**Kubernetes – Deployments**

Deployments are upgraded and higher version of replication controller. They manage the deployment of replica sets which is also an upgraded version of the replication controller. They have the capability to update the replica set and are also capable of rolling back to the previous version.

They provide many updated features of matchLabels and selectors. We have got a new controller in the Kubernetes master called the deployment controller

**Changing the Deployment**

**Updating**

The user can update the ongoing deployment before it is completed. In this, the existing deployment will be settled and new deployment will be created.

**Deleting**

The user can pause/cancel the deployment by deleting it before it is completed. Recreating the same deployment will resume it.

**Rollback**

We can roll back the deployment or the deployment in progress. The user can create or update the deployment by using **DeploymentSpec.PodTemplateSpec = oldRC.PodTemplateSpec.**

**Deployment Strategies**

Deployment strategies help in defining how the new RC should replace the existing RC.

**Recreate**

− This feature will kill all the existing RC and then bring up the new ones. This results in quick deployment however it will result in downtime when the old pods are down and the new pods have not come up.

**Rolling Update**

− This feature gradually brings down the old RC and brings up the new one. This results in slow deployment, however there is no deployment. At all times, few old pods and few new pods are available in this process.

**Create Deployment**

$ kubectl create –f Deployment.yaml -–record

deployment "Deployment" created Successfully.

**Fetch the Deployment**

$ kubectl get deployments

NAME DESIRED CURRENT UP-TO-DATE AVILABLE AGE

Deployment 3 3 3 3 20s

**Check the Status of Deployment**

$ kubectl rollout status deployment/Deployment

**Updating the Deployment**

$ kubectl set image deployment/Deployment tomcat=tomcat:6.0

**Rolling Back to Previous Deployment**

$ kubectl rollout undo deployment/Deployment –to-revision=2

**Kubernetes – Volumes**

In Kubernetes, a volume can be thought of as a directory which is accessible to the containers in a pod. We have different types of volumes in Kubernetes and the type defines how the volume is created and its content.

The concept of volume was present with the Docker, however the only issue was that the volume was very much limited to a particular pod. As soon as the life of a pod ended, the volume was also lost.

On the other hand, the volumes that are created through Kubernetes is not limited to any container. It supports any or all the containers deployed inside the pod of Kubernetes. A key advantage of Kubernetes volume is, it supports different kind of storage wherein the pod can use multiple of them at the same time.

**Types of Kubernetes Volume**

Here is a list of some popular Kubernetes Volumes −

**emptyDir**

− It is a type of volume that is created when a Pod is first assigned to a Node. It remains active as long as the Pod is running on that node. The volume is initially empty and the containers in the pod can read and write the files in the emptyDir volume. Once the Pod is removed from the node, the data in the emptyDir is erased.

**hostPath**

− This type of volume mounts a file or directory from the host node’s filesystem into your pod.

**gcePersistentDisk**

− This type of volume mounts a Google Compute Engine (GCE) Persistent Disk into your Pod. The data in a gcePersistentDisk remains intact when the Pod is removed from the node.

**awsElasticBlockStore**

− This type of volume mounts an Amazon Web Services (AWS) Elastic Block Store into your Pod. Just like gcePersistentDisk, the data in an awsElasticBlockStore remains intact when the Pod is removed from the node.

**Nfs**

− An nfs volume allows an existing NFS (Network File System) to be mounted into your pod. The data in an nfs volume is not erased when the Pod is removed from the node. The volume is only unmounted.

**Iscsi**

− An iscsi volume allows an existing iSCSI (SCSI over IP) volume to be mounted into your pod.

**flocker**

− It is an open-source clustered container data volume manager. It is used for managing data volumes. A flocker volume allows a Flocker dataset to be mounted into a pod. If the dataset does not exist in Flocker, then you first need to create it by using the Flocker API.

**glusterfs**

− Glusterfs is an open-source networked filesystem. A glusterfs volume allows a glusterfs volume to be mounted into your pod.

**rbd**

− RBD stands for Rados Block Device. An rbd volume allows a Rados Block Device volume to be mounted into your pod. Data remains preserved after the Pod is removed from the node.

**Cephfs**

− A cephfs volume allows an existing CephFS volume to be mounted into your pod. Data remains intact after the Pod is removed from the node.

**gitRepo**

− A gitRepo volume mounts an empty directory and clones a git repository into it for your pod to use.

**Secret**

− A secret volume is used to pass sensitive information, such as passwords, to pods.

**persistentVolumeClaim**

− A persistentVolumeClaim volume is used to mount a PersistentVolume into a pod. PersistentVolumes are a way for users to “claim” durable storage (such as a GCE PersistentDisk or an iSCSI volume) without knowing the details of the particular cloud environment.

**downwardAPI**

− A downwardAPI volume is used to make downward API data available to applications. It mounts a directory and writes the requested data in plain text files.

**azureDiskVolume**

− An AzureDiskVolume is used to mount a Microsoft Azure Data Disk into a Pod.

**Persistent Volume and Persistent Volume Claim**

**Persistent Volume (PV)**

− It’s a piece of network storage that has been provisioned by the administrator. It’s a resource in the cluster which is independent of any individual pod that uses the PV.

**Persistent Volume Claim (PVC)**

− The storage requested by Kubernetes for its pods is known as PVC. The user does not need to know the underlying provisioning. The claims must be created in the same namespace where the pod is created.

**Creating Persistent Volume**

kind: PersistentVolume ---------> 1

apiVersion: v1

metadata:

name: pv0001 ------------------> 2

labels:

type: local

spec:

capacity: -----------------------> 3

storage: 10Gi ----------------------> 4

accessModes:

- ReadWriteOnce -------------------> 5

hostPath:

path: "/tmp/data01" --------------------------> 6

In the above code, we have defined −

**kind: PersistentVolume** → We have defined the kind as PersistentVolume which tells kubernetes that the yaml file being used is to create the Persistent Volume.

**name: pv0001** → Name of PersistentVolume that we are creating.

**capacity:** → This spec will define the capacity of PV that we are trying to create.

**storage: 10Gi** → This tells the underlying infrastructure that we are trying to claim 10Gi space on the defined path.

**ReadWriteOnce** → This tells the access rights of the volume that we are creating.

**path: "/tmp/data01"** → This definition tells the machine that we are trying to create volume under this path on the underlying infrastructure.

**Creating PV**

$ kubectl create –f local-01.yaml

persistentvolume "pv0001" created

**Getting Details About PVC**

$ kubectl get pvc

NAME STATUS VOLUME CAPACITY ACCESSMODES AGE

myclaim-1 Bound pv0001 10Gi RWO 7s

**Describe PVC**

$ kubectl describe pv pv0001

-------------------------------------------------------------------------------------------------

**Kubernetes – Secrets**

Secrets can be defined as Kubernetes objects used to store sensitive data such as user name and passwords with encryption.

There are multiple ways of creating secrets in Kubernetes.

* Creating from txt files.
* Creating from yaml file

**Creating From Text File**

In order to create secrets from a text file such as user name and password, we first need to store them in a txt file and use the following command.

$ kubectl create secret generic tomcat-passwd –-from-file = ./username.txt –fromfile = ./.

password.txt

**Creating From Yaml File**

apiVersion: v1

kind: Secret

metadata:

name: tomcat-pass

type: Opaque

data:

password: <User Password>

username: <User Name>

**Creating the Secret**

$ kubectl create –f Secret.yaml

secrets/tomcat-pass

**Using Secrets**

Once we have created the secrets, it can be consumed in a pod or the replication controller as −

* Environment Variable
* Volume

**As Environment Variable**

In order to use the secret as environment variable, we will use env under the spec section of pod yaml file.

env:

- name: SECRET\_USERNAME

valueFrom:

secretKeyRef:

name: mysecret

key: tomcat-pass

**As Volume**

spec:

volumes:

- name: "secretstest"

secret:

secretName: tomcat-pass

containers:

- image: tomcat:7.0

name: awebserver

volumeMounts:

- mountPath: "/tmp/mysec"

name: "secretstest"

----------------------------------------------------------------------------------------------------

**Kubernetes - Network Policy**

Network Policy defines how the pods in the same namespace will communicate with each other and the network endpoint. It requires extensions/v1beta1/networkpolicies to be enabled in the runtime configuration in the API server. Its resources use labels to select the pods and define rules to allow traffic to a specific pod in addition to which is defined in the namespace.

First, we need to configure Namespace Isolation Policy. Basically, this kind of networking policies are required on the load balancers.

kind: Namespace

apiVersion: v1

metadata:

annotations:

net.beta.kubernetes.io/network-policy: |

{

"ingress":

{

"isolation": "DefaultDeny"

}

}

$ kubectl annotate ns <namespace> "net.beta.kubernetes.io/network-policy =

{\"ingress\": {\"isolation\": \"DefaultDeny\"}}"

Once the namespace is created, we need to create the Network Policy.

**Network Policy Yaml**

kind: NetworkPolicy

apiVersion: extensions/v1beta1

metadata:

name: allow-frontend

namespace: myns

spec:

podSelector:

matchLabels:

role: backend

ingress:

- from:

- podSelector:

matchLabels:

role: frontend

ports:

- protocol: TCP

port: 6379