Exercise 7.1: Exposing Applications: Expose a Service

Overview

In this lab we will explore various ways to expose an application to other pods and outside the cluster. We will add to the NodePort used in previous labs other service options.

1. We will begin by using the default service type ClusterIP. This is a cluster internal IP, only reachable from within the cluster. Begin by viewing the existing services.

```
student@cp:~$ kubectl get svc
```

```
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        8d
nginx        ClusterIP   10.108.95.67     <none>        443/TCP        8d
registry     ClusterIP   10.105.119.236   <none>        5000/TCP       8d
secondapp    NodePort    10.111.26.8      <none>        80:32000/TCP   7h
```

2. Save then delete the existing service for secondapp. Ensure the same labels, ports, and protocols are used.

```
student@cp:~$ cd $HOME/app2

student@cp:~/app2$ kubectl get svc secondapp -o yaml > oldservice.yaml

student@cp:~/app2$ cat oldservice.yaml

student@cp:~/app2$ kubectl delete svc secondapp
```

```
service "secondapp" deleted
```

3. Recreate the service using a new YAML file. Use the same selector as the previous pod. Examine the new service after creation, note the `TYPE` and `PORT(S)`.

```
student@cp:~/app2$ vim newservice.yaml
```

newservice.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: secondapp
5  spec:
6    ports:
7    - port: 80
8      protocol: TCP
9    selector:
10     example: second
11   sessionAffinity: None
12 status:
13   loadBalancer: {}
```

```
student@cp:~/app2$ kubectl create -f newservice.yaml
```

```
service/secondapp created
```

```
student@cp:~/app2$ kubectl get svc
```

```
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)      AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP      8d
nginx        ClusterIP   10.108.95.67     <none>        443/TCP      8d
registry     ClusterIP   10.105.119.236   <none>        5000/TCP     8d
secondapp    ClusterIP   10.98.148.52     <none>        80/TCP       14s
```

4. Test access. You should see the default welcome page again.

```
student@cp:~/app2$ curl http://10.98.148.52
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

5. Now create another simple web server. This time use the **httpd** image as the default page is different from **nginx**. Once we are sure it is running we will edit the service selector to point at the new server then back, which could be used as a deployment strategy. Commands have been omitted. You should be able to complete the steps. Refer to previous content otherwise.

```
student@cp:~/app2$ kubectl create deployment newserver --image=httpd
```

6. Locate the `newserver` labels.
7. Use **kubectl edit** to change the service to use `newserver`'s labels as the selector.
8. Test that the service now shows the new content and not the default **nginx** page.

```
student@cp:~/app2$ curl http://10.98.148.52
```

```
<html><body><h1>It works!</h1></body></html>
```

9. Edit the selector back to the nginx server and test. Then remove the `newserver` deployment.
10. To expose a port to outside the cluster we will create a NodePort. We had done this in a previous step from the command line. When we create a NodePort it will create a new ClusterIP automatically. Edit the YAML file again. Add `type: NodePort`. Also add the high-port to match an open port in the firewall as mentioned in the previous chapter. You'll have to delete and re-create as the existing IP is immutable. The NodePort will create a new ClusterIP.

```
student@cp:~/app2$ vim newservice.yaml
```

newservice.yaml

```
1 ....
2     protocol: TCP
3     nodePort: 32000          #<-- Add this and following line
4   type: NodePort
5   selector:
6     example: second
```

```
student@cp:~/app2$ kubectl delete svc secondapp ; kubectl create  -f newservice.yaml
```

```
service "secondapp" deleted
service/secondapp created
```

11. Find the new ClusterIP and ports for the service.

```
student@cp:~/app2$ kubectl get svc
```

```
NAME         TYPE         CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP    10.96.0.1       <none>        443/TCP        8d
nginx        ClusterIP    10.108.95.67    <none>        443/TCP        8d
registry     ClusterIP    10.105.119.236  <none>        5000/TCP       8d
secondapp    NodePort     10.109.134.221  <none>        80:32000/TCP   4s
```

12. Test the low port number using the new ClusterIP for the secondapp service.

```
student@cp:~/app2$ curl 10.109.134.221
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

13. Test access from an external node to the host IP and the high container port. Your IP and port will be different. It should work, even with the network policy in place, as the traffic is arriving via a 192.168.0.0 port. If you don't have a terminal on your local system use a browser.

```
student@cp:~/app2$ curl ifconfig.io
```

```
35.184.219.5
```

```
user@laptop:~/Desktop$ curl http://35.184.219.5:32000
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

14. The use of a LoadBalancer makes an asynchronous request to an external provider for a load balancer if one is available. It then creates a NodePort and waits for a response including the external IP. The local NodePort will work even before the load balancer replies. Edit the YAML file and change the type to be LoadBalancer.

```
student@cp:~/app2$ vim newservice.yaml
```

newservice.yaml

```
1 ....
2   - port: 80
3     protocol: TCP
4     nodePort: 32000
5   type: LoadBalancer    #<-- Edit this line
6   selector:
7     example: second
```

```
student@cp:~/app2$ kubectl delete svc secondapp ; kubectl create -f newservice.yaml
```

```
service "secondapp" deleted
service/secondapp created
```

15. As mentioned the cloud provider is not configured to provide a load balancer; the External-IP will remain in pending state. Some issues have been found using this with VirtualBox.

```
student@cp:~/app2$ kubectl get svc
```

```
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP      10.96.0.1        <none>        443/TCP        8d
nginx        ClusterIP      10.108.95.67     <none>        443/TCP        8d
registry     ClusterIP      10.105.119.236   <none>        5000/TCP       8d
secondapp    LoadBalancer   10.109.26.21     <pending>     80:32000/TCP   4s
```

16. Test again local and from a remote node. The IP addresses and ports will be different on your node.

```
serewic@laptop:~/Desktop$ curl http://35.184.219.5:32000
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

17. You can also use DNS names provided by **CoreDNS** which dynamically are added when the service is created. Start by logging into the `busy` container of secondapp.

```
student@cp:~/app2$ kubectl exec -it secondapp -c busy -- sh
```

On Container

(a) Use the **nslookup** command to find the `secondapp` service. Then find the `registry` service we configured to provide container images. If you don't get the expected output try again. About one out of three requests works.

```
/ $ nslookup secondapp
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        secondapp.default.svc.cluster.local
Address: 10.96.214.133

*** Can't find secondapp.svc.cluster.local: No answer
*** Can't find secondapp.cluster.local: No answer
*** Can't find secondapp.c.endless-station-188822.internal: No answer
<output_omitted>
```

```
/ $ nslookup registry
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        registry.default.svc.cluster.local
Address: 10.110.95.21
<output_omitted>
```

(b) Lookup the FQDN associated with the DNS server IP displayed by the commands. Your IP may be different.

```
/ $ nslookup 10.96.0.10
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

10.0.96.10.in-addr.arpa        name = kube-dns.kube-system.svc.cluster.local
```

(c) Attempt to resolve the service name, which should not bring back any records. Then try with the FQDN. Read through the errors. You'll note that only the default namespaces is checked. You may have to check the FQDN a few times as it doesn't always reply with an answer.

```
/ $ nslookup kube-dns
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find kube-dns.default.svc.cluster.local: NXDOMAIN

*** Can't find kube-dns.svc.cluster.local: No answer
*** Can't find kube-dns.cluster.local: No answer
*** Can't find kube-dns.c.endless-station-188822.internal: No answer
```

```
/ $ nslookup kube-dns.kube-system.svc.cluster.local
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        kube-dns.kube-system.svc.cluster.local
Address: 10.96.0.10

*** Can't find kube-dns.kube-system.svc.cluster.local: No answer
```

(d) Exit out of the container

```
/ $ exit
```

18. Create a new namespace named `multitenant` and a new deployment named `mainapp`. Expose the deployment port 80 using the name `shopping`

```
student@cp:~/app2$ kubectl create ns multitenant
```

```
namespace/multitenant created
```

```
student@cp:~/app2$ kubectl -n multitenant create deployment mainapp --image=nginx
```

```
deployment.apps/mainapp created
```

```
student@cp:~/app2$ kubectl -n multitenant expose deployment mainapp --name=shopping \
                        --type=NodePort --port=80
```

```
service/shopping exposed
```

19. Log back into the secondapp busy container and test access to mainapp.

```
student@cp:~/app2$ kubectl  exec -it secondapp -c busy -- sh
```

On Container

(a) Use **nslookup** to determine the address of the new service. Start with using just the service name. Then add the service name and the namespaces. There are a few hiccups, with how busybox and other applications interact with CoreDNS. Your responses may or may not work. Try each a few times.

```
/ $ nslookup shopping
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find shopping.default.svc.cluster.local: NXDOMAIN

*** Can't find shopping.svc.cluster.local: No answer
<output_omitted>
```

```
/ $ nslookup shopping.multitenant
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find shopping.multitenant: NXDOMAIN

*** Can't find shopping.multitenant: No answer
```

```
/ $ nslookup shopping.multitenant.svc.cluster.local
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        shopping.multitenant.svc.cluster.local
Address: 10.101.4.142

*** Can't find shopping.multitenant.svc.cluster.local: No answer
```

(b) Now try to use the service name and then the name with namespace, to see if it works. The DNS using the namespace should work, even if you don't have access to the default page. RBAC could be used to grant access. Check the service ClusterIP returned and it will match the newly created service.

```
/ $ wget shopping
```

```
wget: bad address 'shopping'
```

```
/ $ wget shopping.multitenant
```

```
Connecting to shopping.multitenant (10.101.4.142:80)
wget: can't open 'index.html': Permission denied
```

(c) As we can see the error is about permissions we will try again, but not try to write locally, but instead to dash (-), which is standard out.

```
~ $ wget -O - shopping.multitenant
```

```
Connecting to shopping.multitenant (10.103.211.64:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>

....

</html>
-                    100% |********************************************|   612
↪  0:00:00 ETA
written to stdout
```