Cluster contains multiple nodes but there will be only one master node.

**kubectl version**: to get version of kubernetes

**POD**

**kubectl run <pod\_name> --image=<image\_name>** -> it will run a new pod with image.

**kubectl run nginx --image=nginx** -> it will run a new pod called nginx with image as nginx.

**kubectl get pods** -> to get all the pods

**kubectl get pods -o wide** -> to get more details for the pods

**kubectl describe pod <pod-name>** -> it will give us the detailed info about the specific pod

Kubernetes yaml file must contain these fields -> apiVersion, kind, metadata, spec

|  |  |
| --- | --- |
| **kind** | **version** |
| Pod | v1 |
| Service | v1 |
| ReplicationController(deprecated) | v1 |
| ReplicaSet | apps/v1 |
| Deployment | apps/v1 |

nginx-pod.yaml

apiVersion: v1

kind: Pod

metadata:

  name: nginx

  tier: frontend

spec:

  containers:

    - name: nginx

      image: nginx

**kubectl apply/create -f nginx-pod.yaml** -> to create a pod with this yaml file.

**kubectl describe pod nginx** -> to give more description of the nginx pod

**kubectl edit pod nginx** -> it will open a vi editor where we can change the pod definition file also this is a in memory pod definition file which is maintained by Kubernetes.

**kubectl delete pod nginx** -> to delete the nginx pod

**kubectl delete --all pods** -> delete all the pods.

**kubectl delete all --all -n {namespace} ->** delete all resources in namespace.

**kubectl run redis --image=redis --dry-run=client -o yaml > redis-pod.yaml** -> It will not create any pod rather it’s a imperative style of writing definition file where a pod definition file will created with the necessary fields.

**ReplicaSet**

ReplicaSet is a group of same pods where we can scale in(reduce) or scale out(increase) the number of the pods.

**kubectl create/apply -f <replicaset-definition.yaml>** -> it will create a replicaset from the definition file.

**kubectl get replicateset** -> to get all the replicaset in the default namespace

**kubectl describe replicaset** -> to get all the replicaset in the default namespace

**kubectl delete replicaset <replicaset-name>** -> to delete the replicaset

nginx-replicaset.yaml

apiVersion: apps/v1

kind: ReplicaSet

metadata:

  name: nginx-replicaset

  labels:

    type: frontend

spec:

  selector:

    matchLabels:

      name: nginx-pod

      type: frontend

  replicas: 1

  template:

    metadata:

      labels:

        name: nginx-pod

        type: frontend

    spec:

      containers:

        - name: nginx

          image: nginx

labels under template section and mathLabels under selector should be same. That is how replicate set identifies the pod and controls the number of the pods. If we try to delete any pod or anyhow any pod got crashed, then Kubernetes automatically brings another pod in.

Scale commands ->

**kubectl replace -f nginx-replicaset.yaml** -> replace the previous nginx-repicaset with the current replicas given in the definition file

**kubectl scale --replicas=6 -f nginx-replicaset.yaml** -> it will override the replicas given in the yaml

**kubectl edit replicaset nginx-replicaset** -> it will open an editor and show the current configuration of the replicaset then we can easily scale out.

**kubectl scale --replicas=6 replicaset nginx-replicaset** -> it will scale out an existing replicaset

**Deployment**

Kubernetes deployment create creates one deployment kind of object.

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

  labels:

    app: nginx-deployment

spec:

  replicas: 2

  selector:

    matchLabels:

      app: nginx-pod

  template:

    metadata:

      name: nginx-pod

      labels:

        app: nginx-pod

    spec:

      containers:

      - name: nginx

        image: nginx

**kubectl get deployments** -> It will fetch all the deployments.

**kubectl describe deployments <deployment-name>** -> It will give us the description of the specific deployment

**kubectl get all** -> To get the resources like pod, replicasets, services, deployments

**kubectl create deployment http-frontend --image=httpd:2.4.alpine** -> it will create one deployment named ad http-frontend and its image will be httpd:2.4.alpine and count of the pod will be 1.

**kubectl scale deployment --replicas=3 httpd-frontend** -> it will scale the deployment and change the number of the pods

**kubectl create -f nginx-deployment.yaml --record** -> It will create a deployment and also record all the changes of the deployment for rollout history.

If we want to revert to previous version, we must add this record flag

**Update and roll back ->**

**kubectl rollout status deployment/nginx-deployment** -> it will give the current rollout status of the deployment. When we change the replicas or the image at that time it will give the information.

**kubectl rollout history deployment/nginx-deployment** -> It will give us all the rollout history of the Kubernetes deployment rollout

For rollout Kubernetes has 2 types of strategy

1. Recreate strategy
2. Rolling strategy

Kubernetes deployment object creates another replicaset and then it will add the pods into the replicaset

**kubectl rollout undo deployment/nginx-deployment** -> It will revert the latest changes back to the previous version.

Changes to the existing deployment:

**kubectl set image deployment nginx-deployment nginx:old-image=nginx:new-image --record** -> It will change the image of the deployment of the nginx deployment and will record it

**kubectl edit deployment nginx-deployment --** ->it will open vi editor and open the current configuration of the nginx-deployment

**Service**

For accessing the pods from the outside of the container we need service.

There are three types of services.

1. NodePort
2. ClusterIP
3. LoadBalancer

NodePort can range from 30000 to 32767. The work of the NodePort is to listen to a particular port and forward it to another node.

#nginx-service.yaml

apiVersion: v1

kind: Service

metadata:

  name: nginx-service

spec:

  type: NodePort

  ports:

    - port: 80

      targetPort: 80 #where the pod will listen (for nginx its 80)

      nodePort: 30008 #In this port the service will be accesible

  selector:

    app: nginx-pod #to connect with specific pods via pod’s label

**Kubectl create -f nginx-deployment.yaml** -> for creating nginx-service with nginx-deployment.yaml

**kubectl get services** -> To get all the services

**kubectl describe service <service\_name>** -> To get details of the specific service

**minikube service nginx-service –url** -> It will print the service url

**ClusterIP**: Internal communication of pods. Service definition is almost same as the NodePort. Here type is **ClusterIP**. **TargetPort** is where the backend is exposed, and **Port** is where the service is exposed. Type ClusterIp is the default service type.

**LoadBalancer**: With the NodePort service we can make external facing application available on the port of the worker nodes.

Let’s say we have four cluster and one each server there are one frontend app deployed. With NodePort we can make external traffic forwarded to frontend pod but again for that we will have 4 urls for 4 services.

So, need to have a loadbalancer here. We can have an external VM where nginx is deployed and then it will loadbalance 4 urls but it will be a complicated thing to manage.

So we can use the inbuilt loadbalancer of different cloud platforms like Azure,GCP or AWS.

**kubectl describe service <service-name>** -> if there is no endpoint then the service is not attached to any pod.

**minikube service <service-name> --url** -> to get the url of the NodePort service.

Imperative style of creating service:

**kubectl expose deployment nginx-deployment --name=nginx-service --target-port=80 --port=80 --type=NodePort** -> It will create a service named as nginx-service of type NodePort with specific port and targetPort and it will match labels of the deployment of nginx-deplpyment and NodePort will be be assigned randomly in the range of 30000 to 32767

**kubectl expose deployment nginx-deployment --name=nginx-service --target-port=80 --port=80 --type=NodePort --dry-run=client -o yaml > nginx-service.yaml** -> It will do just the same as previous just that it will save all the configurations in the nginx-service.yaml

If we have a connection like this:

1. Voting-app -> frontend app for gathering the votes which will save the votes in redis
2. Result-app -> frontend app for showing the votes which will fetch votes from postgres
3. Redis -> save the votes from in a in memory store
4. Postgres -> save the votes in relational db
5. Worker-app -> It will constantly fetch the vote count from redis and constantly update the vote count in postgres

So, our setup will be like this:

1. Deploy the pods/deployment
2. Create the ClusterIP service for Redis and postgres
3. Create the NodePort service for Voting-app and Result-app

**NameSpace**

NameSpace is a way to segregate different resources like dev,qa,prod etc. Default namespace is default.

#dev-namespace.yaml

apiVersion: v1

kind: NameSpace

metadata:

  name: dev

**Kubectl create -f dev-namepace.yaml** -> to create a namespace

**kubectl get pods –namespace=dev** -> To get all the pod inside dev namespace

If we have to set the dev namespace permanently then we can keep it inside the **KubeConfig**

**Kubectl config set-context $( kubectl config current-context ) –namespace=dev**

To limit the resources using in a specific namespace we can use resource quota

**kubectl get ns** -> to get all the namspaces

**kubectl get ns –no-headers | wc -l** -> To get count of the namespaces

In pod-definition.yaml in metadata we can add namespace to mention the namespace where the pod will be deployed

# nginx-pod.yaml

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

**namespace: dev**

  labels:

    app: nginx-pod

spec:

  containers:

    - name: nginx

      image: nginx

**kubectl -n dev get pods –no-header** -> to get the pods in dev namespace

In which namespace the nginx pod is deployed?

Ans: **kubectl get pods –all-namespaces | grep nginx**

In the same namespace we can connect to another pod via pod name but to connect with other resources in another namespace we have to maintain we proper format.

**Resource-name.namespace.resource-type.domain**

example: **db-service.dev.svc.cluster.local**

db-service is the name of the resource, dev is the namespace, svc is the resource type, cluster.local is he domain

**Some of the imperative style command**

**--dry-run** -> by default all the command is run with this –dry-run. As soon as the command is run the resource will be created.

**--dry-run=client** -> It will not create the resource rather it will check the whole command and tell us that the command is correct or not.

-**o yaml > resource-definition.yaml** -> this will create the resource definition in an yaml format

**kubectl create deployment nginx –image=nginx –dry-run=client -o yaml > nginx-deployment.yaml** -> It will first check the command is correct or not. If it is correct then will create one nginx-deployment file with the configuration given

**kubectl expose pod redis –port=6379 –name=redis-service –dry-run=client -o yaml > redis-service.yaml** -> it will create a redis-service yaml with the configuration given

**kubectl create service clusterip redis-service –tcp=6379:6379 –dry-run=client -o yaml > redis-service.yaml** -> it will create a dry run of the redis service and save it in redis-service.yaml

**kubectl expose pod nginx-pod –port=80 –name=nginx-service –type=NodePort –dry-run=client -o yaml > nginx-service.yaml** -> It will expose pod named as nginx pod type of NodePort4

**kubectl create service nodeport nginx-service –tcp=80:80 –node-port=30080 –dry-run=client -o yaml > nginx-service.yaml** -> It will create a service of nodeport with port 80 and tagetport 80 with nodeport 30080

kubectl expose command automatically use the pods labels as the selctors but we can not specify the nodeport. We must add that in the definition file then we can add the nodeport.

Kubectl create service command will not use the pod labels as selectors instead it will assume selectors as **app: service-name** and we can not pass selector in the definition file.

**Commands and args**

In docker we have **ENTRYPOINT** and **CMD** for giving command line arguments, but we can override that using **–entrypoint** and the **extra parameters** passed in the docker run command. Same way we can override the existing command in the pod definition file with **command** and **args. ENTRYPOINT** will associated with **command** and **CMD** will be associated with **args.**

Let’s say we have a dockerfile like this

name: ubuntu-sleeper

image: ubuntu-sleeper

command: [“sleeper2.0”]

args: [“100”]

#ubuntu-sleeper dockerfile

FROM UBUNTU

ENTRYPOINT [“sleep”]

CMD [“10”]

The right-side pod definition file is same as this docker run command:

**docker run –entry-point=sleeper2.0 ubuntu-sleeper 100**

Copy the content of existing pod into a yaml -> **kubectl get pod <pod-name> -o yaml > pod-definition.yaml**

**Environment variables**

We can pass environment variables in these 3 ways.

1. Using env
2. Using config map key valueFrom -> configMapKeyRef
3. Using secret key valueFrom ->secretKeyRef

spec:

  name: postgres

  image: postgres

  env:

  - name: POSTGRES\_PASSWORD

    value: MY\_SECRET\_PASSWORD

spec:

  name: postgres

  image: postgres

  env:

  - name: POSTGRES\_PASSWORD

    valueFrom:

      configMapKeyRef:

        name: posgtres-config

        key: password

        name: postgres-secret

        key: password

spec:

  name: postgres

  image: postgres

  env:

  - name: POSTGRES\_PASSWORD

    valueFrom:

      secretKeyRef:

        name: postgres-secret

        key: password

We can either use **configMapKeyRef** where we can pass keys, or we can use **configMapRef** where we can directly pass the **configMap**. Same goes with Secrets. We can either use **secretKeyRef** or **secretRef**.

We can create configMap with these 3 ways.

1. By passing all the values in command.
2. By passing the properties file or yaml file in the command
3. By using the deifinition file.

**Kubectl create configmap app-config –from-literal app-color=blue**: It will create a configmap with color blue

**kubectl create configmap app-config –from-file /path/app.properties:** It weill create a config file with theapp.properties content

**kubectl apply -f app-config.yaml**: It will create a configMap with app-config definition file

**kubectl get configmaps**: It will fetch all the config maps

**kubectl describe configmap app-config**: It will describe the config map app-config

apiVersion: v1

kind: ConfigMap

metadata:

  name: nginx-configmap

data:

  name: nginx

  tier: frontend

  work: loadbalancing

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  containers:

    - name: nginx

      image: nginx

      envFrom:

        - configMapRef:

            name: nginx-configmap

config map stores everything in plain text format, which is not suitable for storing password, that’s why we need secrets.

We can create secret with these 3 ways.

1. By passing all the values in command.
2. By passing the properties file or yaml file in the command
3. By using the deifinition file.

**Kubectl create secret generic app-secret –from-literal app-color=blue**: It will create a secret with color blue.

**Kubectl create secret generic app-secret –from-file app-secret.properties**: It will create a secret with the app-secret.properties file.

**#app.properties**

db\_host=postgres

db\_password=root

db\_name=database

**#app-sercret.yaml**

apiVersion: v1

kind: Secret

metadata:

  name: app-secret

data:

  db\_host: cG9zdGdyZXMNCg==

  db\_password: cm9vdA0K

  db\_name: ZGF0YWJhc2UNCg==

#nginx-with-secret

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  labels:

    name: nginx-pod

spec:

  containers:

  - name: nginx

    image: nginx

    envFrom:

      - secretRef:

          name: app-secret

There are some other ways to store the information. That is Helm secrets, HarshiCorp vault etc.

**Security Context**

In each container there can be multiple processes running and there can be multiple processes running. These processes are separated by their namesapces. By default, docker runs every process as root user, but we can change it with the following command.

**Docker run –user=1000 ubuntu sleep 10**

docker limits the capabilities of the root user inside the container. It is not the same root user as the host machine root user. Via **linux capabilities** we can add or remove capabilities of the root user inside the container.

**Docker run –cap-add MAC\_ADMIN ubuntu**: It will run ubuntu container with MAC\_ADMIN privilege

**docker run –cap-drop kill ubuntu**: It will drop the privilege of kill a process

**docker run –privileged ubuntu**: It will add all the privileges

This can be configured in the Kubernetes as well. In the pod level and in the container level.

apiVersion: v1

kind: Pod

spec:

  containers:

  - name: ubuntu-pod

    image: ubuntu

    securityContext:

      runAsUser: 1000

      capabilities:

        add: ["MAC\_ADMIN","KILL"]

There are 2 types of accounts in Kubernetes space:

1. User Account: Human user such as admin or developer
2. Service Account: Account created by application to interact with the Kubernetes clusters like monitoring app or build tools such as Jenkins.

**Kubectl create serviceaccount dashboard-serviceaccount**: It will create a service account named as dashboard-serviceaccount. It will also create a secret with a token automatically. Now with this token we can call kubernetes api. Like the following

curl <https://192.168.56.70/api> -insecure --header “Authorization Bearer #TOKEN”

We can create a service account, assign that right permission using role-based access control mechanism and expose the service account token and use it to configure the third-party application to authenticate Kubernetes api.

But in latest version we have to create the token manually and then we have to bind it with service account. **kubectl create sa dashboard-sa**

apiVersion: v1

kind: Secret

metadata:

  name: dashboard-sa-token

  annotations:

    kubernetes.io/service-account.name: "dashboard-sa"

type: kubernetes.io/service-account-token

If the third-party application is hosted in the same Kubernetes, then we can mount the service account token as volume.

We can edit the service account inside the POD, but we cannot change the service account of the deployment. After change there will be a new rollout deployment.

apiVersion: v1

kind: Pod

metadata:

  name: nginx

spec:

  containers:

  - image: nginx

    name: nginx

    volumeMounts:

    - mountPath: /var/run/secrets/tokens

      name: vault-token

  serviceAccountName: build-robot

  volumes:

  - name: vault-token

    projected:

      sources:

      - serviceAccountToken:

          path: vault-token

          expirationSeconds: 7200

          audience: vault

apiVersion: apps/v1

kind: Deployment

metadata:

  name: nginx-deployment

spec:

  selector:

    matchLabels:

      app: nginx-pod

  template:

    metadata:

      labels:

        app: nginx-pod

    spec:

      containers:

      - name: nginx

        image: nginx

      serviceAccountName: nginx-serviceaccount

      automountServiceAccountToken: false

By default, there is service account that is default service account.

**Kubectl get serviceaccount**: It will fetch all the service account

**kubectl describe serviceaccount default**: It will describe the default service account

**kubectl describe secret <token-from- sa>:** With this command we can fetch the token from sa.

This default service account is associated with **image pull registry** and **image pull secrets**. We can also change it. For reference check this: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/>

**Resource requirements**

apiVersion: v1

kind: LimitRange

metadata:

  name: mem-limit-range

  namespace: dev

spec:

  limits:

  - default:

      memory: "512Mi" #cpu usage

    defaultRequest:

      memory: "256Mi" #Memory usage

    type: Container

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  containers:

    - name: nginx

      image: nginx

      resources:

        requests:

          memory: "64Mi"

          cpu: "250m"

        limits:

          memory: "128Mi"

          cpu: "500m"

By default, in Kubernetes there is not restriction on usage and cpu usage. With the Limit range we can set the default memory and cpu of a pod in any namespace. With resource block of the pod definition, we can also restrict the default and maximum memory usage.

**Kubectl apply -f mem-limit-range.yaml**: It will create the memory limit of the container in any namespace

**Taints and tolerations**

It helps to set restrictions to what pod to schedule in which node.

Suppose we have 3 nodes and 4 pods, and we have applied a taint blue on **node-A** and we have added tolerant to **pod-B** so only **pod-B** is capable to be allocated in **node-A**. But in the same time pod-B can also be allocated to another node. Taints imposes a rule on the node that it will only be accepting pods with specific tolerations, but it does not impose any rule on pods.

**Kubectl nodes <node-name> key=value:taint-effect:** It will impose a taint with key value with taint effect. There are 3 taint effects 1. NoExecute 2. PrefferNoSchedule 3. NoSchedule

kubectl nodes node-a color:blue:NoSchedule

By default, the master node has some taints that prevents any pods to schedule on master.

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  tolerations:

    - key: "color"

      operator: "Equal"

      value: "blue"

      effect: "NoSchedule"

  containers:

    - name: nginx

      image: nginx

**Kubectl describe node <node-name> | grep Taint**: to find the taints applied on the node

**kubectl nodes <node-name> key=value:taint-effect - :** It will impose a taint with key value with taint effect.

**Kubectl get pods -o wide**: To see a pod in which node

**Node selectors and Node Affinity**

**kubectl label node <node-name> key=value:** To label a node

The specific pod with specific node with **nodeSelector with key value** will be placed on that node.

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  nodeSelector:

    app: color

  containers:

    - name: nginx

      image: nginx

Node selector is easy to use but lack the advanced features that is why we are using node Affinity. Here also at first, we must label the node

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  affinity:

    nodeAffinity:

      requiredDuringSchedulingIgnoredDuringExecution:

        nodeSelectorTerms:

          - matchExpressions:

            - key: size

              operator: In

              values:

              - "Large"

              - "Medium"

  containers:

    - name: nginx

      image: nginx

To place this nginx pod we must label the node with Large or Medium else pod will not be executed in the node.

There are 3 types of operators:

1. **In**: If the any value from values is there in the labels in the node
2. **No**: If the values from the values are there not in labels of thr node
3. **Exists**: If the key exists in the label of the node

There are 3 types of nodeAffinity selectors.

1. requiredDuringSchedulingIgnoredDuringExecution
2. prefferedDuringSchedulingIgnoredDuringExecution
3. requiredDuringSchedulingRequiredDuringExecution (Available in future)

|  |  |  |
| --- | --- | --- |
| Name | During scheduling | During execution |
| requiredDuringSchedulingIgnoredDuringExecution | Label is required in the node other wise pod will not be executed | In if there is any change in label in node then pod will ignore that and continue executing |
| prefferedDuringSchedulingIgnoredDuringExecution | It will try to find the node with the label. If it does not find the node, then it tries to place the pod in any node | In if there is any change in label in node then pod will ignore that and continue executing |
| requiredDuringSchedulingRequiredDuringExecution | Label is required in the node otherwise pod will not be executed | In if there is any change in label in node then pod should have that label other wise it will stop executing |

With the combination of taints and node affinity we can specify that which pod will place on which node.

**Multi container pod**

A multi container is something that has more than one container in pod.

1. **Side car** -> Log agent with a web server
2. **Adapter** -> Log agent that process different type of web server and pushes in the central log server.
3. **Ambassador**-> Microservice with DB agent which connects to different type and environments of database.

**Readiness and liveness probe**

**Readiness probe**: Sometimes container is up that does not mean that the container is ready for external communication like Jenkins takes time to boot up. So, we can configure an api or a script to check the container is ready or not.

**Liveness probe:** It is way to check periodically that the application is healthy or not, otherwise it destroys the container and starts a new one.

There are 3 ways to check:

1. httpGet
2. tcpSocket
3. start-up script

we can also add initialDelaySeconds, periodSeconds, failureThreshold configure other options

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  containers:

    - name: nginx

      image: nginx

      readinessProbe:

        exec:

- "cat"

- "/app/ready"

      livenessProbe:

        exec:

- "cat"

- "/app/ready"

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  containers:

    - name: nginx

      image: nginx

      readinessProbe:

        tcpSocket:

          port: 8080

      livenessProbe:

        tcpSocket:

          port: 8080

apiVersion: v1

kind: Pod

metadata:

  name: nginx-pod

  namespace: default

  labels:

    app: nginx-pod

spec:

  containers:

    - name: nginx

      image: nginx

      readinessProbe:

        httpGet:

          path: /api/ready

          port: 8080

          httpHeaders:

          - name: Custom-Header

            value: Awesome

        initialDelaySeconds: 10

        periodSeconds: 5

        failureThreshold: 8

      livenessProbe:

        httpGet:

          path: /api/ready

          port: 8080

          httpHeaders:

          - name: Custom-Header

            value: Awesome

        initialDelaySeconds: 10

        periodSeconds: 5

        failureThreshold: 8

**Container logging**

**docker run -d kodecloud/event-simulator**

**docker log -f <container-name>**

this is how we can check the logs of a docker container after running it in detached mode. We can do the same with Kubernetes.

**kubectl create -f event-simulator.yaml**

**kubectl logs -f event-simulator-pod:** To get logs of a specific pod

**kubectl logs -f <pod-name> <container-name>:** To get the logs of a specific container of a specific pod

Monitoring Solutions:

1. Prometheus
2. Elastic stack
3. Datadog
4. Dynatree

Kubernetes runs an agent on each node known as kubelet which is responsible for receiving instruction from Kubernetes master server. It has a subcomponent known as container advisor, it is responsible for retrieving performance metrics from pods and exposing them to kubelet api.

**Minikube addons enable metrics-server:** To enable the metric server on minikube

**kubectl top node:** To find the node with max using the resources

**kubectl top pod:** To find the pod with the max using resources.

**Pod Design**

Labels and selectors are standard methods to group things together. We can keep these labels under metadata. These labels are used to identify a specific pod by the service, replica set and deployment. Though one label is sufficient to identify but we can use as many labels as we want.

Alongside with labels, we can also add annotations. It used to save build information and other things to documentation purpose.

**kubectl get pods -l env=dev**: It will find the pods where label is dev.

**kubectl get pods -l env=dev –no-headers | wc -l:** It will find the count of the pods with this label.

**kubectl get all -l env=prod:** It will find all the objects in prod environment.

**Kubectl get all -l env=prod –no-headers | wc -l :** It will count all the objects in prod environment.

**Kubectl get pods -l env=prod,bu=finance,tier=frontend:** It will find all the pods with all these label.

When we first create a deployment, it creates a rollout, and a new rollout creates a new deployment revision. When there is a change in deployment like image or something then again it creates a rollout which again creates a revision. It helps us to keep track of the changes.

There are two types of deployment strategy.

1. **Recreate:** Destroy all the container then again create the container with changes. During the period when the older version is down the application become inaccessible.
2. **Rolling update:** In this strategy some of the old containers are down and int that place new containers with change are started. If there is no mentioning of the deployment strategy in the yaml file, then this strategy become the default strategy.

If we perform the **kubectl describe deployment <deployment-name>** then in the events, we can see the events how containers went down and up.

When there is upgrade the deployment object internally creates a replicate set and then fill that with the new containers and delete the containers from old replica set.

To undo a change, use the following command

**kubectl rollout undo deployment/<deployment-name>**

In rollback also deployment object creates another replica set and fill that.

Summarize commands:

**kubectl create -f <deployment-definition.yml>**

**kubectl get deployments**

**kubectl apply -f <deployment-definition.yml>**

**kubectl set image deployment/<deployment-name> nginx=nginx:1.9.1**

**kubectl rollout status deployment/<deployment-name>**

**kubectl rollout history deployment/<deployment-name>**

**kubectl rollout undo deployment/<deployment-name>**

Updating a Deployment

Here are some handy examples related to updating a Kubernetes Deployment:

Creating a deployment, checking the rollout status and history:

In the example below, we will first create a simple deployment and inspect the rollout status and the rollout history:

***master $*** kubectl create deployment nginx --image=nginx:1.16

deployment.apps/nginx created

***master $*** kubectl rollout status deployment nginx

Waiting for deployment "nginx" rollout to finish: 0 of 1 updated replica are available...

deployment "nginx" successfully rolled out

***master $*** kubectl rollout history deployment nginx

deployment.extensions/nginx

REVISION CHANGE-CAUSE

1 <none>

Using the --revision flag:

Here the revision 1 is the first version where the deployment was created.

You can check the status of each revision individually by using the –revision flag:

master $ kubectl rollout history deployment nginx --revision=1

deployment.extensions/nginx with revision #1

Pod Template:

Labels: app=nginx pod-template-hash=6454457cdb

Containers: nginx: Image: nginx:1.16

Port: <none>

Host Port: <none>

Environment: <none>

Mounts: <none>

Volumes: <none>

master $

Using the --record flag:

You would have noticed that the “change-cause” field is empty in the rollout history output. We can use the --record flag to save the command used to create/update a deployment against the revision number.

You can now see that the change-cause is recorded for the revision 2 of this deployment.

Let’s make some more changes. In the example below, we are editing the deployment and changing the image from nginx:1.17 to nginx:latest while making use of the –record flag.

master $ kubectl set image deployment nginx nginx=nginx:1.17 --record

deployment.extensions/nginx image updated

master $master $

master $ kubectl rollout history deployment nginx

deployment.extensions/nginx

REVISION CHANGE-CAUSE

1 <none>

2 kubectl set image deployment nginx nginx=nginx:1.17 --record=true

master $ kubectl edit deployments. nginx --record

deployment.extensions/nginx edited

master $ kubectl rollout history deployment nginx

REVISION CHANGE-CAUSE

1 <none>

2 kubectl set image deployment nginx nginx=nginx:1.17 --record=true

3 kubectl edit deployments. nginx --record=true

master $ kubectl rollout history deployment nginx --revision=3

deployment.extensions/nginx with revision #3

Pod Template: Labels: app=nginx

pod-template-hash=df6487dc Annotations: kubernetes.io/change-cause: kubectl edit deployments. nginx --record=true

Containers:

nginx:

Image: nginx:latest

Port: <none>

Host Port: <none>

Environment: <none>

Mounts: <none>

Volumes: <none>

Undo a change:

Let’s now rollback to the previous revision:

master $ kubectl rollout undo deployment nginx

deployment.extensions/nginx rolled back

master $ kubectl rollout history deployment nginx

deployment.extensions/nginxREVISION CHANGE-CAUSE

1 <none>

3 kubectl edit deployments. nginx --record=true

4 kubectl set image deployment nginx nginx=nginx:1.17 --record=true

master $ kubectl rollout history deployment nginx --revision=4

deployment.extensions/nginx with revision #4Pod Template:

Labels: app=nginx pod-template-hash=b99b98f9

Annotations: kubernetes.io/change-cause: kubectl set image deployment nginx nginx=nginx:1.17 --record=true

Containers:

nginx:

Image: nginx:1.17

Port: <none>

Host Port: <none>

Environment: <none>

Mounts: <none>

Volumes: <none>

master $ kubectl describe deployments. nginx | grep -i image:

Image: nginx:1.17

With this, we have rolled back to the previous version of the deployment with the image = nginx:1.17.

**Jobs and CronJobs**

When we have a requirement that we need a certain type of job to execute and finish like batch processing, data processing, analytics then on that time rather than creating and pod we can create any job.

Because the pods keep on restarting when it got terminated but we need to start, execute and finish. The task which are meant to live for a short period of time.

Like when we do the following command:

**docker run ubuntu expr 3+2**

It will run execute and terminate, since the task was completed.

We can replicate the same with Kubernetes with job.

apiVersion: v1

kind: Pod

metadata:

  name: Math-app-pod

  labels:

    name: math-app-pod

spec:

  containers:

  - name: math-app-pod

    image: ubuntu

    command: ["expr", "3", "+", "2"]

By default, ubuntu will run execute and finish and so did the pod. It will go to the complete state. But in attempt of keep on running Kubernetes will keep this pod restarting. This will happen continuously until the threshold is reached. This is default behaviour of the pod and there is a property mentioned under the **spec** which is **restartPolicy**. By default, its value is **Always**.

We can set this value to **Never** to stop restarting.

We can create a job in place of this pod.

apiVersion: batch/v1

kind: Job

metadata:

  name: math-app-job

  labels:

    name: math-app-job

spec:

  template:

    spec:

      containers:

      - name: math-app-pod

        image: ubuntu

        command: ["expr", "3", "+", "2"]

      restartPolicy: Never

**kubectl create -f math-add-job.yaml**: To create a job

**kubectl get jobs**: To get all the jobs

Kubernetes internally create a pod for this and once it got completed it will not try to restart.

We can see the output from the logs of the pod.

**kubectl logs <pod-name>**

To run multiple jobs, we can add **completion** property under **spec**. By default, the pods are created one after another like 2nd pod is created only after the 1st pod is started and finished. If the pod got failed it will try to bring a new pod until it reaches the required number of jobs mentioned. So, we can run the jobs parallelly to save some time. To add parallelism, we can add **parallelism** property under the **spec**.

spec:

  completions: 3

  parallelism: 3

  template:

    spec:

      containers:

      - name: random-error-job

        image: kodekloud/random-error

      restartPolicy: Never

CronJob is a job that can be scheduled with cron expression

apiVersion: batch/v1beta1

kind: CronJob

metadata:

  name: math-cronjob

  labels:

    name: math-cronjob

spec:

  schedule: "\*/1 \* \* \* \*"

  jobTemplate:

    spec:

      completions: 3

      parallelism: 3

      template:

        spec:

          containers:

          - name: math-cronjob

            image: ubuntu

            command: ["expr", "3", "+", "2"]

          restartPolicy: Never

**kubectl create -f math-cronjob.yaml**

**kubectl get cronjob**

**Kubernetes Services/Ingress**

Some references for ingress:

<https://medium.com/digitalfrontiers/kubernetes-ingress-with-nginx-93bdc1ce5fa9>

<https://medium.com/devops-mojo/kubernetes-ingress-overview-what-is-kubernetes-ingress-introduction-to-k8s-ingress-b0f81525ffe2>

<https://katharharshal1.medium.com/setup-nginx-ingress-controller-on-kubernetes-cluster-dd48b2b1ab61>

<https://www.containiq.com/post/kubernetes-ingress>  
<https://www.edc4it.com/blog/k8s-ingress-tls-termination>

ingress controller:

<https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27.0/deploy/static/mandatory.yaml>

<https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27.0/deploy/static/provider/cloud-generic.yaml>

minikube addons enable ingress  
minikube addons enable metrics-server  
minikube dashboard

kubectl proxy --port=8080

curl http://localhost:8080/api/

<https://github.com/nginxinc/kubernetes-ingress>

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

kubectl get all --all-namespaces  
kubectl get cm --all-namespaces  
kubectl describe cm kube-root-ca.crt -n <namespace>  
kubectl delete services,deployments,rs,pods,cm,secrets,job.batch --all -n <namespace>  
kubectl get cm,secrets --all-namespaces  
kubectl delete --all -n ingress-nginx

kubectl create deployment hello-minikube --image=kicbase/echo-server:1.0

kubectl expose deployment hello-minikube --type=NodePort --port=8080

kubectl get services hello-minikube

minikube service hello-minikube

kubectl port-forward service/hello-minikube 7080:8080

kubectl create deployment balanced --image=kicbase/echo-server:1.0

kubectl expose deployment balanced --type=LoadBalancer --port=8080

minikube tunnel

kubectl get services balanced

minikube config set memory 9001

minikube addons list

minikube start -p aged --kubernetes-version=v1.16.1

minikube delete --all

Now, in k8s version **1.20+** we can create an Ingress resource from the imperative way like this:-

**Format - kubectl create ingress <ingress-name> --rule="host/path=service:port"**  
**Example - kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear\*=wear-service:80"**

Find more information and examples in the below reference link:-

<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-em-ingress-em->

**References:-**

<https://kubernetes.io/docs/concepts/services-networking/ingress>  
<https://kubernetes.io/docs/concepts/services-networking/ingress/#path-types>

**if minikube driver is none then follow this:**

minikube addons enable ingress / else download all the ingress controller configs and apply them

kubectl apply -f namespace.yaml

kubectl apply -f deployments.yaml

kubectl apply -f services.yaml

kubectl apply -f ingress.yaml / ingress-resource.yaml

minikube tunnel  
update /etc/host file -> 127.0.0.1 app.reactor.com

we are setting 127.0.0.1 not 192.168.49.2 because in minikube local ingress is in 127.0.0.1

14:35:14:~ kubectl get ingress -A

**NAMESPACE NAME CLASS HOSTS ADDRESS PORTS AGE**

default echo <none> app.reactor.com 192.168.49.2 80 6h6m

Don’t use this annotation in metadata nginx.ingress.kubernetes.io/rewrite-target: /

It is used to rewrite the path. **/echo/sample** will be converted to **/**

Ingress resource port is always set to 80/443

---

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: echo

# annotations:

# nginx.ingress.kubernetes.io/rewrite-target: /

spec:

rules:

- host: app.reactor.com

http:

paths:

- backend:

service:

name: echo

port:

number: 80

pathType: Prefix

path: /echo

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: echo

labels:

type: echo

spec:

replicas: 2

selector:

matchLabels:

type: echo

template:

metadata:

labels:

type: echo

spec:

containers:

- name: echo

image: abhishek1009/echo:latest

ports:

- containerPort: 8000

---

apiVersion: v1

kind: Service

metadata:

name: echo

spec:

selector:

type: echo

ports:

- port: 80

targetPort: 8000

We can write as many rules as we want.

**Network Policies**

Let’s say there is a DB pod and one frontend pod and one backend pod. If we create a service to expose the DB pod via service into some specific port then the DB pod will be accessible by all other pods, but we don’t want that to happen. So we can create some rules or **NetworkPolicy** on DB pod that who can access the pod and whom the db pod can access. There are the ingress policy and egress policy. As name suggest ingress means incoming traffic rule to specific pod and egress means outgoing traffic rules to specific pods.

Kubernetes is configured with all allow rule, that means any pod in any node can connect to any pod. To apply any ingress or egress rule we have to apply **NetworkPolicy**. Just like services and replicasets/deployment Kubernetes will use label and selectors to apply the policies.

---

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: nginx-policy

spec:

podSelector:

matchLabels:

app: nginx

policyTypes:

- Ingress

- Egress

ingress:

- from:

- podSelector:

matchLabels:

type: frontend-api

namespaceSelector:

matchLabels:

env: prod

ipBlock:

cidr: 192.168.5.10/32

ports:

- protocol: TCP

port: 80

egress:

- to:

- podSelector:

matchLabels:

type: backend-api

namespaceSelector:

matchLabels:

env: prod

ipBlock:

cidr: 192.168.5.17/32

ports:

- protocol: TCP

port: 8080

---

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: db-policy

spec:

podSelector:

matchLabels:

type: master-db

policyTypes:

- Ingres

ingress:

- from:

- podSelector:

matchLabels:

type: backend-api

namespaceSelector:

matchLabels:

env: prod

ipBlock:

cidr: 192.168.5.10/32

ports:

- protocol: TCP

port: 3306

In the previous two network policy yamls, for one policy there is only ingress and for another there are both. So by this configuration the first pof only allow ingress and second pod allow both ingress and egress. But we must mention that in **policyType** otherwise no configuration will work.

If we look closely then we can see that **to** and **from** is array so if we give podSelector, namespaceSelector and ipBlock in same element then it will act as an and operation , that means from the app in namespace prod has the matching label and has that specific ip can only access this pod or we can say that this pod will only accept connection from a pod if that pod fulfilled all these 3 conditions.

If these conditions are given in separate items, then it will act as or operation like the following configuration.

- from:

- podSelector:

matchLabels:

type: backend-api

namespaceSelector:

matchLabels:

env: prod

- ipBlock:

cidr: 192.168.5.10/32

- from:

- podSelector:

matchLabels:

type: backend-api

namespaceSelector:

matchLabels:

env: prod

- podSelector:

matchLabels:

type: third-party-api

namespaceSelector:

matchLabels:

env: outsiders

For the 2nd configuration the pod will accept connection from the pod which either be a prod app with backend-api label or a outsiders app with third-party-api label.

Network policies are enforced by the network solutions implemented by the Kubernetes clusters. There are some solutions that doesn’t support network policies.

Solutions that Support Network Policies:

•Kube-router

•Calico

•Romana

•Weave-net

Solutions that DO NOT Support Network Policies:

•Flannel

Even in a cluster configured with a solution that does not support network policies, you can still create the policies, but they will just not be enforced. You will not get an error message saying the networking solution does not support network policies.

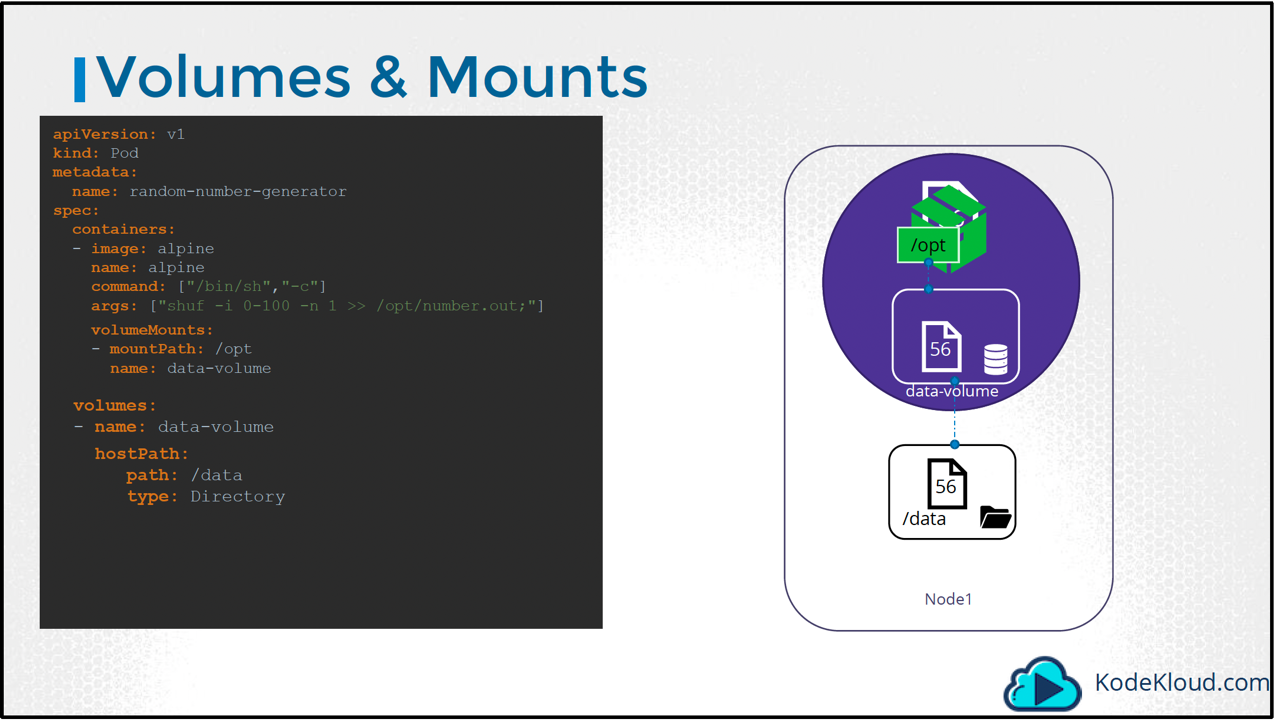
To get networkpolicy -> kubectl get networkpolicies -A

**State Persistence**

Let us look at volumes in Docker first. Docker containers are meant to be transient in nature. Which means they are meant to last only for a short period of time. They are called upon when required to process data and destroyed once finished. The same is true for the data within the container. The data is destroyed along with the container.

To persist data processed by the containers, we attach a volume to the containers when they are created. The data processed by the container is now placed in this volume, thereby retaining it permanently. Even if the container is deleted, the data generated or processed by it remains.

So how does that work in the Kubernetes world. Just as in Docker, the PODs created in Kubernetes are transient in nature. When a POD is created to process data and then deleted, the data processed by it gets deleted as well. For this we attach a volume to the POD. The data generated by the POD is now stored in the volume, and even after the POD is delete, the data remains.



We have a single node Kubernetes cluster. We create a simple POD that generates a random between 1 and 100 and writes that to a file at /data/number.out and then gets deleted along with the random number. To retain the number generated by the pod, we create a volume. And a Volume needs a storage. When you create a volume, you can choose to configure its storage in different ways. We will look at the various options in a bit, but for now we will simply configure it to use a directory on the host. In this case I specify a path /data on the host. This way any files created in the volume would be stored in the directory data on my node.

Once the volume is created, to access it from a container we mount the volume to a directory inside the container. We use the volumeMounts field in each container to mount the data-volume to the directory /opt within the container. The random number will now be written to /opt mount inside the container, which happens to be on the data-volume which is in fact /data directory on the host. When the pod gets deleted, the file with the random number still lives on the host.

However, it is not recommended for use in a multi-node cluster. This is because the PODs would use the /data directory on all the nodes and expect all of them to be the same and have the same data. Since they are on different servers, they are in fact not the same, unless you configure some kind of external replicated clustered storage solution.

Kubernetes supports several types of standard storage solutions such as NFS, glusterFS, Flocker, FibreChannel, CephFS, ScaleIO or public cloud solutions like AWS EBS, Azure Disk or File or Google’s Persistent Disk.

For example, to configure an AWS Elastic Block Store volume as the storage or the volume, we replace hostPath field of the volume with awsElasticBlockStore field along with the volumeID and filesystem type. The Volume storage will now be on AWS EBS.

