KUBERNETES

5 Day Training

Abstract

Kubernetes is an open-source container-orchestration system for automating application deployment, scaling, and management. It was originally designed by Google, and is now maintained by the Cloud Native Computing Foundation.

S, Robin

robin.s@hpe.com

Table of Contents

List of Figures

Orchestration with Kubernetes

General Notes:
- Orchestration → Managing a large number of application across large number of machines.
- Cluster: A bunch of machines working for same application.
- Each machine is called node.

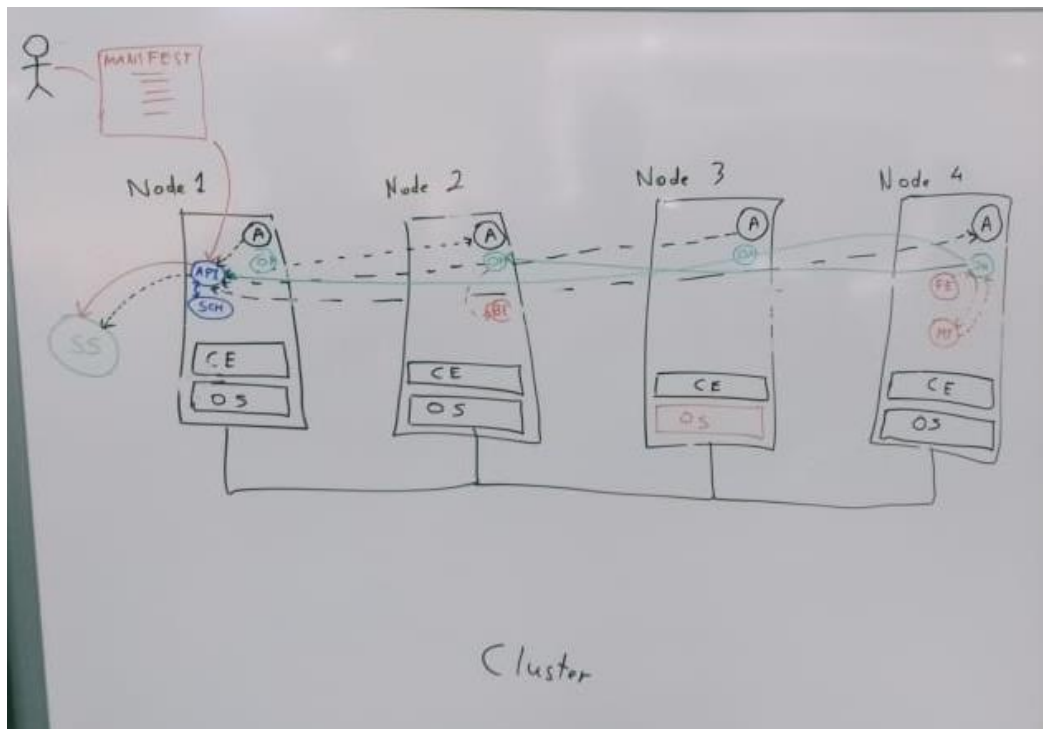*Figure 1: Orchestration Architecture*

- Node contains:
  - OS
  - Container Engine
  - Orchestrator (Agent, API, Scheduler & Shared Storage)
  - Agent
  - API
  - Shared Storage
  - Scheduler
  - Overlay Netwok
- Agent is a software which collects the information of nodes and reports it to API.
- An application definition is known as manifest file.
- Application Definition is stored in Shared Storage.

- Shared Storage is present where API is present
- A replica of shared storage is copied to each node, so that API can be moved to any node.
- API runs on one or more nodes. The info received by API stores on Shared Storage.
- Scheduler makes the decision by contacting Shared Storage directly or via API.
- After Scheduler makes the decision, it writes the result to Shared Storage.
- Agent also asks API whether it has any job for it to perform in that node.
- If an application crashes in a node, then agent restarts the app and reports to API.
- When a node dies, it doesn't report it API and Scheduler decides that particular node is dead.
- If none of the node has the specified requirement of app, that app remains unscheduled.
- Overlay Network, which sits on top of physical network, allows the virtual networking between nodes, so that applications can talk to eachother.
- Overlay network is a component of every node which assigns IP addresses to each container.
- Overlay network from node can talk to overlay network of other node and redirects the request to particular application containers.
- So orchestrator service always tries to maintain desired state (Defined in Manifest file) and current state.
- API + Scheduler is known as control plane.
- The node which runs Control Node is known as Master node.
- Split brain is when two manager gives different decisions.
- To eliminate the split brain condition, we should have odd number manager node.

Kubernetes

- In Kubernetes:
    1. Agent→kubelet
    2. Shared Storage→etcd
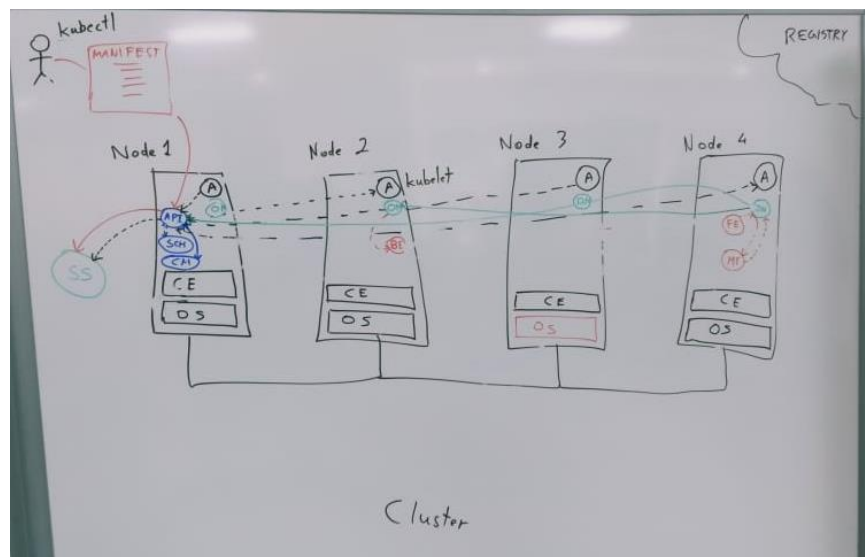   3. Overlay Networking →weave, calico or flannel

- **Kubelet** : It is an agent which uses CRI (Container Runtime Interface) to talk to Container engine (Docker, Rkt, CRI-O) .
- In kubernetes, API uses Shared storage such as etcd, sqlite, zookeeper.
- Overlay Networking software used can be weave, **Calico** (mostly used), flannel via CNI.
- Desired state and Current state are represented as **collection/graph of object**s in Kubernetes.
- The kubernetes objects are:
  1. Node
  2. Pod
  3. Replica Set (Workload Object)
- **Pods :** Is a group of one or more containers with shared storage/network, and a specification for how to run the containers.
- Pods run on nodes and run together as a logical unit.
- API's main functionality is CRUD operations.
- Controller is a software watches the kinds of objects in shared storage
- Controller Manager is higher level software which manages the different controllers.
- Replica set is a high level object which defines the pod object and creates specifies replicas.
- Pod Replicas are created and taken care by Replication Controller and individual pods are taken care by Scheduler.
- It is better to create Workload objects (Replica set) rather than creating pod object.
- Kubernetes API is a REST API. Therefore some sort of security should be maintained.
- Security is of 2 types: SSL and Authorization(Client clide certificate or token)
  1. To access the Kubernetes cluster, we should have SSL certificates (https).
  2. Every Agent should have client side certificate certified by Certificate Authority same as that of Kubernetes cluster.
- Bearer token is be used by agent for accessing the API.
- Agent uses the join token, first time it talks to API. After Node joins the cluster, agent gets its own SSL certificates or token.
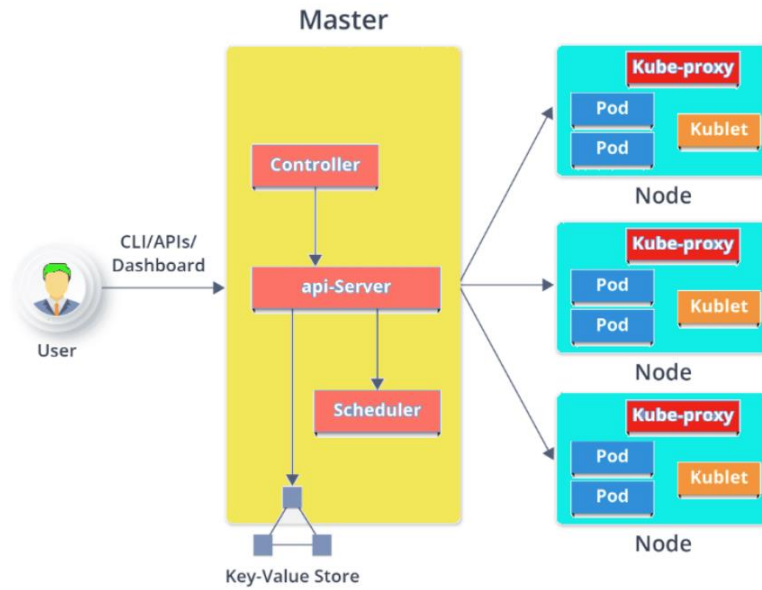
Figure 3: Kubernetes Architecture

- ➢ **Master Node:** Entry Point for All Administrative Tasks.
    - o Multiple Master Can be possible.
    - o In **Multi Master Node System**, Single Master node will be commanding Node for own workers and other Masters too.
    - o Main Master Node uses **etcd** to manage the Workers and Other Master Nodes.
- ➢ **API Server:** API server is the entry point for all the REST commands used to control the cluster.
    - o Resulting state of the cluster is stored in the **distributed key-value store**.
- ➢ **Controller**: Regulates the Kubernetes cluster which manages the different non-terminating control loops.
    - o Performs lifecycle functions such as namespace creation, event garbage collection, node garbage collection, etc.
    - o Controller watches the desired state of the objects it manages and watches their current state through the API server.
    - o If the current state of the objects it manages does not meet the desired state, then the control loop takes corrective steps to make sure that the current state is the same as the desired state.
- ➢ **Scheduler**: Regulates the tasks on slave nodes. It stores the resource usage information for each slave node.
    - o Schedules the work in the form of Pods and Services.
- ➢ **ETCD**: Distributed key-value store which stores the cluster state.
    - o Can be part of the Kubernetes Master, or, it can be configured externally.
    - o It's mainly used for shared configuration and service discovery.
- ➢ **Kubelet**: It is an agent which communicates with the Master node and executes on nodes or the worker nodes.
    - o Kubelet gets the configuration of a Pod from the API server and ensures that the described containers are up and running.

➢ **Kube-Proxy:** Kube-proxy runs on each node to deal with individual host sub-netting and ensure that the services are available to external parties.
  - o   Kube-proxy acts as a network proxy and a load balancer for a service on a single worker node.
  - o   It is the network proxy which runs on each worker node and listens to the API server for each service endpoint creation/deletion.

- There are few softwares wich are used only to install kubernetes:
  1. KOPS → installs kubernetes in Amazon Cloud
  2. Kube-spray → software Incubators
  3. minikube → which creates a single node kubernetes cluster with all the requirements.
  4. kubeadm → it installs the kubernetes cluster with creation of certificate, api, scheduler.

Pre-installed software in OVF

- kubelet (Agent)
- kubeadm
- kubectl → Its Cmd line interface to interact with kubernetes

VM Setup

- In virtual Box, File→ preferences →Network → rename it to kubenet
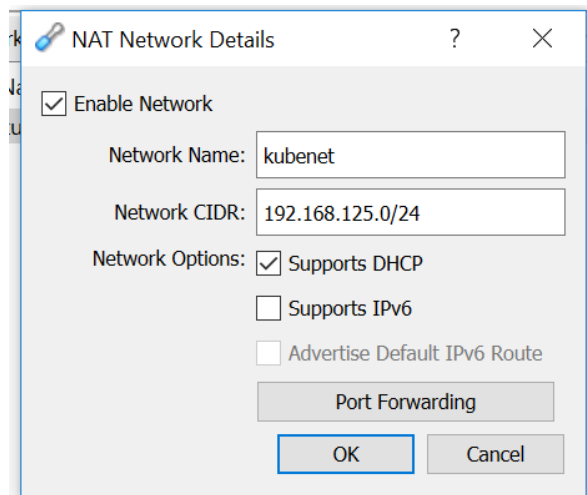
- import OVF → primary group =/kube
- MAC address policy→generate new MAC addresses and import.
- right click → setting → network → NAT Network → select kubenet

- Repeat same steps to create second VM
- detachable start Krishna VM
- VM Credentials:
  - user: user1
  - password: Pass@word1
- ip addr　　　　//shows IP addresses
- cd rw-installscripts

In second VM

- sudo ./set-hostname.sh radha　//changes the hostname to radha
- Kubernetes needs swap memory to be disabled.
- In VM File→preferences →network→ Port forwarding

	IPv4	IPv6							

Name	Protocol	Host IP	Host Port	Guest IP	Guest Port
Krishna ssh	TCP		10022	192.168.125.4	22
Radha app port	TCP		30010	192.168.125.4	30010
Radha ssh	TCP		20022	192.168.125.5	22

To setup proxy and connect to wifi
- we need two level of setup.
  - proxy.sh
  - Daemon service
- sudo vi /etc/profile.d/proxy.sh
  - export http_proxy=http://web-proxy.in.hpecorp.net:8080
  - export https_proxy=http://web-proxy.in.hpecorp.net:8080
  - export no_proxy=localhost,127.0.0.1,192.168.125.4,192.168.125.5,192.168.125.6
- logout & login the VM
- sudo vi proxy.conf
  - [Service]
  - Environment= ="HTTP_PROXY=http://web-proxy.in.hpecorp.net:8080"
- sudo mkdir -p /etc/systemd/system/docker.service.d
- sudo cp proxy.conf /etc/systemd/system/docker.service.d/
- sudo systemctl daemon-reload
- sudo service docker restart

Kubernetes Setup
In Krishna VM:

- kubeadm config images pull　　// Pulls required images for kubernetes
- init process has preflight phase
- sudo kubeadm phase preflight　// To run the preflight check

- sudo kubeadm init --apiserver-cert-extra-sans localhost,127.0.0.1   //request can be made to kubernetes cluster with specified hostname & also performs all the phases for init
- Above command makes the node master and kubeadm also gives join token
- kubeadm join 192.168.125.4:6443 --token a4m36x.5t5vlzkpm8txpg3m \   --discovery-token-ca-cert-hash sha256:b38a4ddeb6d1a9bd4648b6c9c9580181d9a 481d0c84f0b39f4e94450742a5303
- kubeconfig is a file which defines information about the kubernetes cluster. It is yaml format
- mkdir .kube → cd .kube
- sudo cp /etc/kubernetes/admin.conf config   //copy file to config
- sudo chown user:user1 config   //Change ownership of kubernetes config file

Working with Kubectl

- Kubectl commands are used to modify Kubernetes objects.
- These commands are internally changed as REST requests to API server.
- kubectl api-resources   //shows list of kinds of objects
- Namespace is a way of grouping together.
- kubectl get <kind>//This command is used to query the shared storage about objects
- kubectl get nodes   //lists all the nodes present
- kubectl get namespaces   //shows the namespaces
- kubectl get pods --namespace <namespace>   //shows pods in that namespace
- kubectl get pods -o wide   //-o is used to change the output format
- kubectl get all --namespace kube-system -o wide   //shows all the objects necessary for running an application.
- daemonset object creates as many pods there are as Nodes. one pod per node.
- Deployment object creates pods evenly among the nodes as far as possible.
- Deployment object internally creates Replica set which in turn creates pods.
- Deployment object is top level object.
- kubectl get <kind> <kindname>   //shows info about only particular object
- kubectl get deployment coredns --namespace kube-system -o yaml   //shows output in yaml
- kubectl get pods -l "day=two"   //dipays pods matching that label

YAML file
- It defines all the object with its properties.
- Properties are in Key:value pairs.
- Values are of 3 types:
  - scalar →single value
  - list
  - map values (object which has its own key and value)
- A yaml key can have a single value which is scalar values.
- While writing a scalar value there should atleast one space after colon[:].

- Scalar value has a datatype. if it is a string enclose within quotes.
- A key can have multiple values known as list/array.

> key:
>
>   - value1
>
>   - value2
>
>   - value3   //two spaces – one space value
>
>   - subkey1: value1

- subkey should be two spaced from its parent key

In deployment yaml file:

- All objects must have atleast 3 things : apiVersion, kind and metadata.
- apiVersion: <apigroup>/<version>
- kind: <object kind>
- metadata:
  - name: <defines name of object>   //name is a must property
  - namespace: <name of namespace>   //deploys in that particular namespace
  - labels:
    - <label-name>: <label value>   //arbitrary values/labels given to objects. easy to query group of objects having same labels. They are indexed.
  - annotations:
    - <key>:<value>   //similar to labels but they are not indexed

Other properties in objects (yaml)

- spec: //describes what happens when this object is created

- kubectl describe <kind> <object-name>   //describes the properties of particular object
- Join Radha to cluster by pasting join token in radha
- Overlay networking has to be created for nodes to be in Ready state.
- we use "Weave" overlay network software

To install Overlay network
  - mkdir overlay
  - wget https://cloud.weave.works/k8s/v1.17/net.yaml
  - kubectl apply -f net.yaml   //creates the objects specified in yaml file
- After above commands, nodes are in ready state and weave pods are created.

Installation of Kubernetes is done by above commands.

Creating our own Objects

- kubectl run → creates executable objects
- <mark>kubectl run --image=nginx:alpine</mark>

  <mark>--generator= <kind>   //specifies the kind of objects</mark>

  <mark><podname></mark>

- <mark>kubectl run --image=nginx:alpine --generator run-pod/v1 pod0</mark>   //creates a pod pod0 with container image nginx:alpine
- <mark>kubectl describe pods pod0</mark>   //describes the pod0 → see the events
- Whenever we create a pod, Doker engine creates a secret container from '\pause' to share the IP with side car container.
- If containers are deleted, it comes up again.
- If node in which pods are running goes down, we may lose the pods.

Let's create *Deployment* workload object
- Create dep1.yaml

```
user1@krishna: ~/day2/sample0
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep1
  labels:
    day: two
    sample: sample0
spec:
  replicas: 1
  selector:
    matchLabels:
      day: two
      sample: sample0
      layer: frontend
  template:
    metadata:
      labels:
        day: two
        sample: sample0
        layer: frontend
    spec:
      containers:
        - name: c1
          image: nginx:alpine
~
```

- kubectl apply -f dep1.yaml --validate=true --dry-run=true   //validate checks the syntactical errors in yaml and dry-run checks the k8s level errors →creates the pod
- kubectl get all   //shows all the objects created with dep1.yaml
- To update the deployment:
- kubectl edit <kind> <name>   //allows us change the requirements
- kubectl describe nodes Krishna
  - observe for taints
  - if the 'taint' is assigned, no pod gets scheduled to that node
  - **toleration** can be used to schedule pods on node even if it has taint.
- kubectl taint   // command is used to customize taint on nodes
- kubectl taint node krishna node-role.kubernetes.io/master-   //removes the taint from that node
- kubectl delete -f dep1.yaml   //deletes all the objects created by that yaml file

user1@krishna: ~/day2/sample1

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep1
  labels:
    day: two
    sample: sample1
spec:
  replicas: 1
  selector:
    matchLabels:
      day: two
      sample: sample1
      layer: frontend
  template:
    metadata:
      labels:
        day: two
        sample: sample1
        layer: frontend
    spec:
      containers:
        - name: c1
          image: nginx:alpine
        - name: c2
          image: chitradoc/ics
~
```

Figure 5: Deployment with two container

- kubectl logs   //command is used to get logs of pods. it can be used only for pod object
- kubectl logs dep1-794b9cf5db-7rssz c1   //Gives logs of container inside pod, we have to specify the container, here it is c1
- kubectl exec -ti dep1-794b9cf5db-7rssz -c c1 /bin/sh   //attaches inside the container of that pod.

Service Object

- In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a selector
- When service is created, it creates a distinct IP address (different as that of pod's).
- Any request made to service forwards it to pod's IP addresses.
- A service also has ports.
- Kube-proxy is the one which receives the requests and forwards it to service.
- create svc1.yaml file

user1@krishna: ~/day2/sample1

```
apiVersion: v1
kind: Service
metadata:
  name: svc1
  labels:
    day: two
    sample: sample1
spec:
  type: ClusterIP
  selector:
    day: two
    sample: sample1
    layer: frontend
  ports:
    - name: nginx
      port: 80
    - name: ics
      port: 81
      targetPort: 8080
```

Figure 6: Service.yaml file

- kubectl apply -f svc1.yaml
- kubectl get svc   //shows all the services created
- kubectl describe svc svc1   //describes the service svc1
- kubectl exec -ti dep2-f54585587-bkpbx /bin/sh
  - inside container→ wget -O- http://svc1   // We will be able to connect to dep1 pods
- sessionAffinity is a property of spec in services.
  - sessionAffinity==stateful application (makes use of memory)
  - sessionAffinity = ClientIP → forwards the requests to one of the pods and subsequent reuqests to same pods
- Any communication between pods (ClusterIP) have to be made by services
- Type= ClusterIP → Used to communicate only among the pods
- Type=NodePort →used to communiocate pods from external node

13

Figure 7: Service with Type: NodePort

- For NodePort type, we have nodePort property which allows us to hardcode the port number.
- wget http://192.168.125.4:30010   //forwards requests to pod ports
- Service Type=LoadBalancer → it gets an external IP address
- Cloud Provider is the interface software that provides the externalIP between Kubernetes and other side client.
- We use LoadBalancer when we don't want to know the multiple IP:ports of nodes of several application.
- LoadBalancer will provide ExternalIP to use instead of node IP with ports.

Storage Volume

- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#volume-v1-core
- Driver is a software that takes care of Volume created.
- vi dep3.yaml

Figure 8: Deployment with volume

- <mark>kubectl apply -f dep3.yaml</mark>   //pod is schedule but shows warning
- change the type to DirectoryOrCreate and apply
- If replicas are more than one, pods created on same node can access the physical path.
- Other pods will have different storage volume mounted to physical path in its respective node.
- Therefore if multiple pods have to access to same mount, hostPath shouldn'y be used.

Persistent Volume Claim

- It is a new object which simply claims the volume.
- PersistentVolumeClaimVolumeSource references the user's PVC in the same namespace. This volume finds the bound PV and mounts that volume for the pod.
- A PersistentVolumeClaimVolumeSource is, essentially, a wrapper around another type of volume that is owned by someone else (the system).
- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#persistentvolumeclaimvolumesource-v1-core

Exercise
- In a clean cluster, do the following

- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#persistentvolumeclaimspec-v1-core
- in root mkdir volume
- vi dep3.yaml

user1@krishna: ~/day3/sample2

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep3
  labels:
    day: three
    sample: sample2
spec:
  replicas: 1
  selector:
    matchLabels:
      day: three
      sample: sample2
      layer: backend
  template:
    metadata:
      labels:
        day: three
        sample: sample2
        layer: backend
    spec:
      volumes:
        - name: vol1
          persistentVolumeClaim:
            claimName: pvc1
      containers:
        - name: c1
          image: nginx:alpine
          volumeMounts:
            - name: vol1
              mountPath: /data
```

Figure 9:Dep3 for persistent volume

- vi pvc1.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc1
  labels:
    day: three
    sample: sample2
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
~
~
```

Figure 10: PVC file

- access Modes→ Tells us how to access the persistent volume
  - ReadWriteOnce→assign volume only to us with read and write

- vi pv1.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
  labels:
    day: three
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage:  2Gi
  local:
    path: /storage/day3/pv1
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
              - radha
```

Figure 11: Persistent volume File

- Kubectl apply all above files.
- if we delete an object it exists until its dependents gets deleted → delete pv, it does not gets deleted until pvc is deleted. pvc is deleted only after deleting pods.

Dynamic Volume provisioner

- it dynamically provisions the physical storage and creates PV's with the help of PVC.
- **StorageClassName** is another property in spec of PVC. When mentioned, it calls the Dynamic Volume provisioner which in turn creates PV's.
- To install DVP
  - curl -L http://tiny.cc/rwprovisioner > provisioner.yaml
  - kubectl apply -f provisioner.yaml
- edit pvc1.yaml
  - add storageClassName: kutti-sc
- kubectl apply -f pvc1.yaml
- kubectl get pv   //shows the created PVs by provisioner
- kubectl describe pv <pv-name>   //shows details of PV like on which node it is created, reclaim policy etc.
- Here reclaim policy is 'Delete' so deleting PVC will delete PV and physical storage also.

Real-Time Application

- Frontend → adminer
- Backend → postgres
- Objects Required:
  - deployment (frontend and backend)
  - service (frontend and backend)
  - pvc
- mkdir dbapp → cd dbapp
- vi db-backend-dep.yaml

Figure 12: db-backend-dep.yaml

- vi db-backend-svc

Figure 13: db-backend-svc.yaml

- vi db-pvc.yaml

Figure 14: db-pvc.yaml

- vi db-frontend-dep.yaml

Figure 15: db-frontend-dep.yaml

- vi db-frontend-svc.yaml

Figure 16: db-frontend-svc.yaml

- Now all the required deployments and services are created.
- kubectl apply -f .   //applies all the files present in the folder.
- Go to the browser→http://localhost:30010
    - o it opens the adminer webpage

Network Policy Objects

- Network policy is used to tell which pod can communicate to other pods via services.
- If replicas of frontend is more than one and refresh adminer in browser, it gets redirected again to login page because service points to different pods.
- Therefore add sessionAffinity in frontend-svc file.

Figure 17: svc with ClientIP property

- Two types of Policy Types:
    - o Ingress: tells which pods can communicate to target pods.

- o Egress: tells which pods can be communicated by target pod.
- vi db-policy.yaml
- kubectl apply -f db-policy.yaml
- Now we will be able to ping from only pods which matches the labels mentioned in network policy.


```
user1@krishna: ~/day3/dbapp
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
  labels:
    day: four
    sample: dbapp
spec:
  policyTypes:
    - Ingress
  podSelector:
    matchLabels:
      day: three
      sample: dbapp
      layer: backend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              day: three
              sample: dbapp
              layer: frontend
```

Figure 18: db-policy.yaml

- kubectl get networkpolicies.networking.k8s.io   //displays the network policies

HELM Charts

- it is a set of manifest files grouped together into a single file.
- it is a template file which creates all the objects in a single shot.
- **Helm** uses a packaging format called **charts**. A **chart** is a collection of files that describe a related set of Kubernetes resources. A single **chart** might be used to deploy something simple, like a memcached pod, or something complex, like a full web app stack with HTTP servers, databases, caches, and so on.

HELM tool

- Tool used to package multiple manifest file and package into HELM chart.
- It also installs, apply and deploy the manifests from charts.
- To install HELM tool:
  - o in home directory, cd rw-installinstallscripts
  - o ./install-helm.sh

22

- helm has two parts: client side and server side
- client side to contact kubernetes requires ssl certificates.
- client side helm tool access same kube config file to communicate to kubernetes.
- in helm ver2, it uses Tiller software to deploy manifest files in kubernetes.
- to install Tiller software,
    - ./init-helm.sh
- To verify Tiller is installed
    - kubectl get pods --namespace kube-system -o wide   //shows the pod called tiller-

Working with HELM tool

- helm version   //shows the version of helm
- https://hub.helm.sh/
- helm install --name my-release stable/drupal
- helm install --name r2 --set global.storageClass=kutti-sc --set drupalUsername=admin --set drupalPassword=something --set service.type=NodePort --set service.nodePorts.http=30010 stable/drupal   //To change the parameters while installing
- helm install --debug --dry-run --name r3 --set global.storageClass=kutti-sc --set drupalUsername=admin --set drupalPassword=something --set service.type=NodePort --set service.nodePorts.http=30010 stable/drupal   //only dry-run

- Statefulsets is a workload object which creates pvc along with pods.
- when statefulsets is deleted, it does not delete pvc
- helm list   //shows the current objects
- helm list -a   //shows all the objects even if its deleted
- helm delete --purge <name>   //deletes entirely -- does not show in 'helm list -a'
- goto browser http://localhost:30010

Exercise:

- mkdir charts→ cd charts
- helm create dbapp   //creates a 'dbapp' chart
- cd dbapp
- vi charts.yaml

Figure 19: charts.yaml

- vi values.yaml   //it contains the all the sample parameters needed for the application
- there will be charts sub directory→which contains yaml files
- delete test file   //we do not require now
- in deployment.yaml, we see {{ }} → tis is known as 'go template'.
- here we use these templates
  - {{.Release.Name}}
  - {{.Chart,Name}}
  - {{.Values.}}
- {{ func value1 value2 }}
- {{ value | func }}
- {{- .Release.Name -}}   //'-' removes the space depending upon the position it is placed
- {{ include "abapp.name"}}   //if there is 'include' it is known as macro
- Go template is used to fetch values from the helm.

- Delete all the yaml files
- Copy all the yaml files created in previous project to templates folder.
- vi notes.txt and remove all   //instruction to do while deploying
- vi _helpers.tpl
- add this macro at bottom

Figure 20: _helper.tpl

- vi db-pvc.yaml

user1@krishna: ~/charts/dbapp/templates

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: {{ .Release.Name }}-db-pvc
  labels:
    day: three
    sample: dbapp
 {{- include "db.labels" . | nindent 4 }}
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: {{ .Values.storageClassName }}
~
```

Figure 21: db-pvc.yaml

- helm install --debug --dry-run --name test1 .   //run inside template folder →shows the yaml files
- helm install --set storageClassName=robin --debug --dry-run --name test1 .   //shows the yaml file with storageClassName is given name robin
- vi db-backend-dep.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-db-backend-dep
  labels:
    day: three
    sample: dbapp
    {{- include "db.labels" . | nindent 4 }}
spec:
  replicas: 1
  selector:
    matchLabels:
      day: three
      sample: dbapp
      layer: backend
    {{- include "db.labels" . | nindent 6}}
  template:
    metadata:
      labels:
        day: three
        sample: dbapp
        layer: backend
    {{- include "db.labels" . | nindent 8}}
    spec:
      volumes:
        - name: vol1
          persistentVolumeClaim:
            claimName: {{ .Release.Name }}-db-pvc
      containers:
        - name: c1
          image: postgres:alpine
          volumeMounts:
            - name: vol1
              mountPath: /var/lib/postgresql/data
          env:
            - name: POSTGRES_USER
              value: admin
            - name: POSTGRES_PASSWORD
              value: something
```

Figure 22: db-backend-dep.yaml

- helm install --name tr1 ..
- helm install --name tr2 ..   //creates two instances
- helm package ..   //creates a helm chart package

Secret Object
- It is an storage object where confidential parameters are stored that can be used by the objects.
- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#secret-v1-core
- open db-backend-dep.yaml and do the following changes in env section.

```
      env:
        - name: POSTGRES_USER
          value: admin
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: dbsecret
              key: dbpwd
```

Figure 23: secrets -dep file

- open db-frontend-dep.yaml and do the following changes.

```
    spec:
      volumes:
        - name: vol1
          secret:
            secretName: dbsecret
      containers:
        - name: c1
          image: adminer
          volumeMounts:
            - name: vol1
              mountPath: /my/secrets
          env:
            - name: ADMINER_DEFAULT_SERVER
              value: db-backend
```

Figure 24: secrets - Frontend-dep file

- Create a secret object.
- vi db-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: dbsecret
  labels:
    day: four
    sample: dbapp
stringData:
  dbpwd: something
  anotherpwd: another
  essay.txt: |
    Hello there, this is an essay.
    This is second line.
    This is third line.
  lastpass: last
```

Figure 25: db-secrets.yaml

- String object just stores the data.
- The data in secrets are encrypted to **base64** encryption.
- If we inspect the objects using secrets, secret data (ENV) will be displayed in encrypted format.
- But if we exec a container (deployment pod) and type 'env', it displays all the secrets in decrypted format.

Stateful set Object

- Clusterable (Stateful) applications such as Apache Kafka, MongoDB etc requires pods of different specification but same/distributed storage. They cannot use deployment object because all replicas of pods will be exact copy.
- For this problem, Kubernetes provide **Stateful set** object.
- https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
- StatefulSet is the workload API object used to manage stateful applications.
- Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods.
- Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.
- Stateful set object has template of pod →similar to deployment
- Even Stateful set has replicas property.
- But they create next pod only after complete creation of current pod.
- Stateful set has template for PVC. Each one for each pod.
- They can resolve podname to its IP address unlike deployment.
- They use Headless service uses a ClusterIP service (pod - pod communication) type but do not have IP address.

- podname/servicename resolves to particular pod.
- Statefulset does not give randomID after podname. It gives name such as 'statefulset-01'.
- When we reduce the replicas, earlier pods remain and latest pods will get deleted.
- Also it does not delete PVC even if we delete pods or reduce the replicas.
- PVC is never deleted until deleted manually even if the statefulset object is deleted.
- Use Statefulset only if the application has distributed storage and need different spec pods.
- All service has two names.
  - one is what we specify in manifest file
  - second is fully qualified domain name <svc>.<namespace>.svc.cluster.local

Exercise

- create a directory day5
- mkdir statefulset→cd statefulset
- curl -L http://tiny.cc/rw-sst-sample >sst.yaml
- kubectl apply -f sst.yaml   //objects are created sequentially
- kubectl get all -o wide
- The application used here is rabbitMQ
- RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, Message Queuing Telemetry Transport, and other protocols.

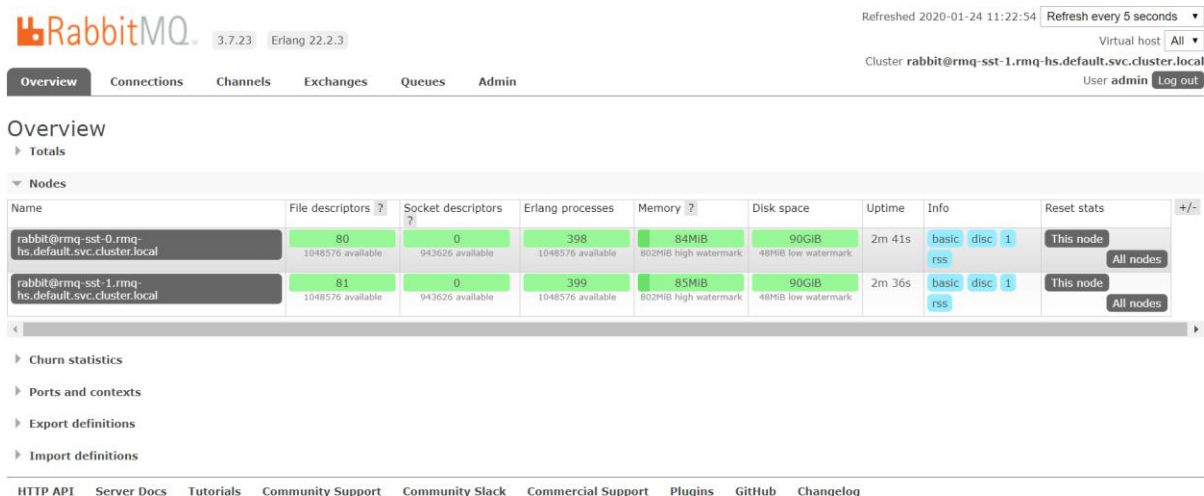*Figure 26: RabbitMQ Application*

Service Account Object
- Application running inside Kubernetes do not require client side certificate, so they use ServiceAccountObject to talk to Kubernetes API.
- A **service account** represents an identity for processes that run in a pod. When a process is authenticated through a **service account**, it can contact the API server and access cluster resources.

29

- It creates a token to authenticate instead of client side certificate.
- it allows operation on all objects except secret object when no permission is given.
- To give permission, cluster role is created by service account object.
- This role can be bound to Namespace or cluster.
- For example, dynamic volume provisioner has readonly permission to pvc file and write for PV objects
- Go to provisioner folder and vi provisioner.yaml
- ClusterRole object defines a bunch of rules
- Service Account object just represents the identity account
- clusterRoleBinding object just binds the ClusterRole object to Service Account object
- Whenever Service Account is created it also creates corresponding secret object.
- This secret object is volume mounted to all the containers inside the pod.
- This service object contains 3 thing:
    - Token
    - NameSpace name
    - certificate

- kubectl apply -f dep1.yaml
- kubectl describe <podname> →observe secret and mountpath
- kubectl exec -ti <podname> /bin/sh
- cd <mountpath>   //Here we'll be able to see the three secrets

Scheduling pods based on resources

- There is a property 'resouces' where we can specify the requirements such as memory limits
- kubectl describe pod <podname>   //Observe QoS name
- QoS can be BestEffort, Burstable Guaranteed
- BestEffort → If there is no resource specifies and pods gets scheduled in any node.
- Burstable → If resource is specified and scheduler schedules based on node resource availability
- Guranteed→ if requests and limit are both same i.e we know the exact requirement
- vi dep1.yaml
- do the changes in spec of containers

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep1
  labels:
    day: two
    sample: sample0
spec:
  replicas: 5
  selector:
    matchLabels:
      day: two
      sample: sample0
      layer: frontend
  template:
    metadata:
      labels:
        day: two
        sample: sample0
        layer: frontend
    spec:
      containers:
        - name: c1
          image: nginx:alpine
          resources:
            limits:
              memory: 800Mi
            requests:
              memory: 750Mi
```

Figure 27: Resources in Container

- in above yaml
  - limits specify the maximum resource required
  - requests specify the minimum resource required
- kubectl apply -f dep1.yaml
- kubectl get pods   //Notice there will be 2 pods per node and 5th pod is pending
- Because node RAM is 2Gb and each replica requires 750Mb. For each node 2 replicas will consume 1.5Gb. Therefore fifth pod won't have 750Mb free space and hence pending state.
- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#resourcerequirements-v1-core

Figure 28: Resource with cpu requirement

- in above yaml
    - cpu → specifies the cpu core

Scheduling pods based on Affinity and AntiAffinity

- There is a property in pod spec where we have different options for affinity
- https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/#affinity-v1-core
- kubectl apply -f db-backebd-dep.yaml
- vi db-frontend-dep.yaml

```
                                as backend
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchLabels:
                    day: three
                    sample: dbapp
                    layer: backend
                topologyKey: kubernetes.io/hostname
```

Figure 29: podAntiAffinity

- kubectl get pods   //all pods will be created on other node than backend pod is created.
- podAntiAffinity → tells pod to be scheduled in different node than specified pod is created.

Probes in Kubernetes

- A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the Container.
- Each probe has one of three results:
  - Success: The Container passed the diagnostic.
  - Failure: The Container failed the diagnostic.
  - Unknown: The diagnostic failed, so no action should be taken
- The kubelet can optionally perform and react to three kinds of probes on running Containers:
  - **livenessProbe**: Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is Success.
  - **readinessProbe**: Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is Failure. If a Container does not provide a readiness probe, the default state is Success.
  - **startupProbe**: Indicates whether the application within the Container is started. All other probes are disabled if a startup probe is provided, until it succeeds. If the startup probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a startup probe, the default state is Success.

- In pod spec, there is property called initContainer.
  - initContainer: List of initialization containers belonging to the pod. Init containers are executed in order prior to containers being started. If any init container fails, the pod is considered to have failed and is handled according to its restartPolicy.

33

- Agent will take care of the liveness and readiness probe.

Job Object

- A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (ie, Job) is complete. Deleting a Job will clean up the Pods it created.
- A simple case is to create one Job object in order to reliably run one Pod to completion. The Job object will start a new Pod if the first Pod fails or is deleted (for example due to a node hardware failure or a node reboot).

CronJob

- A *Cron Job* creates Jobs on a time-based schedule.
- One CronJob object is like one line of a *crontab* (cron table) file. It runs a job periodically on a given schedule, written in Cron format.

nginx-ingress

It is reverse-proxy service software which listens on a port forward requests to pods.

Similar to nginx-ingress is **traefik**

Monitoring

Software monitors the process the application and create metrics.

The different software used are

- logstash:
  - Logstash is an open source data collection engine with real-time pipelining capabilities. Logstash can dynamically unify data from disparate sources and normalize the data into destinations of your choice. Cleanse and democratize all your data for diverse advanced downstream analytics and visualization use cases.
- elastic search:
  - Elasticsearch is a search engine based on the Lucene library. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch is developed in Java.
- kibana:

- Kibana is an open source data visualization dashboard for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts and maps on top of large volumes of data.
- Beats: This software sits next to our application.
  - The open source platform for building shippers for log, network, infrastructure data, and more — and integrates with **Elasticsearch**, Logstash & **Kibana**.

HorizontalPodAutoScaler

- The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metrics). Note that Horizontal Pod Autoscaling does not apply to objects that can't be scaled, for example, DaemonSets.
- The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller.
- Core Kubernetes does not have any software to receive the metrics
- kubectl top pod <podname>   //shows error because no metric server
- kubectl apply -f http://tiny.cc/rw-metrics-server
- kubectl top podORNode <name>   //now shows the output

Exercise
- This works only after installing hpa.
- mkdir hpa →cd hpa
- curl -L http://tiny.cc/rw-hpa-sample >hpa.yaml
- kubectl apply -f hpa.yaml   //creates the objects
- observe horizontalAutoPodAutoScaler object
- watch kubectl get pods -o wide   //initially only one replica is present
- in Radha
  - vi client.sh
    - while [1]; do
    - curl http://192.168.125.5:31000
    - done
  - chmod +x client.sh
  - ./client.sh
- observe the Krishna VM → pods automatically scaled up

Contact Details:

Raj Chaudhury

9930217314

raj@rajware.net chitra@rajware.net