# Exposing Kubernetes workloads

## service & labels

To expose a pod you should create a service. We will use the earlier created pods. With the service you will create a clusterIP to forward traffic to the different pods. To create a service use:

kubectl create service clusterip example --tcp=80:80  
service "example" created

Now you have a service with a cluster IP created. The service will direct all traffic send to its cluster IP to the pod with a label app=example.

To view the details of this service use:

$ kubectl describe service example  
Name: example  
Namespace: platform  
Labels: app=example  
Selector: app=example  
Type: ClusterIP  
IP: 10.0.19.127  
Port: 80-80 80/TCP  
Endpoints: <none>  
Session Affinity: None  
No events.

As you can see no Endpoints are present yet. This is because no pods exist with the label app=example. To direct traffic from this service to the correct pod you should add this label to the pod example. Do this the following command:

$ kubectl label pod example app=example  
pod "example" labeled

Now when you do a describe of the service example you should see an endpoint belonging to this service:

$ kubectl describe service example  
Name: example  
Namespace: platform  
Labels: app=example  
Selector: app=example  
Type: ClusterIP  
IP: 10.0.19.127  
Port: 80-80 80/TCP  
Endpoints: 10.47.0.14:80  
Session Affinity: None  
No events.

For other pods within this namespace the example pod can be reached by the name example.

A service can front more than one pods. It actually fronts every pod with matching labels. We can for example create a second pod and add the same labels to it:

$ kubectl run second-example-pod --image=nginx --port 80 --restart=Never  
pod "second-example-pod" created

When we also add the same label to this pod, our service will also note this pod as an endpoint:

$ kubectl label pod second-example-pod app=example  
pod "second-example-pod" labeled  
  
$ kubectl describe service example  
Name: example  
Namespace: platform  
Labels: app=example  
Selector: app=example  
Type: ClusterIP  
IP: 10.0.19.127  
Port: 80-80 80/TCP  
Endpoints: 10.35.0.2:80,10.47.0.14:80  
Session Affinity: None  
No events.

You can see that there are now 2 endpoints behind this service. The service will loadbalance between those endpoints.

## ingresses

To make a service available outside the cluster we can either use a nodeport or an ingress. Nodeports have the disadvantage that you need to keep record of the ports you use to expose a service. Each nodeport is exposed on every node on the cluster. A better way is using ingresses. An ingress uses name based or path based http routing to send the traffic to the correct service within the cluster.

First we create a file named example-ingress.yml containing:

apiVersion: networking.k8s.io/v1beta1  
kind: Ingress  
metadata:  
 name: example  
spec:  
 rules:  
 - http:  
 paths:  
 - path: /  
 backend:  
 serviceName: example  
 servicePort: 80

Take a good look at this configuration

And create this ingress using:

$ kubectl apply -f example-ingress.yml  
ingress "example" created

To so the detailed information about the ingress you created you can use the kubectl describe command:

# kubectl describe ingress example  
Name: example  
Namespace: default  
Address: 192.168.1.101  
Default backend: default-http-backend:80 (<none>)  
Rules:  
 Host Path Backends  
 ---- ---- --------  
 \*  
 / example:80 (10.42.0.23:80,10.42.0.29:80,10.42.0.30:80 + 1 more...)  
Annotations:  
 kubectl.kubernetes.io/last-applied-configuration: {"apiVersion":"networking.k8s.io/v1beta1","kind":"Ingress","metadata":{"annotations":{},"name":"example","namespace":"default"},"spec":{"rules":[{"http":{"paths":[{"backend":{"serviceName":"example","servicePort":80},"path":"/"}]}}]}}  
  
Events: <none>

You can see the address where the ingress is exposed. This is the IP adress of the server. So now you can reach the website with curl [IP address] or with the browser pointing towards the public ip of the server. Note! it's required that there is an open firewall port between you browser and the server.

We can clean up the pods and services by deleting them:

$ kubectl delete pod example  
pod "example" deleted  
  
$ kubectl delete pod second-example-pod  
pod "second-example-pod" deleted  
  
$ kubectl delete service example  
service "example" deleted

#### Extra exercise

Create a service for the created MySQL deployment which is only reachable from the internal cluster