In this lab, we will deploy web application and database on **Kubernetes Cluster** on IBM Cloud.

Pre-req:

Lab : Kubernetes Part 1 Completed.

STEP 1: Deploying the Application

Now that we have both a Docker image (our application) and a Kubernetes cluster available on the IBM Cloud, we can deploy the middle tier of our application. In order to do that we need to tell Kubernetes how to deploy the application

Lab: Deploy application on IBM Cloud Kubernetes Cluster

We do this using a yaml file that describes how Kubernetes should construct our “pod” that is running our container/image:

```
$ cat webpod.yaml
apiVersion: "v1"
kind: Pod
metadata:
  name: web
  labels:
    app: demo
spec:
  containers:
    - name: web
      image: <namespace>/webapp
      ports:
        - containerPort: 5000
```

Let’s discuss the contents of this file:

- Kind: Pod, we are creating a Pod
- We are adding one label, setting `app` to `demo`.
- We have a whole separate section for the ‘spec’ of the pod
- This spec contains the definition of what containers to run
- We’re running our previously built webapp container
- This is a webserver, so we need to expose the port that is doing the serving.

Using kubectl we will now create the “web” Pod Object in Kubernetes. To recall, this stores the pod definition into the underlying datastore. The definition will be seen by several controllers in sequence who will process it. The scheduler will schedule it, and the chosen kubelet will try to run the container.

```
$ kubectl create -f webpod.yaml
pod "web" created
```

Let’s see what the pod definition looks like now.

Eventually the pod enters a running state.

```
$ kubectl get pod web
NAME      READY     STATUS    RESTARTS   AGE
web       1/1       Running   0          8s
```

The “web” Pod is now running

```
$ kubectl exec -it web bash
root@web:/usr/src/app#
```

Keep in mind that when the Pod is run it will look for “redis” service to connect to which does not exist yet. Let’s check the output of the webapp.

```
root@web:/usr/src/app# curl 127.0.0.1:5000  # responds with error that it cannot find redis service
ConnectionError: Error -2 connecting to redis:6379. Name or service not known
```

We cannot ping the web app from the node because the port 5000 is not exposed (`bc cs nodes` to see worker ip). We need to expose “web” Pod as a service in order to be able to reach this service on the Node.
Let us expose the web app as a service at port 80

STEP 2: Exposing our Application

Once the Container is tested, we need to expose it externally to the cluster. To do that, we will create a Kubernetes Service. The Service will give us a stable reference point to access the underlying pod.

```
$ cat websvc.yaml
apiVersion: v1
kind: Service
metadata:
  name: web
  labels:
    app: demo
spec:
  selector:
    app: demo
  type: NodePort
  ports:
   - port: 5000
     nodePort: 31000

$ kubectl create -f websvc.yaml
service "web" created


$ kubectl get svc
NAME         CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
web          10.10.10.70    <nodes>       5000:31000/TCP     3m
```

Note : The service is listening on the port 30951, a random port on the node. Once we know the ip of the node we should be able to ping the service. Let's get the name by looking at the details of the cluster in the container service.

```
$ bx cs workers <your cluster name>
OK
ID                                                     Public IP        Private
IP     Machine Type   State    Status   Version
kube-hou02-pa1e3ee39f549640aebea69a444f51fe55-w1   173.193.99.136
10.76.194.30   free           normal   Ready    1.5.6_1500*

$ curl 173.193.99.136:31000
...
"ConnectionError: Error -2 connecting to redis:6379. Name or service not known."
...
```

Should still show that redis is not available.

Service Ports

•**ClusterIP**: Exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default ServiceType

•**NodePort**: Exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

•**LoadBalancer**: Exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

STEP 3: Deploy the Database

We are now ready to create the backend database for the 3-Tier application. Our backend is redis pod that will store the number of times a web page is requested, redis will just keep a counter that the web server will query every time we access the page.

```
$ cd db
$ cat dbpod.yaml
apiVersion: "v1"
kind: Pod                       => kind of resource is "pod"
metadata:
  name: redis                   => resource name "redis"
  labels:
    name: redis
    app: tutorial
spec:                  => spec usually has one container but allows multiple
  containers:
    - name: redis
      image: redis:latest    =>  use the docker image "redis:latest"
      ports:
        - containerPort: 6379     => redis server listening on port 6379
          protocol: TCP
```

Let us now test redis pod.
List if any pods running, we should see your web pod running.

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
web       1/1       Running   0          8s
```

Now create the "redis" Pod

```
$ kubectl create -f dbpod.yaml
pod "redis" created
```

Listing pods again will show both web and redis pods runnning.

```
$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
web       1/1       Running   0          8m
redis     1/1       Running   0          14s
```

We can also get more details using the following command which shows the port redis is listening on. We can detect that port is not exposed.

```
$ kubectl describe pods
```

Let us test redis Pod alone.

```
$ kubectl exec -it redis bash

$ redis-cli ping
PONG

$ redis-cli
127.0.0.1:6379> set mykey test-only
OK
127.0.0.1:6379> get mykey
"test-only"
127.0.0.1:6379>
```

STEP 4: Converting the Database into a Service

Our web application still cannot find redis as it is not exposed in the cluster because the redis service needs to expose itself with a dns entry reachable in the cluster.

```
$ curl 173.193.99.136:31000
```

Let us expose the redis service within the cluster so that web pod can reach to it

```
$ cat dbsvc.yaml
apiVersion: v1
kind: Service     => exposed redis as a cluster service
metadata:
  name: redis
  labels:
    name: redis
    app: demo
spec:
  ports:
  - port: 6379    => expose redis at port 6379 in the cluster
    name: redis
    targetPort: 6379
  selector:
    name: redis
    app: demo
```

Now let's deploy it:

```
$ kubectl create -f dbsvc.yaml
service "redis" created
```

And, verify that it worked:

```
$ kubectl get svc
NAME         CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
redis        10.10.10.144   <none>        6379/TCP         42s
web          10.10.10.70    <nodes>       5000:31000/TCP     9m
```

Notice the "PORT(S)" column shows that "redis" is listening on port 6379 at the cluster level, while "web" is listening on port 5000. However, the ":31000" for "web" indicates that Kubernetes has also mapped port 5000 to 31000 on the node. This means that sending a request to the node directly, at that port, should get routed to the "web" container:

```
$ curl 173.193.99.136:31000  # now the web svc can find redis.
Hello Container World from web! I have been seen 1 times.
```

Congratulations!! You have deployed application on IBM Cloud using Kubernetes Container Service