

DOCKER

Docker is one of the tools that used the idea of the isolated resources to create a container that allows applications to be packaged with all the dependencies installed and ran wherever we wanted.

Docker can only run-on Linux machines this means I can't install Docker directly on Windows or any other OS.

Container

A container is a set of isolated processes and resources. Linux achieves this by using namespaces, which allows processes to access only resources in that particular namespace, which allows having a process tree means set of processes that is completely independent of the rest of the systems processes.

Virtualization (VM)

- VM is way of running virtual OS on top a host OS using a special software called **Hypervisor**.
- VM directly shares the hardware of the host OS.

Diff B/w Virtualization and Containerisation

VM	vs	Containerisation
1. Virtualization at hardware level		1. Virtualization at OS level
2. Heavyweight - consume more host		2. Lightweight resources
3. VM uses hypervisor		3. containerisation tool is used
4. limited performance - Boot up time is more which being in minutes		4. Native performance - usually boot fast in seconds.
5. Consumes more storage		5. Shares OS storage means only uses required storage.
6. Supports all OS		6. Supports on Linux

Host Machine

This is the machine in which docker is running

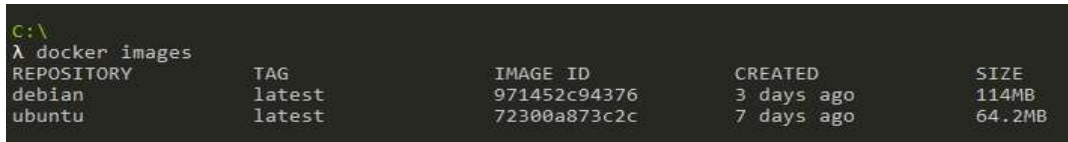
Docker images

Docker Image is a read only template consists of set instructions to create a container that can run on docker platform.

Docker will never save two layers with the same content and it is just used for reference, if there is any same layer between 2 images.

Each instruction in docker file creates a layer & these layers are read only. Once the image is created, we cannot edit it, but we can update by keeping it as base image.

Docker images



```
c:\> docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
debian	latest	971452c94376	3 days ago	114MB
ubuntu	latest	72300a873c2c	7 days ago	64.2MB

docker build -t imagename:tag .

docker build -t image_name -f path .

List images

docker images

To pull / download docker image

docker pull <image_name>:<tag_name>

To delete docker image

docker rmi <image_name>:<tag_name>

(OR)

docker rmi <image_id>

To delete all the images

docker rmi \$(docker images -q)

To inspect images

docker images inspect image name

To tag a docker image

docker tag <old_image> <new_image>

ex: docker tag ubuntu: latest my_ubuntu:1.0

To delete dangling images / to delete all unwanted images / images which are not used

docker image prune

Docker containers

A container is a set of isolated resources. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. They are like virtual machines

To create/run a container from image

-it - Interactive Terminal

-d - detached mode (whenever we create a container it will auto login to avoid this

we can create a container in detached mode

--name used to provide user defined container name

To list the running containers	<code>docker ps</code> (OR) <code>docker container ls</code>
To stop a container	<code>docker stop<container_name></code>
To delete a stop container	<code>docker rm container_name</code> or <code>docker rm <container_id></code>
To delete a running container	Forceful deletion <code>docker rm -f <container_id></code> Graceful deletion <code>docker rm \$(docker stop container id)</code>
To list all the containers (running and stopped)	<code>docker ps -a</code>
To list all stopped containers	<code>docker ps -a --filter status=exited</code>
To delete all stopped container	<code>docker rm \$(docker ps -aq --filter status=exited)</code>
To check the logs of containers	<code>docker logs <container_id></code>
To run a command inside a container/login	<code>docker exec -it <container_id> <command></code>
To remove all stopped/unwanted container	<code>docker container prune</code>
To check cpu usage of container	<code>docker stats</code>
To secure docker images	<code>docker scan</code> , Impec, ECR
To view detailed information of object	<code>docker inspect container id</code>
To inspect on container	<code>docker container inspect container name</code>

The difference between “**docker run**” and “**docker exec**” is that “**docker exec**” executes a command on a running container. On the other hand, “**docker run**” creates a temporary container, executes the command in it and stops the container when it is done.

States of container / Lifecycle of container

Docker ps -a → To know the status of container

1. Created - If container is newly created and container is not yet started.
2. Running - A currently running container. It means there is no problem with container to run the process.
3. Exited - A container ran and completed or execution with failure.
4. Paused - A container whose process has been paused. (We can unpause the container)
5. Dead - If docker daemon tried and failed to stop a container (host ram full)
6. Restarting - Container will be in the phase of restarting the main process.

Custom Docker Image / Docker file

- Dockerfile is a set of instructions used to create custom image on top stock image or any other image as base image.

To build a docker file

docker build -t image_name:tag .

docker build -t image_name -f path .

By default, image created are stored under **/var/lib/docker**

To create a container -> docker run -it -d -rm -name container_name image_name

(here rm will delete the container once container is created or process is stopped, it will delete container)

FROM

- FROM must be the first non-command instruction in the Dockerfile.

- FROM is used to specify base image, on top of this image all the next instructions will be executed.

The FROM instruction initializes a new build stage and sets the *Base Image* for subsequent instructions.

FROM <image_name>:<tag>

RUN

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile

RUN apt update

RUN apt install git

RUN ls

Difference b/w RUN & CMD

"RUN" is used during the build process to make changes to the image, while "CMD" is used to specify the default command that should be executed when a container is started.

ADD

The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

Multiple <src> resources may be specified but if they are files or directories, their paths are interpreted as relative to the source of the context of the build.

COPY and ADD

- Both copy and add instruction is used to copy files and directories from host machine to the image.

- The source path to copy files should always be evaluated with reference to Dockerfile.

ADD supports extra source formats

- If the source is a compressed file add will automatically uncompressed it to the destination.

- If the source is a link to a downloadable file, it will download to the destination.

CMD and ENTRYPOINT

CMD: -

The main purpose of the CMD command is to launch the software required in a container. For example, the **user** may need to run an **executable .exe file or a Bash terminal as soon as the container starts** – the CMD command can be used to handle such requests.

- If we use multiple CMD in the same Docker file only the latest one will be considered and all the other CMD will be ignored.

Entry Point: -

An ENTRYPOINT instruction is used to set executables that will always run when the container is initiated.

- If we use multiple ENTRYPOINT in the same Docker file only the latest one will be considered and all the other ENTRYPOINT will be ignored.

Difference b/w CMD & Entry point

- If we use both CMD and ENTRYPOINT in the same Docker file, Entrypoint will get the highest priority and the command of CMD will become as arguments to Entry point

- CMD command can be overridden at the runtime.

- ENTRYPOINT can't be overridden at the runtime but the runtime command will become parameters to ENTRYPOINT command.

Note: Q. Can we override ENTRYPOINT

Yes, after docker 1.6 version docker has given option to overwrite Entrypoint command at the runtime using `--entrypoint`

Docker run `--it --entrypoint --name container name image name`

ENV

- This instruction is used to set the environment variable inside the container.

The ENV instruction sets the environment variable `<key>` to the value `<value>`. This value will be in the environment for all subsequent instructions in the build stage

ENV `<variable_name> <value>`

Ex:-

```
FROM ubuntu:latest
```

```
ENV MY_NAME John
```

```
RUN echo "Hello, $MY_NAME!"
```

```
CMD ["echo", "My name is $MY_NAME"]
```

To set Multiple variable

```
ENV<variable_name>=<value><variable_name>=<value><variable_name>=<value> ....
```

To create environment variables at run time (temporary)

- using -e or --env option at the runtime we can create env variables

- For multiple variables use multiple -e

ex: **docker run -e <variable_name>=<value> -e <variable_name>=<value>**

The best way to load multiple env variable is using env file

using --env-file <file_path> at the runtime (with docker run command) we can load the env file containing n number variables.

ARG

The ARG instruction defines a variable that users can pass at build-time with docker build

```
ARG <var_name>=<default_value>
```

To pass the value at build time

```
docker build --build-arg <var_name>=<value> ....
```

Difference b/w ARG and ENV

"ARG" defines build-time variables that can be passed to the Docker build command using the "--build-arg" option. **These variables are only available during the build process and are not persisted in the resulting image.** They are typically used for values that are specific to the build process, such as version numbers or build settings.

"ENV" sets environment variables that are available during the build process and when a container is running. **These variables are persisted in the resulting image and can be accessed by any subsequent commands or processes.** They are typically used for values that are needed by applications running inside the container, such as **database credentials or API keys.**

"**ARG**" variable is only available during the build process and is not persisted in the resulting image, while the "**ENV**" variable is available during the build process and when the container is running and is persisted in the resulting image.

WORKDIR

"WORKDIR" is used to set the working directory for any subsequent instructions, such as "RUN" and "CMD". This makes it easier to write and maintain Docker files, as you don't have to specify the full file paths for every instruction.

The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction

WORKDIR <path>

EXPOSE

"EXPOSE" is used to inform Docker that the container will listen on the specified network ports at runtime

- All containers in the same network will have access to exposed port.

EXPOSE <port_number>

Docker Volumes

Docker volumes are a way to persist data in a Docker container. Volumes are directories (or files) that can be mounted to a container, allowing data to be stored outside the container's file system.

Container layer is temporary layer, if we lose the container, we lose data. So, to retain/persist the container runtime data we need docker volumes.

Bind Mounts

- we can mount host machine filesystem (files and directories) to the container

docker run -v <host_path>:<container_path>

- In the case of bind mounts, the first field is the path to the file or directory on the **host machine**.
- The second field is the path where the file or directory is mounted in the container.
- Bind mounts are useful when you want to share files between the host and the container, or when you need to access configuration files or data that is stored on the host machine.

Docker Volumes

- These are docker managed filesystem and we use docker commands to manage these volumes

- Volumes are easier to manage, backup or migrate than bind mounts.

- Volumes supports many drivers which means we can mount many types of filesystems.

Volume can be mounted to multiple container with single volume

- Default location of docker volume is **/var/lib/docker/volumes**

docker run -v <volume_name>:<container_path>

To create volume docker volume create <volume_name>

To list volume docker volume ls

To Delete volume docker volume rm <volume_name>

Uses of Docker Volumes

- **Data persistence:** Docker volumes allow you to store data generated by your containers, such as log files or databases, outside of the container's file system. This means that even if the container is removed or destroyed, the data remains intact.
- **Sharing data between containers:** Docker volumes can be shared between containers, making it easy to share data between services or applications running in different containers.
- **Backup and restore:** With Docker volumes, you can easily backup and restore data. This is useful when you need to migrate your containers to a different host or when you want to roll back to a previous version of your application.
- **Performance:** Using Docker volumes can improve the performance of your containers. Since volumes are stored outside of the container's file system, the container can access the data more quickly.

In summary, bind mounts provide direct access to the host machine's file system, while anonymous volumes are managed by Docker and are not associated with a specific location on the host machine. The choice between bind mounts and anonymous volumes depends on the specific use case and whether you want to share data between containers or with the host machine.

How to Create a Docker Image From a Container

1. Step 1: Create a Base Container. ...
2. Step 2: Inspect Images. ...
3. Step 3: Inspect Containers. ...
4. Step 4: Start the Container. ...
5. Step 5: Modify the Running Container. ...
6. Step 6: Create an Image From a Container. ...
7. Step 7: Tag the Image. ...
8. Step 8: Create Images With Tags.

Namespace

- Docker uses Linux namespaces to provide isolated workspace for processes called container
- when a container is created, docker creates a set of namespaces for it and provides a layer of isolation for container.
- Each container runs in a different namespace

Namespaces provide a way to create a separate instance of a global resource, such as network interfaces, file systems, process IDs, and more, for each container. This way, containers can operate as if they are running on their own isolated system, without interfering with other containers running on the same host.

namespace (To list - lsns)

Cgroups

- Linux OS uses c groups to manage the available hardware resources such as CPU, RAM, Disk .
- we can control the access and, we can apply the limitations

cgroups (control groups) provide a way to limit the resource usage of a container, such as CPU, memory, disk I/O, and more. With cgroups, Docker can restrict the amount of resources that a container can use, ensuring that other containers and the host system are not adversely affected.

To list - lscgroup

pid - namespace for managing processes (process isolation)

user - namespace for non-root user on linux.

uts - namespace for unix timesharing system which isolates kernel and version identifiers,

ipc - (interprocess communication) namespace for managing the process communication.

mnt - namespace for managing filesystem mounts.

net - namespace for managing network interfaces.

Docker networking

Publish

PUBLISH = Expose + outside world port mapping

- public will bind the container port to the host port which we can access from outside world using <host_ip>:<port_mapped>

- To publish a port docker run -p <host_port>:<container_port>

-P publish_all It binds all the exposed ports of the container to host machine port

To map direct IP address to the host

port to port

ip:<host_port>:<container_port>

ip::<container_port>

Range of ports

many to one **-p 8080-8090:8080**

many to many **-p 8080-8085:8086-8091**

- The total number of host ports in range should be same as the total number of container ports range.

Docker network types

1. Bridge

- This is a private internal network created by docker on the host machine by name docker0
- This is the default network type for all the container which are created without any network configurations.
- By default, all the containers in the same bridge can communicate with each other without any extra configuration.
- We cannot use container name for communication only IP address is allowed in default bridge.

Custom bridge

To create bridge network docker network create --driver bridge my_bridge

- In custom bridge containers can communicate with each other with container name and also with IP address.

2. Host (Similar to port Mapping)

- This driver removes the network isolation between docker and the host.
- The containers are directly connected to host machine network without extra layer of any docker network.
- Shares the same TCP/IP stack and same namespace of host machine.
- All the network interfaces which are there in host machine are accessible by this container.

3. None

- Containers are not attached to any network by docker.
- All the required network configurations need to be done manually.
- The host or any other containers won't be able to communicate with this container until a custom network is configured.

Docker Architecture

Docker Daemon

The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to docker daemon, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker REST API

Docker provides an API for interacting with the Docker daemon by using http client

Docker CLI

The CLI uses Docker APIs to control or interact with the Docker daemon through scripting or direct CLI commands. Many other Docker applications use the underlying API and CLI

Docker Objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

Multistage build

- Multistage build allows us to define multiple FROM in same Dockerfile.
- Dependency between multiple FROM is maintained by naming FROM using AS keyword and we can refer this name in another FROM.

```
FROM <base_image> AS <STAGE_NAME>
```

- Only the final FROM image is created leaving back all the other FROM.
- Copy only the required files from the named FROM stage like below.

```
FROM final_build
```

```
COPY --from <STAGE_NAME> <src_named_stage> <dest>
```

2. Always try to use the slim / alpine / stretch version of base image instead of using the fully loaded base image.

Advantages

- Make intermediate image layers shareable
- Keeps final images small and docker file readable
- Simplifies the images stages instantly

example: https://github.com/jaintpharsha/docker_multi_build.git

```
FROM node:10 AS ui-build
```

```
WORKDIR /usr/src/app
```

```
COPY WebApp/ ./WebApp/
```

```
RUN cd WebApp && npm install @angular/cli && npm install && npm run build
```

```
FROM node:slim AS server-build
```

```
WORKDIR /root/
```

```
COPY --from=ui-build /usr/src/app/WebApp/dist ./WebApp/dist
```

```
COPY package*.json ./
```

```
RUN npm install
```

```
COPY index.js .
```

```
EXPOSE 3070
```

```
ENTRYPOINT ["node"]
```

```
CMD ["index.js"]
```

Docker-compose

Docker-compose is a tool for defining and running multiple container docker application with a single command.

- We use YAML file to do docker related configurations then with a single command we can execute this YAML file to create docker objects defined in this file.

Can start and stop all services with single command

Can scale up selected services when required`

```

docker-compose.yml
  version: "3.8"
  services:
    jenkins:
      image: jenkins/jenkins:lts
      container_name: dc-jenkins
      ports:
        - "8080:8080"
        - "5000:5000"
      networks:
        - my_brid
    alpine:
      build: .

```

* How multistage build works?

Within a docker file we can use multiple FROM statements, in an each FROM statement we can use different base image and each FROM specifies new stage to build.

We can selectively copy the artifacts from one stage to another stage leaving behind that we don't want in the final stage.

- we use --copy - from to grab the required artifact from the previous FROM

we use. As parameter alias to get the dependency from one stage to another Stage

After all these stages the final / latest from will be executed to build an image

How to reduce the size of the docker image or container?

- **Use a smaller base image:** Start with a smaller base image, like Alpine Linux or BusyBox, instead of a full-fledged Linux distribution like Ubuntu or CentOS. These smaller images come with fewer pre-installed packages and libraries, which means a smaller image size.
- **Reduce the number of layers:** Each instruction in a Dockerfile creates a new layer in the image, and each layer adds to the size of the image. To reduce the size of the image, you

should combine multiple instructions into a single layer. You can also use multi-stage builds to remove unnecessary layers.

- **Remove unnecessary files:** Remove unnecessary files and directories from the image, such as log files, documentation, or source code.
- **Use a .dockerignore file:** Use a .dockerignore file to exclude files and directories from the build context. This can help reduce the size of the image by excluding unnecessary files from the build.
- **Minimize dependencies:** Only include the packages and libraries that are necessary for your application to run. Removing unnecessary dependencies can help reduce the size of the image.
- **Use smaller binaries:** Use smaller binaries for your application, such as statically-linked binaries, which include all dependencies in a single file.
- **Use compressed layers:** Use compressed layers to reduce the size of the image. You can use the --compress flag with the docker build command to compress each layer.

How do you increase size of the docker container size?

1. Stop the docker daemon after take the backup of the container
2. Modify the docker config file in /etc/sysconfig/docker-storage
3. In docker Storage option, specify the memory that you need to Increase.
4. Restart the service

Docker File vs Docker Compose File

A Docker File is a simple text file that contains the commands a user could call to assemble an image

Docker Compose is a tool that allows to define and run multi-container Docker Applications

KUBERNETES

Kubernetes Installation

Minimum system requirement for master node is 2-core cpu and 4GB RAM

1. `sudo apt update`
2. `sudo apt-get install -y apt-transport-https`
3. `sudo su -`
4. `curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add`
5. `echo 'deb http://apt.kubernetes.io/ kubernetes-xenial main' > /etc/apt/sources.list.d/kubernetes.list`
6. `sudo apt update`
7. `sudo apt-get install -y docker.io`
8. `sudo systemctl start docker`
9. `sudo systemctl enable docker.service`
9. `sudo apt-get install -y kubelet kubeadm kubectl kubernetes-cni`
10. Create a **ami from above instance to create workernodes**

11. After ami is available, login again to **master node**

(Make sure docker is running)

12. `sudo su -`

13. **kubeadm init**

ERROR1: if we get kubelet isn't running or healthy

kubelet doesn't get access to docker engine which means

we need to configure cgroup of docker

create a file `/etc/docker/daemon.json` with below content

```
{  
    "exec-opts": ["native.cgroupdriver=systemd"]  
}
```

Reload docker daemon

`systemctl daemon-reload`

```
systemctl restart docker
systemctl restart kubelet
```

Run kubeadm init again

ERROR2: if we get fileavailable error just delete those files

ERROR3: if kubelet is running kill it

```
lsof -i :<kublet_port>
```

```
kill -9 <process_id>
```

Run kubeadm init again

if we get kubeadm join command at the end means master node setup is successful and save the join command.

14. configure K8S kubectl

- exit from the root

- copy the default k8s conf file to our home directory

```
mkdir -p $HOME/.kube
```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

15. We install k8s CNI Driver

```
sudo sysctl net.bridge.bridge-nf-call-iptables=1
```

```
kubectl apply -f "https://cloud.weave.works/k8s/v1.13/net.yaml"
```

check for node status - kubectl get nodes

16. Login to worker node

```
sudo su -
```

create a file /etc/docker/daemon.json with below content

```
{
    "exec-opts": ["native.cgroupdriver=systemd"]
}
```

Reload docker daemon

```
systemctl daemon-reload
```

```
systemctl restart docker
```


systemctl restart kubelet

Now open ports of master nodes

Run the join command with token which we got from master node

Repeat the same steps in other worker nodes

Kubernetes Architecture

The architecture of k8s differs from master and worker node

Master node components

1. Api Server / kube-api-server

- It is the main management point of the cluster and also called as brain of the cluster.
- All the components are directly connected to API server, they communicate through API server only and no other component will communicate directly with each other.
- This is the only component which connects and got access to etcd.
- All the cluster requests are authenticated and authorized by API server.
- API server has a watch mechanism for watching the changes in cluster.

2. etcd

- etcd is a distributed, consistent key value store used for storing the complete cluster information/data.
- etcd contains data such as configuration management of cluster distributed work and basically complete cluster information.

Etcd default location

/etc/Kubernetes/pki/etcd-manager-main/etcd-clients-ca

3. scheduler / kube-scheduler

- The scheduler always watches for a new pod request and decides which worker node this pod should be created.
- Based on the worker node load, affinity and anti-affinity, taint configuration pod will be scheduled to a particular node.

4.controller manager

- It is a daemon that always runs and embeds core control loops known as controllers which will be always tries to match the current state of cluster to desired state.

- K8s has some inbuild controllers such as Deployment, Daemon Set, Replica Set, Replication controller, node controller, jobs, cronjob, endpoint controller, namespace controller etc.

Worker node components

kubelet

- It is an agent that runs on each and every worker node and it always watches the API server for pod related changes running in its worker node.

- kubelet always make sure that the assigned pods to its worker node is running.

- kubelet is the one which communicates with containerisation tool (docker daemon) through docker API (CRI). work of kubelet is to create and run the pods. Always reports the status of the worker node and each pod to API server. (Uses a tool call Cadvisor)

- Kubelet is the one which runs probes.

kube service proxy (in k8s service means networking)

- Service proxy runs on each and every worker node and is responsible for watching API server for any changes in service configuration (any network related configuration).
- Based on the configuration service proxy manages the entire network of worker node.

Container runtime interface (CRI)

- This component initially identifies the container technology and connects it to kubelet.
- This is responsible for running container.

Pod

- pods are the smallest deployable object in Kubernetes.
 - pod should contain at least one container and can have n number of containers.
- If pod contains more than one container all containers share same memory assigned to that pod.

YAML file

- Filetype. yaml / .yml
- YAML file will contain key value pairs where key is fixed and defined by the tool and value is user defined configuration. (Yet Another Markup Language)

Pod Yaml Example

apiVersion: v1

- This is used to specify the version of API to create a particular k8s object.
- The field is case sensitive, it will be in camelCase
- The types of API we have in k8s are alpha, beta and stable versions.

kind: Pod

- used to specify the type of k8s object to create.
- Always object name first letter is capital.

metadata:

contains the information that helps us to uniquely identify the object.

There are 3 types of metadata

1. name

2. labels

- k8s labels are used to identify the object.

ex: name, environment, tier, release.

3. annotations

spec:

- actual configuration of the objects

apiVersion: v1

kind: Pod

metadata:

name: my-first-pod

spec:

containers:

- name: my-nginx

image: nginx:latest

ports:

- containerPort: 80

TO create / apply a configuration

kubectl apply -f <file>.yaml

To list objects

kubectl get <object_type>

List deployment

kubectl get deployments

To delete objects

kubectl delete <object_type>

K8S Labels and selectors

Labels

- K8S labels is a metadata key value which can be applied to any object in k8s.
- Labels are used to identify by using selectors.
- Multiple objects can have same label, multiple labels to same object and Label length should be less than 63 characters.

TO list all labels of a object

```
kubectl get <object_type> <object_name> --show-labels
```

Labels can be used to group resources

For example, you could add a label to a set of pods indicating that they belong to a particular application or environment, and then use that label to select all of the pods belonging to that application or environment.

Selectors

- Selectors are used to filter and identify the labelled k8s object.

You can use selectors to filter resources based on their labels and perform actions on them. For **example**, you could use a selector to find all of the pods that belong to a particular application or environment, and then scale up or down the number of replicas of those pods.

Equality-Based

- It will use only one label in comparison and it will look for objects with exact same string in label.
- we can use 3 types of operators equal (= or ==) and not-equal (!=)

example: selectors:

```
matchLabels:
  app=nginx
(or)
app: nginx
```

set-based

- This type of selector allows us to filter objects based on multiple set of values to a label key.
- 3 types of operators we can use in, notin and exists.

example: selectors:

```
matchLabels:
  app in (nginx, my-nginx)
```

app exists (nginx, my-nginx)

app notin (nginx, my-nginx)

Annotations

- These are used for record purpose only and to provide some user information to objects.

example: personal info, phone number, image registry, author

Replica Set vs Replication Controller

- Both ensures that a specified number of identical replicas of pod are always running at any given point of time.
- Replication controller is a very old way of replicating the pod and now it is replaced by ReplicaSet
- The only difference b/w them is their selector types.

Replication Controller supports only **equality-based selector**.

ReplicaSet supports both **equality-based and set-based selectors**,

Works for stateful applications

Deployment controller / Deployment / k8s deployment

- Deployment is used to create replicas of pod and it makes sure at a given point of time the number of replicas of pod is always running.
- Deployment internally uses ReplicaSet to replicate the pods.
- If we update the configuration in deployment, it will automatically update it to all the pods.
- Rollout and Rollback of pod update is possible.
- we can pause a deployment whenever we need, works for stateless applications
- Deployment has got its own internal autoscaler which is of type horizontal scaler

Deployment = pod + ReplicaSet + autoscaling + Rolling Updates

To apply calling

kubectl autoscale deployment.v1.apps/<deployment_name> --min=5 --max=20 --cpu-percent=50

- **scaleup and scale down** is possible by increasing and decreasing the replica count at any given point of time.

kubectl scale deployment.v1.apps/<deployment_name> --replicas=10

- **Deployment is a cluster level object.**

kubectl apply -f deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-new
  labels:
    app: my-deployment-nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - container_port: 80
```

Daemon Set

- Daemon Set ensures that a copy of pod is always running on all the worker nodes in the cluster.

-If a new node is added or if deleted DaemonSet will automatically adds/deletes the pod.

Usage - we use DaemonSet to deploy **monitoring agents** in every **worker node**.

- Log collection daemons: to grab the logs from worker and all the pods running in it

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
spec:
```

```

selector:
  matchLabels:
    app: daemonset-nginx
template:
  metadata:
    labels:
      app: daemonset-nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
          - containerPort: 80

```

Stateful Applications

- User session data is saved at the server side.
- if server goes down, it is difficult to transfer the session data to another server.
- This type of application will not work if we want to implement autoscaling.

Ex:- Databases, Facebook, Banking App, MySQL, Mango DB, Online Shopping Cart

Stateless Applications

- user session data is never saved at the server side.
- using a common authentication gateway / client token method to validate the users once for multiple microservices. Ex:-UDP, DNS, HTTP, Chatbot

(csrf token)

Stateful Set

- StatefulSet = Deployment + sticky identity for each and every pod replica.
- Unlike a deployment a StatefulSet maintains a sticky identity for each of the pod.

Ex – Mysql database, FTP, Telnet

Node controller

- Looks for node statuses and responds to API server only when a node is down.

Endpoint Controller

- Populates the information of endpoints of all the objects.

Monolithic and Microservice architecture

Monolithic architecture

- A monolithic application has a single code base with multiple modules in it.
- It is a single build for entire application.
- To make minor changes to application, we need to re-build and re-deploy the complete application.
- scaling is very challenging.

Microservice architecture

- A microservice application is composed of small (micro) services.
- Each service will have a different code base.
- Application are divided into as small as possible sub applications called service which are independent to each other which are called loosely coupled.
- Each service can be managed separately, and it is deployable separately.
- Services need not to share same technology stack or frameworks.

Service (svc)

Services are a way to provide a stable network endpoint to a set of pods, allowing other applications to communicate with them.

- Services are always created and works at cluster level.
- k8s prefers to use 30000 - 50000 range of ports to define services.

1. ClusterIP

- This is the default type of service and provides a virtual IP address that is only accessible within the cluster. It can be used for communication between pods in the same cluster.
- ClusterIP cannot be accessed outside cluster & default loadbalancers of k8s.

apiVersion: v1

kind: Service

metadata:

 name: my-svc

spec:

 type: ClusterIP

 selector:

 app: my-nginx

 ports:

 - name: http

 port: 30080

targetPort: 8080

2. NodePort

- A node port service is to get the external traffic directed to our services / applications running inside a pod within the cluster.

NodePort is a type of service in Kubernetes that exposes a service on a static port on each node in the cluster. It is typically used when an application needs to be accessed from outside the cluster, such as from the internet or from other applications that are not part of the cluster.

- By default, NodePort acts as a load balancer.
- Automatically a ClusterIP will be created internally.

NodePort = ClusterIP + a port mapping to the all the nodes of cluster.

- If we won't specify any port while creating nodeport, k8s will automatically assign a port between the range 30000 - 32767

- By default, nodeport will open the port in all the node in cluster including master node.

apiVersion: v1

kind: Service

metadata:

name: my-svc

spec:

type: NodePort

selector:

app: my-nginx

ports:

- name: http

nodePort:30082

port: 8080

targetPort: 80

3. Load Balancer

- It is a type of service which is used to link external load balancer to the cluster.

- This type of service is used by cloud providers and this service is completely depends on cloud providers.

- K8s now provides a better alternative for this service type which is called Ingress.

(In our company we use ingress)

Namespaces

- k8s name space is to support create a virtual cluster within a physical cluster, and provide a secure and isolated environment for running applications.
- Each and every object in k8s must be in a namespace, Namespaces are cluster level.
- If we won't specify namespace, objects will be created in default namespace of k8s.
- By default, pods in same namespace can communicate with each other.

Type of default NS

1. default

- This NS is used for all the objects which are not belongs to any other namespace.
- If we won't specify any namespace while creating an object in k8s then that object will be created in default namespace.

2. kube-system

- This namespace is always used for objects created by the k8s system.

3. kube-public

- The objects in this namespace are available or accessible to all.
- All the objects in these namespaces are made public.

4. kube-node-lease

- This namespace holds lease objects associated with each node.
- Node lease allows the kubelet to send heartbeats so that the control plane can detect node failure.

To list namespace

kubectl get namespaces

To list objects in a namespace

kubectl get pods --namespace <NS_name>

(OR)

kubectl get pods -n <NS_name>

To list objects from all namespaces

kubectl get pods --all-namespaces

To create a namespace

kubectl create namespace <ns_name>

To create k8s object in a namespace

1. in the spec file

metadata:

namespace: <ns_name>

2. Using the apply command

kubectl apply -n <ns_name> -f <spec>.yaml

Note: what if we use both inside specfile and also in apply command

- apply command check and compares the namespace and won't allow to create the object if the namespace is different.

How a microservice will communicate with other microservice

What is service discovery in k8s

Service Discovery

There are 2 ways of discovering a service

DNS

- DNS server is added to the cluster in order to watch the k8s service requests.
- API servers create DNS record sets for each new service.
- Record A type is used in k8s service discovery and this DNS is created on service and pod objects.

syntax of DNS

`<object_name>.<namespace_name>.<object_type>.<k8s_domain>`

ex: `my-app-svc.default.svc.local`

ex: `my-app-svc.default.svc.example.com`

ENV variables

- whichever the pods that runs on a node, k8s adds environment variables for each of them to identify the service running in it.

Headless service (Static IP)

- When we neither need nor want load balancing and a single IP point to a service, we need use headless service.

This allows the client to connect directly to one of the pods without going through a load balancer or a proxy.

- Headless service returns all the ips of the pods it is selecting.
- Headless service is created by specifying none for clusterIP

Headless services are useful in scenarios where you need to connect to the individual pods directly, such as with StatefulSets or database clusters.

Pod phases / status / states / life cycle (PRFUST)

1. Pending

- This is the status of pod when pod will be waiting for k8s cluster to accept it.
- Pod will be downloading the image from registry.
- Pod will be in pending till the scheduler assigns a node to the pod.
- There are resource constraints preventing the containers from starting.]

2. Running

- The pod has been assigned a node and all the containers inside the pod is running.
- At least one container is in running state and others in starting or restarting state then pod

will show running state.

3. Failed

- At least one of the containers in the Pod has exited with a status code other than 0. The Pod is considered to be in the "Failed" phase

4. Succeeded

- All the containers in pod have been terminated successfully/gracefully.

5. Unknown

- The state of the Pod cannot be determined. This typically occurs when there is an error communicating with the Kubernetes API server.

6. Terminating

- when pod is being deleted.

Container status

Running

- Means container is running the process inside without any error

Terminated

- Process inside the container has completed the execution or may be failed due to some error.

Waiting

- If a container is not running or neither in terminated state.

Common errors

ImagePullBackOff

- Docker image registry is not accessible.
- Image name / tag version specified is incorrect.

n "ImagePullBackOff" error in Kubernetes indicates that the container runtime was unable to pull the container image specified in the pod specification. Here are some possible reasons for this error:

1. Incorrect image name or tag: The container image name or tag specified in the pod specification may be incorrect, or the image may not be available in the specified registry. Verify that the image name and tag are correct and that the image is available in the specified registry.
2. Incorrect authentication: If the container image is stored in a private registry, the authentication credentials specified in the pod specification may be incorrect or insufficient. Verify that the credentials are correct and that the user has the necessary permissions to access the image.
3. Network issues: The container runtime may be unable to connect to the registry to download the image due to network connectivity issues. Verify that the network is working properly and that the container runtime can access the registry.
4. Resource constraints: The node hosting the pod may not have enough resources (such as memory or disk space) to download and run the container image. Verify that the node has sufficient resources to accommodate the pod.

troubleshoot an "ImagePullBackOff" error in Kubernetes, you can follow these steps:

1. Check the pod status: Use the `kubectl get pods` command to check the status of the pod. If the pod is in an error state, look for the "ImagePullBackOff" message in the "STATUS" column.
2. Check the container image: Verify that the container image name and tag specified in the pod specification are correct. Check the spelling and make sure the image exists in the registry.
3. Check the registry credentials: If the container image is stored in a private registry, verify that the authentication credentials specified in the pod specification are correct. Use the `kubectl describe secret <secret-name>` command to check the details of the secret.
4. Check the network connectivity: Verify that the node running the pod can connect to the registry to download the container image. Check the network connectivity between the node and the registry.
5. Check the resource constraints: Verify that the node running the pod has enough resources (such as CPU, memory, and disk space) to download and run the container image. Check the resource requests and limits specified in the pod specification.
6. Check the container runtime logs: Use the `kubectl logs <pod-name> <container-name>` command to check the container logs. Look for any errors related to image pulling or network connectivity.
7. Manually pull the image: Try running the `docker pull <image-name>:<tag>` command on the node where the pod is running to see if the image can be downloaded manually. If the command fails, it may indicate a network connectivity issue.

CrashLoopBackOff

- We get this error when probe check has failed.

- Docker image may be faulty.

A CrashLoopBackOff error is a common error message in Kubernetes that indicates that a pod has failed to start and is unable to recover. This error occurs when a pod crashes and Kubernetes attempts to restart it, but the pod continues to fail.

Here are some possible reasons for a CrashLoopBackOff error:

1. Incorrect pod configuration: The pod's configuration may be incorrect, causing the container to fail repeatedly. Check the pod's YAML file for errors, such as incorrect syntax, missing or incorrect environment variables, or incorrect image or container specifications.
2. Insufficient resources: The pod may not have enough resources to run properly, such as CPU or memory. Check the resource limits and requests for the pod and ensure that they are appropriate for the workload.
3. Unhealthy container: The container in the pod may not be healthy and may be crashing repeatedly. Check the container logs for any error messages and debug the issue accordingly.
4. Network connectivity issues: The pod may be unable to connect to its required network resources, such as a database or API endpoint. Check the pod's network settings and ensure that it can access the necessary resources.
5. Dependency issues: The pod may have dependencies on other services or resources that are not available or not functioning properly. Check for any missing dependencies or errors in other services or resources that the pod relies on.

to troubleshoot a "CrashLoopBackOff" error in Kubernetes, you can follow these steps:

1. Check the pod status: Use the `kubectl get pods` command to check the status of the pod. If the pod is in an error state, look for the "CrashLoopBackOff" message in the "STATUS" column.
2. Check the container logs: Use the `kubectl logs <pod-name> <container-name>` command to check the container logs. Look for any errors or exceptions that may have caused the container to crash. The logs may contain clues as to what went wrong.
3. Check the container configuration: Verify that the container configuration (specified in the pod specification) is correct. Make sure that the container image name and tag are correct, and that any environment variables or configuration files are properly specified.
4. Check the resource constraints: Verify that the node running the pod has enough resources (such as CPU and memory) to run the container. Check the resource requests and limits specified in the pod specification.
5. Check for persistent issues: If the container crashes repeatedly, there may be a persistent issue with the container or application code. Check for issues such as

- infinite loops, memory leaks, or other bugs that may be causing the container to crash.
6. Check for compatibility issues: Verify that the container image is compatible with the Kubernetes version and operating system running on the node. Check for any known compatibility issues between the container image and the node.
 7. Check for dependencies: If the container has dependencies on other services or resources, make sure that those dependencies are properly configured and available. Check for issues such as network connectivity, DNS resolution, or missing configuration files.

RunContainerError

- Configmap / secrets are missing.
- Volumes are not available

k8s volumes

Persistent volume (PV)

- It is a storage space which can be claimed to any pod in the cluster.
 - These are cluster level object and not bound to namespace.
- Even if the pod got deleted or rescheduled, data will be persisted
- we can control the access to volume in 3 ways:
- **ReadOnlyMany(ROX)** allows being mounted by multiple nodes in read-only mode.
 - **ReadWriteOnce(RWO)** allows being mounted by a single node in read-write mode.
 - **ReadWriteMany(RWX)** allows multiple nodes to be mounted in read-write mode.

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-pv

labels:

volume: test

spec:

storageClassName: local

accessModes:

- ReadWriteOnce

capacity:

storage: 2Gi

hostPath:

path: "/home/ubuntu/my-pv-volume"

Persistent volume claim (pvc)

- This is the object used to claim / mount the required amount of storage from persistent volume to any pod in the cluster.
- After we create the Persistent Volume Claim, the Kubernetes control plane looks for a Persistent Volume that satisfies the claim's requirements.
- If the control plane finds a suitable Persistent Volume with the same Storage Class, it binds the claim to the volume.

Edit /Resize PVC → `kubectrl edit pvc $your_pvc`

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  volumeName: my-pv
  storageClassName: local
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

using this in a pod

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pvc-pod
spec:
  volumes:
    - name: pvc-volume
      persistentVolumeClaim:
        claimName: my-pvc # This name should be same the PVC object name
  containers:
    - name: my-nginx
      image: nginx:latest
```


ports:

- containerPort: 80

volumeMounts:

- mountPath: "/usr/share/nginx/html"

name: pvc-volume # This name should be same as the above volume

name

Command to add file inside a pv

kubectrl exec my-pod -- sh -c 'echo "Hello, world!" > /data/myfile.txt'

Pod patterns / container types

Init containers

- Init containers are the containers that will run completely before starting the main app container.
- These are pod level objects

These are some of the **scenarios** where you can use this pattern

- You can use this pattern where your application or main containers need some prerequisites such as **preparing the environment, installing some software, database setup, permissions on the file system before starting.**

- You can use this pattern where you want to **delay the start of the main containers.**

apiVersion: v1

kind: Pod

metadata:

name: init-container

spec:

containers:

- name: my-nginx

image: nginx:latest

ports:

- containerPort: 80

initContainers:

- name: busybox

image: busybox

command: ["/bin/sh"]

```
args: ["-c","echo '<html><h2>THis is a init container</h2></html>' >> /work-  
dir/index.html"]
```

```
volumeMounts:
```

```
- name: workdir
```

```
mountPath: "/work-dir"
```

```
volumes:
```

```
- name: workdir
```

```
emptyDir: {}
```

Sidecar container

- These are the containers that will **run along** with the **main app container**.
- We have an app container which is working fine but we want to extend the functionality without changing the existing code in main container for this purpose we can use sidecar container.
- We use this container to feed the log data to monitoring tools.

These are some of the **scenarios** where you can use this **pattern**

- Whenever you want to extend the functionality of the existing single container pod without touching the existing one.
- You can use this pattern to synchronize the main container code with the git server pull.
- You can use this pattern for sending log events to the external server.
- You can use this pattern for network-related tasks.

```
apiVersion: v1
```

```
kind: Pod
```

```
metadata:
```

```
name: init-container
```

```
spec:
```

```
containers:
```

```
- name: my-nginx
```

```
image: nginx:latest
```

```
ports:
```

```
- containerPort: 80
```

```
volumeMounts:
```

```

      - name: workdir
        mountPath: "/var/log/nginx"
    - name: sidecar-busybox
      image: busybox
      command: ["/bin/sh", "-c","while true; do cat /var/log/nginx/access.log
/var/log/nginx/error.log; sleep 15; done"]
      volumeMounts:
        - name: workdir
          mountPath: "/var/log/nginx"
    volumes:
    - name: workdir
      emptyDir: {}

```

Adaptor container

- In this pattern we use a sidecar container to feed the log data to a monitoring tool.

<https://www.magalix.com/blog/kubernetes-patterns-the-ambassador-pattern>

Probes

- Probes can track success or failure of the other applications.
- When there is a subsequent failure occurs, we can define probe to get triggered.
- Probes work at **container level**.

Common fields in probes

1. **initialDelaySeconds**: This field specifies the number of seconds to wait before starting the probe after the container starts. This can be useful for allowing the container to fully initialize before running the probe.
2. **timeoutSeconds**: This field specifies the number of seconds to wait for a probe to complete before considering it a failure. If the probe takes longer than the specified timeout, Kubernetes will mark it as failed.
3. **periodSeconds**: This field specifies the number of seconds between probe runs. Kubernetes will run the probe at the specified interval to check the container's health.
4. **successThreshold**: This field specifies the number of consecutive successful probe runs required for the container to be considered healthy. If the probe fails, the counter is reset to zero.
5. **failureThreshold**: This field specifies the number of consecutive failed probe runs required for the container to be considered unhealthy. If the probe succeeds, the counter is reset to zero.

6. **httpGet**: This field specifies an HTTP request to be sent to the container for the probe. It includes parameters such as the request path, port number, and HTTP method.
7. **tcpSocket**: This field specifies a TCP socket to connect to for the probe. It includes parameters such as the port number and the timeout for the socket connection.
8. **exec**: This field specifies a command to be run inside the container for the probe. It includes parameters such as the command to be run and any arguments.

These fields allow for the fine-tuning of probe behaviour, including how frequently they run, how long they take to timeout, and what type of checks are performed. By using these fields effectively, operators can ensure that their containers are running correctly and that any issues are detected and addressed quickly.

Liveness probe

- The livenessprobe is used to determine if the application inside the container is healthy or needs to be restarted.
- If livenessprobe fails it will mark the container to be restarted by kubelet.

1. LivenessProbe with http

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-http
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/liveness
      args:
        - /server
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 3
        periodSeconds: 3
```

2. TCP

```
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 3
  periodSeconds: 3
```

3. exec

```
livenessProbe:
  exec:
    command: ["/bin/sh"]
  initialDelaySeconds: 3
  periodSeconds: 3
```

4. named port

```
ports:
  - name: liveness-port
    containerPort: 8080
```

hostPort: 8080

```
livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  initialDelaySeconds: 3
  periodSeconds: 3
```

Readiness Probe

- Readiness Probe is used to determine that an application running inside a container is in a state to accept the traffic.

- Sometimes, applications are temporarily unable to serve traffic. For example, an application might need to load large data or configuration files during startup, or depend on external services after startup. In such cases, you don't want to kill the application, but you don't want to send it requests either. Kubernetes provides readiness probes to detect and mitigate these situations

- When this probe is successful/fails, the traffic from the load balancer is allowed/halted to the application inside the container.

Readiness Probe:

tcpSocket:

port: 8080

initialDelaySeconds: 15

periodSeconds: 10

Startup Probe

- This probe will run at the initial start of the container.
- This probe allows us to give maximum start-up time for application before running livenessProbe or readinessprobe.

startupProbe:

httpGet:

path: /health

port: 8080

initialDelaySeconds: 3

periodSeconds: 3

RBAC (Role Base Access Control)

How user access is maintained in k8s?

Role-Based access control

- accounts
- Roles
- Binding of roles

<https://medium.com/rahasak/kubernetes-role-base-access-control-with-service-account-e4c65e3f25cc>

Accounts

There are 2 types accounts in k8s

1. USER ACCOUNT

- It is used to for human users to control the access to k8s cluster.

2. SERVICE ACCOUNT

- It is used to give access to k8s cluster to external tools/applications and also, to give access to other components inside the cluster.
- Any application running inside or outside the cluster need a service account to access API server.
- When a SA is created, first k8s creates a token and keeps that token in a secret object and then the secret object is linked to the service account.

To list service account → `kubectl get sa` or `kubectl get serviceaccounts`

To create SA → `kubectl create serviceaccount <service_account_name>`

To check secret object of SA → `kubectl describe secrets <secret_name>`

Attach a Service Account to pod

```

apiVersion: v1
kind: Pod
metadata:
  name: sa-pod
spec:
  serviceAccountName: my-sa
  containers:
    - name: liveness
      image: k8s.gcr.io/liveness
      args:
        - /server
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
        initialDelaySeconds: 3
        periodSeconds: 3

```

Roles

- For user roles are the set of rules which defines the access level to k8s resources.
- Roles are always user defined to any type of account.
- Roles work at namespace level.

Common fields in roles

apiGroups: List the api groups to control the access to a account.

Subject: Users, service_account or Groups

Resources: K8S objects on which we want to define this role

ex: Pods, Deployments etc..

Verbs: The operations/actions that a user can perform in k8s cluster

["get", "list", "create", "update", "delete", "watch", "patch"]

Create a role

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: "*"

resources: ["pods"]

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

To create role in cli

kubectl create role <role_name> --resource=pods --verb=list -n <namespace>

ClusterRole

- It is a cluster wide role which is a non-namespace object.

- Cluster Role defines permissions on namespace objects and it grants permissions across all the namespaces or individual namespace also.

usage:

If we need to define permissions inside a namespace use Roles.

If we need to define permissions at cluster wide use Cluster Role

Create a role

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: cluste-role-pod-reader

rules:

```
- apiGroups: "*"
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

RoleBinding and ClusterRoleBinding

- Role binding as name indicates is used to bind roles to subjects (user, sa, groups)
 - we can use role binding to bind cluster role to a role of a particular namespace.

RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: my-role-binding
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-reader # the name of role we want to attach the account.
subjects:
```

- kind: ServiceAccount

```
  name: my-sa # the name of account need to attached to role
  namespace: default
```

TO check the roles affect

```
kubectl          auth          can-i          list          svc
as=system:serviceaccount:<namespace>:<service_account> -n <namepsace>
```

ex: `kubectl auth can-i list svc --as=system:serviceaccount:default:my-sa -n default`

ClusterRoleBinding

Cluster Role binding is used to bind cluster roles to subjects (user, sa, groups)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
```

```

    name: my-cluste-role-binding
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: <cluster_role_name> # name of cluster role we want to attach the account.
  subjects:
    - kind: ServiceAccount
      name: my-sa # the name of account needs to attached to role
      namespace: default

```

Node selector

- Node selector is a way of binding pods to a particular worker node based on labels given to that node.
- Logical expressions type of selection cannot be achieved in node selector.

To label a node

```
kubectl label node <node_name> <key>=<value>
```

Node Affinity and anti-affinity (Inter-pod affinity)

- Node selector with logical expressions is Affinity
- Using node affinity we can spread pods across worker nodes based on CPU and RAM capacity (memory-intense mode), Availability zones (HA mode).
- required During Scheduling Ignored During Execution The scheduler can't schedule the Pod unless the rule is met.

- preferred During Scheduling Ignored During Execution

The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod in normal way.

- IgnoredDuringExecution - if the node labels change after Kubernetes schedules the Pod, the Pod continues to run.

```

spec:
  containers:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:

```

```

labelSelector:
  - matchExpressions:
      - key: env

operator: in
values:
  - test
  - prod

```

Anti-Afinity (Inter-pod affinity)

- This is used to define whether a given pod should or should not be scheduled on a particular node based on labels.

```

spec:
  containers:
    ....
  ifNotPresent:
nodeSelector:
  env: test

spec:
  containers:
    ...
  affinity:
    podAntiAfinity:
requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      - matchExpressions:
          - key: env
            operator: in
            values:
              - test
              - qa

```

sta

Taints and Tolerations

- Taints are used to repel the pods from a specific worker node.
- We can apply taint to worker nodes which tells scheduler to repel all pods except the pods with toleration defined in it for the taint.

- **2 operators** we can use Equal and Exists (If we use Exists, no value required)

- Below are the effects we can use,

1) **NoSchedule** - This taint means unless a pod with toleration k8s won't be able to schedule a pod to tainted node.

2) **NoExecute** - To delete all the pods except some required pods we can use this.

To taint a worker node

```
kubecttl taint nodes <node_name> <taint_key>=<taint_value>:<taint_effect>
```

To remove the taint (use - at the end)

```
kubecttl taint nodes <node_name> <taint_key>=<taint_value>:<taint_effect>-
```

To add tolerations

(Single toleration with Equal)

spec:

tolerations:

```
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

(single toleration with Exists)

spec:

tolerations:

```
- key: "key1"
  operator: "Exists"
  effect: "NoSchedule"
```

(multiple tolerations)

```
spec:
  tolerations:
    - key: "key1"
      operator: "Equal"
      value: "value1"
      effect: "NoSchedule"
    - key: "key2"
      operator: "Equal"
      value: "value2"
      effect: "NoExecute"
```

ConfigMap and secrets

Config Map

- Configmap is k8s object that allows us to separate the configuration data/files from the image content of the pod.
- Using this we can keep the configurations of same application portable can be used in multiple pods.
- Configmaps are used for non-confidential data.

Create a Configmap

1. Create a file by name "app.properties"

```
environment=test
database_url="192.168.10.9"
```

2. Create configmaps

- a. load the single config file

```
kubectl create configmap <configmap_name> --from-file configs/app.properties
```

- b. load the multiple config file

```
kubectl create configmap <configmap_name> --from-file configs/
```

3. using configmaps in pod

```
spec:
  containers:
    - name: nginx
      image: nginx
      env:
        - name: CURRENT_NAME
```

valueFrom:

configMapKeyRef:

name: <configmap_name

key: environment

Secrets

- Secrets are used for confidential data.
- k8s by default uses base-64 encoding method to encrypt the data.

type

Opaque - for user related data in base64

service account token - kubernetes.io/service-account-token

docker config - kubernetes.io/dockerconfig

1. get the encoded values

echo "production" | base64

output: cHJvZHVjdGlvbgo=

2. use the above encoded value in secret

apiVersion: v1

kind: Secret

metadata:

name: my-secret

type: Opaque

data:

environment: cHJvZHVjdGlvbgo=

3. using secrets in pod

spec:

containers:

- name: nginx

image: nginx

env:

- name: CURRENT_NAME

valueFrom:

secretKeyRef:

name: <secret_name>

key: environment

Count quota

- We use k8s quotas to precisely specify the number of objects a user can work with.
- We can limit the no's of pods to namespace or memory to a pod
- We can define quotas for the below objects

count/persistentvolumeclaims

count/services

count/secrets

count/configmaps

count/replicationcontrollers

Applying the resource quota for an object (count quota)

apiVersion: v1

kind: ResourceQuota

metadata:

name: count-quota

spec:

hard:

pods: "2"

or

count/pods: "2"

Applying quotas for CPU, RAM, DISK SPACE

apiVersion: v1

kind: ResourceQuota

metadata:

name: ram-quota

spec:

hard:

request.memory: "500Mi"

limits.memory: "800Mi"

Limitations

CPU

- 1 cpu, in k8s is equal to 1 core/cpu 100%.

ex: 0.1 cpu - 10 % of 1 cpu - 1 gup - 1 hyperthread - 1 Nueral

Memory

- It will be in terms of bytes
- It should fixed numbers always

syntax: Ei, Pi, Ti, Gi, Mi, Ki

using quotas in pod

spec:

containers:

- name: nginx

image: nginx

resources:

requests:

memory: "500Mi"

cpu: 0.5

limits:

memory: "1Gi"

cpu: 1

Note: In terms of CPU k8s does not go behind the limits (it never throttles).

but in terms of memory (RAM) k8s allows the pods to go behind the limit of RAM.

Ingress

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

It may provide load balancing, SSL Termination, name based virtual hosting

Ingress exposes HTTP and HTTPS routes from outside the cluster to service within the cluster

Cloud load balancer are costly as most of the times billing will be per requests so to avoid this the Kubernetes solution is Ingress

1. Ingress controller

- This controller is used to execute the ingress resources which contains routing rules and brings the external traffic based on routing rules to the internal service.

- This controller will automatically monitor the existing resources and also identifies new ingress resources.

- This will be a third-party controller (tools) which we need to install it for one time in our cluster as controller.

- We are using nginx as ingress controller

2. Ingress resources

- In k8s Ingress resource is type of object which is used to define routing rules based on path of incoming traffic to internal cluster service.

- api for ingress resource networking.k8s.io/v1

- Ingress can be used for reverse proxy means to expose multiple services under same IP.

- Ingress can be used to apply ssl/tls certificates.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: my-ingress
spec:
  rules:
    - host: example.com
      http:
        paths:
          - path: /
            backend
              serviceName: my-svc
              servicePort: 80
          - path: /login
            backend
              serviceName: my-login
              servicePort: 8090
```

Note: / - means the request from "http://www.example.com"

/Login - means the request from "http://www.example.com/login"

serviceName should be same as the name of the service metadata.

Network policy

- By default, in k8s any pod can communicate with each other within the cluster across the different namespaces and worker node.
- The default network of k8s is of open stack model this opens a huge risk for potential security issues.
- We can use network policies to apply Deny all for the cluster and we can write policies to allow only required requests to cluster.
- Network policies are defined for ingress and egress.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
    - Ingress
```

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: nginx-pod
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
  ingress:
    - from:
      - podSelector:
          matchLabels:
```

app: backend

ports:

- port: 80

protocol: TCP

calico

- Calico is created by a company called Tigra

called "Tigra".

- Tigra supports a wide range of CNI for Kubernetes only.

- Using this CNI plugin we can extend the usage of network policies and we can

improve

the security over network.

- Using calico we can define dynamic network policies such as auto calculated from

many

sources of data is possible.

- multiple port mapping is supported in calico

Cronjob and Jobs

- The main function of a job is to create one or more pods and tracks the success status of pods.

- Jobs ensure that the specified number of pods is completed successfully and when the job is completed, pods go to the shutdown state and Job goes to completed state.

- Mainly we use jobs to run pod temporarily till the task is completed and to run tasks parallelly.

apiVersion: batch/v1

kind: Job

metadata:

name: my-job

spec:

template:

spec:

containers:

- name: busybox

image: busybox

```
command: ["echo", "This is first job"]  
restartPolicy: Never
```

restartPolicy

- This is applied to pod not for the Job

The different type of jobs or common parameters is,

Completions

- This is the number of times the job to run. default is 1.
- If, completions is 5 then job will run 5 times means 5 pods.

```
apiVersion: batch/v1
```

```
kind: Job
```

```
metadata:
```

```
  name: my-job
```

```
spec:
```

```
  completions: 5
```

```
  template:
```

```
    spec:
```

```
      containers:
```

```
        - name: busybox
```

```
          image: busybox
```

```
          command: ["echo", "This is first job"]
```

```
          restartPolicy: Never
```

Parallelism

- By default, jobs run serially so to run jobs parallelly we need to use the parallelism.
- parallelism is used to set the number of job that need to run parallelly.

```
apiVersion: batch/v1
```

```
kind: Job
```

```
metadata:
  name: my-job
spec:
  completions: 5
  parallelism: 2
  template:
    spec:
      containers:
      - name: busybox
        image: busybox
        command: ["echo", "This is first job"]
      restartPolicy: Never
```

backoffLimit

- If the container is failing for some reason which affects the completion the job, then still job creates more pods one after another until it succeeds which will simply put a load on the cluster, in this case backoffLimit is used.
- backoffLimit ensure the number pods to limit after failure.
- backoffLimit: 2, once pods fails for 2 times it won't create more pods.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  backoffLimit: 2
  template:
    spec:
      containers:
      - name: busybox
        image: busybox
        command: ["sleep", "60"]
      restartPolicy: Never
```

activeDeadlineSecond

- This is used to set the execution time for pod and if pod takes more than this deadline time then pods will be terminated automatically.

apiVersion: batch/v1

kind: Job

metadata:

name: my-job

spec:

activeDeadlineSecond: 20

template:

spec:

containers:

- name: busybox

image: busybox

command: ["sleep", "60"]

restartPolicy: Never

Scheduled / CronJob

apiVersion: batch/v1

kind: CronJob

metadata:

name: hello

spec:

schedule: "* * * * *

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: busybox:1.28

imagePullPolicy: IfNotPresent

command:

- /bin/sh

- -c

- date; echo Hello from the Kubernetes cluster
restartPolicy: OnFailure

1. SuccessfulJobHistoryLimit and FailedJobHistoryLimit

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  successfulJobHistoryLimit: 2
  failedJobHistoryLimit: 1
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox:1.28
              imagePullPolicy: IfNotPresent
              command:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Deployment strategies

Rolling Update

- By default, deployment in k8s uses rolling update strategy. Rolling updates allow Deployments update to take place with zero downtime by incrementally updating Pods instances with new ones.

Rolling updates allow the following actions:

- Promote an application from one environment to another (via container image updates)
- Rollback to previous versions

- Continuous Integration and Continuous Delivery of applications with zero downtime

- To override the default behaviour

spec:

strategy:

type: RollingUpdate

rollingUpdate:

maxSurge: 1

maxUnavailable: 25%

Recreate

The Recreate strategy will bring all the old pods down immediately and then creates new updated pods to match the replica count.

spec:

strategy:

type: Recreate

Blue/Green deployment

Blue-green deployment is a release strategy used in software development and deployment. The approach involves creating two identical environments, one called "blue" and the other "green." At any given time, only one environment is live while the other is idle.

When a new version of the software is ready to be deployed, it is deployed to the idle environment (e.g., green). Once the new version is tested and verified, traffic is routed to the green environment, and the blue environment becomes idle. This allows for a smooth transition without any downtime or interruption of service.

The main benefit of blue-green deployment is that it reduces the risk of downtime during a deployment. If something goes wrong with the new version, the deployment can quickly and easily be rolled back by routing traffic back to the blue environment. Additionally, the idle environment can be used for testing and staging, ensuring that the new version is ready for production before it goes live.

Overall, blue-green deployment is a useful strategy for ensuring continuous delivery of high-quality software with minimal disruption to end-users



Canary release

- A canary release is a software testing technique used to reduce the risk of introducing a new software version into production by gradually rolling out the change to a small subgroup of users, before rolling it out to the entire platform/infrastructure.

Multi master cluster (quorum)

What is the size of the k8s cluster?

- Always the count of master nodes should be an odd number
- We are using load balancer / multiple people are working they may delete the worker nodes so, I never kept exact count of nodes but on average we have 20 to 25 worker nodes.

How many numbers of master nodes are there in your k8s cluster?

- Always tell odd number of nodes (any odd number starting with 3, 5, 7, 9)
- Based on the quorum value we choose only odd number of nodes starting with 3 to achieve better fault tolerance cluster.

Pod Eviction

- Kubernetes evicts pods if the node resources are running out such as CPU, RAM and storage.
- Pod with failed state will be evicted first because they may not run but could still be using cluster resources and then k8s runs decision making based.

Kubernetes looks at two different reasons to make eviction decision:

1. QoS (Quality of Service) class.

For every container in the pod:

- There must be a memory limit and a memory request.
- The memory limit must equal the memory request.
- There must be a CPU limit and a CPU request.
- The CPU limit must equal the CPU request.

2. Priority class.

- A pod's priority class defines the importance of the pod compared to other pods running in the cluster.
- Based on the priority low to high pods will be evicted.

k8s Auto Scale

Horizontal autoscaler

- The Horizontal Pod Autoscaler changes the shape of your Kubernetes workload by automatically increasing or decreasing the number of Pods in response to the workload's CPU or memory consumption, or in response to custom metrics reported
- Automatic scaling of the horizontal pod does not apply to objects that cannot be scaled.

ex: DaemonSets.

Metric server

- The Kubernetes Metrics Server is a cluster-wide aggregator of resource usage data. The Kubernetes Metrics Server collects resource metrics from the kubelet running on each worker node and exposes them in the Kubernetes API server through the Kubernetes Metrics API

```
apiVersion: autoscaling/v2beta2
```

```
kind: HorizontalPodAutoscaler
```

```
metadata:
```

```
  name: php-apache-hps
```

```
spec:
```

```
  scaleTargetRef:
```

```
    apiVersion: apps/v1
```

```
    kind: Deployment
```

```
    name: php-apache
```

```
  minReplicas: 1
```

```
  maxReplicas: 10
```

```
  metrics:
```

```
    - type: Resource
```

```
      resource:
```

```
        name: cpu
```

```
        target:
```

```
          type: Utilization
```

```
          averageUtilization: 50
```

```
----- or -----
```

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

To list HPA

→

kubectl get hpa

Vertical Pod Auto-Scaler (VPA)

- The Vertical Pod Autoscaler automatically adjusts the CPU and memory reservations for your pods to help "right size" your applications. This adjustment can improve cluster resource utilization and free up CPU and memory for other pods

- When we describe vpa, it will show recommendations for the Memory/CPU requests, Limits and it can also automatically update the limits.

```
apiVersion: autoscaling.k8s.io/v1
```

```
kind: VerticalPodAutoscaler
```

```
metadata:
```

```
  name: my-app-vpa
```

```
spec:
```

```
  targetRef:
```

```
    apiVersion: "apps/v1"
```

```
    kind: Deployment
```

```
    name: my-app
```

```
  updatePolicy:
```

```
    updateMode: "Auto"
```

Horizontal / Vertical Cluster Auto-Scaler

- Cluster Autoscaler is a tool that automatically adjusts the size of the Kubernetes cluster when one of the following conditions is true:

1. Some pods failed to run in the cluster due to insufficient resources,
2. Some nodes in the cluster that have been overloaded for an extended period and

their pods can be placed on other existing node

- Cluster autoscaler tools are mostly provided by public cloud providers.

Common Reasons Kubernetes Deployments Fail

1. Wrong Container Image / Invalid Registry Permissions
2. Application Crashing after Launch
3. Missing ConfigMap or Secret
4. Liveness/Readiness Probe Failure
5. Exceeding CPU/Memory Limits
6. Resource Quotas

7. Insufficient Cluster Resources
8. Persistent Volume fails to mount
9. Validation Errors
10. Container Image Not Updating

Docker Swarm vs Kubernetes

Features	Kubernetes	Docker Swarm
Installation & Cluster Configuration	Installation is complicated; but once setup, the cluster is very strong	Installation is very simple; but cluster is not very strong
GUI	GUI is the Kubernetes Dashboard	There is no GUI
Scalability	Highly scalable & scales fast	Highly scalable & scales 5x faster than Kubernetes
Auto-Scaling	Kubernetes can do auto-scaling	Docker Swarm cannot do auto-scaling
Load Balancing	Manual intervention needed for load balancing traffic between different containers in different Pods	Docker Swarm does auto load balancing of traffic between containers in the cluster
Rolling Updates & Rollbacks	Can deploy Rolling updates & does automatic Rollbacks	Can deploy Rolling updates, but not automatic Rollbacks
Data Volumes	Can share storage volumes only with other containers in same Pod	Can share storage volumes with any other container
Logging & Monitoring	In-built tools for logging & monitoring	3rd party tools like ELK should be used for logging & monitoring

Interview Questions on Kubernetes

Why do we need Kubernetes?

- It is the orchestration tool which maintains the container, the main purpose is when the load an Yaml file Kubernetes will try to manage the same configuration specified in manifest file even though if there is change in cluster or configuration, Kubernetes will keep watching the cluster to maintain the desired state
- Controls and automates the deployments and updates
- Provide security services on storage, users, networking
- K8s is self-monitoring which constantly check the health of the nodes
- Provides feature of horizontal scaling and vertical
- Automatic the rollout and rollback

What happens when one of the k8s nodes fails?

- Initially the node specifies the state has not ready.
- Status of the running pods on the failed node will change to a either unknown

- Then the pod eviction method takes place
- Kubernetes automatically evicts the pods and then try to recreate the new one with the old Volume.
- The deployment controller terminates pods running on the failed node and recreate the pods in the available worker node.
- In case evicted pod get stuck in the terminating state & the attached Volumes cannot be released/ reused, then newly created pod is stucked in the container create state
- So, we need to forcefully delete the stuck pods Or k8s will take couple of minutes (6m) to delete the volume attachment objects associated with the pods & finally detach the volume from the lost node

Liveness probes in Kubernetes

Let say we have an microservice that written in a Go and this microservice have some bugs on some part of the code which causes a freeze in a run time, to avoid hitting the bug, we can configure a liveness probe to determine if the microservice is in a frozen state the microservice container will be restarted and come to normal condition.

or

Suppose the pod is running with the application inside the container due to some reasons memory leak, CPU usage the application is not responding to our requests and stuck with the error state

Here the liveness proxy checks for the health condition. if the health is not in good condition kubelet tries to restarts the container.

Pod Eviction:

Kubernetes evicts the pod if the node is running out of resources such as CPU, RAM and Storage

Failed pods will be evicted first because pod is not running in the cluster and simply it is consuming the resources.

k8s looks for a different reason to make an eviction.

1. QDs (Quality of service): - For every container in the pod there should be a memory. Limit and Memory request and these limit & request should be equal for CPU

2. Priority class:

based on the priority low or high the pod will be get evicted, priority class defines the soup of pod compared to other pods running in the cluster

Pod is getting restarting, what is the problem? How do you trouble shoot?

- 1) Pod node is unavailable so it shows as restarting
- 2) Resource usage is not configured properly: - Suppose we have allocated 600mb of memory for a container and it tries to allocate the more than the limit

Uses cases of Init container

1. That can give access to the secrets that app container cannot access
2. Init container is used to start the service that used by app container
3. Init container which delays for few seconds to see the dependencies are running or not and then starts the main app container.

Side Car Container use case

Considering we have webserver container having / running the nginx image the access and errors logs produced by the webserver are not enough to place on the persistent volume, however the developer needs access to the 24 hrs. of logs in order to trace the bugs & issues. So, we need to ship the access & logs to the log-aggregator service, so here we implement the sidecar container by deploying as a 2nd container that ships the access & logs from nginx to the monitoring services

Since Containers are running in the same pod we can use shared empty Dir Volume to read & write logs

How do you know if the pod is running?

Kubectrl describe pod < pod name> -- namespace

kube-system status field should be running any other status will indicate the issues with the environment

How do i fix the pod which is there in a pending state

1. Gather the information where the root cause occurs.
Kubectrl describe pod -n namespace – p pod name
kubectrl describe nodes
2. Examine pod events output.
3. Check the kubelet logs
4. Checking the kubelet is running or not
5. Debug pulling image
6. Check component status

How to debug when Kubernetes nodes are in not ready state?

1. Describe the nodes. which is in not ready state look for the conditions, capacity and allocations
2. Ssh to that node
3. Check whether the kubelet is running or not
4. Make sure that the docker is running or not
5. Check the logs in depth using command `journalctl -u kubelet` (what is the error we can see)
6. After fixing reset kubelets & docker
7. Make sure that the node has enough space and CPU utilization.

How to Troubleshoot Pod Errors?

`kubectl get pods`

`Kubectl describe pod name`

May be certificate is invalid

When it is not able to pull an image

How to reboot/restart the particular node in k8s

Before restarting it is recommended to take a backup of etcd data to avoid the data loss

Login to worker node that you want to restart as a root

Jump to bin dir and stop the k8s service & unmount the k8s volume

If the node contains the database reboot it

Restart the node by command `reboot`

If restart fails perform the hard reboot

How to establish communication between the parts across the different name spaces

By default coordinates allows any part to communicate with each other with different name spaces and worker nodes

We can use network policies to apply deny all for cluster and we can write policies to allow only required request to cluster

Network policy is defined only for ingress and egress

What is service endpoint

Which maintains the records of IP address of the pod that able to communicate with each other

Kubectl get endpoints

Or

Kubectl describe endpoints

Container port

It is a port in which app can be reached out inside the container

Target port

It is a port that exposed inside the cluster and service which connects to the pod to the other services

What if the pod continuously restarting

May be Container restarts if the liveness props fail

Describe the part and check the errors in the event section and check the logs of the pod and get the reason

Kubernetes Security Best Practices

- RBAC
- K8S advanced security modules
- Pod security policy
- Network policy

Kubectl get nodes

The connection to the server localhost:8080 was refused - did you specify the right host or port?

This is because if we don't see kube config

To check

kubectl config current-context

Kubectl config view

Describe the node

Describe the pod

When I type kubectl get pods the output is not showing

There are several reasons why the kubectl get pods command might not show any output:

- No pods are running: If there are no pods currently running in the cluster, the `kubectl get pods` command will not show any output.
- Incorrect namespace: If the pods you are trying to access are in a different namespace than the one currently selected in your `kubectl` configuration, you will need to specify the correct namespace using the `-n` flag. For example, `kubectl get pods -n mynamespace`.
- No permissions: If you do not have the appropriate permissions to access the pods, the `kubectl get pods` command will not show any output. Check with your cluster administrator to ensure that you have the necessary permissions.
- Incorrect label selector: If you are using a label selector to filter the pods, make sure that the selector is correct and matches the labels assigned to the pods.
- Connection issues: If there are connection issues between your `kubectl` client and the Kubernetes API server, the `kubectl get pods` command may not work. Check your network connection and ensure that you are using the correct Kubernetes configuration.
- Name of pods is incorrect: If you are trying to get pods by name, make sure that the name is spelled correctly and matches the pods in the cluster.