kubernetes(k8s):

Kubernetes is a system for managing containerized applications across a cluster of nodes. In simple terms, you have a group of machines (e.g. VMs) and containerized applications (e.g. Dockerized applications), and Kubernetes will help you to easily manage those apps across those machines.

or

Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF).

Components of k8s:

Pod:

A pod is the smallest deployable unit that can be managed by Kubernetes. A pod is a logical group of one or more containers that share the same IP address and port space.

Nodes (Minions):

A node is a machine either physical or virtual machine on which Kubernetes is installed. A node is a worker machine and this is where containers inside the pods will be launched by Kubernetes.

Cluster:

If our node fails, our app will be down. So a cluster is a set of nodes grouped together. Even if one node fails, your application will still be accessible from the other nodes. In addition, having multiple nodes helps in sharing the computational load as well.

A Kubernetes cluster consists of one or more nodes managed by Kubernetes. The nodes are bare-metal servers, on-premises VMs, or VMs on a cloud provider. Every node contains a container runtime (for example a Docker Engine), Kubelet (responsible for starting, stopping, and managing individual containers by requests from the Kubernetes control plane), and kube-proxy (responsible for networking and load balancing).

Architecture:

K8s comes with master and worker architecture.

Master Node:

Who is responsible for managing the cluster? Where is the information about the members of the cluster stored? How are the nodes monitored? When a node fails, how do you move the workload of the failed node to another worker node? â€” Here comes Master Node

A Kubernetes cluster also contains one or more master nodes that run the Kubernetes control plane. The control plane consists of different processes, such as an

\*API server (provides JSON over HTTP API),

\*scheduler (selects nodes to run containers),

\*controller manager (runs controllers), and

\*etcd (a globally available configuration store).

Note: When you install Kubernetes on a System, you are actually installing the following components: an API Server, an ETCD service, a kubelet service, a Container Runtime, Controllers. and Schedulers.

Note:

Horizontal scaling means scaling by adding more machines to your pool of resources (also described as â€œscaling outâ€), whereas vertical scaling refers to scaling by adding more power (e.g. CPU, RAM) to an existing machine (also described as â€œscaling upâ€).

1.Kube-APIServer:

\*It is the main management point(front-end) of the entire cluster. APIs allow applications to communicate with one another.

\*When we interact with the k8s cluster using kubectl command, we are actually communicating with the API-server.

\*It is the only component that connects to etcd, all the components must go through API-server for communication.

\*It is resposible for the authentication and authorization.

\*It also implements a watch mechanism to maintain a desired state, if anything is changed it will try and revert it back to desired state.

It is designed to scale horizontally â€” that is, it scales by deploying more instances. The users, management devices, and command line interfaces all talk to the API server to interact with the Kubernetes cluster.

2.Kube-Scheduler:

\*The scheduler is responsible for workload utilization and allocating pod to a new node.

\*It watches newly created pods that have no node assigned and selects a node for them to run on and binds them to the nodes via the binding pod subresource API.

\*it is responsible for deciding to which node a new pod must be allocated based on the availability of requested resources, service requirements, affinity and anti-affinity specifications.

3.Kube-Controller-manager:

\*This is a component on the master that runs controllers which watches the state of the cluster through the api server watch feature.

\*each and every controller always tries to match actual state with the desired state. it will send message to the k8s API-server to make any necessary changes.

\*Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

These controllers include:

\***Node Controller**: Responsible for noticing and responding when nodes go down.

\***Replication Controller**: Responsible for maintaining the correct number of pods for every replication controller object in the system.

\***Endpoints Controller**: Populates the Endpoints object (that is, it joins Services and Pods).

\***Service Account and Token Controllers**: Create default accounts and API access tokens for new namespaces.

\***Node Controller**: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding.

\*Route Controller: For setting up routes in the underlying cloud infrastructure.

\***Service Controller**: For creating, updating, and deleting cloud provider load balancers.

\***Volume Controller**: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes.

**4.ETCD:**

\*It is a distributed, consistent(high availability) key-value store which stores the configuration information of the complete k8s cluster.

\*configuration information represents the desired state of the cluster. eg: what/which nodes exits in the cluster, what pods are running and in which node and all the configuration of those objects.

\*to back up the cluster, we need to backup the etcd.

Kubernetes Node Components:

**Kubelet**:

\*Kubelet is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

\*Itâ€™s a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and write values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

**Kube-Proxy:**

This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containersâ€™ health checkup, etc.

how to transfer an ami from one region to another region?

====================================================================================================================

====================================================================================================================

https://github.com/justmeandopensource/kubernetes/blob/master/docs/install-cluster-ubuntu-20.md

after installation open inbound port 6443 for master node in aws(to allow worker nodes to join master).

creating pods:

apiVersion: v1

kind: Pod

metadata:

name: static-web

labels:

role: myrole

spec:

containers:

- name: web

image: nginx

ports:

- name: web

containerPort: 80

protocol: TCP

https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/pods/simple-pod.yaml

kubectl apply -f pod.yaml

kubectl get pods

to get the list of kinds and versions:

for kind in `kubectl api-resources | tail +2 | awk '{ print $1 }'`; do kubectl explain $kind; done | grep -e "KIND:" -e "VERSION:"

creating deployment:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

livenessProbe:

httpGet:

path: /

port: 3000

initialDelaySeconds: 5

https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/application/deployment.yaml

what happens if I run kubectl command?

steps to backup the etcd?

\*play.etcd.io

======================================================================\

kubernetes objects:

Kubernetes objects are entities provided by Kubernetes for deploying, maintaining, and scaling applications either on cloud or on-premise infrastructure.

or

Kubernetes Objects are persistent entities in the cluster. These objects are used to represent the state of the cluster.

The following are some of the Kubernetes Objects:

\*Pod

\*NameSpaces

\*ReplicaSet (Manages Pods)

\*Deployment (Manages Pods)

\*StatefulSets

\*DaemonSets

\*Services

\*ConfigMaps

\*Volumes

kubectl api-resources

1.Deployment:

\*is the most common way of creating pods.

\*it provides updates for pods and its replica sets.

uses:

\*we create deployment to get replicas of pods.

\*we can rollback to an earlier deployment revision.

\*we can scale-up and scale-down for load balancing.

apiVersion - Which version of the Kubernetes API you're using to create this object

kind - What kind of object you want to create

metadata - Data that helps uniquely identify the object, including a name string, UID, and optional namespace(all are pre-defined keys)

spec - What state you desire for the object (the configurations defined for the desired state of the objects).

labels and selectors:

\*Labels are nothing but key-value pairs assigned to Kubernetes Objects like Pods, Service, etc.

\*Labels can be used to organize and to select Kubernetes objects.

\*Labels can be attached to objects at creation time and can be modified at any time.

\*We can add multiple Labels to Kubernetes objects.

multiple labels:

labels: {

app: nginx

tier: frontend

}

or

labels:

- app: nginx

- tier: frontend

selectors: using the matching label of objects, selectors can be used to identify and group this objects.

types:

equity/equality based selectors: This allows filtering by key and value, where matching objects should satisfy all the specified labels.

only three types of operators are allowed:

=, ==, !=

eg:

environment != production

app == nginx

tier = frontend

set based selectors: This allows filtering keys according to a set of values.

only three types of operators are allowed:

in, notin and exists

eg:

environment in (production, qa, dev)

tier notin (frontend, backend)

replication contoller vs replica set:

Replica Set(rs) and Replication Controller(rc) do almost the same thing. Both of them ensure that a specified number of pod replicas are running at any given time. The difference comes with the usage of selectors to replicate pods. Replica Set uses equality based as well as Set-Based selectors while replication controllers use Equity-Based selectors.

DaemonSet:

\*A DaemonSet ensures that all (or some) Nodes run a copy of a Pod.

\*As nodes are added to the cluster, Pods are added to them.

\*As nodes are removed from the cluster, those Pods are garbage collected(removed).

\*Deleting a DaemonSet will clean up the Pods it created.

Some typical uses of a DaemonSet are:

\*running a cluster storage daemon on every node

\*running a logs collection daemon on every node

\*running a node monitoring daemon on every node

what is the diff between

replica set and replication controller.

daemonset and deployment.

how to list the pods running on a particular node?

==========================================================

kubectl get - list resources

kubectl describe - show detailed information about a resource

kubectl logs - print the logs from a container in a pod

kubectl exec - execute a command on a container in a pod

https://kubernetes.io/docs/reference/kubectl/cheatsheet/

https://www.magalix.com/blog/understanding-kubernetes-objects

Container states :

Once the scheduler assigns a Pod to a Node, the kubelet starts creating containers for that Pod using a container runtime. There are three possible container states:

Waiting, Running, and Terminated.

pod lifecycle/states/phases:

\*Pending:

The Pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.

\*Running:

The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting.

\*Succeeded:

All containers in the Pod have terminated in success, and will not be restarted.

\*Failed:

All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system.

\*Unknown:

For some reason the state of the Pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the Pod should be running.

probes:

A probe is a diagnosis that is made regularly by the Kubelet on a running container.

liveness probes:

The kubelet uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

readiness probes:

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

startup probes:

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

Configure Probes

Probes have a number of fields that you can use to more precisely control the behavior of liveness and readiness checks:

\*initialDelaySeconds: Number of seconds after the container has started before liveness or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.(it will delay the probe startup until the given time is crossed)

\*periodSeconds: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.

\*timeoutSeconds: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.

\*successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness and startup Probes. Minimum value is 1.

\*failureThreshold: When a probe fails, Kubernetes will try failureThreshold times before giving up. Giving up in case of liveness probe means restarting the container. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

https://www.magalix.com/blog/kubernetes-and-containers-best-practices-health-probes

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#configure-probes

difference between docker swarm and kubernetes.

================================================================================================================

================================================================================================================

https://sysdig.com/blog/debug-kubernetes-crashloopbackoff/

eg:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

<probe-type>:

<probe-action>:

path: <path\_of\_application>

port: 3000

initialDelaySeconds: 5

periodSeconds: 10

timeoutSeconds: 5

successThreshold: 5

failureThreshold: 5

Container probes

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. There are three types of handlers:

\*ExecAction: Executes a specified command inside the container. The diagnostic is considered successful if the command exits with a status code of 0.

\*TCPSocketAction: Performs a TCP check against the Pod's IP address on a specified port. The diagnostic is considered successful if the port is open.

\*HTTPGetAction: Performs an HTTP GET request against the Pod's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

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.

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 respond to 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.

1.HTTP Probe:

The httpGet parameter accepts the path to which it sends the HTTP GET request (in our example, itâ€™s / but in real-world scenarios, it may be something like /api/v1/status). The livenessProbe also accepts initialDelaySeconds parameter which instructs the probe operation to wait for a specified number of seconds before starting. This is useful when your container needs some time to start and be fully functional and restarting it prematurely will make it indefinitely unavailable.

apiVersion: apps/v1

kind: Deployment

metadata:

name: k8s-probes

labels:

app: nginx

spec:

replicas: 1

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

livenessProbe:

initialDelaySeconds: 5

periodSeconds: 5

timeoutSeconds: 60

successThreshold: 2

failureThreshold: 3

httpGet:

host:

scheme: HTTP

path: /

httpHeaders:

- name: Host

value: myapplication1.com

port: 80

2. Exec Probe(shell command execution)

Exec probe executes a command inside the container without a shell. The commandâ€™s exit status determines a healthy state - zero is healthy; anything else is unhealthy.

livenessProbe:

initialDelaySeconds: 1

periodSeconds: 5

timeoutSeconds: 1

successThreshold: 1

failureThreshold: 1

exec:

command:

- cat

- /etc/nginx/nginx.conf

3. TCP Probe(tcp port check):

first it will check the port in the pod is open or not then check if kubelet is able to connect to that specific port. then the action is considered successfull.

livenessProbe:

initialDelaySeconds: 1

periodSeconds: 5

timeoutSeconds: 1

successThreshold: 1

failureThreshold: 1

tcpSocket:

host:

port: 80

list http error types?

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

================================================================================================================

Services:

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them internally and externally. (sometimes this pattern is called a micro-service).

uses:

\*services in k8s is a way of defining network configuration for pods.

\*services always point to pods(they do not point deployment or replicaset etc.), service points to pods directly using labels.

\*in k8s pods can communicate with each other by using IP address in whatever node the pod might be placed.

why do we need service?

Kubernetes pods are ephemeral(lasting for a very short time) in nature. Deployment object(s) can create and destroy pods dynamically. Each pod does have itâ€™s own IP address, hence in a deployment, the set of pods running change all the time, so do the IP address for the pods

types of service:

ClusterIP( will expose the ports):

\*Exposes the Service on a cluster-internal IP.

\*The service is only accessible from within the Kubernetes cluster â€“ you canâ€™t make requests to your Pods from outside the cluster!

\*This is the default ServiceType.

NodePort(will publish the ports):

\*Exposes the Service on each Node's IP at a static port (the NodePort).

\*A ClusterIP Service, to which the NodePort Service routes, is automatically created.

\*You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.

or

\*This makes the service(application running in the pod) accessible on a static port on every node in the cluster.

\*Using NodePort, we can access the pod running inside the k8s cluster from the outside world.

LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.

ExternalName: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

===============================

ClusterIP:

my-cluster-ip-demo.yaml file:

apiVersion: v1

kind: Service

metadata:

name: Backend

spec:

type: ClusterIP

ports:

- targetPort: 80

port: 80

my-demo-pod.yaml file :

apiVersion: v1

kind: Pod

metadata:

name: my-demo-pod

labels:

app: my-test-pod

type: mobile-front-end-app

spec:

containers:

- name: nginx-container

image: nginx

task:

\*modify the above my-cluster-ip-demo.yaml file to connect with the my-demo-pod.yaml

NodePort:

kind: Service

apiVersion: v1

metadata:

name: hostname-service

spec:

type: NodePort

selector:

app: echo-hostname

ports:

- nodePort: 30163

port: 8080

targetPort: 80

with explaination:

kind: Service

apiVersion: v1

metadata:

name: hostname-service

spec:

# Expose the service on a static port on each node

# so that we can access the service from outside the cluster

type: NodePort

# When the node receives a request on the static port (30163)

# "select pods with the label 'app' set to 'echo-hostname'"

# and forward the request to one of them

selector:

app: echo-hostname

ports:

# Three types of ports for a service

# nodePort - a static port assigned on each the node

# port - port exposed internally in the cluster

# targetPort - the container port to send requests to

- nodePort: 30163

port: 8080

targetPort: 80

task:

make nginx application accessible from outside world using NodePort.

LoadBalancer:

This service type works when you are using a cloud provider to host your Kubernetes cluster. When you choose LoadBalancer as the service type, the cluster will contact the cloud provider and create a load balancer. Traffic arriving at this load balancer will be forwarded to the backend pods. The specifics of this process is dependent on how each provider implements its load balancing technology.

Different cloud providers handle load balancer provisioning differently. For example, some providers allow you to assign an IP address to the component, while others choose to assign short-lived addresses that constantly change. Kubernetes was designed to be highly portable. You can add loadBalancerIP to the service definition file. If the provider supports it, it will be implemented. Otherwise, it will be ignored. Letâ€™s have a sample service definition that uses LoadBalancer as its type:

apiVersion: v1

kind: Service

metadata:

name: frontend

spec:

type: LoadBalancer

loadBalancerIP: 78.11.24.19

selector:

app: web

ports:

- protocol: TCP

port: 80

targetPort: 80

=======================================================================

Services without selectors

Services most commonly abstract access to Kubernetes Pods, but they can also abstract other kinds of backends. For example:

\*You want to have an external database cluster in production, but in your test environment you use your own databases.

\*You want to point your Service to a Service in a different Namespace or on another cluster.

\*You are migrating a workload to Kubernetes. While evaluating the approach, you run only a portion of your backends in Kubernetes.

service:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

ports:

- protocol: TCP

port: 80

targetPort: 9376

Because this Service has no selector, the corresponding Endpoints object is not created automatically. You can manually map the Service to the network address and port where it's running, by adding an Endpoints object manually:

EndPoint:

apiVersion: v1

kind: Endpoints

metadata:

name: my-service

subsets:

- addresses:

- ip: 192.0.2.42

ports:

- port: 9376

In Kubernetes Services, we saw that we could use labels to match a frontend service with a backend pod automatically by using a selector. If any new pods had a specific label, the service would know how to send traffic to it. Well the way that the service knows to do this is by adding this mapping to an endpoint. Endpoints track the IP Addresses of the objects the service send traffic to. When a service selector matches a pod label, that IP Address is added to your endpoints.

Multi-Port Services:

For some Services, you need to expose more than one port. Kubernetes lets you configure multiple port definitions on a Service object. When using multiple ports for a Service, you must give all of your ports names so that these are unambiguous. For example:

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- name: http

protocol: TCP

port: 80

targetPort: 9376

- name: https

protocol: TCP

port: 443

targetPort: 9377

=================================

=================================

Namespaces:

You can think of a Namespace as a virtual cluster inside your Kubernetes cluster. You can have multiple namespaces inside a single Kubernetes cluster, and they are all logically isolated from each other. They can help you and your teams with organization, security, and even performance!

\*namespaces are a way of creating virtual subclusters on the same physical cluster.

\*we can create any number of multiple namespaces within one k8s physical cluster and they are all logically isolated from one another.

(logical seperation of cluster resources between multiple users,teams and projects.)

The purpose of Kubernetes namespaces is to define a scope for object names. This enables you to use the same name in other namespaces without the objects getting confused. For example, you can contain the same objects and object names in duplicated environments, like test or dev.

types:

\*default: The default namespace for objects with no other namespace

\*kube-system: The namespace for objects created by the Kubernetes system

\*kube-public: This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

\*kube-node-lease: This namespace for the lease objects associated with each node which improves the performance of the node heartbeats as the cluster scales.

Heartbeats:

Heartbeats, sent by Kubernetes nodes, help determine the availability of a node.

There are two forms of heartbeats: updates of NodeStatus and the Lease object. Each Node has an associated Lease object in the kube-node-lease namespace. Lease is a lightweight resource, which improves the performance of the node heartbeats as the cluster scales.

===========================================================================

kubectl create namespace <namespace-name>

kubectl get namespace

kubectl get pods --namespace=<namespace-name>

to switch the namespace:

kubectl config set-context --current --namespace=dev

to check the current namespace:

kubectl config get-contexts

kubectl delete namespace <namespace-name>

to see all the resources under a namespace:

kubectl get all --namespace=dev

uses:

\* we can create same resources under different namespaces.

\* we cannot create namespace inside a namespace.

\* different services created in different namespaces can communicate with each other by default.

major functionalities of namespace:

\*within the namespace pod to pod communication is open but b/w namespace pod to pod communication will happen only through service.

\*namespaces are only hidden from each other but actually they are not fully isolated from each other because one service in a namespace can talk to another namespace service by using internal k8s DNS(service name)

\*using namespace we can improve role base access controls(RBAC) by limiting users and processes to certain namespace.

\*we can apply resource quotas using namespaces.

When to Use Multiple Namespaces?

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

Small teams or smaller organizations may be perfectly content using the default namespace. This is particularly relevant if there is no need to isolate developers or users from each other. However, there are many useful benefits to having multiple namespaces, including:

Isolation: Large or growing teams can use namespaces to isolate their projects and microservices from each other. Teams can re-use the same resource names in different workspaces without a problem. Also, taking an action on items in one workspace never affects other workspaces.

Organization: Organizations that use a single cluster for development, testing, and production can use namespaces to sandbox dev and test

environments. This ensures production code is not affected by changes that developers or testers make in their own namespaces throughout the application lifecycle.

Permissions: Namespaces enable the use of Kubernetes RBAC, so teams can define roles that group lists of permissions or abilities under a

single name: This can ensure that only authorized users have access to resources in a given namespace.

Resource Control: Policy-driven resource limits can be set on namespaces by defining resource quotas for CPU or memory utilization. This can ensure that every project or namespace has the resources it needs to run, and that no one namespace is hogging all available resources.

Performance: Using namespaces can help improve performance of a given cluster. If a cluster is separated into multiple namespaces for different projects, the Kubernetes API will have fewer items to search when performing operations. This can reduce latency and speed overall application performance for each application running on the cluster.

===========================================================================

https://medium.com/platformer-blog/building-a-kubernetes-1-20-cluster-with-kubeadm-4b745eb5c697

Kubernetes Ingress vs LoadBalancer vs NodePort?

difference b/w kubectl create and kubectl apply?

Ingress:

An API object that manages external access to the services in a cluster, typically HTTP.

Ingress may provide load balancing, SSL termination and name-based virtual hosting.

Ingress Controllers:

In order for the Ingress resource to work, the cluster must have an ingress controller running.

Unlike other types of controllers which run as part of the kube-controller-manager binary, Ingress controllers are not started automatically with a cluster.

https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-ingress-guide-nginx-example.html

https://www.nginx.com/resources/glossary/kubernetes-ingress-controller/

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

https://banzaicloud.com/blog/k8s-ingress/

========================================================================

how to schedule pods to specific node:

kubectl label nodes <your-node-name> <label\_key:label\_value>

kubectl label nodes <your-node-name> disktype=ssd

kubectl get nodes --show-labels

1.Create a pod that gets scheduled to your chosen node

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

env: test

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

nodeSelector:

disktype: ssd

2.Create a pod that gets scheduled to specific node

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

nodeName: foo-node # schedule pod to specific node

containers:

- name: nginx

image: nginx

imagePullPolicy: IfNotPresent

==================================================================================

Statefull and Stateless applications:

The key difference between stateful and stateless applications is that stateless applications donâ€™t â€œstoreâ€ data whereas stateful applications require backing storage.

difference between Statefull and Stateless applications?

headless service:

A headless service is a service with a service IP but instead of load-balancing it will return the IPs of our associated Pods. This allows us to interact directly with the Pods instead of a proxy. It's as simple as specifying None for .spec.clusterIP and can be utilized with or without selectors - you'll see an example with selectors in a moment.

or

Headless service is the same as default ClusterIP service, but lacks the loadbalancing or proxying. Allowing you to connect to pod directly.

What is a Headless Service?

When there is no need of load balancing or single-service IP addresses.We create a headless service which is used for creating a service grouping. That does not allocate an IP address or forward traffic.So you can do this by explicitly setting ClusterIP to â€œNoneâ€ in the mainfest file, which means no cluster IP is allocated.

For example, if you host MongoDB on a single pod. And you need a service definition on top of it for taking care of the pod restart.And also for acquiring a new IP address. But you donâ€™t want any load balancing or routing. You just need the service to patch the request to the back-end pod. So then you use Headless Service since it does not have an IP.

Kubernetes allows clients to discover pod IPs through DNS lookups. Usually, when you perform a DNS lookup for a service, the DNS server returns a single IP which is the serviceâ€™s cluster IP. But if you donâ€™t need the cluster IP for your service, you can set ClusterIP to None , then the DNS server will return the individual pod IPs instead of the service IP.Then client can connect to any of them.

apiVersion: v1

kind: Service

metadata:

name: my-headless-service

spec:

clusterIP: None # <--

selector:

app: test-app

ports:

- protocol: TCP

port: 80

targetPort: 3000

eg:

\*Create a deployment with five pods.

apiVersion: apps/v1

kind: Deployment

metadata:

name: api-deployment

labels:

app: api

spec:

replicas: 5

selector:

matchLabels:

app: api

template:

metadata:

labels:

app: api

spec:

containers:

- name: api

image: eddiehale/hellonodeapi

ports:

- containerPort: 3000

\*Create a regular service:

apiVersion: v1

kind: Service

metadata:

name: normal-service

spec:

selector:

app: api

ports:

- protocol: TCP

port: 80

targetPort: 3000

\*And a headless service:

apiVersion: v1

kind: Service

metadata:

name: headless-service

spec:

clusterIP: None # <-- Don't forget!!

selector:

app: api

ports:

- protocol: TCP

port: 80

targetPort: 3000

\*Apply the yaml and verify everything deployed correctly:

kubectl apply -f deployment.yaml

kubectl get all

\*deploy a Pod and execute a few commands to test.

kubectl run --rm utils --stdin --tty --image eddiehale/utils /bin/bash

once inside the pod do the below:

run nslookup on each service to see what DNS entries exist. If we nslookup normal-service one DNS entry and IP is returned, where nslookup headless-service returns the list of associated Pod IPs with the service DNS:

nslookup normal-service

nslookup headless-service

=====================================================================================

Affinity and Anti-affinity:

1. pod affinity/anti-affinity

2. node affinity/anti-affinity

pod affinity:

pod affinity and anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled based on labels on pods that are already running on the node rather than based on labels on nodes.

The legal operators for pod affinity and anti-affinity are In, NotIn, Exists, DoesNotExist.

apiVersion: v1

kind: Pod

metadata:

name: with-pod-affinity

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: security

operator: In

values:

- S1

topologyKey: topology.kubernetes.io/zone

podAntiAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 100

podAffinityTerm:

labelSelector:

matchExpressions:

- key: security

operator: In

values:

- S2

topologyKey: topology.kubernetes.io/zone

containers:

- name: with-pod-affinity

image: k8s.gcr.io/pause:2.0

============================================

the pod affinity rule indicates that the pod can schedule onto a node only if that node has at least one already-running pod with a label that has the key security and value S1.

apiVersion: v1

kind: Pod

metadata:

name: with-pod-affinity

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: security

operator: In

values:

- S1

topologyKey: failure-domain.beta.kubernetes.io/zone

containers:

- name: with-pod-affinity

image: docker.io/ocpqe/hello-pod

podAntiAffinity:

The pod anti-affinity rule says that the pod prefers to not schedule onto a node if that node is already running a pod with label having key security and value S2.

apiVersion: v1

kind: Pod

metadata:

name: with-pod-antiaffinity

spec:

affinity:

podAntiAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 100

podAffinityTerm:

labelSelector:

matchExpressions:

- key: security

operator: In

values:

- S2

topologyKey: kubernetes.io/hostname

containers:

- name: with-pod-affinity

image: docker.io/ocpqe/hello-pod

requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that must be met for a pod to be scheduled onto a node (similar to nodeSelector but using a more expressive syntax), while the latter specifies preferences that the scheduler will try to enforce but will not guarantee.

==================================================================================

node affinity:

node affinity allows us to create pods in specific nodes based on the node properties and with different usecases:

1. creating a pod in particular node based on the node AZ(availablity zone)

2. creating a pod in a node with specific CPU, RAM, memory and disktypes.

This will ensure that the Pod is scheduled in the node that has the label disktype=ssd.

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: disktype

operator: In

values:

- ssd

\*The new node affinity syntax supports the following operators: In, NotIn, Exists, DoesNotExist, Gt, Lt. You can use NotIn and DoesNotExist to achieve node anti-affinity behavior.

https://medium.com/the-programmer/working-with-node-affinity-in-kubernetes-40bc79d16f2f

=======================================================================================

Taints and Tolerations

Node affinity, is a property of Pods that attracts them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite -- they allow a node to repel a set of pods.

Tolerations are applied to pods, and allow (but do not require) the pods to schedule onto nodes with matching taints.

Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints.

to taint a node:

kubectl taint nodes <node-name> special=true:NoSchedule

The taint has the format <taintKey>=<taintValue>:<taintEffect>.

supported taint effects are:

NoSchedule , NoExecute , and PreferNoSchedule.

to remove the taint from node:

kubectl taint nodes <node-name> special:NoSchedule-

kubectl taint nodes ip-172-31-29-231 value1:NoSchedule-

Tolerations:

apiVersion: v1

kind: Pod

metadata:

name: pod-1

labels:

security: s1

spec:

containers:

- name: bear

image: supergiantkir/animals:bear

tolerations:

- key: "special"

operator: "Equal"

value: "true"

effect: "NoSchedule"

types of operators in tolerations:

1. Equal

tolerations:

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoSchedule"

2. Exists

tolerations:

- key: "key1"

operator: "Exists"

effect: "NoSchedule"

types of effects:

1. NoSchedule :

This means that no pod will be able to schedule onto node unless it has a matching toleration.

2. NoExecute :

f a taint with effect NoExecute is added to a node, then any pods that do not tolerate the taint will be evicted immediately, and pods that do tolerate the taint will never be evicted.

3. PreferNoSchedule:

This is a "preference" or "soft" version of NoSchedule -- the system will try to avoid placing a pod that does not tolerate the taint on the node, but it is not required.

https://www.densify.com/kubernetes-autoscaling/kubernetes-taints

================================================================

Storage Classes:

A StorageClass provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators. Kubernetes itself is unopinionated about what classes represent. This concept is sometimes called "profiles" in other storage systems.

volumes:

persistent volumes

\* A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.

\* Persistent Volumes are not bound to any namespaces.

\* Persistent Volumes are cluster level resources just like nodes.

Lifecycle of a volume and claim:

1. Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

\*Static

A cluster administrator creates a number of PVs.

\*Dynamic

When none of the static PVs the administrator created match a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC.

2. Binding

A user creates, or in the case of dynamic provisioning, has already created, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes.

3. Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a Pod.

The access modes are:

ReadWriteOnce -- the volume can be mounted as read-write by a single node

ReadOnlyMany -- the volume can be mounted read-only by many nodes

ReadWriteMany -- the volume can be mounted as read-write by many nodes

In the CLI, the access modes are abbreviated to:

RWO - ReadWriteOnce

ROX - ReadOnlyMany

RWX - ReadWriteMany

create one ebs volume:

https://www.nebulaworks.com/insights/posts/leveraging-aws-ebs-for-kubernetes-persistent-volumes/

example:

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-static

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteOnce

storageClassName: gp2

awsElasticBlockStore:

fsType: ext4

volumeID: vol-0fb7c2b5bc3f6ed39

capacity: storage size

accessModes: access type, here it is the ReadWriteOnce, which means that this PV can be attached to an only one WorkerNode at the same time

storageClassName: storage access, see below

awsElasticBlockStore: used device type

fsType: a filesystem type to be created on this volume

volumeID: an AWS EBS disc ID

\*Both PVC and PV must have the same class, otherwise, a PVC will not find a PV, and STATUS of such a PVC will be Pending.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc-static

spec:

accessModes:

- ReadWriteOnce

storageClassName: gp2

resources:

requests:

storage: 3Gi

volumeName: pv-static

deployment with volumes:

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

volumeMounts:

- mountPath: /data

name: pv-static

volumes:

- name: pv-static

persistentVolumeClaim:

claimName: pvc-static

difference b/w PV and PVC?

using local storage as PV:

PV:

apiVersion: v1

kind: PersistentVolume

metadata:

name: example-pv

spec:

capacity:

storage: 2Gi

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Delete

storageClassName: local-storage

local:

path: /home/ubuntu/demopvc

============================================

apiVersion: v1

kind: PersistentVolume

metadata:

name: example-pv

spec:

capacity:

storage: 100Gi

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Delete

storageClassName: local-storage

local:

path: /home/ubuntu/demopvc

nodeAffinity:

required:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/hostname

operator: In

values:

- example-node

PVC:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: example-pvc

labels:

app: nginx

spec:

accessModes:

- ReadWriteOnce

storageClassName: local-storage

resources:

requests:

storage: 1Gi

volumeName: example-pv

deployment

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

volumeMounts:

- mountPath: /home/ubuntu/demopvc

name: example-pv

volumes:

- name: example-pv

persistentVolumeClaim:

claimName: example-pvc

What is PV reclaim policy?

PersistentVolumes can have various reclaim policies, including "Retain", "Recycle", and "Delete". For dynamically provisioned PersistentVolumes, the default reclaim policy is "Delete". This means that a dynamically provisioned volume is automatically deleted when a user deletes the corresponding PersistentVolumeClaim

===============================================================================

what are your day to day activities as devops engineer/ explain your roles and responsibilities in your previous organisation/project?

roles and responsibilities:

\* Understanding customer requirements and start the implementation.

\* Implementing various development, testing, automation tools, and IT infrastructure.

\* Planning the team structure, activities, and involvement in project management activities.

\* Setting up tools and required infrastructure

\* Defining and setting development, test, release, update, and support processes for DevOps operation

\* Have the technical skill to review, verify, and validate the software code developed in the project.

\* Troubleshooting techniques and fixing the code bugs

\* Monitoring the processes during the entire lifecycle for its adherence and updating or creating new processes for improvement and minimizing the wastage

\* Encouraging and building automated processes wherever possible

\* Incidence management and root cause analysis.

\* Coordination and communication within the team and with customers

\* Selecting and deploying appropriate CI/CD tools

\* Strive for continuous improvement and build continuous integration, continuous development, and constant deployment pipeline (CI/CD Pipeline)

\* Mentoring and guiding the team members

\* Managing periodic reporting on the progress to the management and the customer

day to day activities:

\* I'll start my day by checking the emails, as soon as I logged into my system.

\* we'll be having a daily standup meeting to discuss about issues/progress.

\* check if there are any priority tickets/issues that has to be taken care as soon as possible. then start working on the same.

\* we'll get issues from github like merge conflicts, user permission issues/access issues(user is unable to commit the files) etc. and will resolve them accordingly.

\* in jenkins we'll get issues like enviromnet issues, build failure issues(mainly compilation errors). we debug to know the type of issue and will resolve them.

\* we'll be monitoring the automation continuously(k8s cluster) for any errors like pods will not be scheduled properly to the nodes, pods getting terminated automcatically, pods will be in pending status.

\* we'll work on automation like writing scripts, setting up CI/CD.

\* we'll handle the deployments for QA, UAT and production evironments.

\* if there are any deployment issues(build is not getting deployed in respective server) we'll resolve them.

\* we'll do automatic rollouts and rollbacks.

=================================================================================

https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

kubernetes dashboard:

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.2.0/aio/deploy/recommended.yaml

kubectl get svc -n kubernetes-dashboard

change the service type to NodePort make dashboard accessible through internet.

kubectl edit svc <service\_name>

Creating sample user:

Creating a Service Account:

We are creating Service Account with name admin-user in namespace kubernetes-dashboard first.

apiVersion: v1

kind: ServiceAccount

metadata:

name: admin-user

namespace: kubernetes-dashboard

Creating a ClusterRoleBinding:

In most cases after provisioning cluster using kops, kubeadm or any other popular tool, the ClusterRole cluster-admin already exists in the cluster. We can use it and create only ClusterRoleBinding for our ServiceAccount. If it does not exist then you need to create this role first and grant required privileges manually.

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: admin-user

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-admin

subjects:

- kind: ServiceAccount

name: admin-user

namespace: kubernetes-dashboard

Getting a Bearer Token

kubectl -n kubernetes-dashboard get secret $(kubectl -n kubernetes-dashboard get sa/admin-user -o jsonpath="{.secrets[0].name}") -o go-template="{{.data.token | base64decode}}"

Now copy the token and paste it into Enter token field on the login screen.

=======================================================================================================

=======================================================================================================

Helm charts:

manifests:

Kubernetes manifests are used to create, modify and delete Kubernetes resources such as pods, deployments, services or ingresses. It is very common to define manifests in form of .yaml files and send them to the Kubernetes API Server via commands such as kubectl apply -f my-file. yaml or kubectl delete -f my-file.

Helm is a Package Manager for Kubernetes. However, itâ€™s not just the Package Manager, it is also a Deployment Management for Kubernetes. In simpler terms, instead of having to define various Kubernetes resources to deploy an application, with Helm you just type a few commands in the terminal and the application configuration will be setup.

Achitecture:

It comes with client and server architecture:

Helm Client: Provides the developer to use it a command-line interface (CLI) to work with Charts, Config, Release, Repositories. Helm Client will interact with Tiller Server, to perform various actions such as install, upgrade and rollback with Charts, Release.

Tiller Server: an in-cluster server in the Kubernetes cluster, interacting with the Helm Client and communicating with the Kubernetes API server. Thus, Helm can easily manage Kubernetes with tasks such as install, upgrade, query and remove for Kubernetes resources.

Helm has 4 basic concepts:

Chart: a collection of YAML files; bundle of the Kubernetes resources needed to build a Kubernetes application.

Config: a configuration in the values.yaml file, which contains configuration explicit to a release of Kubernetes application. It can be the config for service, ingress, deployment, etc.

Release: a chart instance is loaded into Kubernetes. It can be viewed as a version of the Kubernetes application running based on Chart and associated with a specific Config.(running instance of a chart)

Repositories: a repository of published Charts. These can be private repositories that are only used within the company or public through the Helm Hub.

Helm helps in three key ways:

Improves productivity

Reduces the complexity of deployments of microservices

Enables the adaptation of cloud native applications

Why use Helm?

Writing and maintaining Kubernetes YAML manifests for all the required Kubernetes objects can be a time consuming and tedious task. For the simplest of deployments, you would need at least 3 YAML manifests with duplicated and hardcoded values. Helm simplifies this process and creates a single package that can be advertised to your cluster.

What are Helm charts?

Helm Charts are simply Kubernetes YAML manifests combined into a single package that can be advertised to your Kubernetes clusters. Once packaged, installing a Helm Chart into your cluster is as easy as running a single helm install, which really simplifies the deployment of containerized applications.

Describing a Helm chart

\* .helmignore: This holds all the files to ignore when packaging the chart. Similar to .gitignore, if you are familiar with git.

\* Chart.yaml: This is where you put all the information about the chart you are packaging. So, for example, your version number, etc. This is where you will put all those details.

\* Values.yaml: This is where you define all the values you want to inject into your templates. If you are familiar with terraform, think of this as helms variable.tf file.

\* Charts: This is where you store other charts that your chart depends on. You might be calling another chart that your chart need to function properly.

\* Templates: This folder is where you put the actual manifest you are deploying with the chart. For example you might be deploying an nginx deployment that needs a service, configmap and secrets. You will have your deployment.yaml, service.yaml, config.yaml and secrets.yaml all in the template dir. They will all get their values from values.yaml from above.

https://helm.sh/docs/intro/install/

first step, we need to tell Helm what location to search by adding a Helm repository:

helm repo add bitnami https://charts.bitnami.com/bitnami

install the actual container:

helm install my-apache bitnami/apache --version 8.0.2

kubectl get pods

helm list

helm upgrade my-apache bitnami/apache --version 8.0.3

helm rollback my-apache 1

helm delete my-apache

create a new Helm Chart from scratch:

helm create mychart

helm lint ./mychart/

To see the values rendred in yaml files before installing:

helm install --dry-run --debug ./mychart/ --generate-name

helm install example ./mychart/ --set service.type=NodePort

kubectl get all

helm uninstall <chart\_name>

https://ibm.github.io/helm101/Lab1/

===========================================================================================

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your container images, so that your applications are easily portable.

Note: ConfigMap does not provide secrecy or encryption. If the data you want to store are confidential, use a Secret rather than a ConfigMap, or use additional (third party) tools to keep your data private.

configmaps and secrets are the namespace level objects.

configmaps are used to store the data in plain text which is open and readable by all the users who have access to cluster.

kubectl create configmap my-config --from-literal DB\_NAME=mongoDb

apiVersion: v1

kind: ConfigMap

metadata:

name: game-demo

data:

# property-like keys; each key maps to a simple value

player\_initial\_lives: "3"

ui\_properties\_file\_name: "user-interface.properties"

# file-like keys

game.properties: |

enemy.types=aliens,monsters

player.maximum-lives=5

user-interface.properties: |

color.good=purple

color.bad=yellow

allow.textmode=true

Here's an example Pod that uses values from game-demo to configure a Pod:

apiVersion: v1

kind: Pod

metadata:

name: configmap-demo-pod

spec:

containers:

- name: demo

image: alpine

command: ["sleep", "3600"]

env:

# Define the environment variable

- name: PLAYER\_INITIAL\_LIVES # Notice that the case is different here

# from the key name in the ConfigMap.

valueFrom:

configMapKeyRef:

name: game-demo # The ConfigMap this value comes from.

key: player\_initial\_lives # The key to fetch.

- name: UI\_PROPERTIES\_FILE\_NAME

valueFrom:

configMapKeyRef:

name: game-demo

key: ui\_properties\_file\_name

volumeMounts:

- name: config

mountPath: "/config"

readOnly: true

volumes:

# You set volumes at the Pod level, then mount them into containers inside that Pod

- name: config

configMap:

# Provide the name of the ConfigMap you want to mount.

name: game-demo

# An array of keys from the ConfigMap to create as files

items:

- key: "game.properties"

path: "game.properties"

- key: "user-interface.properties"

path: "user-interface.properties"

This is an example of a Pod that mounts a ConfigMap in a volume:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: redis

volumeMounts:

- name: foo

mountPath: "/etc/foo"

readOnly: true

volumes:

- name: foo

configMap:

name: myconfigmap

==============================================

Secrets:

Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys.which will be encrypted in base64. Storing confidential information in a Secret is safer and more flexible than putting it in a Pod definition or in a container image.

secrets can be created through cli or yml files.

secrets are always stored in tmpfs of nodes.

secrets size cannot be more than 1MB.

secrets can be mounted as data volumes or env variables in a pod.

To create base64 values

echo -n "value" | base64

kubectl create secret generic db-secret --from-literal DB\_USER=admin --from-literal DB\_PASS=password

to see the yaml file of the objects:

kubectl get <object\_type> <object\_name> -o yaml

to edit the yaml files of the objects:

kubectl edit <object\_type> <object\_name> -n <namespace\_name>

apiVersion: v1

kind: Secret

metadata:

name: secret-sa-sample

annotations:

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

type: kubernetes.io/service-account-token

data:

# You can include additional key value pairs as you do with Opaque Secrets

extra: YmFyCg==

or

apiVersion: v1

kind: Secret

metadata:

name: my-secret-2

type: Opaque

data:

username: YWRtaW4=

password: cGFzc3dvcmQ=

This is an example of a Pod that mounts a Secret in a volume:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mypod

image: redis

volumeMounts:

- name: foo

mountPath: "/etc/foo"

readOnly: true

volumes:

- name: foo

secret:

secretName: mysecret

diff b/w configmap and a secret in k8s?

=========================================================================

Helm V2 and V3 differences?

Why should we always maintain odd number of master nodes in a cluster?

Having multiple master nodes ensures that services remain available should master node(s) fail. In order to facilitate availability of master services, they should be deployed with odd numbers (e.g. 3,5,7,9 etc.) so quorum (master node majority) can be maintained should one or more masters fail.

or

Because if you have an even number of servers, it's a lot easier to end up in a situation where the network breaks and you have exactly 50% on each side. With an odd number, you can't (easily) have a situation where more than one partition in the network thinks it has majority control.

quorum(master node majority):

For a cluster with n members, quorum is calculated as (n/2)+1 (one greater than 50%).

quorum formula: N/2 + 1

where N is the sum of replication factors in each data center.

So to make your Kubernetes cluster tolerant to partial master failure, you will need master running on at least three nodes. In a 3 master node cluster, quorum will be 2, so it can still function if one of the nodes goes down(same applies for etcd also).

https://platform9.com/blog/support/create-multi-master-highly-available-kubernetes-clusters/

Deployment strategies:

recreate: terminate the old version and release the new one

ramped: release a new version on a rolling update fashion, one after the other

blue/green: release a new version alongside the old version then switch traffic

canary: release a new version to a subset of users, then proceed to a full rollout

a/b testing: release a new version to a subset of users in a precise way (HTTP headers, cookie, weight, etc.). This doesnâ€™t come out of the box with Kubernetes, it imply extra work to setup a smarter loadbalancing system (Istio, Linkerd, Traeffik, custom nginx/haproxy, etc).

shadow: release a new version alongside the old version. Incoming traffic is mirrored to the new version and doesn't impact the response.

https://www.weave.works/blog/kubernetes-deployment-strategies

Bastion server:

Bastion hosts are instances that sit within your public subnet and are typically accessed using SSH (for Linux) or RDP (for Windows). It acts as a â€˜jumpâ€™ server, allowing you to use SSH or RDP to login to other instance in private subnet.

Extra Info:

A jump server is a virtual machine that is used to manage other systems. It is sometimes called a â€œpivot serverâ€ for this reason: once you are logged in, you can â€œpivotâ€ to the other servers. It is usually security hardened and treated as the single entryway to a server group from within your security zone, or inside the overall network. A jump server is a â€œbridgeâ€ between two trusted networks. The two security zones are dissimilar but both are controlled.

StatefulSets:

StatefulSets will represent the set of pods with different (unique), persistent identities, and elastic hostnames (stable). It makes you assure about the ordering of scaling and deployments

StatefulSets are valuable for applications that require one or more of the following:

\*Stable, unique network identifiers

\*Stable, persistent storage

\*Ordered, graceful deployment and scaling

\*Ordered, graceful deletion and termination

If an application doesnâ€™t require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application with a controller such as Deployments or ReplicaSets that provides a set of stateless replicas.

StatefulSet Components

1. A Headless Service

2. A StatefulSet

3. A PersistentVolume

https://medium.com/avmconsulting-blog/deploying-statefulsets-in-kubernetes-k8s-5924e701d327

create a stateful set:

1.

---

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

2.

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx

serviceName: "nginx"

replicas: 3

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumes:

- name: www

persistentVolumeClaim:

claimName: myclaim

3.

---

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: myclaim

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 8Gi

kubectl create -f statefulset.yaml

It will create three Pods named web-0,web-1,web-2

kubectl get statefulsets

Scale-up a stateful set:

kubectl scale statefulset web --replicas=5

https://medium.com/avmconsulting-blog/deploying-statefulsets-in-kubernetes-k8s-5924e701d327

Example 2:

The following is an example of a Service and StatefulSet manifest file:

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx # Label selector that determines which Pods belong to the StatefulSet

# Must match spec: template: metadata: labels

serviceName: "nginx"

replicas: 3

template:

metadata:

labels:

app: nginx # Pod template's label selector

spec:

terminationGracePeriodSeconds: 10

containers:

- name: nginx

image: k8s.gcr.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi

In this example:

\*A Service object named nginx is created, indicated by the metadata: name field. The Service targets an app called nginx, indicated by labels: app: nginx and selector: app: nginx. The Service exposes port 80 and names it web. This Service controls the network domain and to route Internet traffic to the containerized application deployed by the StatefulSet.

\*A StatefulSet named web is created with three replicated Pods (replicas: 3).

\*The Pod template (spec: template) indicates that its Pods are labelled app: nginx.

\*The Pod specification (template: spec) indicates that the StatefulSet's Pods run one container, nginx, which runs the nginx-slim image at version 0.8. The container image is hosted by Container Registry.

\*The Pod specification uses the web port opened by the Service.

\*template: spec: volumeMounts specifies a mountPath, which is named www. The mountPath is the path within the container at which a storage volume should be mounted.

\*The StatefulSet provisions a PersistentVolumeClaim, www, with 1GB of provisioned storage.

In sum, the Pod specification contains the following instructions:

\*Label each Pod as app: nginx.

\*In each Pod, run one container named nginx.

\*Run the nginx-slim image at version 0.8.

\*Have Pods use port 80.

\*Save data to the mount path.

https://blog.container-solutions.com/kubernetes-deployment-strategies

https://github.com/ContainerSolutions/k8s-deployment-strategies

https://medium.com/velotio-perspectives/demystifying-high-availability-in-kubernetes-using-kubeadm-3d83ed8c458b#:~:text=Steps%20to%20achieve%20Kubernetes%20HA&text=Master%20Node%3A%20Each%20master%20node,all%20the%20data%20of%20cluster.&text=In%20this%20blog%2C%20we%20will%20use%20k8s%201.10

https://blog.container-solutions.com/kubernetes-deployment-strategies

difference b/w deployment and statefulset in k8s.

=========================================================================================

Resourcequota:

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. It can limit the quantity of objects that can be created in a namespace by type, as well as the total amount of compute resources that may be consumed by resources in that namespace.

In simple words:

ResourceQuotas are used to limit resources like cpu,memory, storage, and services.

https://www.eksworkshop.com/intermediate/201\_resource\_management/resource-quota/

create a namespace for resource for resource quota testing:

kubectl create namespace <name-space-for-quota-demo>

kubectl create namespace resource-quota-demo

below will restrict the number of pods and configmaps to be created by specified count.

apiVersion: v1

kind: ResourceQuota

metadata:

name: count-quota-1

namespace: resource-quota-demo

spec:

hard:

pods: 2

configmaps: 1

or

apiVersion: v1

kind: ResourceQuota

metadata:

name: count-quota-1

namespace: resource-quota-demo

spec:

hard:

count/pods: 2

count/configmaps: 1

kubectl -n resource-quota-demo create configmap cm1 --from-literal=name=praveen

kubectl -n resource-quota-demo get configmap

kubectl -n resource-quota-demo describe quota count-quota-1

you will get error if you are trying to create more resources than allocated in quota

kubectl -n resource-quota-demo create configmap cm2 --from-literal=name=praveen

kubectl -n resource-quota-demo run nginx --image=nginx --replicas=1

kubectl -n resource-quota-demo scale deployment nginx --replicas=2

to limit the resources by memory usage:

apiVersion: v1

kind: ResourceQuota

metadata:

name: demo-mem-quota

namespace: resource-quota-demo

spec:

hard:

limits.memory: "500Mi"

----------------------------------

to limit the resources by cpu usage:

apiVersion: v1

kind: LimitRange

metadata:

name: mem-limitrange

namespace: resource-quota-demo

spec:

limits:

- default:

cpu: 0.5

defaultRequest:

cpu: 0.25

type: Container

--------------------------------------------

pod example with memory limits:

you will get error if you try to create a pod without resource limits:

apiVersion: v1

kind: Pod

metadata:

name: static-web

namespace: resource-quota-demo

spec:

containers:

- name: web

image: nginx

ports:

- name: web

containerPort: 80

protocol: TCP

resources:

limits:

memory: "200Mi"

requests:

memory: "50Mi"

----------------------------------------

A LimitRange is a policy to constrain resource allocations (to Pods or Containers) in a namespace. A LimitRange provides constraints that can: Enforce minimum and maximum compute resources usage per Pod or Container in a namespace. Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace

apiVersion: v1

kind: LimitRange

metadata:

name: mem-limitrange

namespace: resource-quota-demo

spec:

limits:

- default:

cpu: 0.5

defaultRequest:

cpu: 0.25

type: Container

-----------------------------------------------

soft limit requests.memory is been added in order to request the memory within the limits.memory.

requests.memory should not exceed limits.memory.

we use requests.memory to limit memory for created per object within the limits.memory.

apiVersion: v1

kind: ResourceQuota

metadata:

name: demo-mem-quota

namespace: resource-quota-demo

spec:

hard:

limits.memory: "500Mi"

requests.memory: "100Mi"

======================================================================================================

======================================================================================================

Kubernetes configuration file

Kubernetes uses a YAML file called kubeconfig to store cluster authentication information for kubectl. kubeconfig contains a list of contexts to which kubectl refers when running commands. By default, the file is saved at $HOME/.kube/config.

A context is a group of access parameters. Each context contains a Kubernetes cluster, a user, and a namespace. The current context is the cluster that is currently the default for kubectl: all kubectl commands run against that cluster.

kubectl config current-context

kubectl config view

Types of users in Kubernetes:

1. User Account:

- kubernetes-admin is the default admin user who set up the cluster.

- we can create users, groups by which we can connect to API server from kubectl using different user context.

- this is basically us, humans (admin, devops, developer, client)

2. service Account:

A service account provides an identity for processes that run in a Pod.

When you (a human) access the cluster (for example, using kubectl), you are authenticated by the apiserver as a particular User Account (currently this is usually admin, unless your cluster administrator has customized your cluster). Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (for example, default).

\* A service account is also used by applications running outside the cluster to interact with the resources of kubernestes cluster.

\* "default" is the default service account of the cluster.

Ex: To give access to prometheus monitoring tool to the cluster resources we need to create a service account.

when a service account is created, k8s first will create a secret token object then it creates the service account and attach both of them.

kubectl create serviceaccount <account-name>

or

apiVersion: v1

kind: ServiceAccount

metadata:

name: my-sa

automountServiceAccountToken: false

In version 1.6+, you can opt out of automounting API credentials for a service account by setting automountServiceAccountToken: false on the service account:

Configure Service Accounts for Pods

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

serviceAccountName: build-robot

automountServiceAccountToken: false

============================================================================================

Note:

Horizontal scaling means scaling by adding more machines to your pool of resources (also described as â€œscaling outâ€), whereas vertical scaling refers to scaling by adding more power (e.g. CPU, RAM) to an existing machine (also described as â€œscaling upâ€).

RBAC(Role-based access control):

Role-based access control (RBAC) is a method of restricting network access based on the roles of individual users within an enterprise. RBAC lets employees have access rights only to the information they need to do their jobs and prevents them from accessing information that doesn't pertain to them.

The RBAC API declares four kinds of Kubernetes object:

1.Role.

2.ClusterRole.

3.RoleBinding.

4.ClusterRoleBinding.

Role:

A Role always sets permissions within a particular namespace; when you create a Role, you have to specify the namespace it belongs in.

Role example

Here's an example Role in the "default" namespace that can be used to grant read access to pods:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""] # "" indicates the core API group

resources: ["pods"]

verbs: ["get", "watch", "list"]

ClusterRole:

ClusterRole, by contrast, is a non-namespaced resource. The resources have different names (Role and ClusterRole) because a Kubernetes object always has to be either namespaced or not namespaced; it can't be both.

A ClusterRole can be used to grant the same permissions as a Role. Because ClusterRoles are cluster-scoped, you can also use them to grant access to:

\*cluster-scoped resources (like nodes)

\*non-resource endpoints (like /healthz)

\*namespaced resources (like Pods), across all namespaces

example of a ClusterRole that can be used to grant read access to secrets in any particular namespace, or across all namespaces (depending on how it is bound):

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

# "namespace" omitted since ClusterRoles are not namespaced

name: secret-reader

rules:

- apiGroups: [""]

#

# at the HTTP level, the name of the resource for accessing Secret

# objects is "secrets"

resources: ["secrets"]

verbs: ["get", "watch", "list"]

RoleBinding and ClusterRoleBinding:

A role binding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted. A RoleBinding grants permissions within a specific namespace whereas a ClusterRoleBinding grants that access cluster-wide.

A RoleBinding may reference any Role in the same namespace. Alternatively, a RoleBinding can reference a ClusterRole and bind that ClusterRole to the namespace of the RoleBinding. If you want to bind a ClusterRole to all the namespaces in your cluster, you use a ClusterRoleBinding.

example of a RoleBinding that grants the "pod-reader" Role to the user "jane" within the "default" namespace. This allows "jane" to read pods in the "default" namespace.

apiVersion: rbac.authorization.k8s.io/v1

# This role binding allows "jane" to read pods in the "default" namespace.

# You need to already have a Role named "pod-reader" in that namespace.

kind: RoleBinding

metadata:

name: read-pods

namespace: default

subjects:

# You can specify more than one "subject"

- kind: User

name: jane # "name" is case sensitive

apiGroup: rbac.authorization.k8s.io

roleRef:

# "roleRef" specifies the binding to a Role / ClusterRole

kind: Role #this must be Role or ClusterRole

name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to

apiGroup: rbac.authorization.k8s.io

RoleBinding refers to a ClusterRole, "dave" (the subject, case sensitive) will only be able to read Secrets in the "development" namespace, because the RoleBinding's namespace (in its metadata) is "development".

apiVersion: rbac.authorization.k8s.io/v1

# This role binding allows "dave" to read secrets in the "development" namespace.

# You need to already have a ClusterRole named "secret-reader".

kind: RoleBinding

metadata:

name: read-secrets

#

# The namespace of the RoleBinding determines where the permissions are granted.

# This only grants permissions within the "development" namespace.

namespace: development

subjects:

- kind: User

name: dave # Name is case sensitive

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: secret-reader

apiGroup: rbac.authorization.k8s.io

ClusterRoleBinding example:

To grant permissions across a whole cluster, you can use a ClusterRoleBinding. The following ClusterRoleBinding allows any user in the group "manager" to read secrets in any namespace.

apiVersion: rbac.authorization.k8s.io/v1

# This cluster role binding allows anyone in the "manager" group to read secrets in any namespace.

kind: ClusterRoleBinding

metadata:

name: read-secrets-global

subjects:

- kind: Group

name: manager # Name is case sensitive

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: secret-reader

apiGroup: rbac.authorization.k8s.io

\*After you create a binding, you cannot change the Role or ClusterRole that it refers to. If you try to change a binding's roleRef, you get a validation error. If you do want to change the roleRef for a binding, you need to remove the binding object and create a replacement.

In this case, pods is the namespaced resource for Pod resources, and log is a subresource of pods. To represent this in an RBAC role, use a slash (/) to delimit the resource and subresource. To allow a subject to read pods and also access the log subresource for each of those Pods, you write:

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-and-pod-logs-reader

rules:

- apiGroups: [""]

resources: ["pods", "pods/log"]

verbs: ["get", "list"]

https://kubernetes.io/docs/reference/access-authn-authz/rbac/

https://www.adaltas.com/en/2019/08/07/users-rbac-kubernetes/

=========================================================================================================

Adding user in k8s:

Normal user

A few steps are required in order to get a normal user to be able to authenticate and invoke an API. First, this user must have certificate issued by the Kubernetes cluster, and then present that Certificate to the API call as the Certificate Header or through the kubectl.

Create private key

The following scripts show how to generate PKI private key and CSR. It is important to set CN and O attribute of the CSR. CN is the name of the user and O is the group that this user will belong to. You can refer to RBAC for standard groups.

openssl rand -out /home/ubuntu/.rnd -hex 256

openssl req -new -newkey rsa:4096 -nodes -keyout kubeuser -out kubeuser-k8s.csr -subj "/CN=kubeuser/O=manager"

csr.register.yaml

cat kubeuser-k8s.csr | base64 | tr -d "\n"

apiVersion: certificates.k8s.io/v1

kind: CertificateSigningRequest

metadata:

name: kubeuser-access

spec:

groups:

- system:authenticated

request: 

signerName: kubernetes.io/kube-apiserver-client

usages:

- client auth

kubectl get csr

kubectl describe csr kubeuser-access

\*Approve certificate signing request

Use kubectl to create a CSR and approve it.

kubectl certificate approve kubeuser-access

\*Get the certificate

Retrieve the certificate from the CSR:

kubectl get csr/kubeuser-access -o yaml

\*The certificate value is in Base64-encoded format under status.certificate.

Export the issued certificate from the CertificateSigningRequest.

kubectl get csr kubeuser-access -o jsonpath='{.status.certificate}'| base64 -d > kubeuser.crt

\*Create Role and RoleBinding

With the certificate created. it is time to define the Role and RoleBinding for this user to access Kubernetes cluster resources.

This is a sample script to create a Role for this new user:

kubectl create role developer --verb=create --verb=get --verb=list --verb=update --verb=delete --resource=pods --resource=deployments

\*This is a sample command to create a RoleBinding for this new user:

kubectl create rolebinding developer-binding-kubeuser --role=developer --user=kubeuser

\*Add to kubeconfig:

The last step is to add this user into the kubeconfig file.

First, you need to add new credentials:

kubectl config set-credentials kubeuser --client-key=kubeuser --client-certificate=kubeuser.crt --embed-certs=true

\*Then, you need to add the context:

kubectl config set-context developer-binding-kubeuser --cluster=kubernetes --user=kubeuser

\*To test it, change the context to user:

kubectl config use-context developer-binding-kubeuser

========================================================================

RBAC Service account setup:

step 1: Create a service account

kubectl create serviceaccount <account\_name>

step 2: Create a cluster role

kubectl create clusterrole <role\_name> --verb=get --verb=list --verb=watch --resource=pods --resource=deployments

step 3: Create a clusterRoleBinding

kubectl create clusterrolebinding <binding\_name> --serviceaccount='default:<account\_name>' --clusterrole='<role\_name>'

step 4: Get the TOKEN generated for service account

1. (In this step get the secret token name)

kubectl describe serviceaccount <account\_name>

2. (In this step get the token value from the secret)

kubectl describe secret <secret\_token\_name>

3. Copy the token from the secret and assign it to variable called TOKEN

TOKEN="<token\_copied\_from\_previous\_step>"

or

single command to perform all the above operations:

TOKEN="$(kubectl describe secrets "$(kubectl describe serviceaccount sv-readonly | grep Tokens | awk '{print $2}')" | grep token: | awk '{print $2}')"

step 5: set the credentials for the user/serviceaccount in .kube/config file (context)

kubectl config set-credentials <user\_name> --token=$TOKEN

step 6: create the context for the above config

kubectl config set-context <context\_name> --cluster=kubernetes --user=<user\_name>

step 7: finally switch the context to new serviceaccount user

kubectl config use-context <context\_name>

NOTE: To check the authorization use the below command:

kubectl auth can-i <verb> <resource\_type>

eg:

kubectl auth can-i create pods

kubectl auth can-i delete pods

kubectl auth can-i get pods

kubectl auth can-i get deployments

kubectl auth can-i create deployments

=================================================================================================================

https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/

https://www.eksworkshop.com/beginner/130\_exposing-service/ingress/

https://github.com/aws-samples/amazon-eks-cicd-codebuild

---

- name: Install Apache Tomcat10 using ansible

hosts: webserver

remote\_user: ubuntu

become: true

tasks:

- name: Update the System Packages

apt:

upgrade: yes

update\_cache: yes

- name: Create a Tomcat User

user:

name: tomcat

- name: Create a Tomcat Group

group:

name: tomcat

- name: Install JAVA

apt:

name: default-jdk

state: present

- name: Create a Tomcat Directory

file:

path: /opt/tomcat10

owner: tomcat

group: tomcat

mode: 755

recurse: yes

- name: download & unarchive tomcat10

unarchive:

src: https://mirrors.estointernet.in/apache/tomcat/tomcat-10/v10.0.4/bin/apache-tomcat- 10.0.4.tar.gz

dest: /opt/tomcat10

remote\_src: yes

extra\_opts: [--strip-components=1]

- name: Change ownership of tomcat directory

file:

path: /opt/tomcat10

owner: tomcat

group: tomcat

mode: "u+rwx,g+rx,o=rx"

recurse: yes

state: directory

- name: Copy Tomcat service from local to remote

copy:

src: /etc/tomcat.service

dest: /etc/systemd/system/

mode: 0755

- name: Start and Enable Tomcat 10 on sever

systemd:

name: tomcat

state: started

daemon\_reload: true

﻿