**What is container orchestration?**

Container orchestration automates the deployment, management, scaling, and networking of containers.

**Why Do We Need Container Orchestration?**

1. Provisioning and deployment
2. Configuration and scheduling
3. Resource allocation
4. Container availability
5. Scaling or removing containers based on balancing workloads across your infrastructure
6. Load balancing and traffic routing
7. Monitoring container health
8. Keeping interactions between containers secure

**Container orchestration tools**

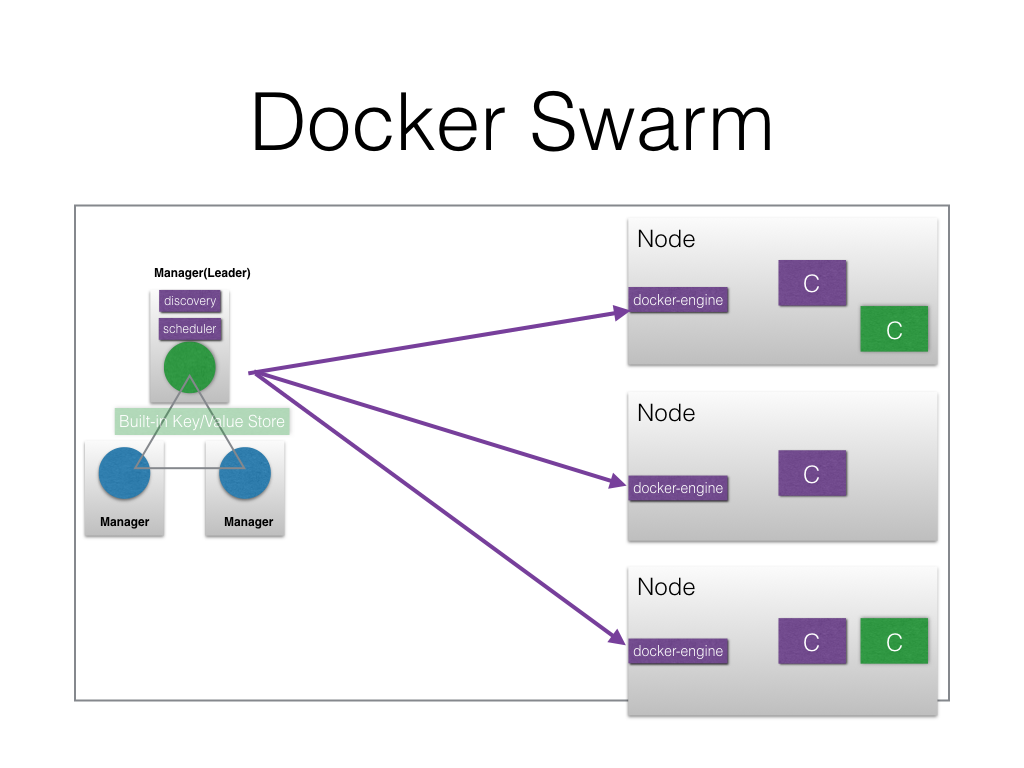
Container orchestration tools provide a framework for managing containers and microservices architecture at scale.

There are many container orchestration tools that can be used for container lifecycle management. Some popular options are **Kubernetes, Docker Swarm,** and **Apache Mesos.**

**Kubernetes vs. Docker Swarm**



**Docker Swarm Cluster Architecture**



**Swarm mode key concepts**

* A swarm consists of multiple Docker hosts which run in **swarm mode** and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles.
* When you create a service, you define its optimal state (number of replicas, network and storage resources available to it, ports the service exposes to the outside world, and more).
* A **service** is the definition of the tasks to execute on the manager or worker nodes. When you create a service, you specify which container image to use and which commands to execute inside running containers.
* A **task** carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.
* The swarm manager uses **ingress load balancing** to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a **PublishedPort** or you can configure a PublishedPort for the service. You can specify any unused port. If you do not specify a port, the swarm manager assigns the service a port in the 30000-32767 range.

https://docs.docker.com/engine/swarm/

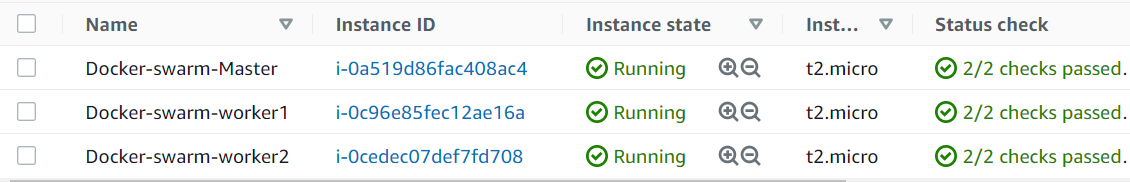
https://docs.docker.com/engine/swarm/key-concepts/

**Demo On Docker Swarm Cluster**

**Prerequisites:**

* Ubuntu 64-bit operating system / You can take any Linux OS.
* In Base OS, Docker must be installed and running

**Step-1 Installing Docker on all nodes:**

****

#sudo apt-get update

#sudo apt install docker.io -y

#systemctl start docker

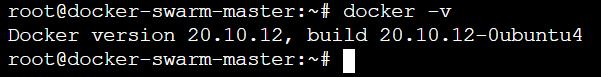
#systemctl enable docker

#systemctl status docker

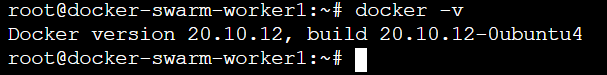
#docker info

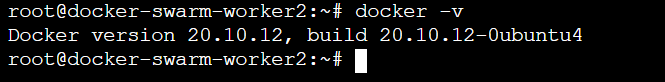
#docker version

#docker –v

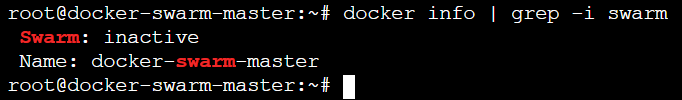


* Same steps have to perform on all node.



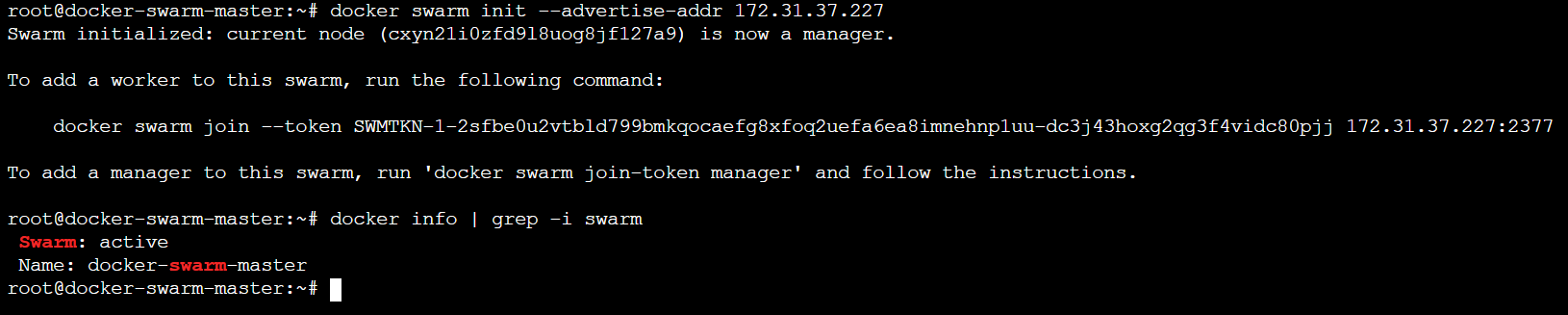


**Step-2 Enable Docker Swarm Mode:**

****

# docker swarm init --advertise-addr <ip-address>

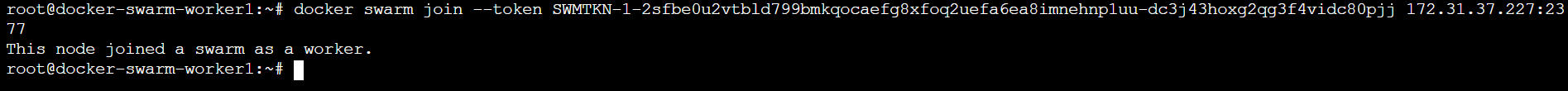
# docker swarm init



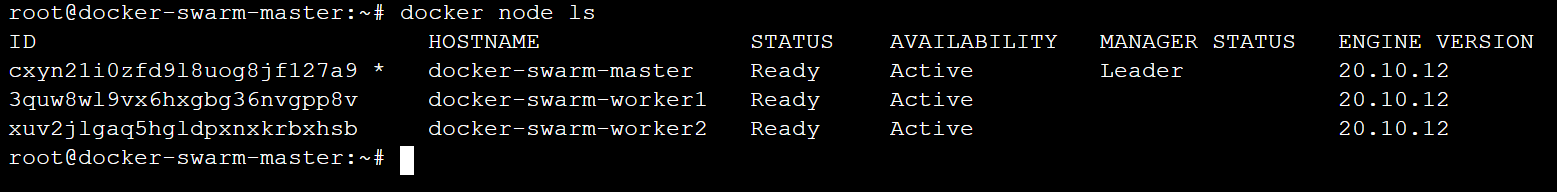
**Note:** Before adding nodes into the cluster make sure the port no: 2377 should be allowed in firewall.

**Adding nodes into the cluster:**

docker swarm join --token SWMTKN-1-2sfbe0u2vtbld799bmkqocaefg8xfoq2uefa6ea8imnehnp1uu-dc3j43hoxg2qg3f4vidc80pjj 172.31.37.227:2377



# docker node ls



you can run the following command on a manager node to retrieve the join command for a worker and Manager:

# docker swarm join-token worker

# docker swarm join-token manager

**Now you can play with Docker Swarm:**

1. **Creating Services:**

Syntax:

docker service create --name <NAME> --replicas <#> -p <HP:CP> IMAGE

Example:

#docker service create --name web -p 80:80 nginx

#docker service create --name web -p 80:80 --replicas 3 nginx

Scaling up and down:

#docker service scale service-name=<no. of replicas>

#docker service scale web=3

Get details of service:

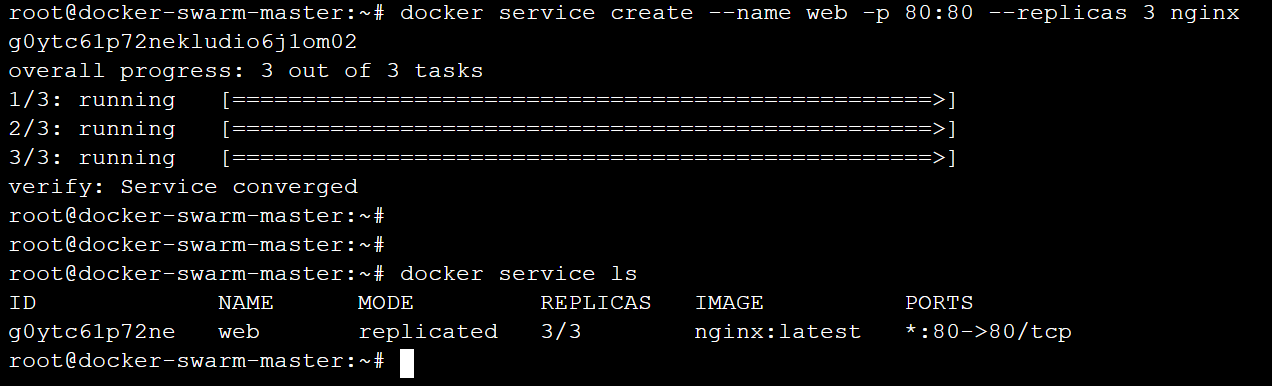
#docker service ls

#docker service ps web(service-name)

#docker service inspect --pretty <service-name>

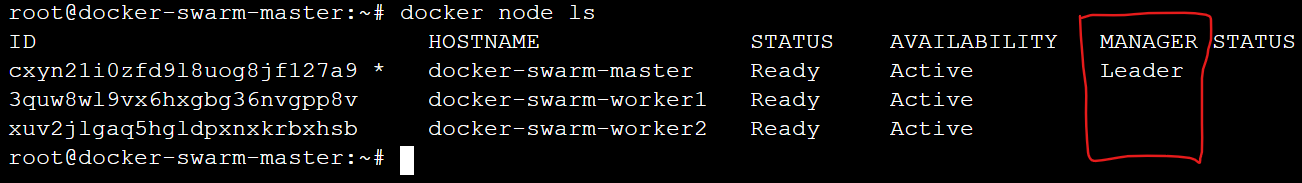
Deleting a service

#docker service rm <service-name>



1. **Node Status:** identifies the nodes participation in the swarm management.

|  |  |
| --- | --- |
| Status | Nodes Participation |
| BLANK | Worker Node. |
| LEADER | Manager Node (Primary). |
| REACHABLE | Candidate to be leader node (Promote). |
| UNAVAILABLE | Manager node that cannot communicate with other nodes. |
| DRAIN | Do not assign/create containers |



#docker node inspect --pretty <NODE>

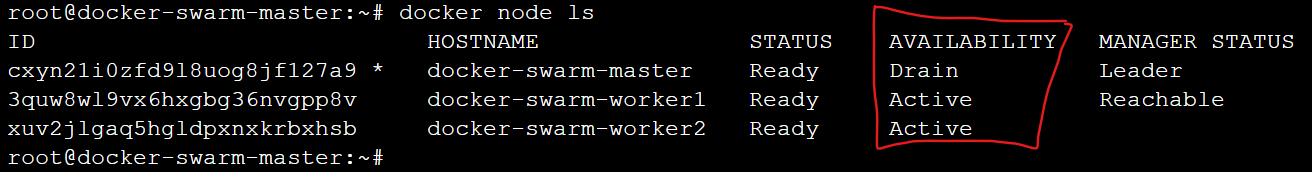
#docker node promote <NODE>

#docker node demote <NODE>

# docker node update --availability drain <NODE>

# docker node update --availability drain docker-swarm-master

# docker node update --availability active <NODE>



1. **Rolling updates:**

* While updating a service, you can define how many containers should be updated at a time, and what happen if new containers start falling.
* Rolling updates are designed to update your workloads without downtime.

#docker service update --image <IMAGE> <service>

#docker service update --replicas=5 <service>

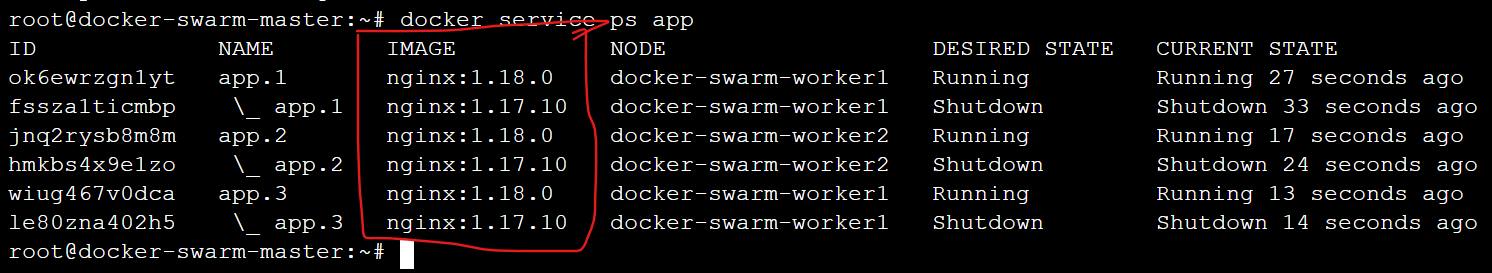
#docker service update --rollback <service>

#docker service update --update-failure-action=rollback <service>

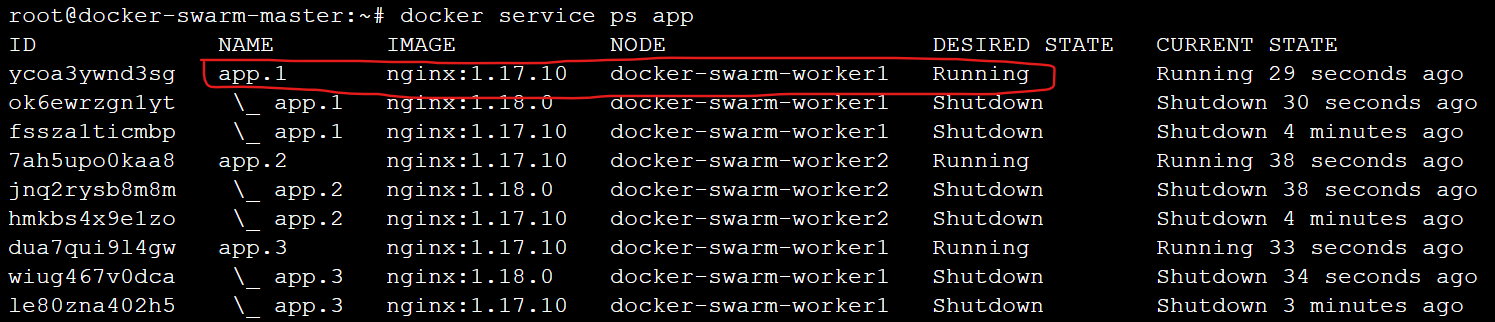
**Take a Demo**:

#docker service create --name app -p 80:80 --replicas 3 nginx:1.17.10

# docker service update --image nginx:1.18.0 app



# docker service update --rollback app



**Service Mode:**

**Replicated:**

* A replicated service specifies a number of identical tasks you want to run.
* It is default service mode.

**Global:**

* A global service runs one replica per node on all the nodes in the swarm, with no pre-specified number of tasks/nodes.
* Every new node added will get the replica created and vice versa.
* Example: when we want to run some daemons on each node like: Logstash, Datadog, Prometheus etc.

# docker service ls

# docker service create -p 80:80 --name web --mode global nginx

https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/

**Deploy a stack to a swarm**

When running Docker Engine in swarm mode, you can use **“docker stack deploy”** to deploy a complete application stack to the swarm.

The **deploy command** accepts a stack description in the form of a Compose file.

docker stack deploy command supports any Compose file of version “3.0” or above.

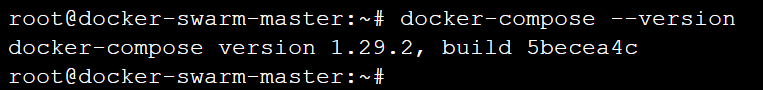
Docker compose should be installed.

**Docker compose installation:**

# sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

# sudo chmod +x /usr/local/bin/docker-compose

# docker-compose –version



# docker stack deploy –c docker-compose.yml <STACK-NAME>

# docker stack ls

# docker stack ps <STACK-NAME>

# docker stack services <STACK-NAME>

# docker stack rm <STACK-NAME>

**Take a simple demo:**

1. Write a simple compose file.

---

version: "3.4"

services:

demoweb:

image: nginx

ports:

- 80:80

volumes:

- /tmp:/usr/share/nginx/html

deploy:

mode: replicated

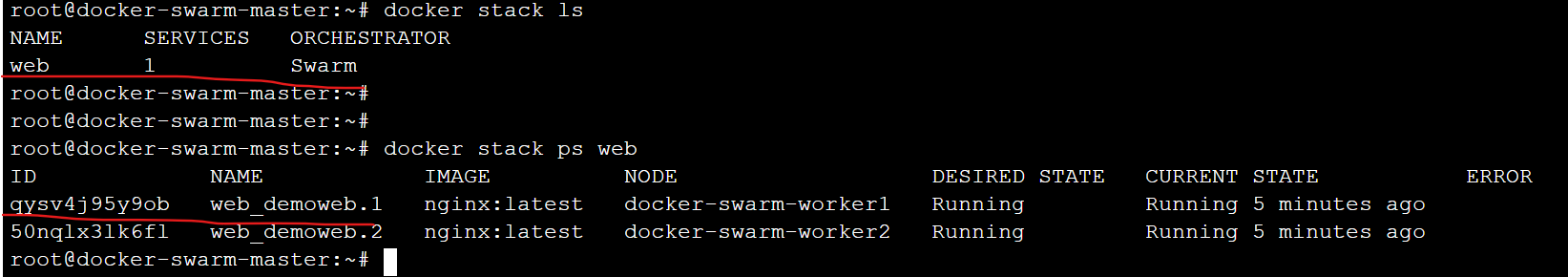
replicas: 2

save as docker-compose.yml

1. Create a stack named as “web”.

# docker stack deploy -c docker-compose.yml web(stack-name)

# docker stack ls



# docker stack ps web

# docker stack services web

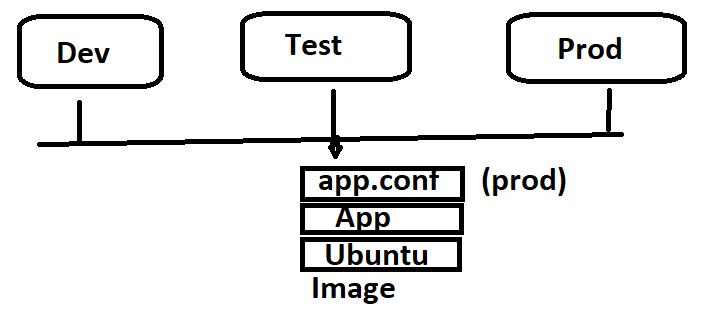
# docker stack rm web

https://docs.docker.com/engine/swarm/stack-deploy/

**Service Discovery and Load balancing:**

* The swarm manager use ingress load balancing to expose the service, you want to make available externally to the swarm.
* The ingress controller will do round robin by default.

**Docker config**



* Allow you to store non-sensitive data, such as configuration files in the cluster.
* Use a Docker Config file to keep your application code separate from your configuration.

**Take a demo:**

1. Create a sample file

#echo "Testing of config file" > httpd.conf

1. #docker config create testconfig httpd.conf

#docker config ls

1. Create a service

# docker service create --name app --config src=testconfig,target=/var/www/html/ -p 80:80 httpd

**Container Health Check**

Health check is used to determine the health of a running container.

While working with Docker, two ways of defining a health check exist:

* Dockerfile
* Docker-Compose file.

The Docker Compose Healtcheck contains five properties:

**test:** This property specifies the command that will be executed and is the health check of the container. This command HAS TO be available and working if everything is set up correctly.

**interval:** This property specifies the number of seconds to initially wait before executing the health check and then the frequency at which subsequent health checks will be performed.

**timeout:** This property specifies the number of seconds Docker a waits for your health check command to return an exit code before declaring it as failed.

**retries:** This property specifies the number of consecutive health check failures required to declare the container as unhealthy.

**start\_period:** This property specifies the number of seconds your container needs to bootstrap. During this period, health checks with an exit code greater than zero won’t mark the container as unhealthy; however, a status code of 0 will mark the container as healthy.

References:

<https://howchoo.com/devops/how-to-add-a-health-check-to-your-docker-container#:~:text=Add%20a%20health%20check%20to%20the%20Dockerfile,-Since%20the%20goal&text=So%20add%20this%20line%20to,the%20last%20line%20(CMD).&text=In%20this%20case%2C%20we%20are,more%20information%20on%20the%20options>.

<https://medium.com/geekculture/how-to-successfully-implement-a-healthcheck-in-docker-compose-efced60bc08e>

Take a simple demo:

1. Create docker compose file as docker-compose.yml

---

version: "3.4"

services:

demoweb:

image: nginx

ports:

- 80:80

deploy:

mode: replicated

replicas: 2

healthcheck:

test: curl http://localhost

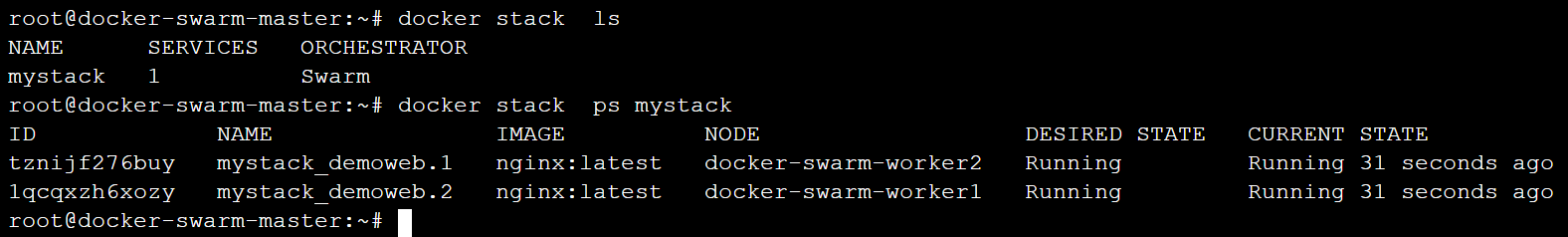
interval: 10s

timeout: 10s

retries: 3

start\_period: 10s

# docker stack deploy -c docker-compose.yml mystack



# docker service ls

# docker service ps mystack\_demoweb

