**Kubernetes**

**Menions**: This is an individual node used in Kubernetes

* Combination of these minions is called as Kubernetes cluster
* Master is the main machine which triggers the container orchestration
* It distributes the workload to the Slaves
* Slaves are the nodes that accept the workload from the master

and handle activities load balancing, autoscaling, high availability etc

Kubernetes uses various of types of Object:

**1 Pod:** This is a layer of abstraction on top of a container. This is the smallest

object that Kubernetes can work on. In the Pod we have a container.

The advantage of using a Pod is that kubectl commands will work on the Pod and the

Pod communicates these instructions to the container. In this way we can use the

same kubectl irrespective of which technology containers are in the Pod.

**2 Service:** This is used for port mapping and network load balancing

**3 NameSpace:** This is used for creating partitions in the cluster. Pods running

in a namespace cannot communicate with other pods running in another namespace

**4 Secrets:** This is used for passing encrypted data to the Pods

**5 ReplicationController:** This is used for managing multiple replicas of PODs

and also performing scaling

**6 ReplicaSet**: This is similar to replicationcontroller but it is more advanced

where features like selector can be implemented

**7 Deployment:** This used for performing all activities that a Replicaset can do

it can also handle rolling update

**8 PersistantVolume:** Used to specify the section of storage that should be used for volumes

**9 PersistantVolumeClaims:** Used to reserve a certain amount of storage for a pod from the persistent volume.

**10 Statefulsets**: These are used to handle stateful application like data bases

where consistency in read write operations has to be maintained.

**11 HorrizontalPodAutScaller**: Used for auto scaling of pods depending on the load

**Kubernetes Architecture**

**Master Components:**

**Container runtime**: This can be docker or any other container technology

**apiServer:** Users interact with the apiServer using some client like ui, command line tool like kubelet. It is the apiServer which is the gateway to the cluster

It works as a gatekeeper for authentication and it validates if a specific

user has permissions to execute a specific command. For Example if we want to

deploy a pod or a deployment first apiServers validates if the user is authorized to perform that action and if so it passes to the next process

ie the "Scheduler"

**Scheduler**: This process accepts the instructions from apiServer after validation

and starts an application on a specific node or set of nodes. It estimates

how much amount of h/w is required for an application and then checks which

slaves have the necessary h/w resources and instructs the kubelet to deploy

the application

**kubelet**: This is the actual process that takes the orders from scheduler and

deploy an application on a slave. This kubelet is present on both master and slave

**controller manager:** This check if the desired state of the cluster is always

maintained.If a pod dies it recreates that pod to maintain the desired state

**etcd**: Here the cluster state is maintained in key value pairs.

* It maintains info about the slaves and the h/w resources available on

the slaves and the pods running on the slaves

* The scheduler and the control manager read the info from this etcd

and schedule the pods and maintain the desired state

**Worker components**

**container run time:** Docker or some other container technology

**kubelet**: This process interacts with container run time and the node

and it start a pod with a container in it

**kubeproxy:** This will take the request from services to pod

* It has the intelligence to forward a request to

a nearby pod. Eg If an application pod wants to communicate with a db pod

then kubeproxy will take that request to the nearby pod

**Day 20**

1 To see the list of nodes in the Kubernetes cluster

* kubectl get nodes

2 To get info about the nodes along with ipaddress and docker version etc

* kubectl get nodes -o wide

3 To get detailed info about the nodes

* kubectl describe nodes node\_name

Create nginx as a pod and name it webserver

* kubectl run --image nginx webserver

To see the list of pods

* kubectl get pods

To get info about the pods along with ipaddress

* kubectl get pods -o wide

To get detailed info about the pods

kubeclt describe pods webserver

=========================================================================

**Create a mysql pod and also pass the necessary environment variables**

kubectl run --image mysql:5 db --env MYSQL\_ROOT\_PASSWORD=intelliqit

Check if the pod is running

kubectl get pods

To delete the mysql pod

kubectl delete pods db

=========================================================================

**Kubernetes Definition file**

Kubernetes performs container orchestration uisng certain definition

file. These files are created using yml and they have 4 top level

fields

1. apiVersion:
2. kind:
3. metadata:
4. spec:

**apiVersion**: Every Kubernetes object uses a specific Kubernetes code

library that is called apiVersion. Only once this code library is imported

we can start working on specific objects

**kind**: This represents the type of Kubernetes object that we want to us

eg: Pod,Replicaset,Service etc

**metadata**: Here we give a name to the Kubernetes object and also some

labels. These labels can be used later for performing group activities

**spec**: This is where we store info about the exact docker image, container name

environment varibales, port mapping etc

**Kind** **apiVersion**

1. Pod v1
2. Service v1
3. NameSpace v1
4. Secrets v1
5. ReplicationController v1
6. PersistantVolume v1
7. PersistantVolumeClaim v1
8. ReplicaSet apps/v1
9. Deployment apps/v1
10. StatefulSet apps/v1
11. DaemonSet apps/v1

=========================================================================

**UseCase-1**

Create a pod definition file to start an nginx in a pod

name the pod as nginx-pod,name the container as webserver

**vim pod-defintion1.yml**

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

author: intellqit

type: reverse-proxy

spec:

containers:

- name: appserver

image: nginx

To create a pod from the above file

* kubectl create -f pod-defintion1.yml

To see the list of pods

* kubectl get pods

To see the pods along with the ipaddress and name of the slave where it is running

* kubectl get pods -o wide

To delete the pods created from the above file

* kubectl delete -f pod-definition1.yml

=========================================================================

**Create a pod definition file to start a Postgres container**

Name of the container should be mydb, pass the necessary environment

variables, this container should run in a pod called Postgres-pod

and give the labels as author=Bharath and type=database

**vim pod-definition2.yml**

apiVersion: v1

kind: Pod

metadata:

name: postgres-pod

labels:

author: intelliqit

type: database

spec:

containers:

- name: mydb

image: postgres

env:

- name: POSTGRES\_PASSWORD

value: myintelliqit

- name: POSTGRES\_USER

value: myuser

- name: POSTGRES\_DB

value: mydb

To create pods from the above defintion file

* kubectl create -f pod-defintion2.yml

To delete the pods

* kubectl delete -f pod-definition2.yml

**========================================================================**

**Use Case 3**

Create a pod defintion file to start a jenkins container in a pod

called jenkins-pod,also perform port mapping to access the jenkins

from a browser

**vim pod-definition3.yml**

apiVersion: v1

kind: Pod

metadata:

name: jenkins-pod

labels:

author: intelliqit

type: ci-cd

spec:

containers:

- name: myjenkins

image: Jenkins/Jenkins:lts

ports:

- containerPort: 8080

hostPort: 8080

...

To create pods from the above file

* kubectl create -f pod-defintion3.yml

To see the list of pods along with nodes where they are running

* kubectl get nodes -o wide

To get the external ip of the node

* kubectl get node -o wide

To access then jenkins from browser

* external\_ip\_of\_slavenode:8080

**Day 21**

======================================================================

**ReplicationController**

This is a high level Kubernetes object that can be used for handling

multiple replicas of a Pod. Here we can perform Load Balancing

and Scaling

ReplicationController uses keys like "replicas,template" etc in the "spec" section

In the template section we can give metadata related to the pod and also use

another spec section where we can give containers information

**Create a replication controller for creating 3 replicas of httpd**

**vim repilication-controller.yml**

---

apiVersion: v1

kind: ReplicationController

metadata:

name: httpd-rc

labels:

author: intelliqit

spec:

replicas: 3

template:

metadata:

name: httpd-pod

labels:

author: intelliqit

spec:

containers:

- name: myhttpd

image: httpd

ports:

- containerPort: 80

hostPort: 8080

...

To create the httpd replicas from the above file

* kubectl create -f replication-controller.yml

To check if 3 pods are running an on whcih slaves they are running

* kubectl get pods -o wide

To delete the replicas

* kubectl delete -f replication-controller.yml

=========================================================================

**ReplicaSet**

This is also similar to ReplicationController but it is more advanced and it can also handle load balancing and scalling

It has an additional field in spec section called as "selector"

This selector uses a child element "matchLabels" where the

it will search for Pod based on a specific label name and try to add

them to the cluster

**Create a replicaset file to start 4 tomcat replicas and then perform scalling**

**vim replica-set.yml**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: tomcat-rs

labels:

type: webserver

author: intelliqit

spec:

replicas: 4

selector:

matchLabels:

type: webserver

template:

metadata:

name: tomcat-pod

labels:

type: webserver

spec:

containers:

- name: mywebserver

image: tomee

ports:

- containerPort: 8080

hostPort: 9090

To create the pods from the above file

kubectl create -f replica-set.yml

Scalling can be done in 2 ways

a) Update the file and later scale it

b) Scale from the coomand prompt withbout updating the defintion file

a) Update the file and later scale it

Open the replicas-set.yml file and increase the replicas count from 4 to 6

kubectl replace -f replicas-set.yml

Check if 6 pods of tomcat are running

kubectl get pods

b) Scale from the coomand prompt withbout updating the defintion file

kubectl scale --replicas=2 -f replica-set.yml

================================================================

Deployment

================

This is also a high level Kubernetes object which can be used for

scalling and load balancing and it can also perfrom rolling update

Create a deployment file to run nginx with 3 replicas

vim deployment1.yml

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

author: intelliqit

type: proxyserver

spec:

replicas: 3

selector:

matchLabels:

type: proxyserver

template:

metadata:

name: nginx-pod

labels:

type: proxyserver

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

hostPort: 8888

To create the deployment from the above file

kubectl create -f deployment.yml

To check if the deployment is running

kubectl get deployment

To see if all 3 pod of nginx are running

kubectl get pod

Check the version of nginx

kubectl describe pods nginx-deployment | less

==================================================================================

Create a mysql deployment

vim deployment2.yml

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: mysql-deployment

labels:

type: db

author: intelliqit

spec:

replicas: 3

selector:

matchLabels:

type: db

template:

metadata:

name: mysql-pod

labels:

type: db

spec:

containers:

- name: mydb

image: mysql

ports:

- containerPort: 3306

hostPort: 8080

env:

- name: MYSQL\_ROOT\_PASSWORD

value: intelliqit

=========================================================================

**Service Object**

This is used for network load balancing and port mapping

It uses 3 ports

1. Target port: Pod or container port
2. port: Service port
3. hostPort: Host machines port to make it accessible from external network

Service objects are classified into 3 types

1. **clusterIP**: This is the default type of service object used in

Kubernetes and it is used when we want the Pods in the cluster to

communicate with each other and not with external networks

1. **nodePort**: This is used if we want to access the pods from an external

network and it also performs network load balancing ie even if a pod

is running on a specific salve we can access it from other slave in

the cluster

1. **Load Balancer**: This is similar to Nodeport and it is used for external

connectivity of a Pod and also network load balancing and it also assigns

a public ip for all the slave combined together

**Use Case**

=================

Create a service definition file for port mapping an nginx pod

**vim pod-defintion1.yml**

---

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

author: intellqit

type: reverse-proxy

spec:

containers:

- name: appserver

image: nginx

=========================================================

**vim service1.yml**

---

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

type:

ports:

- targetPort: 80

port: 80

nodePort: 30008

selector:

author: intellqit

type: reverse-proxy

Create pods from the above pod definition file

* kubectl create -f pod-definition1.yml

Create the service from the above service definition file

* kubectl create -f service.yml

Now nginx can be accesed from any of the slave

kubectl get nodes -o wide

Take the external ip of any of the nodes:30008

=========================================================================

Create a service object of the type LoadBalancer for a tomcat pods

**vim servcie2.yml**

---

apiVersion: v1

kind: Service

metadata:

name: tomcat-service

spec:

type: LoadBalancer

ports:

- targetPort: 80

port: 80

selector:

author: intellqit

type: appserver

**vim pod0defintion5.yml**

**vim pod-definition2.yml**

---

apiVersion: v1

kind: Pod

metadata:

name: tomcat-pod

labels:

type: appserver

author: intelliqit

spec:

containers:

- name: tomcat

image: tome

=========================================================================

**ClusterIp**: used in cases of databases, where the resources should not communicate with the outside world.

-cluster ip is the default service object of k8s, even though if we don't give type = cluster Ip, automatically it created clusterip

Create a service object of the type load balancer for postgres pod

**vim service3.yml**

apiVersion: v1

kind: Service

metadata:

name: postgres-service

spec:

type: ClusterIp

ports:

- targetPort: 5432

port: 5432

selector:

author: intellqit

type: db

**vim pod-defintion6.yml**

apiVersion: v1

kind: Pod

metadata:

name: mysql-pod

labels:

type: db

author: intelliqit

spec:

containers:

- name: mydb

image: mysql

env:

name: MYSQL\_ROOT\_PASSWORD

value: intelliqit

=========================================================================

Node affinity: This is a feature of Kubernetes attracts pods to a

specific slave

To see the list of a labels

* kubectl get nodes --show-labels

To label a slave

* kubectl label nodes <your-node-name> key=value
* kubectl label nodes gke-cluster-1-default-pool-3cde7c4a-hl74 slave1=intelliqit1

=====================================================================

**Pod Defintion file to implement node affinity**

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: slave1

operator: In

values:

- intelliqit1

containers:

- name: nginx

image: nginx

=================================================================

**Deployment file to implement node affintiy**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

type: proxy

spec:

replicas: 2

selector:

matchLabels:

type: proxy

template:

metadata:

name: nginx-pod

labels:

type: proxy

spec:

containers:

- name: mynginx

image: nginx

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: slave1

operator: In

values:

- intelliqit1

================================================================

**Taints and toleration**

**Node affinity,** is a property of Pods that attracts them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

To create a taint for a node

* kubectl taint nodes node1 node=intelliqit:NoSchedule

To delete the tain

* kubectl taint nodes node1 node=intelliqit:NoSchedule-

**Deployment definition file to use the above taint**

apiVersion: apps/v1

kind: Deployment

metadata:

name: httpd-deployment

labels:

type: webserver

spec:

replicas: 3

selector:

matchLabels:

type: webserver

template:

metadata:

name: httpd-pod

labels:

type: webserver

spec:

containers:

- name: myhtppd

image: httpd

tolerations:

- key: slave3

operator: Equal

value: intelliqit3

effect: NoSchedule

=========================================================================

**DaemonSets**: These are used to run a single pod on each and every slave, The no salve count will become the desired count of the Daemon sets

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: ghost-daemon

labels:

type: cms

spec:

selector:

matchLabels:

type: cms

template:

metadata:

name: ghost-pod

labels:

type: cms

spec:

containers:

- name: ghost

image: ghost

...

=========================================================================

**Secrets**

============

This is used to send encrypted data to the definition files

Generally passwords for Databases can be encrypted using this

**Create a secret file to store the MySQL password**

vim secret.yml

---

apiVersion: v1

kind: Secret

metadata:

name: mysql-pass

type: Opaque

stringData:

password: intelliqit

username: sai

...

To deploy the secret

* kubectl create -f secret.yml

**Create a pod definition file to start a mysql pod and pass the environment**

varible using the above secret

vim pod-defitintion5.yml

---

apiVersion: v1

kind: Pod

metadata:

name: mysql-pod

labels:

author: intelliqit

type: db

spec:

containers:

- name: mydb

image: mysql:5

env:

- name: MYSQL\_ROOT\_PASSWORD

valueFrom:

secretKeyRef:

name: mysql-pass

key: password

...

To create pods from above file

* kubect create -f pod-defintion5.yml

==================================================================

**Create a secret definition file for postgres secret**

apiVersion: v1

kind: Secret

metadata:

name: postgres-secret

type: Opaque

stringData:

password: intelliqit

username: myuser

dbname: mydb

Create postgres deployment and use the above secret

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: postgres-deployment

labels:

app: db

spec:

replicas: 2

selector:

matchLabels:

app: db

template:

metadata:

name: postgres-pod

labels:

app: db

spec:

containers:

- name: mydb

image: postgres

env:

- name: POSTGRES\_PASSWORD

valueFrom:

secretKeyRef:

name: postgres-secret

key: password

- name: POSTGRES\_USER

valueFrom:

secretKeyRef:

name: postgres-secret

key: username

- name: POSTGRES\_DB

valueFrom:

secretKeyRef:

name: postgres-secret

key: dbname

=========================================================================

Kubernetes Project Code can be downlaoded from github

https://github.com/krishnain/my-kubernetes-project.git

=========================================================================

**Helm Chart** is a very feature-rich framework when you are working with complex Kubernetes cluster and deployment. Helm chart provides a very convenient way to pass values.yaml and use it inside your Helm Chart

Create your first Helm Chart

We are going to create our first helloworld Helm Chart using the following command

helm create mynginx

tree mynginx

Update the service.type from ClusterIP to NodePort inside the values.yml

To install the chart

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

helm install <FIRST\_ARGUMENT\_RELEASE\_NAME> <SECOND\_ARGUMENT\_CHART\_NAME>

helm install nginx mynginx

Verify the helm install command

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

helm list -a

Get kubernetes Service details and port

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

kubectl get service

=========================================

How to ADD upstream Helm chart repository

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

helm repo add <REPOSITORY\_NAME> <REPOSITORY\_URL>

To add any chart repository you should know the name and repository url.

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

helm repo add bitnami https://charts.bitnami.com/bitnami

Verify the repository

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

helm search repo bitnami

To see the list of repositories added

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

helm repo list

Updating the helm repo

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

Lets see how you can update your helm repositories. (The update command is necessary if haven’t updated your Helm chart repository in a while, so might miss some recent changes)

Here is the command to update Helm repository

helm repo update

Removong a repository

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

helm repo remove bitnami

===========================================

Demo

===============

In this tutorial, we are going to install WordPress with MariaDB using the Helm Chart on Kubernetes cluster. With this installation, we are going to see - How we can upgrade as well as rollback the Helm Chart release of WordPress. This complete setup inherited the benefits of the Kubernetes .i.e. scalability and availability.

Since we are installing WordPress, so we need to have a database running behind the WordPress application. From the database standpoint, we are going to use MariaDB. Helm chart ships all these components in a single package, so that we need not worry about installing each component separately.

To search for all wordpress relates repositories

helm search hub wordpress

If the output of the above command is too large we can use

helm search hub wordpress --max-col-width=0

Ensure that the binami is installed

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

helm repo add bitnami https://charts.bitnami.com/bitnami

heml repo list

Readme.md

=================

This Readme.md contains the installation instructions and it can be viewed using the following command

helm show readme bitnami/wordpress --version 10.0.3

To update the username and password

vim wordpress-values.yml

wordpressUsername: admin

wordpressPassword: admin

wordpressEmail: selenium.saikrishna@gmail.com

wordpressFirstName: Sai

wordpressLastName: Krishna

wordpressBlogName: mywordpress.com

service:

type: LoadBalancer

Create a new namespace

kubectl create namespace nswordpress

Versify the namesapce

kubectl get namespace

Run the below command to install wordpess in the namepsace

helm install wordpress bitnami/wordpress --values=wordpress-values.yaml --namespace nswordpress --version 10.0.3

To see the resources running in a specific namespace

watch -x kubectl get all --namespace nswordpress

To remove

kubect uninstall wordpress