1. <https://www.learnitguide.net/2018/08/what-is-kubernetes-learn-kubernetes.html>
2. <https://www.learnitguide.net/2018/08/install-and-configure-kubernetes-cluster.html>
3. <https://www.learnitguide.net/2018/08/create-kubernetes-deployment-services.html>
4. <https://www.learnitguide.net/2018/08/create-kubernetes-yaml-for-deployment.html>
5. <https://www.learnitguide.net/2020/03/kubernetes-volumes-explained.html>
6. <https://www.learnitguide.net/2020/03/kubernetes-persistent-volumes-and-claims.html>
7. <https://www.learnitguide.net/2020/03/add-new-worker-node-kubernetes-cluster.html>
8. <https://www.learnitguide.net/2020/05/kubernetes-services-explained-examples.html>

What is Kubernetes

What is Container Orchestration Engine

Why do we need Kubernetes

Kubernetes is the Greek word for helmsman or captain of a ship

Kubernetes is also referred to as k8s, as there are 8 characters between k and s

container management (orchestration) tool

developed by Google lab (& later donated to CNCF)

open source

written on Golang

also called K8s

Container Management / Orchestration tool

Container Orchestration tool or engine automates deploying, scaling and managing the containerized application on a group of servers

e.g.

Kubernetes

Docker Swarm

Apache Mesos Marathon

Docker is a tool designed to make it easier to deploy and run applications by using containers

Organizations have to use multiple containers to

Ensure availability

Load balancing

Scale-up and down based on user load

deploying

scheduling

scaling

load balancing

batch execution

rollbacks

monitoring

Docker - creates containers

Kubernetes - manages containers

Objective - learn:

**Features of Kubernetes.**

Automatic bin packing

Service discovery & load balancing

Storage orchestration

Self-healing

Automated rollouts and rollbacks

Secret & configuration management

Batch execution

Horizontal scaling

Automatic bin packing

Automatically places containers based on their resource requirements

like CPU & Memory (RAM),

while not sacrificing availability

Saves resources

Service discover & load balancing

Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them

With this system, Kubernetes has control over network and communication between pods and can load load balance across them

Storage Orchestration

Kubernetes allows to mount the storage system of your choice

Local

Cloud (AWS)

Network (NFS)

Self-healing

If a container fails - restarts container

If node dies - replaces and reschedule containers on other nodes

If container does not respond to user defined health check - kills container

This is taken care by Kubernetes ReplicationController

Secret & configuration management

Kubernetes manages secrets and configuration details for an application separately from the container image,

Deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

Automated rollouts and rollbacks

Rollout: deploy changes to the application or its configuration

Rollback: revert the changes & restore to the previous state

Kubernetes ensures there is no downtime during this process

Batch execution

Kubernetes supports batch execution, long-running jobs, and replaces failed containers

Horizontal scaling

In Kubernetes, we can scale up or down the containers

- using commands

- from the dashboard (kubernetes ui)

- automatically based on CPU usage

4 Components of Master node

API Server - for all communications ( JSON over HTTP API)

Scheduler - schedules pods on nodes

Controller Manager - runs controllers

Etcd - open source, distributed key-value database from CoreOS

3 Components of Worker node

kubelet

kube-proxy

Container runtime

Addons for DNS, Dashboard, monitoring, logging etc

**Step 1 - Start Minikube**

Minikube has been installed and configured in the environment. Check that it is properly installed, by running the *minikube version* command:

minikube version

Start the cluster, by running the *minikube start* command:

minikube start --wait=false

Great! You now have a running Kubernetes cluster in your online terminal. Minikube started a virtual machine for you, and a Kubernetes cluster is now running in that VM.

**Step 2 - Cluster Info**

The cluster can be interacted with using the *kubectl* CLI. This is the main approach used for managing Kubernetes and the applications running on top of the cluster.

Details of the cluster and its health status can be discovered via kubectl cluster-info

To view the nodes in the cluster using kubectl get nodes

If the node is marked as **NotReady** then it is still starting the components.

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 (it is ready to accept applications for deployment).

$ kubectl cluster-info

Kubernetes master is running at https://172.17.0.36:8443

KubeDNS is running at https://172.17.0.36:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

$ kubectl get nodes

NAME STATUS ROLES AGE VERSION

minikube Ready master 73s v1.17.3

**Step 3 - Deploy Containers**

With a running Kubernetes cluster, containers can now be deployed.

Using kubectl run, it allows containers to be deployed onto the cluster - kubectl create deployment first-deployment --image=katacoda/docker-http-server

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.

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

The command below finds the allocated port and executes a HTTP request.

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

The result is the container that processed the request.

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

deployment.apps/first-deployment created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

first-deployment-666c48b44-bb7vb 1/1 Running 0 79s

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

service/first-deployment exposed

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

$ echo "Accessing host01:$PORT"

Accessing host01:31146

$ curl host01:$PORT

<h1>This request was processed by host: first-deployment-666c48b44-bb7vb</h1>

$

**Step 4 - Dashboard**

Enable the dashboard using Minikube with the command minikube addons enable dashboard

Make the Kubernetes Dashboard available by deploying the following YAML definition. This should only be used on Katacoda.

kubectl apply -f /opt/kubernetes-dashboard.yaml

The Kubernetes dashboard allows you to view your applications in a UI. In this deployment, the dashboard has been made available on port *30000* but may take a while to start.

To see the progress of the Dashboard starting, watch the Pods within the *kube-system* namespace using kubectl get pods -n kubernetes-dashboard -w

Once running, the URL to the dashboard is <https://2886795300-30000-elsy02.environments.katacoda.com/>

Use "kubectl options" for a list of global command-line options (applies to all commands).

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

deployment.apps/first-deployment created

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

first-deployment-666c48b44-bb7vb 1/1 Running 0 79s

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

service/first-deployment exposed

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

$ echo "Accessing host01:$PORT"

Accessing host01:31146

$ curl host01:$PORT

<h1>This request was processed by host: first-deployment-666c48b44-bb7vb</h1>

$ minikube addons enable dashboard

\* The 'dashboard' addon is enabled

$ kubectl apply -f /opt/kubernetes-dashboard.yaml

namespace/kubernetes-dashboard configured

service/kubernetes-dashboard-katacoda created

$ kubectl get pods -n kubernetes-dashboard -w

NAME READY STATUS RESTARTS AGE

dashboard-metrics-scraper-7b64584c5c-2k98q 1/1 Running 0 100s

kubernetes-dashboard-79d9cd965-vrkz6 1/1 Running 0 101s

In this scenario you'll learn how to bootstrap a Kubernetes cluster using Kubeadm.

Kubeadm solves the problem of handling TLS encryption configuration, deploying the core Kubernetes components and ensuring that additional nodes can easily join the cluster. The resulting cluster is secured out of the box via mechanisms such as RBAC.

More details on Kubeadm can be found at <https://github.com/kubernetes/kubeadm>

**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 [here](https://kubernetes.io/docs/admin/addons/).

**Task**

In this scenario we'll use WeaveWorks. The deployment definition can be viewed at cat /opt/weave-kube

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

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

master $ **cat /opt/weave-kube**

apiVersion: v1

kind: List

items:

- apiVersion: v1

kind: ServiceAccount

metadata:

name: weave-net

labels:

name: weave-net

namespace: kube-system

- apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRole

metadata:

name: weave-net

labels:

name: weave-net

rules:

- apiGroups:

- ''

resources:

- pods

- namespaces

- nodes

verbs:

- get

- list

- watch

- apiGroups:

- extensions

resources:

- networkpolicies

verbs:

- get

- list

- watch

- apiGroups:

- 'networking.k8s.io'

resources:

- networkpolicies

verbs:

- get

- list

- watch

- apiGroups:

- ''

resources:

- nodes/status

verbs:

- patch

- update

- apiVersion: rbac.authorization.k8s.io/v1beta1

kind: ClusterRoleBinding

metadata:

name: weave-net

labels:

name: weave-net

roleRef:

kind: ClusterRole

name: weave-net

apiGroup: rbac.authorization.k8s.io

subjects:

- kind: ServiceAccount

name: weave-net

namespace: kube-system

- apiVersion: rbac.authorization.k8s.io/v1beta1

kind: Role

metadata:

name: weave-net

namespace: kube-system

labels:

name: weave-net

rules:

- apiGroups:

- ''

resources:

- configmaps

resourceNames:

- weave-net

verbs:

- get

- update

- apiGroups:

- ''

resources:

- configmaps

verbs:

- create

- apiVersion: rbac.authorization.k8s.io/v1beta1

kind: RoleBinding

metadata:

name: weave-net

namespace: kube-system

labels:

name: weave-net

roleRef:

kind: Role

name: weave-net

apiGroup: rbac.authorization.k8s.io

subjects:

- kind: ServiceAccount

name: weave-net

namespace: kube-system

- apiVersion: extensions/v1beta1

kind: DaemonSet

metadata:

name: weave-net

labels:

name: weave-net

namespace: kube-system

spec:

# Wait 5 seconds to let pod connect before rolling next pod

minReadySeconds: 5

template:

metadata:

labels:

name: weave-net

spec:

containers:

- name: weave

command:

- /home/weave/launch.sh

env:

- name: HOSTNAME

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: spec.nodeName

image: 'weaveworks/weave-kube:2.5.1'

imagePullPolicy: IfNotPresent

readinessProbe:

httpGet:

host: 127.0.0.1

path: /status

port: 6784

resources:

requests:

cpu: 10m

securityContext:

privileged: true

volumeMounts:

- name: weavedb

mountPath: /weavedb

- name: cni-bin

mountPath: /host/opt

- name: cni-bin2

mountPath: /host/home

- name: cni-conf

mountPath: /host/etc

- name: dbus

mountPath: /host/var/lib/dbus

- name: lib-modules

mountPath: /lib/modules

- name: xtables-lock

mountPath: /run/xtables.lock

readOnly: false

- name: weave-npc

env:

- name: HOSTNAME

valueFrom:

fieldRef:

apiVersion: v1

fieldPath: spec.nodeName

image: 'weaveworks/weave-npc:2.5.1'

imagePullPolicy: IfNotPresent

#npc-args

resources:

requests:

cpu: 10m

securityContext:

privileged: true

volumeMounts:

- name: xtables-lock

mountPath: /run/xtables.lock

readOnly: false

hostNetwork: true

hostPID: true

restartPolicy: Always

securityContext:

seLinuxOptions: {}

serviceAccountName: weave-net

tolerations:

- effect: NoSchedule

operator: Exists

volumes:

- name: weavedb

hostPath:

path: /var/lib/weave

- name: cni-bin

hostPath:

path: /opt

- name: cni-bin2

hostPath:

path: /home

- name: cni-conf

hostPath:

path: /etc

- name: dbus

hostPath:

path: /var/lib/dbus

- name: lib-modules

hostPath:

path: /lib/modules

- name: xtables-lock

hostPath:

path: /run/xtables.lock

type: FileOrCreate

updateStrategy:

type: RollingUpdate

master $ **kubectl apply -f /opt/weave-kube**

serviceaccount/weave-net created

clusterrole.rbac.authorization.k8s.io/weave-net created

clusterrolebinding.rbac.authorization.k8s.io/weave-net created

role.rbac.authorization.k8s.io/weave-net created

rolebinding.rbac.authorization.k8s.io/weave-net created

daemonset.extensions/weave-net created

master **$ kubectl get pod -n kube-system**

NAME READY STATUS RESTARTS AGE

coredns-fb8b8dccf-dnhw9 1/1 Running 0 5m50s

coredns-fb8b8dccf-x47sk 1/1 Running 0 5m50s

etcd-master 1/1 Running 0 5m5s

kube-apiserver-master 1/1 Running 0 5m8s

kube-controller-manager-master 1/1 Running 0 5m13s

kube-proxy-2w4cw 1/1 Running 0 5m50s

kube-scheduler-master 1/1 Running 1 4m45s

weave-net-r65fw 2/2 Running 0 68s

master $

**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 172.17.0.17:6443

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.

**kubeadm token list**

TOKEN TTL EXPIRES USAGES DESCRIPTION EXTRA GROUPS

102952.1a7dd4cc8d1f4cc5 23h 2020-05-13T02:00:43Z authentication,signing The default bootstrap token generated by 'kubeadm init'. system:bootstrappers:kubeadm:default-node-token

master $

**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](https://github.com/kubernetes/dashboard/wiki/Creating-sample-user).

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://2886795290-8443-cykoria03.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>.