ECGR 4090/5090 Cloud Native Application Architecture
Lab 10 : Kubernetes Basics

In this lab, you will get hands-on experience with Kubernetes (K8s). In this lab, we will be using a local K8s installation using MicroK8s.
MicroK8s is a CNCF certified upstream Kubernetes deployment from Canonical (the company behind Ubuntu) that can run entirely on your laptop or an edge device (such as Raspberry Pi).

What is K8s used for?
K8s coordinates a cluster of shared-nothing computers that are connected to work as a single unit. K8s automates the distribution and scheduling of application containers (such as the Docker containers from previous labs) across the cluster. In this lab, our cluster will only be a single machine. But remember that K8s can scale across hundreds of machines in a production cluster.

Microk8s Installation

Follow Steps 1 - 4 outlined here to get Microk8s up and running.
https://microk8s.io/docs/getting-started

Microk8s command reference
https://microk8s.io/docs/command-reference

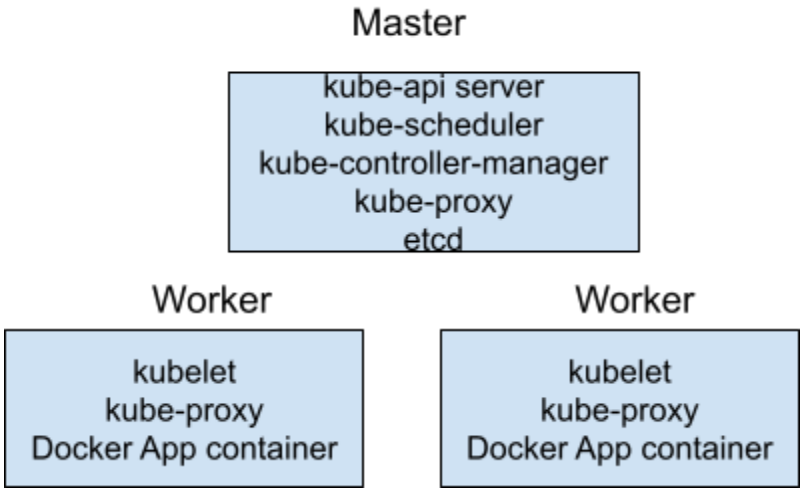
$ microk8s start // To start cluster
$ microk8s stop // To stop cluster
$ microk8s enable <add on name> // Enables add on

Let's enable the following addons
$ *microk8s enable dns hostpath-storage // storage deprecated*

kubectl is used to run K8s commands. By aliasing 'microk8s kubectl' to kubectl, you will be typing in commands as you would on a regular K8s distribution.
$ *alias kubectl='microk8s kubectl'*

Find K8s version
$ *kubectl version --short*

K8s components



Master and worker: Master and workers are on different nodes of the cluster. The nodes could be VMs or physical machines. Master controls the cluster, workers do the computation.

Pod: In K8s, pod is the basic unit of scheduling. A pod is a group of tightly coupled containers always scheduled on a single node. Pods share resources such as storage and cluster IP address

kube-api server: The nodes communicate with the master using the Kubernetes API, which the master exposes. End users can also use the Kubernetes API directly to interact with the cluster.

kube-scheduler: Schedules jobs across worker nodes

kube-controller manager: Controls the state of the cluster. For example, the node controller monitors the health of nodes; the replication controller maintains the specified pod replicas

kubelet: An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

kube-proxy: kube-proxy maintains network rules on nodes. These network rules allow network communication to pods from network sessions inside or outside of the cluster.

Deploy an Application

To deploy an application, you create a K8s Deployment configuration, The Deployment instructs Kubernetes how to create and update instances of your application. Once you've created a Deployment, the Kubernetes master schedules the application instances included in that Deployment to run on individual Nodes in the cluster.

Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster. This provides a self-healing mechanism to address machine failure or maintenance.

On the Microk8s command line
View the nodes in the cluster

$ kubectl get nodes

Deploy the NGINX app (pronounced as “EngineX”)
Note: NginX, is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache. The container is automatically downloaded from Dockerhub

$ kubectl create deployment nginx --image=nginx

List deployments

$ kubectl get deployments

We see that there is 1 deployment running a single instance of your app. The instance is running inside a Docker container on your node. It may take some time to get running! Keep checking.

Pods that are running inside Kubernetes are running on a private, isolated network. By default they are visible from other pods and services within the same kubernetes cluster, but not outside that network. When we use kubectl, we're interacting through an API endpoint to communicate with our application.

The kubectl command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminated by pressing control-C and won't show any output while it's running.

We open a second terminal window to run the proxy. Give admin privileges like before.
$ sudo usermod -a -G microk8s $USER
$ sudo chown -f -R $USER ~/.kube
$ su - $USER
Alias kubectl like we did earlier.
$ *alias kubectl='microk8s kubectl'*
Start proxy
$ echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please click the first Terminal Tab\n"; kubectl proxy

We now have a connection between our host (the online terminal) and the Kubernetes cluster. We can query the version directly through the API using the curl command:
From the first terminal,
$ curl http://localhost:8001/version

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

First we need to get the Pod name, and we'll store in the environment variable POD_NAME:

$ *export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}'); echo Name of the Pod: $POD_NAME*

In order for the new deployment to be accessible without using the Proxy, a Service is required (covered later)

Exploring the application

Let’s examine the application in more detail.

Verify it’s still running

$ kubectl get pods

Next, to view what containers are inside that Pod and what images are used to build those containers

$ kubectl describe pods

To access the pod using a proxy from outside the cluster issue the following commands.

$ curl http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME/proxy/

You should be able to see the NGINX welcome message.

Anything that the application would normally send to STDOUT becomes logs for the container within the Pod. To retrieve the logs (here there is only one container in the pod) -

$ kubectl logs $POD_NAME

Executing a command on the container - list the environment variables:

$ *kubectl exec $POD_NAME -- env*

Start a bash session in the Pod's container -

$ kubectl exec -it $POD_NAME -- bash

Using services to expose the application

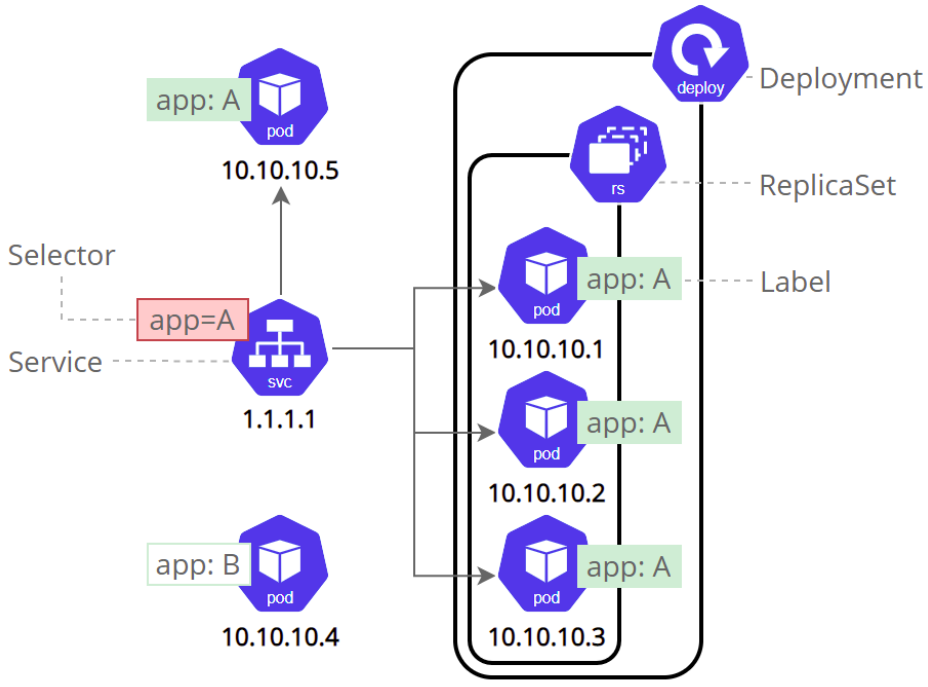
A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods (for example frontend and backend pods). A Service is defined using YAML (preferred) or JSON manifest file, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a LabelSelector (key-value pairs). The service abstraction hides the event of pods dying and new pods replacing them with different IP addresses.

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a type in the ServiceSpec:

ClusterIP (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.

NodePort - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>. This is a superset of ClusterIP.

LoadBalancer - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. This is a superset of NodePort.



(From kubernetes.io)

Hello World application

Let's deploy a new application. We'll examine the YAML manifest file describing the application later in the lab.

$ *kubectl apply -f https://k8s.io/examples/service/access/hello-application.yaml*

Check if the deployment is ready. This may take some time. Keep rechecking.
$ kubectl get deployments hello-world

List the current Services from our cluster

$ kubectl get services

We have a Service called kubernetes that is created by default when Microk8s starts the cluster. To create a new service and expose it to external traffic we'll use the expose command with NodePort as parameter

$ kubectl expose deployment hello-world --type=NodePort --name=hello-service

Check if new service has started

$ *kubectl get services hello-service*

Create an environment variables to capture the node port and node ip values

$ *export NODE_PORT=$(kubectl get services/hello-service -o go-template='{{(index .spec.ports 0).nodePort}}'); echo NODE_PORT=$NODE_PORT*

$ export NODE_IP=$(kubectl describe nodes | grep InternalIP | awk '{print $2}');echo $NODE_IP

Now we can test that the app is exposed outside of the cluster using curl. You should get the Hello Kubernetes message

$ curl http://$NODE_IP:$NODE_PORT

Should print the "Hello Kubernetes!" message

Creating K8s deployments with YAML manifest files

Here, we use YAML to describe a deployment. YAML (YAML Ain't Markup Language) is a human readable format for specifying configuration files. YAML is a superset of JSON. Unlike JSON, YAML uses indentation instead of curly braces. It should be noted that only spaces are used for indentation (no tabs).

Below is the YAML manifest file for the hello-world deployment that you deployed on the cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      app: hello-world
  replicas: 2
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
        - name: hello-world
          image: gcr.io/google-samples/node-hello:1.0
          ports:
            - containerPort: 8080
```

A brief description of the YAML for the deployment above -

We first specify the apiVersion and kind. Next we specify the name. We can also specify any other metadata we want. Under the deployment spec, the label selector is used to specify which pods are part of the deployment. The replica key specifies that whatever pods we deploy, we always want to have 2 replicas that the deployment should keep alive. Everything under the template key is a pod specification. As a part of the pod spec, we have the keys specifying the container properties including its name, image, and the exposed ports.

Note: Similar YAML manifest files can be used to describe pods and services.

You can get the YAML of the deployment directly (yq needs to be installed)

$ *kubectl get deploy hello-world -o yaml | yq eval 'del(.metadata.resourceVersion, .metadata.uid, .metadata.annotations, .metadata.creationTimestamp, .metadata.selfLink, .metadata.managedFields)' -*

Recovery from failure

Let's see what happens when one of the pods of the hello-world deployment is killed.

Get the pods names

$ kubectl get pods

Kill one of the hello-world pods

$ kubectl delete pods <pod name>

Examine the pods again.

$ kubectl get pods

You will see that K8s has regenerated a new pod in place of the killed pod. Check the different age of the new pod as compared to the other pod in the hello-world replica set.

K8s thus helps in application resiliency. In a multi-node cluster, if one of the nodes fail, K8s will reschedule the pods to another node.

Scaling the application

Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources. Scaling will increase the number of Pods to the new desired state.

Running multiple instances of an application will require a way to distribute the traffic to all of them. Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment. Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

Scaling is accomplished by changing the number of replicas in a Deployment. Once you have multiple instances of an Application running, you would be able to do Rolling updates without downtime.

Scale the number of replicas from 2 to 4.

$ *kubectl scale deployments/hello-world --replicas=4*

Check if the pods associated with the deployment scaled

$ *kubectl get deployments hello-world*

Let’s check the pods system wide as well to confirm that we have 4 hello-world pods with different IP addresses

$ *kubectl get pods -o wide*

The change is registered in the log as well

$ *kubectl describe deployments/hello-world*

We can scale up and down from the command line as well. Let’s scale down from the command line.

$ *kubectl scale deployments/hello-world --replicas=2*

Check to see the state of the pods

$ *kubectl get pods -o wide*

You should see that 2 of 4 pods now have status as “terminating”

Performing a rolling update

Users expect applications to be available all the time and developers are expected to deploy new versions of them several times a day. In Kubernetes this is done with rolling updates. Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources.

Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update. An available Pod is an instance that is available to the users of the application.

Rolling updates allow the following actions:

- Promote an application from one environment to another (via container image updates)
- Rollback to previous versions
- Continuous Integration and Continuous Delivery of applications with zero downtime (we’ll delve into this in more detail in a later lab).

Let’s change the container image of the application to a new image

$ *kubectl set image deployments/hello-world hello-world=k8s.gcr.io/echoserver:1.4*

Check the status of the new Pods, and view the old one terminating

$ *kubectl get pods*

Verify the update

$ *curl http://$NODE_IP:$NODE_PORT*

You should get a whole new message from the new application image.

Note: Typically, the updates are incremental, such as bug fixes, and addition of new features.

You can also check the rollout status

$ *kubectl rollout status deployments/hello-world*

Let's also see how to rollback updates if something goes wrong.

We are going to update with an image that does not exist.

$ *kubectl set image deployments/hello-world*
hello-world=gcr.io/google-samples/kubernetes-bootcamp:v10

Check the status of the pods

$ *kubectl get pods*

We see an *ImagePullBackoff* status.

Get more information

$ *kubectl describe pods*

Let's undo the rollout

$ *kubectl rollout undo deployments/hello-world*

The rollout command reverted the deployment to the previous known state of the image.
Updates are versioned and you can revert to any previously known state of a Deployment.

Check the status of the pods

$ *kubectl get pods*

All is well again! The rollback was successful.

Cleaning up

Let's clean up all the resources.

$ kubectl get deployments

$ kubectl delete deployment <deployment name>

$ kubectl get services

$ kubectl delete service <service name>

Note: Do not delete the kubernetes service

All pods associated with the deleted deployment should be deleted as well. Check with

$ kubectl get pods

To stop cluster

$ microk8s stop

To do -

Deploy the containerized webserver from Lab 7 under Kubernetes. You should describe your deployment in a YAML manifest file. The web server should have a replica set of 2 pods. Test the webserver from outside the cluster.

Now make a minor modification in the webserver code (say, change a print message). Generate and redeploy the new image using K8s rollout feature.

Hint: Host your Docker image in a local registry for use by Kubernetes

https://microk8s.io/docs/registry-built-in

Note: We'll examine automating this in the Continuous Integration / Continuous Deployment lab.