Exercise 6.4: Implement a NetworkPolicy

An early architecture decision with Kubernetes was non-isolation, that all pods were able to connect to all other pods and nodes by design. In more recent releases the use of a `NetworkPolicy` allows for pod isolation. The policy only has effect when the network plugin, like **Project Calico**, are capable of honoring them. If used with a plugin like **flannel** they will have no effect. The use of `matchLabels` allows for more granular selection within the namespace which can be selected using a `namespaceSelector`. Using multiple labels can allow for complex application of rules. More information can be found here: https://kubernetes.io/docs/concepts/services-networking/network-policies

1. Begin by creating a default policy which denies all traffic. Once ingested into the cluster this will affect every pod not selected by another policy, creating a mostly-closed environment. If you want to only deny ingress or egress traffic you can remove the other `policyType`.

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

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

allclosed.yaml

```
1  apiVersion: networking.k8s.io/v1
2  kind: NetworkPolicy
3  metadata:
4    name: deny-default
5  spec:
6    podSelector: {}
7    policyTypes:
8    - Ingress
9    - Egress
```

2. Before we can test the new network policy we need to make sure network access works without it applied. Update **secondapp** to include a new container running **nginx**, then test access. Begin by adding two lines for the **nginx** image and name `webserver`, as found below. It takes a bit for the pod to terminate, so we'll delete then edit the file.

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

```
  pod "secondapp" deleted
```

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

second.yaml

```
1  ....
2  spec:
3       serviceAccountName: secret-access-sa
4       securityContext:
5         runAsUser: 1000
6       containers:
7       - name: webserver                    #<-- Add this and following line
8         image: nginx
9       - name: busy
10        image: busybox
```

```
11          command:
12  ....
```

3. Create the new pod. Be aware the pod will move from `ContainerCreating` to Error to `CrashLoopBackOff`, as only one of the containers will start. We will troubleshoot the error in following steps.

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

```
pod/secondapp created
```

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

```
NAME                        READY   STATUS             RESTARTS   AGE
nginx-6b58d9cdfd-9fnl4      1/1     Running            1          2d
Registry-795c6c8b8f-hl5wf   1/1     Running            2          2d
secondapp                   1/2     CrashLoopBackOff   1          13s
<output_omitted>
```

4. Take a closer look at the events leading up to the failure. The images were pulled and the container was started. It was the full execution of the container which failed.

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

```
<output_omitted>
25s         Normal    Scheduled   Pod    Successfully assigned default/secondapp to cp
4s          Normal    Pulling     Pod    pulling image "nginx"
2s          Normal    Pulled      Pod    Successfully pulled image "nginx"
2s          Normal    Created     Pod    Created container
2s          Normal    Started     Pod    Started container
23s         Normal    Pulling     Pod    pulling image "busybox"
21s         Normal    Pulled      Pod    Successfully pulled image "busybox"
21s         Normal    Created     Pod    Created container
21s         Normal    Started     Pod    Started container
1s          Warning   BackOff     Pod    Back-off restarting failed container
```

5. View the logs of the **webserver** container mentioned in the previous output. Note there are errors about the user directive and not having permission to make directories.

```
student@cp:~/app2$ kubectl logs secondapp webserver
```

```
2018/04/13 19:51:13 warn 1 the "user" directive makes sense
only if the cp process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
nginx: warn the "user" directive makes sense only if the cp
process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
2018/04/13 19:51:13 emerg 11: mkdir\(\) "/var/cache/nginx/client temp"
failed (13: Permission denied)
nginx: emerg mkdir() "/var/cache/nginx/client\_temp" failed
(13: Permission denied)
```

6. Delete the pods. Edit the YAML file to comment out the setting of a UID for the entire pod.

```
student@cp:~/app2$ kubectl delete -f second.yaml
```

```
pod "secondapp" deleted
```

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

```
spec:
  serviceAccountName: secret-access-sa
#  securityContext:                        #<-- Comment this and following line
#    runAsUser: 1000
  containers:
  - name: webserver
```

7. Create the pod again. This time both containers should run. You may have to wait for the previous pod to fully terminate, depending on how fast you type.

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

```
pod/secondapp created
```

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

```
NAME                        READY     STATUS      RESTARTS   AGE
<output_omitted>
secondapp                   2/2       Running     0          5s
```

8. Expose the **webserver** using a `NodePort` service. Expect an error due to lack of labels.

```
student@cp:~/app2$ kubectl expose pod secondapp --type=NodePort --port=80
```

```
error: couldn't retrieve selectors via --selector flag or
introspection: the pod has no labels and cannot be exposed
See 'kubectl expose -h' for help and examples.
```

9. Edit the YAML file to add a label in the metadata, adding the `example: second` label right after the pod name. Note you can delete several resources at once by passing the YAML file to the delete command. Delete and recreate the pod. It may take up to a minute for the pod to shut down.

```
student@cp:~/app2$ kubectl delete -f second.yaml
```

```
pod "secondapp" deleted
```

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

```
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
  labels:                       #<-- This and following line
    example: second
spec:
#  securityContext:
#    runAsUser: 1000
<output_omitted>
```

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

```
pod/secondapp created
```

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

```
NAME                        READY     STATUS      RESTARTS   AGE
<output_omitted>
```

```
secondapp                  2/2       Running     0          15s
```

10. This time we will expose a `NodePort` again, and create the service separately, then add a label to illustrate how labels are essential for tying resources together inside of kubernetes.

```
student@cp:~/app2$ kubectl create service nodeport secondapp --tcp=80
```

```
service/secondapp created
```

11. Look at the details of the service. Note the selector is set to `app: secondapp`. Also take note of the nodePort, which is 31655 in the example below, yours may be different.

```
student@cp:~/app2$ kubectl get svc secondapp -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2020-04-16T04:40:07Z"
  labels:
    example: second
  managedFields:

....

spec:
  clusterIP: 10.97.96.75
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31665
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: secondapp
  sessionAffinity: None
  type: NodePort
status:
  loadBalancer: {}
```

12. Test access to the service using **curl** and the `ClusterIP` shown in the previous output. As the label does not match any other resources, the **curl** command should fail. If it hangs **control-c** to exit back to the shell.

```
student@cp:~/app2$ curl http://10.97.96.75
```

13. Edit the service. We will change the label to match **secondapp**, and set the `nodePort` to a new port, one that may have been specifically opened by our firewall team, port 32000.

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

```
<output_omitted>
  ports:
  - name: "80"
    nodePort: 32000       #<-- Edit this line
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second       #<-- Edit this line, note key and parameter change
  sessionAffinity: None
```

```
<output_omitted>
```

14. Verify the updated port number is showing properly, and take note of the ClusterIP. The example below shows a ClusterIP of `10.97.96.75` and a port of `32000` as expected.

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

```
NAME         TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
<output_omitted>
secondapp    NodePort   10.97.96.75    <none>        80:32000/TCP   5m
```

15. Test access to the high port. You should get the default nginx welcome page both if you test from the node to the `ClusterIP:<low-port-number>` and from the exterior `hostIP:<high-port-number>`. As the high port is randomly generated make sure it's available. Both of your nodes should be exposing the web server on port 32000. The example shows the use of the **curl** command, you could also use a web browser.

```
student@cp:~/app2$ curl http://10.97.96.75
```

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

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

```
35.184.219.5
```

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

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

16. Now test egress from a container to the outside world. We'll use the **netcat** command to verify access to a running web server on port 80. First test local access to nginx, then a remote server.

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

On Container

```
/ $ nc -vz 127.0.0.1 80
```

```
127.0.0.1 (127.0.0.1:80) open
```

```
/ $ nc -vz www.linux.com 80
```

```
www.linux.com (151.101.185.5:80) open
```

```
/ $ exit
```