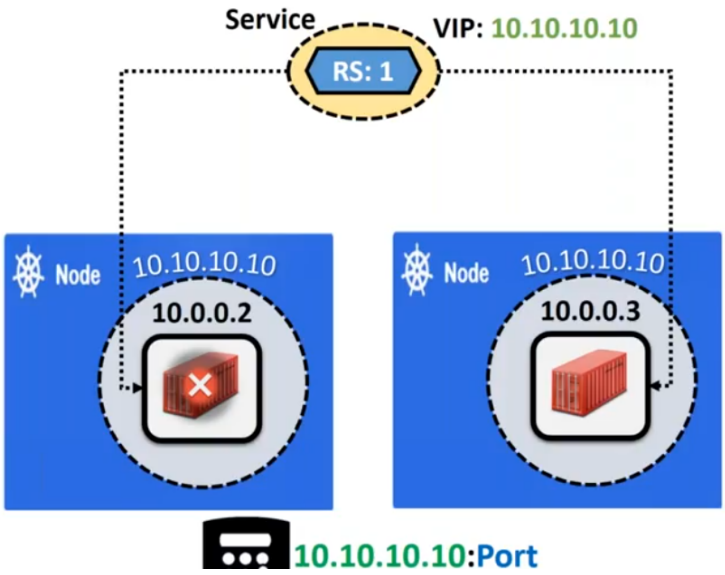
```
kind: Pod
apiVersion: v1
metadata:
  name: testpod4
spec:
  containers:
    - name: c01
      image: nginx
      ports:
       - containerPort: 80
```



$ curl <Pod1-IP>:80
$ curl <Pod2-IP>:80

- **Pod to Pod communication** on different worker?
- **Accessing Pod outside Cluster ?**





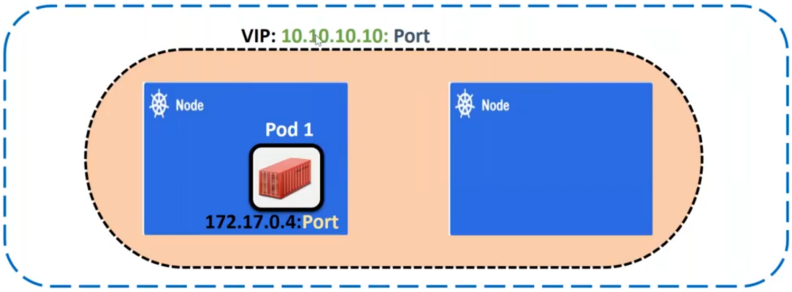
Services

- When using RC, pods are terminated and created during scaling or replication operations
- When using Deployments, while updating the image version the pods are terminated & new pods take the place of older pods.
- Pods are very dynamic i.e they come & go on the k8s cluster and on any of the available nodes & it would be difficult to access the pods as the pods IP changes once its recreated
- Service Object is an logical bridge between pods & endusers, which provides Virtual IP (VIP) address.
- Service allows clients to reliably connect to the containers running in the pod using the VIP.
- The VIP is not an actual IP connected to a network interface, but its purpose is purely to forward traffic to one or more pods.
- kube-proxy is the one which Keeps the mapping between the VIP and the pods up-to-date, which queries the API server to learn about new services in the cluster.

- Although each Pod has a unique IP address, those IPs are not exposed outside the cluster.
- Services helps to expose the VIP mapped to the pods & allows applications to receive traffic
- Labels are used to select which are the Pods to be put under a Service.
- Creating a Service will create an endpoint to access the pods/Application in it.
- Services can be exposed in different ways by specifying a type in the Service Spec:
  - ✓**ClusterIP** (default)
  - ✓**NodePort**
  - ✓**LoadBalancer -** Created by cloud providers that will route external traffic to every node on the NodePort (ex: ELB on AWS).
  - ✓**Headless -** creates several Endpoints that are used to produce DNS records. Each DNS record is bound to a pod
- By default service can run only between ports **30000-32767**

CLUSTER IP

- Exposes VIP only reachable from within the cluster
- Mainly used to communicate between components of Microservices




```
kind: Deployment
apiVersion: apps/v1
metadata:
   name: mydeployments
spec:
   replicas: 1
   selector:      # tells the controller which pods to watch/belong to
    matchLabels:
     name: deployment
   template:
     metadata:
       name: testpod8
       labels:
         name: deployment
     spec:
      containers:
        - name: c00
          image: httpd
          ports:
          - containerPort: 80
```

Services – Example11: service.yml

```
kind: Service                 # Defines to create Service type Object
apiVersion: v1
metadata:
 name: demoservice
spec:
 ports:
  - port: 80                  # Containers port exposed
    targetPort: 80            # Pods port
 selector:
  myvalue: demo               # Apply this service to any pods which has the specific label
 type: ClusterIP              # Specifies the service type i.e ClusterIP or NodePort
```

```
kind: Service
apiVersion: v1
metadata:
  name: demoservice
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    name: deployment
  type: ClusterIP
```

- List the services available in the cluster

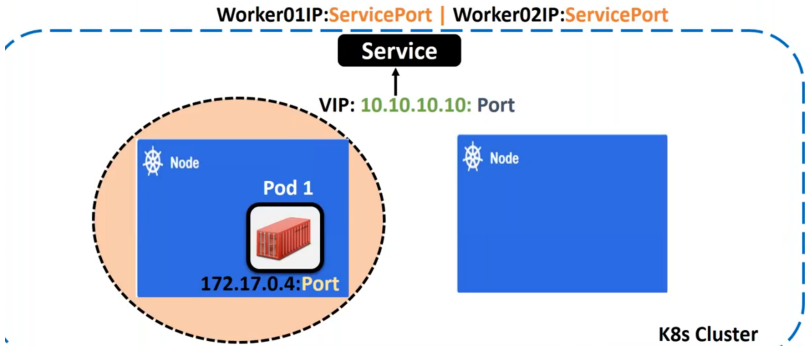
```
$ kubectl apply –f service.yml
$ kubectl get svc
$ kubectl describe pod <podname>
  - curl <podIP>:80
$ kubectl describe svc <servicename>
  - curl <serviceIP>:80
```

- Delete a service

```
$ kubectl delete svc <servicename>
```

NODEPORT

- Makes a Service accessible from outside the cluster
- Exposes the Service on the same port of each selected Node in the cluster using NAT.



Services – Example12: service.yml

```
kind: Service                 # Defines to create Service type Object
apiVersion: v1
metadata:
 name: demoservice
spec:
 ports:
  - port: 80                  # Containers port exposed
    targetPort: 80            # Pods port
 selector:
  myvalue: demo               # Apply this service to any pods which has the specific label
 type: NodePort
```

Volumes

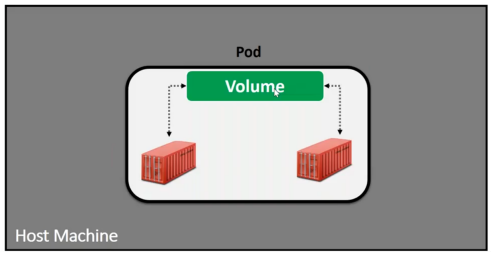
- Containers are ephemeral (short lived in nature).
- All data stored inside a container is deleted if the container crashes. However, the kubelet will restart it with a clean state, which means that it will not have any of the old data
- To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium and its content are determined by the Volume Type.
- In Kubernetes, a 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.

Volume Types

A Volume Type decides the properties of the directory, like size, content, etc. Some examples of Volume Types are:

- node-local types such as **emptyDir** and **hostPath**
- file-sharing types such as **nfs**
- cloud provider-specific types like **awsElasticBlockStore**, **azureDisk**, or **gcePersistentDisk**
- distributed file system types, for example **glusterfs** or **cephfs**
- special-purpose types like **secret**, gitRepo
- **persistentVolumeClaim**

EmptyDir



EmptyDir

- Use this when we want to share contents between **multiple containers** on the **same pod** & not to the host machine
- An **emptyDir** volume is first created when a Pod is assigned to a Node, and exists as long as that Pod is running on that node.
- As the name says, it is initially **empty**.
- Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container.
- When a Pod is removed from a node for any reason, the data in the emptyDir is deleted forever.
- A Container crashing does **not** remove a Pod from a node, so the data in an emptyDir volume is safe across Container crashes.

emptyDir – Example: emptyDir.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: myvolemptydir
spec:
  containers:
  - name: c1
    image: centos
    command: ["/bin/bash", "-c", "sleep 10000"]
    volumeMounts:                    # Mount definition inside the container
      - name: xchange
        mountPath: "/tmp/xchange"    # Path inside the container to share
  - name: c2
    image: centos
    command: ["/bin/bash", "-c", "sleep 10000"]
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:                           # Definition for host
  - name: xchange
    emptyDir: {}
```

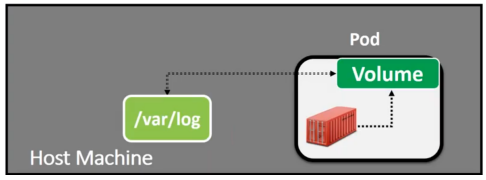
```
$ kubectl apply –f emptyDir.yml
$ kubectl describe pods myvolemptydir
```

- create a file in c1 container

```
$ kubectl exec myvolemptydir -c c1 -i -t -- bash
➢ echo "Testing" > /tmp/xchange/test
$ kubectl exec myvolemptydir -c c2 -i -t – bash
➢ cat /tmp/xchange/test
➢ mount|grep "/tmp/data"
```

hostPath

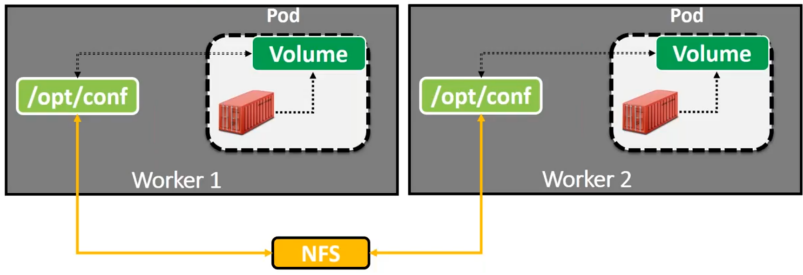
- Use this when we want to access the content of a pod/container from hostmachine
- A hostPath volume mounts a file or directory from the host node's filesystem into your Pod.



hostPath – Example: hostPath.yml

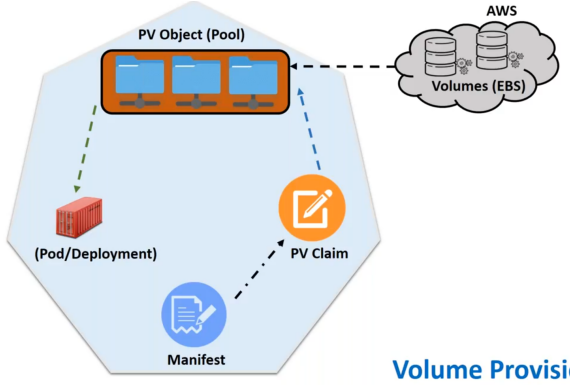
```
apiVersion: v1
kind: Pod
metadata:
  name: myvolhostpath
spec:
  containers:
  - image: centos
    name: testc
    command: ["/bin/bash", "-c", "sleep 10000"]
    volumeMounts:
    - mountPath: /tmp/hostpath
      name: testvolume
  volumes:
  - name: testvolume
    hostPath:
      path: /tmp/data    # directory location on host
      # If this dir doesnt exist at the given path, an empty directory will be created with 755
```

PersistentVolumes



PersistentVolumes

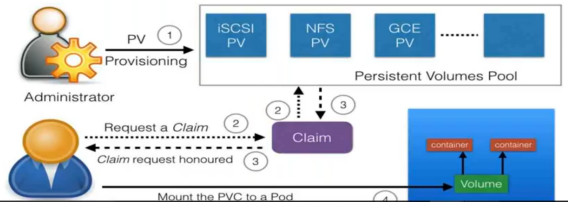
- In a typical IT environment, storage is managed by the storage/system administrators. The end user will just get instructions to use the storage, but does not have to worry about the underlying storage management.
- In the containerized world, we would like to follow similar rules, but it becomes challenging, given the many Volume Types we have seen earlier. Kubernetes resolves this problem with the **PersistentVolume (PV)** subsystem.
- A persistent volume (PV) is a cluster-wide resource that you can use to store data in a way that it persists beyond the lifetime of a pod.
- The PV is not backed by locally-attached storage on a worker node but by networked storage system such as EBS or NFS or a distributed filesystem like Ceph.
- K8s provides APIs for users and administrators to manage and consume storage. To manage the Volume, it uses the PersistentVolume API resource type, and to consume it, uses the PersistentVolumeClaim API resource type.



Volume Provisioning

PersistentVolumeClaims

- In order to use a PV you need to claim it first, using a persistent volume claim (PVC).
- The PVC requests a PV with your desired specification (size, access modes, speed, etc.) from Kubernetes and once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.
- After a successful bound to a pod, you can mount it as a volume.
- Once a user finishes its work, the attached PersistentVolumes can be released. The underlying PersistentVolumes can then be reclaimed and recycled for future usage.



awsElasticBlockStore

- An awsElasticBlockStore volume mounts an Amazon Web Services (AWS) EBS Volume into your Pod. Unlike emptyDir, which is erased when a Pod is removed, the contents of an EBS volume are preserved and the volume is merely unmounted.
- This means that an EBS volume can be pre-populated with data, and that data can be "handed off" between Pods.
- There are some restrictions when using an awsElasticBlockStore volume:
  - ✓the nodes on which Pods are running must be AWS EC2 instances
  - ✓those instances need to be in the same region and availability-zone as the EBS volume
  - ✓EBS only supports a single EC2 instance mounting a volume

PersistentVolumeClaim – Example15: pvc.yml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myebsvolclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

$ kubectl apply –f pvc.yml

$ kubectl get pvc

AWS PersistentVolume – Example: ebspv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myebsvol
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  awsElasticBlockStore:
    volumeID: <MY-EBS-VOLUME-ID>
    fsType: ext4
```

$ kubectl apply –f ebspv.yml

Azure PersistentVolume – Example15: azpv.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: myazvol
spec:
  capacity:
    storage: 1Gi
  storageClassName: default
  accessModes:
    - ReadWriteOnce
  azureDisk:
    kind: Managed
    diskName: <diskname>
    diskURI: /subscriptions/{subscription-id}/resourceGroups/{resource-group-name}/providers/Microsoft.Compute/disks/{diskname}
    fsType: ext4
  persistentVolumeReclaimPolicy: Retain
  claimRef:
    name: myebsvolclaim
```

$ kubectl apply –f azpv.yml

$ kubectl get pv

Deployment with PersistentVolume –

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pvdeploy
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
    matchLabels:
      app: mypv
  template:
    metadata:
      labels:
        app: mypv
    spec:
      containers:
      - name: shell
        image: centos
        command: ["bin/bash", "-c", "sleep 10000"]
        volumeMounts:
        - name: mypd
          mountPath: "/tmp/persistent"
      volumes:
        - name: mypd
          persistentVolumeClaim:
            claimName: myebsvolclaim
```

$ kubectl apply -f ebsdeploy.yml

$ kubectl get pods

$ kubectl get pvc

$ kubectl get pv

$ kubectl exec -it <podname> -- bash

- touch /tmp/persistent/TESTFILE

- Create a new deployment/pod & you can see the same file exists on new pod

$ kubectl exec -it <newpodname> -- bash

- ls /tmp/persistent/TESTFILE

$ kubectl delete pvc <pvclaimname>

Health Checks

- A Pod is considered ready when all of its Containers are ready.
- In order to verify if a container in a pod is **healthy** and **ready** to serve traffic, Kubernetes provides for a range of health checking mechanisms
- **Health checks**, or **probes** are carried out by the kubelet to determine when to restart a container (for livenessProbe) and used by services and deployments to determine if a pod should receive traffic (for readinessProbe).
- For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.
- One use of readiness probes is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.
- For running healthchecks we would use cmds specific to the application.
- If the command succeeds, it **returns 0**, and the kubelet considers the Container to be **alive** and **healthy**. If the command returns a **non-zero value**, the kubelet kills the

LivenessProbe With CMD – Example12: livenessprobecmd.yml

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: mylivenessprobe
spec:
  containers:
  - name: liveness
    image: ubuntu
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 1000
    livenessProbe:                  # define the health check
      exec:
        command:                    # command to run periodically
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5        # Wait for the specified time before it runs the first probe
      periodSeconds: 5              # Run the above command every 5 sec
```

```
    initialDelaySeconds: 15          # Wait
    periodSeconds: 5
    timeoutSeconds: 30
```

```
8:11:00 - container started
    :15 - application started
------------------------------ health started
8:11:20 - first health check
```

Using ConfigMap through Volumes – Example: volconfig.yml

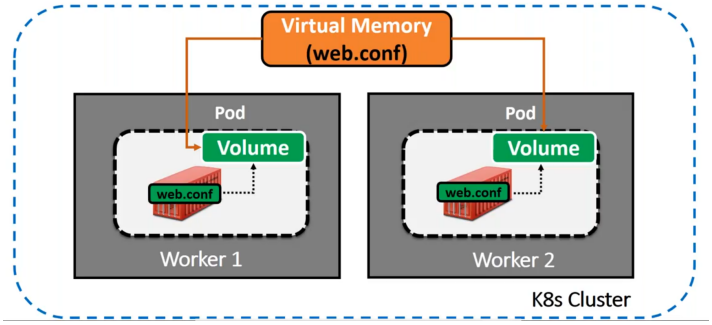
```
apiVersion: v1
kind: Pod
metadata:
 name: myvolconfig
spec:
 containers:
 - name: c1
   image: centos
   command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
   volumeMounts:
    - name: testconfigmap
      mountPath: "/tmp/config"   # the config files will be mounted as ReadOnly by default here
 volumes:
 - name: testconfigmap
   configMap:
     name: <mapname>   # this should match the config map name created in the first step
     items:
     - key: <filename> # the name of the file used during creating the map
       path: <filename> # the name of the file to be present inside the volume
```

$ kubectl apply -f volconfig.yml

- Create a container & verify the Configs mounted

$ kubectl exec myvolconfig -i -t -- /bin/bash

- ls /tmp/config (readonly)

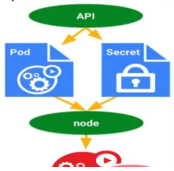


Using Configmap through Environment Variables – Example: envconfig.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: myenvconfig
spec:
 containers:
 - name: c1
   image: centos
   command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
   env:
   - name: MYENV       # env name in which value of the key is stored
     valueFrom:
       configMapKeyRef:
         name: <mapname>      # name of the config created
         key: <keyname>       # name of the file or key to be used
```

Secrets

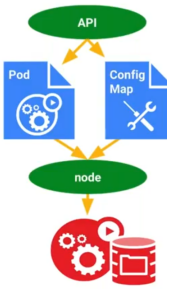
- You don't want sensitive information such as a database password or an API key kept around in clear text
- Secrets provide you with a mechanism to use such information in a safe and reliable way with the following properties:
  - ✓Secrets are namespaced objects, that is, exist in the context of a namespace
  - ✓You can access them via a volume or an environment variable from a container running in a pod
  - ✓The secret data on nodes is stored in tmpfs volumes (Tmpfs is a file system which keeps all files in virtual memory. Everything in tmpfs is temporary in the sense that no files will be created on your hard drive.)
  - ✓A per-secret size limit of 1MB exists
  - ✓The API server stores secrets as plaintext in etcd



- Secrets can be created :
  - ❖ from a text file
  - ❖ from a Yaml file
- Secrets can be accessed in following ways:
  - ❖ use secrets as environment variables
  - ❖ use secrets as volumes in the pod

Create a ConfigMap:

- We can create configmaps:
  - ❖ From command
  - ❖ From files



- ConfigMap can be accessed in following ways:
  - ❖ As environment variables
  - ❖ As volumes in the pod

- **Create ConfigMaps from files**

$ kubectl create configmap <mapname> --from-file=<filetoread>

$ kubectl create configmap mymap --from-file=sample.conf

$ kubectl describe configmaps/<mapname>

$ kubectl get configmap <mapname> -o yaml

- Create a secret from a text file or dir (generic type)

$ echo "root" > username.txt; echo "password" > password.txt

$ kubectl create secret <type> <secretname> --from-file=<filetoread>

$ kubectl create secret generic mysecret --from-file=username.txt --from-file=password.txt

$ kubectl describe secrets/<secretname>

Using Secrets through Volumes – Example17: volsecrets.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: myvolsecret
spec:
 containers:
 - name: c1
   image: centos
   command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
   volumeMounts:
    - name: testsecret
      mountPath: "/tmp/mysecrets"   # the secret files will be mounted as ReadOnly by default here
 volumes:
 - name: testsecret
   secret:
     secretName: <secretname>   # this should match the secret name created in the first step
```

Namespace

- You can name your object, but if many are using the cluster then it would be difficult for managing.
- A namespace is a group of related elements that each have a unique name or identifier. Namespace is used to uniquely identify one or more names from other similar names of different objects, groups or the namespace in general.
- Kubernetes namespaces help different projects, teams, or customers to share a Kubernetes cluster & provides:
  - A scope for every Names.
  - A mechanism to attach authorization and policy to a subsection of the cluster.
- By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster
- We can use resource quota on specifying how many resources each namespace can use.
- Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.
- Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all.

Create new namespaces

- Lets assume we have shared Kubernetes cluster for dev & QA use cases.
- The dev team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this no restrictions are put on who can or cannot modify resources to enable agile development.
- For QA team we can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments.
- Partition the Kubernetes cluster into two namespaces: dev & QA

$ kubectl get namespaces

Namespace – Example16: nsdev.yml & nsqa.yml

```
---nsdev.yml---
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
---nsqa.yml---
apiVersion: v1
kind: Namespace
metadata:
  name: testing
  labels:
    name: testing
```

$ kubectl apply -f nsdev.yml
$ kubectl apply -f nsqa.yml
$ kubectl get namespaces (or) $ kubectl get namespaces --show-labels

- Create a Sample pod:

```
kind: Pod
apiVersion: v1
metadata:
 name: testpod
spec:
 containers:
   - name: c00
     image: ubuntu
     command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
```

- Lauch a pod in the new namespace temporarily:

$ kubectl apply --namespace=<namespace> -f <yml>

- List the pods specific to namespace

$ kubectl get pods # this will not list any pods
$ kubectl get pods --namespace=<namespace>
$ kubectl delete namespace <namespace>

- Save the namespace for all subsequent kubectl commands in that context:

$ kubectl config set-context $(kubectl config current-context) --namespace=<insert-namespace-name-here>

- Create a new pod & validate it

$ kubectl apply –f <yml>
$ kubectl config view | grep namespace:

- To see which Kubernetes resources are and aren't in a namespace:

$ kubectl api-resources --namespaced=true # In a namespace
$ kubectl api-resources --namespaced=false # Not in a namespace

Managing Compute Resources for Containers

- A pod in Kubernetes will run with no limits on CPU and memory
- You can optionally specify how much CPU and memory (RAM) each Container needs.
- Scheduler decides about which nodes to place Pods, only if the Node has enough CPU resources available to satisfy the Pod CPU request
- CPU is specified in units of **cores**, and memory is specified in units of **bytes**.
- Two types of constraints can be set for each resource type: **requests** and **limits**
  - ✓A **request** is the amount of that resources that the system will guarantee for the container, and Kubernetes will use this value to **decide on which node to place the po**
  - ✓ A **limit** is the maximum amount of resources that Kubernetes will allow the container use. In the case that request is not set for a container, it defaults to limit. If limit is not set, then if defaults to 0 (unbounded).
- CPU values are specified in "millicpu" and memory in MiB

Pod Resources – Example5: podresources.yml

```
apiVersion: v1
kind: Pod
metadata:
 name: resources
spec:
 containers:
 - name: resource
   image: centos
   command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
   resources:                          # Describes the type of resources to be used
    requests:
      memory: "<value in mebibytes>" # A mebibyte is 1,048,576 bytes, ex: 64Mi
      cpu: "<value in millicores >"       # CPU core split into 1000 units (milli = 1000), ex: 100m
    limits:
      memory: "<value in mebibytes>" # ex: 128Mi
      cpu: "<value in millicores >"  # ex: 200m
```

$ kubectl apply -f podresources.yml

$ kubectl describe pods resources

- To see the pods that uses the most cpu and memory

$ kubectl top pod --all-namespaces

- To see the total cpu, memory usage of all pods

$ kubectl get nodes --no-headers | awk '{print $1}' | xargs -I {} sh -c 'echo {}; kubectl describe node {} | grep Allocated -A 5 | grep -ve Event -ve Allocated -ve percent -ve -- ; echo'

Resource Quotas

- A Kubernetes cluster can be divided into namespaces. If a Container is created in a namespace that has a default CPU limit, and the Container does not specify its own CPU limit, then the Container is assigned the default CPU limit.
- Namespaces can be assigned ResourceQuota objects, this will limit the amount of usage allowed to the objects in that namespace. You can limit:
  - ❑Compute
  - ❑Storage
  - ❑Memory
- Here are two of the restrictions that a resource quota imposes on a namespace:
  - ✓Every Container that runs in the namespace must have its own CPU limit.
  - ✓The total amount of CPU used by all Containers in the namespace must not exceed a specified limit.

Ex: Lets examine a deployment which has 3 replicas where each containers uses 200m, however set the namespace to use 400m allocated

-- cpulimit.yaml--

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: myquota
spec:
  hard:
    limits.cpu: "400m"
    limits.memory: "400Mi"
    requests.cpu: "200m"
    requests.memory: "200Mi"
```

$ kubectl apply -f cpulimit.yml --namespace=testing

ResourceQuota - Example21: quotadeploy.ym

```
kind: Deployment
apiVersion: apps/v1
metadata:
 name: deployments
spec:
 replicas: 3
 selector:
  matchLabels:
   objtype: deployment
 template:
  metadata:
   name: testpod8
   labels:
    objtype: deployment
  spec:
   containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 5 ; done"]
      resources:
        requests:
          cpu: "200m"
```

$ kubectl apply -f quotadeploy.yml --namespace=testing

$ kubectl get pods --namespace=testing

$ kubectl describe deployments deployments --namespace=testing

$ kubectl delete -f cpulimit.yml --namespace=testing

Configure Default CPU Requests and Limits for a Namespace

- **Create a LimitRange and a Pod: cpudefault.yml**

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 1
    defaultRequest:
      cpu: 0.5
    type: Container
```

$ kubectl apply -f cpudefault.yml --namespace=testing

- **Create a pod - cpupod.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo
spec:
  containers:
  - name: default-cpu-demo-ctr
    image: nginx
```

$ kubectl apply -f cpupod.yml --namespace=testing

$ kubectl get pod default-cpu-demo --output=yaml --namespace=testing

- The output shows that the Pod's Container has a CPU request of 500 millicpus and a CPU limit of 1 cpu. These are the default values specified by the LimitRange

What if you specify a Container's limit, but not its request?

- **Create a Pod in which the Container specifies a CPU limit, but not a request: cpupod2.yml**

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-2
spec:
  containers:
  - name: default-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "1"
```

- If request is not specified & limit is given, then request = limit

$ kubectl apply -f cpupod2.yml --namespace=testing

$ kubectl get pod default-cpu-demo-2 --output=yaml --namespace=testing

- The output shows that the Container's CPU request is set to match its CPU limit.
- Notice that the Container was not assigned the default CPU request value of 0.5 cpu.

What if you specify a Container's request, but not its limit?

- **Create a Pod in which the Container specifies a CPU request, but not a limit: cpupod3.yml**

```
apiVersion: v1
kind: Pod
metadata:
 name: default-cpu-demo-3
spec:
 containers:
 - name: default-cpu-demo-3-ctr
  image: nginx
  resources:
   requests:
    cpu: "0.75"
```

```
$ kubectl create -f cpupod3.yml --namespace=testing
$ kubectl get pod default-cpu-demo-3 --output=yaml --namespace=testing
```

- The output shows that the Container's CPU request is set to the value specified in the Container's configuration file.
- The Container's CPU limit is set to 1 cpu, which is the default CPU limit for the namespace

Configure Min & Max Memory Constraints for a Namespace

- **Create a LimitRange and a Pod: memdefault.yml**

```
apiVersion: v1
kind: LimitRange
metadata:
 name: mem-min-max-demo-lr
spec:
 limits:
 - max:
    memory: 1Gi
  min:
    memory: 500Mi
  type: Container
```

```
$ kubectl apply -f memdefault.yml --namespace=testing
$ kubectl get limitrange mem-min-max-demo-lr --namespace=testing
```

- **Create a pod - mempod.yml**

```
apiVersion: v1
kind: Pod
metadata:
 name: constraints-mem-demo
spec:
 containers:
 - name: constraints-mem-demo-ctr
  image: nginx
  resources:
   limits:
    memory: "800Mi"
   requests:
    memory: "600Mi"
```

```
$ kubectl apply -f mempod.yml --namespace=testing
$ kubectl get pod constraints-mem-demo --output=yaml --namespace=testing
```

The Output shows that the container has a memory request of 600MiB and a memory limit of 800MiB. These satisfy the constraints imposed by LimitRange.

Attempt to create a Pod that exceeds the maximum memory constraint

- **Create a Pod in which the Container specifies a memory request of 800 MiB and a memory limit of 1.5 GiB: mempod2.yml**

```
apiVersion: v1
kind: Pod
metadata:
 name: constraints-mem-demo-2
spec:
 containers:
 - name: constraints-mem-demo-2-ctr
  image: nginx
  resources:
   limits:
    memory: "1.5Gi"
   requests:
    memory: "800Mi"
```

```
$ kubectl apply -f mempod2.yml --namespace=testing
$ kubectl get pod constraints-mem-demo-2  --output=yaml --namespace=testing
```

- **The output shows that the Pod does not get created, because the Container specifies a memory limit that is too large**

Attempt to create a Pod that does not meet the minimum memory request

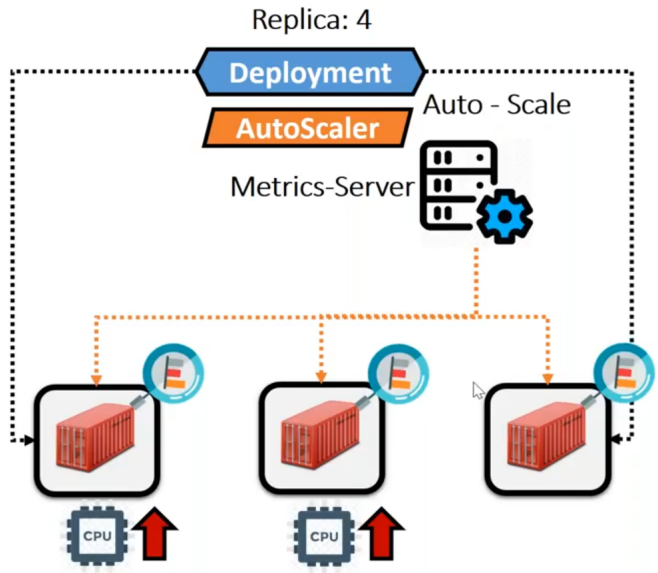
- **Create a Pod in which the Container specifies a memory request of 100 MiB and a memory limit of 800 MiB: mempod3.yml**

```
apiVersion: v1
kind: Pod
metadata:
 name: constraints-mem-demo-3
spec:
 containers:
 - name: constraints-mem-demo-3-ctr
  image: nginx
  resources:
   limits:
    memory: "800Mi"
   requests:
    memory: "100Mi"
```

Create a Pod that does not specify any memory request or limit

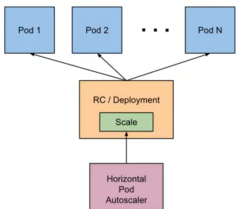
- **Create a Pod in which the Container does not specify a memory request, and it does not specify a memory limit: mempod4.yml**

```
apiVersion: v1
kind: Pod
metadata:
 name: constraints-mem-demo-4
spec:
 containers:
 - name: constraints-mem-demo-4-ctr
  image: nginx
```



Autoscalling (Horizontal Pod Autoscaler)

- K8s has the possibility to automatically scale pods based on observed CPU utilization (or with custom metrics), which is Horizontal Pod Autoscaling
- Scaling can be done only for scalable objects like controller, deployment or replica set.
- HPA is implemented as a Kubernetes API resource and a controller.
- The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.



How does the Horizontal Pod Autoscaler work?

- The HPA is implemented as a control loop, with a period controlled by the controller manager's --**horizontal-pod-autoscaler-sync-period** flag (default value of **30 seconds**).
- During each period, the controller manager **queries** the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition.
- For **per-pod** resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each pod targeted by the HorizontalPodAutoscaler.
- Then, if a target utilization value is set, the controller calculates the utilization value as a **percentage** of the equivalent resource request on the containers in each pod.
- If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted pods, and produces a ratio used to scale the number of desired replicas
- Cooldown period to wait before another downscale operation can be performed is controlled by --**horizontal-pod-autoscaler-downtime-stabilization** flag (default value of **5 min**).

metrics-server needs to be deployed in the cluster to provide metrics via the resource metrics API

- **Download Metrics-Server configuration file (https://github.com/kubernetes-sigs/metrics-server)**

$ wget -O metricserver.yml https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.3.6/components.yaml

- **Add the following lines as an arg to the Deployment object**

```
args:
  - --cert-dir=/tmp
  - --secure-port=4443
  - --kubelet-insecure-tls  (this line mainly)
```

```
$ kubectl apply -f metricserver.yml

$ kubectl get pods -n kube-system
 - check if metric-server pod is running ex: metrics-server-64b57fd654-rfbqc

$ kubectl logs <metric-server-pod> -n kube-system
  - Generated self-signed cert (/tmp/apiserver.crt, /tmp/apiserver.key)
    Serving securely on [::]:4443

$ kubectl top nodes or kubectl top pods

$ kubectl autoscale deployment mydeploy --cpu-percent=20 --min=1 --max=10
```

- To test Increase the load, by running this inside any pod

```
$ apt update
$ while true; do wget -q -O- http://localhost; done
```

```
---------------
kind: Deployment
apiVersion: apps/v1
metadata:
   name: mydeploy
spec:
   replicas: 1
   selector:
    matchLabels:
     name: deployment
   template:
     metadata:
       name: testpod8
       labels:
         name: deployment
     spec:
      containers:
        - name: c00
          image: httpd
          ports:
          - containerPort: 80
          resources:
            limits:
              cpu: 500m
            requests:
```

```
      containers:
        - name: c00
          image: httpd
          ports:
          - containerPort: 80
          resources:
            limits:
              cpu: 500m
            requests:
              cpu: 200m
```

```
root@ip-172-31-44-51:/home/ubuntu# kubectl config set-context $(kubectl config current-context) --n
amespace=default
Context "minikube" modified.
root@ip-172-31-44-51:/home/ubuntu#
```

HPA Object

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
 name: myhpamem
spec:
 maxReplicas: 5
 minReplicas: 1
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: mydeploy
 metrics:
 - type: Resource
  resource:
   name: memory
   targetAverageUtilization: 30
```

```
$ kubectl apply -f deployhpamem.yml
$ kubectl apply -f hpamem.yml

$ kubectl top nodes or kubectl top pods

$ kubectl get all
```

- To test Increase the memory, by running this inside any pod

```
$ apt update
$ apt install -y stress
$  stress --vm 1 --vm-bytes 100M
```

Setup Master on AWS EC2 – Ubuntu (2 cpu):

- Install Docker CE

```
$ apt update
$ apt install -y docker.io
```

- Add the repo for Kubernetes

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat << EOF > /etc/apt/sources.list.d/kubernetes.list
  deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

- Install Kubernetes components

```
$ apt update
$ apt install -y kubelet kubeadm kubectl
```

- Initialize the cluster using the IP range for Flannel.

```
$ kubeadm init --pod-network-cidr=10.244.0.0/16
```

Copy the kubeadmin join command that is in the output. We will need this later.

- Add port obtained from above cmd in the Inbound rules

- Exit sudo and copy the admin.conf to your home directory and take ownership as normal user

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

- You should now deploy a pod network to the cluster

```
$ kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

Setup node on AWS EC2 – Ubuntu: (t2.micro)

- Install Docker CE

```
$ apt update
$ apt install -y docker.io
```

- Add the repo for Kubernetes

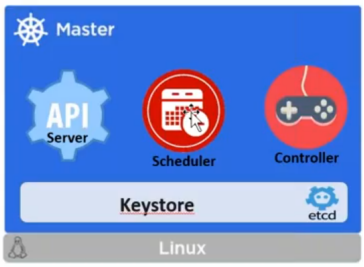
```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat << EOF > /etc/apt/sources.list.d/kubernetes.list
  deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

- Install Kubernetes components

```
$ apt update
$ apt install -y kubeadm
```

- **Join the cluster by running the output cmd obtained from 'kubadm init' on master as root:**

Kubernetes Master



- Master is the brain of the cluster
- Master orchestrates the cluster
- We will access the master to initiate changes and not the nodes directly
- Master consists of these components:
  - ✓ kube-apiserver
  - ✓ kube-scheduler
  - ✓ kube-controller-manager
  - ✓ etcd
- All the master components are set up on a single host
- Multiple masters are configured for High-Availability
- We can communicate to the master node via the CLI, the GUI (Dashboard), or via APIs

kube-apiserver

- Frontend to the cluster
- Our communication to cluster happens only through apiserver
- Maintains RESTful web services for querying and defining our desired cluster and workload state by consuming the YAML content

etcd

- Acts as the memory for the Brian(master)
- It stores the configuration information i.e Cluster state
- It is a high availability key value store that can be distributed among multiple nodes
- It is accessible only by Kubernetes API server

kube-controller-manager

- Watches over the state of the cluster and their job is to steer the cluster into the desired state at any time
- If actual state is different, scheduler will be invoked to ensure the correct number of containers are running
- Responsible for noticing and responding when nodes go down
- Responsible for maintaining the correct number of Instances specified

kube-scheduler

- Scheduler works with the API server to schedule workloads on the nodes.
- It uses different node for matching the desired state and spreads containers across the cluster

Kubernetes Nodes(Minions)

- Nodes are the machines in which containers run
- Node consists of these components:
  - ✓ kubelet
  - ✓ kube-proxy
  - ✓ Container engine
- These components, receive workloads to execute, and update the cluster on their status

- To run and manage a container's lifecycle, we need a container runtime on the worker node. Some examples of container runtimes are:
  - containerd
  - rkt
  - lxd.

kubelet

- The main kubernetes agent
- Interacts with the API server on the Master
- Registers the Node with the cluster
- Updates the workloads that have been invoked by the scheduler:
  - Mount Volumes
  - Run containers
  - Report the status of the node
  - Run container liveness probe

Kube-proxy

- Instead of connecting directly to Containers to access the applications, we use a logical construct called a Service as a connection endpoint. A Service groups related Pods and, when accessed, load balances to them
- Proxy which runs on each worker node and listens to the API server, sets up the routes so that it can reach to it

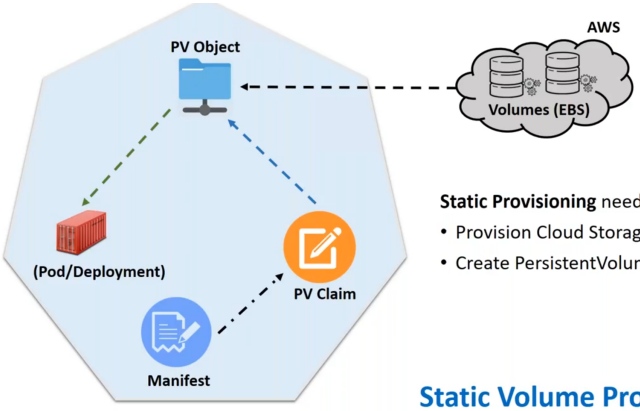
- DaemonSet objects are used when we need a copy of the Pod on all the nodes in the cluster.
- Example: When we have to run some daemon process on each node like **logstash**, **Datadog**, **Prometheus**, **ceph** etc
- As **New nodes** are added to the cluster, DaemonSet creates a **new Pod** on them
- As nodes are **removed** from the cluster, those Pods are **garbage collected**
- We can create Pods on Only **Some Nodes** using NodeSelector
- If a Pod is deleted, then Daemonset will **recreate** the Pod on the same Node
- Daemonsets were introduced to replace **Static Pods** (created and managed by kubelet daemon on a specific node without API server)

```yaml
apiVersion: apps/v1
kind: DaemonSet       # Type of Object
metadata:
  name: demodaemonset
  namespace: default
  labels:
    env: demo
spec:
  selector:
    matchLabels:
      env: demo
  template:
    metadata:
      labels:
        env: demo
    spec:
      containers:
      - name: demoset
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 8 ; done"]
```

Run Only on Some Nodes

```yaml
apiVersion: apps/v1
kind: DaemonSet       metadata:
  name: demodaemonset
  namespace: default
  labels:
    env: demo
spec:
  selector:
    matchLabels:
      env: demo
  template:
    metadata:
      labels:
        env: demo
    spec:
      containers:
      - name: demoset
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-Adam; sleep 8 ; done"]
      nodeSelector:                    # Run pods only on matching nodes with label
        server: group1
```

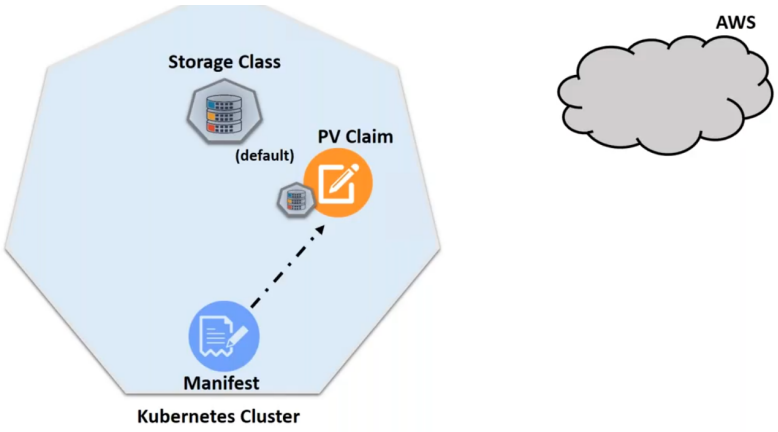
```
$ kubectl label nodes <NodeName> server=group1
```



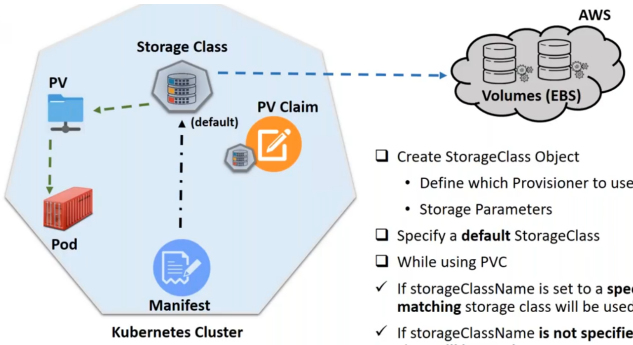
Static Provisioning need:

- Provision Cloud Storage Volumes
- Create PersistentVolume objects

Static Volume Provisioning



Dynamic Volume Provisioning



- ❑ Create StorageClass Object
  - Define which Provisioner to use
  - Storage Parameters
- ❑ Specify a **default** StorageClass
- ❑ While using PVC
- ✓ If storageClassName is set to a **specific** value, then the **matching** storage class will be used
- ✓ If storageClassName **is not specified**, then **default** storage class **will be used**

Dynamic Volume Provisioning

Cloud Provider	Default StorageClass Name	Default Provisioner
Amazon Web Services	gp2	aws-ebs
Microsoft Azure	standard	azure-disk
Google Cloud Platform	standard	gce-pd
OpenStack	standard	cinder
VMware vSphere	thin	vsphere-volume

Following are the steps what we need for AWS:

- ❑ The hostname of each node must match EC2's private DNS entry for that node
- ❑ An IAM role and policy that EC2 instances can assume as an instance profile
- ❑ Kubernetes-specific tags applied to the AWS resources used by the cluster
- ❑ Particular command-line flags added to the Kubernetes API server, Kubernetes controller manager, and the Kubelet

Storage.yml

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: demosc
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2                    # gp2/io1
  zones: ap-south-1a
  iopsPerGB: "100"
  fsType: ext4
volumeBindingMode: Immediate
```

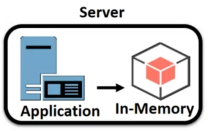
pvc.yml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
  labels:
    app: nginx
spec:
  storageClassName: demosc
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

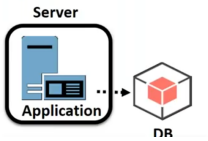
Pod.yml

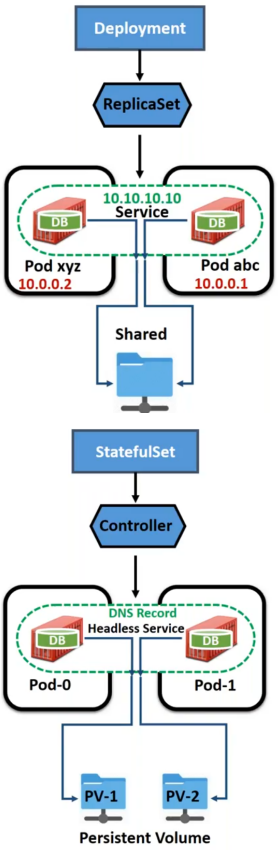
```yaml
kind: Pod
apiVersion: v1
metadata:
  name: custompod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: dynavol
  volumes:
    - name: dynavol
      persistentVolumeClaim:
        claimName: mypvc
```

- **Stateful** applications store data about the state from client requests on the server itself and use that state to process further requests.
- Examples of Stateful applications include **MongoDB**, **Cassandra**, and **MySQL**.



- **Stateless** app is an application program that does not save client data generated in one session for use in the next session with that client.
- Examples of stateless applications include major websites (**Amazon**, **Google**, **Facebook**, **Twitter**), **WhatsApp**, **AWS(S3**, EBS)





Stateful Applications need:

- Unique Identity for each Pod
- Every Pod need its own persistent volume
- Pods need to be created in Order(Master/Client)
- Pods need graceful shutdown

StatefulSet Deployments provide:

- ✓ Stable, unique network identifiers.
- ✓ Stable, persistent storage.
- ✓ Ordered, graceful deployment and scaling.
- ✓ Ordered, automated rolling updates

- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods.
  - ❑ Since we don't need any load balancing or routing
  - ❑ You only need the service to patch the request to the backend pod. Hence, the name: headless: a service that does have an IP

```
# Creating Statefulset
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: webapp
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

```
# Headless Service
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
```

- Create Statefulset Object

```
$ kubectl apply -f sts.yml
$ kubectl get service nginx
$ kubectl get statefulset webapp
$ kubectl get pvc
$ kubectl get pv
$ kubectl describe svc nginx
$ kubectl describe statefulset webapp
$ kubectl describe pvc <pvc-name>
```

- Examining the Pod's Ordinal Index

```
$ kubectl run -it --image busybox:1.28 dns-test --restart=Never --rm nslookup webapp-0.nginx
$ kubectl describe statefulset/webapp
```

- Accessing one Pod from another using Headless service (Unique identifier)

(Service name will always point to the first one which is the Master replica)

```
$ kubectl exec -it webapp-0 -- curl nginx                  # service-name
$ kubectl exec -it webapp-0 -- curl webapp-1.nginx         # [pod/host-name].service-name
```

```
kubectl get storageclass
```

- Writing to Stable Storage

```
$ kubectl exec webapp-0 -- sh -c 'echo TEST > /usr/share/nginx/html/index.html'
$ kubectl exec -it webapp-0 -- curl nginx                  # service-name
$ kubectl exec -it webapp-0 -- curl webapp-1.nginx         # [pod/host-name].service-name
```

- Scaling up/down a StatefulSet

```
$ kubectl scale sts webapp --replicas=5    # Notice the Pods will be created in order
$ kubectl scale sts webapp --replicas=2    # Notice the Pods will be deleted in order
```

- Replication of Pod with Persistent Volume

```
$ kubectl delete pod webapp-0
$ kubectl exec -it webapp-0 -- curl nginx     # Notice hostname & index.html will be same
```

- Delete Statefulset, PVC & PV separately

```
$ kubectl delete -f sts.yml
$ kubectl delete pvc –l app=nginx
```

We have ReplicaSets, DaemonSets, StatefulSets, and Deployments they all share one common property: they ensure that their pods are always running. If a pod fails, the controller restarts it or reschedules it to another node to make sure the application the pods is hosting keeps running

What if we want the pod to terminate?

Scenarios:

- ❑ **Log rotation**
- ❑ **Take Backup of a DB**
- ❑ **Helm Charts use Jobs to run install, setup, or test commands on clusters**
- ❑ **Running batch processes**
- ❑ **Run a task at an scheduled interval**

```
apiVersion: batch/v1
kind: Job
metadata:
  name: testjob
spec:
  template:
    metadata:
      name: testjob
    spec:
      containers:
      - name: counter
        image: centos:7
        command: ["bin/bash", "-c", "echo Hi-Adam; sleep 5"]
      restartPolicy: Never
```

```
$ kubectl apply –f job.yml
```

❑ A job doesn't get deleted by itself, we have to deleteit

```
$ kubectl delete –f job.yml
```

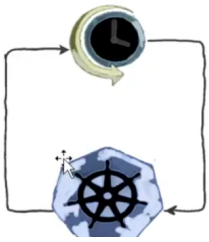
```
apiVersion: batch/v1
kind: Job
metadata:
  name: testjob
spec:
  parallelism: 5                # Runs for pods in parallel
  activeDeadlineSeconds: 10  # Timesout after 30 sec
  template:
    metadata:
      name: testjob
    spec:
      containers:
      - name: counter
        image: centos:7
        command: ["bin/bash", "-c", "echo Hi-Adam; sleep 20"]
      restartPolicy: Never
```

```
$ kubectl apply –f job.yml
```

❑ A job doesn't get deleted by itself, we have to deleteit

```
$ kubectl delete –f job.yml
```

The Cron Job Pattern



- If we have multiple nodes hosting the application for high availability, which node handles cron?
- What happens if multiple identical cron jobs run simultaneously?

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: adam
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - image: ubuntu
            name: adam
            command: ["/bin/bash", "-c", "echo Hi-Adam; sleep 5"]
          restartPolicy: Never
```

```
$ kubectl apply –f cronjob.yml
```

❑ A job doesn't get deleted by itself, we have to deleteit

```
$ kubectl delete –f cronjob.yml
```

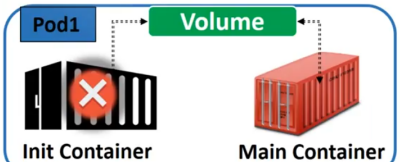
Scenario 01: Seeding a Database

Scenario 02: Delaying The Application Launch Until The Dependencies Are Ready

Scenario 03: Clone a Git repository into a Volume

Scenario 04: Generate configuration file dynamically

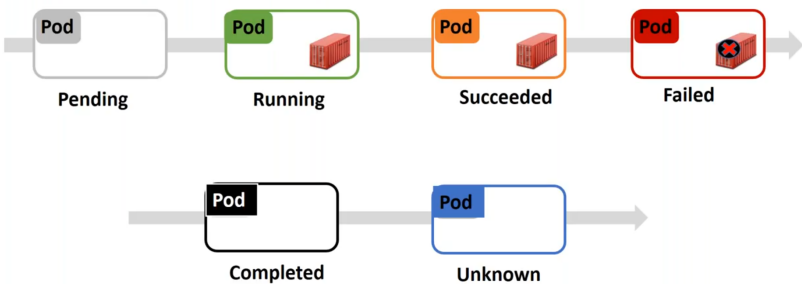
- Init containers are specialized containers that run before app containers in a Pod .
- Init containers always run to completion
- If a Pod's init container fails, Kubernetes repeatedly restarts the Pod until the init container succeeds
- Init containers do not support readiness probes




```
apiVersion: v1
kind: Pod
metadata:
  name: myvolemptydir
spec:
  initContainers:
  - name: c1
    image: centos
    command: ["/bin/sh", "-c", "echo STAYHOME-STAYSAFE > /tmp/xchange/testfile; sleep 30"]
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/xchange"
  containers:
  - name: c2
    image: centos
    command: ["/bin/bash", "-c", "while true; do echo `cat /tmp/data/testfile`; sleep 5; done"]
    volumeMounts:
      - name: xchange
        mountPath: "/tmp/data"
  volumes:
  - name: xchange
    emptyDir: {}
```

Pod Phases

- The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle.
- Here are the possible values for phase:



Pending:

- The Pod has been accepted by the Kubernetes system, but its not **running**
- one or more of the Container images is still **downloading**
- If the pod cannot be scheduled because of resource **constraints**

Running:

- The Pod has been **bound to a node**
- All of the Containers have been **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.
- The Container either exited with non-zero status or was terminated by the system

Unknown:

- State of the Pod could not be **obtained**
- Typically due to an **Error** in **Network** or **communicating** with the host of the Pod

Completed:

- The pod has run to completion as there's nothing to keep it running eg. Completed Jobs

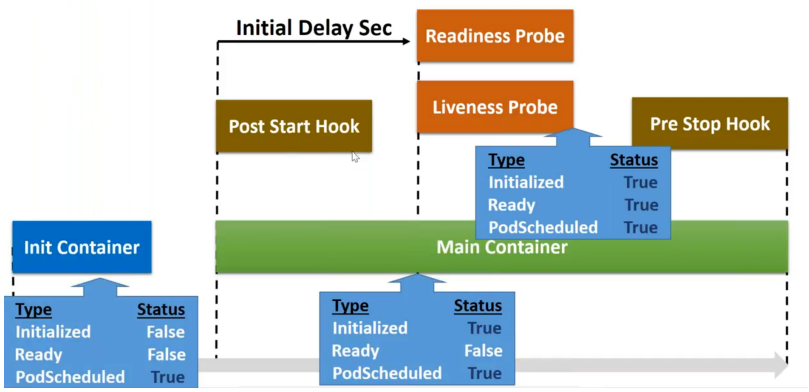
Pod Conditions

- A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed.
- Using **'kubectl describe pod PODNAME'** you can get the condition of a Pod
- These are the possible types:
  - **PodScheduled**: the Pod has been scheduled to a node
  - **Ready**: the Pod is able to serve requests and will be added to the load balancing pools of all matching Services
  - **Initialized**: all init containers have started successfully
  - **Unschedulable**: the scheduler cannot schedule the Pod right now, for example due to lacking of resources or other constraints
  - **ContainersReady**: all containers in the Pod are ready

Container States

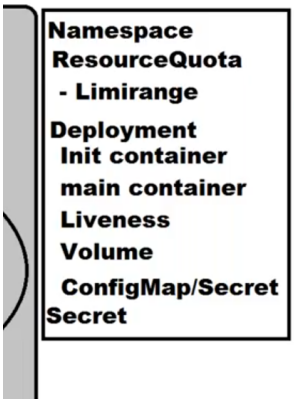
- Once Pod is assigned to a node by scheduler, kubelet starts creating containers using container runtime
- Using **'kubectl get pod PODNAME –o yaml'** state is displayed for each container within that Pod
- There are three possible states of containers:
  - **Waiting**: A container in Waiting state still indicates that its pulling images, applying Secrets, etc
  - **Running**: Indicates that the container is executing without issues. Once a container enters into Running
  - **Terminated**: Indicates that the container completed its execution and has stopped running.A container enters into this when it has successfully completed execution or when it has failed for some reason

Pod Lifecycle



Executing commands when the container starts/stops with the postStart/preStop hook

```
apiVersion: v1
kind: Pod
metadata:
 name: client
spec:
 containers:
  - image: nginx
    name: client
    lifecycle:
     postStart:
       exec:
        command: [ "sh", "-c", "sleep 10"]
     preStop:
       httpGet:
        port: 8080
        path: /shutdown
```

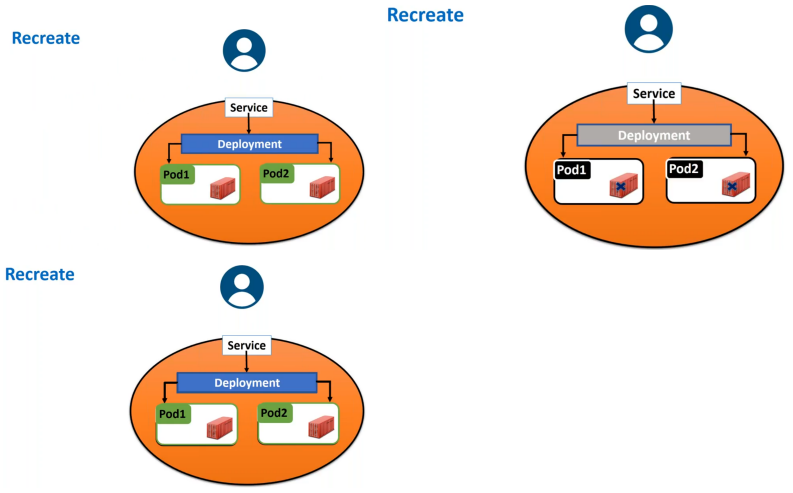


Software Deployments

- Software deployment includes all the process required for preparing a software application to run and operate in a specific environment.
- It involves installation, configuration, testing and making changes to optimize the performance of the software.

Types of deployments ?

- ✓ **Recreate**: Version A is terminated then version B is rolled out.
- ✓ **Ramped** (also known as rolling-update or incremental): Version B is slowly rolled out and replacing version A.
- ✓ **Blue/Green**: Version B is released alongside version A, then the traffic is switched to version B.
- ✓ **Canary**: Version B is released to a subset of users, then proceed to a full rollout.



Recreate

Pro:

- Application state entirely renewed

Cons:

- Downtime that depends on both shutdown and boot duration of the application

✓ Suitable for development environment

Build1.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-1.10
spec:
 replicas: 2
 selector:
  matchLabels:
   name: nginx
   version: "1.10"
 template:
  metadata:
   labels:
    name: nginx
    version: "1.10"
  spec:
   containers:
    - name: nginx
      image: nginx:1.10
      ports:
       - name: http
         containerPort: 80
```

```yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 labels:
  name: nginx
spec:
 ports:
  - name: http
    port: 80
    targetPort: 80
 selector:
  name: nginx
  version: "1.10"
```

- Apply the Build Manifest

```
$ kubectl apply -f build1.yml
```

- Make a note of the Service VIP

```
$ kubectl describe svc nginx | grep IP
```

- Check the version of Nginx running

```
$ curl -s http://<IP_OBTAINED_ABOVE>/version|grep nginx
```

- To see the pods getting created, run this in one terminal

```
$ watch kubectl get pods
```

Build2.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-1.10
spec:
 replicas: 2
 strategy:
  type: Recreate
 selector:
  matchLabels:
   name: nginx
   version: "1.10"
 template:
  metadata:
   labels:
    name: nginx
    version: "1.10"
  spec:
   containers:
    - name: nginx
      image: nginx:1.11
      ports:
       - name: http
         containerPort: 80
```

```yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 labels:
  name: nginx
spec:
 ports:
  - name: http
    port: 80
    targetPort: 80
 selector:
  name: nginx
  version: "1.10"
```

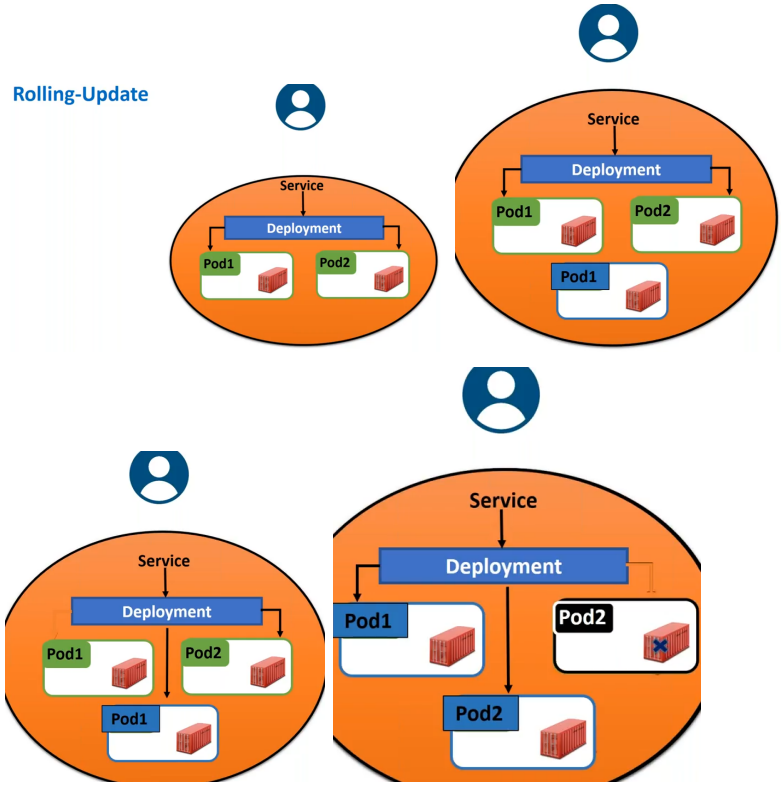
- Apply the new Build

```
$ kubectl apply -f build2.yml
```

- Check the version of Nginx running

```
$ curl -s http://<IP_OBTAINED_ABOVE>/version|grep nginx
```

Rolling-Update



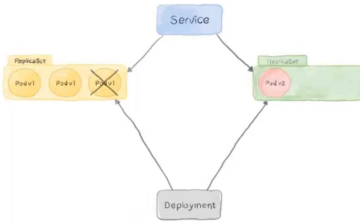
Rolling-Update

Pro:

- Version is slowly released across instances
- Convenient for stateful applications that can handle rebalancing of the data

Cons:

- Rollout/rollback can take time
- Supporting multiple APIs is hard
- No control over traffic



✓ Suitable for QA, Stage or UAT environment

✓ For stateful applications suitable on Production environment

- Apply the Build Manifest

```
$ kubectl apply -f build1.yml
```

- Make a note of the Service VIP

```
$ kubectl describe svc nginx | grep IP
```

- Check the version of Nginx running

```
$ curl -s http://<IP_OBTAINED_ABOVE>/version|grep nginx
```

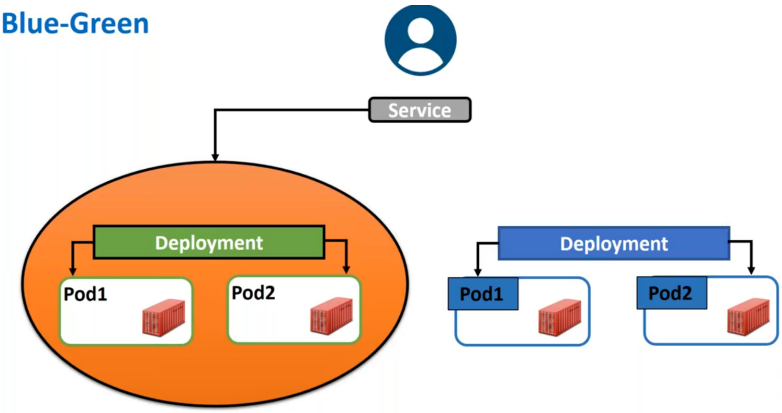
- To see the pods getting created, run this in one terminal

```
$ watch kubectl get pods
```

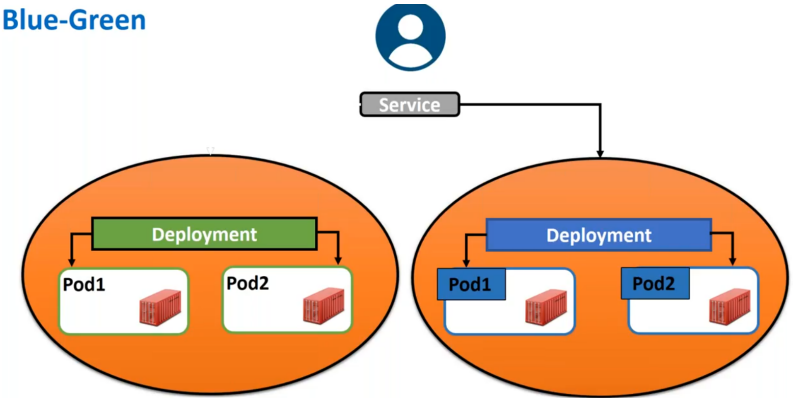
- Remove the strategy type from Spec & Apply the New Build Manifest

```
$ kubectl apply -f build2.yml
```

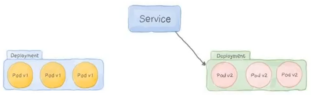
Blue-Green



Blue-Green



Blue-Green



Pro:

- Instant rollout/rollback
- Avoid versioning issue, change the entire cluster state in one go

Cons:

- Requires double the resources
- Proper test of the entire platform should be done before releasing to production
- Handling stateful applications can be hard

✓ Suitable for Production environment

Build1.yml (Blue)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 2
  selector:
    matchLabels:
      name: nginx
      version: "1.10"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.10"
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - name: http
              containerPort: 80
```

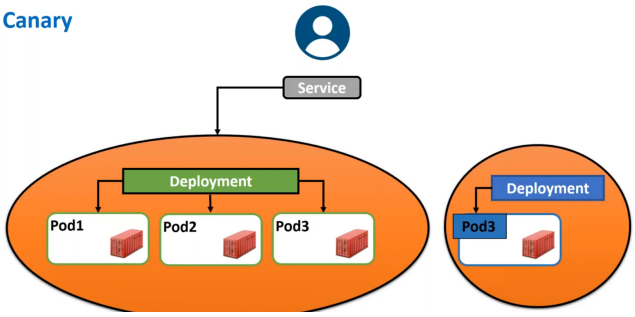
```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    name: nginx
    version: "1.10"
```

Build2.yml (Green)

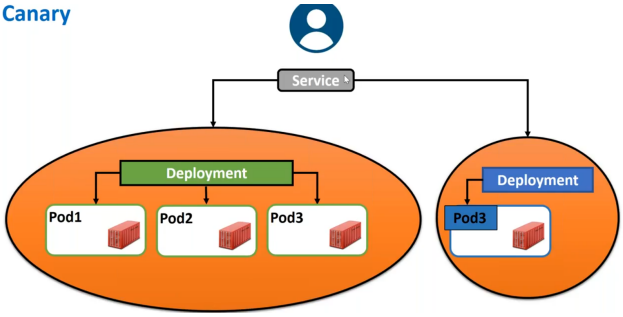
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.11
spec:
  replicas: 2
  selector:
    matchLabels:
      name: nginx
      version: "1.11"
  template:
    metadata:
      labels:
        name: nginx
        version: "1.11"
    spec:
      containers:
        - name: nginx
          image: nginx:1.11
          ports:
            - name: http
              containerPort: 80
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    name: nginx
    version: "1.11"
```

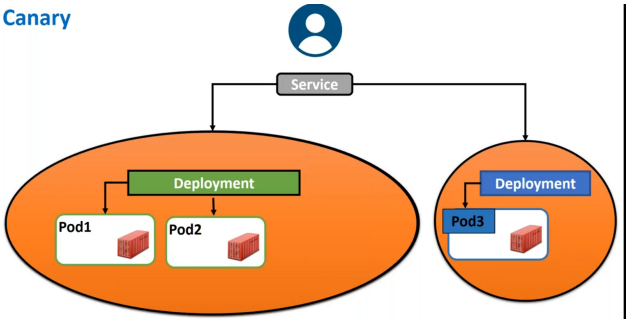
Canary



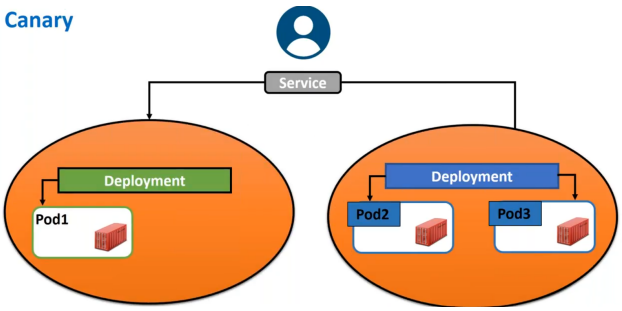
Canary



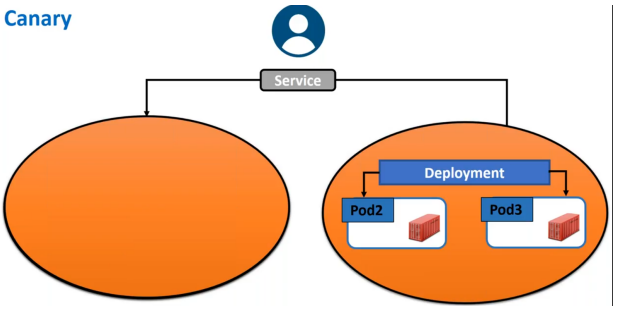
Canary



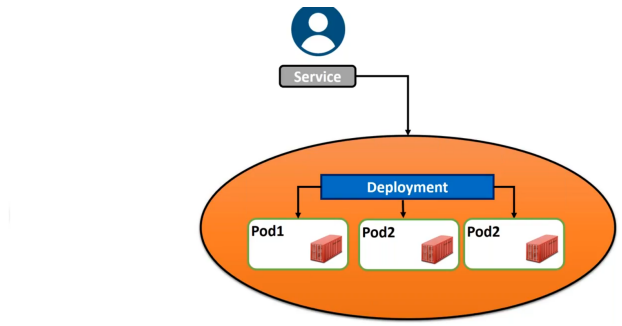
Canary



Canary



Canary



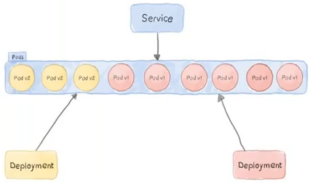
Canary

Pro:

- Version released for a subset of users
- Convenient for error rate and performance monitoring
- Fast rollback

Cons:

- Slow Rollout
- Fine tuned traffic distribution can be expensive



✓ Suitable for Production environment

Build1.yml (Blue)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.10
spec:
  replicas: 3
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.10
          ports:
            - name: http
              containerPort: 80
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    name: nginx
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    name: nginx
```

- Apply the Blue Build Manifest

```
$ kubectl apply -f build1.yml
```

- Make a note of the Service VIP

```
$ kubectl describe svc nginx | grep IP
```

- Check the version of Nginx running

```
$ curl -s http://<IP_OBTAINED_ABOVE>/version|grep nginx
```

Build2.yml (Green)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-1.11
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.11
          ports:
            - name: http
              containerPort: 80
```

- When starting with K8s, we tend to use full administrator credentials …
- We need different capabilities to be associated with different types of roles within a Kubernetes cluster

Developer | Operator | Administrator

- That is where the notion of **RBAC** or Role-Based Access Control comes into play
- By default, RBAC is enabled in Kubernetes from version 1.8
- We use **rbac.authorization.k8s.io** API group for creating authorization policies

RBAC concepts

The RBAC model in Kubernetes is based on three elements:

- **Resources**: The set of Kubernetes API Objects available in the cluster. Examples like Pods, Deployments, Services, Nodes, and PersistentVolumes, among others
- **Subjects**: users (human or machine users) or groups of users
- **Verbs**: The set of operations that can be executed to the resources above. Different verbs are available (examples: get, watch, create, delete, etc.), but ultimately all of them are Create, Read, Update or Delete

Understanding RBAC API Objects: Roles & RoleBinding

- Roles will connect API Resources and Verbs specific to one **namespace**
- Roles can be reused for different subjects by **Binding** to specific **entity-subjects**
- If we want the role to be applied cluster-wide, the equivalent object is called **ClusterRoles** and use **ClusterRoleBindings** for binding to cluster-level subjects