Lab 6 – Cluster Networking

Introduction

Kubernetes assumes that pods can communicate with other pods in the cluster, no matter of which host they land on. In a kubernetes cluster, every pod has its own IP address, so the cluster administrator does not need to create links between pods and never needs to deal with mapping container address to host address.

Kubernetes network model, based on CNI, Container Networking Interface, requires that the container address ranges should be routable. This is different from the default docker network model that provides a docker bridge with IP address in a given default subnet. In the default Docker model, each container will get an IP address in that subnet and uses the docker bridge IP as it's default gateway.

In this lab, we will learn and walk through in kubernetes networking.

1. Pod Networking

In a kubernetes cluster, when a pod is deployed, it gets an IP address from the cluster IP address range defined in the inital setup.

1.1 Create the nginx pod image, name as **nginx-pod1.yaml** file

Copy

```
cat > nginx-pod1.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  name: nginx1

  namespace: default

  labels:

    run: nginx
```

```
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
EOF
```

1.2 Create a **nginx** pod using nginx-pod1.yaml

Copy

```
kubectl create -f nginx-pod1.yaml
```

Output:

```
pod "nginx1" created
```

1.3 Verify that the pod status and get the IP address of it

Copy

```
kubectl get pod nginx1 -o wide
```

Sample output:

```
NAME      READY     STATUS    RESTARTS   AGE       IP            NODE
nginx1    1/1       Running   0          58s       10.244.1.22   pod38
-node.onecloud.com
```

Note: Wait till pod changes to **Running** status, before proceeding to next step .

1.4 Capture the **pod_ip** from the nginx1

Copy

```
pod_ip_nginx1=`kubectl get pod nginx1 -o wide | grep nginx1 | awk '{print $6}'`

echo $pod_ip_nginx1
```

Sample output:

```
10.244.1.22
```

1.5 Accessing the **nginx** application with assigned pod_ip address

Copy

```
curl http://$pod_ip_nginx1
```

Output:

```
Welcome to nginx!
```

The containers are not using port 80 on the host node where the container is running. This means we can run multiple nginx pods on the same node all using the same container port 80 and access them from any other pod or node in the cluster using their IP.

1.6 Create a second nginx pod name as **nginx-pod2.yaml** file

Copy

```
cat > nginx-pod2.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  name: nginx2
  namespace: default

  labels:

    run: nginx

spec:

  containers:

  - name: nginx

    image: nginx:latest

    ports:

    - containerPort: 80

EOF
```

1.7 Create the second nginx pod using **nginx-pod2.yaml** file

Copy

```
kubectl create -f nginx-pod2.yaml
```

Output:

```
pod "nginx2" created
```

1.8 Verify the pods status is in **“running”**

Copy

```
kubectl get pods -o wide
```

Sample output:

```
NAME      READY     STATUS    RESTARTS   AGE       IP             NODE
```

```
nginx1    1/1       Running   0          2m        10.244.1.22   pod38
-node.onecloud.com

nginx2    1/1       Running   0          53s       10.244.1.23   pod38
-node.onecloud.com
```

Note: Wait till the pod changes to **Running** state, before proceeding to next step.

Both pods run on the same host node, as we see from their IP address.

1.9 Capture the nginx2 **pod_ip** from the pod status

Copy

```
pod_ip_nginx2=`kubectl get pod nginx2 -o wide | grep nginx2 | awk '{pr
int $6}'`

echo $pod_ip_nginx2
```

Sample output:

```
10.244.1.23
```

1.10 We can still access both pods from any other node in the cluster

Copy

```
curl http://$pod_ip_nginx2:80
```

Output:

```
Welcome to nginx!
```

We do not need to expose container port on host to access nginx application as it is required in standard docker networking model.

1.11 Cleanup the pods:

Copy

```
kubectl delete pod nginx1 nginx2
```

Output:

```
pod "nginx1" deleted

pod "nginx2" deleted
```

2. Host Networking

2.1 Create a pods to use the same host IP address as defined in the **nodejs-pod-hostnet.yaml** configuration file.

Copy

```
cat > nodejs-pod-hostnet.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  name: nodejs

  namespace:

  labels:

spec:

  containers:

  - name: nodejs

    image: kalise/nodejs-web-app:latest

    ports:

    - containerPort: 8080
  hostNetwork: true

EOF
```

2.2 Create the nodejs pod as **nodejs-pod-hostnet.yaml** file

Copy

```
kubectl create -f nodejs-pod-hostnet.yaml
```

Output:

```
pod "nodejs" created
```

2.3 Wait for few minutes the Redis pod is get ready **running** status

Copy

```
kubectl get pods -o wide
```

Sample output:

```
NAME      READY     STATUS     RESTARTS    AGE       IP            NODE

nodejs    1/1       Running    0           1m        10.1.64.110   pod38
-node.onecloud.com
```

Note: Wait till pod changes to **Running** status, before proceeding to next step .

2.4 Access the **nodejs** application:

Copy

```
curl http://10.1.64.110:8080
```

Output:

```
Hello World! from 10.1.64.110
```

However, with the **hostNetwork: true** we cannot start more than one pod listening on the same host port. In general, pods with host network are only used for system or daemon applications that do not need to be scaled.

2.5 Cleanup the nodejs:

Copy

```
kubectl delete pod nodejs
```

Output:

```
pod "nodejs" deleted
```

3. Exposing Services

In kubernetes, services are used not only to provides access to other pods inside the same cluster but also try to access from clients outside the cluster. In this section, we're going to create a deploy of two nginx replicas and expose them to the external world via **nginx service**.

3.1 Create the nginx server to deploy, name as **nginx-deploy.yaml** file

Copy

```
cat > nginx-deploy.yaml <<EOF

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  generation: 1

  labels:

    run: nginx

  name: nginx

  namespace:
```

```
spec:
  minReadySeconds: 15
  replicas: 3
  selector:
    matchLabels:
      run: nginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - image: nginx:latest
        imagePullPolicy: Always
        name: nginx
        ports:
```

```
        - containerPort: 80
          protocol: TCP
      dnsPolicy: ClusterFirst
      restartPolicy: Always
EOF
```

3.2 Create a deployment pod using **nginx-deploy.yaml** file

Copy

```
kubectl create -f nginx-deploy.yaml
```

Output:

```
deployment.extensions "nginx" created
```

3.3 Verify the pods status is in “**running**”

Copy

```
kubectl get pods
```

Sample output:

```
NAME                     READY     STATUS    RESTARTS   AGE
nginx-6bfb654d7c-mk2f7   1/1       Running   0          1m
nginx-6bfb654d7c-tscws   1/1       Running   0          1m
nginx-6bfb654d7c-xdc9t   1/1       Running   0          1m
```

Note: Wait till all pods get ready as “**running**” state.

3.4 Verify that the created nginx pod IPs

Copy

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

Sample output:

```
    podIP: 10.244.1.24

    podIP: 10.244.0.12

    podIP: 10.244.1.25
```

Pods are running on different host nodes as we can see from their IP addresses.

3.5 To create a **nginx** service, we can expose the deploy on **port 80** by running

Copy

```
kubectl expose deploy/nginx --port=80 --target-port=80 --name=nginx-se
rvice
```

Output:

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

3.6 Verify the services status for nginx

Copy

```
kubectl get services -l run=nginx
```

Sample output:

```
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AG
E

nginx-service   ClusterIP   10.103.200.91   <none>        80/TCP    5s
```

3.7 Get more detailed about nginx service, by running the below command

Copy

```
kubectl describe service nginx-service
```

Sample output:

```
Name:              nginx-service

Namespace:         default

Labels:            run=nginx

Annotations:       <none>

Selector:          run=nginx

Type:              ClusterIP

IP:                10.103.200.91

Port:              <unset>  80/TCP

TargetPort:        80/TCP

Endpoints:         10.244.0.12:80,10.244.1.24:80,10.244.1.25:80

Session Affinity:  None

Events:            <none>
```

3.8 capture the service IP from the nginx-service

Copy

```
service_ip=`kubectl describe service nginx-service | grep "^IP" | awk
'{print $2}'`

echo $service_ip
```

Sample output:

```
10.103.200.91
```

3.9 Create the pod from a **busybox.yaml** file

Copy

```
cat > busybox.yaml <<EOF

apiVersion: v1

kind: Pod

metadata:

  name: busybox

  namespace:

spec:

  containers:

  - image: busybox

    command:

      - sleep

      - "3600"

    imagePullPolicy: IfNotPresent

    name: busybox

  restartPolicy: Always

EOF
```

3.10 Any other pod in the cluster is able to access the nginx service without worring about pod IP addresses

Copy

```
 kubectl create -f busybox.yaml
```

Output:

```
pod "busybox" created
```

3.11 Verify the busybox pod status is in “**running**” state

Copy

```
kubectl get pods busybox
```

Sample output:

```
NAME      READY     STATUS    RESTARTS   AGE

busybox   1/1       Running   0          4s
```

3.12 Download the busybox application via service_ip

Copy

```
kubectl exec -it busybox -- wget -O - $service_ip
```

Output:

```
Welcome to nginx!

-                    100% |*******************************|   612   0:
00:00 ETA
```

The service is not reachable from any cluster host. If we try to access the service we do not get anything

3.13 Take a note of **service ip**:

Copy

```
echo $service_ip
```

Sample output:

```
10.103.200.91
```

Note: Run the below command on **pod-master**.

3.14 Access the busybox application via service_ip from the **master node**

Copy

```
curl 10.103.200.91:80
```

Without specifying the type of service, kubernetes by default uses the **Type: ClusterIP** option, which means that the new service is only exposed only within the cluster. It is kind of like internal kubernetes service, so not particularly useful if you want to accept external traffic.

When creating a service, currently, kubernetes provides four options of service types:

- **ClusterIP:** it exposes the service only on a cluster internal IP making the service only reachable from within the cluster. This is the default Service Type.
- **NodePort:** it exposes the service on each node public IP on a static port as defined in the NodePort option. It will be possible to access the service, from outside the cluster.
- **LoadBalancer:** it exposes the service by creating an external load balancer. It works only on some public cloud providers. To make this working, remember to set the option –cloud-provider in the kube controller manager startup file
- **ExternalName:** it maps the service to the contents of the externalName option, e.g. search.google.com, by returning a name record with its value.

In this section we are going to use the NodePort service type to expose the service.

Switch back to **pod-master**

3.15 Delete the the service which we created earlier

Copy

```
kubectl delete svc/nginx-service
```

Output:

```
service "nginx-service" deleted
```

3.16 Create a new service with NodePort type

Copy

```
kubectl expose deploy/nginx --port=80 --target-port=80 --type=NodePort
--name=nginx-service
```

Output:

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

3.17 Describe the nginx service

Copy

```
kubectl describe svc/nginx-service
```

Output:

```
Name:                     nginx-service

Namespace:                default

Labels:                   run=nginx

Annotations:              <none>

Selector:                 run=nginx

Type:                     NodePort

IP:                       10.101.42.3

Port:                     <unset>  80/TCP
```

```
TargetPort:                80/TCP

NodePort:                    32650/TCP

Endpoints:                 10.244.0.12:80,10.244.1.24:80,10.244.1.25:80

Session Affinity:          None

External Traffic Policy:   Cluster

Events:                    <none>
```

3.18 The NodePort type opens a service port on every worker node in the cluster. The service port is mapped to a port on the public IP node as in the NodePort. On any worker node, it is available at

Copy

```
node_port=`kubectl describe svc/nginx-service | grep ^NodePort | awk '
{print $3}' | cut -d "/" -f 1`

echo $node_port
```

Output:

```
32650
```

Note: Take a note of **node_port**.

3.19 Verify that the node_port is available in netstat

Copy

```
netstat -natp | grep $node_port
```

Output:

```
tcp6       0      0 :::32650                :::*                    LI
STEN      17041/kube-proxy
```

The kube-proxy service on the worker node, is in charge of doing this job by configuring the IPtables. Now it is possible to access the nginx service from ouside the cluster by pointing to any worker node

Note: Run the below command to **pod-node**.

3.20 Access the nginx application via clusterIP

Copy

```
curl pod-master:$node_port
```

or

Open a web browser and type the below URL

Note: Configure port-forwarding for NodePort (Ex: 32650).

Copy

```
http://localhost:node_port
```

3.21 The NodePort is randomly selected from the **30000-32767** range. If you want to force a specific port, define it in a file **nginx-nodeport-svc.yaml**

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    run: nginx
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 8090
  selector:
    run: nginx
  type: NodePort
```

Now that we have a port open on every worker node, we can configure an external load balancer or edge router to route the traffic to any of the worker nodes.

4. Cleanup

Let’s cleanup by switching back to **pod-master**

4.1 Delete the nginx deployment

Copy

```
kubectl delete  deploy nginx
```

Output:

```
deployment.extensions "nginx" deleted
```

4.2 Delete the nginx service

Copy

```
kubectl delete svc nginx-service
```

Output:

```
service "nginx-service" deleted
```

4.3 Delete a busybox pods

Copy

```
 kubectl delete pod busybox
```

Output:

```
pod "busybox" deleted
```

5. Service discovery

To enable service name discovery in a kubernetes cluster, we need to configure an embedded DNS service to resolve all DNS queries from pods trying to access services. The embedded DNS should be manually installed during cluster setup since it is part of the cluster architecture, unless users are going to use other custom solutions for service discovery, e.g. consul.

5.1 Verify the embedded DNS lives in the **kube-system namespace**

Copy

```
kubectl get all -n kube-system
```

Output:

```
NAME                                                READY     STATUS
RESTARTS   AGE

etcd-pod38-master.onecloud.com                      1/1       Running
0          10h

kube-apiserver-pod38-master.onecloud.com            1/1       Running
0          9h

kube-controller-manager-pod38-master.onecloud.com   1/1       Running
0          10h

kube-dns-86f4d74b45-7c28w                           3/3       Running
0          10h

kube-flannel-ds-bnc2d                               1/1       Running
0          9h

kube-flannel-ds-mlklx                               1/1       Running
0          9h

kube-proxy-52flb                                    1/1       Running
0          10h

kube-proxy-w8pjn                                    1/1       Running
0          9h

kube-scheduler-pod38-master.onecloud.com            1/1       Running
0          10h

kubernetes-dashboard-7d5dcdb6d9-8px9b               1/1       Running
0          1h



NAME                   TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)
AGE
```

```
kube-dns               ClusterIP   10.96.0.10          53/UDP,53/TCP
10h

kubernetes-dashboard   NodePort    10.99.33.21         443:32380/TCP
1h



NAME              DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE
NODE SELECTOR                   AGE

kube-flannel-ds   2         2         2         2            2
beta.kubernetes.io/arch=amd64   9h

kube-proxy        2         2         2         2            2
10h



NAME                   DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AG
E

kube-dns               1         1         1            1           10
h

kubernetes-dashboard   1         1         1            1           1h



NAME                              DESIRED   CURRENT   READY     AGE

kube-dns-86f4d74b45               1         1         1         10h

kubernetes-dashboard-7d5dcdb6d9   1         1         1         1h
```

It consists of a controller, a service and a pod running a DNS server, a dnsmaq for caching and healthz for liveness probe. Each time a user starts a new pod, kubernetes injects certain nameservice lookup configuration into new pods allowing to query the DNS records in the cluster. Each time a new service is created, kubernetes registers this service name into the embedded DNS server allowing all pods to query the DNS server for service name resolution.

5.2 Create a nginx deploy and create the service. Since we’re not interested (yet) to expose the service outside the cluster, we leave the default service type, i.e. the ClusterIP mode. This allows only pods inside the cluster can access the service.

Copy

```
kubectl create -f nginx-deploy.yaml
```

Output:

```
deployment.extensions "nginx" created
```

5.3 Expose the deployed nginx server to the port 8080 and target-port is 80

Copy

```
kubectl expose deploy/nginx --port=8080 --target-port=80 --name=nginx-
service
```

Output:

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

5.4 Verify all the nginx status is in “**running**”

Copy

```
kubectl get all -l run=nginx
```

Output:

```
NAME                        READY     STATUS    RESTARTS   AGE

nginx-6bfb654d7c-mjfvg   1/1       Running   0          2m

nginx-6bfb654d7c-tw9z5   1/1       Running   0          2m

nginx-6bfb654d7c-vsjz8   1/1       Running   0          2m
```

```
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    A
GE

nginx-service   ClusterIP   10.101.204.84           8080/TCP   2m



NAME     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE

nginx    3         3         3            3           2m



NAME               DESIRED   CURRENT   READY     AGE

nginx-6bfb654d7c   3         3         3         2m
```

Note: Wait till all pods changes to **running** state.

5.5 Start a test pod and check if it access the nginx service

Copy

```
kubectl create -f busybox.yaml
```

Output:

```
pod "busybox" created
```

5.6 Verify the **busybox** pod is in running status:

Copy

```
kubectl get pod
```

Output:

```
NAME                         READY     STATUS     RESTARTS   AGE
```

```
busybox                  1/1       Running   0          5s

nginx-6bfb654d7c-mjfvg   1/1       Running   0          3m

nginx-6bfb654d7c-tw9z5   1/1       Running   0          3m

nginx-6bfb654d7c-vsjz8   1/1       Running   0          3m
```

5.7 Capture and assign the nginx-service ip to the varaiable

Copy

```
service_ip=`kubectl describe service nginx-service | grep "^IP" | awk
'{print $2}'`

echo $service_ip
```

Sample output:

```
10.101.204.84
```

5.8 Download the busybox application via assigned service_ip varaible

Copy

```
kubectl exec -ti busybox -- wget $service_ip:8080
```

Output:

```
Connecting to 10.101.204.84:8080 (10.101.204.84:8080)

index.html           100% |*******************************|   612   0:
00:00 ETA
```

5.9 Check if service DNS lookup configuration has been injected by kubernetes

Copy

```
kubectl exec -ti busybox -- cat /etc/resolv.conf
```

Output:

```
nameserver 10.96.0.10

search default.svc.cluster.local svc.cluster.local cluster.local onecloud

options ndots:5
```

5.10 Now check if service discovery works by resolv the service name

Copy

```
kubectl exec -ti busybox -- nslookup nginx-service
```

Sample output:

```
SServer:    10.96.0.10

Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local



Name:      nginx-service

Address 1: 10.101.204.84 nginx-service.default.svc.cluster.local
```

This mechanism permits kubernetes pods to be linked each other without dealing with IP service assignment.

5.11 Lets cleanup all the replicasets

Copy

```
kubectl delete rs --all
```

Output:

```
replicaset.extensions "nginx-6bfb654d7c" deleted
```

5.12 Delete the ngix services

Copy

```
kubectl delete svc nginx-service
```

Output:

```
service "nginx-service" deleted
```

5.13 Delete the deployed nginx server

Copy

```
 kubectl delete deploy nginx
```

Output:

```
deployment.extensions "nginx" deleted
```

5.14 Delete all the pods

Copy

```
kubectl delete pod --all
```

Output:

```
pod "busybox" deleted
```

6. Accessing services

In this section, we’re going to deploy a WordPress application made of two services:

1. Worpress service
2. MariaDB service

We’ll use the service discovery feature to permit the worpress pod to access the MariaDB pod without knowing the IP address. Also we’ll expose the Worpress service to external world.

6.1 Create the **MariaDB** deploy as mariadb-deploy.yaml file

Copy

```
cat > mariadb-deploy.yaml << EOF

---

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

  generation: 1

  labels:

    run: mariadb

  name: mariadb

  namespace: default

spec:

  replicas: 1

  selector:

    matchLabels:

      run: mariadb

  strategy:

    rollingUpdate:

      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: mariadb
    spec:
      containers:
      - image: bitnami/mariadb:latest
        imagePullPolicy: Always
        name: mariadb
        ports:
        - containerPort: 3306
          protocol: TCP
        env:
        - name: MARIADB_ROOT_PASSWORD
          value: bitnami123
        - name: MARIADB_DATABASE
          value: workpress
        - name: MARIADB_USER
          value: bitnami
        - name: MARIADB_PASSWORD
          value: bitnami123
        volumeMounts:
        - name: mariadb-data
          mountPath: /bitnami/mariadb

      volumes:
      - name: mariadb-data
        emptyDir: {}
      dnsPolicy: ClusterFirst
      restartPolicy: Always
EOF
```

6.2 Create and deploy the mariadb using **mariadb-deploy.yaml** file

Copy

```
kubectl create -f mariadb-deploy.yaml
```

Output:

```
deployment.extensions "mariadb" created
```

6.3 Verify the mariadb server pod status

Copy

```
kubectl get all -l run=mariadb
```

Output:

```
NAME                       READY     STATUS    RESTARTS   AGE
mariadb-7cdc54c57b-xlntc   1/1       Running   0          3m

NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
mariadb   1         1         1            1           3m

NAME                 DESIRED   CURRENT   READY     AGE
mariadb-7cdc54c57b   1         1         1         3m
```

Note: Wait for couple of minutes, till pod changes to **running** state.

6.4 Create a service called mariadb as **mariadb-svc.yaml** file

Copy

```
cat > mariadb-svc.yaml <<EOF
apiVersion: v1
kind: Service
metadata:
  name: mariadb
  labels:
    run: mariadb
spec:
  ports:
  - protocol: TCP
```

```
    port: 3306

    targetPort: 3306

  selector:

    run: mariadb

EOF
```

6.5 Expose it as an internal service

Copy

```
kubectl create -f mariadb-svc.yaml
```

Output:

```
service "mariadb" created
```

6.6 Verify the mariadb service status

Copy

```
kubectl get service -l run=mariadb
```

Output:

```
NAME      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE

mariadb   ClusterIP   10.110.222.104   <none>        3306/TCP   51s
```

6.7 Get the detailed information about mariadb service

Copy

```
kubectl describe svc mariadb
```

Sample output:

```
Name:              mariadb

Namespace:         default

Labels:            run=mariadb

Annotations:       <none>

Selector:          run=mariadb

Type:              ClusterIP

IP:                10.110.222.104

Port:              <unset>  3306/TCP

TargetPort:        3306/TCP

Endpoints:         10.244.1.34:3306

Session Affinity:  None

Events:            <none>
```

This service will be used by the wordpress application as database backend. Thanks to the DNS service discovery embedded in the kubernetes cluster, the worpres application has not to take care of the mariadb database IP address. It should only reference a generic mariadb host. The embedded DNS will resolve this name into the real IP address of the mariadb service. Also, since we are not controlling where kubernetes start the mariadb pod, we are not worring about of the real IP of the mariadb pod.

6.8 Here the **wordpress-deploy.yaml** file defining the wordpress application

Copy

```
cat > wordpress-deploy.yaml <<EOF

---

apiVersion: extensions/v1beta1

kind: Deployment
```

```
metadata:
  generation: 1
  labels:
    run: blog
  name: wordpress
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      run: blog
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        run: blog
    spec:
      containers:

      - image: bitnami/wordpress:latest

        imagePullPolicy: Always

        name: wordpress

        ports:

        - containerPort: 80

          protocol: TCP

        - containerPort: 443

          protocol: TCP

        env:

        - name: MARIADB_HOST

          value: mariadb

        - name: MARIADB_PORT

          value: '3306'

        - name: WORDPRESS_DATABASE_NAME

          value: workpress

        - name: WORDPRESS_DATABASE_USER

          value: bitnami

        - name: WORDPRESS_DATABASE_PASSWORD

          value: bitnami123

        - name: WORDPRESS_USERNAME
```

```
    value: admin
  - name: WORDPRESS_PASSWORD
    value: password
  volumeMounts:
  - name: wordpress-data
    mountPath: /bitnami/wordpress
  - name: apache-data
    mountPath: /bitnami/apache
  - name: php-data
    mountPath: /bitnami/php

volumes:
- name: wordpress-data
  emptyDir: {}
- name: apache-data
  emptyDir: {}
- name: php-data
  emptyDir: {}

dnsPolicy: ClusterFirst
restartPolicy: Always
```

```
EOF
```

6.9 Deploy the wordpress application

Copy

```
kubectl create -f wordpress-deploy.yaml
```

Output:

```
deployment.extensions "wordpress" created
```

6.10 Verify the deployed wordpress pod status

Copy

```
kubectl get all -l run=blog
```

Sample output:

```
NAME                         READY     STATUS    RESTARTS   AGE
wordpress-8675b5c947-ht4ss   1/1       Running   0          1m


NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
wordpress   1         1         1            1           1m


NAME                   DESIRED   CURRENT   READY     AGE
wordpress-8675b5c947   1         1         1         1m
```

Note: Wait till pod changes to **running** state.

6.11 Now we need to expose the frontend wordpress application outside the cluster. To make this, we’ll create a nodeport worpress service and expose it on a given port. Here the service definition as in the **wordpress-svc.yaml** file

Copy

```
cat > wordpress-svc.yaml <<EOF

apiVersion: v1

kind: Service

metadata:

  name: wordpress

  labels:

    run: blog

spec:

  ports:

  - protocol: TCP

    port: 80

    targetPort: 80

    nodePort: 31080

  selector:

    run: blog

  type: NodePort

EOF
```

6.12 Create the wordpress service

Copy

```
kubectl create -f wordpress-svc.yaml
```

Output:

```
service "wordpress" created
```

6.13 Verify the wordpress service status

Copy

```
watch kubectl get all -l run=blog
```

Sample output:

```
NAME                         READY     STATUS    RESTARTS   AGE
wordpress-8675b5c947-ht4ss   1/1       Running   0          1m

NAME        TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
wordpress   NodePort   10.109.5.169           80:31080/TCP   6s

NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
wordpress   1         1         1            1           1m

NAME                   DESIRED   CURRENT   READY     AGE
wordpress-8675b5c947   1         1         1         1m
```

Note: Wait for couple of minutes to get pod status as **running** and then press **<ctrl+c>** to interrupt.

6.14 Get more description about deployed wordpress service

Copy

```
kubectl describe svc/wordpress
```

Sample output:

```
Name:                     wordpress

Namespace:                default

Labels:                   run=blog

Annotations:              <none>

Selector:                 run=blog

Type:                     NodePort

IP:                       10.109.5.169

Port:                     <unset>  80/TCP

TargetPort:               80/TCP

NodePort:                 <unset>  31080/TCP

Endpoints:                10.244.1.35:80

Session Affinity:         None

External Traffic Policy:  Cluster

Events:                   <none>
```

6.15 This service will be accessible from all worker nodes in the cluster thanks to the **kube-proxy** job. Try to access it from any external client by pointing to any of the worker node

Copy

```
wget pod-master:31080
```

Sample output:

```
--2018-04-15 23:57:34--  http://pod38-master.onecloud.com:31080/

Resolving pod38-master.onecloud.com (pod38-master.onecloud.com)... 10.
1.64.236

Connecting to pod38-master.onecloud.com (pod38-master.onecloud.com)|10
.1.64.236|:31080... connected.

HTTP request sent, awaiting response... 200 OK

Length: unspecified [text/html]

Saving to: ‘index.html.1’



    [ <=>
] 53,738      --.-K/s   in 0.005s



2018-04-15 23:57:34 (9.47 MB/s) - ‘index.html.1’ saved [53738]
```

or

6.16 Open a web browser and type the below URL

Copy

```
http://localhost:31080
```

Note: Configure port-forwarding for 31080

6.17 Lets’s Cleanup the all the deployments

Copy

```
kubectl delete deploy --all
```

Output:

```
deployment.extensions "mariadb" deleted

deployment.extensions "wordpress" deleted
```

6.18 Delete all the services

Copy

```
kubectl delete svc mariadb wordpress
```

Output:

```
service "mariadb" deleted

service "wordpress" deleted
```

6.19 Verify that the pods are deleted

Copy

```
 kubectl get pods
```

Output:

```
No resources found.
```

7. External Services – Try on your own

The service abstraction in kubernetes can be used to model also external services that are not part of the cluster. For example, a pre-existing Oracle database can be modeled as a common standard service to be accessed from an application running in the cluster as pod.

The only requirement is the worker nodes should be able to reach the address of the external database.

An external service does not use label selectors since there are no pods to bind in the cluster.

7.1 An external service definition file **mysql-external-svc.yaml** looks like the following

```
apiVersion: v1

kind: Service

metadata:

  name: external-mysql

  namespace:

spec:
  ports:

  - port: 3306

    protocol: TCP

    targetPort: 3306

  type: ClusterIP
```

7.2 Create the external service

```
kubectl create -f mysql-external-svc.yaml
```

7.3 Get the more detailed external mysql

```
kubectl get services external-mysql -o wide
```

Output:

```
NAME             CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE       SE
LECTOR

external-mysql   10.32.107.128   <none>        3306/TCP   41m       <n
one>
```

7.4 Create the external serviceBy inspecting the service, we find that no endpoints are available since it is an headless service. The endpoints need to be manually created as in the mysql-external-ep.yaml configuration file

```
apiVersion: v1

kind: Endpoints

metadata:

  name: external-mysql
  namespace:
subsets:
- addresses:
  - ip: 10.10.10.3
  ports:
  - port: 3306
    protocol: TCP
```

The IP address above is the actual IP address of the MySQL server running outside the kubernetes cluster.

7.5 Create the external serviceTo test the external MySQL server is modeled as an internal service in kubernetes, start a simple MySQL client running in a pod and connect to the external database by specifying the name of the external service as it is discovered by the embedded DNS in kubernetes

```
kubectl run -it --rm ephemeral --image=mysql -- /bin/sh -l
 mysql -h external-mysql -u root -p
MySQL [(none)]>
```

7.6 Exit from the mysql database

```
exit
```