**Launch Single Node Kubernetes Cluster (MiniKube)**

minikube version

minikube start --wait=false

//Details of the cluster and its health status can be discovered via

kubectl cluster-info

// This command shows all nodes that can be used to host our applications. Now we have only one node, and we can see that it’s status is ready

kubectl get nodes

//Used to deploy image

kubectl create deployment first-deployment --image=katacoda/docker-http-server

//get status of deployment

Kubectl get deployment

// The status of the deployment can be discovered via the running Pods –

kubectl get pods

// Once the container is running it can be exposed via different networking options, depending on requirements. One possible solution is NodePort, that provides a dynamic port to a container.This will create the service with type NodePort

kubectl expose deployment first-deployment --port=80 --type=NodePort

//Get all the services

Kubectl get services

export PORT=$(kubectl get svc first-deployment -o go-template='{{range.spec.ports}}{{if .nodePort}}{{.nodePort}}{{"\n"}}{{end}}{{end}}') echo "Accessing host01:31000" curl host01:$PORT

//Scaling up the pods

kubectl scale -n default deployment first-deployment --replicas=2

**Launch Multiple Node Kubernetes Cluster (Kubeadm)**

Following steps in master

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.

// 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)

//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

#### Deploy Container Networking Interface (CNI)

The Container Network Interface (CNI) defines how the different nodes and their workloads should communicate

<https://kubernetes.io/docs/concepts/cluster-administration/addons/>

//This can be deployed using kubectl apply.

kubectl apply -f /opt/weave-kube

//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

#### 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

//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 172.17.0.11:6443

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 172.17.0.11:6443

#### 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

// Use the following command to expose the container port 80 on the host 8000 binding to the external-ip of the host.

kubectl expose deployment http --external-ip="172.17.0.67" --port=8000 --target-port=80

//Use the command command to create a second http service exposed on port 8001.

kubectl run httpexposed --image=katacoda/docker-http-server:latest --replicas=1 --port=80 --hostport=8001

### Deploy Containers Using YAML

One of the most common Kubernetes object is the deployment object. The deployment object defines the container spec required, along with the name and labels used by other parts of Kubernetes to discover and connect to the application.

apiVersion: apps/v1

kind: Deployment

metadata:

name: webapp1

spec:

replicas: 1

selector:

matchLabels:

app: webapp1

template:

metadata:

labels:

app: webapp1

spec:

containers:

- name: webapp1

image: katacoda/docker-http-server:latest

ports:

- containerPort: 80

//This is deployed to the cluster with the command

kubectl create -f deployment.yaml

//As it's a Deployment object, a list of all the deployed objects can be obtained via

kubectl get deployment

//Details of individual deployments can be outputted with

kubectl describe deployment webapp1

#### Create Service

Kubernetes has powerful networking capabilities that control how applications communicate. These networking configurations can also be controlled via YAML.

apiVersion: v1

kind: Service

metadata:

name: webapp1-svc

labels:

app: webapp1

spec:

type: NodePort

ports:

- port: 80

nodePort: 30080

selector:

app: webapp1

Update the deployment.yaml file to increase the number of instances running. For example, the file should look like this:

replicas: 4

Updates to existing definitions are applied using kubectl apply. To scale the number of replicas, deploy the updated YAML file using

kubectl apply -f deployment.yaml

Instantly, the desired state of our cluster has been updated, viewable with

kubectl get deployment

Additional Pods will be scheduled to match the request.

kubectl get pods

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

#### Networking Introduction

alias k=kubectl

#### Cluster IP

Cluster IP is the default approach when creating a Kubernetes Service. The service is allocated an internal IP that other components can use to access the pods.

By having a single IP address it enables the service to be load balanced across multiple Pods.

Target Port:

Target ports allows us to separate the port the service is available on from the port the application is listening on. TargetPort is the Port which the application is configured to listen on. Port is how the application will be accessed from the outside.

<Service Object>

Spec:

Ports:

-port:8080

tragetPort:80

<Deployment Object>

Spec:

Containers:

Ports:

* containerPort: 80

After the service and pods have deployed, it can be accessed via the cluster IP as before, but this time on the defined port 8080.

#### NodePort

While TargetPort and ClusterIP make it available to inside the cluster, the NodePort exposes the service on each Node’s IP via the defined static port. No matter which Node within the cluster is accessed, the service will be reachable based on the port number defined.

<Service Object>

Spec:

Ports:

-port:80

nodePort: 30080

<Deployment Object>

Spec:

Containers:

Ports:

* containerPort: 80

The service can now be reached via the Node's IP address on the NodePort defined.

curl <node\_Ip>:30080

#### External IPs

Another approach to making a service available outside of the cluster is via External IP addresses.

<Service Object>

Spec:

Ports:

-port:80

externalIPs:

* HOSTIP

<Deployment Object>

Spec:

Containers:

Ports:

* containerPort: 80

kubectl describe svc/webapp1-externalip-svc

#### Load Balancer

When running in the cloud, such as EC2 or Azure, it's possible to configure and assign a Public IP address issued via the cloud provider. This will be issued via a Load Balancer such as ELB. This allows additional public IP addresses to be allocated to a Kubernetes cluster without interacting directly with the cloud provider.

<Service Object>

Spec:

Type: LoadBalancer

Ports:

-port:80

<Deployment Object>

Spec:

Containers:

Ports:

* containerPort: 80

kubectl get svc

curl <external\_ip>:80

#### Create Ingress Routing

Kubernetes have advanced networking capabilities that allow Pods and Services to communicate inside the cluster's network. An Ingress enables inbound connections to the cluster, allowing external traffic to reach the correct Pod.

Ingress enables externally-reachable urls, load balance traffic, terminate SSL, offer name based virtual hosting for a Kubernetes cluster.

In this scenario you will learn how to deploy and configure Ingress rules to manage incoming HTTP requests.

#### Deploy Ingress

The YAML file ingress.yaml defines a Nginx-based Ingress controller together with a service making it available on Port 80 to external connections using ExternalIPs. If the Kubernetes cluster was running on a cloud provider then it would use a LoadBalancer service type