**Task 1: Ubuntu VM creation, docker apache container**

- Create an Ubuntu VM on a local machine.

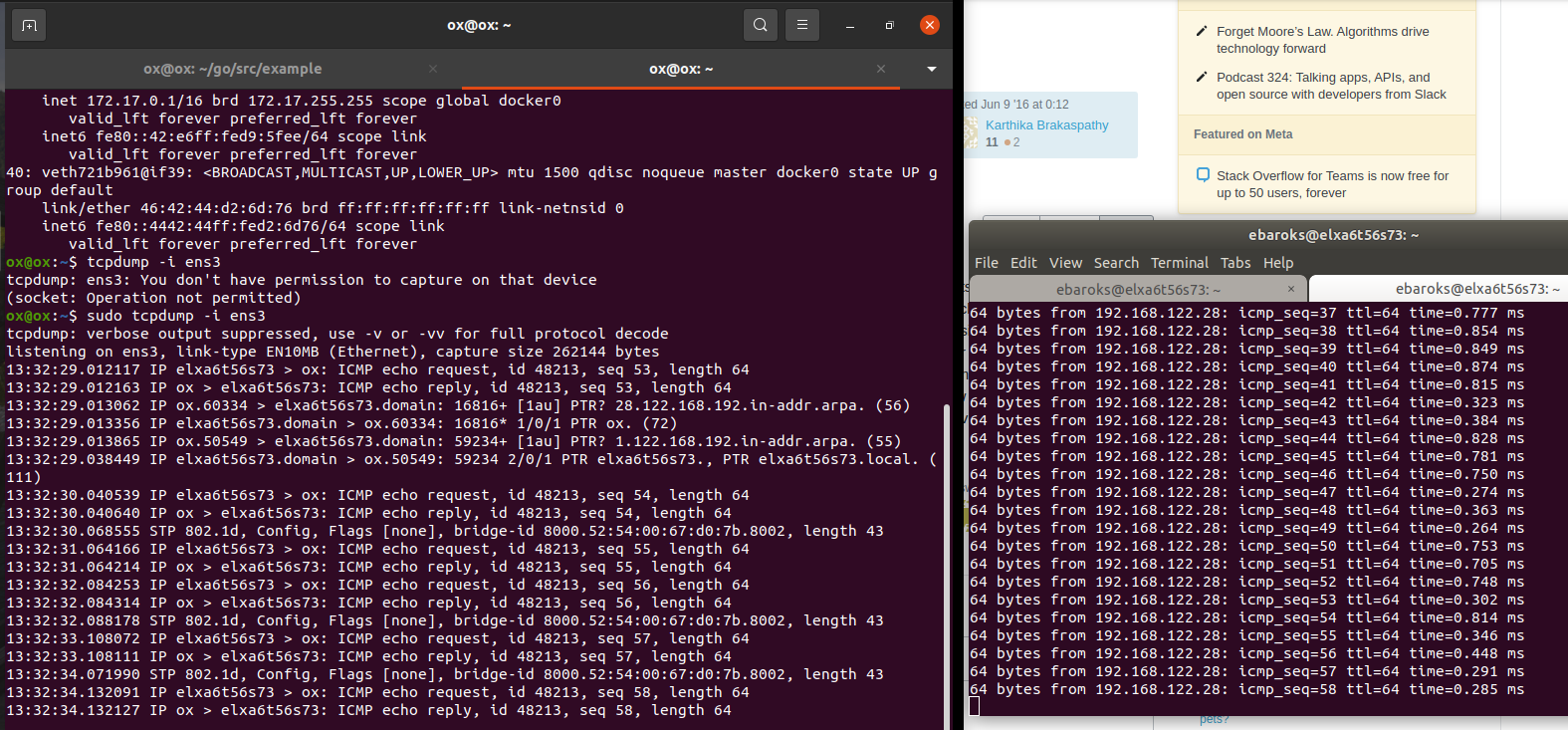
- make sure you are able to ping the VM IP from the laptop

- Install docker on the VM

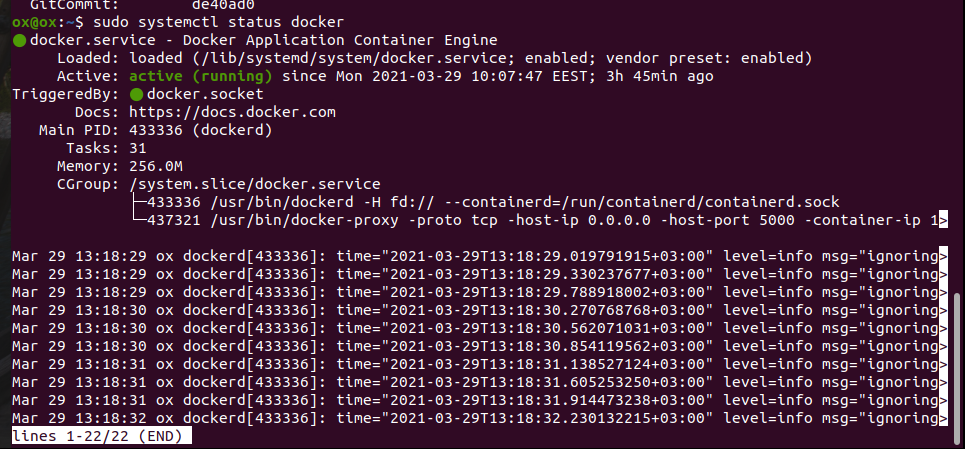
- Run apache2 container

- Open browser in laptop and access apache2 container test page which was started in the previous step.

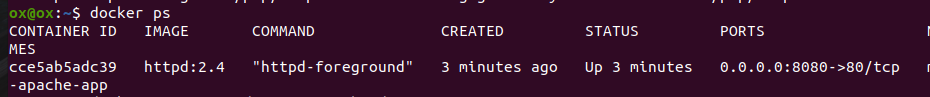
Checking connectivity between VM and host:



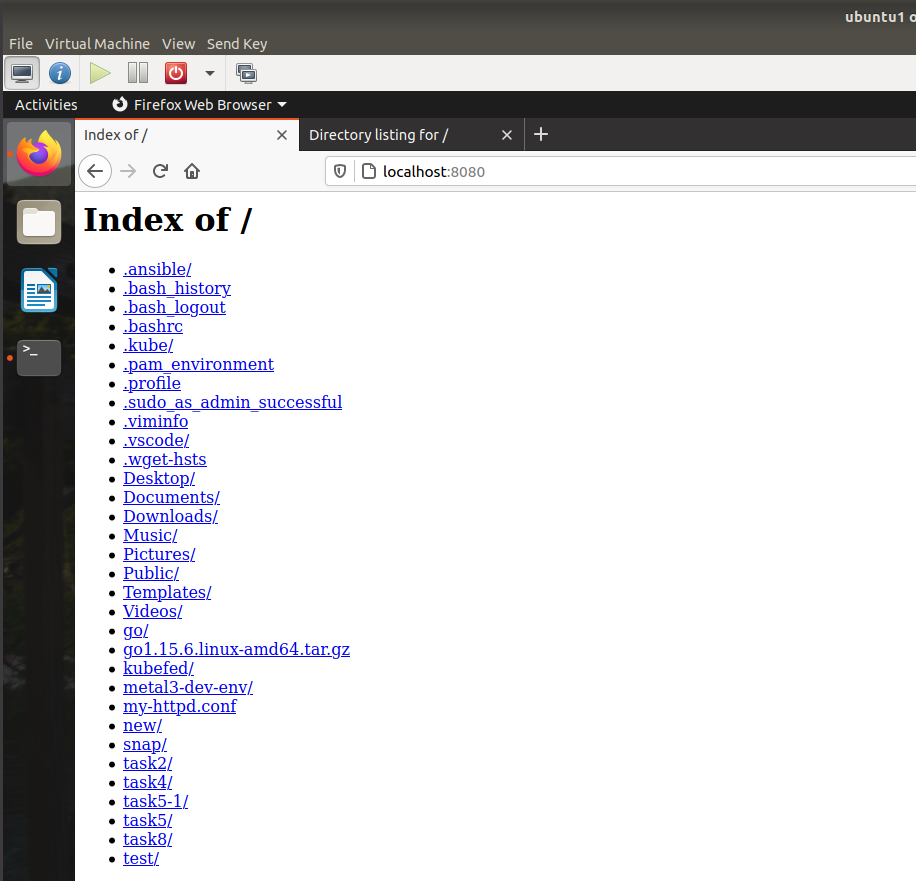
Checking the docker status:



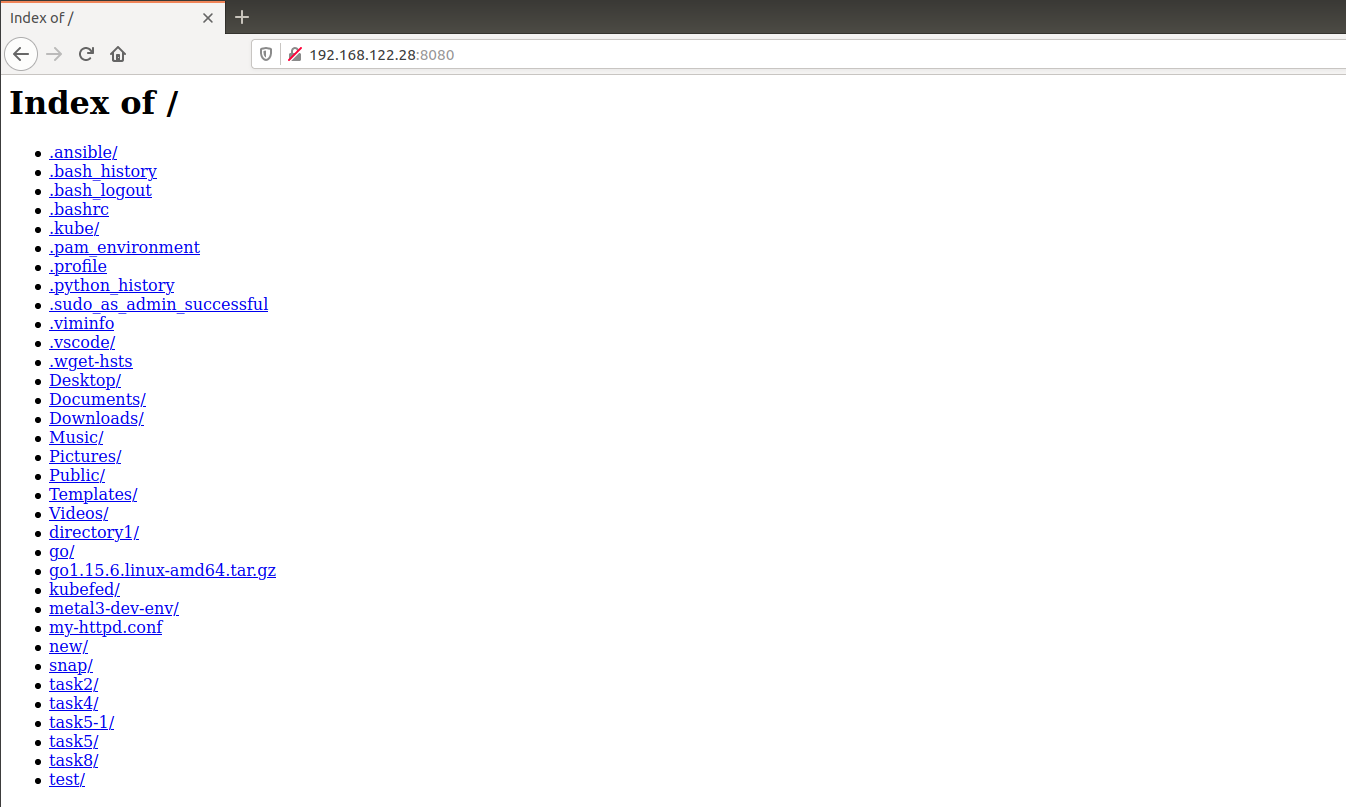
Running contaner $ docker run -dit --name my-apache-app -p 8080:80 -v "$PWD":/usr/local/apache2/htdocs/ httpd:2.4



Access from VM:



Access from host machine:



**Task 2: Container RUN\_COUNT**

- Create a container by using new Dockerfile. The container should run a shell script which creates a test file and write this string in file "RUN\_COUNT = 1" in it and the container should exit. When you run the container next time, it should read the file which was created on the previous run. Increment the read count value and write the updated value to the file. e.g the second time you run the container, the value in the file should be "RUN\_COUNT = 2". And similarly, next time it should be "RUN\_COUNT = 3"

Dockerfile

FROM ubuntu:18.04

WORKDIR /opt

CMD /opt/script.sh

script.sh

#!/bin/bash

if [ ! -f /opt/test.txt ];

then

touch /opt/test.txt

echo "RUN\_COUNT = 1" > /opt/test.txt

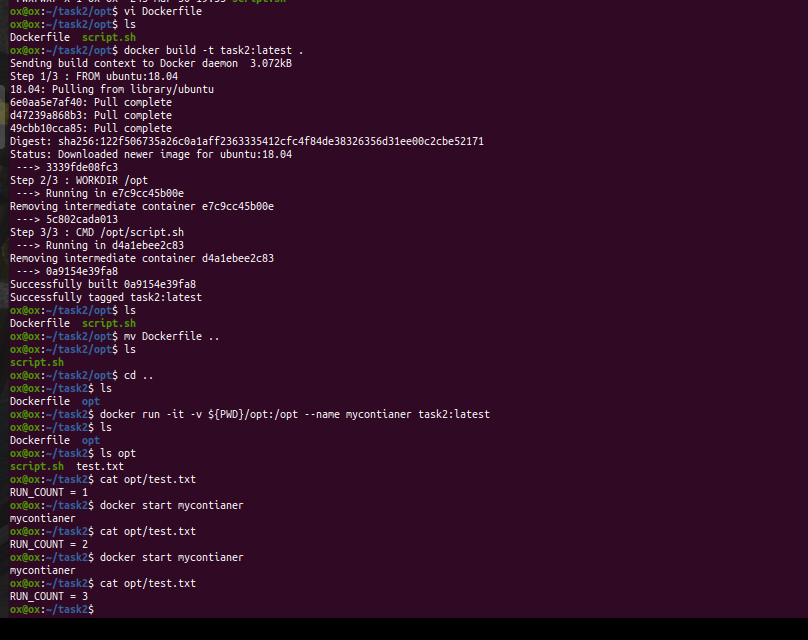
else

value=`awk '{print $3}' /opt/test.txt`

i=$(($value+1))

echo "RUN\_COUNT = $i" > /opt/test.txt

fi



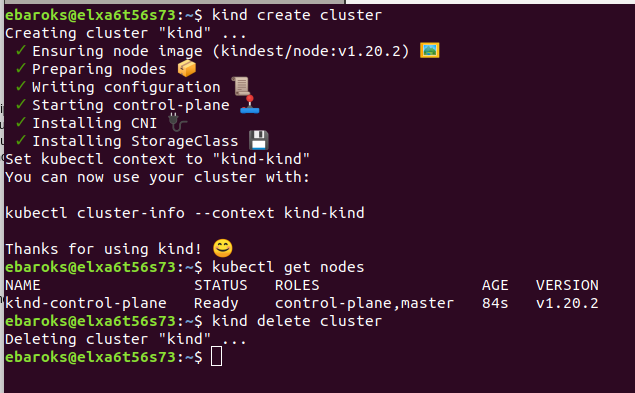
**Task 3: Create Kubernetes cluster in Ubuntu VM (Using kind cluster)**

- Create a kind cluster in the Ubuntu VM by following the instructions here. https://kind.sigs.k8s.io/docs/user/quick-start/

- Once cluster is setup, run "kubectl get nodes" and verify how many kubernetes nodes exist in the cluster

- destroy the cluster

- create the kind cluster again with 4 worker nodes.



Config for multiple nodes creation:

# this config file contains all config fields with comments

# NOTE: this is not a particularly useful config file

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

# patch the generated kubeadm config with some extra settings

kubeadmConfigPatches:

- |

apiVersion: kubelet.config.k8s.io/v1beta1

kind: KubeletConfiguration

evictionHard:

nodefs.available: "0%"

# patch it further using a JSON 6902 patch

kubeadmConfigPatchesJSON6902:

- group: kubeadm.k8s.io

version: v1beta2

kind: ClusterConfiguration

patch: |

- op: add

path: /apiServer/certSANs/-

value: my-hostname

# 1 control plane node and 3 workers

nodes:

# the control plane node config

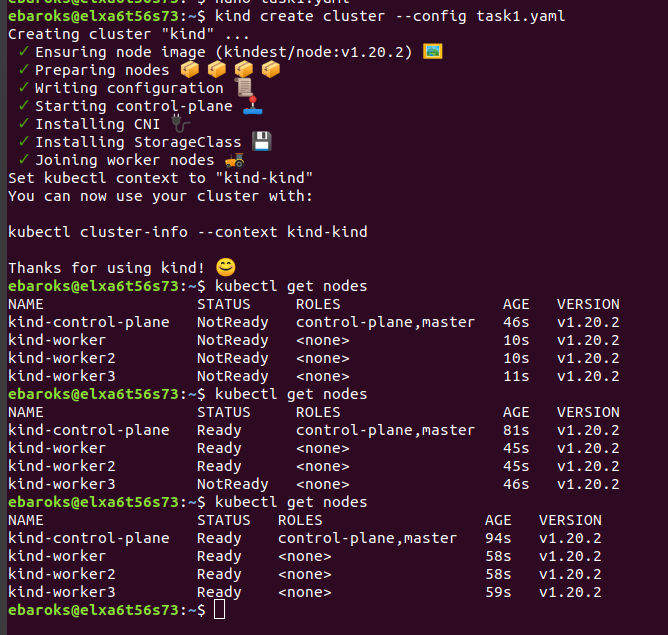
- role: control-plane

# the three workers

- role: worker

- role: worker

- role: worker



**Task 4: Golang for reading YAML file**

- Write a program in golang to read the following YAML file <https://stackoverflow.com/questions/30947534/how-to-read-a-yaml-file>

**Text**

---

this:

is:

- my

- test

- yaml file

- Modify it the YAML file so that it looks like below

**Text**

---

this:

is:

my:

- test

- yaml file

task4.go

package main

import (

"fmt"

"io/ioutil"

"log"

"gopkg.in/yaml.v2"

)

type Input struct {

This struct {

Is []string `yaml:"is"`

} `yaml:"this"`

}

type Output struct {

This struct {

Is struct {

My []string `yaml:"my"`

} `yaml:"is"`

} `yaml:"this"`

}

func getInput() Input {

var c Input

yamlFile, err := ioutil.ReadFile("task4.yaml")

if err != nil {

log.Printf("yamlFile.Get err #%v ", err)

}

err = yaml.Unmarshal(yamlFile, &c)

if err != nil {

log.Fatalf("Unmarshal: %v", err)

}

return c

}

func getOutput(c Input) Output {

var o Output = Output{}

o.This.Is.My = c.This.Is

o.This.Is.My = []string{c.This.Is[1], c.This.Is[2]}

return o

}

func main() {

c := getInput()

o := getOutput(c)

output, err := yaml.Marshal(o)

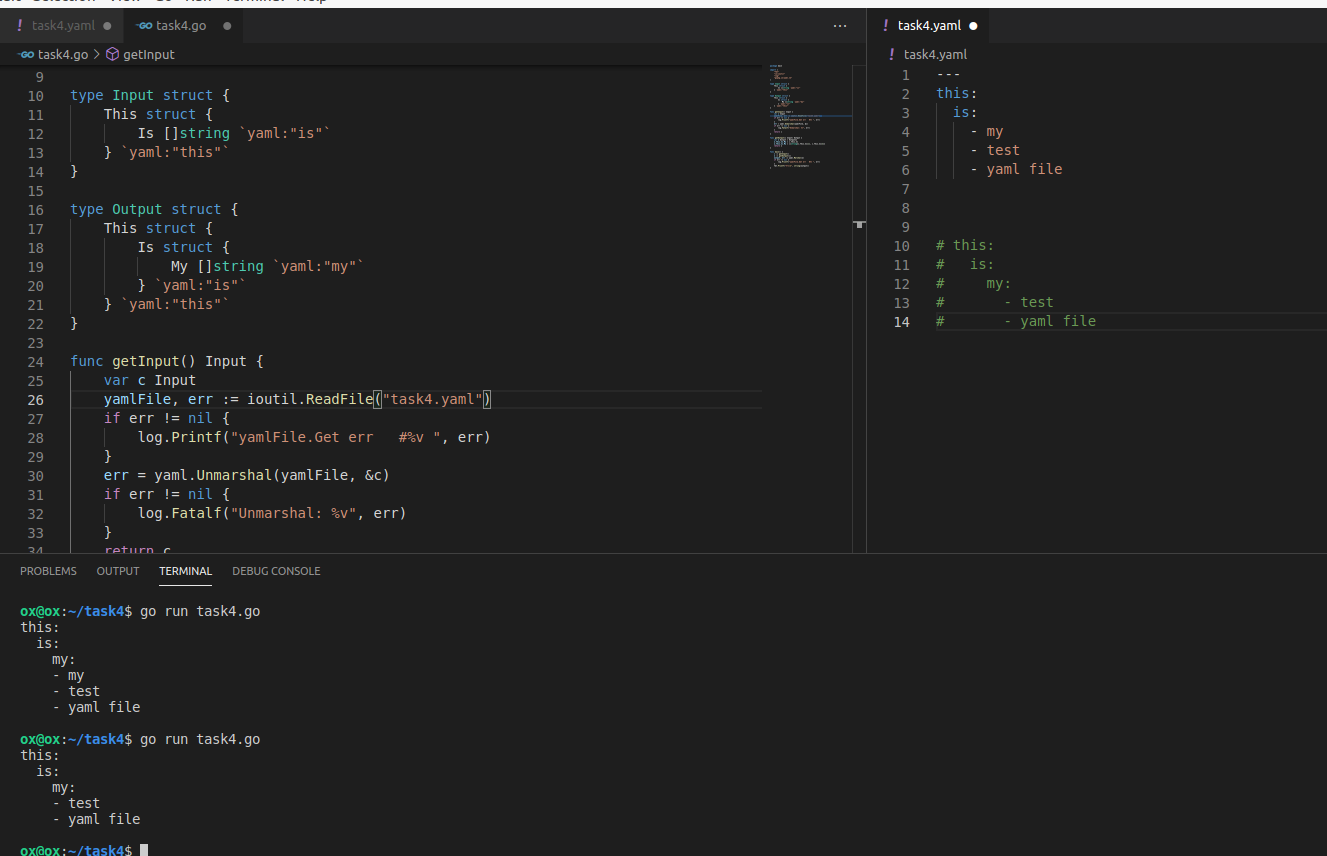
if err != nil {

log.Printf("yamlFile.Get err #%v ", err)

}

fmt.Printf("%+v\n", string(output))

}



**Task 5: Use Golang Cobra to read input parameters**

- Modify the program in Task 4 to take two parameters as an input, which are described below. Use Golang Cobra framework to read input parameters:

Param1 : full path for input yaml file

Param2 : username

- This time, it should read the yaml file modified by the program in the last step and modify "my:" to give the String username "username:". Check the example below

**Text**

---

this:

is:

username:

- test

- yaml file

MAIN.GO

package main

import (

"task5\_yaml/cmd"

)

func main() {

cmd.Execute()

}

ROOT.GO

package cmd

import (

"fmt"

"log"

"regexp"

"github.com/spf13/cobra"

"gopkg.in/yaml.v2"

)

var (

// Used for flags.

rootCmd = &cobra.Command{

Use: "task5",

Short: "task5 application",

Args: cobra.ExactArgs(2),

Run: func(cmd \*cobra.Command, args []string) {

validateUsername(args[1])

validateYamlFile(args[0])

C := getInput(args[0], args[1])

output, err := yaml.Marshal(C)

if err != nil {

log.Printf("yamlFile.Get err #%v ", err)

}

//C.This.Is = map[string][]string{args[1]: C.This.Is["username"]}

//fmt.Println(C)

fmt.Printf("%+v\n", string(output))

},

}

)

func validateUsername(username string) {

r, \_ := regexp.Compile("^[a-zA-z][a-zA-z\_0-9]{0,10}")

if !r.MatchString(username) {

log.Fatalf("Provide a valid username")

}

if len(username) > 10 || len(username) < 4 {

log.Fatalf("Provide at least 4 characters, or your username is too long")

}

}

func validateYamlFile(filename string) {

r, \_ := regexp.Compile("^.\*(yaml)$")

if !r.MatchString(filename) {

log.Fatalf("Provide a yaml file")

}

}

func Execute() error {

return rootCmd.Execute()

}

TASK5.GO

package cmd

import (

"io/ioutil"

"log"

"gopkg.in/yaml.v2"

)

type Input struct {

This struct {

Is map[string][]string `yaml:"is"`

} `yaml:"this"`

}

func getInput(filename string, username string) Input {

var c Input

yamlFile, err := ioutil.ReadFile(filename)

if err != nil {

log.Printf("yamlFile.Get err #%v ", err)

}

err = yaml.Unmarshal(yamlFile, &c)

if err != nil {

log.Fatalf("Unmarshal: %v", err)

}

c.This.Is[username] = c.This.Is["my"]

delete(c.This.Is, "my")

return c

}

**Task 6: Packaging and running program in kubernetes**

- Create a container to package the binary built in Task 5. make sure the container does not exit after the binary has exited and hang indefinitely until the container is killed.

- Push the container to any publicly accessible docker registry

- write manifests so that the container is run on the kind cluster created in Task 3

- You should be able to provide the input YAML file using a config map.

- Check the output of the program by going to the container running in Kubernetes and verify the file is modified.

Dockerfile

FROM golang:1.16-alpine AS build

WORKDIR /app/

#COPY main.go go.\* /app/

COPY . /app

RUN CGO\_ENABLED=0 go build

FROM alpine

WORKDIR /app/

COPY script.sh /app

COPY --from=build /app/task5\_yaml /app/task5\_yaml

ENTRYPOINT ["/app/script.sh"]

script.sh

#!/bin/sh

./task5\_yaml $\*

while true; do sleep 1000; done

manifest.yaml

apiVersion: v1

kind: Pod

metadata:

name: my-container-task6

spec:

hostNetwork: true

containers:

- name: task5image

image: ebaroks/imagetask6

args: ['/tmp/task5.yaml', 'username']

volumeMounts:

- name: configvolume

mountPath: /tmp/task5.yaml

subPath: task5.yaml

env:

- name: CONFIG\_USERNAME

valueFrom:

secretKeyRef:

name: task7-secret

key: username

- name: CONFIG\_PASSWORD

valueFrom:

secretKeyRef:

name: task7-secret

key: password

volumes:

- name: configvolume

configMap:

name: task6

configmap.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: task6

data:

task5.yaml: |

this:

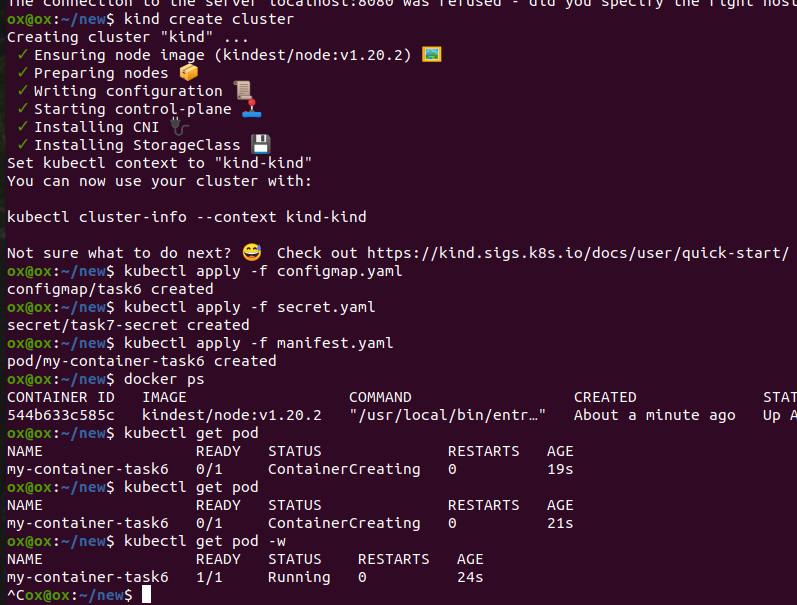
is:

my:

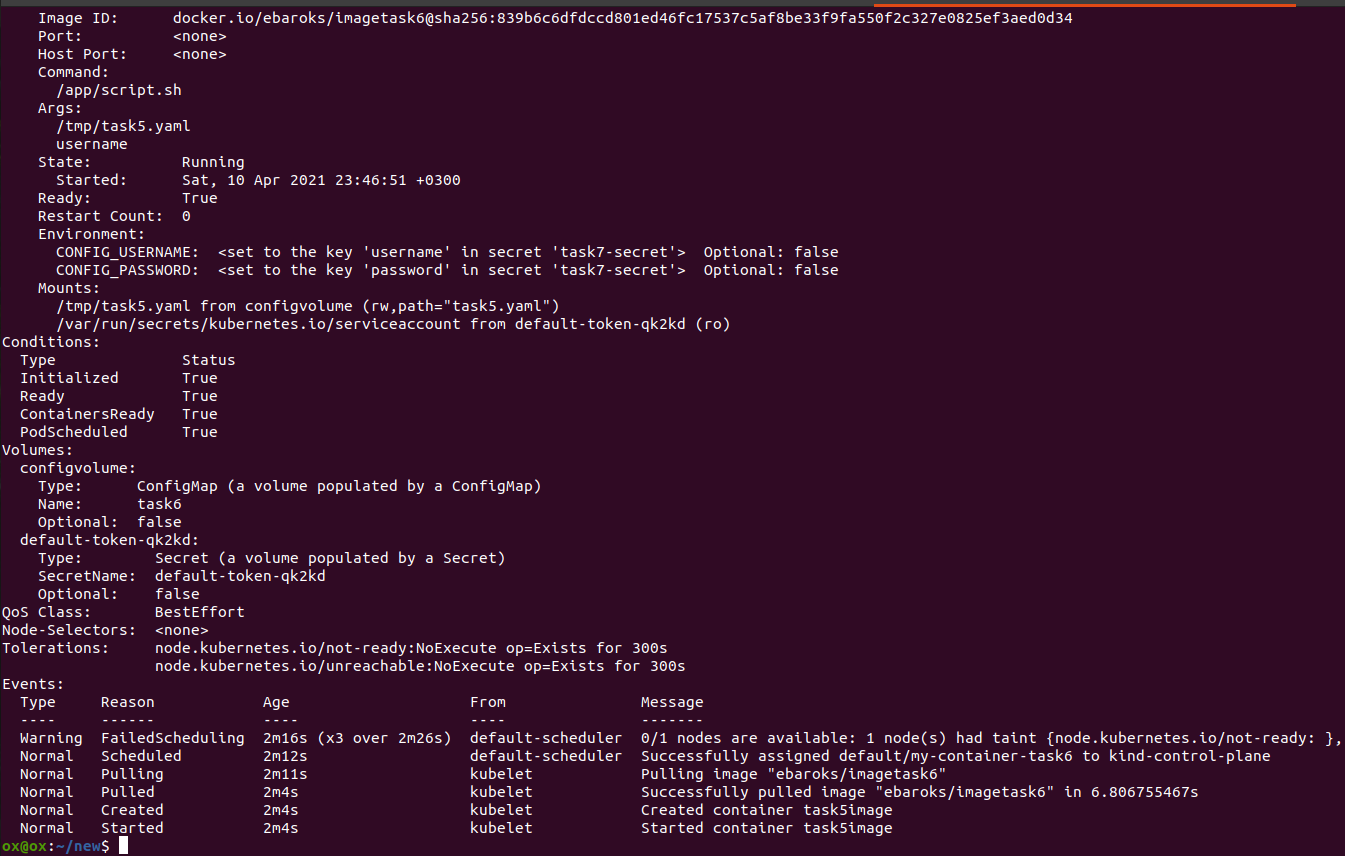
- test

- yaml file

Running the container with a binary inside:



Description of the running container



**Task 7: Using Kubernetes secrets**

- Now modify the manifest in Task 6 and provide the input file using a secret this time.

secret.yaml

apiVersion: v1

kind: Secret

metadata:

name: task7-secret

type: Opaque

data:

username: dXNlcm5hbWU=

password: cGFzc3dvcmQ=

Secrets are attached to the Pod via Environment variables (CONFIG\_USERNAME, CONFIG\_PASSWORD).

Configmap from the task6 is attached to the Pod via Volume.

**Task 8: Using kubernetes client-go library**

- Write a small program that reads Kubernetes manifest files created in Task 6 and create the same container using the golang code, without using the kubectl.

- You can use the client-go library in golang for this purpose.

The following resources were used to write a program:

<https://github.com/kubernetes/client-go/>

<https://pkg.go.dev/k8s.io/api/core/v1>

<https://v1-16.docs.kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#pod-v1-core>

<https://gist.github.com/dlorenc/2ac000a25656447a0c06e79150407b4c> k8s yaml parse

<https://github.com/hossainemruz/k8s-client-go-practice/blob/56eeb12f1ce54d3fe4d428e7546d70cd5bde3b50/main.go> client go clientset

task5.go

package cmd

import (

"bufio"

"context"

"fmt"

"io"

"log"

"os"

"path/filepath"

core "k8s.io/api/core/v1"

metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"

"k8s.io/apimachinery/pkg/runtime"

"k8s.io/apimachinery/pkg/util/yaml"

"k8s.io/client-go/kubernetes"

"k8s.io/client-go/kubernetes/scheme"

"k8s.io/client-go/tools/clientcmd"

"k8s.io/client-go/util/homedir"

)

func deployManifest(filename string, username string) {

f, err := os.Open(filename)

if err != nil {

log.Fatalf("yamlFile.Get err #%v ", err)

}

defer f.Close()

b := bufio.NewReader(f)

r := yaml.NewYAMLReader(b)

doc, err := r.Read()

if err == io.EOF {

log.Fatalf("Empty Yaml file: %s\n%s", filename, err)

}

if err != nil {

log.Fatal(err)

}

d := scheme.Codecs.UniversalDeserializer()

obj, \_, err := d.Decode(doc, nil, nil)

if err != nil {

log.Fatalf("could not decode yaml: %s\n%s", filename, err)

}

fmt.Println(obj)

fmt.Println("---------------")

clientset := createClientSet()

deployPod(clientset, obj)

}

func createClientSet() \*kubernetes.Clientset {

// var kubeconfig \*string

// if home := homedir.HomeDir(); home != "" { // check if machine has home directory.

// // read kubeconfig flag. if not provided use config file $HOME/.kube/config

// kubeconfig = flag.String("kubeconfig", filepath.Join(home, ".kube", "config"), "(optional) absolute path to the kubeconfig file")

// } else {

// kubeconfig = flag.String("kubeconfig", "", "absolute path to the kubeconfig file")

// }

// flag.Parse()

home := homedir.HomeDir()

kubeconfig := filepath.Join(home, ".kube", "config")

// build configuration from the config file.

config, err := clientcmd.BuildConfigFromFlags("", kubeconfig)

if err != nil {

panic(err)

}

// create kubernetes clientset. this clientset can be used to create,delete,patch,list etc for the kubernetes resources

clientset, err := kubernetes.NewForConfig(config)

if err != nil {

panic(err)

}

return clientset

}

func deployPod(clientset \*kubernetes.Clientset, obj runtime.Object) {

// now create the pod in kubernetes cluster using the clientset

podobj := obj.(\*core.Pod)

\_, err := clientset.CoreV1().Pods("default").Create(context.Background(), podobj,

metav1.CreateOptions{})

if err != nil {

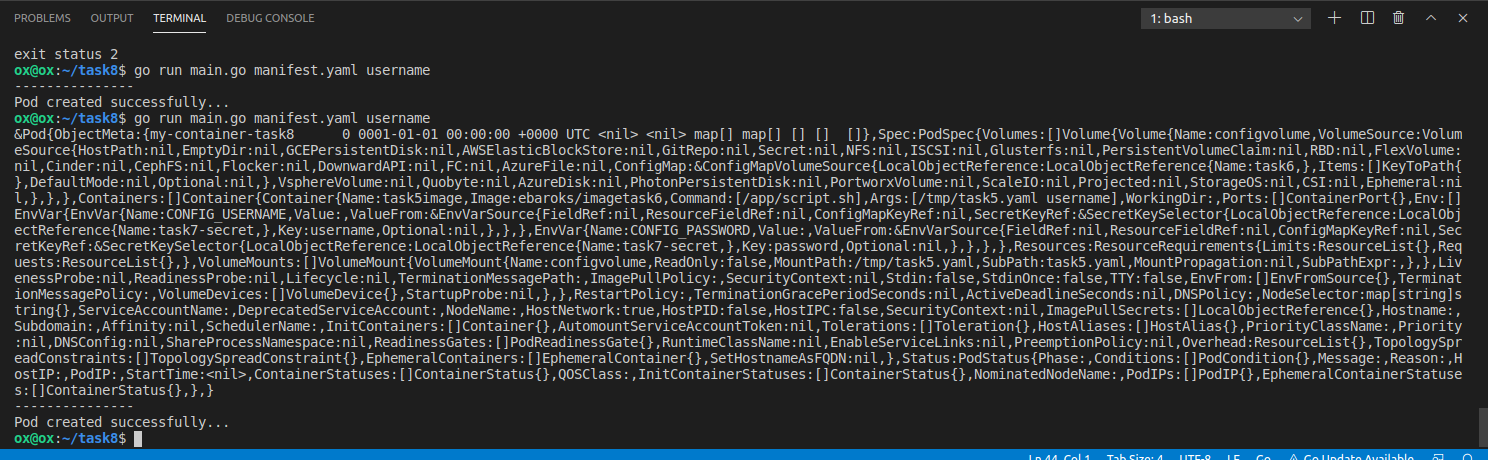
panic(err)

}

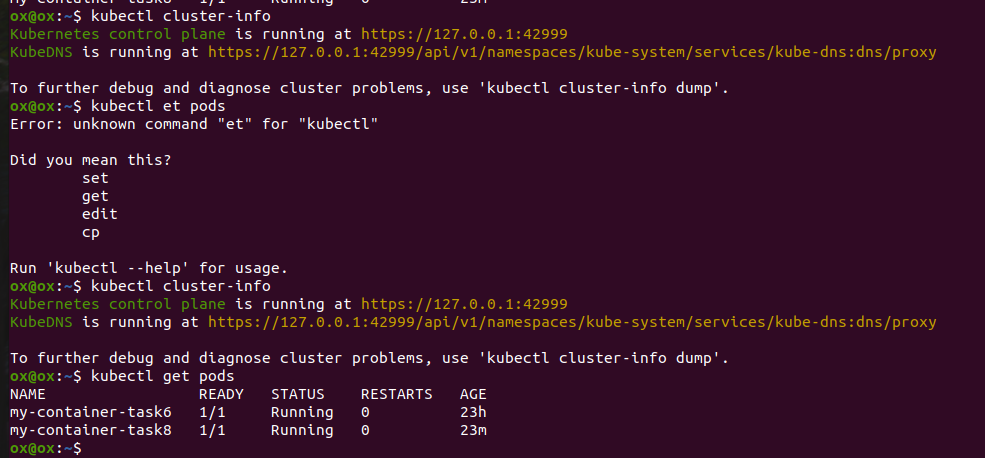
fmt.Println("Pod created successfully...")

}

Output of the running program:



Check container creation:



The code written for the tasks can be found here: <https://github.com/oksanabaranova/Learning_tasks>