**Learning Kubernetes on EKS by Doing Part 3— Services**

**Introduction**

In Kubernetes, a Service is an abstraction of a logical set of Pods and a policy used to access them. Services use *Label Selectors*to identify the set of Pods.

Although each Pod has a unique IP address, these IPs are not exposed outside the cluster without a Service. Any time we need to provide access to one or more pods to another application or for external access, we should create and configure a Service.

There are several Service types in Kubernetes.

* **ClusterIP**: This is the default Service Type, only reachable from within the cluster. Consider this as an internal load balancer.
* **NodePort**: The NodePort opens a static port on each node’s external networking interface. A ClusterIP is also automatically created when we create a NodePort Service. External traffic to the port will be routed to the internal cluster IP. We can use the combination of “NodeIP : NodePort” to call the NodePort Service from outside the cluster.
* **LoadBalancer**: The LoadBalancer Service type creates a cloud provider’s external load balancer. The NodePort and ClusterIP will also be created automatically when we create a LoadBalancer Service. External traffic to the load balancer will be routed to the NodePort which in turn will be routed to internal ClusterIP.

**ClusterIP Service**

We are using the same nginx Deployment template.

**deployment.yaml**apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: nginx-deployment  
 labels:  
 app: nginx  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: nginx  
 template:  
 metadata:  
 labels:  
 app: nginx  
 spec:  
 containers:  
 - name: nginx  
 image: nginx:1.9.1  
 ports:  
 - containerPort: 80

We will create a deployment using the template

$ **kubectl create -f deployment.yaml**  
deployment.apps/nginx-deployment created

We then create a ClusterIP Service to let this Deployment be exposed to other internal applications within the cluster. Notice that the selector in this Service spec is **app : nginx**. It is the same as in the Deployment selector’s matchLabels.

**clusterip.yaml**apiVersion: v1  
kind: Service  
metadata:  
 name: my-service  
spec:  
 type: **ClusterIP**  
 selector:  
 app: nginx  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 80

We will create a ClusterIP using the template

$ **kubectl create -f clusterip.yaml**  
service/myservice created

Check to see that Pods are created and running and that each Pod has its own internal IP address.

$ **kubectl get pods -o wide | awk {'print $1" " $3 " " $6'} | column -t**  
NAME STATUS IP  
nginx-deployment-6fdbb596db-cfv9w Running 192.168.28.26  
nginx-deployment-6fdbb596db-czxhq Running 192.168.19.32  
nginx-deployment-6fdbb596db-q6gfs Running 192.168.68.220

Check to see that the Service was created. Notice that the Selector is **app=nginx**, Type is ClusterIP and IP address is internal IP 10.100.168.154. This serves as an internal load balancer for three Pod IPs.

$ **kubectl describe service myservice**  
Name: myservice  
Namespace: default  
Labels: <none>  
Annotations: <none>  
Selector: app=nginx  
Type: ClusterIP  
IP: 10.100.168.154  
Port: <unset> 80/TCP  
TargetPort: 80/TCP  
Endpoints: 192.168.19.32:80,192.168.28.26:80,192.168.68.220:80

The ClusterIP Service is created and named “myservice”. It can be accessed by other Pods in the same cluster, but **can not be accessed externally** outside of the cluster.

We can create a new Pod within the same Kubernetes cluster and test if we can access “myservice” from this Pod.

Here we start and log in a new Pod running ubuntu Linux.

$ **kubectl run my-test --rm -i --tty --image ubuntu -- bash**root@my-test-5795787c7d-b6h92:/#

From prompt, we first install curl and then we can test accessing myservice.

root@my-test-5795787c7d-b6h92:/# apt-get update && apt-get install curl  
root@my-test-5795787c7d-b6h92:/# **curl myservice**  
<!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>

Or accessing the internal IP address of the ClusterIP Service.

root@my-test-5795787c7d-b6h92:/# **curl --silent 10.100.70.238:80 | grep title**  
<title>Welcome to nginx!</title>

**NodePort Service**

To create a NodePort Service, we can use a similar template as the clusterip.yaml file. We will create a file called “nodeport.yaml” with type as “NodePort”.

apiVersion: v1  
kind: Service  
metadata:  
 name: my-service  
spec:  
 type: **NodePort**  
 selector:  
 app: nginx  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 80

Let’s delete the ClusterIP Service and create a NodePort Service with the same server name — “myservice”.

$ kubectl delete service myservice  
service "myservice" deleted  
$ **kubectl create -f nodeport.yaml**  
service/myservice created

We can get information about “myservice”

$ **kubectl get service/myservice**  
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE  
myservice NodePort 10.100.238.34 <none> 80:32009/TCP 1h

The Service type is a NodePort. Notice that a ClusterIP is also created automatically which takes the route from the NodePort.

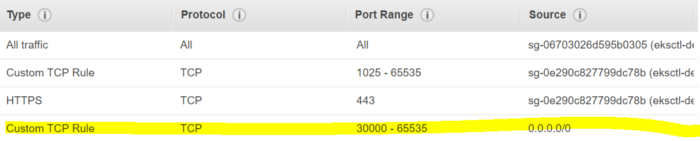
The NodePort Service is**exposed externally** on the available worker nodes at port 32009.

We can check the node’s public IP address.

$ **kubectl get nodes -o wide | awk {'print $1" " $2 " " $7'} | column -t**  
NAME STATUS EXTERNAL-IP  
ip-192-168-7-94.us-west-2.compute.internal Ready 34.214.98.130  
ip-192-168-74-68.us-west-2.compute.internal Ready 52.25.93.45

Before we test accessing NodeIP:NodePort from an outside cluster, we need to modify the security group of the nodes to **allow incoming traffic** through the port 32009.

Login to the AWS console and find the nodes, editing its security group by adding one more inbound rule(in yellow).



Now the NodePort Service can be accessed from an outside cluster by using NodeIP:NodePort on any available nodes.

$ **curl --silent 34.214.98.130:32009 | grep title**  
<title>Welcome to nginx!</title>$ **curl --silent 52.25.93.45:32009 | grep title**  
<title>Welcome to nginx!</title>

**LoadBalancer Service**

In a cloud environment like AWS, creating an external load balancer is better than creating a NodePort for exposing Services.

We can create a load balancer Service by using the same template except changing the type to “LoadBalancer”.

apiVersion: v1  
kind: Service  
metadata:  
 name: myservice  
spec:  
 type: LoadBalancer  
 selector:  
 app: nginx  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 80

We can then create a load balancer Service.

$ kubectl delete service myservice  
service "myservice" deleted  
$ **kubectl create -f loadbalancer.yaml**  
service/myservice created

We can get information about “myservice”

$ **kubectl get service/myservice | awk {'print $1" " $2 " " $4 " " $5'} | column -t**  
NAME TYPE EXTERNAL-IP PORT(S)  
myservice LoadBalancer adb010d2006b511e99d7702abbb5e7b9-679524454.us-west-2.elb.amazonaws.com 80:31099/TCP

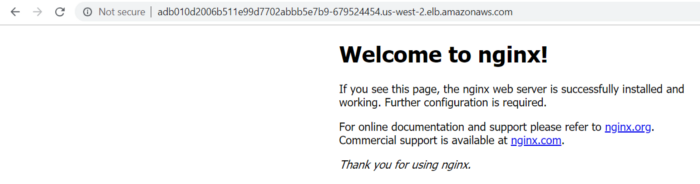
From the AWS console, we can see that the classic load balancer is created.



Now, we can test if we can access the ELB externally.

$ **curl -silent adb010d2006b511e99d7702abbb5e7b9-679524454.us-west-2.elb.amazonaws.com:80 | grep title**  
<title>Welcome to nginx!</title>

We can also access the ELB from a browser



**Conclusion**

A ClusterIP is a Service that works as an internal load balancer for related Pods. A ClusterIP is not able to be accessed directly from outside of the Kubernetes cluster without a NodePort or a LoadBalancer. NodePort and LoadBalancer are Services to accept requests from external applications.

Next, we will discuss **Kubernetes Ingress using ALB Ingress Controller**which can allow us to share one ALB for multiple Services.

**Other Parts of this blog series :**

**Learning Kubernetes on EKS by Doing Part 4— Ingress**

In this blog we are going to discuss Kubernetes Ingress in AWS using [**AWS ALB Ingress Controller**](https://aws.amazon.com/about-aws/whats-new/2018/11/amazon-eks-adds-alb-support-with-aws-alb-ingress-controller/)**.**

*Kubernetes Ingress exposes HTTP and HTTPS routes from outside the cluster to Services within the cluster. Traffic routing is controlled by rules defined on the ingress resource.*

Creating Kubernetes Ingress by using ALB Ingress Controller allows multiple Kubernetes Services to share one ALB to expose to internet.

We will explain the setup using these steps.

(1) Deploy ALB Ingress Controller on EKS.

(2) Deploy two demo applications.

(3) Create two NodePort service.

(4) Create Ingress using host based routing of ALB.

**Deploying ALB Ingress Controller on EKS**

1. Setup an EKS from [Part 1](https://medium.com/@zhaimo/learning-kubernetes-by-doing-part-1-setting-up-eks-in-aws-50dcf7a76247), creating an EKS cluster named “demo-eks-cluster”
2. Download the ALB Ingress manifest update cluster-name with EKS cluster name in alb-ingress-controller.yaml

$ wget https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/v1.0.1/docs/examples/alb-ingress-controller.yaml$ wget https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/v1.0.1/docs/examples/rbac-role.yaml

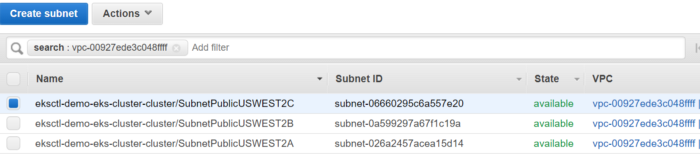
Update cluster-name with the EKS cluster name in alb-ingress-controller.yaml

- --cluster-name=demo-eks-cluster

3. Login to the AWS console. Then, create an IAM policy and manually update the EKS subnets for auto discovery.

In the IAM console, create a policy using the contents of the template [iam-policy.json](https://raw.githubusercontent.com/kubernetes-sigs/aws-alb-ingress-controller/v1.0.0/docs/examples/iam-policy.json). Attach the IAM policy to the EKS worker nodes role.

Login to the AWS console and find the public subnets created when setting up EKS.



In the tagging section of these public subnets, add these tags with values.

kubernetes.io/cluster/demo-eks-cluster : shared  
kubernetes.io/role/elb : ''  
kubernetes.io/role/internal-elb : ''

4. Deploy the alb ingress controller using modified alb-ingress-controller.yaml file.

$ **kubectl apply -f rbac-role.yaml**  
clusterrole.rbac.authorization.k8s.io/alb-ingress-controller created  
clusterrolebinding.rbac.authorization.k8s.io/alb-ingress-controller created  
serviceaccount/alb-ingress created  
$ **kubectl apply -f alb-ingress-controller.yaml**  
deployment.apps/alb-ingress-controller created

**Deploying two sample applications**

We will create two independent Deployments, both will expose to internet using the same ALB but different target group.

First deployment, nginx-deployment, for nginx app.

deployment.yamlapiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: nginx-deployment  
 labels:  
 app: nginx  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: nginx  
 template:  
 metadata:  
 labels:  
 app: nginx  
 spec:  
 containers:  
 - name: nginx  
 image: nginx:1.9.1  
 ports:  
 - containerPort: 80

Second deployment, helloapp-deployment, for helloapp app.

deployment-helloapp.yamlapiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: helloapp-deployment  
 labels:  
 app: helloapp  
spec:  
 replicas: 3  
 selector:  
 matchLabels:  
 app: helloapp  
 template:  
 metadata:  
 labels:  
 app: helloapp  
 spec:  
 containers:  
 - name: helloapp  
 image: gcr.io/google-samples/hello-app:1.0  
 ports:  
 - containerPort: 8080

**Templates for two NodePort services**

myservice for nginx app.

nodeport.yamlapiVersion: v1  
kind: Service  
metadata:  
 name: myservice  
spec:  
 type: NodePort  
 selector:  
 app: nginx  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 80  
~

helloapp-service for helloapp app.

nodeport-helloapp.yamlapiVersion: v1  
kind: Service  
metadata:  
 name: helloapp-service  
spec:  
 type: NodePort  
 selector:  
 app: helloapp  
 ports:  
 - protocol: TCP  
 port: 80  
 targetPort: 8080

**Creating an Ingress using host based routing of ALB**

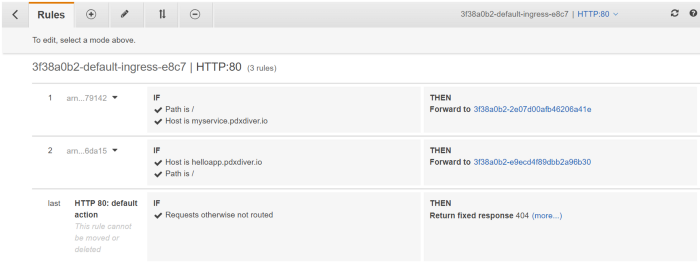
To create the Ingress template, we are using host based routing of ALB. In this example, pdxdiver.io is a public domain registered by AWS route 53 registrar in our AWS account. In your case you may choose your own domain.

ingress-demoapp.yamlapiVersion: extensions/v1beta1  
kind: **Ingress**  
metadata:  
 name: "ingress"  
 annotations:  
 kubernetes.io/ingress.class: alb  
 alb.ingress.kubernetes.io/scheme: internet-facing  
 labels:  
 app: demoapp  
spec:  
 rules:  
 **- host: myservice.pdxdiver.io**  
 http:  
 paths:  
 - path: /  
 backend:  
 serviceName: myservice  
 servicePort: 80  
 - **host: helloapp.pdxdiver.io**  
 http:  
 paths:  
 - path: /  
 backend:  
 serviceName: helloapp-service  
 servicePort: 80

Checking Ingress created

$ kubectl describe ing ingress  
Name: ingress  
Namespace: default  
Address: 3f38a0b2-default-ingress-e8c7-693678468.us-west-2.elb.amazonaws.com  
Default backend: default-http-backend:80 (<none>)  
Rules:  
 Host Path Backends  
 ---- ---- --------  
 myservice.pdxdiver.io  
 / myservice:80 (<none>)  
 helloapp.pdxdiver.io  
 / helloapp-service:80 (<none>)  
Annotations:  
 alb.ingress.kubernetes.io/scheme: internet-facing  
 kubernetes.io/ingress.class: alb  
Events: <none>

Checking listener on ALB created, we can see HTTP routes to two different hosts are forwarding to different target groups.



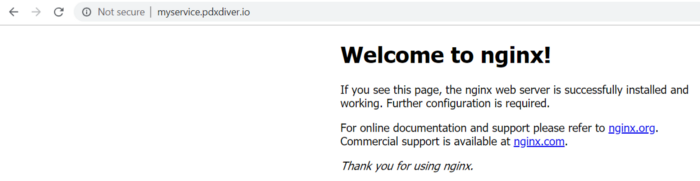
**Configure Route 53** for myservice.pdxdiver.io and helloapp.pdxdiver.io.

Login to Route 53 and find existing public domain pdxdiver.io in “Hosted Zones”. Choose “Create Record Set” and set name as “myservice.pdxdiver.io”. Choose type “A”, alias “Yes” and then choose the ALB created by Ingress controller. In this example, it is “3f38a0b2-default-ingress-e8c7–693678468.us-west-2.elb.amazonaws.com”.

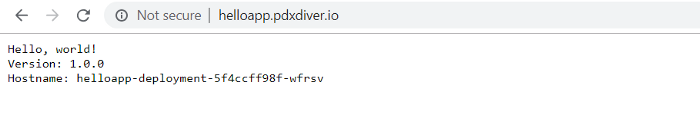
Create a second Record set for helloapp-service using the same steps as above.

Now go to a browser to test myservice.pdxdiver.io and helloapp.pdxdiver.io. We can see nginx and helloapp both running as expected.

**myservice.pdxdiver.io**



**helloapp.pdxdiver.io**



**Conclusion**

By using ALB Ingress Controller, we can consolidate multiple external exposing services under one ALB. This not only can save costs but also can take advantage of ALB to allow path and host based routing.