Use Case5:

Kubernetes Project

Secrets

Node affinity

Taints and Tolerations

What is Kubernetes?

Kubernetes, often abbreviated as "K8s", orchestrates containerized applications to run on a cluster of hosts. The K8s system automates the deployment and management of cloud native applications using on-premises infrastructure or public cloud platforms. It distributes application workloads across a Kubernetes cluster and automates dynamic container networking needs. Kubernetes also allocates storage and persistent volumes to running containers, provides automatic scaling, and works continuously to maintain the

desired state of applications, providing resiliency.

Kubernetes Features

Kubernetes has many features that help orchestrate containers across multiple hosts, automate the management of K8s clusters, and maximize resource usage through better utilization of infrastructure. Important features include:

Auto-scaling: Automatically scale containerized applications and their resources up or down based on usage.

Lifecycle management: Automate deployments and updates with the ability to:

- Rollback to previous versions.
- Pause and continue a deployment.

Declarative model: Declare the desired state, and K8s works in the background to maintain that state and recover from any failures.

Resilience and self-healing: Auto placement, auto restart, auto replication and auto scaling provide application self-healing.

Persistent storage: Ability to mount and add storage dynamically.

Load balancing: Kubernetes supports a variety of internal and external load balancing options to address diverse needs.

DevSecOps support: DevSecOps is an advanced approach to security that simplifies and automates container operations across clouds, integrates security throughout the container lifecycle, and enables teams to deliver secure, high-quality software more quickly. Combining DevSecOps practices and Kubernetes improves developer productivity.

Kubernetes Architecture and How it Works?

Containers encapsulate an application in a form that's portable and easy to deploy. The Kubernetes architecture is designed to run containerized applications. A Kubernetes cluster consists of at least one control plane and at least one worker node (typically a physical or virtual server). The control plane has two main responsibilities. It exposes the Kubernetes API through the API server and manages the nodes that make up the cluster. The control plane makes decisions about cluster management and detects and responds to cluster events.

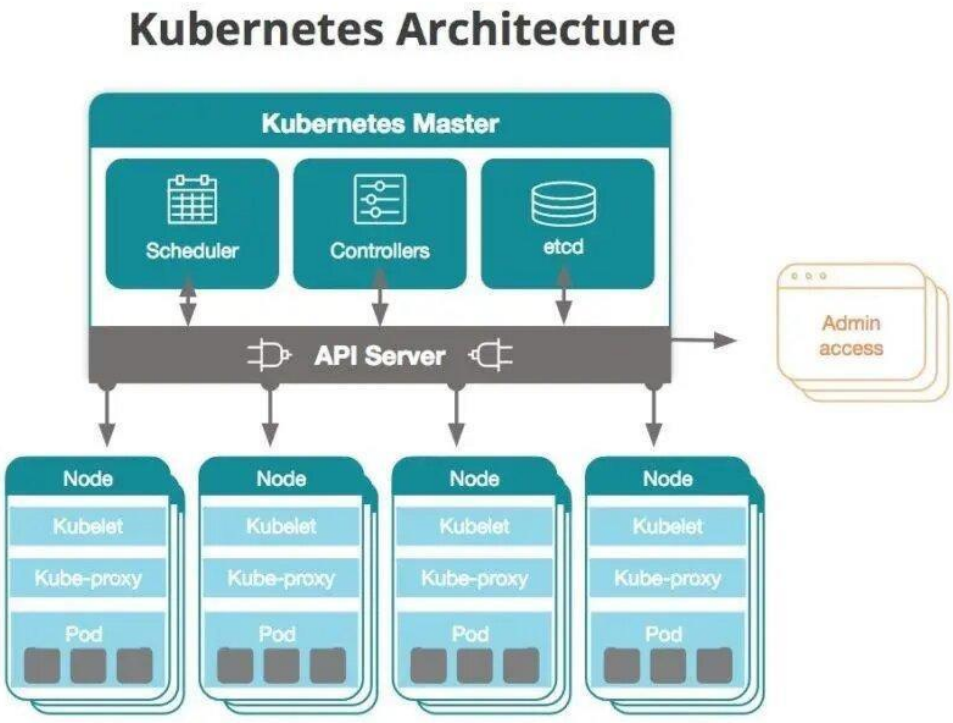
The smallest unit of execution for an application running in Kubernetes is the Kubernetes Pod, which consists of one or more containers. Kubernetes Pods run on worker nodes.

A K8s node has three major components:

Kubelet: An agent that makes sure that the necessary containers are running in a Kubernetes Pod.

Kube-proxy: A network proxy that runs on each node in a cluster to maintain network rules and allow communication.

Container runtime: The software responsible for running containers. Kubernetes supports any runtime that adheres to the Kubernetes CRI (Container Runtime Interface).



Kubernetes Architecture:-

On 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 is having permissions to execute a specific command. 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 slave 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 checks 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 also 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.

On Worker components: container run time: Docker or some other container technology.

kubelet: This process interacts with container run time and the node and it starts a pod with a container in it.

kube-proxy: 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 kube-proxy will take that request to the nearby pod.

Kubernetes Advantages

The Kubernetes platform has become popular because it provides a number of important advantages:

Portability: Containers are portable across a range of environments from virtual environments to bare metal. Kubernetes is supported in all major public clouds, as a result, you can run containerized applications on K8s across many different environments.

Simplified CI/CD: CI/CD is a DevOps practice that automates building, testing and deploying applications to production environments. Enterprises are integrating Kubernetes and CI/CD to create scalable CI/CD pipelines that adapt dynamically to load.

Cost efficiency: Kubernetes' inherent resource optimization, automated scaling, and flexibility to run workloads where they provide the most value means your IT spend is in your control.

Scalability: Cloud native applications scale horizontally. Kubernetes uses "auto-scaling," spinning up additional container instances and scaling out automatically in response to demand.

API-based: The fundamental fabric of Kubernetes is its REST API. Everything in the Kubernetes environment can be controlled through programming.

Integration and extensibility: Kubernetes is extensible to work with the solutions you already rely on, including logging, monitoring, and alerting services. The Kubernetes community is working on a variety of

open source solutions complementary to Kubernetes, creating a rich and fast-growing ecosystem.

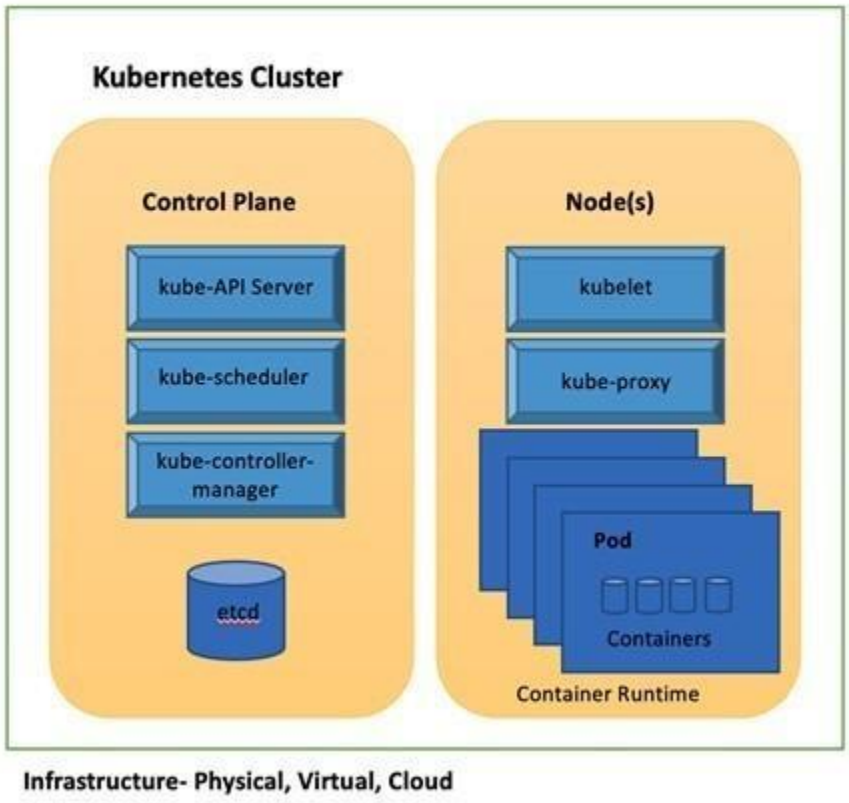
What is Container Orchestration?

Container orchestration automates most of the tasks required to run containerized workloads and services, including the operations that are essential to the Kubernetes container lifecycle: provisioning, deployment, scaling, networking and load balancing.

Kubernetes

Minions: This is an individual node used in Kubernetes; a combination of these minions is called the Kubernetes cluster.

Master is the main machine which triggers the container orchestration. It distributes the work load to the Slaves. Slaves are the nodes that accept the work load from the master and handle activates load balancing, auto-scaling, high availability etc.



Kubernetes uses various of types of Object:

- **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.

- **Service:** This is used for port mapping and network load balancing.
- **Namespace:** This is used for creating partitions in the cluster. Pods running in a namespace cannot communicate with other pods running in another namespace.
- **Secrets:** This is used for passing encrypted data to the Pods.

- **Replication Controller:** This is used for managing multiple replicas of PODs and also performing scaling.
- **Replica Set:** This is similar to a replication controller but it is more advanced where features like selector can be implemented.
- **Deployment:** This is used for performing all activities that a Replica set can do; it can also handle rolling updates.
- **Volume:** Used to preserve the data even when the pods are deleted.
- **Statefulsets:** These are used to handle stateful applications like data bases where consistency in read write operations has to be maintained.

Cloud Based K8's Services

AWS - Elastic Kubernetes Services (EKS)
Google - Google kubernetes Engine(GKE)
Azure - Azure Kubernetes Services(EKS)

Setup of Kubernetes

Free

1 http://katakoda.com

(or)

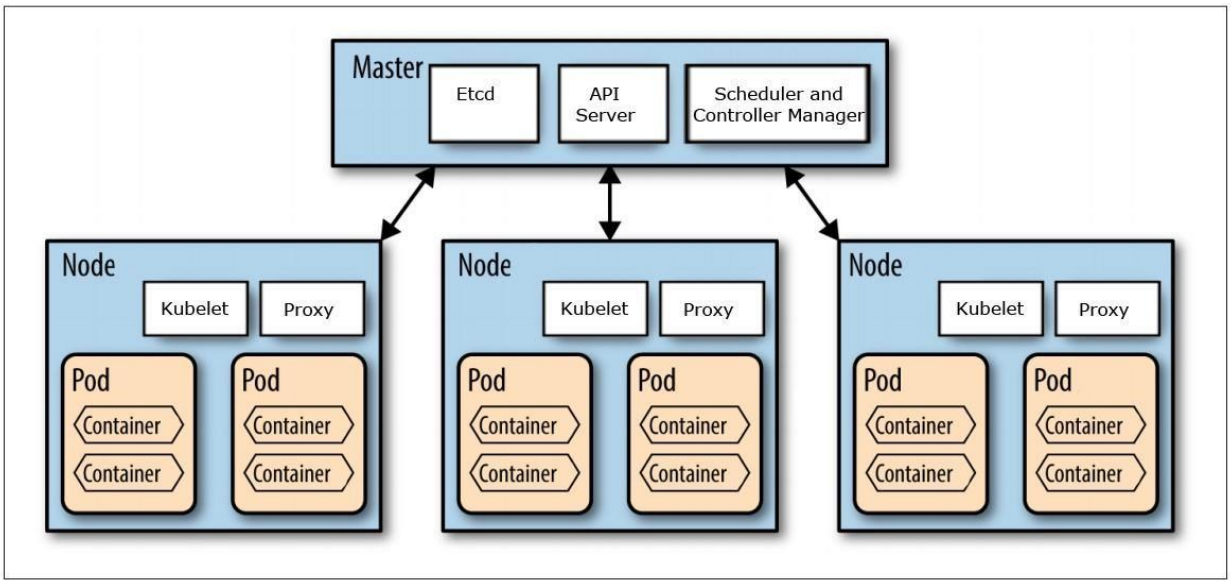
2 http://playwithk8s.com

Paid

1) Signup for a Google cloud account

2) Click on Menu icon on top right corner--->Click on Kubernetes Engine-->Clusters

3)Click on Create cluster--->Click on Create



Setup of Kubernetes using Kubeadm

Install, start and enable docker service.

yum install -y -q yum-utils device-mapper-persistent-data lvm2 > /dev/null 2>&1

yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo > /dev/null 2>&1

yum install -y -q docker-ce >/dev/null 2>&1

systemctl start docker systemctl enable docker

Disable SELINUX

setenforce 0

sed -i --follow-symlinks 's/^SELINUX=enforcing/SELINUX=disabled/' /etc/sysconfig/selinux

Disable SWAP

```
sed -i '/swap/d' /etc/fstab

swapoff -a
```

Update sysctl settings for Kubernetes networking.

```
cat >>/etc/sysctl.d/kubernetes.conf<<EOF

net.bridge.bridge-nf-call-ip6tables = 1

net.bridge.bridge-nf-call-iptables = 1

EOF

sysctl --system
```

Add Kubernetes to yum repository cat

```
>>/etc/yum.repos.d/kubernetes.repo<<EOF

[kubernetes] name=Kubernetes

baseurl=https://packages.cloud.google.com/yum/repos/kubernet

es-el7-x86_64 enabled=1 gpgcheck=1 repo_gpgcheck=1

gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.g
pg


https://packages.cloud.google.com/yum/doc/rpm-package-key.g
pg

EOF
```

Install Kubernetes yum install -y kubeadm-1.19.1 kubelet-

1.19.1 kubectl-1.19.1

Enable and start Kubernetes service

systemctl start kubelet

systemctl enable kubelet

Repeat the above steps on Master and slaves

On Master

Initialize the Kubernetes cluster

kubeadm init --apiserver-advertise-address=ip_of_master --pod-network-cidr=192.168.0.0/16

To be able to use the kubectl command to connect and interact with the cluster, the user needs a kube config file.

mkdir /home/centos/.kube cp /etc/kubernetes/admin.conf

/home/centos/.kube/config chown -R centos:centos

/home/centos/.kube

Deploy calico network

kubectl create -f

https://docs.projectcalico.org/v3.9/manifests/calico.yaml

For slaves to join the cluster kubeadm token

create --print-join-command

Check the pods of kube-system are running

kubectl get pods -n kube-system

Use Case1:

Create nginx as a pod and name it webserver.

```
kubectl run --image nginx webserver
```

To see the list of pods running

```
kubectl get pods
```

To see more info about the pods like their ip and slave where they are running.

```
kubectl  get pods -o wide kubectl
describe pods webserver kubectl
describe pods webserver | less
```

To delete the pod `kubectl delete`

```
pods webserver
```

Use Case2:

Create a mysql pod and name it mydb and go into its interactive terminal and create a few tables.

```
kubectl run --image mysql:5 mydb --env
MYSQL_ROOT_PASSWORD=sai
```

To check the pods

```
kubectl get pods
```

To go into the interactive terminal

```
kubectl exec -it mydb -- bash
```

To login into the db

```
mysql -u root -p
Password: sai
```

Create tables here

Kubernetes Definition files

Objects in the Kubernetes cluster are deployed using these definition files. They are created using yml and they generally use these 4 top level fields. They are

1)apiVersion:

2)Kind:

3)Metadata:

4)spec:

- apiVersion : This specifies the code library that has to be imported to create a particular kind of Kubernetes object.
- kind: Here we specify the type Kubernetes object that we want to create (Pod, ReplicaSet, Deployment, Service etc.)
- metadata: Here we can give additional info about the Pod like the name of the Pod, some labels etc.
- spec: This is where exact info about the object that is created is specified like containers, info port mapping, no of replicas etc.

Kind	apiversions
Pod	V1
Service	V1
Secret	V1
NameSpace	V1
Replication Controller	V1

Volume	V1
Replica set	apps/V1
Deployment	apps/V1
Statefuleset	apps/V1

Create a pod definition file to start a nginx pod with a name webserver.

1) vim pod-defintion1.yml

```
--apiVersion: v1
       kind: Pod
       metadata:
        name: nginx-pod
        labels:
         type: proxy
         author: pythonlife
       spec:
        containers:
         - name: webserver
              image: nginx
...
```

2) Create pod from the above file

```
kubectl apply -f pod-defintion1.yml
```

3) To check the list of pods

```
kubectl get pods
```

4) To delete the pods

```
kubectl delete -f pod-defintion1.yml
```

Use Case3:

Create a postgres-pod and give the labels as author=pythonlife and type=db, also pass the necessary environment variables.

```
1 vim pod-definition2.yml
  —
      apiVersion: v1 kind:
      Pod metadata:
      name: postgres-pod
      labels: author:
      pythonlife type: db
      spec:
       containers:
            - name: mydbimage:
              postgres env:
            - name:
              POSTGRES_PASSWORD
              value: pythonlife
            - name:
              POSTGRES_USERvalue:
              myuser - name:
              POSTGRES_DB value:
              mydb
...
```

To create pods from the above file

```
kubectl apply -f pod-defintion2.yml
```

Use Case4:

Create a jenkins-pod and also perform necessary port mapping.

```
vim pod-definition3.yml
--apiVersion: v1 kind:
        Pod metadata:
        name: jenkins-pod
        labels:
          type: ci-cd
          author: pythonlife
        spec:
         containers:
            - name: jenkinsimage:
              jenkins/jenkins ports:
            - containerPort: 8080
              hostPort: 8080
...
```

To create the pods from the above file

```
kubectl apply -f pod-defintion3.yml
```

To check if the jenkins pod is running

```
kubectl get pods -o wide
```

To access jenkins from browser

```
kubectl get nodes -o wide
```

Capture the external ip of the node where jenkins pod is running in browser external ip:8080.

To fix the pod

```
kubectl exec -it jenkins-pod -- bash
```

The above command will take into pod account. use cat and paste url of jenkins pod to login in jenkins.

Namespace

Namespaces are used to create partitions in the Kubernetes cluster. Pods running in different namespaces cannot communicate with each other.

To create Namespaces
vim namespace.yml

```
--apiVersion: v1 kind:
        Namespace
        metadata:
                name: test-ns
...
```

```
kubectl apply -f namespace.yaml
```

To see the list of namespace

```
kubectl get namespace
```

Create a pod in the above name space.
vim pod-defintion5.yml

```
--apiVersion: v1
        kind: Pod
        metadata:
```

```
        name: tomcat-pod
        namespace: test-ns
        labels:
         author: pythonlife
         type: appserver
       spec:
        containers:
         - name: mytomcat
              image: tomee ports:
              - containerPort:
              8080 hostPort: 8080
...
```

kubectl apply -f pod-defintion5.yml

Create a Volume. vim volumes.yml

```
--apiVersion: v1 kind:
      Pod metadata:
      name: redis-pod
      labels: author:
      pythonlife
      spec:
       containers:
        - name: redisimage: redis
          volumeMounts: - name:
          redis-volume mountPath:
          /data/redis
       volumes:
        - name: redis-
          volumeemptyDir: {}
...
```

Create a pod from the above file

kubectl create -f volumes.yml

To check if the volume is mounted

kubectl exec -it redis-pod -- bash

Go to the redis folder and create some files.

```
cd redis
cat > file
Store some data in this file
```

To kill the redis pod install

```
procps
```

```
apt-get update apt-get install -y
procps
```

Identify the process id of redis

```
ps aux
kill 1
```

Check if the redis-pod is recreated

```
kubectl get pods
```

We will see the restart count changes for this pod.
If we go into this pod's interactive terminal.

```
kubectl exec -it redis-pod -- bash
```

We will see the data but not the s/w's (procps) we installed.

```
cd redis
ls
ps (This will not work)
```

KOPS: Kubernetes operations

Kubernetes Operations, or Kops, is an open source project used to set up Kubernetes clusters easily and swiftly. It's considered the "kubectl" way of creating clusters.

Kops allows deployment of highly available Kubernetes clusters on AWS and Google (GCP) clouds. It includes features such as dry-runs and automatic idempotency, terraform config generation making Kops a great option for do-it-yourself (DIY) developers.

Kubernetes on GCP using Kops:

1. Launch Linux instance in GCP (Kubernetes Client)
2. Create and attach IAM role to Instance.

   Kops need permissions to access like,

   S3, EC2, VPC, Route53, Auto scaling etc..

3. Install Kops on EC2

   curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/late st | grep tag_name | cut -d '"' -f 4)/kops-linux-amd64 chmod +x kops-linux-amd64 sudo mv kops-linux-amd64 /usr/local/bin/kops

4. 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

5. Create S3 bucket in AWS

S3 bucket is used by kubernetes to persist cluster state, let's create s3 bucket using aws cli (Note: Make sure you choose bucket name that is unique across all aws accounts.) aws s3 mb s3://sai.in.k8s --region ap-south-1 6. Create private hosted zone in AWS Route53 Head over to aws Route53 and create hostedzone

Choose name for example (sai.in)

Choose type as private hosted zone for VPC

Select default vpc in the region you are setting up your cluster, Hit create

7. Configure environment

variables.Open .bashrc file vi

~/.bashrc

Add the following content into .bashrc, you can choose any arbitrary name for cluster and make sure the buck name matches the one you created in the previous step. export KOPS_CLUSTER_NAME=sai.in export KOPS_STATE_STORE=s3://sai.in.k8s

Then running command to reflect variables added to .bashrc source

~/.bashrc

8. Create ssh key pair

   This keypair is used for ssh into the kubernetes cluster.

   Ssh-keygen

9. Create a Kubernetes cluster

   definition.kops create cluster \

   --state=${KOPS_STATE_STORE} \

   --node-count=2 \

   --master-size=t3.medium \

   --node-size=t3.medium \

   --zones=ap-south-1a,ap-south-1b \

   --name=${KOPS_CLUSTER_NAME} \

   --dns private \

   --master-count 1

10. Create kubernetes clusterkops

    update cluster --yes --admin

    Above command may take some

    time to create the required

    infrastructure resources on AWS.

    Execute the validate command to

    check its status and wait until the

    cluster becomes ready.

    kops validate cluster

For the above command, you might see validation failed error initially when you create a cluster and it is expected behavior, you have to wait for some more time and check again.

11. To connect to the master.ssh

    admin@api.javahome.in Destroy

    the kubernetes cluster kops delete

    cluster --yes

    Update Nodes and Master in the cluster

We can change the number of nodes and number of masters using the following commands.

kops edit ig nodes change minSize and maxSize to

0 kops get ig- to get master node name kops edit ig

- change min and max size to 0 kops update cluster

--yes

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: pythonlife
        spec:
         replicas: 3 template:
         metadata: name:
         httpd-pod labels:
         author: pythonlife
          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 which 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 scaling. It has an additional field in the spec section called "selector".

This selector uses a child element "matchLabels" where it will search for Pod based on a specific label name and try to add them to the cluster.

Create a replica set file to start 4 tomcat replicas and then perform scaling vim replica-set.yml

```
--apiVersion: apps/v1
      kind: ReplicaSet
      metadata: name:
      tomcat-rs labels:
        type: webserver
        author: pythonlife
      spec:
       replicas: 4
       selector:
       matchLabels:
         type: webserver

       template:
        metadata:
         name: tomcat-pod
         labels: type:
              webserver
        spec:
         containers: - name:
              mywebserver image:
```

```
        tomcat ports: -
            containerPort: 8080
            hostPort: 9090
...
```

To create the pods from the above file

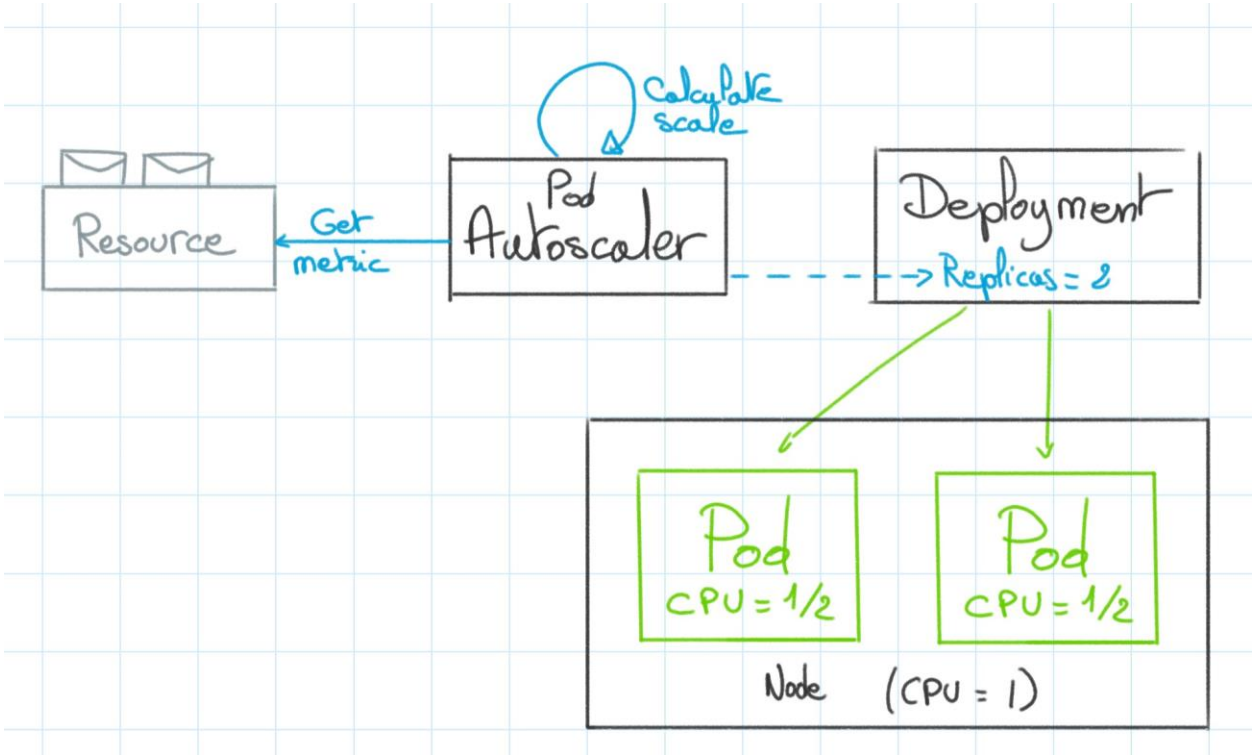
```
kubectl create -f replica-set.yml
```

Scaling in Kubernetes

Scaling can be done in 2 ways

a) Update the file and later scale it.

b) Scale from the command prompt without updating thedefinition 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 command prompt without updating thedefinition file.

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

Deployment

This is also a high level Kubernetes object which can be used for scaling and load balancing and it can also perform rolling updates.

1.Create a deployment file to run nginx:1.7.9 with 3 replicas

vim deployment1.yml

```
--apiVersion: apps/v1 kind:
      Deployment metadata:
      name: nginx-deployment
      labels:
        author: pythonlife
        type: proxyserver
      spec:
       replicas: 3
       selector:
       matchLabels:
         type: proxyserver
```

```
template: metadata:
 name: nginx-pod
 labels: type:
 proxyserver
 spec:
  containers: - name:
      nginx image:
      nginx:1.7.9 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
```

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 networks.

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.

2 **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.

3 **LoadBalancer:** 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 Case5:

Create a service definition file for port mapping an nginx pod.

1)vim pod-defintion1.yml

```
--apiVersion: v1 kind:
      Pod metadata:
      name: nginx-pod
      labels:
        author: pythonlife
        type: proxy
      spec:
       containers:
```

```
        - name:
          appserverimage:
          nginx
…
```

2)vim service1.yml

```
--apiVersion: v1
      kind: Service
      metadata:
       name: nginx-service
      spec: type:
       NodePort
       ports:
        - targetPort: 80 port:
          80 nodePort: 30008
       selector: author:
        codingrad type:
        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 accessed from any of the slave

```
kubectl get nodes -o wide
```

Take the external ip of any of the nodes:30008

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: pythonlife
      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 variable using the above secret.

vim pod-defitintion5.yml

```
--apiVersion: v1 kind:
      Pod metadata:
      name: mysql-pod
      labels: author:
      pythonlife type: db
      spec:
       containers:
          - name: mydbimage:
            mysql:5 env:
          - name:
            MYSQL_ROOT_PASSWOR
```

DvalueFrom:
secretKeyRef: name:
mysql-pass key: password
...
To create pods from above file

kubectl create -f pod-defintion5.yml

Node affinity

This is a feature of Kubernetes where we can run pods on a specific node.

kubectl get nodes --show-labels kubectl label nodes

<your-node-name> slave1=pythonlife1

1)vim node-affinity1.yml
—

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  affinity:
       nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution
      : nodeSelectorTerms: - matchExpressions:
     - key: slave1 operator: In
       values:
```

```
        - pythonlife1 containers:
        - name: nginximage: nginx
          ports:
        - containerPort: 80
          hostPort: 8080
…
```

Node affinity using deployment definition file.
vim node-affinity2.yml

```
--apiVersion: apps/v1 kind:
      Deployment metadata:
      name: tomcat-deployment
      labels:
            type: appserver
            author: pythonlife
      spec:
        replicas: 2
        selector:
            matchLabels:
            type: appserver
        template:
            metadata:
            name: tomcat-pod
            labels:
            type: appserver
            spec:
            containers:
                  - name:
                    mytomcatimage:
                    tomee affinity:
                    nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
```

```
                - key: slave1
                  operator: In values:
                - pythonlife1
...
```

Taints and Tolerations

Taints and Tolerations 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 node1=pythonlife1:NoSchedule
```

To delete the taint

```
kubectl taint nodes node1 node1=pythonlife1:NoSchedule-
```