
Kubernetes
Basic to Advance
Written By: Sagar Choudhary

Introduction

History:

- google develop an internal system called 'Borg' (later named as omega) to deploy and manage thousands of google application and services on their cluster
- in 2014, google introduce k8s as an open source platform written in Golang and later donated to CNCF (cloud native computing foundation)
- Kubernetes playground
- play with k8s
- GKS google
- AKS azure
- EKS amazon

Problem with scaling up the container:

- can't communicate with each other
- autoscaling
- load balancing
- container had to be manage carefully

Terms to know:

- monolithic application: single stone application, every
- Microservice: each task is deploy in diff-2 services, connect with each other via API
- Orchestration tool = container management tool

Kubernetes = k8s

- k8s is an open source container management tool which automates container deployment , container scaling, load balancing
- it schedule, run, manage isolated containers which are running on virtual/physical/cloud machine
- all top cloud provider support k8s

feature of k8s:

- Orchestration (clustering of any no of cluster running on different n/w)
- autoscaling
- load balancing
- platform independent (cloud / virtual / physical)
- fault tolerance (node / pod failure)
- rollback
- health monitoring of pod
- batch execution

Comparation between k8s and docker swarm:

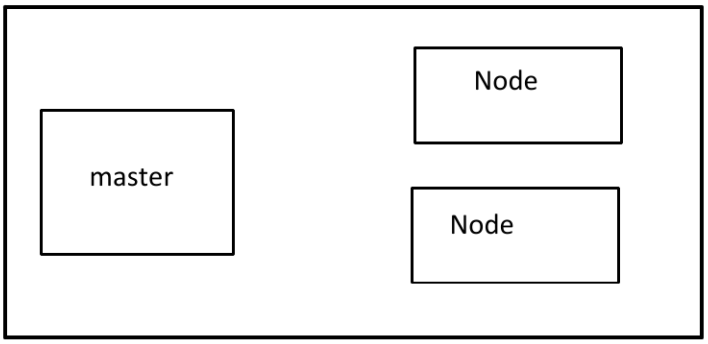
Feature	K8S	Docker Swarm
installation & cluster configuration	Complicated & time consuming	Fast & Easy
Supports	Work with all type of container like Rocket, Docker, ContainerD	Only work with docker
GUI	available	Not Available
Data Volumes	Only shared with containers in same pod	Can be shared with any other container
Update & Rollback	Process schedule to maintain services while updating	Progressive update and Service health monitoring while update
Autoscaling	Available	Not Available
Monitoring	Inbuilt tool	3rd party tool

Note: Apache Marathon is also one more tool in market for container management

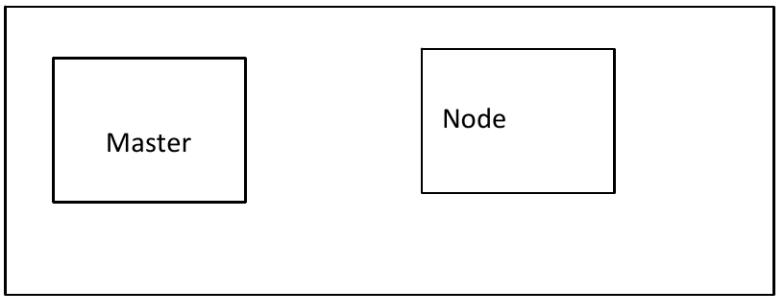
K8s Architecture

There are 3 type of Architecture: very high level

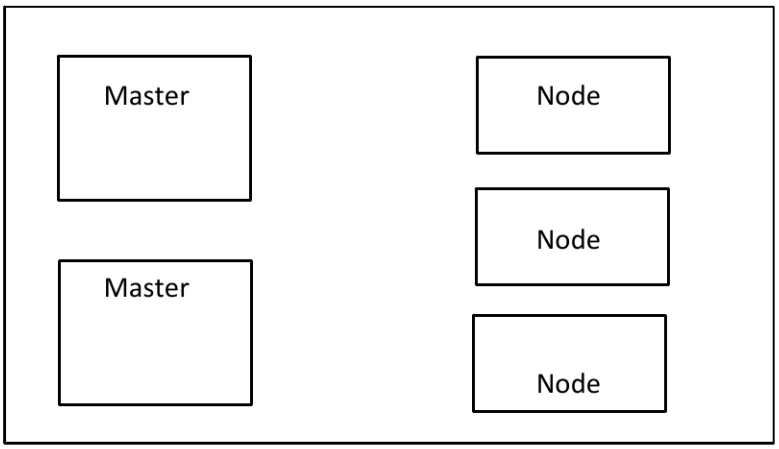
1. **Single master multiple nodes**



2. **Single Master single node**

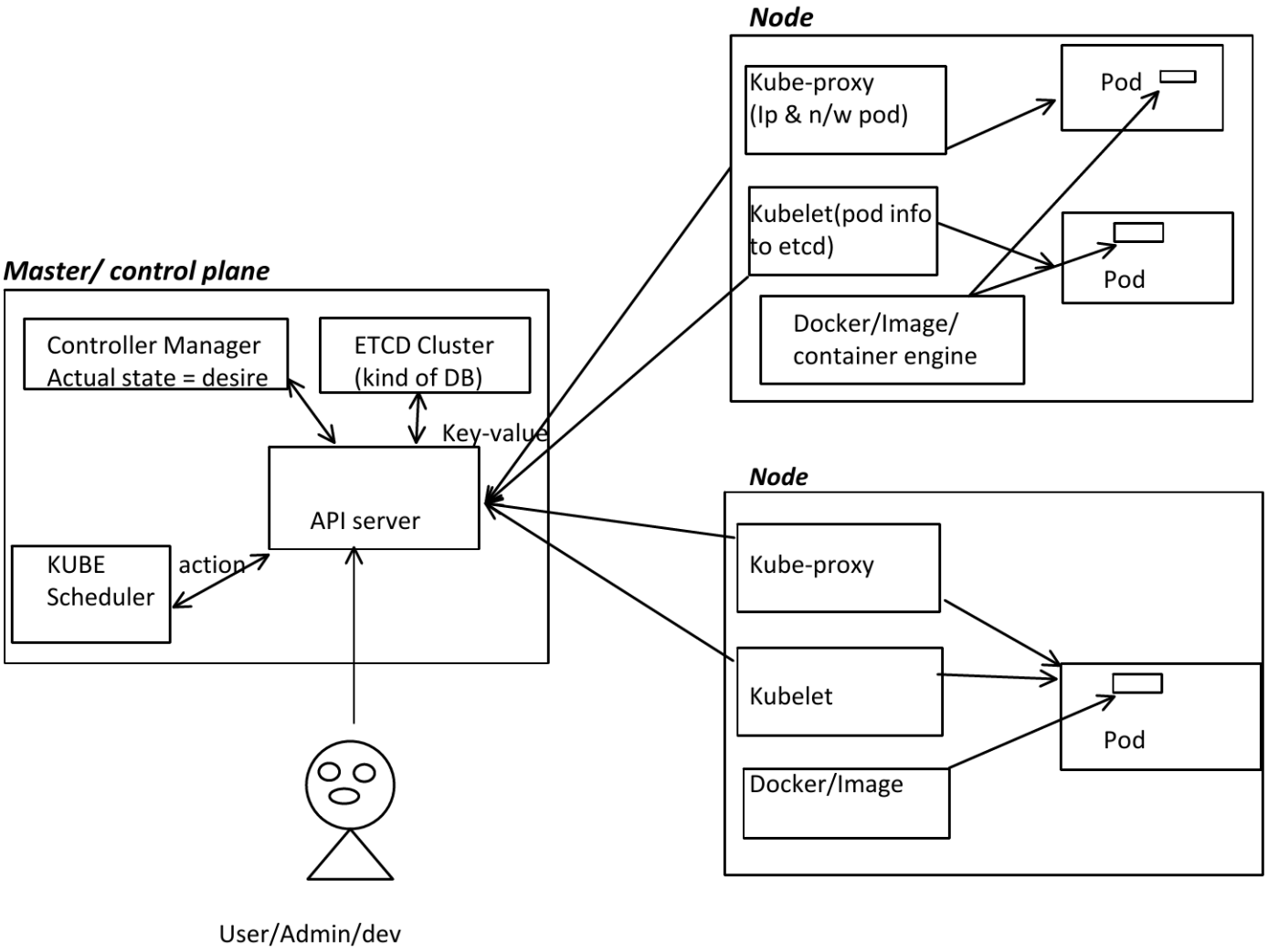


3. **Multiple master multiple nodes**

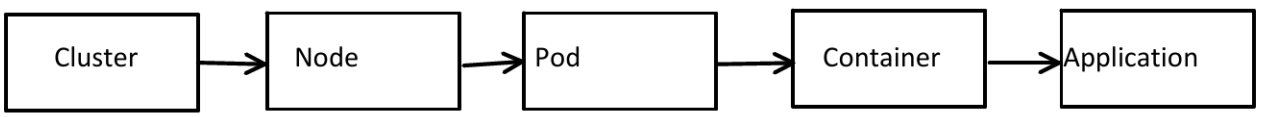


Note: this is very high level architecture of Kubernetes once this is clear then we can understand more deeper in Upcoming chapter

Low level k8s architecture



Request flow High Level Diagram



Working:

1. Create the manifest file for k8s objects (json/yml/yaml)
2. Apply these files to cluster (to master) to bring into desired state
3. Pods runs on node which is controlled by master

Role Of Master:

Kubernetes cluster runs on VM/ BareMetal / cloud or mix

1. K8s having one or more master and one or more workers
2. The master is now going to run set of k8s process . These process with insure smooth functioning of master these process are called control plane
3. Can be multi master for high availability
4. Master runs control plane to run cluster smoothly

Components Of master plane:

1. API Server
2. Kube Scheduler
3. Controller Manager
4. ETCD (not part of k8s but without this k8s won't work so consider this also a part of k8s)

1. **API Server:**
   a. It interact directly with users
   b. It meant to scale automatically according to load or request load
   c. Front end of control plane
2. **ETCD:**
   a. Store metadata or status of cluster
   b. Consistent and high availability
   c. Store data in key value form

   Features:
   a. Fully replicated: entire state is available on every node of cluster
   b. Secure: implements TLS with optional client-certificate authentication 7
   c. Fast: benchmark at 10,000 writes per second
3. **Kube Scheduler:**
   a. When user make a request for creation or management of pods , it is going to take the action on his request
   b. If any mismatch occurs in number of pods runs then it will make them as desired no pods
   c. When we are not explicitly assign the node for pod creation then it will automatically decide the best node and create pod there. But if you want to create pod on specific node then assign node in manifest file.
   d. It's take the hardware configuration information from etcd and that help to decide the best node for pod creation

4. **Controller Manager:**
   a. Make sure actual state equals to desired state
   b. If k8s on cloud then "cloud controller manager"
   c. If k8s on non-cloud "kube-controller manager"

   Controller Components:
   a. Node Controller: for checking of nodes that has detect in cloud after it's stop responding.
   b. Route Controller: Responsible to setting up n/w, route
   c. Service Controller: Responsible for load balancing
   d. Volume Controller: Managing Volumes

Components Of Worker Plane:

1. Kube Proxy
2. Kubelet
3. Pods
4. Container Engine

1. **Kube Proxy:**

    a. It is responsible for networking and responsible to allocate the IP for pods
    b. It's runs on each node
    c. It's communicate to master via the API Server

2. **Kubelet:**

    a. Agent running on node
    b. Listen the k8s master (pod creation request)
    c. Provide pod information to etcd via API Server
    d. use port 10255
    e. Send success / failure status to control plane

3. **Pods:**

    a. smallest unit of k8s
    b. One pod can contains multiple container but recommend only one container in one pod
    c. Pod having it's IP address but container don't have
    d. Cluster has at least one master node and one worker node
    e. K8s cannot start container without pods
    f. Auto scaling and auto healing by default not provided by pod for this high level k8s object required
    g. Pod crashed is also one more limitation but fix this by high level objects

4. **Container Engine:**

    a. Work with kubelet
    b. Pulling image
    c. Start/stop container
    d. Expose port which is specified in manifest

Important Notes:

a. If we are using single cloud then command will use "kubectl"
b. If we are using on premise then command will use "kubeadm"
c. If we are using on hybrid/federated then command will use "kubefed"
d. https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html read this and now easily you can understand

Kubernetes Objects

What Are Kubernetes Objects?

1. Kubernetes objects are represented in JSON or YAML files and describe the state of your cluster.
2. The state of the cluster defines what workloads/containerized application should be running in it.
3. You can create these objects, and then Kubernetes makes sure that your desired state is maintained. For example, if you create a deployment in Kubernetes, it makes sure it's running even if the cluster restarts after a crash. That's the beauty of Kubernetes.
4. Every k8s objects include the 2 nested field that govern the object config the object spec and the object status
5. The Spec which we written in yaml file that describe the desired state or characteristic which we want
6. The Status describe the actual status
7. Objects always identify by the unique ID
8. Its represent in Json / Yaml files

There are two categories of objects in Kubernetes, which we'll discuss more later on:

1. **basic objects:** Pods, Service, Volumes, Namespace, etc., which are independent and don't require other objects
2. **high-level objects (controllers):** Deployments, Replication Controllers, ReplicaSets, StatefulSets, Jobs, etc., which are built on top of the basic objects

All Kubernetes Object list :

1. Pod : A thin wrapper around one or more containers
2. Service: Maps a fixed IP address to a logical group of pods
3. Volume: a directory with data that is accessible across multiple containers in a Pod
4. Namespace: a way to organize clusters into virtual sub-clusters
5. ReplicaSets: Ensures a defined number of pods are always running
6. Replica Controller : Ensures a defined number of pods are always running
7. Secrets: an object that contains a small amount of sensitive data such as a password, a tok1en, or a key
8. Config Maps: an API object that lets you store configuration for other objects to use
9. Deployments : Details how to roll out (or roll back) across versions of your application
10. StatefulSets: the workload API object used to manage stateful applications
11. Jobs: Ensures a pod properly runs to completion and stop after process complete it's execution
12. Daemon Sets: Implements a single instance of a pod on all (or filtered subset of) worker node(s)
13. Label: Key/Value pairs used for association and filtering

K8s objects management

Relationship Between k8s Objects

1. pods maintains container
2. ReplicaSets manage pods
3. Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features
4. Service expose the pod process to the outside world
5. Config Map and Secrets Both are store the data the same way, with key/value pairs, but ConfigMaps are meant for plain text data, and secrets are meant for data that you don't want anything or anyone to know about except the application
6. the replication controller only supports equality-based selectors whereas the replica set supports set-based selectors
7. Replica Set is the next generation of Replication Controller. Replication controller is kind of imperative, but replica sets try to be as declarative as possible.

State Of Objects:

1. Replicas
2. Image
3. Name
4. Port
5. Volume

K8s Objects Management:

The kubectl command line tool supports several different ways to create and manage Kubernetes Objects

Management Technique	Operate On	Recommended Environment
Imperative Command	Live Objects	Dev / QA / lower Env
Declarative Objects	Files (json/yaml)	Production

Minikube Installation & Pod

Steps to Install minikube and run pod

Go to Aws account and create & launch instance
Ubuntu18.04 -> t2.medium (minimum 2 CPU required)

Do SSH and after that run following command

1. sudo su
2. Now install docker
   a. sudo apt update && apt -y install docker.io
3. install Kubectl
   a. curl -LO https://storage.googleapis.com/kubern... -s https://storage.googleapis.com/kubern... && chmod +x ./kubectl && sudo mv ./kubectl /usr/local/bin/kubectl
4. install Minikube
   a. curl -Lo minikube https://storage.googleapis.com/miniku... && chmod +x minikube && sudo mv minikube /usr/local/bin/
5. minikube dashboard

For Window 11 / 10 minikube install

Install Minikube on Windows - ShellHacks

Fundamental Of Kubernetes Object Pod:

1. A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers.
2. When a Pod creation happen then master will automatically decide that on which node it should create until you have not specified the node.
3. Pods remain on node until is not terminated or until node failure not happen , until pod is not deleted, lack of required resource of pod creation
4. If node die then after a timeout period pod will also get delete
5. If a pod get deleted then same pod (id) cannot restart always start a new pod with different unique ID
6. Volume inside pod also die if the pods die
7. Controller can manage pod autoscaling, self-healing etc.

Example OF Pod Yaml File:

```
apiVersion: v1
kind: Pod
metadata:
  name: singlecontainerpod
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
```

Defaults to Always kubectl apply -f pod1.yml

Pod example

Pod Example with annotation:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
 annotations:
    description: this is demo of annotation
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Defaults to Always kubectl apply -f pod1.yml

Pod Example with multiple containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
 - name: container2
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo c2; sleep 5 ; done"]
```

Defaults to Always kubectl apply -f pod1.yml

Pod Example with environment variable

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: container1
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
    env:
    - name: myname
      value: sagar
```

Defaults to Always kubectl apply -f pod1.yml

Useful Commands for Pod

Create a pod from yaml file

Command: Kubectl create -f filename.yaml

Update a pod from yaml file

Command: Kubectl apply -f filename.yaml

Those are two different approaches:

1. **Imperative Management**
   kubectl create is what we call Imperative Management. On this approach you tell the Kubernetes API what you want to create, replace or delete, not how you want your K8s cluster world to look like.

2. **Declarative Management**
   kubectl apply is part of the Declarative Management approach, where changes that you may have applied to a live object (i.e. through scale) are "**maintained**" even if you apply other changes to the object.

Note: In layman's They do different things. If the resource exists, kubectl create will error out and kubectl apply will not error out.

Delete a pod from yaml file

Command: Kubectl delete -f filename.yaml

List all pods in the default namespace

Command: Kubectl get pods

List all pods in the specific namespace

Command: Kubectl get pods-n namespace_name

List all pods in the current namespace, with more details

Command: kubectl get pods -o wide

Get Pods full Information with events:

Command: kubectl describe pods pod_name

Show log Of a running Pod:

```
kubectl logs my-pod                 #dump pod logs (stdout)
kubectl logs -l name=myLabel        # dump pod logs, with label name=myLabel (stdout)
kubectl logs my-pod -c my-container       # dump pod container logs (stdout, multi-container case)
kubectl logs -l name=myLabel -c my-container   # dump pod logs, with label name=myLabel (stdout)

kubectl logs -f my-pod                      # stream pod logs (stdout)
kubectl logs -f my-pod -c my-container          # stream pod container logs (stdout, multi-container case)
kubectl logs -f -l name=myLabel --all-containers    # stream all pods logs with label name=myLabel (stdout)


kubectl logs my-pod --previous       # dump pod logs (stdout) for a previous instantiation of a container
kubectl logs my-pod -c my-container --previous     # dump pod container logs (stdout, multi-container case)
for a previous instantiation of a container
```

Label & Selectors

Labels:

1. Labels are the mechanism you use to organize the Kubernetes objects
2. Labels are key value paired without any predefined meaning that can be attach to objects
3. Labels are similar to tag in AWS and GIT
4. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system
5. Labels can be attached to objects at creation time and subsequently added and modified at any time
6. You are free to choose labels as you need it to refers an environment which is used for dev or Testing or production, refer a product group like Deployment A, Deployment B
7. **Valid label value:**
   - must be 63 characters or less (can be empty),
   - unless empty, must begin and end with an alphanumeric character ([a-z0-9A-Z]),
   - could contain dashes (-), underscores (_), dots (.), and alphanumeric between.
8. Apply label on pod via imperative method
   a. Kubectl label pod{object} object_name env=dev
9. **Example of Declarative :**

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Command:

a. Kubectl get pods{object} --show-labels
b. Kubectl get pods{object} -l env=dev
c. Kubectl get pods{object} -l env!=dev

Note: ***there is 3 way to delete an object***

a. From yaml file
b. Kubectl delete object object_name
c. Kubectl delete object -l env=dev

Selectors:

1. Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).
2. Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.
3. The API currently supports two types of selectors
   a. *equality-based (=, !=)*
   b. Set-based (in, notin, exists)

Command of set-based: Kubectl get pods{object} -l 'env in (dev,test,qa)'

Node Selector

1. You can constrain a Pod so that it can only run on particular set of node(s).
2. There are several ways to do this and the recommended approaches all use label selectors to facilitate the selection.
3. Generally such constraints are unnecessary, as the scheduler will automatically do a reasonable placement (for example, spreading your Pods across nodes so as not place Pods on a node with insufficient free resources).
4. However, there are some circumstances where you may want to control which node the Pod deploys to, for example, to ensure that a Pod ends up on a node with an SSD attached to it, or to co-locate Pods from two different services that communicate a lot into the same availability zone.
5. You can use any of the following methods to choose where Kubernetes schedules specific Pods:
    a. Node-Selector
    b. Affinity and Anti-Affinity
    c. Nodename

NodeSelector:

a. nodeSelector is the simplest recommended form of node selection constraint.
b. You can add the nodeSelector field to your Pod specification and specify the node labels you want the target node to have. Kubernetes only schedules the Pod onto nodes that have each of the labels you specify.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
      nodename: minikubenode
```

Affinity and anti-affinity

nodeSelector is the simplest way to constrain Pods to nodes with specific labels. Affinity and anti-affinity expands the types of constraints you can define. Some of the benefits of affinity and anti-affinity include:

- The affinity/anti-affinity language is more expressive. nodeSelector only selects nodes with all the specified labels. Affinity/anti-affinity gives you more control over the selection logic.
- You can indicate that a rule is *soft* or *preferred*, so that the scheduler still schedules the Pod even if it can't find a matching node.
- You can constrain a Pod using labels on other Pods running on the node (or other topological domain), instead of just node labels, which allows you to define rules for which Pods can be co-located on a node.

The affinity feature consists of two types of affinity:

- *Node affinity* functions like the nodeSelector field but is more expressive and allows you to specify soft rules.
- *Inter-pod affinity/anti-affinity* allows you to constrain Pods against labels on other Pods

There are two types of node affinity:

- *requiredDuringSchedulingIgnoredDuringExecution*: The scheduler can't schedule the Pod unless the rule is met. This functions like nodeSelector, but with a more expressive syntax.
- *preferredDuringSchedulingIgnoredDuringExecution*: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod

Note: want to see the example and want to read more than refer this link.

Scaling & Replication Controller

1. Kubernetes was designed to orchestration multiple container and replication
2. With the help of multiple container we can get following things
    a. Reliability :

       By having multiple version of an application, you prevent problem by one or more fails

    b. Load Balancing:

       By having multiple container of an application, enable you to easily send traffic to diff-2 instance to prevent overloading of a single instance or node

    c. Scaling:

       When load become too much for existing instance then k8s enable you to easily up your application with additional resources. And when load decrease then scale down is also possible.

    d. Rolling Update:

       Update to a service by replacing pods one by one.

Replication controller:

1. If there are too many pods, the Replication Controller terminates the extra pods. If there are too few, the Replication Controller starts more pods
2. A relocation controller is an object that enable you to easily create multiple pods, then make sure that number of pods always exists.
3. if a pod get crashed, terminated then RC will automatically recreate the new pod with similar configuration.
4. RC is recommended if you just want to make sure 1 pod is always running, even after system reboot.
5. You can run the RC with 1 replica & the RC make sure that pod is always running.

Example:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: rcpod
spec:
  replicas: 3
  selector:
    app: rcpod
  template:
    metadata:
      name: rcpod
      labels:
        app: rcpod
    spec:
      containers:
      - name: rcpod
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo c1; sleep 5 ; done"]
```

Interactive method to scaleup and scale down number of pods from RC

Command: *kubectl scale --replicas=8 rc -l app=rcpod*

Replica Sets

ReplicaSets:

1. ReplicaSets is a next generation Replica Controller
2. The replica controller only supports equality based selector but replica sets supports set based selector i.e. filtering according to the set of values.
3. A Replica Set'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

How a ReplicaSet works:

1. In replicasets there is a lot of properties which helps replicasets to work properly.
2. There is selector properties which helps replicasets to recreate the pods based on label when pod failure happen, also helped to group the same label pods.
3. There is one more properties called template that help RS to create pod according to the given pod template
4. Replicas properties is used to provide the number of desired pods to create

When to use a ReplicaSet:

1. A ReplicaSet ensures that a specified number of pod replicas are running at any given time. 2.
2. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.
3. This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section

Example:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Deployments

Deployments:

1. Replication controller & ReplicaSets is not able to do the update and rollback apps in the cluster
2. A deployment is act as a supervisor for pod, giving you fine-grained control over how and when a new pod is Rolled out, updated or rollback to a previous state
3. When using deployment object we first define the state of the app, then k8s cluster schedule maintained app instance onto specific individual nodes
4. A deployment provides declarative updates for pods & replicasets
5. K8s then monitors if the node hosting an instance goes down or pod is deleted then deployment control it's replicas
6. this is provide a self-healing mechanism to address machine failure or maintenance

Use Case :

The following are typical use cases for Deployments:

- Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
- Declare the new state of the Pods by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
- Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
- Scale up the Deployment to facilitate more load.
- Pause the rollout of a Deployment to apply multiple fixes to its PodTemplateSpec and then resume it to start a new rollout.
- Use the status of the Deployment as an indicator that a rollout has stuck.
- Clean up older ReplicaSets that you don't need anymore

Notes:

1. If there are problem in deployment, k8s will automatically rollback to the previous version, however you can explicitly rollback to a specific version, as in our case to revision 1 (the original pod version)
2. You can rollback to a specific version by "--to-revision" option

   i.e. Kubectl rollout undo deploy/deployment_name --to-revision=2
3. The name of replicasets is always formatted as [deployment-name]-[random string]
4. Command: kubectl get deploy

   When you inspect the deployment in your cluster the flowing field display
   a. NAME: list the name of the deployment in the namespace
   b. READY: ready/desired i.e. 2/2
   c. UP-TO-DATE: display the no of replica that have been updated to achieve the desired state
   d. AVAILABLE: Display how many replicas of the application are available to your users
   e. AGE: Display the amount of time that app has been Running

Commands:

1. Create the Deployment by running the following command:
   a. kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
2. Run *kubectl get deployments* to check if the Deployment was created
3. To see the Deployment rollout status, run *kubectl rollout status deployment/ubuntu-deployment*
4. To see the ReplicaSet (rs) created by the Deployment, run *kubectl get rs*

Deployment Example & Commands

Failed Deployment:

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

- Insufficient quota
- Readiness probe failures
- Image pull errors
- Insufficient permissions
- Limit ranges
- Application runtime misconfiguration

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ubuntu-deployment
  labels:
    app: ubuntu
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ubuntu
  template:
    metadata:
      labels:
        app: ubuntu
    spec:
      containers:
      - name: ubuntu
        image: ubuntu:20.04
        command:  ["/bin/bash", "-c", "while true; do echo 'hello sagar'; sleep 5; done"]
```

Updating a Deployment

Follow the steps given below to update your Deployment:

1. Let's update the nginx Pods to use the nginx:1.16.1 image instead of the nginx:1.14.2 image
   kubectl set image deployment.v1.apps/ubuntu-deployment ubuntu=ubuntu:16.04
   Also change the echo statement in command

2. See the rollout status, run: *kubectl rollout status deployment/ubuntu-deployment*
3. Run *kubectl get rs* to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 3 replicas, as well as scaling down the old ReplicaSet to 0 replicas

Checking Rollout History of a Deployment

Follow the steps given below to check the rollout history:

1. First, check the revisions of this Deployment:
   kubectl rollout history deployment/ubuntu-deployment

2. To see the details of each revision, run:
   kubectl rollout history deployment/ubuntu-deployment --revision=2

Delpoyment Commands

Rolling Back to a Previous Revision

Follow the steps given below to rollback the Deployment from the current version to the previous version, which is version 2.

1. Now you've decided to undo the current rollout and rollback to the previous revision:
   kubectl rollout undo deployment/ubuntu-deployment

   Alternatively, you can rollback to a specific revision by specifying it with --to-revision:
   kubectl rollout undo deployment/ubuntu-deployment --to-revision=2
2. Check if the rollback was successful and the Deployment is running as expected, run:
   kubectl get deployment ubuntu-deployment
3. Get the description of the Deployment:
   kubectl describe deployment ubuntu-deployment

Scaling a Deployment

1. You can scale a Deployment by using the following command:
   kubectl scale deployment/ubuntu-deployment --replicas=10
2. Assuming horizontal Pod autoscaling is enabled in your cluster, you can setup an autoscaler for your Deployment and choose the minimum and maximum number of Pods you want to run based on the CPU utilization of your existing Pods.
   kubectl autoscale deployment/ubuntu-deployment --min=10 --max=15 --cpu-percent=80
3. ***Proportional scaling***
   RollingUpdate Deployments support running multiple versions of an application at the same time. When you or an autoscaler scales a RollingUpdate Deployment that is in the middle of a rollout (either in progress or paused), the Deployment controller balances the additional replicas in the existing active ReplicaSets (ReplicaSets with Pods) in order to mitigate risk. This is called *proportional scaling*.

Pausing and Resuming a rollout of a Deployment

When you update a Deployment, or plan to, you can pause rollouts for that Deployment before you trigger one or more updates. When you're ready to apply those changes, you resume rollouts for the Deployment. This approach allows you to apply multiple fixes in between pausing and resuming without triggering unnecessary rollouts

1. Get the Deployment details:
   kubectl get deploy
2. Pause by running the following command:
   kubectl rollout pause deployment/ubuntu-deployment
3. Then update the image of the Deployment:
   kubectl set image deployment.v1.apps/ubuntu-deployment ubuntu=ubuntu:16.04
4. Notice that no new rollout started:
   kubectl rollout history deployment/ubuntu-deployment
5. Get the rollout status to verify that the existing ReplicaSet has not changed: kubectl get rs
6. You can make as many updates as you wish, for example, update the resources that will be used:
   kubectl set resources deployment/ubuntu-deployment -c=ubuntu--limits=cpu=200m,memory=512Mi
7. Eventually, resume the Deployment rollout and observe a new ReplicaSet coming up with all the new updates:
   kubectl rollout resume deployment/ubuntu-deployment
8. Watch the status of the rollout until it's done.
   kubectl get rs -w

StatefulSet:

1. StatefulSet is the workload API object used to manage stateful applications.
2. Manages the deployment and scaling of a set of Pods, *and provides guarantees about the ordering and uniqueness* of these Pods
3. Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a *sticky identity* for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

Example:

1. Let's just start with a simple example and we are tasked to deploy a database server. So we install and setup MySQL on the server and create a database. Our database is now operational. Other applications can now write data to our database.
2. To withstand failures we are tasked to deploy a High Availability solution. So we deploy additional servers and install MySQL on those as well. We have a blank database on the new servers.
3. So how do we replicate the data from the original database to the new databases on the new servers. Before we get into that,
4. So back to our question on how do we replicate the database to the databases in the new server.
5. There are different topologies available.
   The most straight forward one being a ***single master multi slave topology***, where all writes come in to the master server and reads can be served by either the master or any of the slaves servers.
6. So the master server should be setup first, before deploying the slaves. Once the slaves are deployed, perform an initial clone of the database from the master server to the first slave.
7. After the initial copy enable continuous replication from the master to that slave so that the database on the slave node is always in sync with the database on the master.
8. Note that both these slaves are configured with the address of the master host. When replication is initialized you point the slave to the master with the master's hostname or address.
9. That way the slaves know where the master is.

Kubernetes deployment problem with master slave:

1. Let us now go back to the world of Kubernetes and containers and try to deploy this setup.
2. In the Kubernetes world, each of these instances, including the master and slaves are a POD part of a deployment.
3. In step 1, we want the master to come up first and then the slaves. And in case of the slaves we want slave 1 to come up first,
4. perform initial clone of data from the master, and we want slave 2 to come up next and clone data from slave 1.
5. With *deployments you can't guarantee that order.*
6. With deployments all pods part of the deployment come up at the same time.
7. So the first step can't be implemented with a Deployment.
8. As we have seen while working with deployments the *pods come up with random names.*
9. So that won't help us here. Even if we decide to designate one of these pods as the master,
10. and use it's name to configure the slaves,
11. if that *POD crashes* and the deployment creates a new pod in it's place, it's going to come up with a *completely new pod name*.
12. And now the slaves are pointing to an address that does not exist.
13. And because of all of these, the remaining steps can't be executed.

StatefulSets

StatefulSets fix the master slave architecture Problem:

1. that's where stateful sets come into play. Stateful sets are similar to deployment sets, as in they create PODs based on a template. But with some differences.
2. With stateful sets, *pods are created in a sequential order*.
3. After the first pod is deployed, it must be in a running and ready state before the next pod is deployed.
4. So that helps us ensure master is deployed first and then slave 1 and then slave 2.
5. Stateful sets assign a unique ordinal index to each POD – a number starting from 0 for the first pod and increments by 1.
6. Each Pod gets a name derived from this index, combined with the stateful set name.
7. So the first pod gets mysql-0, the second pod mysql-1 and third mysql-2.
8. SO *no more random names*. You can rely on these names going forward.
9. We can designate the pod with the name mysql-0 as the master, and any other pods as slaves.
10. Pod mysql-2 knows that it has to perform an initial clone of data from the pod mysql-1. If you scale up by deploying another pod, mysql-3, then it would know that it can perform a clone from mysql-2.
11. To enable continuous replication, you can now point the slaves to the master at mysql-0. Even if the master fails, and the pod is recreated is created, it would still *come up with the same name*.
12. Stateful sets maintain a sticky identity for each of their pods.
13. The master is now always the master and available at the address mysql-0.
14. And that is ***why you need stateful sets***

Limitations

- The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested storage class, or pre-provisioned by an admin.
- Deleting and/or scaling a StatefulSet down will *not* delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.
- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. *You are responsible for creating this Service*.
- StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.
- When using Rolling Updates with the default Pod Management Policy (OrderedReady), it's possible to get into a broken state that requires manual intervention to repair.

Stable Storage

1. For each VolumeClaimTemplate entry defined in a StatefulSet, each Pod receives one PersistentVolumeClaim. In the nginx example above, each Pod receives a single PersistentVolume with a StorageClass of my-storage-class and 1 Gib of provisioned storage.
2. If no StorageClass is specified, then the default StorageClass will be used. When a Pod is (re)scheduled onto a node, its volumeMounts mount the PersistentVolumes associated with its PersistentVolume Claims.
3. Note that, the PersistentVolumes associated with the Pods' PersistentVolume Claims are not deleted when the Pods, or StatefulSet are deleted. This must be done manually.

StatefulSets Example

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  minReadySeconds: 10 # by default is 0
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:k
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

Kubernetes Network Model

The Kubernetes network model:

1. Every Pod in a cluster gets its own unique cluster-wide IP address.
2. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports.
3. This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.
4. Kubernetes imposes the following fundamental requirements on any networking implementation
    - pods can communicate with all other pods on any other node without NAT
    - agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note:

1. For those platforms that support Pods running in the host network (e.g. Linux), when pods are attached to the host network of a node they can still communicate with all pods on all nodes without NAT.
2. This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers.
3. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project.

This is the same basic model.

1. Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address and MAC address.
2. This means that containers within a Pod can all reach each other's ports on *localhost*.
3. This also means that containers within a Pod must coordinate port usage, but this is no different from processes in a VM. ***This is called the "IP-per-pod" model.***
4. How this is implemented is a detail of the particular container runtime in use.
5. It is possible to request ports on the Node itself which forward to your Pod (called host ports), but this is a very niche operation.
6. How that forwarding is implemented is also a detail of the container runtime.
7. The Pod itself is blind to the existence or non-existence of host ports.

Kubernetes networking addresses four concerns:

- Containers within a Pod use networking to communicate via loopback.
- Cluster networking provides communication between different Pods.
- The Service resource lets you expose an application running in Pods to be reachable from outside your cluster.
- You can also use Services to publish services only for consumption inside your cluster

Networking

Cluster Networking:

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. Highly-coupled container-to-container communications: this is solved by Pods and localhost communications.
2. Pod-to-Pod communications.
3. Pod-to-Service communications.
4. External-to-Service communications.

1. ***Container to container communication inside pod Example:***

```
kind: Pod
apiVersion: v1
metadata:
  name: testpod
spec:
  containers:
    - name: c00
      image: ubuntu
      command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5 ;
done"]
    - name: c01
      image: httpd
      ports:
        - containerPort: 80
```

Commands:

a. Kubectl apply -f filename.yaml
b. kubectl logs -f testpod -c c00
c. kubectl exec -it testpod -c c00 /bin/bash
d. Apt update && apt install curl
e. Curl localhost:80
f. Now it will show the output of application which is running inside 2nd containers

2. ***Pod to Pod Communication within same node Example:***

```
kind: Pod
apiVersion: v1
metadata:
 name: testpod1
spec:
 containers:
  - name: c00
    image: ubuntu
    command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5 ; done"]
---
kind: Pod
apiVersion: v1
metadata:
 name: testpod2
spec:
 containers:
  - name: c01
    image: httpd
    ports:
     - containerPort: 80
```

a. Pod to communication will happen via the Ips.
b. By default pod Ip will not accessible outside the node.

Networking & Service

Commands:

1. Kubectl apply -f filename.yaml
2. Kubectl get all
3. kubectl exec -it pod/testpod1 -- /bin/bash
4. apt install && apt install curl
5. Ping IPaddressOFPod2:80

Issue:

Each pod gets its own IP address and when we deploy the pod via deployment or ReplicaSets then it creates the pod with a unique IP address and we provide that IP to other users/pods/applications/frontend to use. And now let's suppose due to some issue pods get terminated and now RS will again create the pod with a new IP address then again we need to update all users/pods/applications/frontend and if it happens a lot of time so it's hard to remember the IP address of pods or hard to update at runtime. *so solution of this issue is service object.*

In other words:

Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are nonpermanent resources. If you use a Deployment to run your app, it can create and destroy Pods dynamically.

Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.

This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

Service:

1. When using RC,RS,deployement, pods are terminated and created during scaling or replication operations
2. When using deployment , while updating the image version the pods are terminated and new pods take the place of other pods.
3. Pods are very dynamic i.e. they come & go on the k8s cluster and on any of the available nodes & it would be difficult to access the pods as the pods IP changes its recreated.
4. Service objects is an logical bridge between pods and end user, which provide virtual IP
5. Service allow client to reliably connect to the the containers running in the pod using the VIP
6. The VIP is not a actual IP connected to a network interface, but its purpose is purely forward traffic to one or more pods
7. kube proxy is the one which keeps the mapping between the VIP and pods up to date, which queries the API server to learn about new services in the cluster
8. Although each pod having unique IP address, but those are not accessible outside the cluster.
9. Services help to expose the VIP mapped to the pod & allow application to receive traffic
10. Label are used to select which are the pods to be put under the service
11. Creating a service will create an endpoint to access the pods/application in it.
12. Service can be expose in 3 different way be specifying a type in the service spec
    a. Cluster IP
    b. Node Port
    c. Load Balancer
13. By default service can run only between ports 30,000 - 32767.
14. The set of pods targeted by a service usually determined by a selector

ClusterIP

ClusterIP:

1. Expose virtual IP reachable from within the cluster
2. Mainly used to communicate between components of microservices
3. Also used to communicate between pods on different-2 nodes & default service type.

Example:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
     containers:
      - name: c00
        image: httpd
        ports:
        - containerPort: 80
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: ubuntudeployment
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: ubuntudeployment
  template:
    metadata:
      name: testpod2
      labels:
        name: ubuntudeployment
    spec:
     containers:
      - name: c01
        image: ubuntu
        command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5 ; done"]
---
kind: Service                   # Defines to create Service type Object
apiVersion: v1
metadata:
 name: demoservice
spec:
 ports:
  - port: 80                    # Containers port exposed
    targetPort: 80              # Pods port
 selector:
  name: httpddeployment             # Apply this service to any pods which has the specific label
 type: ClusterIP                # Specifies the service type i.e ClusterIP or NodePort
```

ClusterIP commands & NodePort

Commands:

1. kubectl apply -f filename.yaml
2. To see all resource is running or not: kubectl get all
3. kubectl get pod -o wide
4. Copy the httpddeployment pod IP i.e. 172.17.0.7
5. Now go inside the ubuntu pod
6. And run apt update && apt install curl -y
7. Run "curl 172.17.0.7:80" and you will get output.
8. Run "curl clusterIP:80 i.e. curl 10.104.84.124:80 "
9. Now you hit via the pod IP and it's working
10. But if someone delete the pod or due to any reason the pod terminated
11. Then pod will get new IP and if you hit the command again with old pod IP then it will not give any output
12. Now we copy the Cluster Ip and again go inside the ubuntu pod
13. kubectl exec -it pod/ubuntudeployment-594f56844c-4w6sk -- /bin/bash
14. Now run "curl clusterIP:80 i.e curl 10.104.84.124:80 "
15. It will work same
16. So now we not need to worry about POD IP.

NodePort:

1. Make your service available outside your cluster
2. Expose the service on the same port of each selected node in the cluster using NAT.
3. This is top level of service of clusterIP
4. Here only port we need to know and instead of IP we will use Public Ip of our host.
5. If you set the type field to NodePort, the Kubernetes control plane allocates a port from a range specified by --service-node-port-range flag (default: 30000-32767).
6. Each node proxies that port (the same port number on every Node) into your Service. Your Service reports the allocated port in its .spec.ports[*].nodePort field
7. When you only pass .spec.type to NodePort then it will take any random unique port from the default range
8. But If you want to specify particular IP(s) to proxy the port, you can set the --nodeport-addresses flag for kube-proxy or the equivalent nodePortAddresses field of the kube-proxy configuration file to particular IP block(s).

Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    # By default and for convenience, the `targetPort` is set to the same value as the `port` field.
    - port: 80
      targetPort: 80
      # Optional field
      # By default and for convenience, the Kubernetes control plane will allocate a port from a range (default: 30000-32767)
      nodePort: 30007
```

NodePort Example & command

Example:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: httpddeployment
  template:
   metadata:
     name: testpod1
     labels:
       name: httpddeployment
    spec:
     containers:
      - name: c00
        image: httpd
        ports:
        - containerPort: 80
---
kind: Service                   # Defines to create Service type Object
apiVersion: v1
metadata:
 name: demoservice
spec:
 ports:
  - port: 80                    # Containers port exposed
    targetPort: 80              # Pods port
 selector:
  name: httpddeployment                 # Apply this service to any pods which has the specific label
 type: NodePort                 # Specifies the service type i.e ClusterIP or NodePort
```

Commands:

1. kubectl apply -f filename.yaml
2. Kubectl get all
3. To check all resource is up and running
4. Sometime your pod will show imagepullerror then check image path and path is correct then delete your pod and deployment recreate it automatically
5. For delete pod use the following command:

   kubectl delete pod/pod_name
6. And now deployment will recreate
7. Now check the minikube public service URL of nodeport to access our pod service from outside the cluster for that use the below command
8. minikube service list
9. Now get the URL and hit from browser and you get the desired output

Load Balancer Service

LoadBlancer:

1. Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
2. This service example we can't run on minikube you can test it on any cloud provider K8s service or instance.
3. On cloud providers which support external load balancers, setting the type field to loadBlancer provisions a load balancer for your Service.
4. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer is published in the Service's .status.loadBalancer field.

For example

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
     containers:
      - name: c00
        image: httpd
        ports:
        - containerPort: 80
---
kind: Service                   # Defines to create Service type Object
apiVersion: v1
metadata:
 name: demoservice
spec:
 ports:
   - port: 80                   # Containers port exposed
     targetPort: 80             # Pods port
 selector:
   name: httpddeployment
   type: LoadBlancer
```

Commands:

1. kubectl apply -f filename.yaml
2. Kubectl get all
3. Kubectl get svc
4. Copy the loadBlancer Ip with port
5. Run on browser and you can able to get the pod which is running inside the cluster

Headless Service:

1. When there is no need of load balancing or single-service IP addresses.
2. We create a headless service which is used for creating a service grouping. That does not allocate an IP address or forward traffic.
3. So you can do this by explicitly setting ClusterIP to “None” in the mainfest file, which means no cluster IP is allocated.

For example:

1. if you host MongoDB on a single pod. And you need a service definition on top of it for taking care of the pod restart And also for acquiring a new IP address.
2. But you don’t want any load balancing or routing. You just need the service to patch the request to the back-end pod.
3. So then you use Headless Service since it does not have an IP.
4. Kubernetes allows clients to discover pod IPs through DNS lookups.
5. Usually, when you perform a DNS lookup for a service, the DNS server returns a single IP which is the service’s cluster IP.
6. But if you don’t need the cluster IP for your service, you can set ClusterIP to None , then the DNS server will return the individual pod IPs instead of the service IP.
7. Then client can connect to any of them.

Command:

Note: example file you will get on next page and copy that file and run.

1. Kubectl apply -f filename.yaml
2. Kubectl get all
3. Now you can see there is there is 1 service running
4. Service is ClusterIP type but without IP and here this also called headless service.
5. Now go inside the ubuntu pod by using the below command

   kubectl exec -it pod/pod_name -- /bin/bash
6. Now install the curl and nslookup to test the headless service benefits by using the below command

   apt update&& apt install curl -y && apt install dnsutils -y
7. Now run the command " nslookup headlessservice "
8. And you get the dns and then run the below command and you get the desired result

   curl headlessservice.default.svc.cluster.local:80

Conclusion:

1. With a Headless Service, clients can connect to its pods by connecting to the service’s DNS name.
2. But using headless services, DNS returns the pod’s IPs and client can connect directly to the pods instead via the service proxy

Headless Service Example

Example:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
     containers:
       - name: c00
         image: httpd
         ports:
         - containerPort: 80
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: ubuntudeployment
spec:
  replicas: 1
  selector:     # tells the controller which pods to watch/belong to
   matchLabels:
    name: ubuntudeployment
  template:
    metadata:
      name: testpod2
      labels:
        name: ubuntudeployment
    spec:
     containers:
       - name: c01
         image: ubuntu
         command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep 5 ; done"]
---
kind: Service                   # Defines to create Service type Object
apiVersion: v1
metadata:
 name: headlessservice
spec:
 clusterIP: None
 ports:
  - port: 80                    # Containers port exposed
    targetPort: 80              # Pods port
 selector:
   name: httpddeployment              # Apply this service to any pods which has the specific label
```

Topology-aware traffic routing with topology keys

Service Topology:

1. Service Topology enables a service to route traffic based upon the Node topology of the cluster.
2. For example, a service can specify that traffic be preferentially routed to endpoints that are on the same Node as the client, or in the same availability zone.
3. By default, traffic sent to a ClusterIP or NodePort Service may be routed to any backend address for the Service or in other words by default traffic can goes on any node.
4. But Kubernetes 1.7 made it possible to route "external" traffic to the Pods running on the same Node that received the traffic or same region or zone.
5. If your cluster has the Service Topology feature gate enabled, you can control Service traffic routing by specifying the topologyKeys field on the Service spec.
6. If topologyKeys is not specified or empty, no topology constraints will be applied.
7. Consider a cluster with Nodes that are labeled with their hostname, zone name, and region name. Then you can set the topologyKeys values of a service to direct traffic as follows.

Examples

The following are common examples of using the Service Topology feature.

1. ***Only Node Local Endpoints***
   A Service that only routes to node local endpoints. If no endpoints exist on the node, traffic is dropped:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
```

2. ***Prefer Node Local Endpoints***
   A Service that prefers node local Endpoints but falls back to cluster wide endpoints if node local endpoints do not exist:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
```

3. ***Only Zonal or Regional Endpoints***
   A Service that prefers zonal then regional endpoints. If no endpoints exist in either, traffic is dropped.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
```

4. ***Prefer Node Local, Zonal, then Regional Endpoints***
   A Service that prefers node local, zonal, then regional endpoints but falls back to cluster wide endpoints.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  topologyKeys:
    - "kubernetes.io/hostname"
    - "topology.kubernetes.io/zone"
    - "topology.kubernetes.io/region"
    - "*"
```

Constraints

- Service topology is not compatible with externalTrafficPolicy=Local, and therefore a Service cannot use both of these features. It is possible to use both features in the same cluster on different Services, only not on the same Service.
- Valid topology keys are currently limited to kubernetes.io/hostname, topology.kubernetes.io/zone, and topology.kubernetes.io/region, but will be generalized to other node labels in the future.
- Topology keys must be valid label keys and at most 16 keys may be specified.
- The catch-all value, "*", must be the last value in the topology keys, if it is used

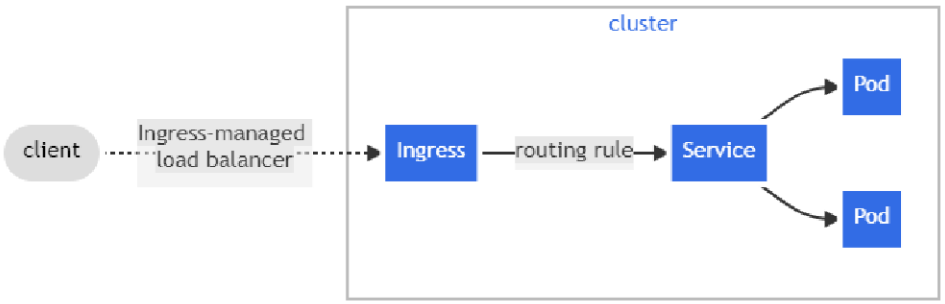
DNS for Services and Pods:

1. Kubernetes creates DNS records for Services and Pods. You can contact Services with consistent DNS names instead of IP addresses.
2. Every Service defined in the cluster is assigned a DNS name.
3. By default, a client Pod's DNS search list includes the Pod's own namespace and the cluster's default domain
4. If are pod on test namespace and service in prod namespace then without specifying the namespace it not possible that we can find service from the test namespace.
5. For more details about dns refer this link

Ingress

Introduction:

1. An API object that manages external access to the services in a cluster
2. typically it works on http request
3. Ingress may provide load balancing, SSL termination and name-based virtual hosting
4. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.
5. Traffic routing is controlled by rules defined on the Ingress resource.



6. An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting
7. usually ingress provide a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic
8. An Ingress does not expose arbitrary ports or protocols.
9. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer
10. Ingress mostly apply on service so we can say that it's high level object of service.

Volumes

Extra topic

Tuesday, July 19, 2022 10:57 AM

Taint & toleration

https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

K8s CNI:

https://www.tigera.io/learn/guides/kubernetes-networking/kubernetes-cni/