RX-M Cloud Native Consulting

Kubernetes

Lab 5 – Services

Kubernetes pods are mortal. They are born and they die, and they are not resurrected. ReplicaSets create and destroy Pods dynamically (e.g. when scaling up or down or when doing rolling updates). While each pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem; if some set of backend Pods provides functionality to other frontend Pods inside the Kubernetes cluster, how do the frontends find out and keep track of the backends?

Services

A Kubernetes Service is an abstraction which defines a logical set of pods and a policy by which to access them. The set of pods targeted by a Service is determined by a label selector. As an example, consider an image-processing backend which is running with 3 replicas. Those replicas are fungible, frontends do not care which backend they use. While the actual pods that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of backends themselves. The Service abstraction enables this decoupling.

For applications integrated with the Kubernetes control plane, Kubernetes offers a simple Endpoints API that is updated whenever the set of pods in a Service changes. For Kubernetes hosted applications, Kubernetes offers a virtual-IP-based façade which redirects connections to the backend pods. For applications outside of the Kubernetes cluster, Kubernetes offers a cluster wide port forwarding feature that provides a way for external traffic to enter the pod network and reach a service's pods.

In Kubernetes, a Service is just a JSON object in etcd, similar to a Pod. Like all of the "REST" objects, a Service definition can be POSTed to the kube-apiserver to create a new service instance. The service controller then acts on service specifications reserving Cluster IPs and Node Ports as needed. This in turn causes the KubeProxy agents and/or SDN implementations to take local action on each node (modifying iptables/ipvs/etc.).

1. A Simple Service

Let's begin by creating a simple service using a service config file. Before you begin delete any services (except the kubernetes service), resource controllers, pods or other resources you may have running.

Now create a simple service called "testweb" with its own *ClusterIP* passing traffic on port 80 and configure the service to use the selector "run=testweb".

Something like this:

```
user@ubuntu:~/$ cd ~

user@ubuntu:~$ mkdir ~/svc && cd ~/svc

user@ubuntu:~/svc$ nano svc.yaml

user@ubuntu:~/svc$ cat svc.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: testweb
  labels:
    name: testweb
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    run: testweb
```

```
user@ubuntu:~/svc$
```

Now create the service:

```
user@ubuntu:~/svc$ kubectl apply -f svc.yaml

service/testweb created

user@ubuntu:~/svc$
```

> N.B. Your host, service and pod ip addresses will be different from those shown in the lab examples. Be sure to substitute the correct addresses from you lab system as you work through the lab.

List the services in your namespace:

```
user@ubuntu:~/svc$ kubectl get services

NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   156m
testweb      ClusterIP   10.98.126.85   <none>        80/TCP    14s

user@ubuntu:~/svc$ SVC=$(kubectl get service testweb -o template --template={{.spec.clusterIP}})
&& echo $SVC

10.98.126.85

user@ubuntu:~/svc$
```

Great we have a service running. Now what?

Let's try to use it. To make use of services we need to be in the pod network. Some SDN solutions provide an onramp (route) on the nodes to the pod network but the ClusterIP is virtual so it is often not available outside of running pods. Services are always available inside pods though so we'll run a pod to use as a test client.

Run a pod for testing the service:

```
user@ubuntu:~/svc$ kubectl run -it --generator=run-pod/v1 testclient --image=busybox:latest

If you don't see a command prompt, try pressing enter.

/ #
```

Try to `wget` your service's Cluster IP from inside the pod:

```
/ # wget -O - 10.98.126.85

Connecting to 10.98.126.85 (10.98.126.85:80)

wget: can't connect to remote host (10.98.126.85): Connection refused

/ #
```

We have created a service and it has an IP but the IP is virtual and there's no pod(s) for the proxy to send the traffic to. Services truly can outlive their implementations.

To fix this lack of implementation we can create a pod with a label that matches the service selector.

Leave your test pod running, open a new terminal on the host and create a simple nginx pod to support your service and run it.

```
user@ubuntu:~$ cd ~/svc

user@ubuntu:~/svc$ nano web.yaml

user@ubuntu:~/svc$ cat web.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: bigwebstuff
  labels:
    run: testweb
spec:
  containers:
  - name: web-container
    image: nginx
    ports:
    - containerPort: 80
```

```
user@ubuntu:~/svc$
```

Now run the pod:

```
user@ubuntu:~/svc$ kubectl apply -f web.yaml

pod/bigwebstuff created

user@ubuntu:~/svc$
```

With the pod up, retry the service IP from the test pod:

```
/ # wget -O - 10.98.126.85

Connecting to 10.98.126.85 (10.98.126.85:80)
writing to stdout
<!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>
-                    100%
|********************************************************************************************
**********************************************************************************************
**********************************************************************************************
```

```
**********|   612  0:00:00 ETA
written to stdout

/ #
```

On the host, describe your service to verify the wiring between the ClusterIP and the container in the pod:

```
user@ubuntu:~/svc$ kubectl describe service testweb
```

```
Name:              testweb
Namespace:         default
Labels:            name=testweb
Annotations:       kubectl.kubernetes.io/last-applied-configuration:
                     {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":
{"name":"testweb"},"name":"testweb","namespace":"default"},"spec...
Selector:          run=testweb
Type:              ClusterIP
IP:                10.98.126.85
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.32.0.5:80
Session Affinity:  None
Events:            <none>
```

```
user@ubuntu:~/svc$
```

So as you can see our nginx container must be listening on 10.32.0.5.

Use kubectl to dispaly the pod and host IPs:

```
user@ubuntu:~/svc$ kubectl get pod bigwebstuff -o json | jq -r '.status | .podIP, .hostIP'

10.32.0.5
192.168.228.157

user@ubuntu:~/svc$
```

Try hitting the pod by its pod IP from the test container:

```
/ # wget -O - 10.32.0.5

Connecting to 10.32.0.5 (10.32.0.5:80)
writing to stdout
<!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>
-                    100%
|*****************************************************************************************************
*****************************************************************************************************
*****************************************************************************************************
**********|   612  0:00:00 ETA
written to stdout

/ #
```

All we needed to do to enable our service was to create a pod with the right label. Note that if our pod container dies, no one will restart it as things now stand, but our service will carry on.

Exit the testclient container before proceeding:

```
/ # exit

Session ended, resume using 'kubectl attach testclient -c testclient -i -t' command when the pod
is running

user@ubuntu:~/svc$
```

2. Add a Resource Controller to Your Service

To improve the robustness of our service implementation we can switch from a pod to a resource controller. Change your config to instantiate a deployment which creates 3 replicas and with a template just like the pod we launched in the last step.

```
user@ubuntu:~/svc$ nano webd.yaml

user@ubuntu:~/svc$ cat webd.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bigwebstuff
  labels:
    name: bigwebstuff
spec:
  replicas: 3
  selector:
    matchLabels:
      run: testweb
  template:
    metadata:
      labels:
        run: testweb
    spec:
      containers:
      - name: podweb
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
user@ubuntu:~/svc$
```

Now create the deployment:

```
user@ubuntu:~/svc$ kubectl apply -f webd.yaml
```

```
deployment.apps/bigwebstuff created

user@ubuntu:~/svc$
```

Now let's see what happened to our pods and our service:

```
user@ubuntu:~/svc$ kubectl describe service testweb
```

```
Name:              testweb
Namespace:         default
Labels:            name=testweb
Annotations:       kubectl.kubernetes.io/last-applied-configuration:
                     {"apiVersion":"v1","kind":"Service","metadata":{"annotations":{},"labels":
{"name":"testweb"},"name":"testweb","namespace":"default"},"spec...
Selector:          run=testweb
Type:              ClusterIP
IP:                10.98.126.85
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.32.0.5:80,10.32.0.6:80,10.32.0.7:80
Session Affinity:  None
Events:            <none>
```

```
user@ubuntu:~/svc$
```

List the running Pods:

```
user@ubuntu:~/svc$ kubectl get pods

NAME                           READY   STATUS    RESTARTS   AGE
bigwebstuff                    1/1     Running   0          4m8s
bigwebstuff-6c4c6bf494-8bhgx   1/1     Running   0          60s
bigwebstuff-6c4c6bf494-bkrqv   1/1     Running   0          60s
bigwebstuff-6c4c6bf494-tx27b   1/1     Running   0          60s
testclient                     1/1     Running   1          5m30s

user@ubuntu:~/svc$
```

- What happened to our old pod?
- How many pods did the RS create?
- What does the age output in the get pods command tell you?
- What are the pods names with the suffixes from?

Service selector and Replica Set selector behavior differ in subtle ways. You will notice the Service has 4 pods, while the Replica Set has 3.

Challenge

Without changing any of the running resources, create a new deployment that runs the `httpd` image with 2 replicas such that the service will send traffic to it as well. Test your service to verify proper operation using curl in the test container.

> N.B. Apache httpd returns a different startup message than that returned by nginx

Cleanup

Delete the deployment, pods and service created in this lab using `kubectl delete`:

```
user@ubuntu:~/svc$ kubectl delete deploy bigwebstuff
```

```
deployment.apps "bigwebstuff" deleted

user@ubuntu:~/svc$ kubectl delete pods bigwebstuff testclient

pod "bigwebstuff" deleted
pod "testclient" deleted

user@ubuntu:~/svc$ kubectl delete svc testweb

service "testweb" deleted

user@ubuntu:~/svc$ cd ~

user@ubuntu:~$
```

Use `kubectl get all` to list all the remaining resources in the default namespace:

```
user@ubuntu:~$ kubectl get all

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   165m

user@ubuntu:~$
```

All that should remain is the kubernetes service.

Congratulations you have completed the lab.