**Kubernetes Architecture**

Kubernetes is a stack of services that work together to manage all the hosts, Kubernetes cluster is what we call them together AKA K8s.

All those services or components are shown in the below three diagrams.

**POD:**

A pod is a group of one or more containers (such as Docker containers), the shared storage for those containers, and options about how to run the containers.

Pods are always co-located and co-scheduled, and run in a shared context.

A pod models an application-specific "logical host" - it contains one or more application containers which are relatively tightly coupled -- in a pre-container world, they would have executed on the same physical or virtual machine.

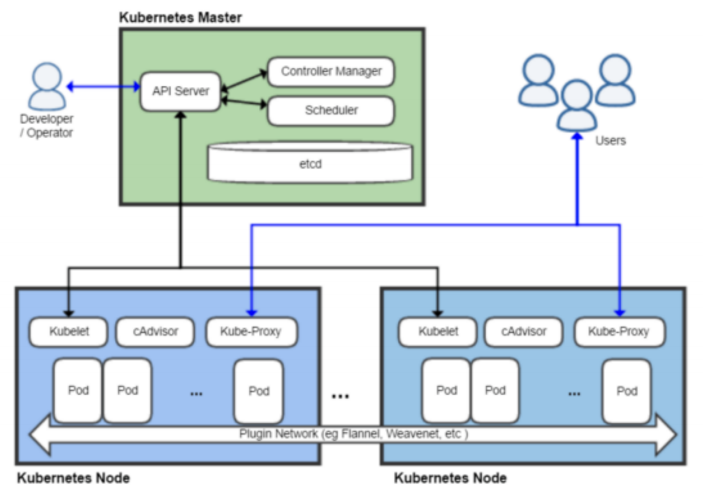
**SERVICE:**

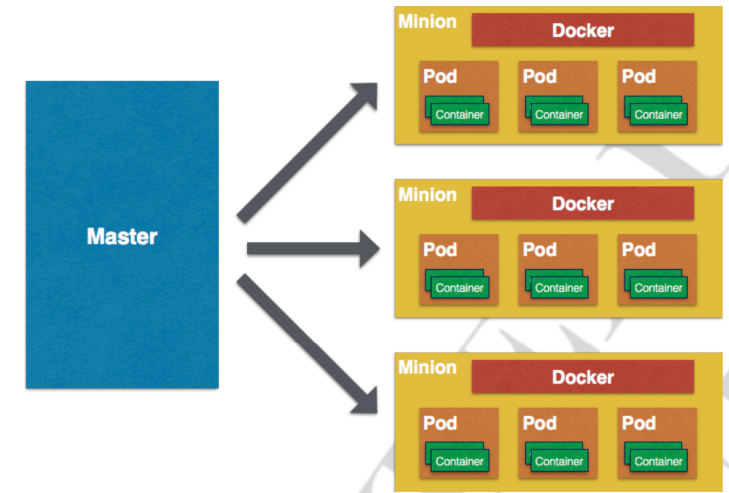
Kubernetes Pods are mortal.

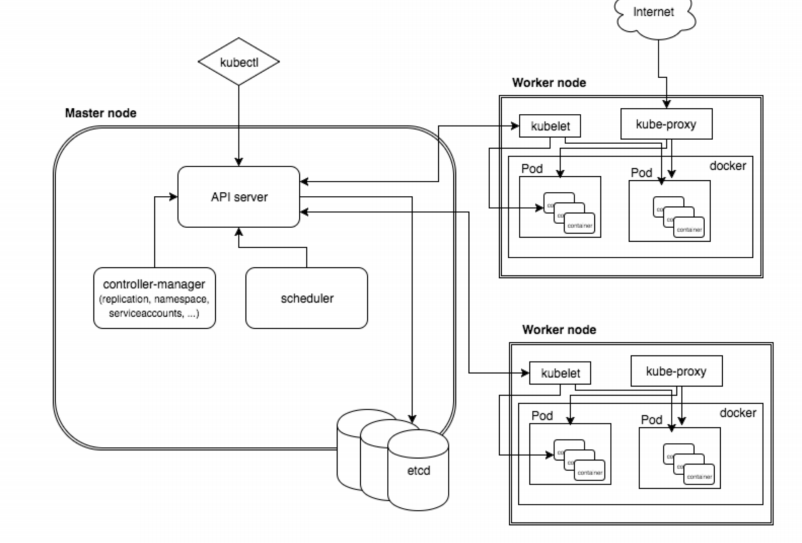
They are born and when they die, they are not resurrected. (they are not bring back to life)

ReplicationControllers in particular create and destroy Pods dynamically(e.g. when scaling up or down or when doing rolling updates).

While each Pods gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem. Service will be on top of the POD and will have a stable IP, it's like a load balancer, so not matter what nodes you have under load balancer and what their IP is it can be accessed by the Load balancer. Its similar for the service in Kubernetes.







**Node:**

A node is a worker machine in Kubernetes, previously known as a minion. A node may be a VM or physical machine, depending on the cluster. Each node contains the services necessary to run [pods](https://kubernetes.io/docs/concepts/workloads/pods/pod/) and is managed by the master components. The services on a node include the [container runtime](https://kubernetes.io/docs/concepts/overview/components/" \l "node-components), kubelet and kube-proxy.

**Master Node Architecture**

**Etcd:** It is an open source key-value store developed by CoreOs team.

Kubernetes uses 'Etcd' to store the configuration data accessed by all nodes (minions and master) in the cluster.

Example of data stored by Kubernetes in etcd are jobs being scheduled, created and deployed pod/service details and state, namespaces and replication information, etc.

**Kube-ApiServer:**

The kubernetes api-server generally validates the configuration data store in 'Etcd' and the details of the deployed container that are in agreement.

It also provides a RESTful interface to make communication easy.

**Kube-Schedule Server:**

The deployment of configured pods and services onto the nodes is done by the scheduler component.

It is responsible for assigning task to minions in the cluster.

Scheduler has the information regarding resources available on the members of the cluster, as well as the ones required for the configured service to run and hence is able to decide where to deploy a specific service.

**Kube-Controller-Manager:**

It is generally responsible for handling the cluster level function such as replication controller.

Whenever the desired state of the cluster changes it is written to Etcd and then the controller manager tries to bring up the cluster in the desired state.

A controller uses API server to watch the shared state of the cluster and makes corrective changes to the current state to bring it to the desired one.

One example is Replication Controller which takes care of the PODS in the system, if any POD fails it replaces that with a new POD.

**Minion Node or Worker Node Architecture**

**Docker:** One of the basic requirement of nodes is Docker. Docker is responsible for pulling down and running container from Docker images.

**Kube-Proxy:** Every node in the cluster runs a simple network proxy. Using proxy node in cluster routes request to the correct container in a node.

**kubelet:** It is an agent process that runs on each node.

It is responsible for managing pods and their containers.

It Deal with pods specification which are defined in YAML or JSON format.

Kubelet takes the pod specifications and checks whether the pods are running health or not.

**Flannel:** It is an overlay network that works on assigning a range of subnet address. It is used to assign IPs to each pods running in the cluster and to make the pod-to-pd and pod-to-services communications.

**Objective of kubernetes is to**

1.Automate Application deployment

2.Auto Scaling

3.Automatic Cluster maintenance

To interact with Kubernetes cluster we have kubernetes client. (ie kubectl)

In our example:

Kubernetes cluster has One master node and 3 worker Nodes.

Kubernetes Master Node receives request from kubernetes client and it issues those instructions to the worker nodes to perform a specific task in the cluster.

For example: If we want to deploy some application into kubernetes cluster with 3 replicas.

1st we issue the instruction to master using kubernetes client(kubectl).

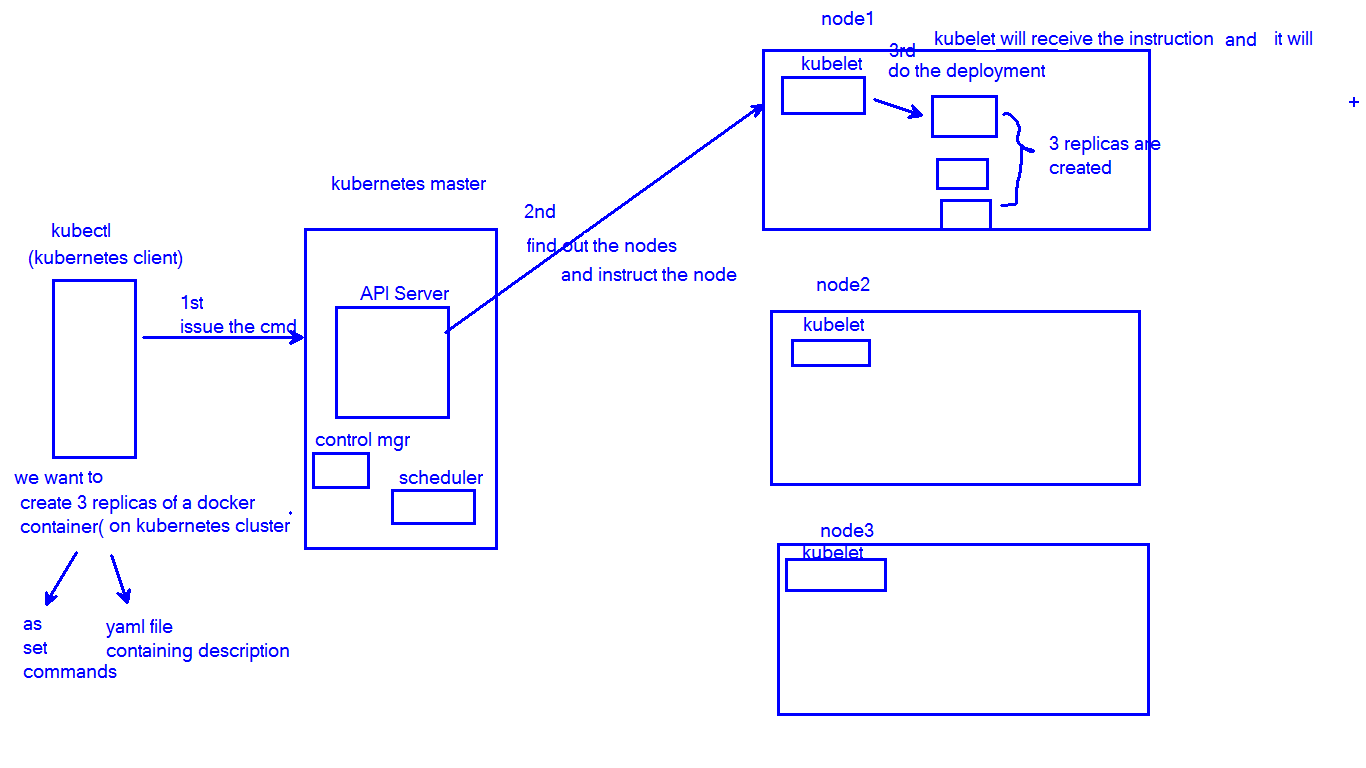
2nd Master receives the instruction from client. It does a scheduling.

Scheduling: scheduling means finding out nodes on which it has to create 3 replicas.

After finding out those nodes it sends the instruction to those nodes with details about the containers it has to create and how many replicas it has to create on the cluster. (as per our example 3 replicas)

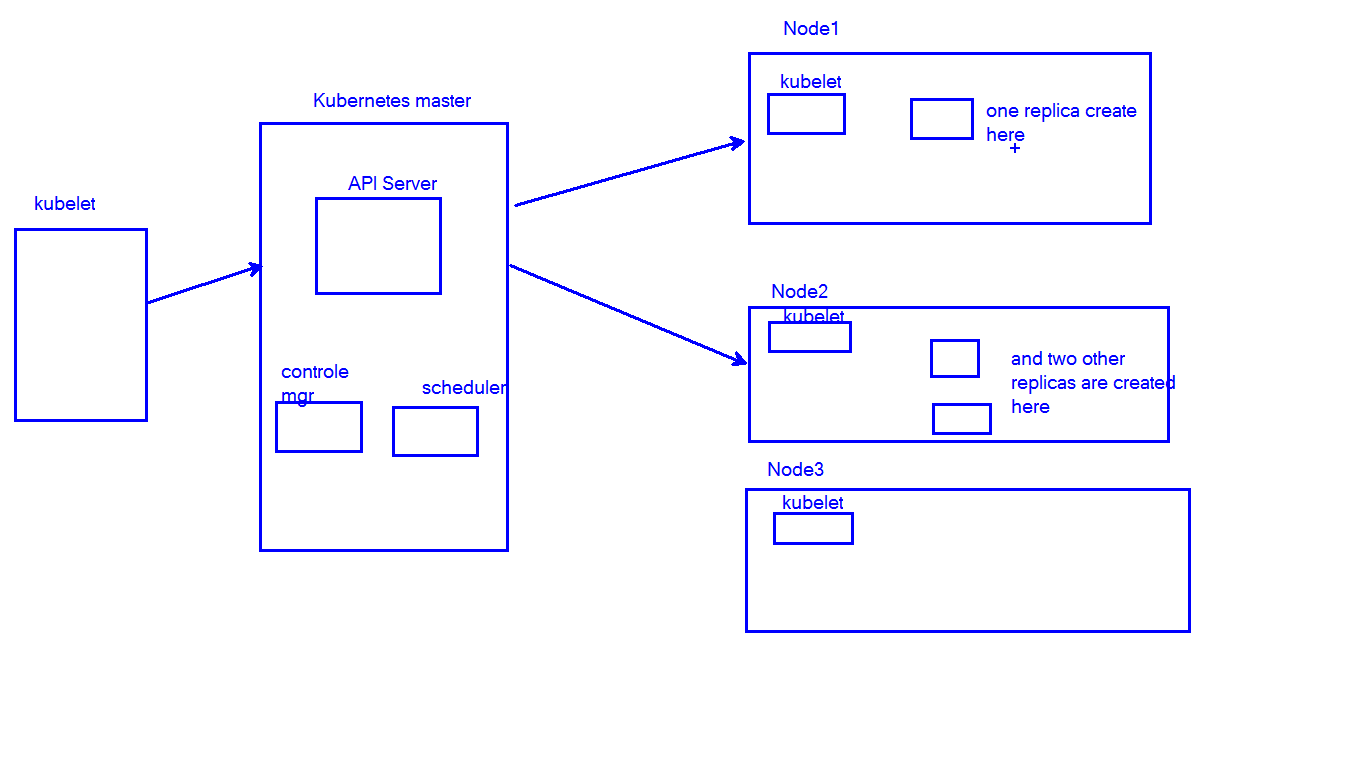
3rd kubelet will receive the instruction and it will do the deployment

See the below diagram for more details:



replicas can be created in as single node or may be spawned in multiple nodes.

Here replica of the container means pod with container.



**Node failure Case:**

In a cluster if a specific node fails then what happens to the containers running on that node?

Ans: The node will be automatically recovered by kubernetes.

When there is any node failure master detects that failure and it will have the information about what are the containers running on that specific node.

And Master tries to reschedule things and find out a node and create that replica on a specific node.

(In simple kubernetes will do scheduling and find out node where it can create the replica of the same container.)

**POD in action:**

What is Pod in kubernetes?

ans: Atomic unit of deployment.

To deploy any application in kubernetes cluster we minimum need a pod.

Where pods are located?

Ans: On worker Nodes we generally deploy the PODS

.

How to communicate with POD?

Ans: Using IP, Every POD contains its own IP Address.

What is there in POD?

Ans: POD contains one or more containers.

(one main container + helper containers)

Note: The IP address allocated to the POD is reachable with in the cluster from any Node.

How one POD can Communicate with another POD running in same node or different nodes?

Ans: In cluster the POD running on one node can communicate with POD running on another node by using POD IP.

Note: In POD multiple containers must Occupy different PORT Numbers.(unique port number)

We can understand that two containers in the same POD cannot run on same port number.

We can have multiple PODS on single node.

On single node we can run multiple containers of same application type or different type of application.

Assume that we have two spring boot applications as container: 1) search service 2) booking service

Assume that POD1 and POD2 are on the same worker node then POD1 has search service and POD2 can have search service or it can have booking service.

Containers with in same POD can communicate with each other using local host.

In kubernetes we have different kind of objects and they can be deployed Example: POD, Deployments, Services, replica sets etc.

How to deploy a POD or any object?

Ans: To deploy a pod / or any object we can use either commands or by describing the commands in an YAML file.

In the .yaml file for the Kubernetes object you want to create, you’ll need to set values for the following fields:

* apiVersion - Which version of the Kubernetes API you’re using to create this object
* kind - What kind of object you want to create
* metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
* spec - What state you desire for the object

refer the below for more info:

**Understanding Kubernetes Objects**

<https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>

[https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#node-v1-core](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/" \l "node-v1-core)

<https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/>

[https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#types-kinds](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md" \l "types-kinds)

**example:**

**pod.yaml**

apiVersion: v1

kind: Pod

metadata:

name: tomcat-pod

labels:

app: tomcat-pod-l

spec:

containers:

- name: tomcat-pod-c

image: tomcat

ports:

- containerPort: 8080

Install a yaml file by using minikube / google cloud Kubernetes / aws kops / aws eks(Amazon Elastic Kubernetes Service)

**Note:** master node and worker nodes are physical machines or virtual machines in Kubernetes cluster.

Pods are logical hosts.

Pod will internally contain docker containers.

Deployment is collection of identical pods.

Deployment ensures that they have the correct config and that the right number of them exist.

**Setup Minikube:**  
Minikube can be downloaded from its github repo  
  
<https://github.com/kubernetes/minikube>  
  
source: <https://kubernetes.io/docs/setup/learning-environment/minikube/>  
  
Note: Virtualbox is the dependency for minikube installation. So first install virtual box.  
  
we are going to setup minikube on a linux machine so check the latest release url and   
download the package.  
<https://kubernetes.io/docs/tasks/tools/install-minikube/>  
  
Install Minikube via direct download  
  
sudo apt-get update  
sudo apt install curl  
curl -Lo minikube <https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64>  && chmod +x minikube    
  
sudo mkdir -p /usr/local/bin/  
sudo install minikube /usr/local/bin/  
  
**Kubectl**   
Kubectl is a command line interface for running command against kubernetes clusters.  
  
Installing and setting up kubectl   
<https://kubernetes.io/docs/tasks/tools/install-kubectl/>  
  
Note: kubectl (kube control) will read configuration from /home/username/.kube/config file and it will contain the IP, Port, Auth details to connect to kubernetes cluster.  
  
apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod  
   labels:  
    app: tomcat-pod-l  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080  
         
  
To create a pod from yml:  
*$kubectl create -f pod.yml*

Exposing the pod as Internal facing service

*$kubectl expose pod tomcat-pod --name=tomcat-service --type=ClusterIP*

Exposing the pod as External facing service

*$kubectl expose pod tomcat-pod --name=tomcat-service --type=NodePort*

*$kubectl expose pod tomcat-pod --name=tomcat-service --type=LoadeBalancer*

To enable firewall in google cloud node port

If the nodes in your cluster have external IP addresses, find the external IP address of one of your nodes:

*$kubectl get nodes --output wide*

The output shows the external IP addresses of your nodes:

NAME STATUS ROLES AGE VERSION EXTERNAL-IP

gke-svc-... Ready none 1h v1.9.7-gke.6 203.0.113.1

Not all clusters have external IP addresses for nodes.

For example, the nodes in private clusters do not have external IP addresses.

Create a firewall rule to allow TCP traffic on your node port:

$gcloud compute firewall-rules create test-node-port --allow tcp:[NODE\_PORT]

How to access tomcat in this example?

Ans: http://any-node-external-ip:nodeport

To get all node details of a Kubernetes cluster:

$Kubectl get nodes

$Kubectl get node -o wide

To get running pod details:  
$kubectl get pods

$kubectl get pods -o wide

To delete an existing pod  
$kubectl delete -f pod.yml  
or  
$kubectl delete pods tomcat-pod(the name of the pod)

To get running service details:

$kubectl get services

To delelete a service

$kubectl delete service nameoftheserice

Creating the pod with replicaset, deployment without using yaml/json  
$kubectl run tomcat --image=tomcat:8.5 --port=8080  
or  
$kubectl run --generator=run-pod/v1 tomcat --image=tomcat:8.5 --port=8080  
$kubectl get pods

$kubectl get deployments

$kubectl get replicaset

$kubectl get replicationcontroller

If we delete the deployment all will be delete try the below commands:

$kubectl delete deployment tomcat

$kubectl get deployment

$kubectl get replicaset

$kubectl get pods

Creating only pod without using yaml /json

$kubectl run tomcat --image=tomcat --port=8080 --restart=Never

Output

pod/tomcat created

More information about restart option:

--restart=Always:

The restart policy for this Pod. Legal values [Always, OnFailure, Never].

If set to Always a deployment is created,

if set to OnFailure a job is created,

if set to Never, a regular pod is created.

For the latter two --replicas must be 1. Default Always [...]

$kubectl get pod -o wide

$kubectl get deployment -o wide

We can expose pod as service.

We can also expose a deployment as service.

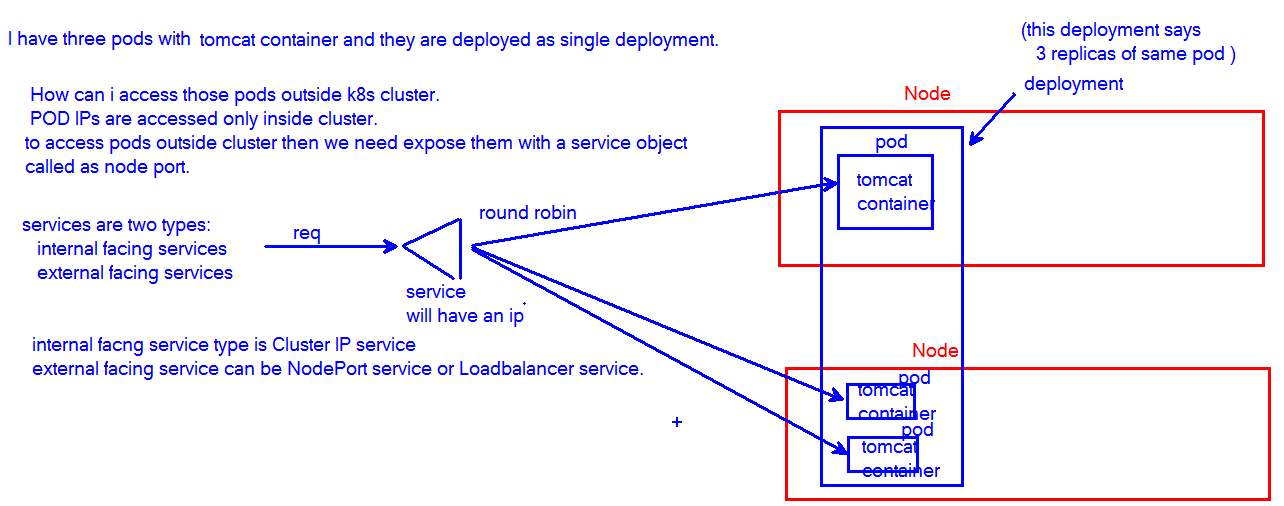
We rarely create only pods in production.

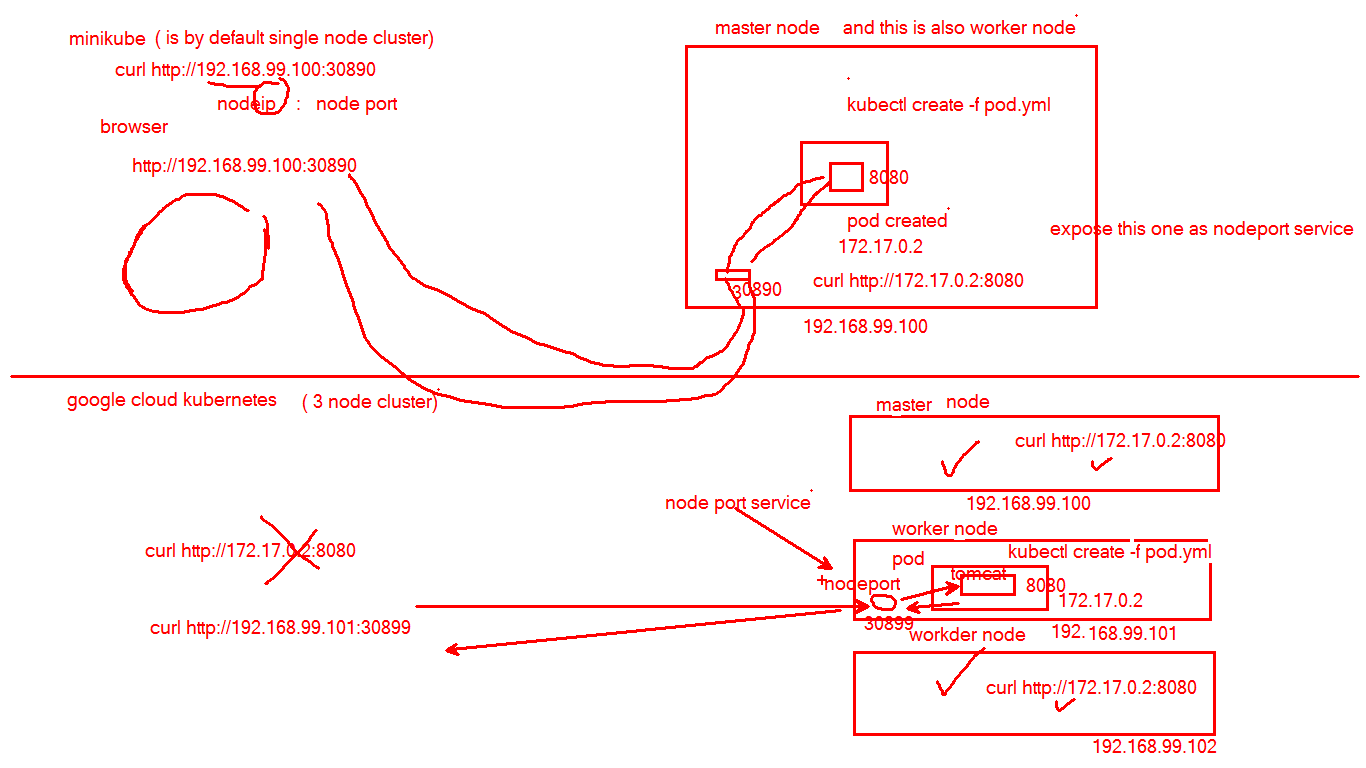
Mostly We will create pods with deployments in production.

So in production we do expose deployments as services. (culsterIP , NodePort, LoadBalancer types etc)

$kubectl expose deployment tomcat --name=tomcat-service --type=NodePort

Troubleshooting with kubectl  
kubectl get -> list resources  
examples: kubectl get nodes  
          kubectl get pods  
          kubectl get deployments  
          kubectl get services  
  
kubectl describe -> show detailed informatin about a resource  
examples:  
kubectl describe pods tomcat-pod(pod name)  
kubectl describe nodes node-name  
kubectl describe deployments deployment-name  
kubectl describe services service-name  
  
kubectl logs -> print the logs from a container in a pod  
examples: kubectl logs pod-name  
  
kubectl exec -> execute a command on a container in pod  
examples: kubecechoservertl exec -it pod-name /bin/bash  
  
By using these commands we can see when application were deployed  
                                   what is their current status  
                                   where they are running  
                                   what is their configuration.  
  
  
$kubectl run tomcat --image=tomcat:8.5 --port=8080  
$kubectl expose deployment tomcat --name=tomcat-service --type=NodePort

$minikube service tomcat-service --url  
  




**ReplicationController:**

A ReplicationController ensures that a specified number of pod replicas are running at any one time.

In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

**Note:** A [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/) that configures a [ReplicaSet](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) is now the recommended way to set up replication.

https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/

**Use case to understand labels and selectors.**

**Label selectors**

Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

**Pod Selector**

The .spec.selector field is a label selector. A ReplicationController manages all the pods with labels that match the selector. It does not distinguish between pods that it created or deleted and pods that another person or process created or deleted. This allows the ReplicationController to be replaced without affecting the running pods.

**tomcat-pod1.yaml**

apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod1  
   labels:  
    app: tomcat-l  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080

**tomcat-pod2.yaml**

apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod2  
   labels:  
    app: tomcat-l  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080

**tomcat-pod3.yaml**

apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod3  
   labels:  
    app: tomcat-l  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080

**tomcat-pod4.yaml**

apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod4  
   labels:  
    app: tomcat-l  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080

**tomcat-pod5.yaml**

apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod5  
   labels:  
    app: tomcat-l2  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080

create all the five pods

kubectl create -f tomcat-pod1.yaml

kubectl create -f tomcat-pod2.yaml

kubectl create -f tomcat-pod3.yaml

kubectl create -f tomcat-pod4.yaml

kubectl create -f tomcat-pod5.yaml

first four pods label is tomcat-l

fifth pod label is tomcat-l2

create replication controller for 3 tomcat pods and give label selector as tomcat-l

**rc.yaml**

apiVersion: v1

kind: ReplicationController

metadata:

name: tomcat-rc

spec:

replicas: 3

selector:

app: tomcat-l

template:

metadata:

name: tomcat

labels:

app: tomcat-l

spec:

containers:

- name: tomcat-c

image: tomcat

ports:

- containerPort: 8080

$kubectl create -f rc.yaml

$kubectl get pods

How many we will see?

1 terminating or terminated

We will see only 4.

3 are having tomcat-l as label

1 I having tomcat-l1 as label (ie tomcat-pod5)

Replication controller selects the pods based on selector label given in yaml file (ie tomcat-l)

**Note:** Delete all pods and replication controller and try the below the commands.

**rc.yaml**

apiVersion: v1

kind: ReplicationController

metadata:

name: tomcat-rc

spec:

replicas: 3

selector:

app: tomcat-l

template:

metadata:

name: tomcat

labels:

app: tomcat-l

spec:

containers:

- name: tomcat-c

image: tomcat

ports:

- containerPort: 8080

$kubectl create -f rc.yaml

$kubectl get pods

Now you can see three tomcat pods are create

$kubectl get replicationcontroller

Or

$kubectl get rc

$kubectl describe rc

Create a new pod

**tomcat-pod.yaml**

apiVersion: v1  
kind: Pod  
metadata:  
   name: tomcat-pod  
   labels:  
    app: tomcat-l  
spec:  
  containers:  
    - name: tomcat-pod-c  
      image: tomcat  
      ports:  
        - containerPort: 8080

$kubectl create -f tomcat-pod.yaml

Tomcat-pod is created

$kubectl get pods (immediately then tomcat-pod will be under termination)

Now delete a pod

$Kubectl delete pod nameofthepod

$kubectl get pods

You will observe it will create one more pod

scaleup/down the replication controller with replication controller name

$kubectl describe rc tomcat-rc

$kubectl scale --replicas=4 rc tomcat-rc

$kubectl get pod

$kubectl scale --replicas=1 rc tomcat-rc

$kubectl get pod

scale up / down with the replication controller’s with definition file.

$kubectl scale --replicas=4 -f rc.yaml

$kubectl get pod

$kubectl scale --replicas=1 -f rc.yaml

**ReplicaSet**

A ReplicaSet’s purpose is to maintain a stable set of replica Pods running at any given time.

As such, it is often used to guarantee the availability of a specified number of identical Pods.

Replica Set and Replication Controller do almost the same thing.

Both of them ensure that a specified number of pod replicas are running at any given time.

The difference comes with the usage of selectors to replicate pods.

Replica Set use Set-Based selectors while replication controllers use Equity-Based selectors.

**replicatSet.yaml**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: tomcat

spec:

replicas: 2

selector:

matchLabels:

app: tomcat-r-l

template:

metadata:

labels:

app: tomcat-r-l

spec:

containers:

- name: tomcat-c

image: tomcat

ports:

- containerPort: 8080

$kubectl describe rs/tomcat

$kubectl get pods

and use the above pod name here

$kubectl get pods tomcat-kg2c8 -o yaml

<https://stackoverflow.com/questions/41325087/what-is-the-difference-between-a-pod-and-a-deployment>

# Labels and Selectors

Labels are key/value pairs that are attached to objects, such as pods, nodes etc.

Labels are used for grouping the objects based on a label name.

 Later on, we or any process can select the group of objects based on labels.

Labels can be attached to objects at creation time and subsequently added and modified at any time.

Each object can have a set of key/value labels defined.

Each Key must be unique for a given object.

Syntax:

"metadata": {

**"labels"**: {

**"key1"** : "value1",

**"key2"** : "value2"

  }

}

Example: a pod has the below label

"metadata": {

**"labels"**: {

**"environment"** : "production",

**"release"** : "Stable"

  }

}

Example: another pod has the below label

"metadata": {

**"labels"**: {

**"environment"** : "dev",

**"release"** : "snapshort"

  }

}

Example: third pod has the below label

"metadata": {

**"labels"**: {

**"environment"** : "dev",

**"release"** : "stable"

  }

}

Labels allow for efficient queries and watches and are ideal for use in UIs and CLIs.

**Label Selectors:**

Label Selectors are used to identify a group of k8s objects based on same label.

The API currently supports two types of selectors:

·         equality-based

·         set-based.

***Equality-based*** **requirement**

*Equality-* or *inequality-based* requirements allow filtering by label keys and values.

Three kinds of operators are admitted =,==,!=.

Assume that we/replication controller are going to select some pods based on the below selection process.

environment = dev

release != snapshort

then as per above example the third pod will be selected

Assume that we/replication controller are going to select some pods based on the below selection process.

release != stable

then as per above example the two pods will be selected

***Set-based*** **requirement**

*Set-based* label requirements allow filtering keys according to a set of values.

 Three kinds of operators are supported: in, notin and exists.

If selector is having:

Example1:

environment in (production, dev)

then three pods will be selected as per the above example.

Example2:

release notin (snapshot, ga)

then two pods will be selected as per the above example.

Other examples:

production

Then only one pod will be selected as per the above example.

!production

Then two pods will be selected as per the above example.

#### **Service and ReplicationController**

The set of pods that a service targets is defined with a label selector. Similarly, the population of pods that a replicationcontroller should manage is also defined with a label selector.

Labels selectors for both objects are defined in json or yaml files using maps, and only equality-based requirement selectors are supported:

"selector": {

**"environment"** : "production",

}

or

selector:

environment: production

this selector (respectively in json or yaml format) is equivalent to environment=production or envronment in (production).

**Resources that support set-based requirements**

Newer resources, such as [Job](https://kubernetes.io/docs/concepts/jobs/run-to-completion-finite-workloads/), [Deployment](https://kubernetes.io/docs/concepts/workloads/controllers/deployment/), [Replica Set](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/), and [Daemon Set](https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/), support set-based requirements as well.

#### matchLabels / matchExpressions

* **matchLabels** is a map of **{key,value}** pairs.
  + A single {key,value} in the matchLabels map is equivalent to an element of matchExpressions, whose key field is “**key**”, the operator is “**In**”, and the values array contains only “value”.
* **matchExpressions** is a list of pod selector requirements.
  + Valid operators include In, NotIn, Exists, and DoesNotExist.
  + The values set must be non-empty in the case of In and NotIn.
* All of the requirements, from both matchLabels and matchExpressions are **AND**ed together – they must all be satisfied in order to match.

*# kubectl explain deployment.spec.selector.*

selector:

matchLabels:

component: redis

matchExpressions:

- {key: tier, operator: In, values: [cache]}

- {key: environment, operator: NotIn, values: [dev]}

### LIST and WATCH filtering

LIST and WATCH operations may specify label selectors to filter the sets of objects returned using a query parameter.

Both requirements are permitted (presented here as they would appear in a URL query string):

* equality-based requirements: ?labelSelector=environment%3Dproduction,tier%3Dfrontend
* set-based requirements: ?labelSelector=environment+in+%28production%2Cqa%29%2Ctier+in+%28frontend%29

Both label selector styles can be used to list or watch resources via a REST client.

For example, targeting apiserver with kubectl and using equality-based one may write:

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

or using set-based requirements:

$kubectl get pods -l 'environment in (production),tier in (frontend)'

As already mentioned set-based requirements are more expressive.  For instance, they can implement the OR operator on values:

kubectl get pods -l 'environment in (production, qa)'

or restricting negative matching via exists operator:

kubectl get pods -l 'environment,environment notin (frontend)'

Below are pending for next session

Deployments

Services

ConfigMap and secrets

Ingress