Chapter3

Kubernetes in practice

2019-08-26

This chapter introduces some of the fundamental objects and features of kubernetes.

# Labels

Imagine you have a POD that’s need to be host on a machine with certain specifications ( SSD HD, physical location , processing power , ..,etc ) OR imagine you want to search or group your PODs for easier administration what would you do ? then label is your way to go, in Kubernetes Label is a Key/value pairs attached to an object let’s see how can we use label to make a POD is lunched on a certain machine

**Note**

* In kubernetes, any objects can be identified using a label.
* You can assign multiple labels per object but avoid using too much label or too little, too much would get you confused and too little won’t give the real benefits of grouping, selecting and searching.
* Best practice is to assign labels to indicate
  + application/program ID use this POD
  + owner (who manage this POD/application)
  + stage (the POD/application in development/testing/ production as well version)
  + resource requirements (SSD, CPU, storage)
  + location (preferred location/zone/ Datacenter to run this POD/application)

Let’s assign label (stage: testing) & (zone: production) to two nodes respectively then try to lunch a POD in a node which has the label (stage: testing)

kubectl get nodes --show-labels  
  
NAME STATUS ROLES AGE VERSION LABELS  
cent222 Ready <none> 2h v1.9.2 <none>  
cent111 NotReady <none> 2h v1.9.2 <none>  
cent333 Ready <none> 2h v1.9.2 <none>  
  
  
kubectl label nodes cent333 stage=testing  
kubectl label nodes cent222 stage=production  
  
kubectl get nodes --show-labels  
  
NAME STATUS ROLES AGE VERSION LABELS  
cent222 Ready <none> 2h v1.9.2 stage=production  
cent111 NotReady <none> 2h v1.9.2 <none>  
cent333 Ready <none> 2h v1.9.2 stage=testing

now let’s lunch a basic Nginx POD tagged with stage=testing in the nodeSelector and confirm it will land on a node tagged with stage=testing. kube-scheduler uses labels mentioned in the nodeSelector section of the pod yaml to select the node to launch the pod.

**Note**

kube-scheduler picks the node based on various factors like individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

[root@cent111]# cat > web-server.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
 name: nginx  
 labels:  
 app: webserver  
spec:  
 containers:  
 - name: nginx  
 image: nginx  
 nodeSelector:  
 stage: testing  
  
[root@cent111]# kubectl create -f web-server.yaml  
pod "wordpress" created  
  
[root@cent111]# kubectl get pods --output=wide  
NAME READY STATUS RESTARTS AGE IP NODE  
wordpress 1/1 Running 0 48s 10.47.255.238 cent333

**Note**

You can assign POD to certain node without label by adding the argument nodeName: nodeX under spec in the YAML file where nodeX is the name of the node

# NameSpace

## what is NameSpace

As in many other platforms, normally there is more than one users (or teams) working on a kubernetes cluster. suppose a pod named 'webserver1' has been built by 'dev' department, when 'sales' department attempts to launch a pod with the same name, the system will give an error:

Error from server (AlreadyExists): error when creating "webserver1.yaml": pods "webserver1" already exists

Kubernetes won’t allow the same object name for the kubernetes resources appear more than once in the same scope.

'Namespaces' (or 'NS' for short) provides the scopes for the kubernetes resources like project/tenant in openstack. Names of resources need to be unique within a namespace, but not across namespaces. it is a nature way to divide cluster resources between multiple users.

Kubernetes starts with three initial namespaces:

* default: The default namespace for objects with no other namespace.
* kube-system: The namespace for objects created by the Kubernetes system.
* kube-public: initially created by kubeadm tool when deploying a cluster. by convention the purpose of this NS is to make some resources readable by all users without authentication. it exists mostly in kubernetes clusters bootstapped with kubeadm tool only.

## create a NS

creating a NS is pretty simple. just kubectl command does the magic. you dont need to have a yaml file.

root@test3:~# kubectl create ns dev  
namespace/dev created

new namespace dev is now created

root@test3:~# kubectl get ns  
NAME STATUS AGE  
default Active 15d  
dev Active 5s #<-----

now webserver1 pod in dev NS won’t conflict with webserver1 pod in sales NS.

$ kubectl get pod --all-namespaces -o wide  
NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
dev webserver1 1/1 Running 4 2d4h 10.47.255.249 cent222 <none>  
sales webserver1 1/1 Running 4 2d4h 10.47.255.244 cent222 <none>

## quota

similiar to openstack 'tenant', you can now apply constraints that limits resource consumption per namespace. for example, you can limit the quantity of objects that can be created in a namespace, total amount of compute resources that may be consumed by resources, etc. the constraint in k8s is called 'quota'. here is an example:

kubectl -n dev create quota quota-onepod --hard pods=1

we just created a quota 'quota-onepod', and the constraint we gave is 'pods=1' - only one pod is allowed to be created in this NS.

$ kubectl get quota -n dev  
NAME CREATED AT  
quota-onepod 2019-06-14T04:25:37Z  
  
$ kubectl get quota -o yaml  
apiVersion: v1  
items:  
- apiVersion: v1  
 kind: ResourceQuota  
 metadata:  
 creationTimestamp: 2019-06-14T04:25:37Z  
 name: foobar  
 namespace: quota-onepod  
 resourceVersion: "823606"  
 selfLink: /api/v1/namespaces/dev/resourcequotas/quota-onepod  
 uid: 76052368-8e5c-11e9-87fb-0050569e6cfc  
 spec:  
 hard:  
 pods: "1"  
 status:  
 hard:  
 pods: "1"  
 used:  
 pods: "1"  
kind: List  
metadata:  
 resourceVersion: ""  
 selfLink: ""

now create a pod in it:

$ kubectl create -f pod-nginx.yaml -n dev  
pod/nginx created

it works fine. now create a second pod in it:

$ kubectl create -f pod-2containers.yaml -n dev  
Error from server (Forbidden): error when creating "pod/pod-2containers.yaml": pods "pod-1" is forbidden: exceeded quota: quota-onepod, requested: pods=1, used: pods=1, limited: pods=1

immediately we run into an error saying "exceeded quota".

this new pod will be created after the quota is removed:

$ kubectl delete quota quota-onepod -n dev  
resourcequota "quota-onepod" deleted  
$ kubectl create -f pod/pod-2containers.yaml -n dev  
pod/pod-1 created

# Replication Controller

you have learned how to launch a pod that representing your containers from its yaml file in chapter 2. one question will rise in your mind: what if we need 3 exactly the same pods (each runs a apache container) to make sure the web service appears more robust? shall we change the name in yaml file then repeat the same commands to create required pods? or maybe with a shell script? kubernetes already has the objects to address this exact demand and the right answer are RC - replicationController or RS - ReplicaSet

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.

## create RC

let’s look at how it works with an example. first create a yaml file for a RC object named myweb.

#myweb-rc.yaml  
apiVersion: v1  
kind: ReplicationController  
metadata:  
 name: myweb  
spec:  
 replicas: 3  
 selector:  
 app: myweb  
 template:  
 metadata:  
 labels:  
 app: myweb  
 spec:  
 containers:  
 - name: myweb  
 image: kubeguide/tomcat-app:v1  
 ports:  
 - containerPort: 8080

again, kind indicates the object type that this yaml file is to define, here it is a RC instead of a pod. in metadata it is showing the RC’s name as myweb. in spec is the detail specification of this RC object. replicas 3 indicates a same pod will be cloned to make sure the total number of pods created by the RC is always 3. template gives information about the containers that will run in the pod, same as what you saw in a pod yaml file.

now use this yaml file to create the RC object:

kubectl create -f myweb-rc.yaml  
replicationcontroller "myweb" created

if you are quick enough, you may capture the intermediate status when the new pods are being created:

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5ggv6 1/1 Running 0 9s  
myweb-lbj89 0/1 ContainerCreating 0 9s  
myweb-m6nrx 0/1 ContainerCreating 0 9s

eventually you will see 3 pods launched:

$ kubectl get rc  
NAME DESIRED CURRENT READY AGE  
myweb 3 3 3 3m29s

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5ggv6 1/1 Running 0 21m  
myweb-lbj89 1/1 Running 0 21m  
myweb-m6nrx 1/1 Running 0 21m

RC works with pod directly. the workflows are shown in this diagram:

|=> pod  
 |  
RC =============>========|=> pod  
 |  
 |=> pod

with replicas parameter specified in RC object yaml file, the kubernetes replication controller, running as part of kube-controller-manager process in the master node, will keep monitoring the number of running pods spawned by the RC, and automatically launch new ones should any of them runs into failures. the key to learn is, individual pod may die any time, but the "pool" as a whole is always up and running, making a robust service. you will understand this better when you learn kubernetes service.

## test RC

you can test rc’s impact by deleting one of the pod. to delete a resource with kubectl, use kubectl delete sub-command:

$ kubectl delete pod myweb-5ggv6  
pod "myweb-5ggv6" deleted

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5ggv6 0/1 Terminating 0 22m #<---  
myweb-5v9w6 1/1 Running 0 2s #<---  
myweb-lbj89 1/1 Running 0 22m  
myweb-m6nrx 1/1 Running 0 22m

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5v9w6 1/1 Running 0 5s  
myweb-lbj89 1/1 Running 0 22m  
myweb-m6nrx 1/1 Running 0 22m

as you can see, when one pod is being "Terminating", immediately a new pod is spawned. eventually the old pod will go away and new pod will be up and running. total number of running pod remains unchanged.

you can also scale up/down replicas with rc. for example to scale "up" from number of 3 to 5:

$ kubectl scale rc myweb --replica=5  
replicationcontroller/myweb scaled

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5v9w6 1/1 Running 0 8s  
myweb-lbj89 1/1 Running 0 22m  
myweb-m6nrx 1/1 Running 0 22m  
myweb-hnnlj 0/1 ContainerCreating 0 2s  
myweb-kbgwm 1/1 ContainerCreating 0 2s

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5v9w6 1/1 Running 0 10s  
myweb-lbj89 1/1 Running 0 22m  
myweb-m6nrx 1/1 Running 0 22m  
myweb-hnnlj 1/1 Running 0 5s  
myweb-kbgwm 1/1 Running 0 5s

there are other benefits with RC. actually since this abstraction is so popular and heavily used in practice that, two very similar objects RS - ReplicaSet and Deploy - Deployment have been developped with more powerful features introduced. Roughly, you can call them "next generation of RC". let’s stop exploring more RC features for now and move our focus to these 2 new objects.

# ReplicaSet

ReplicaSet, or RS object, is pretty much the same thing as a RC object, with just one major exception - the looks of selector.

$ cat myweb-rs.yaml  
apiVersion: apps/v1  
kind: ReplicaSet  
metadata:  
 name: myweb  
spec:  
 replicas: 1  
 selector:  
 matchLabels: #<---  
 app: myweb #<---  
 matchExpressions: #<---  
 - {key: app, operator: In, values: [myweb]} #<---  
 template:  
 metadata:  
 labels:  
 app: myweb  
 spec:  
 containers:  
 - name: myweb  
 image: kubeguide/tomcat-app:v1  
 ports:  
 - containerPort: 8080  
 env:  
 - name: MYSQL\_SERVICE\_HOST  
 value: 'mysql'  
 - name: MYSQL\_SERVICE\_PORT  
 value: '3306'  
 - name: MYSQL\_ROOT\_PASSWORD  
 value: "123456"

RC uses "Equality-based" selector only while RS support extra selector format - "set-based". function-wise the two forms of selector do the same job - to "select" the pod with a matching "label".

#RS:  
selector:  
 matchLabels:  
 app: myweb  
 matchExpressions:  
 - {key: app, operator: In, values: [myweb]}

#RC:  
selector:  
 app: myweb

$ kubectl create -f myweb-rs.yaml  
replicaset.extensions/myweb created

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-lkwvt 1/1 Running 0 8s

a RS is created and it launchs a pod, just same as what a RC would do. if you compare the kubectl describe on the 2 objects:

$ kubectl describe rs myweb  
......  
Selector: app=myweb,app in (myweb) #<---  
......  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal SuccessfulCreate 15s replicaset-controller Created pod: myweb-kt9zx

$ kubectl describe rc myweb  
......  
Selector: app=myweb #<---  
......  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal SuccessfulCreate 19s replication-controller Created pod: myweb-tbbhc

as you see, most part of the output are the same, with only exception of selector format. you can also scale the RS same way as you do with RC:

$ kubectl scale rs myweb --replicas=5  
replicaset.extensions/myweb scaled

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-4jvvx 1/1 Running 0 3m30s  
myweb-722pf 1/1 Running 0 3m30s  
myweb-8z8f8 1/1 Running 0 3m30s  
myweb-lkwvt 1/1 Running 0 4m28s  
myweb-ww9tn 1/1 Running 0 3m30s

# Deployment

now you may start to wonder why kubernetes has different objects to do the almost same job. as mentioned earlier the features of RC has been extended through the RS and deployment. we’ve seen the RS , which has done the same job of RC only with a different selector format, now we’ll check out the other new object DEPLOY - deployment and explore the features coming from it.

## create a deployment

simply changing kind attribute from ReplicaSet to deployment we get the yaml file of a deployment object:

$ cat myweb-deployment.yaml  
apiVersion: apps/v1  
kind: Deployment #<---  
metadata:  
 name: myweb  
...(everything else remains the same as replicaset)...

$ kubectl create -f myweb-deployment.yaml  
deployment.extensions/myweb created

$ kubectl get deployment  
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE  
deployment.apps/myweb 1 1 1 1 21s

Actually the deployment is a relatively higher level of abstraction than RC and RS. deployment does not create a pod directly, the describe command reveals this:

$ kubectl describe deployments myweb  
Name: myweb  
Namespace: default  
CreationTimestamp: Sat, 25 May 2019 16:00:26 -0400  
Labels: app=myweb  
Annotations: deployment.kubernetes.io/revision: 1  
Selector: app=myweb,app in (myweb)  
Replicas: 1 desired | 1 updated | 1 total | 1 available | 0 unavailable  
StrategyType: RollingUpdate  
MinReadySeconds: 0  
RollingUpdateStrategy: 1 max unavailable, 1 max surge  
Pod Template:  
 Labels: app=myweb  
 Containers:  
 myweb:  
 Image: kubeguide/tomcat-app:v1  
 Port: 8080/TCP  
 Host Port: 0/TCP  
 Environment:  
 MYSQL\_SERVICE\_HOST: mysql  
 MYSQL\_SERVICE\_PORT: 3306  
 MYSQL\_ROOT\_PASSWORD: 123456  
 Mounts: <none>  
 Volumes: <none>  
Conditions:  
 Type Status Reason  
 ---- ------ ------  
 Available True MinimumReplicasAvailable  
OldReplicaSets: <none>  
NewReplicaSet: myweb-c586fd645 (1/1 replicas created) #<---  
Events: <none>

## deployment work flow

what happens is when you create a Deployment, a replica set is created automatically. The pods defined in a Deployment object are created and supervised by the Deployment’s replicaset.

the workflow is shown in the below diagram:

|=> pod  
 |  
deployment =====> RS ====|=> pod  
 |  
 |=> pod

You might still be wondering why you need RS as one more layer sitting between deployment and pod. you will find the answer in the next section.

## rolling update

"rolling update" feature is one of the "more powerful feature" coming with deployment object. in this section we’ll demonstrate the feature with a test case, then we’ll explain how it works.

### test rolling update

suppose we have a nginx-deployment, with replica=3 an pod image 1.7.9. later we want to upgrade the image from version 1.7.9 to new image version 1.9.1. with kuberctl we can use set image option and specify the new version number to trigger the update:

$ kubectl set image deployment/nginx-deployment nginx=nginx:1.9.1  
deployment.extensions/nginx-deployment image updated

now chck the deployment information again:

$ kubectl describe deployment/nginx-deployment  
Name: nginx-deployment  
Namespace: default  
CreationTimestamp: Tue, 11 Sep 2018 20:49:45 -0400  
Labels: app=nginx  
Annotations: deployment.kubernetes.io/revision=2  
Selector: app=nginx  
Replicas: 3 desired | 1 updated | 4 total | 3 available | 1 unavailable  
StrategyType: RollingUpdate  
MinReadySeconds: 0  
RollingUpdateStrategy: 25% max unavailable, 25% max surge  
Pod Template:  
 Labels: app=nginx  
 Containers:  
 nginx:  
 Image: nginx:1.9.1 #<------  
 Port: 80/TCP  
 Host Port: 0/TCP  
 Environment: <none>  
 Mounts: <none>  
 Volumes: <none>  
Conditions:  
 Type Status Reason  
 ---- ------ ------  
 Available True MinimumReplicasAvailable  
 Progressing True ReplicaSetUpdated  
OldReplicaSets: nginx-deployment-67594d6bf6 (3/3 replicas created)  
NewReplicaSet: nginx-deployment-6fdbb596db (1/1 replicas created)  
Events:  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal ScalingReplicaSet 4m deployment-controller Scaled up replica  
 set nginx-deployment-67594d6bf6 to 3 #<---  
 Normal ScalingReplicaSet 7s deployment-controller Scaled up replica  
 set nginx-deployment-6fdbb596db to 1 #<---

two changes we can observe here:

* image version in deployment is updated
* a new RS nginx-deployment-6fdbb596db is created, with a replica set to 1

and with the new RS with replica being 1, a new pod ("the fourth one") is now generated

$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
nginx-deployment-67594d6bf6-88wqk 1/1 Running 0 4m  
nginx-deployment-67594d6bf6-m4fbj 1/1 Running 0 4m  
nginx-deployment-67594d6bf6-td2xn 1/1 Running 0 4m  
nginx-deployment-6fdbb596db-4b8z7 0/1 ContainerCreating 0 17s #<------

the new pod is with new image:

$ kubectl describe pod/nginx-deployment-6fdbb596db-4b8z7 | grep Image:  
...(snipped)...  
 Image: nginx:1.9.1 #<---  
...(snipped)...

while the old pod is still with old image

$ kubectl describe pod/nginx-deployment-67594d6bf6-td2xn | grep Image:  
...(snipped)...  
 Image: nginx:1.7.9 #<------  
...(snipped)...

wait and keep checking the pods status, eventually all old pods are terminated and 3 new pods are running - the pod name confirms they are new ones:

$ kubectl get pods  
NAME READY STATUS RESTARTS AGE  
nginx-deployment-6fdbb596db-4b8z7 1/1 Running 0 1m  
nginx-deployment-6fdbb596db-bsw25 1/1 Running 0 18s  
nginx-deployment-6fdbb596db-n9tpg 1/1 Running 0 21s

so the "update" is done and all pods are now running with new version of the image.

### how it works

after you see our update process, you may argue that: hold on…​ this is now "update", this should be called "replacement" - kubernetes use 3 new pods running with new image to replace the old pods! precisely speaking, yes that is true. but that is how it works kubernetes’s "philosophy" - pod is cheap and replacement is easier. imaging how much work it will be when you have to "login" each pod, uninstall old images, cleaning up the environment and only to install a new image. let’s look at more details about this process and understand why it is called a "rolling" update.

when you update the pod with new software, the deployment object introduces a new RS that will start the pod update process. the idea is NOT to "login" to the existing pod and do the image update in there, instead, the new RC just creates a new pod equiped with the new software release in it. once this new (and "additional") pod is up and running, the original RS will be "scaled down" by one, making the total number of running pod remaining unchanged. new RS will continue to scale up by one and original RS scales down by same number. this process repeats until number of pods created by new RS reaches the original replica number defined in the deployment, and that is the time when all of the original RS’s pods are terminated. this process is depicted in this diagram:

| |=> pod-v1  
deployment ==|==> RS ====|=> pod-v1  
 | (v1) |=> pod-v1

| |=> pod-v1  
 |==> RS ====|=> pod-v1  
 | (v1) |  
deployment ==|  
 | |=> pod-v2  
 |==> RS ====|  
 | (v2) |

| |=> pod-v1  
 |==> RS ====|  
 | (v1) |  
deployment ==|  
 | |=> pod-v2  
 |==> RS ====|=> pod-v2  
 | (v2) |

| |  
 |==> RS ====|  
 | (v1) |  
deployment ==|  
 | |=> pod-v2  
 |==> RS ====|=> pod-v2  
 | (v2) |=> pod-v2

| |=> pod-v2  
deployment ==|==> RS ====|=> pod-v2  
 | (v2) |=> pod-v2

as you can see in this figure, this whole process of creating a new RS, scaling up the new RS and scaling down the old one simultaneously, is fully automated and taken care of by the deployment object. it is deployment who is deploying and driving ReplicaSet object, which, in this sense working as merely a backend of it.

this is why deployment is considered a higher layer object in kubernetes, also the reason why it is officially recommended to never use ReplicaSet alone without deployment.

### record

deployment also has the ability to "record" the whole process, so in case needed, you can review the update history after the update job is done:

$ kubectl describe deployment/nginx-deployment  
Name: nginx-deployment  
...(snipped)...  
NewReplicaSet: nginx-deployment-6fdbb596db (3/3 replicas created)  
Events:  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal ScalingReplicaSet 28m deployment-controller Scaled up replica set nginx-deployment-67594d6bf6 to 3 #<------  
 Normal ScalingReplicaSet 24m deployment-controller Scaled up replica set nginx-deployment-6fdbb596db to 1 #<------  
 Normal ScalingReplicaSet 23m deployment-controller Scaled down replica set nginx-deployment-67594d6bf6 to 2 #<------  
 Normal ScalingReplicaSet 23m deployment-controller Scaled up replica set nginx-deployment-6fdbb596db to 2 #<------  
 Normal ScalingReplicaSet 23m deployment-controller Scaled down replica set nginx-deployment-67594d6bf6 to 1 #<------  
 Normal ScalingReplicaSet 23m deployment-controller Scaled up replica set nginx-deployment-6fdbb596db to 3 #<------  
 Normal ScalingReplicaSet 23m deployment-controller Scaled down replica set nginx-deployment-67594d6bf6 to 0 #<------

### pause/resume/undo

additionally, you can also pause/resume the update process to verify the changes before proceeding:

$ kubectl rollout pause deployment/nginx-deployment  
$ kubectl rollout resume deployment/nginx-deployment

you can even "undo" the update when things are going wrong during the maintenance window

$ kubectl rollout undo deployment/nginx-deployment

$ kubectl describe deployment/nginx-deployment  
Name: nginx-deployment  
...(snipped)...  
NewReplicaSet: nginx-deployment-6fdbb596db (3/3 replicas created)  
NewReplicaSet: nginx-deployment-67594d6bf6 (3/3 replicas created)  
Events:  
 Type Reason Age From Message  
 ---- ------ --- ---- -------  
 Normal DeploymentRollback 8m deployment-controller Rolled back deployment "nginx-deployment" to revision 1 #<------  
 Normal ScalingReplicaSet 8m deployment-controller Scaled up replica set nginx-deployment-67594d6bf6 to 1 #<------  
 Normal ScalingReplicaSet 8m deployment-controller Scaled down replica set nginx-deployment-6fdbb596db to 2 #<------  
 Normal ScalingReplicaSet 8m deployment-controller Scaled up replica set nginx-deployment-67594d6bf6 to 2 #<------  
 Normal ScalingReplicaSet 8m deployment-controller Scaled up replica set nginx-deployment-67594d6bf6 to 3 #<------  
 Normal ScalingReplicaSet 8m deployment-controller Scaled down replica set nginx-deployment-6fdbb596db to 1 #<------  
 Normal ScalingReplicaSet 8m deployment-controller Scaled down replica set nginx-deployment-6fdbb596db to 0 #<------

Typically you do this when something is broken in your deployment. comparing with how much work it takes to prepare for the software upgrade during maintenance window in the old days, this is going to be a killing feature to have!

**Tip**

This is pretty much similar as the junos’s rollback magic command that you probably use everyday when you need to quickly revert the changes you make to your router.

# secret

## secret introduction

in any modern network system, user or administrator need to deal with sensitive information, such as username/passwords/ssh keys/etc, in the platfrom. same thing applies to the pods in kubernetes environment. However, exposing these information in your pod specs as cleartext may introduce security concerns and you need a tool/method to resolve the issue - at least to avoid the cleartext credentials as much as possible.

Kubernetes Secrets object is designed specifically for this purpose. it encodes all sensitive data and expose it into pods in a "controlled way".

this is the offical definition of kubernetes secrets:

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets. To use a secret, a pod needs to reference the secret.

there are many different types of secrets each serving a specific usage case. there are also many methods to create a secret and a lot of different ways to refer it in a pod. a complete discussion of secrets is out of the scope of this book. refer to the offical document to get all details and track all up-to-date changes.

In this section, we’ll look at some commonly used secret types. you will also learn several methods to create a secret and how to refer it in your pods. in the end, we will summarize the main benefits of kubernetes secrets object to understand how it will help to improve the sytem security.

opaque

this type of Secret can contain arbitrary key-value pairs, so it is treated as "unstructured" data from kubernetes’s perspective. all other types of secret has constaint content.

kubernetes.io/dockerconfigjson

this type of secret is used to authenticate with a private container registry (e.g. a juniper server) to pull your own private image.

tls

tls secret contains a TLS private key and certificate. it is used to secure an Ingress. you will see an example of Ingress with tls secret in chapter 4.

kubernetes.io/service-account-token

when processes running in containers of a pod access the apiserver, they has to be authenticated as a particular Account (e.g., account default by default). this account that is associated with a pod is called a service-account. kubernetes.io/service-account-token type of secret contains information about kubernetes service-account. we won’t elaborate this type of secret and service-account in this book.

## opaque secret

secret of type opaque, represents "arbitrary" user-owned data - usually you want to put in secret some kind of "sensitive" data, for example username, password, security pin, etc, just about anything you believe is sensitive and you want to carry into your pod.

### define opaque secret

first, to make our sensitive data looks "less sensitive", let’s encode them with base64 tool:

$ echo -n 'username1' | base64  
dXNlcm5hbWUx  
$ echo -n 'password1' | base64  
cGFzc3dvcmQx

then, we put the encoded version of the data in a secret definition yaml file:

apiVersion: v1  
kind: Secret  
metadata:  
 name: secret-opaque  
type: Opaque  
data:  
 username: dXNlcm5hbWUx  
 password: cGFzc3dvcmQx

alternatively, you can define the same secret from kubectl CLI directly, with --from-literal option:

kubectl create secret generic secret-opaque \  
 --from-literal=username='username1' \  
 --from-literal=password='password1'

either way, a secret will be generated:

$ kubectl get secrets  
NAME TYPE DATA AGE  
secret-opaque Opaque 2 8s  
  
$ kubectl get secrets secret-opaque -o yaml  
apiVersion: v1  
data:  
 password: cGFzc3dvcmQx  
 username: dXNlcm5hbWUx  
kind: Secret  
metadata:  
 annotations:  
 kubectl.kubernetes.io/last-applied-configuration: |  
 {"apiVersion":"v1","data":{"password":"cGFzc3dvcmQx","username":"dXNlcm5hbWUx"},"kind":"Secret","metadata":{"annotations":{},"name":"secret-opaque","namespace":"ns-user-1"},"type":"Opaque"}  
 creationTimestamp: 2019-08-22T22:51:18Z  
 name: secret-opaque  
 namespace: ns-user-1  
 resourceVersion: "885702"  
 selfLink: /api/v1/namespaces/ns-user-1/secrets/secret-opaque  
 uid: 5a78d9d4-c52f-11e9-90a3-0050569e6cfc  
type: Opaque

### refer opaque secret

once created, next you will need to use the secret in a pod, and the user information contained in secret will be carried into the pod. as mentioned there are different ways to refer the opaque secret in a pod and correspondingly the result will be different.

typically, user information carried from secret can be in one of these forms in a container:

* files
* environmental variables

here we’ll demonstrate using secret to generate environmental variables in container:

#pod-cirros-secret.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
 name: cirros  
 labels:  
 app: cirros  
spec:  
 containers:  
 - name: cirros  
 image: cirros  
 imagePullPolicy: Always  
 #envFrom:  
 #- secretRef:  
 # name: test-secret  
 env:  
 - name: SECRET\_USERNAME  
 valueFrom:  
 secretKeyRef:  
 name: secret-opaque  
 key: username  
 - name: SECRET\_PASSWORD  
 valueFrom:  
 secretKeyRef:  
 name: secret-opaque  
 key: password  
 restartPolicy: Always

spawn the pod and container from this yaml file:

$ kubectl apply -f pod/pod-cirros-secret.yaml  
pod/cirros-secret created

login the container and verify the generated environmental variables:

$ kubectl exec -it cirros-secret -- printenv | grep SECRET  
SECRET\_USERNAME=username1  
SECRET\_PASSWORD=password1

the original "sensitive" data we encoded with base64 is now present in the container!

## dockerconfigjson secret

dockerconfigjson secret, as the name indicates, carries the docker account credential information that is typically stored in a .docker/config.json file. the image in kubernetes pod may point to a private container registry. in that case, kubernetes need to authenticate with that registry in order to pull the image. dockerconfigjson type of secret is designed for this very purpose.

### docker credential data

the most straightforward method to create a kubernetes.io/dockerconfigjson type of secret is to provide login information directly to kubectl command and let it to generate the secret:

$ kubectl create secret docker-registry secret-jnpr1 \  
 --docker-server=hub.juniper.net \  
 --docker-username=JNPR-FieldUser213 \  
 --docker-password=CLJd2jpMsVc9zrAuTFPn  
secret/secret-jnpr created

**verify the secret creation.**

$ kubectl get secrets  
NAME TYPE DATA AGE  
secret-jnpr kubernetes.io/dockerconfigjson 1 6s #<---  
default-token-hkkzr kubernetes.io/service-account-token 3 62d

please note that only the first line in the output is the secret we just created. the second one is a kubernetes.io/service-account-token type of secret that was created by kubernetes system automatically when the contrail setup is up and running.

now inspect the details of the secret:

$ kubectl get secrets secret-jnpr -o yaml  
apiVersion: v1  
data:  
 .dockerconfigjson: eyJhdXRocyI6eyJodWIuanVuaXBlci5uZXQvc2...<snipped>...  
kind: Secret  
metadata:  
 creationTimestamp: 2019-08-14T05:58:48Z  
 name: secret-jnpr  
 namespace: ns-user-1  
 resourceVersion: "870370"  
 selfLink: /api/v1/namespaces/ns-user-1/secrets/secret-jnpr  
 uid: 9561cdc3-be58-11e9-9367-0050569e6cfc  
type: kubernetes.io/dockerconfigjson

not surprisingly, we don’t see any sensitive information in the form of cleartext, there is a data portion of the output where we see a very long string as the value of key .dockerconfigjson. it seems to has a transformed look from the original data, but at least it does not that "sensitive" anymore now - overall one purpose of using secret is to improve the system security.

however, the transformation is done by "encoding", not "encryption", so there is still a way to manually retrieve the original "sensitive" information back: just pipe the value of key .dockerconfigjson into base64 tool, the original username and password information is printed again:

**decode the secret data.**

$ echo "eyJhdXRocyI6eyJodWIuanVua..." | base64 -d | python -mjson.tool  
{  
 "auths": {  
 "hub.juniper.net": {  
 "auth": "Sk5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4=",  
 "password": "CLJd2jpMsVc9zrAuTFPn",  
 "username": "JNPR-FieldUser213"  
 }  
 }  
}

some highlights in the above output:

* python -mjson.tool is used to format the decoded json data before printing to the terminal.
* there is an auth key-value pair. it is the token generated based on the authentication information you gave (username and password).
* later on when equiped with this secret, a pod will use this token, instead of the username and password to authenticate itself towards the private docker registry hub.juniper.net in order to pull an docker image

**Tip**

here is another way to decode the data directly from the secret object:

$ kubectl get secret secret-jnpr1 \  
 --output="jsonpath={.data.\.dockerconfigjson}" \  
 | base64 --decode | python -mjson.tool  
{  
 "auths": {  
 "hub.juniper.net/security": {  
 "auth": "Sk5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4=",  
 "password": "CLJd2jpMsVc9zrAuTFPn",  
 "username": "JNPR-FieldUser213"  
 }  
 }  
}

the --output=xxxx option filters the kubectl get output so only value of .dockerconfigjson under data is printed. the value is then piped into base64 with option --decode (alias of -d) to get it decoded.

a docker-registry Secret created manually like this will only work with a single private registry. to support multiple private container registries we can create a secret from docker credential file.

### docker credential file (~/.docker/config.json)

as the name of key .dockerconfigjson in the secret we created indicates, it serves similar role as the docker config file: .docker/config.json. actually you can generate the secret directly from the docker config file.

**generate docker credential info.**

first let’s check the docker config file:

$ cat .docker/config.json  
{  
 ......  
 "auths": {},  
 ......  
}

nothing really. depending on the usage of the setups you may see different output here. but the point is that this docker config file will be updated automatically each time when you docker login a new registry.

let’s test it out.

$ cat mydockerpass.txt | \  
 docker login hub.juniper.net \  
 --username JNPR-FieldUser213 \  
 --password-stdin  
Login Succeeded

in file mydockerpass.txt is login password for my username JNPR-FieldUser213. saving the password in a file and then piping it to the docker login command with --password-stdin option has an advantage of not exposing the password cleartext in the shell history.

**Tip**

if you want you can give the password directly, and you will get a friendly warn that this is "insecure".

$ docker login hub.juniper.net --username JNPR-FieldUser213 --password MYPASS  
WARNING! Using --password via the CLI is insecure. Use --password-stdin.  
Login Succeeded

now the docker credential info is generated in the updated config.json file:

$ cat .docker/config.json  
{  
 ......  
 "auths": { #<---  
 "hub.juniper.net": {  
 "auth": "Sk5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4="  
 }  
 },  
 ......  
}

The login process creates or updates a config.json file that holds an authorization token.

**create secret from .docker/config.json file.**

$ kubectl create secret generic secret-jnpr2 \  
 --from-file=.dockerconfigjson=/root/.docker/config.json \  
 --type=kubernetes.io/dockerconfigjson  
secret/secret-jnpr2 created  
  
$ kubectl get secrets  
NAME TYPE DATA AGE  
secret-jnpr2 kubernetes.io/dockerconfigjson 1 8s #<---  
default-token-hkkzr kubernetes.io/service-account-token 3 63d  
secret-jnpr kubernetes.io/dockerconfigjson 1 26m

$ kubectl get secrets secret-jnpr2 -o yaml  
apiVersion: v1  
data:  
 .dockerconfigjson: ewoJImF1dGhzIjogewoJCSJodWIuanVuaXBlci5uZXQiOiB7CgkJCSJhdXRoIjogIlNrNVFVaTFHYVdWc1pGVnpaWEl5TVRNNlEweEtaREpxY0UxelZtTTVlbkpCZFZSR1VHND0iCgkJfQoJfSwKCSJIdHRwSGVhZGVycyI6IHsKCQkiVXNlci1BZ2VudCI6ICJEb2NrZXItQ2xpZW50LzE4LjAzLjEtY2UgKGxpbnV4KSIKCX0sCgkiZGV0YWNoS2V5cyI6ICJjdHJsLUAiCn0=  
kind: Secret  
metadata:  
 creationTimestamp: 2019-08-15T07:35:25Z  
 name: csrx-secret-dr2  
 namespace: ns-user-1  
 resourceVersion: "878490"  
 selfLink: /api/v1/namespaces/ns-user-1/secrets/secret-jnpr2  
 uid: 3efc3bd8-bf2f-11e9-bb2a-0050569e6cfc  
type: kubernetes.io/dockerconfigjson  
  
$ kubectl get secret secret-jnpr2 --output="jsonpath={.data.\.dockerconfigjson}" | base64 --decode  
{  
 ......  
 "auths": {  
 "hub.juniper.net": {  
 "auth": "Sk5QUi1GaWVsZFVzZXIyMTM6Q0xKZDJqcE1zVmM5enJBdVRGUG4="  
 }  
 },  
 ......  
}

### yaml file

you can also create a secret directly from a yaml file, just the same way you create other objects like service or Ingress.

manually encode the content of .docker/config.json file:

$ cat .docker/config.json | base64  
ewoJImF1dGhzIjogewoJCSJodWIuanVuaXBlci5uZXQiOiB7CgkJCSJhdXRoIjogIlNrNVFVaTFH  
YVdWc1pGVnpaWEl5TVRNNlEweEtaREpxY0UxelZtTTVlbkpCZFZSR1VHND0iCgkJfQoJfSwKCSJI  
dHRwSGVhZGVycyI6IHsKCQkiVXNlci1BZ2VudCI6ICJEb2NrZXItQ2xpZW50LzE4LjAzLjEtY2Ug  
KGxpbnV4KSIKCX0sCgkiZGV0YWNoS2V5cyI6ICJjdHJsLUAiCn0=

then put the base64 encoded value of .docker/config.json file as data in below yaml file:

apiVersion: v1  
kind: Secret  
type: kubernetes.io/dockerconfigjson  
metadata:  
 name: secret-jnpr3  
 namespace: ns-user-1  
data:  
 .dockerconfigjson: ewoJImF1dGhzIjogewoJCSJodW......

$ kubectl apply -f secret-jnpr.yaml  
secret/secret-jnpr3 created  
  
$ kubectl get secrets  
NAME TYPE DATA AGE  
default-token-hkkzr kubernetes.io/service-account-token 3 64d  
secret-jnpr1 kubernetes.io/dockerconfigjson 1 9s  
secret-jnpr2 kubernetes.io/dockerconfigjson 1 6m12s  
secret-jnpr3 kubernetes.io/dockerconfigjson 1 78s

keep in mind that Base64 is all about "encoding" instead of "encryption", it is considered the same as plain text. so sharing this file compromised secret.

### refer dockerconfigjson secret in pod: imagePullSecrets

after a secret is created, it can be referred by a pod/RC or deployment in order to pull an image from the private registry. there are many ways to refer the secrets. in this section we’ll look at using imagePullSecrets under pod spec to refer the secret.

An imagePullSecret is a way to pass a secret that contains a Docker (or other) image registry password to the Kubelet so it can pull a private image on behalf of your Pod.

create a pod pulling Juniper CSRX container from private repository:

apiVersion: v1  
kind: Pod  
metadata:  
 name: csrx-jnpr  
 labels:  
 app: csrx  
 annotations:  
 k8s.v1.cni.cncf.io/networks: '[  
 { "name": "vn-left-1" },  
 { "name": "vn-right-1" }  
 ]'  
spec:  
 containers:  
 #- name: csrx  
 # image: csrx  
 - name: csrx  
 image: hub.juniper.net/security/csrx:18.1R1.9  
 ports:  
 - containerPort: 22  
 #imagePullPolicy: Never  
 imagePullPolicy: IfNotPresent  
 stdin: true  
 tty: true  
 securityContext:  
 privileged: true  
 imagePullSecrets:  
 - name: secret-jnpr

generate the pod:

$ kubectl apply -f csrx/csrx-with-secret.yaml  
pod/csrx-jnpr created

the csrx is up and running:

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
csrx-jnpr 1/1 Running 0 20h

behind the scene, the pod authenticates itself towards the private registry, pulls the image, and launchs the CSRX container.

$ kubectl describe pod csrx  
......  
Events:  
19h Normal Scheduled Pod Successfully assigned ns-user-1/csrx to cent333  
19h Normal Pulling Pod pulling image "hub.juniper.net/security/csrx:18.1R1.9"  
19h Normal Pulled Pod Successfully pulled image "hub.juniper.net/security/csrx:18.1R1.9"  
19h Normal Created Pod Created container  
19h Normal Started Pod Started container

## tls secret

### create tls secret

### refer tls secret

## secret benefit

as you can see from our test, the Secret objects is created independently of the pods, and inspecting the object spec does not print the sensitive information directly on the screen.

Secrets are not written to the disk, but instead it is stored in a tmpfs FS only on nodes that need them. Also, Secrets are deleted when the pod that is dependent on them is deleted.

On most native Kubernetes distributions, communication between users and the apiserver is protected by SSL/TLS. Therefore, Secrets transmitted over these channels are properly protected.

Any given pod does not have access to the Secrets used by another pod, which facilitates encapsulation of sensitive data across different pods. Each container in a pod has to request a Secret volume in its volumeMounts for it to be visible inside the container. This feature can be used to construct security partitions at the pod level.

# Service

POD gets instantiated, terminated and moved from one Node to another, in doing so POD changes IP address so how would we keep track of that to get uninteruppted functonalites from pod? Even if the POD isn’t moving, how traffic reach group of PODs via single entity?

the answer for both questions is Kubernetes 'SVC - services'.

Services is an abstraction that defines a logical set of Pods and a policy by which you can access them, you may think of Services as your waiter in a big restaurant, this waiter isn’t cooking nor preparing the food but he just abstract everything happing at the kitchen for you as you deal only with this waiter.

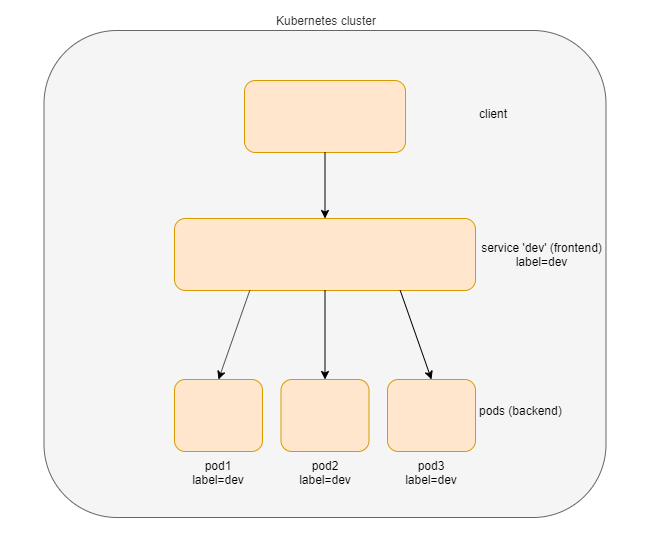
Simply Service is a layer 4 loadbalancer exposes pods functionalities via specific ip and port. The service and pods are linked via labels like RS.

so let’s understand different type of services:

* ClusterIP
* NodePort
* LoadBalancer

## ClusterIP service

the ClusterIP type of service is the simplest one. it is the default mode if the ServiceType is not specified. the kubernetes official website gives this diagram to illustrate how clusterIP service works:



ClusterIP service is exposed on a clusterIP and a service port. when client pods need to access the service it sends request toward this clusterIP and service port. This model works great if all requests are coming from inside of the same cluster. The nature of the ClusterIP limits the scope of this service to be only within the cluster. overall by default the ClusterIP is not reachable from external.

### create clusterIP service

let’s create our first service, with service type clusterIP.

$ cat service-web-clusterip.yaml  
apiVersion: v1  
kind: Service  
metadata:  
 name: service-web-clusterip  
spec:  
 ports:  
 - port: 8888  
 targetPort: 80  
 selector:  
 app: webserver

the yaml file looks pretty simple and self-explanatory. it defined a service service-web-clusterip with the "service port" 8888, mapping to targetPort which means "container port" 80 in some pod. the selector indicates that whichever pod with a label app: webserver will be choosen to be the backend pod responding service request.

now generate the service object by apply the yaml file:

$ kubectl apply -f service-web-clusterip.yaml  
service/service-web-clusterip created

following kubectl commands are commonly used to quickly verify the service and backend pod objects.

$ kubectl get svc -o wide  
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE SELECTOR  
service-web-clusterip ClusterIP 10.101.150.135 <none> 8888/TCP 9m10s app=webserver  
  
$ kubectl get pod -o wide -l 'app=webserver'  
No resources found.

the service is created successfully, there is no doubt about it. but there is no pods for the service. the reason is there is no pod with the label matching to the selector in the service. now we just need to create the pod with a proper label.

we can define a pod directly, but given the benefits of RC and deployment over pod as we’ve introduced earlier, use RC or deployment is more pratical. later on you will understand this is the right choice. in our example we define a RC object named rc-webserver.

$ cat rc-webserver.yaml  
apiVersion: v1  
kind: ReplicationController  
metadata:  
 name: rc-webserver  
 labels:  
 app: webserver  
spec:  
 replicas: 1 #<---  
 selector:  
 app: webserver  
 template:  
 metadata:  
 name: webserver  
 labels:  
 app: webserver #<---  
 spec:  
 containers:  
 - name: webserver  
 image: savvythru/contrail-frontend-app  
 securityContext:  
 privileged: true  
 ports:  
 - containerPort: 80

the RC rc-webserver has a label app: webserver, matching the SELECTOR in defined in our service. replicas: 1 instruct RC controller to launch only 1 pod at the moment.

$ kubectl apply -f rc-webserver.yaml  
replicationcontroller/rc-webserver created  
  
$ kubectl get pod -o wide -l 'app=webserver'  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
rc-webserver-vl6zs 1/1 Running 0 24s 10.47.255.238 cent333 <none>

immediately the pod is choosen to be the backend. here are some brief summaries about the output:

* the service got a "ClusterIP" or "service IP" of 10.106.176.17 allocated from the service IP pool.
* service port is 8888 as what is defined in yaml.
* by default the protocol type is TCP if not declared in yaml file. you can use protocol: UDP to declare a UDP service.
* the backend pod can be located with the label selector

**Tip**

the example shown use a "equality-based" selector (-l) to locate the backend pod, you can also use a "set-based" syntax to archive the same effect. for example: kubectl get pod -o wide -l 'app in (webserver)'

### verify cluserIP service

Now to verify if the service actually works, let’s start another pod as a client to initiate a http request toward the service. for this test we’ll login to a cirros pod and use curl command to send a http request toward the service. you’ll see the cirros pod being used as a client to send request throughout of this book.

$ kubectl exec -it cirros -- curl 10.101.150.135:8888  
<html>  
<style>  
 h1 {color:green}  
 h2 {color:red}  
</style>  
 <div align="center">  
 <head>  
 <title>Contrail Pod</title>  
 </head>  
 <body>  
 <h1>Hello</h1><br><h2>This page is served by a <b>Contrail</b>  
 pod</h2><br><h3>IP address = 10.47.255.238<br>Hostname =  
 rc-webserver-vl6zs</h3>  
 <img src="/static/giphy.gif">  
 </body>  
 </div>  
</html>

the http request toward the service reaches a backend pod running the web server application, which responds with a HTML page.

to better demonstrate which pod is providing the service, we are running a customized pod image that runs a simple web server. the web server is configured in such a way that whenever receiving a request, it will return a simple HTML page with local pod IP and hostname embeded. This way the curl returns something more meaningful in our test.

the returned HTML looks relatively "OK" to read, but there is a way to make it more "eye-friendly":

$ kubectl exec -it cirros -- curl 10.101.150.135:8888 | w3m -T text/html | head  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.238  
 Hostname = rc-webserver-vl6zs

the w3m tool is a "lightweight" console based web browser installed in the host. with w3m we can render a html webpage into text, which is more readable than the HTML page.

now we are convinced our service works. requests to service has been redirected to the correct backend pod, with a pod IP 10.47.255.238, pod name rc-webserver-vl6zs.

### specify a clusterIP

if you want to have a specific 'clusterIP', you can mention it in the spec. Ip address should be in service ip pool.

Sample yaml with specific 'clusterIP'

$ cat service-web-clusterip.yaml  
apiVersion: v1  
kind: Service  
metadata:  
 name: service-web-clusterip  
spec:  
 clusterIP: 10.101.150.150 #<---  
 ports:  
 - port: 8888  
 targetPort: 80  
 selector:  
 app: webserver

## NodePort service

NodePort service exposes a service on each node’s ip at a static port. It maps a static port on each node with a port of the application the POD as shown in the diagram

![node%20port%20chapter%203](data:text/html; charset=utf-8;base64,)

there is 2 very important parts in this services YAML file ports and selector.

targetPort is the actual port used by the application in here its port 80 as we are planning to run a web server and nodeport is port on each node.

selector is the label selector which determine which set of pods targeted by this services, in here any POD with label app: FRONT-END will be serviced by this services

apiVersion: v1  
kind: Service  
metadata:  
 name: web-app  
spec:  
 selector:  
 app: webserver  
 type: NodePort  
 ports:  
 - targetPort: 80  
 port: 80  
 nodePort: 32001 #<--- (optional)

**Note**

* Kubernetes by default allocate node port from (30000-32767) range if it is not mentioned in the spec. it could be changed using the flag --service-node-port-range. nodePort value also can be set. but it should be in the configurad range.
* The default service type is ClusterIP
* Be aware with the change of the Node ip address as it could effect your services

now let’s expose a nginx pod with the services shown above

[root@cent11]# cat nginx.yaml  
 apiVersion: v1  
 kind: Pod  
 metadata:  
 name: nginx-pod  
 labels:  
 app: webserver  
 spec:  
 containers:  
 - name: nginx-c  
 image: nginx  
  
[root@cent11]# kubectl create -f web-app.yaml  
service "web-app" created  
  
[root@cent11]# kubectl describe service web-app  
Name: web-app  
Namespace: default  
Labels: <none>  
Annotations: <none>  
Selector: app=webserver  
Type: NodePort  
IP: 10.98.108.168  
Port: <unset> 80/TCP  
TargetPort: 80/TCP  
NodePort: <unset> 32001/TCP  
Endpoints: 10.47.255.252:80  
Session Affinity: None  
External Traffic Policy: Cluster  
Events: <none>

Now we can test that by send CURL -i which is a http request using the CLI toward the (any)node IP address

[root@cent11 ~]# kubectl get pod -o wide  
  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
nginx-pod 1/1 Running 0 20m 10.47.255.252 cent222 <none>  
  
[root@cent11 ~]# kubectl describe node cent22 | grep InternalIP  
InternalIP: 10.85.188.17  
  
[root@cent11 ]#curl 10.85.188.17:32001  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
<style>  
 body {  
 width: 35em;  
 margin: 0 auto;  
 font-family: Tahoma, Verdana, Arial, sans-serif;  
 }  
</style>  
</head>  
<body>  
<h1>Welcome to nginx!</h1>  
<p>If you see this page, the nginx web server is successfully installed and  
working. Further configuration is required.</p>  
  
<p>For online documentation and support please refer to  
<a href="http://nginx.org/">nginx.org</a>.<br/>  
Commercial support is available at  
<a href="http://nginx.com/">nginx.com</a>.</p>  
  
<p><em>Thank you for using nginx.</em></p>  
</body>  
</html>

## loadbalancer service

essentially, a loadBalancer service goes one more step beyond what the NodePort service does. it exposes the Service externally using a cloud provider’s loadbalancer. loadbalancer by its nature automatically includes all features and functions of NodePort and ClusterIP Services.

Kubernetes clusters running on cloud providers support the automatic provision of a load balancer. the only difference between the 3 type of services are the type value. to reuse the same NodePort service yaml file and create a loadbalancer service, just set the type to LoadBalancer:

$ cat service-web-lb.yaml  
apiVersion: v1  
kind: Service  
metadata:  
 name: service-web-lb  
spec:  
 ports:  
 - port: 8888  
 targetPort: 80  
 selector:  
 app: webserver  
 type: LoadBalancer #<---

the cloud will see this keyword and a load balancer will be created. meanwile an external public loadbalancerIP is allocated serving as the frontend virtual IP. traffic coming to this loadbalancerIP will be redirected to the service backend pod. please keep in mind that this "redirection" process, is solely an transport layer operation. loadbalancerIP and port will be translated to private backend cluster IP and it’s targetPort. it does not involve any application layer activities. there is no such things like parsing URL, proxy HTTP request, etc like what will happen in HTTP proxying process. because the loadbalancerIP is publicly reachable, any Internet host whoever has access to the it (and the service port) can access the service provided by kubernetes cluster.

from Internet host’s perspective, when it requests service, it refers this public external loadbalancerIP plus service port and the request will reaches the backend pod. the loadbalancerIP is acting as a "gateway" between service inside of the cluster and outside world.

Some cloud providers allow you to specify the loadBalancerIP. In those cases, the load-balancer is created with the user-specified loadBalancerIP. If the loadBalancerIP field is not specified, the loadBalancer is set up with an ephemeral IP address. If you specify a loadBalancerIP but your cloud provider does not support the feature, the loadbalancerIP field that you set is ignored.

how is a loadbalancer implemented in loadbalancer service is "vendor-specific". a GCE loadbalancer may work in a totally different way with a AWS loadbalancer. we’ll have a detail demonstration about how loadbalancer service works in contrail kubernetes environment in chapter 4.

### externalIPs

Exposing service outside of the cluster can also be achieved via externalIPs option. here is an example:

apiVersion: v1  
kind: Service  
metadata:  
 name: service-web-externalips  
spec:  
 ports:  
 - port: 8888  
 targetPort: 80  
 selector:  
 app: webserver  
 externalIPs: #<---  
 - 101.101.101.1 #<---

In the Service spec, externalIPs can be specified along with any of the ServiceTypes. externalIPs are not managed by Kubernetes and are the responsibility of the cluster administrator.

**Note**

externalIPs are different from loadbalancerIP. loadbalancingIP is the IP assigned by cluster administrator, while externalIPs comes with the loadbalancer created by the cluster that supports it.

## service implementation: kube-proxy

By default kubernetes uses kube-proxy module for services, but CNI providers can have there own implementations for services.

**kube-proxy deployment mode.**

kube-proxy can be deployed in one of the 3 modes:

* user-space proxy-mode
* iptables proxy-mode
* ipvs proxy-mode

when the traffic hits the node, it would be forwarded to one of the back end pod via a depolyed kube-proxy forwarding plane. the detail explanations and comparison of these 3 modes will not be covered by this book, but you can check kubernetes official website for more informations. in chapter4 we’ll illustrate how contrail as CNI provider implements the service.

# Endpoints

in our 'service' introduction, there is one object that is involved but we haven’t explored is 'EP - endpoint'. we’ve learned it is through label selector that a particular pod or group of pods with matching labels are choosen to be the backend, so that the service request traffic will be redirected to them. The IP and port information of the "matching" pods are maintained in the 'endpoint' object. The pods may die and spawn anytime, the "mortal" nature of the pod will most possibly make the new pods be respawned with new IP address. during this dynamic process the 'endpoints' will always be updated accordingly to reflect the current backend pod IPs, so the service traffic redirection will act properly. (CNI providers who has their own service implementation update the backends of the service based on the endpoints objects)

here is an example to demonstrate some quick steps to verify the service, corresponding endpoint and the pod with matching labels

create a service:

$ cat svc/service-web-clusterip.yaml  
apiVersion: v1  
kind: Service  
metadata:  
 name: service-web-clusterip  
spec:  
 ports:  
 - port: 8888  
 targetPort: 80  
 selector:  
 app: webserver

list the endpoint:

$ kubectl get ep  
NAME ENDPOINTS AGE  
service-web-lb 10.47.255.252:80 5d17h

locate pod with the label that is used by selector in service:

$ kubectl get pod -o wide -l 'app=webserver'  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE LABELS  
rc-webserver-rjlgr 1/1 Running 4 5d17h 10.47.255.252 cent333 <none> app=webserver

scale the backend pods

$ kubectl scale rc webserver --replicas=3

$ kubectl get pod -o wide -l 'app=webserver'  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE LABELS  
rc-webserver-rjlgr 1/1 Running 4 5d17h 10.47.255.252 cent333 <none> app=webserver  
rc-webserver-45skv 1/1 Running 0 5s 10.47.255.251 cent222 <none> app=webserver  
rc-webserver-m2cp5 1/1 Running 0 5s 10.47.255.250 cent111 <none> app=webserver

$ kubectl get ep  
NAME ENDPOINTS AGE  
service-web-lb 10.47.255.250:80,10.47.255.251:80,10.47.255.252:80 5d17h

**Service without SELECTOR.**

in the preceding example, the Endpoints object is generated automatically by the kubernetes system whenever a service is created, and at least one pod with matching label exists. Another use case of endpoint, is for a service that has no label selector defined. in that case you can manually map the service to the network address and port where it’s running, by adding an endpoint object manually and you can connect the endpoint with the service. this can be very useful in some scenarios. for example, in your setup you have a backend web server running in a physical server, you still want to integrate it into a kubernetes Service. you just create the service as usual, and then create an endpoint with an "address" and "port" pointing to the web server. that’s it! the Service does not care about the backend type, it just redirect the service request traffic exactly the same way as if all backend is pod.

# Ingress

You’ve now seen ways of exposing a service to clients outside the cluster. another method is Ingress

## kubernetes Ingress introduction

in service section, we understand that service works in transport layer. in reality, you access all services via URLs.

'Ingress' or 'ing' for short is another core concept of kubernetes allows HTTP/HTTPS routing that does not exist in service. Ingress is built on top of service. with Ingress, you can define URL-based rules to distribute HTTP/HTTPS routes to multiple different "backend services" ie ingress exposes services via HTTP/HTTPS routes. we’ve learned a lot about kubernetes service so far, so you understand what will happen after that - the requests will be forwarded to each service’s corresponding backend pods.

## Ingress vs service

there are similiarities between loadbalancer service and ingress. both can expose service to outside of the cluster. but there are some main differences.

**operation layer/level.**

Ingress operates at the application layer of the OSI network model, while service operates at transport layer only. Ingress understand the HTTP/HTTPS protocol, service only does forwarding based on IP and port, which means it does not care about the application layer protocol (HTTP/HTTPS) details. Ingress can operate at transport layer. Operating ingress at transoport layer does not make sense since service does the same unless there is a special reason to do.

**forwarding mode.**

Ingress does the application layer "proxy", in pretty much the same way a traditional web loadbalancer does. a typical web loadbalancer proxy sitting between machine A (client) and B (server), works at the application layer. it is "aware of" the application layer protocols (HTTP/HTTPS) so the client-server intraction does NOT look "transparent" to the loadbalancer. basically It creates two connections each with source (A) and destination (B) machine. Machine A does not even know about the existence of machine B at all. For machine A, Proxy is the only thing it talks to and it does not care how and where the proxy gets its data.

**number of public IPs.**

each service of the ingress needs an public ip if it is exposed directly to outside of the cluster. when ingress is a front-end to all these services, one public ip would be sufficient which makes life easy for cloud-admin.

## Ingress object

before we talk about Ingress object, the best way to get a feel of it is to look at the yaml definition:

apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
 name: ingress-sf  
spec:  
 rules:  
 - host: www.juniper.net  
 http:  
 paths:  
 - path: /dev  
 backend:  
 serviceName: webservice-1  
 servicePort: 8888  
 - path: /qa  
 backend:  
 serviceName: webservice-2  
 servicePort: 8888

it looks pretty simple. the spec defines only one item that is the rules. the rules says a host, which is "juniper" URL here, may have 2 possible path in the URL string. the path is whatever follows the host in the URL, in this case they are /dev and /qa. each path is then associated to a different service. when Ingress sees HTTP requests arrives, it proxies the traffic to each URL path’s associated backend service. each service, as we’ve learned this in service section, will deliver the request to their corresponding backend path. that’s it. actually this is one of the 3 types of Ingress that kubernetes supports today - "simple fan-out Ingress". later we’ll introduce the other two types of Ingress.

**about URL, host, path.**

term host and path are used frequently in kubernetes Ingress documentations. host:: is "fully qualified domain name" of a server. path, or url-path:: is the rest part of the string after the host in a URL. in the case of having a port in the URL, then it is the strings after the port.

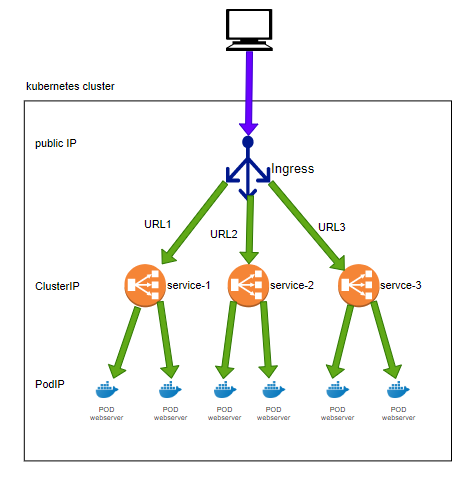
let’s take a look at the following URL:

http://www.juniper.net:1234/my/resource  
 --------------- ---- -----------  
 host port path

http://www.juniper.net:/my/resource  
 --------------- ------------  
 host path

host is www.juniper.net, whatever follows port 1234 is called path, my/resource in this example. if a URL has no port, then the strings following host are path. for more details you can read rfc1738, but for the context of this book understanding what we introduce here would suffice.

if you now think kubernetes Ingress is nothing but to define some rules, and the rules are just to instruct the system to direct incoming request to different services, based on the URLs, you are basically right in the high level. the figure below shows the dependency between the 3 kubernetes object: Ingress, service and pod:



Ingress

in practice there are other things you need to understand. in reality to handle the ingress rules, you need at least another component called ingress  
controller

## ingress controller

An ingress controller is responsible for reading the Ingress rules and program the rules into the proxy which does the real work - dispatching traffic based on host / URL.

ingress controllers are tyically implemented by a third party vendors. Different Kubernetes environments have different ingress controller based on the need of the cluster. each ingress controllers have their own implementations to program the ingress rules. bottom line is, there has to be an Ingress controller running in the cluster.

some ingress controller providers:

* nginx
* gce
* haproxy
* avi
* f5
* istio
* contour

You may deploy any number of ingress controllers within a cluster. When you create an ingress, you should annotate each ingress with the appropriate ingress.class to indicate which ingress controller should be used if more than one exists within your cluster. annotation used in ingress objects will be explained in the "annotation" section.

## ingress examples

there are basically 3 types of ingresses:

* Single Service Ingress
* Simple fanout Ingress
* Name based virtual hosting Ingress

we’ve looked at the "simple fanout Ingress". now let’s also look at yaml file example for the other two type of Ingress.

### single service ingress

apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
 name: ingress-single-service  
spec:  
 backend:  
 serviceName: webservice  
 servicePort: 80

this is the simplest form of ingress. the ingress will get an external IP so the service will be exposed to the public, however, it has no rules defined, so it does not parse host or path in the URLs. all requests will goes to one same service.

### simple fanout ingress

apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
 name: ingress-sf  
spec:  
 rules:  
 - host: www.juniper.net  
 http:  
 paths:  
 - path: /dev  
 backend:  
 serviceName: webservice-1  
 servicePort: 8888  
 - path: /qa  
 backend:  
 serviceName: webservice-2  
 servicePort: 8888

we’ve checked this out in the beginning of this section. comparing with single  
service ingress, simple fanout ingress is more practical. it is not only able to expose service via a public IP, but also able to do "URL routing" or "fan out" based on the path. this is a very common usage scenario when a company wants to direct traffic to each department’s dedicated servers based on the "suffix" of URL after the domain name.

### virtual host ingress

apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
 name: ingress-virutal-host  
spec:  
 rules:  
 - host: www.juniperhr.com  
 http:  
 paths:  
 - backend:  
 serviceName: webservice-1  
 servicePort: 80  
 - host: www.junipersales.com  
 http:  
 paths:  
 - backend:  
 serviceName: webservice-2  
 servicePort: 80

name based virtual host is similar to simple fanout ingress in that, it is able to do rule-based URL routing. the unique power of this type of Ingress is that it supports routing HTTP traffic to multiple host names at the same IP address. the example above may not be practical (unless one day the two domains merge!) but it is good enough to showcase the idea. in the yaml file 2 "hosts" are defined, the "juniperhr" and "junipersales" URL respectively. even though ingress will be allocated with one public IP only, based on the host in URL, request toward that same public IP will still be routed to different backend services - that is why it is called a "virtual hosting Ingress". we’ll have a very detail case study in chapter 4 about this example.

**Note**

it is also possible to merge a "simple fanout" Ingress and a "virtual host" Ingress in one Ingress, in this book we won’t cover this topic though.

## multiple ingress controller

you can have multiple ingress controllers in one cluster. in that case the cluster needs to know which one to choose. for example, later on in chapter 4 we’ll talk about contrail’s built-in Ingress controller which, does not stop us from installing another third party Ingress controller like "nginx" Ingress controller. we will end up having 2 Ingress controllers in the same cluster with the names are:

* opencontrail (default)
* nginx

contrail’s implementation is the default one so you don’t have to select it explicitly. to select nginx as Ingress controller, use this annotations kubernetes.io/ingress.class:

metadata:  
 name: foo  
 annotations:  
 kubernetes.io/ingress.class: "nginx"

this will make contrail’s Ingress controller opencontrail to ignore the Ingress configuration.

## TLS support

# kubernetes networking

# Network policy

In Kubernetes pods can reach any pods by default. Then how pods can be secured? The answer is network policy. Networkpolicy is a Kubernetes resource like pod, service, ingress and etc. It defines who are all can reach the pod(ingress) and whom the pod can reach(egress).

## Prerequisites

Network polices are implemented by the network plugin, so you must be using a network solution which supports Network Policy. Simply creating the resource without a controller to implement it will have no effect.

Network policy logically can be divided into two sections. The first section will identify the pod(s) where the Network policy would be applied. The second section will define the ingress and egress rules for the selected pod(s).

## Pod(s) selection

How the pods would be selected? Yes. You are right. Pod(s) are identified using labels.

podSelector:  
 matchLabels:  
 role: db

In the above example the network policy would be applied to the pods which has the label "role: db".

## Ingress and egress rules for group of pod(s)

The second section defines the policy types for the selected pod(s). Policy type can be ingress or egress or both. Ingress is the default policy type. policy identifies the network endpoint where the selected pod(s) can communicate. Network endpoint can be ip address block or pod(s) (all pods or group of pods) in a namespace or selected pods in the same namespace. Ingress network-endpoint has to be defined in the "from" section. Egress network-endpoint has to be defined in the "to" section.

policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 172.17.0.0/16  
 except:  
 - 172.17.1.0/24  
 - namespaceSelector:  
 matchLabels:  
 project: myproject  
 - podSelector:  
 matchLabels:  
 role: frontend  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.0.0.0/24

In the above example:

1. The ingress network points are
   1. 172.17.0.0/16 and port except 172.17.1.0/24
   2. All the pods in namespaces which has the label “project: myproject”.
   3. Pods which has the label "role: frontend"
2. The egress network points are 10.0.0.0/24

Is there any way to select few pods from namespaces instead of all pods in the namespaces? Yes. It can be specified in the namespaceSelector. namespaceSelector can have podSelector. When namespaceSelector has podSelector, network endpoint would be pods with matching labels in the selected namespaces.

The below example shows that allowing connections from pods with label role=client in namespaces with the label user=alice. Please be aware to use correct yaml syntax.

...  
ingress:  
- from:  
 - namespaceSelector:  
 matchLabels:  
 user: alice  
 podSelector:  
 matchLabels:  
 role: client  
...

So far it is fine. Still there is a security concern. Is there any way to specify ports for ingress and egress? Yes. As part of the policy it can be mentioned. If it is not mentioned it applies to all ports. Ports in ingress says that selected pod(s) can allow traffic for the specified ports. Ports in egress says that selected pod(s) can send traffic to specified ports.

Previous example along with port specifications

policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 172.17.0.0/16  
 except:  
 - 172.17.1.0/24  
 - namespaceSelector:  
 matchLabels:  
 project: myproject  
 - podSelector:  
 matchLabels:  
 role: frontend  
 ports:  
 - protocol: TCP  
 port: 6379  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.0.0.0/24  
 ports:  
 - protocol: TCP  
 port: 5978

The above network policy says that all ingress network endpoint can reach selected pod(s) tcp port 6379 and selected pod(s) can reach all egress network endpoint’s tcp port 5978. The rest of the traffic would be blocked.

Sample network-policy

apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
 name: mydb  
spec:  
 podSelector:  
 matchLabels:  
 role: db  
 policyTypes:  
 - Ingress  
 - Egress  
 ingress:  
 - from:  
 - ipBlock:  
 cidr: 172.17.0.0/16  
 except:  
 - 172.17.1.0/24  
 - namespaceSelector:  
 matchLabels:  
 project: myproject  
 - podSelector:  
 matchLabels:  
 role: frontend  
 ports:  
 - protocol: TCP  
 port: 6379  
 egress:  
 - to:  
 - ipBlock:  
 cidr: 10.0.0.0/24  
 ports:  
 - protocol: TCP  
 port: 5978

kubectl create -f mydb-netpol.yaml  
  
kubectl get netpol  
NAME POD-SELECTOR AGE  
mydb role=db 3m5s  
  
kubectl describe netpol mydb  
Name: mydb  
Namespace: default  
Created on: 2019-06-30 07:41:18 -0700 PDT  
Labels: <none>  
Annotations: <none>  
Spec:  
 PodSelector: role=db  
 Allowing ingress traffic:  
 To Port: 6379/TCP  
 From:  
 IPBlock:  
 CIDR: 172.17.0.0/16  
 Except: 172.17.1.0/24  
 From:  
 NamespaceSelector: project=myproject  
 From:  
 PodSelector: role=frontend  
 Allowing egress traffic:  
 To Port: 5978/TCP  
 To:  
 IPBlock:  
 CIDR: 10.0.0.0/24  
 Except:  
 Policy Types: Ingress, Egress

# livenessProbe and readinessProbe

## Liveness Probe

What happen if the application in the POD is running but it can’t serve its main purpose for whatever reason? also applications that runs for long time might transition to broken states. In all cases the last thing you want have is a call reporting a problem in an application that could be easily fixed with restarting the POD. liveness probes is a Kubernetes features made specially for that. liveness probes sent a pre-defined request to the POD on a regular basis then restart the POD if this request failed. The most commonly used liveness probe is HTTP GET request, but it could also be opening TCP socket or issuing a command

this is a HTTP GET request probe example where the “initialDelaySeconds” is the waiting time before the first try to HTTP GET request to port 80 then it will run the probe every 20 second as specified in “periodSeconds” If that failed the POD would be restarted automatically. you have the option to specify the path which in here just the main website. also you can send the probe with customized header

apiVersion: v1  
kind: Pod  
metadata:  
 name: liveness-pod  
 labels:  
 app: tcpsocket-test  
spec:  
 containers:  
 - name: liveness-pod  
 image: virtualhops/ato-ubuntu:latest  
 ports:  
 - containerPort: 80  
 securityContext:  
 privileged: true  
 capabilities:  
 add:  
 - NET\_ADMIN  
 livenessProbe:  
 httpGet:  
 path: /  
 port: 80  
 httpHeaders:  
 - name: some-header  
 value: Running  
 initialDelaySeconds: 15  
 periodSeconds: 20

let’s launch this POD then login to it to terminate the proccess that handle the httpGet

[root@cent11 ~]# kubectl get pod  
NAME READY STATUS RESTARTS AGE  
liveness-pod 1/1 Running 0 114s  
  
  
[root@cent11 ~]# kubectl exec -it liveness-pod bash  
root@liveness-pod:/# sudo netstat -tulpn  
  
Active Internet connections (only servers)  
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name  
tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 111/apache2  
tcp 0 0 0.0.0.0:22 0.0.0.0:\* LISTEN 45/sshd  
tcp6 0 0 :::22 :::\* LISTEN 45/sshd  
  
root@liveness-pod:/# service apache2 stop  
 \* Stopping web server apache2 \*  
  
root@liveness-pod:/# sudo netstat -tulpn  
Active Internet connections (only servers)  
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name  
tcp 0 0 0.0.0.0:22 0.0.0.0:\* LISTEN 45/sshd  
tcp6 0 0 :::22 :::\* LISTEN 45/sshd  
  
[root@cent11 ~]# kubectl get pod  
NAME READY STATUS RESTARTS AGE  
liveness-pod 1/1 Running 1 5m33s

you can see the POD got restarted automatically and in the event it stated the reason for that restart :

Killing container with id docker://liveness-pod:Container failed liveness probe.. Container will be killed and recreated.

[root@cent11 ~]# kubectl describe pod liveness-pod  
Name: liveness-pod  
Namespace: default  
Priority: 0  
PriorityClassName: <none>  
Node: cent22/10.85.188.17  
Start Time: Fri, 05 Jul 2019 16:39:12 -0400  
Labels: app=tcpsocket-test  
Annotations: k8s.v1.cni.cncf.io/network-status:  
 [  
 {  
 "ips": "10.47.255.249",  
 "mac": "02:c2:59:4a:82:9f",  
 "name": "cluster-wide-default"  
 }  
 ]  
Status: Running  
IP: 10.47.255.249  
Containers:  
 liveness-pod:  
 Container ID: docker://01969f51d32f38a15baab18487b85c54cee4125f55c8c7667236722084e4df06  
 Image: virtualhops/ato-ubuntu:latest  
 Image ID: docker-pullable://virtualhops/ato-ubuntu@sha256:fa2930cb8f4b766e5b335dfa42de510ecd30af6433ceada14cdaae8de9065d2a  
 Port: 80/TCP  
 Host Port: 0/TCP  
 State: Running  
 Started: Fri, 05 Jul 2019 16:41:35 -0400  
 Last State: Terminated  
 Reason: Error  
 Exit Code: 137  
 Started: Fri, 05 Jul 2019 16:39:20 -0400  
 Finished: Fri, 05 Jul 2019 16:41:34 -0400  
 Ready: True  
 Restart Count: 1  
 Liveness: http-get http://:80/ delay=15s timeout=1s period=20s #success=1 #failure=3  
 Environment: <none>  
 Mounts:  
 /var/run/secrets/kubernetes.io/serviceaccount from default-token-m75c5 (ro)  
Conditions:  
 Type Status  
 Initialized True  
 Ready True  
 ContainersReady True  
 PodScheduled True  
Volumes:  
 default-token-m75c5:  
 Type: Secret (a volume populated by a Secret)  
 SecretName: default-token-m75c5  
 Optional: false  
QoS Class: BestEffort  
Node-Selectors: <none>  
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s  
 node.kubernetes.io/unreachable:NoExecute for 300s  
Events:  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal Scheduled 7m19s default-scheduler Successfully assigned default/liveness-pod to cent22  
 Warning Unhealthy 4m6s (x3 over 4m46s) kubelet, cent22 Liveness probe failed: Get http://10.47.255.249:80/: dial tcp 10.47.255.249:80: connect: connection refused  
 Normal Pulling 3m36s (x2 over 5m53s) kubelet, cent22 pulling image "virtualhops/ato-ubuntu:latest"  
 Normal Killing 3m36s kubelet, cent22 Killing container with id docker://liveness-pod:Container failed liveness probe.. Container will be killed and recreated.  
 Normal Pulled 3m35s (x2 over 5m50s) kubelet, cent22 Successfully pulled image "virtualhops/ato-ubuntu:latest"  
 Normal Created 3m35s (x2 over 5m50s) kubelet, cent22 Created container  
 Normal Started 3m35s (x2 over 5m50s) kubelet, cent22 Started container

This is a TCP socket probe example. TCP socket probe is similar to the HTTP GET request probes, but it will open TCP socket.

apiVersion: v1  
kind: Pod  
metadata:  
 name: liveness-pod  
 labels:  
 app: tcpsocket-test  
spec:  
 containers:  
 - name: liveness-pod  
 image: virtualhops/ato-ubuntu:latest  
 ports:  
 - containerPort: 80  
 securityContext:  
 privileged: true  
 capabilities:  
 add:  
 - NET\_ADMIN  
 livenessProbe:  
 tcpSocket:  
 port: 80  
 initialDelaySeconds: 15  
 periodSeconds: 20

command is like HTTP GET and TCP socket probes. But the probe will execute the command in the container.

apiVersion: v1  
kind: Pod  
metadata:  
 name: liveness-pod  
 labels:  
 app: command-test  
spec:  
 containers:  
 - name: liveness-pod  
 image: k8s.gcr.io/busybox  
 args:  
 - /bin/sh  
 - -c  
 - touch /tmp/healthy; while true; do sleep 600;done;  
 livenessProbe:  
 exec:  
 command:  
 - cat  
 - /tmp/healthy  
 initialDelaySeconds: 5  
 periodSeconds: 5

## Readiness Probe

Liveness probe make sure that your POD is in good health, but for some application Liveness alone isn’t enough. some application need to load large files before it start. you might think if we set a higher “initialDelaySeconds” value then problem solve. but this not an efficient way. Readiness probe is solution in here specially with Kubernetes services, as the POD will not receive a traffic until it is ready. Whenever the readiness probe fails, the endpoint for the pod would be removed from the service and it would be added back when the readiness probe succeeds. Readiness Probe is configured the same way as liveness probe

apiVersion: v1  
kind: Pod  
metadata:  
 name: liveness-readiness  
 labels:  
 app: tcpsocket-test  
spec:  
 containers:  
 - name: liveness-readiness-pod  
 image: virtualhops/ato-ubuntu:latest  
 ports:  
 - containerPort: 80  
 securityContext:  
 privileged: true  
 capabilities:  
 add:  
 - NET\_ADMIN  
 livenessProbe:  
 httpGet:  
 path: /  
 port: 80  
 httpHeaders:  
 - name: some-header  
 value: Running  
 initialDelaySeconds: 15  
 periodSeconds: 20  
 readinessProbe:  
 tcpSocket:  
 port: 80  
 initialDelaySeconds: 5  
 periodSeconds: 10

**Note**

its recommended to use both Readiness Probe and Liveness Probe where Liveness probe restart the POD if it failed and Readiness Probe make sure the POD is ready before it gets the traffic

## Probe Parameters

Probes have a number of parameters that you can use to more precisely control the behavior of liveness and readiness checks.

1. initialDelaySeconds: Number of seconds after the container has started before liveness or readiness probes are initiated.
2. periodSeconds: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
3. timeoutSeconds: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
4. successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.
5. failureThreshold: When a Pod starts and the probe fails, Kubernetes will try failureThreshold times before giving up. Giving up in case of liveness probe means restarting the Pod. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

HTTP probes have additional parameters that can be set on httpGet.

1. host: Host name to connect to, defaults to the pod IP. You probably want to set “Host” in httpHeaders instead.
2. scheme: Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to HTTP.
3. path: Path to access on the HTTP server.
4. httpHeaders: Custom headers to set in the request. HTTP allows repeated headers.
5. port: Name or number of the port to access on the container. Number must be in the range 1 to 65535.

# Annotation

We have seen before how labels in Kubernetes are used for identifying, selecting and organizing objects, labels are just one way to attach metadata to Kubernetes objects.

Another way is Annotations which is a key/value maps that attach non-identifying metadata to objects, Annotation has a lot of use cases such as attaching

* pointers for logging and analytics
* phone number, directory entries and web site
* timestamps, image hashes and registry address
* network, namespaces
* type of ingress controller

example for annotations: --- apiVersion: v1 kind: Pod metadata: name: annotations-demo annotations: #←-- imageregistry: "https://hub.docker.com/" spec: containers: - name: nginx image: nginx:1.7.9 ports: - containerPort: 80 ---

## Annotations in Ingress

## Annotations in Network Attachment Defination

Annotations can be used to assign network information to POD and we will see later on in chapter 4 how Kubernetes annotation can instruct contrail to attach an interface to certain network

Before seeing Annotations in action lets first create a network with minimum configuration based on the De-facto Kubernetes Network custom resource definition. Network Attachment Definition is used to indicate the CNI as well the paraments of the network where we will attached interface POD to

apiVersion: "k8s.cni.cncf.io/v1"  
kind: NetworkAttachmentDefinition  
metadata:  
 name: net-a  
spec:  
 config: '{  
 "cniVersion": "0.3.0",  
 "type": "awesome-plugin"  
 }'

The type in the example “awesome-plugin” is the name of the CNI which and could be Flannel, Calico, Contrail-K8s-cni , …,etc

Creating a POD and using annotations to attach its interface to a network called net-a

kind: Pod  
metadata:  
 name: my-pod  
 namespace: my-namespace  
 annotations:  
 k8s.v1.cni.cncf.io/networks: net-a

Note: According to De-facto Kubernetes Network custom resource definition the annotation "k8s.v1.cni.cncf.io/networks” is used to represent “ NetworkAttachmentDefinition” and has two format

Network  
 k8s.v1.cni.cncf.io/networks: net-a

Namespace/network name  
 k8s.v1.cni.cncf.io/networks: ns/net-a

**Note**

To maintain compatibility with existing Kubernetes deployments, All pods must still be attached to the cluster-wide default network. which means even if we attached one POD interface to a specific network, this POD would have two interfaces one attached to the cluster-wide default network and the other interface is attached to the network specified in the annotation argument (net-a in this case)