Github: <https://github.com/in28minutes/kubernetes-crash-course>

**Kubernetes**

**Advantage of Docker**

1. Your Application can be run on any machine containing Docker engine.
2. You don’t have to tell about the configuration your App needs to run. You only have to share the image and Docker engine will take care of running it.

**Need of Kubernetes**

You have your application running on Docker, You wants to load balance it. There are several application you have to manage. It is very difficult to manage all the applications.

**Key Terms**

**Kubernates cluster:** Group of Servers that are managed together.

Kubernetes manages servers and these servers are in the cloud so they are virtual servers.

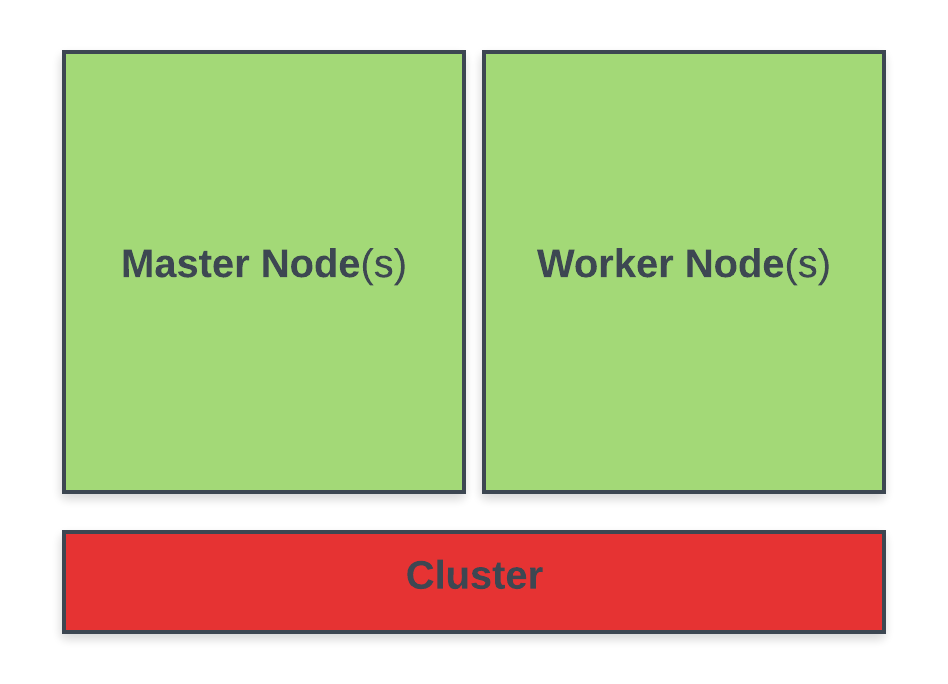
Different cloud providers have different names for these virtual servers.

* Amazon calls it AWS EC2(Elastic compute cloud.)
* Azure calls it virtual machine.
* Google cloud calls it Compute Engine
* Kubernete calls them Nodes.

Kubernetes can manage thousands of such nodes. Now, practically if you have to manage resources, you introduce Managers. So to manage thousands of such nodes we have master nodes.

Typically there is one Master nodes but when the number of Nodes increases there can be multiple master nodes.

1. The Nodes that do the work are called worker nodes.
2. The Nodes that do management work are called Master Nodes.



On a high level, Cluster containes Nodes i.e, Master Nodes and Worker Nodes.

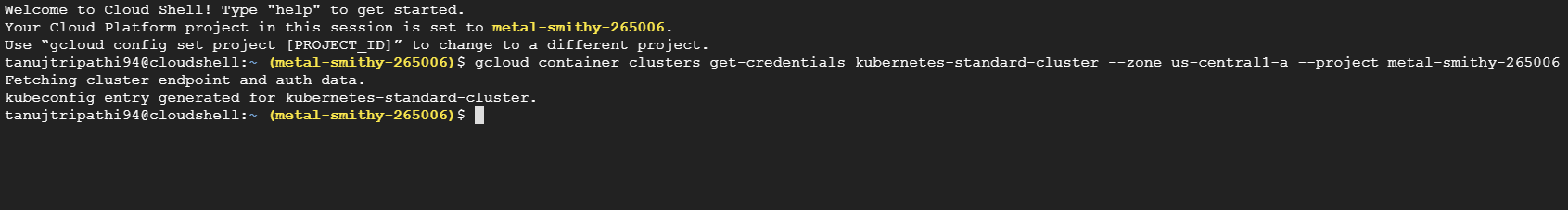
K8S stands of Kubernetes

Steps

1. Create account in Google Cloud.
2. Create cluster in one of your projects.

To connect

gcloud container clusters get-credentials kubernetes-standard-cluster --zone us-central1-a --project metal-smithy-265006



Kubectl is kubernate controller.

To create deployment:

$tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl create deployment hello-world-rest-api --image=in28min/hello-world-rest-api:1.0.0.RELEASE

**# Kubectl create deployment ->** deployment, replicas, pod

**# Kubectl expose deployment ->** service

Above image is pulled from dockerhub

To expose:

$tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl expose deployment hello-world-rest-api --type=LoadBalancer --port=8080

To get pods and replicaset

tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl get pods

NAME READY STATUS RESTARTS AGE  
hello-world-68b4f9488-5vpvx 1/1 Running 0 4m49s

hello-world-rest-api-58ff5dd898-6sqf2 1/1 Running 0 23m

tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl get replicaset

NAME DESIRED CURRENT READY AGE  
hello-world-68b4f9488 1 1 1 5m11s  
hello-world-rest-api-58ff5dd898 1 1 1 24m

**PODS**

POD is a smallest deployable unit in kubernetes.

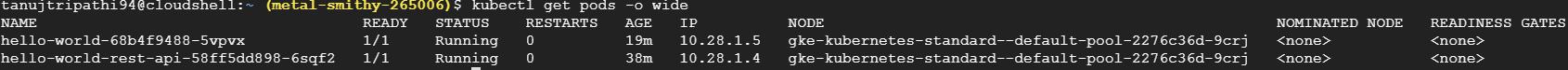
Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.

Pod is nothing but wrapper for a set of containers. A POD has an IP Address

Can you create container in kubernate without POD?

* NO, your container lives inside the pod.

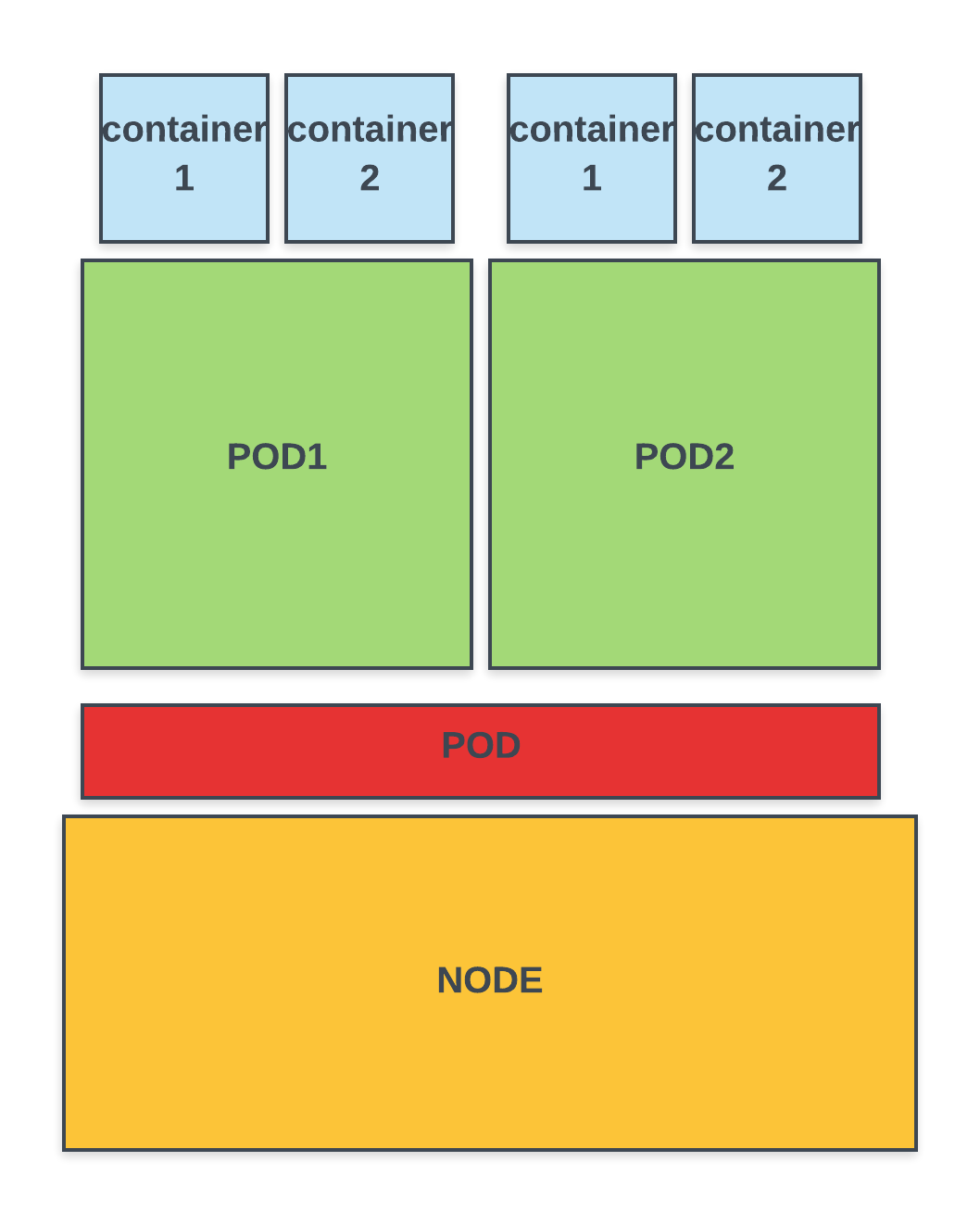
tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl get pods –o wide



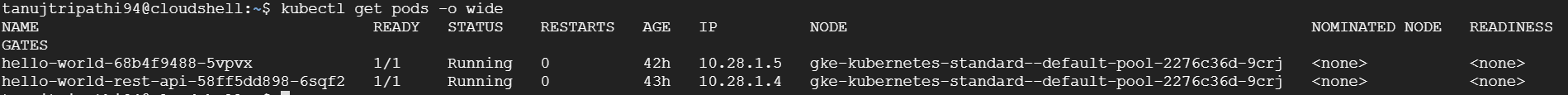
**IP:** Each pod has unique IP address.

**1/1:** Number of containers that are present in the POD

Within a POD there can be multiple containers and they can talk to each other using a localhost.

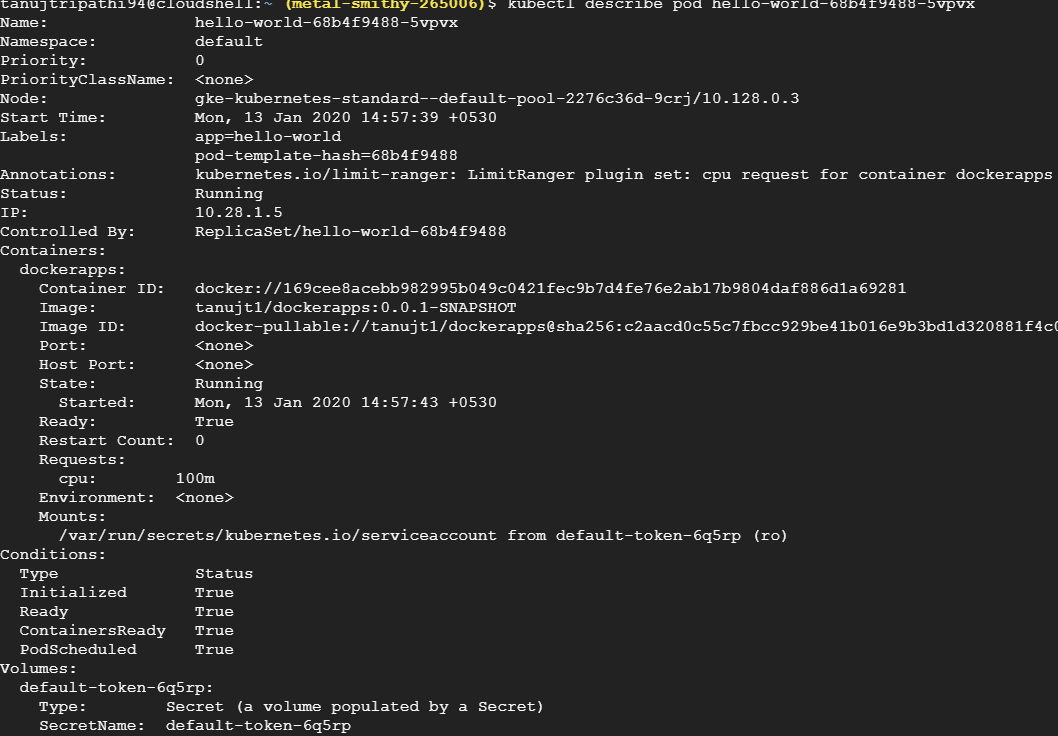


tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl explain pods



To know about the POD

tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl describe pod hello-world-68b4f9488-5vpvx



**ReplicaSet**

Replica set refers to the number of instances of PODS. So, it ensures that certain number of PODS are always running. Even if you kill one of the instance, replicaset will bring up new instance.

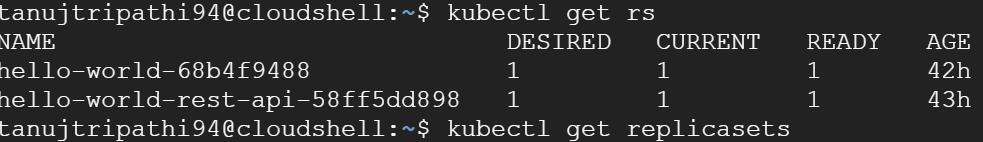
tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl get replicasets

**OR**

tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl get replicaset

**OR**

tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl get rs



When you delete pod using

tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl delete hello-world-rest-api-58ff5dd898

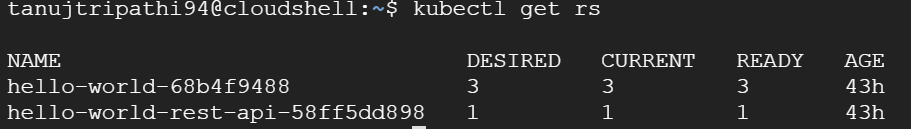
Automatically pod will be created if pods are in lesser number than it is needed. We specified minimum 1 should be present. This is done by replicaset

**SCALING:** If we want to have more instances of pod to be running, we use

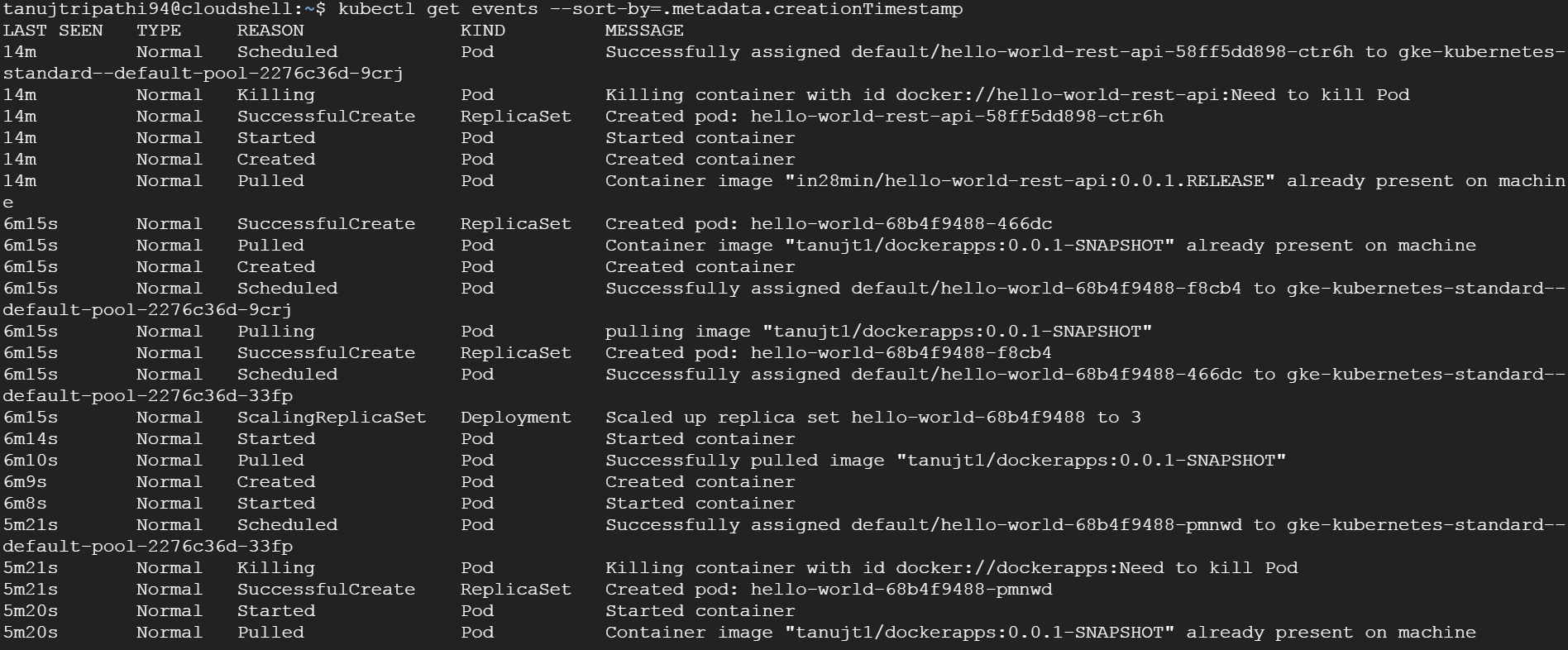
tanujtripathi94@cloudshell:~ (metal-smithy-265006)$ kubectl scale deployment hello-world --replicas=3

delpoyment.extensions/hello-world scaled

Now, replicasets are 3



If you want to see order of all the pods



**Deployment**

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl create deployment hello-world --image=tanujt1/dockerapps:0.0.1.SNAPSHOT

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl set image deployment/hello-world dockerapps=tanujt1/dockerapps:latest

**OR**

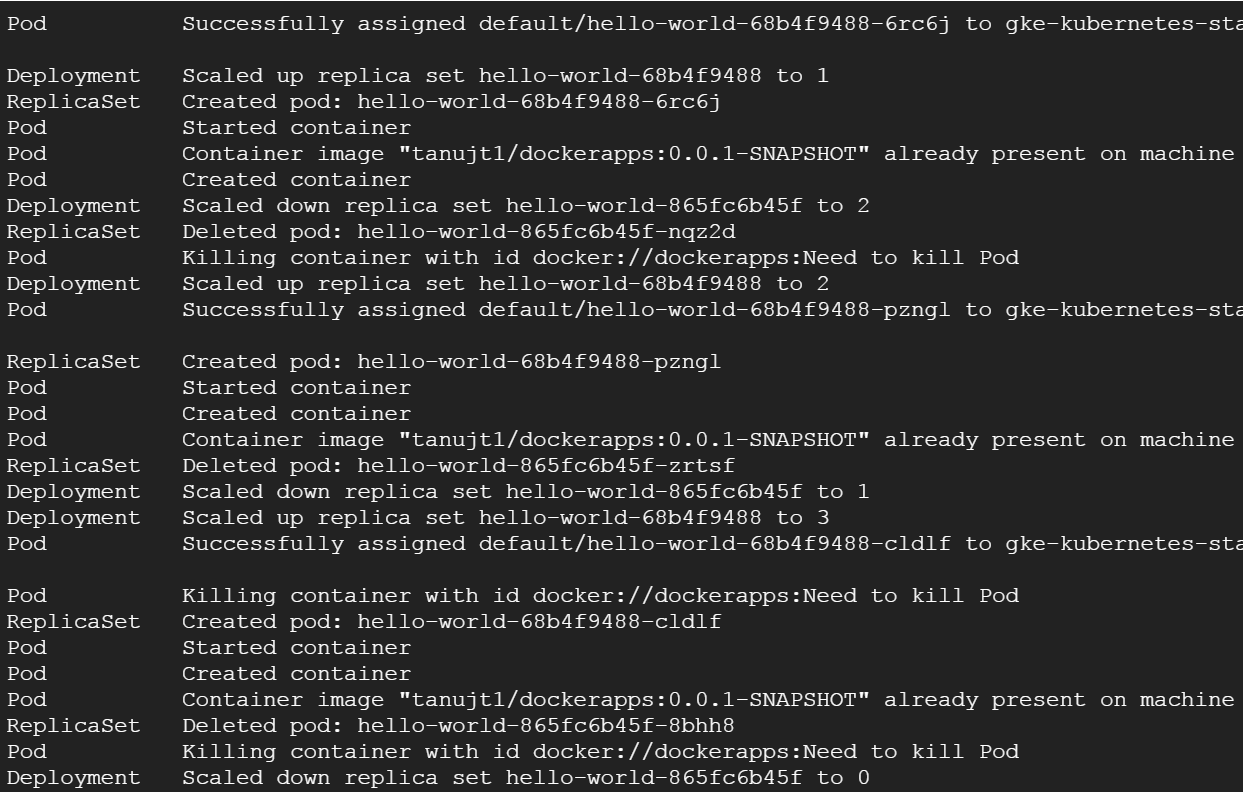
$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl set image deployment hello-world dockerapps=tanujt1/dockerapps:latest

If again you will deploy the same using the command, kubernetes will create one instance of it(new deployment). After this kubernetes will scale **DOWN** old deployment by 1 and scale **UP** the new deployment by 1. Hence both new and old deployment will be 2 and 2. Further it will scale DOWN again the old deployment by 1 and scale UP the new deployment by 1 this making Old deployment instance 1 and new deployment instances by 3. At the end the old deployment instance goes down to 0.

**Steps:**

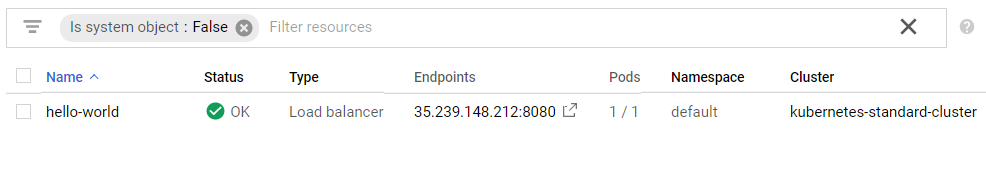
1. Create deployment say V1 with replicas 3 i.e, 3 instances.
2. Update deployment with other image say V2.
3. Kubernetes will create 1 instance of V2 initially.
4. Now, Kubernetes will scale down 1 instance of V1 making 2 instances availabe.
5. Then kubernetes will scale UP 1 instance of V2
6. Now, V1 and V2 both will have 2 instances.
7. Again Kubernetes will bring down 1 instance of V1 making 1 instance availabe.
8. Kubernetes will scale UP one instance making 3 instances of V2 availbale.
9. In this end kubernetes will bring down last 1 instance of V1 thus making instances 0.

REFER

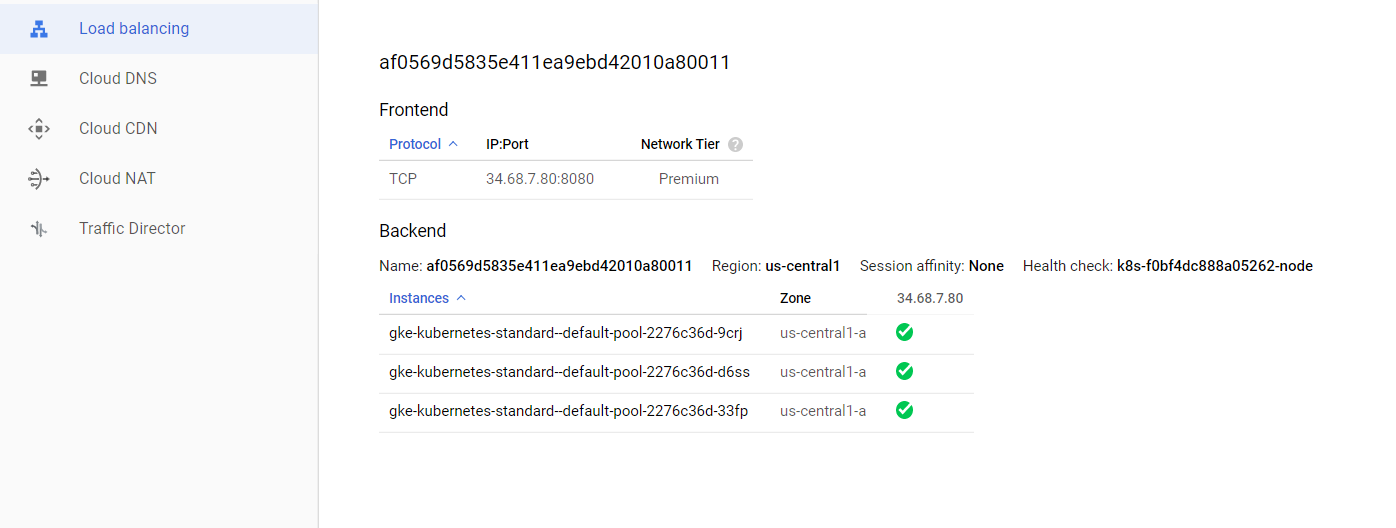


In the above you can see 3 sections and every section is scaling DOWN and deleting old replica set and scaling up new replica set. Replica means instance.

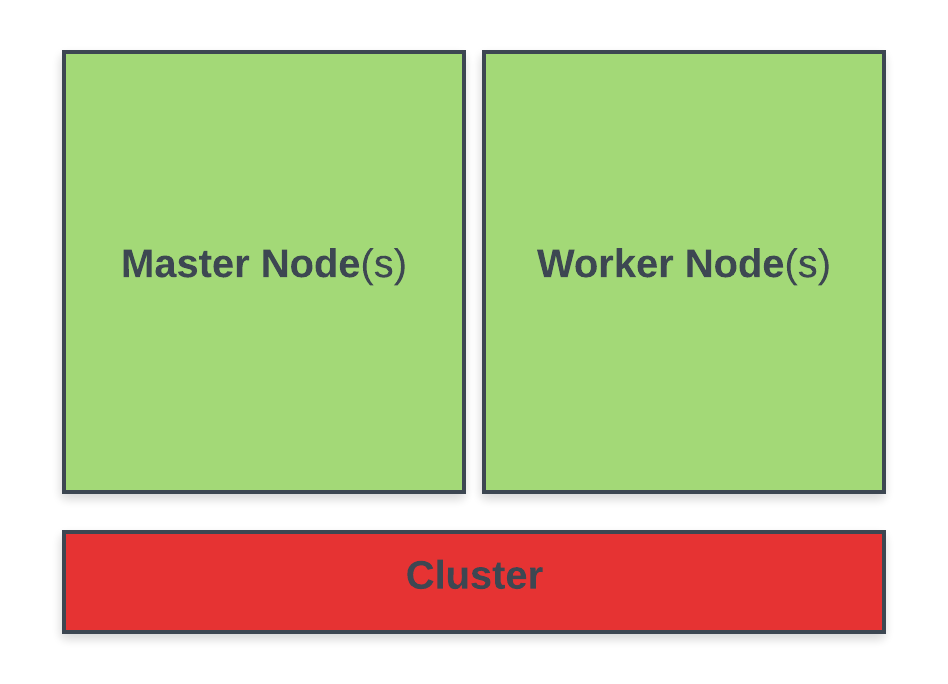
Every POD has its own IP address. If you delete any of the PODS, Kubernetes will bring up another POD with new IP address i.e, suppose if the IP address is 10.4.1.8, new IP will be 10.4.1.9 most probably. In kubernetes a POD is a throw away thing i.e, they are deleted, updated, modified.



Here you can see for above Service Type is Load balancer i.e, load is balanced between all the PODS. In kubernetes for every service there is specific load balancer.

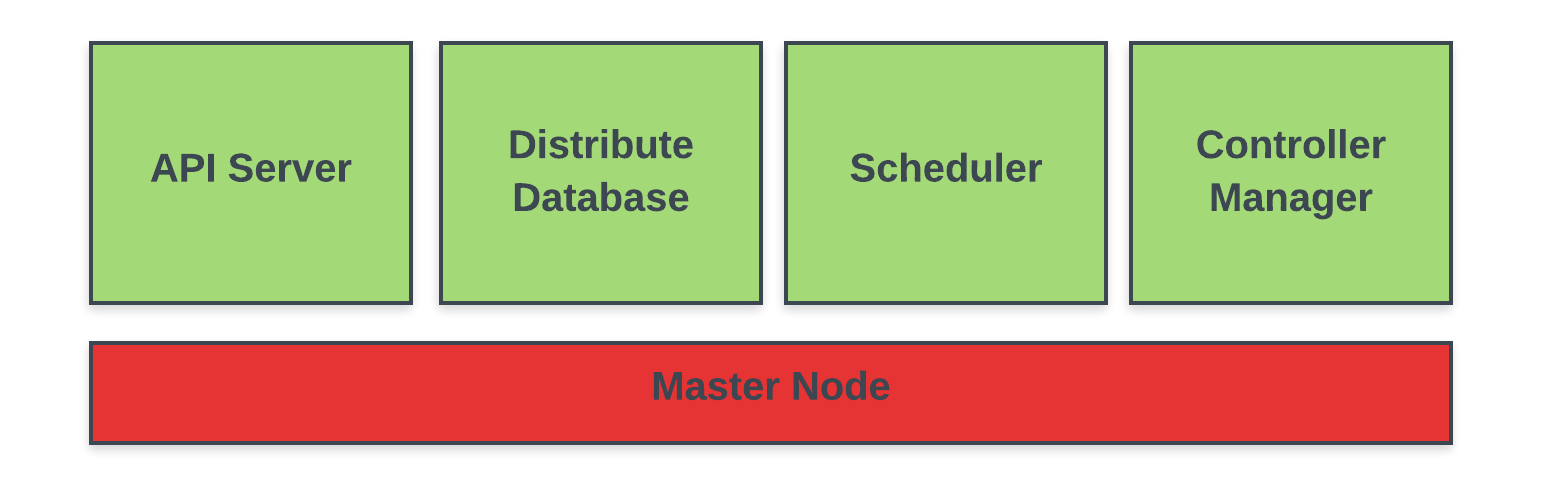


**Kubernetes Architecture**



**Fig: Kubernetes Cluster**

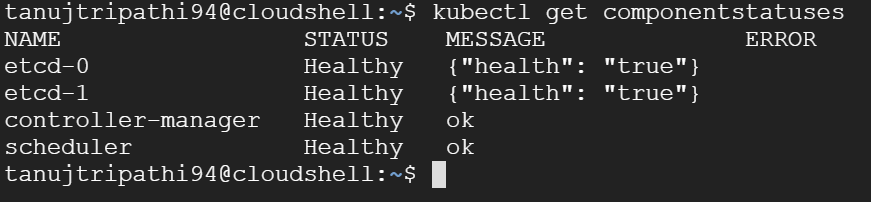
**Master Node**



**API Server :** Whenever we are using kubectl internally it is talking to this API server which is called kube-apiserver.

**Distributed Database**: It is called **etcd**, All the configuration changes that we are making, all the deployments that we are creating, all the scaling operations that we are doing, All the details of those are stored in Distributed Database. What we are doing when we runs commands -> we want some desired state for kubernetes. So this desired state is stored in this etcd. Typlically there are 3 to 5 replicas of this database so that kubernetes cluster state is not lost.

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl get componentstatuses

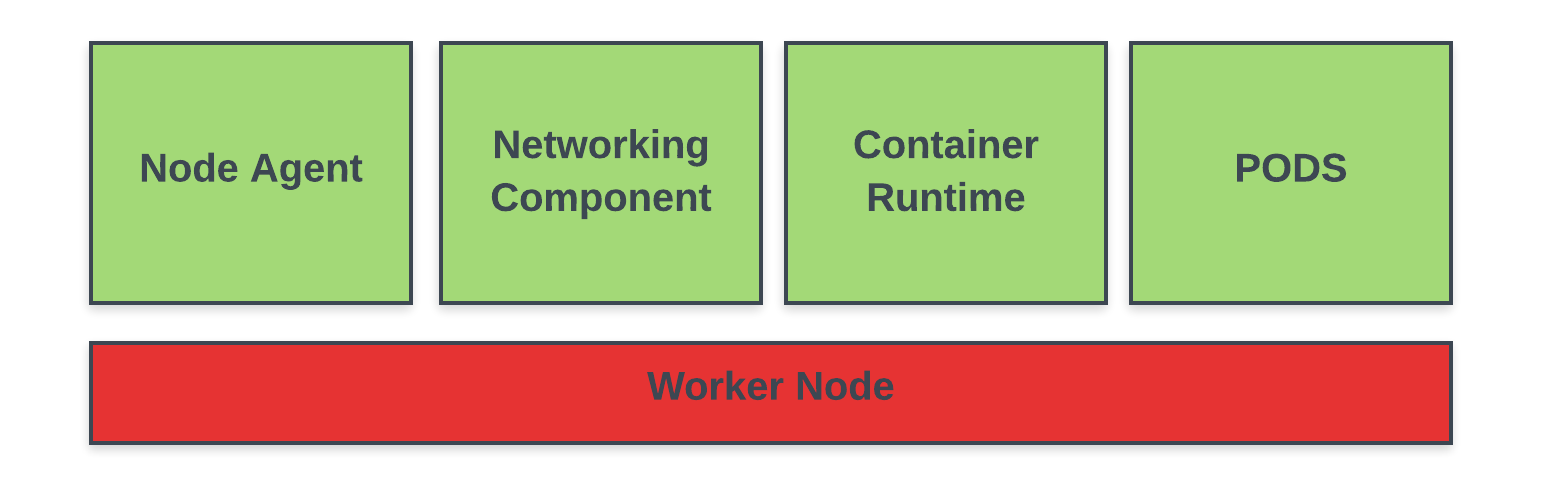


In the above there are two distributed databases.

**Scheduler:** Scheduler is responsible for scheduling PODS in available Nodes i.e, which POD should run in which NODE.

**Controller Manager**: Manages the overall health of the cluster.

**Worker NODE**



**Node Agent:** Its called a kubelet. Its job is to monitor what happing in Node and communicate back it to Master Node.

**Networking component**: Its called kube proxy. We expose our deployment as a service is possible due to this component.

**Container Runtime:** Running container inside POD mostly docker container

**Gcloud & Kubectl locally**

Instead of Google cloud terminal we can use it in locally

Download gloud and kubectl

**Rollout**

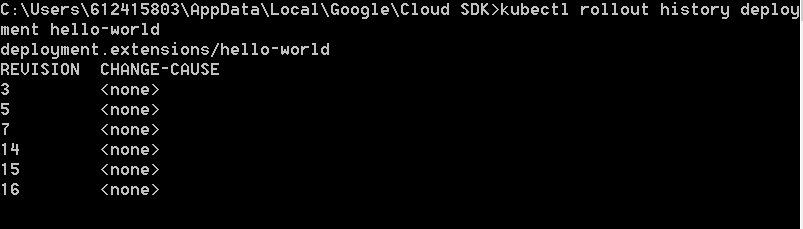
Whenever you update the deployment and change the version, you have rollout to its previous version also.

i.e, If you have version 0.0.1-SNAPSHOT and change it to version 0.0.2-SNAPSHOT using:

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl set image deployment hello-world dockerapps=tanujt1/dockerapps:latest

you can see the history of all the changed version as:

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl rollout history deployment hellow-world

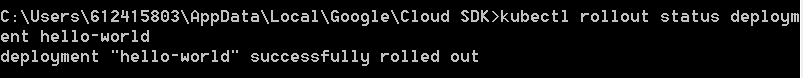


Above are all the version that we moved our deployment to. There is <none> becaue we have not use –record.

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl set image deployment hello-world dockerapps=tanujt1/dockerapps:latest --record

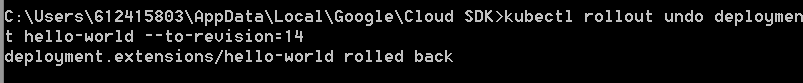
If we want to status of the successful latest deployment:

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl rollout status deployment hellow-world



You can also undo and go back to previous version

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl rollout undo deployment hellow-world –to-revision=14



**Logs**

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl logs hello-world-696699ffs -f

**If you want to continuosly run some api you can use watch**

**Yaml**

You can also get yaml by UI or by command and make change in it and apply

Get yaml and save locally in deployment.yaml file

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl get deployment hello-world –o yaml > deployment.yaml

Apply

$tanujtripathi94@cloudshell:~ (metal-smithy-265006) $ kubectl apply –f deployment.yaml

At the high leve 4 things are present in the yaml file

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

under spec the most important is POD

deployment.yaml

apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
 annotations:  
 deployment.kubernetes.io/revision: "19"  
 creationTimestamp: "2020-01-13T09:27:39Z"  
 generation: 20  
 labels:  
 app: hello-world  
 name: hello-world  
 namespace: default  
 resourceVersion: "726605"  
 selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/hello-world  
 uid: f13fac22-35e6-11ea-9ebd-42010a800114  
spec:  
 progressDeadlineSeconds: 600  
 replicas: 3  
 revisionHistoryLimit: 10  
 selector:  
 matchLabels:  
 app: hello-world  
 strategy:  
 rollingUpdate:  
 maxSurge: 25%  
 maxUnavailable: 25%  
 type: RollingUpdate  
 template:  
 metadata:  
 creationTimestamp: null  
 labels:  
 app: hello-world  
 spec:  
 containers:  
 - image: tanujt1/dockerapps:0.0.1-SNAPSHOT  
 imagePullPolicy: IfNotPresent  
 name: dockerapps  
 resources: {}  
 terminationMessagePath: /dev/termination-log  
 terminationMessagePolicy: File  
 dnsPolicy: ClusterFirst  
 restartPolicy: Always  
 schedulerName: default-scheduler  
 securityContext: {}  
 terminationGracePeriodSeconds: 30  
status:  
 availableReplicas: 3  
 conditions:  
 - lastTransitionTime: "2020-01-15T04:35:18Z"  
 lastUpdateTime: "2020-01-15T04:35:18Z"  
 message: Deployment has minimum availability.  
 reason: MinimumReplicasAvailable  
 status: "True"  
 type: Available  
 - lastTransitionTime: "2020-01-13T09:27:39Z"  
 lastUpdateTime: "2020-01-15T18:02:20Z"  
 message: ReplicaSet "hello-world-68b4f9488" has successfully progressed.  
 reason: NewReplicaSetAvailable  
 status: "True"  
 type: Progressing  
 observedGeneration: 20  
 readyReplicas: 3  
 replicas: 3  
 updatedReplicas: 3

If we are changing the deployment say form 0.0.1 to 0.0.2 then there will be some downtime and the application won’t work as new version will require some time to go UP.

To tackle this we add minReadySeconds under spec in yaml.

spec:  
 minReadySeconds: 45

**While creating we can use the below cleaned yaml which has deployment and service**

apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
 labels:  
 app: hello-world  
 name: hello-world  
 namespace: default  
spec:  
 replicas: 3  
 minReadySeconds: 45  
 selector:  
 matchLabels:  
 app: hello-world  
 strategy:  
 rollingUpdate:  
 maxSurge: 25%  
 maxUnavailable: 25%  
 type: RollingUpdate  
 template:  
 metadata:  
 labels:  
 app: hello-world  
 spec:  
 containers:  
 - image: tanujt1/dockerapps:0.0.1-SNAPSHOT  
 imagePullPolicy: IfNotPresent  
 name: dockerapps  
 resources: {}  
 restartPolicy: Always  
 terminationGracePeriodSeconds: 30  
---  
apiVersion: v1  
kind: Service  
metadata:  
 labels:  
 app: hello-world  
 name: hello-world  
 namespace: default  
spec:  
 ports:  
 - nodePort: 32239  
 port: 8080  
 protocol: TCP  
 targetPort: 8080  
 selector:  
 app: hello-world  
 sessionAffinity: None  
 type: LoadBalancer