Connecting Applications With Services

Exposing pods to the cluster

1. We did this in a previous example, but let’s do it once again and focus on the networking perspective. Create an nginx pod, and note that it has a container port specification:

run-my-nginx.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

2. This makes it accessible from any node in your cluster. Check the nodes the pod is running on: user

```
$ kubectl create -f run-my-nginx.yaml
$ kubectl get pods -l run=my-nginx -o wide
```

Output:

```
NAME                        READY     STATUS    RESTARTS   AG
E       IP             NODE
my-nginx-3800858182-jr4a2   1/1       Running   0          13
s       10.244.3.4     kubernetes-minion-905m
my-nginx-3800858182-kna2y   1/1       Running   0          13
s       10.244.2.5     kubernetes-minion-ljyd
```

3. Check your pods' IPs:

```
$ kubectl get pods -l run=my-nginx -o yaml | grep podIP
```

Output:

```
podIP: 10.244.3.4
podIP: 10.244.2.5
```

You should be able to ssh into any node in your cluster and curl both IPs. Note that the containers are not using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run

multiple nginx pods on the same node all using the same containerPort and access them from any other pod or node in your cluster using IP. Like Docker, ports can still be published to the host node’s interfaces, but the need for this is radically diminished because of the networking model.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

1. You can create a Service for your 2 nginx replicas with kubectl expose:

```
$ kubectl expose deployment/my-nginx
```

Output:

```
service "my-nginx" exposed
```

2. This is equivalent to `kubectl create -f` the following yaml:

nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

This specification will create a Service which targets TCP port 80 on any Pod with the run: my-nginx label, and expose it on an abstracted Service port (targetPort: is the port the container accepts traffic on, port: is the abstracted Service port, which can be any port other pods use to access the Service).

3. View service API object to see the list of supported fields in service definition. Check your Service:

```
$ kubectl get svc my-nginx
```

Output:

```
NAME       CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
my-nginx   10.0.162.149   <none>        80/TCP    21s
```

As mentioned previously, a Service is backed by a group of pods. These pods are exposed through endpoints. The Service's selector will be evaluated continuously and the results will be POSTed to an Endpoints object also named my-nginx. When a pod dies, it is automatically removed from the endpoints, and new pods matching the Service's selector will automatically get added to the endpoints.

4. Check the endpoints, and note that the IPs are the same as the pods created in the first step:

```
$ kubectl describe svc my-nginx
```

Output:

```
Name:                my-nginx
Namespace:           default
Labels:              run=my-nginx
Selector:            run=my-nginx
Type:                ClusterIP
IP:                  10.0.162.149
Port:                <unset> 80/TCP
Endpoints:           10.244.2.5:80,10.244.3.4:80
```

```
Session Affinity:    None
No events.
```

5. Following up now...

```
$ kubectl get ep my-nginx
```

Output:

```
NAME       ENDPOINTS                       AGE
my-nginx   10.244.2.5:80,10.244.3.4:80   1m
```

You should now be able to curl the nginx Service on `<CLUSTER-IP>:<PORT>` from any node in your cluster.

> *NOTE: The Service IP is completely virtual, it never hits the wire*

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the kube-dns cluster addon.

> ***Environment Variables:*** *When a Pod is run on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem.*

1. To see why, inspect the environment of your running nginx pods (your pod name will be different):

```
$ kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep S
ERVICE
```

Output:

```
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Notice there’s no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 pods and waiting for the Deployment to recreate them. This time around the Service exists before the replicas.

2. This will give you scheduler-level Service spreading of your pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
$ kubectl scale deployment my-nginx --replicas=0; kubectl sca
le deployment my-nginx --replicas=2;
```

3. And now, again:

```
$ kubectl get pods -l run=my-nginx -o wide
```

Output:

```
NAME                        READY     STATUS    RESTARTS   AG
E     IP            NODE
my-nginx-3800858182-e9ihh   1/1       Running   0          5s
      10.244.2.7    kubernetes-minion-ljyd
my-nginx-3800858182-j4rm4   1/1       Running   0          5s
      10.244.3.8    kubernetes-minion-905m
```

4. You may notice that the pods have different names, since they are killed and recreated.

```
$ kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep S
ERVICE
```

Output:

```
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
DNS
```

5. Kubernetes offers a DNS cluster addon Service that uses skydns to

automatically assign dns names to other Services. You can check if it's running on your cluster:

```
$ kubectl get services kube-dns --namespace=kube-system
```

Output:

```
NAME       CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

If it isn't running, you can enable it. The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a dns server that has assigned a name to that IP (the kube-dns cluster addon), so you can talk to the Service from any pod in your cluster using standard methods (e.g. gethostbyname).

6. Let's run another curl application to test this:

```
$ kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

Output:

```
Waiting for pod default/curl-131556218-9fnch to be running, s
tatus is Pending, pod ready: false
Hit enter for command prompt
```

7. Then, hit `enter` and run:

```
nslookup my-nginx
```

Output:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure.

For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

1. You can acquire all these from the nginx https example, in short:

```
$ make keys secret KEY=/tmp/nginx.key CERT=/tmp/nginx.crt SEC
RET=/tmp/secret.json
```

2. Then…

```
$ kubectl create -f /tmp/secret.json
```

Output:

```
secret "nginxsecret" created
```

3. Now, let’s verify our secrets, like this:

```
$ kubectl get secrets
```

Output:

```
NAME                  TYPE                                  D
ATA      AGE
default-token-il9rc   kubernetes.io/service-account-token   1
         1d
nginxsecret           Opaque                                2
         1m
```

4. Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

nginx-secure-app.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    run: my-nginx
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
```

```
      labels:
        run: my-nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The nginx server serves http traffic on port 80 and https traffic on 443, and nginx Service exposes both ports.
- Each container has access to the keys through a volume mounted at `/etc/nginx/ssl`. This is setup before the nginx server is started.

5. Okay, out with the old, in with the new:

```
$ kubectl delete deployments,svc my-nginx; kubectl create -f
./nginx-secure-app.yaml
```

6. At this point you can reach the nginx server from any node.

```
$ kubectl get pods -o yaml | grep -i podip
```

Output:

```
    podIP: 10.244.3.5
```

7. Reach out to test the results of our work, using curl, like this:

```
node $ curl -k https://10.244.3.5
```

Output:

```
...
<h1>Welcome to nginx!</h1>
```

> *NOTE: How we supplied the `-k` parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup.*

8. Let’s test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

curlpod.yaml

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: curl-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: curlpod
        command:
        - sh
        - -c
        - while true; do sleep 1; done
        image: radial/busyboxplus:curl
        volumeMounts:
        - mountPath: /etc/nginx/ssl
```

```
          name: secret-volume
```

9. Now, let's create it:

```
$ kubectl create -f ./curlpod.yaml
```

10. Verify:

```
$ kubectl get pods -l app=curlpod
```

Output:

```
NAME                                READY     STATUS    RESTAR
TS   AGE
curl-deployment-1515033274-1410r   1/1       Running   0
     1m
```

11. Execute our command inside now:

```
$ kubectl exec curl-deployment-1515033274-1410r -- curl https
://my-nginx --cacert /etc/nginx/ssl/nginx.crt
```

Output:

```
<title>Welcome to nginx!</title>
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers.

1. The Service created in the last section already used NodePort, so your nginx https replica is ready to serve traffic on the internet if your node has a public IP.

```
$ kubectl get svc my-nginx -o yaml | grep nodePort -C 5
```

Output:

```
uid: 07191fb3-f61a-11e5-8ae5-42010af00002
spec:
  clusterIP: 10.0.162.149
  ports:
  - name: http
    nodePort: 31704
    port: 8080
    protocol: TCP
    targetPort: 80
  - name: https
    nodePort: 32453
    port: 443
    protocol: TCP
    targetPort: 443
  selector:
```

```
    run: my-nginx
```

2. Now, do a get on our nodes and pipe it to grep, like so:

```
$ kubectl get nodes -o yaml | grep ExternalIP -C 1
```

Output:

```
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...
```

3. Work a little curl magic, follow closely:

```
$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
```

Output:

```
...
<h1>Welcome to nginx!</h1>
```

4. Let’s now recreate the Service to use a cloud load balancer, just

change the Type of my-nginx Service from NodePort to LoadBalancer:

```
$ kubectl edit svc my-nginx
```

5. Let’s do a get on our Service(svc), like this:

```
$ kubectl get svc my-nginx
```

Output:

```
NAME       CLUSTER-IP     EXTERNAL-IP        PORT(S)
      AGE
my-nginx   10.0.162.149   162.222.184.144    80/TCP,81/TCP,82
/TCP  21s
```

6. Verify with a the curl command:

```
$ curl https://<EXTERNAL-IP> -k
```

Output:

```
...
<title>Welcome to nginx!</title>
```

The IP address in the `EXTERNAL-IP` column is the one that is available on the public internet. The `CLUSTER-IP` is only available inside your

cluster/private cloud network.

> *NOTE: That on AWS, type LoadBalancer creates an ELB, which uses a (long) hostname, not an IP.*

7. It's too long to fit in the standard `kubectl get svc` output, in fact, so you'll need to do `kubectl describe service my-nginx` to see it. You'll see something like this:

```
$ kubectl describe service my-nginx
```

Output:

```
LoadBalancer Ingress:   a320587ffd19711e5a37606cf4a74574-1142
138393.us-east-1.elb.amazonaws.com
```