EXPERIMENT 9

AIM: Working with Kubernetes (Multi Node Cluster)

Step 1 - Initialise Master

Kubeadm has been installed on the nodes. Packages are available for Ubuntu 16.04+, CentOS 7 or HypriotOS v1.0.1+.

The first stage of initialising the cluster is to launch the master node. The master is responsible for running the control plane components, etcd and the API server. Clients will communicate to the API to schedule workloads and manage the state of the cluster.

Task

The command below will initialise the cluster with a known token to simplify the following steps.

```
kubeadm init --token=102952.1a7dd4cc8d1f4cc5 --kubernetes-version
$(kubeadm version -o short)
```

In production, it's recommend to exclude the token causing kubeadm to generate one on your behalf.

To manage the Kubernetes cluster, the client configuration and certificates are required. This configuration is created when *kubeadm* initialises the cluster. The command copies the configuration to the users home directory and sets the environment variable for use with the CLI.

```
sudo cp /etc/kubernetes/admin.conf $HOME/
sudo chown $(id -u):$(id -g) $HOME/admin.conf
export KUBECONFIG=$HOME/admin.conf
```

Step 2 - Deploy Container Networking Interface (CNI)

The Container Network Interface (CNI) defines how the different nodes and their workloads should communicate. There are multiple network providers available, some are listed.

Task

In this scenario we'll use WeaveWorks. The deployment definition can be viewed at

```
cat /opt/weave-kube.yaml
```

This can be deployed using `kubectl apply`.

```
kubectl apply -f /opt/weave-kube.yaml
```

Weave will now deploy as a series of Pods on the cluster. The status of this can be viewed using the command

```
kubectl get pod -n kube-system
```

When installing Weave on your cluster,
visit https://www.weave.works/docs/net/latest/kube-addon/ for details.

Step 3 - Join Cluster

Once the Master and CNI has initialised, additional nodes can join the cluster as long as they have the correct token. The tokens can be managed via `kubeadm token`, for example

```
kubeadm token list
```.

Task

On the second node, run the command to join the cluster providing the IP address of the Master node.

```
kubeadm join --discovery-token-unsafe-skip-ca-verification --
token=102952.1a7dd4cc8d1f4cc5 172.17.0.32:6443
```

This is the same command provided after the Master has been initialised.

The `--discovery-token-unsafe-skip-ca-verification` tag is used to bypass the Discovery Token verification. As this token is generated dynamically, we couldn't include it within the steps. When in production, use the token provided by `kubeadm init`.

Step 4 - View Nodes

The cluster has now been initialised. The Master node will manage the cluster, while our one worker node will run our container workloads.

Task

The Kubernetes CLI, known as *kubectl*, can now use the configuration to access the cluster. For example, the command below will return the two nodes in our cluster.

```
kubectl get nodes
```

Step 5 - Deploy Pod

The state of the two nodes in the cluster should now be Ready. This means that our deployments can be scheduled and launched.

Using Kubectl, it's possible to deploy pods. Commands are always issued for the Master with each node only responsible for executing the workloads.

The command below create a Pod based on the Docker Image *katacoda/docker-http-server*.

```
kubectl create deployment http --image=katacoda/docker-http-server:latest
```

The status of the Pod creation can be viewed using

```
kubectl get pods
```

Once running, you can see the Docker Container running on the node.

```
docker ps | grep docker-http-server
```

Step 6 - Deploy Dashboard

Kubernetes has a web-based dashboard UI giving visibility into the Kubernetes cluster.

Task

Deploy the dashboard yaml with the command

```
kubectl apply -f dashboard.yaml
```

The dashboard is deployed into the *kube-system* namespace. View the status of the deployment with

```
kubectl get pods -n kube-system
```

A ServiceAccount is required to login. A ClusterRoleBinding is used to assign the new ServiceAccount (*admin-user*) the role of *cluster-admin* on the cluster.

```
cat <<EOF | kubectl create -f -
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kube-system
EOF
```

This means they can control all aspects of Kubernetes. With ClusterRoleBinding and RBAC, different level of permissions can be defined based on security requirements. More information on creating a user for the Dashboard can be found in the Dashboard documentation.

Once the ServiceAccount has been created, the token to login can be found with:

```
kubectl -n kube-system describe secret $(kubectl -n kube-system get secret | grep admin-user | awk '{print $1}')
```

When the dashboard was deployed, it used externalIPs to bind the service to port 8443. This makes the dashboard available to outside of the cluster and viewable at https://2886795301-8443-simba08.environments.katacoda.com/

Use the *admin-user* token to access the dashboard.

For production, instead of externalIPs, it's recommended to use `kubectl proxy` to access the dashboard. See more details at https://github.com/kubernetes/dashboard.