30/06/24

Practice deployment file writing.

Class 5

pulumi

From vs code terminal, ssh to ubuntu4-gui

Scheduler - Part 1

How scheduling works?

Every pod has a field called nodename that, by default, is not set. Kubernetes adds it automatically. The scheduler goes through all the pods and looks for those that don't have this property set. Those are the candidates for scheduling. It then identifies the right node for the pod, by running the scheduling algorithm. Once identified, it schedules the pod on the node by setting the node name property to the name of the node by creating a binding object.

30/06/24

If there is no scheduler, the pods continue to be in a pending state.

What if the pod is already created and want to be assigned the pod to a node?

Kubernetes won't allow to modify nodename property of a pod. Another way to assign a node to an existing pod is to create a binding object and send a post request to the pod binding api, thus mimicking what the actual scheduler does. (pod-bind-definition.yaml). In the binding object, specify a target node with the name of the node. Then send a post request to the pods binding api with the data set to the binding object in a JSON format.

pod-binding-definition.yaml

```
apiVersion: v1
Kind: Binding
metadata:
    name: nginx
```

(same level as metadata)

```
target:
  apiVersion: v1
  Kind: Node
  name: node02
```

30/06/24

So, converting the yaml file into its equivalent json format is a must.

01/07/24

Class 5

Deploy ~~k3s~~ kubernetes using k3s on AWS ~~using~~ by Pulumi

From vscode terminal ssh to ubuntu24-gui

```
sudo apt update
apt install python3.12-venv
```

~~python3 -m venv venv~~

```
sudo mkdir aws-pulumi
sudo chown -R $USER:$(id -g) aws-pulumi/
cd aws-pulumi
~~sudo~~ python3 -m venv venv
source venv/bin/activate
pip install pulumi pulumi-aws
```

```
curl -fsSL https://get.pulumi.com | sh
```

01/07/24

```
export PATH=$PATH:/home/ubuntu24-gui/.pulumi/bin
source ~/.bashrc
source venv/bin/activate
```

Create a new pulumi project

```
pulumi new python -y --dir aws-pulumi/
```

It will ask for access token, login to the pulumi account and create token.

To log in to pulumi

```
pulumi login
```

It will ask for token, paste it.

```
Logged in ~~as~~ to pulumi.com as aminul (. . . . . )
```

```
pulumi new python -y --dir aws-pulumi/
```

~~In VS~~ From vs code remotely connect to ~~the~~ a machine -

Install 'Remote SSH' extension, from search [ ]

(At the top-middle) → click → Show and Run Commands → Remote-SSH

04/09/24

Click on it and follow the rest of the procedures.]

[V.V.I - first complete the remote ssh connection

[if pulumi is installed before remove it -

```
rm /usr/local/bin/pulumi
rm -rf ~/.pulumi
rm -rf ~/Pulumi
```

[and then install using 'curl -fsSL ...']

Install aws cli →

```
sudo apt-get update
sudo apt-get install curl unzip -y
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

In the '__main__.py' file

01/07/24

In the '--main--.py' file

```
import pulumi
import pulumi_aws as aws

vpc = aws.ec2.Vpc("my-vpc",
    cidr_block="10.10.0.0/16",
    enable_dns_hostnames=True,
    enable_dns_support=True,
)
```

To create a new vpc and ~~dns~~ private dns of the vpc.

In the terminal pulumi stack select aminul/aws-k3s-pulumi/dev

```
pulumi up
```

View in Browser (ctrl+O): https://app.pulumi.com/...

E...

Without aws credentials pulumi will show error.

In another terminal-

```
aws configure

AWS Access Key ID [None]: AWIA....
AWS Secret Access Key [None]: CjWx....
```

01/07/24

Default region name [None]: us-east-1
Default output format [None]: json

Create Access key

To search - IAM, ~~set~~

IAM → Access management → Users

Users

User name

KK_labs_user ....

↳ click

[Create user]

KK_labs_user ....

Summary

.....

.... |Groups| ... |Security credentials|

↳ click

Console sign-in

02/07/24

(MFA)

Access key

Create access key

↳ click

Access key ID and Secret Access key will appear.

Don't run two pipelines ~~(what I~~ unless ~~there~~ is lock.

In the provided link executing 'pulumi up' command, go to the overview

No Updates

```
#change directories
#Select this stack
pulumi stack select amirul/aws-k3s-pulumi/dev
# Run the program to update the stack
pulumi up
```

01/07/24

If the credentials changes, again run 'aws configure' and then provide new values. After that, refresh pulumi state.

```
pulumi refresh
```

And then

```
pulumi up
```

Create a public subnet

```
public_subnet = aws.ec2.Subnet("public-subnet",
    vpc_id = vpc.id,
    cidr_block = "10.10.1.0/24"
    map_public_ip_on_launch = True,
    availability_zone = 'us-east-1a'
)
```

```
pulumi up
```

01/07/24

Create Internet Gateway

```
igw = aws.ec2.InternetGateway("igw", vpc_id=vpc.id)
```

```
pulumi up
```

Create Route Table

```
route_table = aws.ec2.RouteTable("route-table",
    vpc_id = vpc.id,
    routes = [{
        "cidr_block": "0.0.0.0/0",
        "gateway_id": igw.id,
    }])
```

Associate the newly created route table to the public subnet

```
route_table = aws.ec2.RouteTable("route-table",
    vpc_id = vpc.id,
    routes = [{
        "cidr_block": "0.0.0.0/0",
        "gateway_id": igw.id,
    }])
```

02/07/24

To ~~create~~ login, an aws ec2, I need a key.
In the host machine

```
sudo ssh-keygen -t rsa -b 4096 -C "aws_k3s_pulumi"
```

-t ; type, -b; bits, -C ; comment

Copy whatever within the 'id_rsa.pub' file.

```
sudo cat /root/.ssh/id_rsa.pub
```

Associate the route table with the previously created subnet

```
rt_assoc_public = aws.ec2.RouteTableAssociation(
    ("rt-assoc-public",
     subnet_id = public_subnet.id,
     route_table_id = route_table.id,
    )
```

```
security_group = aws.ec2.SecurityGroup("web-secgrp",
    description = 'Enable SSH and k3s access',
    vpc_id = vpc.id
```

02/07/24

```
ingress = [
  {
    "protocol": "tcp",
    "from_port": 22,
    "to_port": 22,
    "cidr_blocks": ["0.0.0.0/0"]
  },
  {
    "protocol": "tcp",
    "from_port": 6443,
    "to_port": 6443,
    "cidr_blocks": ["0.0.0.0/0"],
  }
]

egress = [{
    "protocol": "-1",
    "from_port": 0
    "to_port": 0,
    "cidr_blocks": ["0.0.0.0/0"],
}]
```

02/07/24

protocol: -1 ; for all protocols
from_port: 0 ; for all ports

"from_port" and "to_port" combined means a range of ports. For the particular ~~port~~ protocol the range of ports ~~was~~ is to be opened.

```
"from_port" : 6443,
"to_port" : 6443,
```

The above code snippet means only one port will be allowed, so both of the values are same.

```
ami_id = "ami-. . . . "
instance_type = "t3.small"
keypair = aws.ec2.Keypair("k3s-keypair",
    key_name = "k3s-keypair",
    public_key = "The public key generated in the host"
    )
master_node = aws.ec2.Instance("master-node",
    instance_type = instance_type,
    ami = ami_id,
```

(capital)

04/04/24

```
subnet_id = public_subnet.id,
key_name = keypair.key_name,
vpc_security_group_ids = [security_group.id],
tags = {
    "Name": "master-node"
}
}
```

pulumi up

Export public ip of the master node

```
pulumi.export("master_public_ip", master_node.public_ip)
```

pulumi up

In the terminal

```
Outputs:
  + master_public_ip: "34.203.210.200"
```

As I have generated the keypair in my host and the use that public key for creating a keypair for ec2, so the private key is located in the host.

```
sudo chown $USER:root /root/.ssh/id_rsa
```

```
sudo chmod 400 /root/.ssh/id_rsa
```

```
sudo ssh -i /root/.ssh/id_rsa ubuntu@<master ip address>
```

Similarly create worker nodes. And follow the rest of the poridhi AWS K3S Deployment lab.

—o—

Scheduler Part# 2

Labels and Selectors

Labels can be added as much as wanted. Kubernetes objects use labels and selectors internally to connect different objects together. For example, to create a replicaset consisting of three different pods, the labels defined under the template section are the labels configured on the pod. The labels at the beginning of 'metadata' are the labels of the replicaset itself.

The labels on the replicaset will be used if it is configured to discover some

02/09/24

other object to discover the replicaset.

In order to connect, the replicaset to connect to the pod, configure the selector field under the replicaset specification to match the labels defined on the pod. A single label will do if it matches correctly. If there is other parts with the same label, but with a different function, both labels could be specified to ensure that the right pods are discovered by the replicaset. On creation, if the labels match, the replicaset is created successfully. It works the same for other objects like a service.

When a service is created, it uses the selector defined in the service definition file to match the labels set on the pods in the replicaset definition file.

Taints and Tolerations

Taints and Tolerations are used to set restrictions on what pods can be scheduled on a node. For example there are three nodes Node 1, Node 2 and Node 3. Pod A, B, C and D will be deployed to the nodes.

02/07/24

Lets assume that there is dedicated resources on node1 for a particular use case or application. So only those pods that belong to this app to be placed on node1.

First prevent all pods from being placed on the node by placing a taint on the node. Lets call it blue. By default, pods have no toleration which means unless specified otherwise none of the pods can tolerate any taint. So in this case, none of the pods can be placed on node1 as none of them can tolerate the taint blue. This solves half of the requirements, no unwanted pods are going to be placed on this node.

The other half is to enable certain ~~parts~~ pods to be placed on this node. For this it must be specified which parts are tolerant to this particular attempt. In my case, I would like to allow only podD to be placed on this node, so I add a toleration to podD. Pod D is

now tolerant to blue. So when the scheduler tries to place this pod on Node 1, it goes through node 1, can now only accept pods that can tolerate the taint blue.

So with all the taints and tolerations in place, this is how the pods would be scheduled. The scheduler tries to place pod A on node 1 but due to the taint, it is thrown off and it goes to other node 2. This is true for every other pods except pod D.

Remember taints are set on nodes and tolerations are set on pods.

To taint a node

`kubectl taint nodes 'node-name' key=value:taint-effect`

There are 3 taint effects - NoSchedule, PreferNoSchedule and NoExecute.

NoSchedule: The pods not be scheduled on the node.

PreferNoSchedule: Which means the system

02/07/24

will try to avoid placing a pod on the node but that is not gura guaranteed.

NoExecute: Which means that new pods will not be scheduled on the node and existing pods on the node if any will be evicted if they don't tolerate the taint.

These pods may have been scheduled on node before the taint was applied to the node.

Example command

```
kubectl taint nodes node1 app=blue:NoSchedule
```

To add toleration in a pod definition file, add a 'tolerations' section under 'spec'.

```
...
spec:
  ---------
  tolerations:
  - key: "app"
    operator: "Equal"
    value: "blue"
    effect: "NoSchedule"
```

02/07/24

When a cluster is set up, a taint is automatically set on master node that prevents any pods from being scheduled on ~~this~~ master node. ~~The taint~~ It can be modified but the best practice is not to set application workloads on a master node.

—o—

03/07/24

Node Selectors

First label the node before creating the pod.

```
kubectl label nodes <node name> <label-key>=<label-value>
```

```
kubectl label nodes node-1 size=Large
```

In the pod definition file under 'spec'

```
spec:
  . . . . .
  nodeSelector:
    size: Large
```

Node selectors has disadvantages. If it needs more scheduling logic, it can't handle those.

03/07/24

Node Affinity

The primary feature of node affinity is to ensure that pods are hosted on particular nodes.

~~Ans~~ - In the 'spec' section of the pod-definition file

```
spec:
  ..... - - - - - - - - - - - - - - -
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: size
            operator: In
            values:
            - Large
```

The 'In' operator ensure that the pod will be placed on a node whose label size has any value in the list of values specified here.

03/07/24

In this case, it is just one called ~~Range~~. 'Large'. ~~Values can be multiple~~ Under 'values' there can be multiple elements. (e.g. medium, small).

The 'operator' can have value 'NotIn'. The 'Exists' is as same as the 'In'.

What if node affinity could not match a node with a given expression?

The answer is the type of nodeAffinity. In my case, the type is - 'requiredDuringSchedulingIgnoredDuring, Execution'

There are currently two types of node affinity available - required during scheduling, ignored during execution. and preferred during scheduling ignored during execution.

The two new types planned to ~~come~~ introduce in the future -

required During Scheduling, Required During Execution

preferred During Scheduling, Required During Execution

03/07/24

Resource Limits

In pod, there is a resource requirement. In the definition file it can be mentioned like the following under 'spec' -

```
resources:
  requests:
    memory: "4Gi"
    cpu: 2
```

The cpu count can be 0.1. The '1' count of cpu is equal to 1 vCPU. Memory can be ~~mem~~ in Mi (Mebibytes). The container will not consume more than mentioned resources.

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
```

03/07/24

```
(same level as 'metadata')

spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080

    resources:
      requests:
        memory: "1Gi"
        cpu: 1
      limits:
        memory: "2Gi"
        cpu: 2
```

What happens when a pod tries to exceed the specified limit?

In case of the cpu, the system throttles the cpu so that it does not go beyond the specified limit. A container can't use more cpu resources than its limit. However, this is not

The case with the memory. A container can use more memory resources than its limit. So if a pod tries to consume more memory than its limit constantly, the pod will be terminated. In the log file, the OOM error in the logs will be shown.

By default, Kubernetes doesn't have memory or cpu limit is set. So, this means, that any pod can consume as much resources as required.

Let's say, there are two pods competing for cpu resources on the cluster. Without the limit set, one pod can consume all the cpu resources on the node and prevent the second pod of required resources.

~~If the limits and requests are set as same.~~

If No requests is set then requests, limit is set then requests and limits are the same.

If the requests and limits are set, the pods will use resources as they request. But if more

03/07/24

than requested resources are needed, the pod can't use the extra required resources.

If requests and no limits are set, then the pod can use the lowest amount of resources. However because limits are not set when available, any pod can consume as many cpu cycles as available. But, in any point in time, if pod 2 requires additional cpu ~~cy~~ cycles or whatever it ~~is~~ has requested, then it will be guaranteed its requested cpu cycle.

For Memory Behavior

No Requests + No limits → One pod will consume all memory, other one will starving

No Requests + Limits : can't go beyond the limit

Requests + Limits : ~~If other pod cant use the resource less than it~~ can't go beyond limit though the other pod is ~~not~~ using less than limit.

Requests + No limits : If ~~the other~~ pod is ~~not~~

03/07/24

using less than limit, the pod ~~is~~ can go beyond the limit.

Once memory is assigned, only way to retrieve it is to kill the pod and free up all the memory that are used by it.

How to ensure that every pod created has some defaults set?

This is possible with limit ranges.

limit-range-cpu.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default:
      cpu: 500m
    defaultRequest:
      cpu: 500m
    max
      cpu: "1"
```

same level as 'max'

```
    min:
      cpu: 100m
    type: container
```

03/07/24

This applicable in the namespace level.

A resource quota is a namespace level object that can be created to set hard limits for requests and limits.

resource-quota.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
spec:
  hard:
    requests.cpu: 4
    requests.memory: 4Gi
    limits.cpu: 10
    limits.memory: 10Gi
```

03/07/24

Daemon Sets

Daemon sets are like replica sets, as it helps to deploy multiple instances of pod. But it runs one copy of the pod on each node in the cluster. Whenever a new node is added to the cluster, a replica of the pod is automatically added to that node. When a node is removed, the pod is automatically removed.

The daemon set ensures that one copy of the pod is always present in all nodes in the cluster.

For a use case like log viewer or for monitoring, daemon set is a good solution as it can deploy the monitoring agent in the form of a pod in all the nodes in the cluster. So no need to think adding or removing agent from these nodes when there are changes in the cluster.

Another solution is networking. Networking solutions

03/07/24

like weave net requires an agent to be deployed on each node in the cluster.

daemon-set-definition.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon

spec:
  selector:
    matchLabels:
      app: monitoring-agent

  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
      - name: montoring-agent
        image: monitoring-agent
```

kubectl get daemonsets

kubectl describe daemonsets

03/07/24

Static Pods

If the node is not a part of a cluster, the kubelet can be configured to read the pod definition files from a directory on the server designated to store information about pods. (/etc/kubernetes/manifests). The pods definition files in this directory, the kubelet periodically checks this directory for files, reads these files and creates pods on the host.

Not only it creates the pod, it can ensure that the pod stays alive. If the app crashes, the kubelet attempts to restart it. If any change has been made to any of the file within this directory, the kubelet recreates the pod for those changes to take effect.

If a file is removed from this directory, the pod is deleted automatically. So these pods that are created by the kubelet on its own without the intervention from the API server

03/07/24

on rest of the Kubernetes cluster components are known as Static pods. Only pods can be created in this way, not anyother object (replicaset, service etc.). The kubelet works only on pod level and only understands pod, that's why kubelet is able to create pod in this way.

The directory can be any directory.

—o—

Multiple Scheduler

Other than the kube-scheduler, kubernetes can have custom scheduler. The custom scheduler has to have different name. The default scheduler config file is 'scheduler-config.yaml'. Custom scheduler configuration file is as follows

my-scheduler-config.yaml

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
- schedulerName: my-scheduler
leaderElection:
  leaderElect: true
```

(same level as 'leaderElect'
resourceNamespace: kube-system
resourceName: lock-object-my-scheduler

03/07/24

The additional scheduler can be deployed using 'kube-scheduler' binary or from custom binary.

Scheduler as a Pod

my-custom-scheduler.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-custom-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --config=/etc/kubernetes/my-scheduler-config.yaml
    image: k8s.gcr.io/kube-scheduler-amd64:v1.11.3
    name: kube-scheduler
```

kubectl logs my-custom-scheduler -n=kube-system