Kubernetes in practice

2019-07-05

This chapter introduces some of the fundamental and powerful 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 are Key/value pairs attached to an object let’s see how can we use label with node selector to make a POD is lunched on a certain machine

**Tip**

* 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 these two nodes respectively then try to lunch a POD with label zone: testing

kubectl get nodes --show-labels

NAME STATUS ROLES AGE VERSION LABELS  
ip-172-25-1-216.us-west-1.compute.internal Ready <none> 2h v1.9.2 <none>  
ip-172-25-1-56.us-west-1.compute.internal NotReady <none> 2h v1.9.2 <none>  
ip-172-25-1-83.us-west-1.compute.internal Ready <none> 2h v1.9.2 <none>

kubectl label nodes ip-172-25-1-83.us-west-1.compute.internal stage=testing  
kubectl label nodes ip-172-25-1-216.us-west-1.compute.internal stage=production

kubectl get nodes --show-labels

NAME STATUS ROLES AGE VERSION LABELS  
ip-172-25-1-216.us-west-1.compute.internal Ready <none> 2h v1.9.2 stage=production  
ip-172-25-1-56.us-west-1.compute.internal NotReady <none> 2h v1.9.2 <none>  
ip-172-25-1-83.us-west-1.compute.internal Ready <none> 2h v1.9.2 stage=testing

now let’s lunch a basic Nginx POD tagged with stage=testing and confirm it will land on a node tagged with stage=testing

[root@ip-172-25-1-56 /]# cat > web-server.yaml  
apiVersion: v1  
kind: Pod  
metadata:  
 name: nginx  
 labels:  
 app: webserver  
spec:  
 containers:  
 - name: nginx  
 image: nginx  
 nodeSelector:  
 stage: testing

[root@ip-172-25-1-56 /]# kubectl create -f web-server.yaml  
pod "wordpress" created

[root@ip-172-25-1-56 /]# kubectl get pods --output=wide  
NAME READY STATUS RESTARTS AGE IP NODE  
wordpress 1/1 Running 0 48s 10.47.255.250 ip-172-25-1-83.us-west-1.compute.internal

**Tip**

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

Another very important use of labels is linking PODs to Replica-set , Deployment , services …,etc and that’s what we will see next with services and label selector

# NS

## what is NS

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

this is a very common scenario in many today’s network platforms - an object name has to be unique in the same "scope":

* IP prefix in a router
* virtual network in openstack cluster
* a variable name in a program

and the solution is to support multiple scopes or namespaces, respectively: \* VRF \* project/tenant \* function

the solution in k8s is just called 'Namespaces', or 'NS' for short, which provide a scope for names. 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.

to create a NS is pretty simple, you can avoid the need to give a yaml file by using kubectl with '-f' option, followed by '-' and hit enter:

root@test3:~# kubectl create -f -

now the kubectl will wait for you to manually input the definition of NS from 'stdin', you can now input these 4 lines to create a VN:

apiVersion: v1  
kind: Namespace  
metadata:  
 name: dev

when done, press ctr-d to submit the stdin buffer content into kubectl.

namespace/development created

new namespace is now created

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

you may notice there is a NS named 'default' in any k8s setup. that is, as the name indicates, the "default" NS that will be created when you install the setup, same as the "default routing table" in a router, "default tenant" in openstack setup - there needs to be a scope providing you an initial working environment so from there you can create all other objects.

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

$ kubectl get pod --all-namespaces -o wide  
NAMESPACE NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE  
......  
dev csrx 1/1 Running 4 2d4h 10.47.255.249 cent222 <none>  
......  
sales csrx 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 ns-user-2 create quota foobar --hard pods=1

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

$ kubectl get quota -n ns-user-2  
NAME CREATED AT  
foobar 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: ns-user-2  
 resourceVersion: "823606"  
 selfLink: /api/v1/namespaces/ns-user-2/resourcequotas/foobar  
 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 rc with replica=2

$ cat rc-ubuntu.yaml  
apiVersion: v1  
kind: ReplicationController  
metadata:  
name: rc-ubuntuapp  
spec:  
 replicas: 2  
 template:  
 metadata:  
 labels:  
 run: ubuntuapp  
 spec:  
 containers:  
 - name: ubuntuapp  
 image: ubuntu-upstart  
  
$ kubectl apply -f rc-ubuntu.yaml  
replicationcontroller/rc-ubuntuapp created  
  
$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
rc-ubuntuapp-2j84g 1/1 Running 0 10s

what we desired is 2 pods, but only 1 will be created:

$ kubectl get rc  
NAME DESIRED CURRENT READY AGE  
ubuntuapp 2 1 1 3m19s

and the reason is that the 2nd pod creation is "forbidden" due to quota exceeded:

..."rc-ubuntuapp-88cxk" is forbidden: exceeded quota: foobar, requested: pods=1, used: pods=1, limited: pods=1

$ kubectl describe rc  
Name: rc-ubuntuapp  
Namespace: ns-user-2  
Selector: run=ubuntuapp  
......  
Conditions:  
 Type Status Reason  
 ---- ------ ------  
 ReplicaFailure True FailedCreate #<---  
Events:  
 Type Reason Age From Message  
 ---- ------ ---- ---- -------  
 Normal SuccessfulCreate 2m8s replication-controller Created pod: rc-ubuntuapp-2j84g  
 Warning FailedCreate 2m8s replication-controller Error creating: pods "rc-ubuntuapp-88cxk" is forbidden: exceeded quota: foobar, requested: pods=1, used: pods=1, limited: pods=1  
 Warning FailedCreate 2m8s replication-controller Error creating: pods "rc-ubuntuapp-tztv4" is forbidden: exceeded quota: foobar, requested: pods=1, used: pods=1, limited: pods=1  
 ......  
 Warning FailedCreate 77s (x6 over 2m6s) replication-controller (combined from similar events): Error creating: pods "rc-ubuntuapp-rtb56" is forbidden: exceeded quota: foobar, requested: pods=1, used: pods=1, limited: pods=1

new pod will can be create after the quota is removed:

root@test1:~# kubectl delete quota foobar  
resourcequota "foobar" deleted  
  
$ kubectl scale rc rc-ubuntuapp --replicas=3  
replicationcontroller/rc-ubuntuapp scaled  
  
$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
rc-ubuntuapp-2j84g 1/1 Running 0 8m4s  
rc-ubuntuapp-rssl9 1/1 Running 0 16s  
rc-ubuntuapp-z6cmn 1/1 Running 0 16s

# 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 I need 5 exactly the same pods (each runs a apache container) to make sure the web service appears more robust? shall I change the name in yaml file then repeat the same commands to create 5 more pods? or maybe with a shell script? kubernetes already has the objects to address this exact demand and the right answer are RC - replication controller or 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.

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: 5  
 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 5 indicates a same pod will be cloned 4 times to make sure the total number of pods created by the RC is always 5. 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

$ kubectl get rc  
NAME DESIRED CURRENT READY AGE  
myweb 5 5 5 3m29s

eventually you will see 5 pods launched:

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5ggv6 1/1 Running 0 21m  
myweb-5ww92 1/1 Running 0 22m  
myweb-lbj89 1/1 Running 0 21m  
myweb-m6nrx 1/1 Running 0 21m  
myweb-q5gv4 1/1 Running 0 22m

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.

you can test this out by deleting one of the pod:

$ 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-5ww92 1/1 Running 0 23m  
myweb-lbj89 1/1 Running 0 22m  
myweb-m6nrx 1/1 Running 0 22m  
myweb-q5gv4 1/1 Running 0 23m

$ kubectl get pod  
NAME READY STATUS RESTARTS AGE  
myweb-5v9w6 1/1 Running 0 5s  
myweb-5ww92 1/1 Running 0 23m  
myweb-lbj89 1/1 Running 0 22m  
myweb-m6nrx 1/1 Running 0 22m  
myweb-q5gv4 1/1 Running 0 23m

there are other benefits with RC. actually since this abstraction is so popular and heavily used in practice that, two very similar new objects have been designed with more powerful features introduced. The original RC object and its abstraction has been split into 2 new objects named RS(ReplicaSet) and deployment. 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 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: extensions/v1beta1  
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 while RS uses "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 sale 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 two different objects to do the almost same job. as mentioned earlier the features of RC has been extended through the 2 new objects. we’ve seen RS has done the same job of RC just with a different selector format, now we’ll check out the other new object deployment and explore the features coming from it. simply changing kind attribute from ReplicaSet to deployment we get the yaml file of a deployment object:

$ cat myweb-deployment.yaml  
apiVersion: extensions/v1beta1  
kind: Deployment #<---  
metadata:  
 name: myweb  
...(everything else remains the same as replicaset)...

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

$ kubectl get all | grep myweb  
deployment.apps/myweb 1 1 1 1 21s  
replicaset.apps/myweb-c586fd645 1 1 1 21s  
pod/myweb-c586fd645-b2ft8 1/1 Running 0 21s

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>

what happens is when you create a Deployment, a replica set is created underneath. The pods defined in a Deployment object are created and supervised by the Deployment’s replicaset. RC on the other hand, works with pod directly. the workflows differences are shown in this diagram:

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

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

You might still be wondering why you need RS as one more layer sitting in between deployment and pod, after all with RC’s magic it seems sufficient to keep a set of pods running.

the reason is about another important usage scenario in pratice: pod update. "rolling update" feature is one of the "more powerful feature" coming with deployment object. this is how it works:

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 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 the same number of pod. 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  
 |==> RS ====|=> pod  
 | (v1) |=> pod  
 |  
 | |=> pod  
 | |  
deployment ==|==> RS ====|=> pod  
 | (v2) |  
 | |=> pod  
 |  
 | |=> pod  
 |==> RS ====|=> pod  
 (v3) |=> pod

now coming back to the question in your mind. 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 along without deployment. in contrast, RC alone, without this additional higher layer abstraction, is not able to coordinate this process.

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

$ kubectl describe deployment/nginx-deployment  
Name: nginx-deployment  
......  
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 #<------

similarly, 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  
......  
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 (x2 over 1h) 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 #<------

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. Typically you do this when something is broken in your router 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! later in the book we’ll still use pod/RC/deployment to demonstrate different usage case with labs, however, it is rather unlikely that you will ever need to create Pods directly in production.

# endpoint

## service with selector

in our 'service' introduction, there is one object that is involved but we haven’t explored much yet - the endpoint. we’ve learned it is through label selector a particular pod, or more typically, a group of pods with matching labels are choosen to be the backend, so that the service request traffic will be redirected to. The IP and port information of the "matching" pods are maintained in the 'endpoint' object ('EP' as abbr.). 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 'endpoint' object will always be updated accordingly to reflect the current backend pod IPs, so the service traffic redirection will act properly.

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

verify 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-vn-right-1-rjlgr 1/1 Running 4 5d17h 10.47.255.252 cent333 <none> app=webserver

## service with no selector

in the preceding example, the 'endpoint' object is generated automatically by the kubernetes system whenever a service is created, and at least one pod with matching label exists. Another usage case of endpoint, is for a service that has no label selector defined. in that case you can explicitly create an 'endpoint', and by using the same endpoint name with the service object name, 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, or a virtual machine, and even though not running as a container in a kubernetes pod, 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 if the backend is pod or VM, it just redirect the service request traffic exactly the same way as if all backend is pod.

# Service

POD get instantiated, terminated and moved from one Node to another, in doing so POD changes IP address so how would we keep track of that? Even if the POD isn’t moving how traffic from outside reach a certain POD

the answer for both questions is Kubernetes 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

so let’s cover three different type of services Node port , ClusterIP and load balancer and we begin with Node port by creating this ( I need to create a diagram )

## clusterIP service

the clusterIP type of service is the most simple one. it is the default mode if the ServiceType is not given.

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. service "binds" itself to certain backend pods via label mapping between the two objects. endpoint is created for each service as long as there is at least one matching pod available to be its backend. this model works great if all requests are coming from the same cluster. the nature of the clusterIP limits the scope of this service to be only within the same cluster. overall by default the clusterIP is not reachable from external.

### create clusterIP service

let’s create our first service in contrail environment, 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 "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. in our example it will be pod spawned by a RC object named rc-webserver.

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, the associated endpoint, 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 ep -o wide  
NAME ENDPOINTS AGE  
service-web-lb <none> 10m  
  
$ 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 ENDPOINTS. the reason is there is no pod with the label matching to the SELECTOR in the service. this makes good sense - in chapter 3 you’ve learned what kubernetes endpoint provides is nothing but a list of backend pod with label matching the selector. 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 a RC is more pratical. later on you will understand this is the right choice.

$ 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 ep  
NAME ENDPOINTS AGE  
service-web-clusterip 10.47.255.252:80 2m58s  
  
$ 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, and endpoint reflects the update. 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 service has been associated with an "endpoint" of "10.47.255.252:80", which indicates there **is** a backend pod running with that IP, and in that pod there is a container inside of the pod running a service application (web server in our case) listening on port 80.
* 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: 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 the same cirros pod as you’ve seen in "multiple interface pod" section, and use curl command to send a http request toward the service:

$ 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 pod IP and hostname embeded in it. This way the curl returns something more meaningful to our test.

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

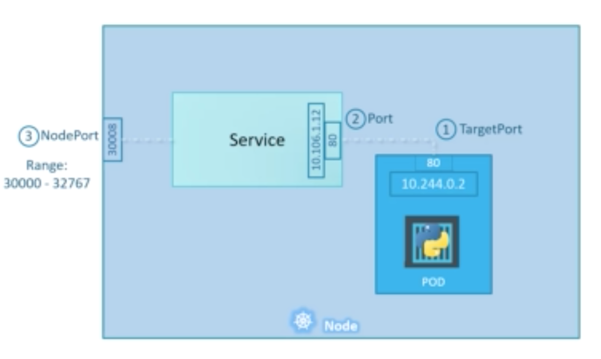
$ kubectl exec -it cirros -- curl 10.101.150.135:8888 | w3m -T text/html | head  
 Hello  
 This page is served by a Contrail pod  
 IP address = 10.47.255.238  
 Hostname = rc-webserver-vl6zs

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.

## NodePort

NodePort service is an easy way to expose an application running on POD by mapping a port in the node that host this POD with a port of the application the POD as shown in the diagram



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 the node hosting that PODs

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

**Tip**

* Kubernetes by default allocate node port from (3000-32767) range it could be change using the flag --service-node-port-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 the pod we created before with services shown after putting in web-app.yaml file

[root@ip-172-25-1-56 /]# kubectl create -f web-app.yaml  
service "web-app" created

[root@ip-172-25-1-56 /]# kubectl describe services web-app  
Name: web-app  
Namespace: default  
Labels: <none>  
Annotations: <none>  
Selector: app=webserver  
Type: NodePort  
IP: 10.98.21.191  
Port: <unset> 80/TCP  
TargetPort: 80/TCP  
NodePort: <unset> 32001/TCP  
Endpoints: 10.47.255.250:80  
Session Affinity: None  
External Traffic Policy: Cluster  
Events: <none>

Now we can test that by just send CURL -i to sent http request using the CLI

[root@computeee centos]# curl -i 10.98.21.191:80  
HTTP/1.1 200 OK  
Server: nginx/1.15.12  
Date: Tue, 14 May 2019 18:33:07 GMT  
Content-Type: text/html  
Content-Length: 612  
Last-Modified: Tue, 16 Apr 2019 13:08:19 GMT  
Connection: keep-alive  
ETag: "5cb5d3c3-264"  
Accept-Ranges: bytes

<!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 services are the type value. to reuse the same NodePort service yaml file and create a loadbalancer service, just change the type to LoadBalancer:

type: LoadBalancer

the cloud will see this keyword and a load balancer will be created, with a public IP serving as the frontend virtual IP. traffic coming to this virtual IP will be redirect to the service backend pod. because the loadbalancer VIP is publicly reachable, any client whoever has access to the VIP and port can access the service provided by kubernetes cluster.

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

# TODO: Ingress

general (non-contrail) ingress

# 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

In this book we will focus on using Annotations 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

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

2-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)

# 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 TCP socket probe example where the “initialDelaySeconds” is the waiting time before the first try to open a TCP socket 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

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

HTTP GET request probe is similar to the TCP socket probes, but it will sent HTTP GET request, and 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

## 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 report ready. Readiness Probe is configured the same way as liveness prob

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  
 readinessProbe:  
 tcpSocket:  
 port: 80  
 initialDelaySeconds: 5  
 periodSeconds: 10

**Tip**

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

# 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