Docker

|  |  |
| --- | --- |
| Docker Compose | Docker Swarm |
| Compose: is used to control multiple containers on a single system. Much like the Dockerfile we looked at to build an image, there is a text file that describes the application: which images to use, how many instances, the network connections, etc. But Compose only runs on a single system so while it is useful, we are going to skip Compose[1](https://training.play-with-docker.com/ops-s1-swarm-intro/#fn-compose) and go straight to Docker Swarm Mode. | Swarm:  tells Docker that you will be running many Docker engines and you want to coordinate operations across all of them. Swarm mode combines the ability to not only define the application architecture, like Compose, but to define and maintain high availability levels, scaling, load balancing, and more. With all this functionality, Swarm mode is used more often in production environments than it’s more simplistic cousin, Compose. |
|  |  |

# Application example

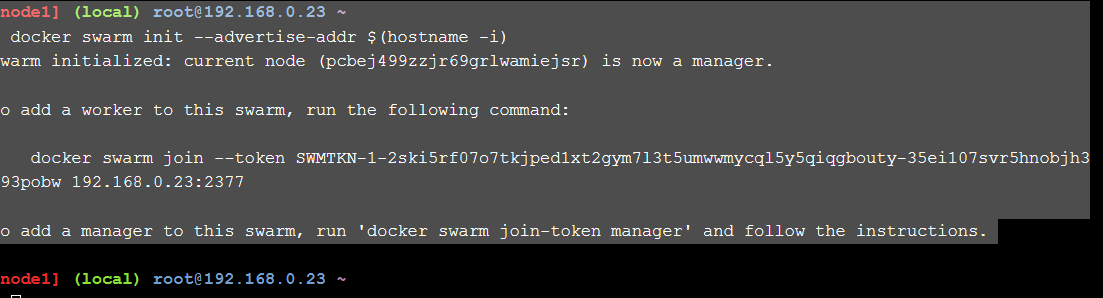
Voting application

## Initialize Your Swarm

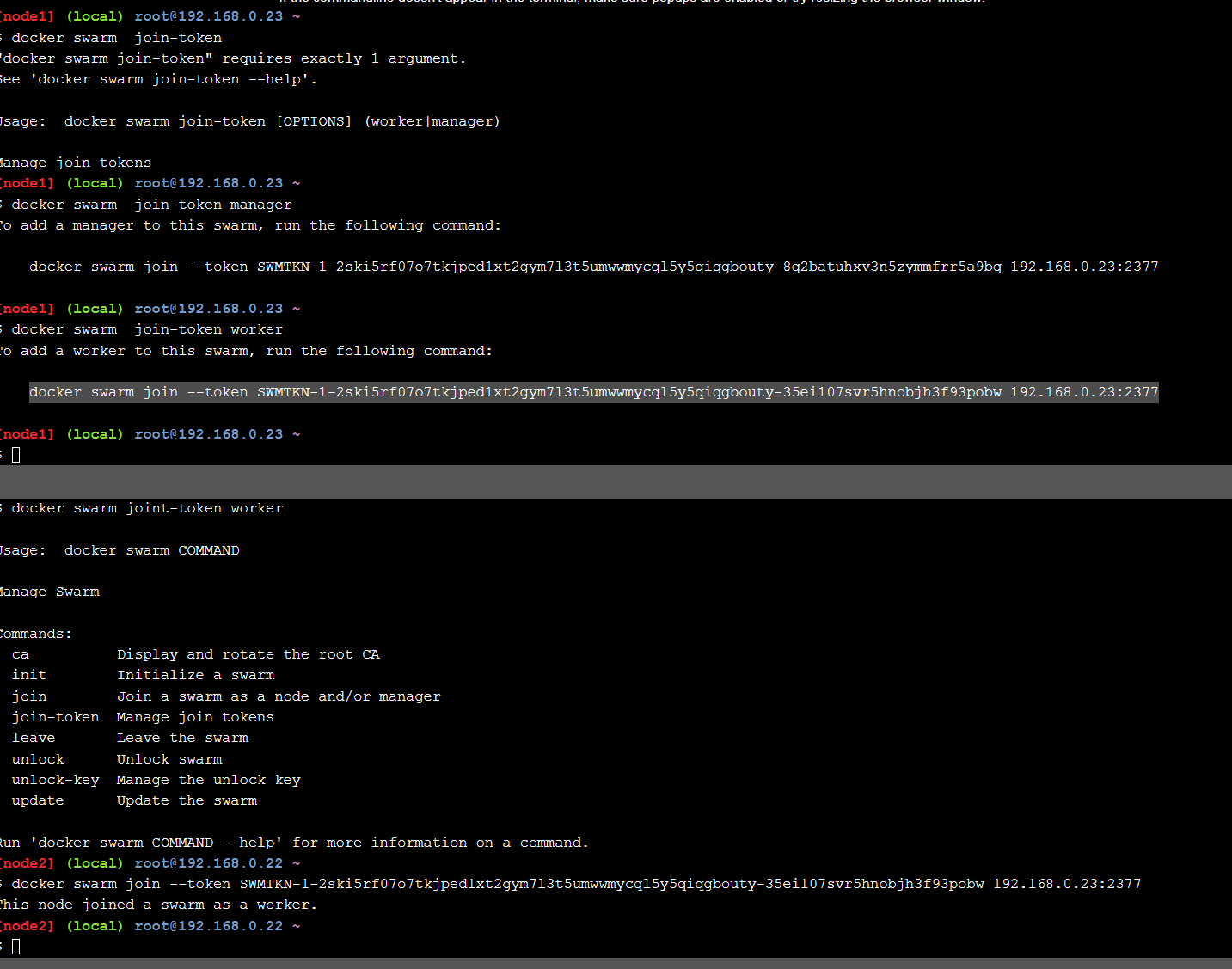
Most production swarms have at least three manager nodes in them and many worker nodes. Three managers is the minimum to have a true high-availability cluster with quorum. Note that manager nodes can run your container tasks the same as a worker node, but this functionality can also be separated so that managers only perform the management tasks.

Command to initialize Docker Swarm Mode:

$docker swarm init –advertise-addr ${hostname –i)



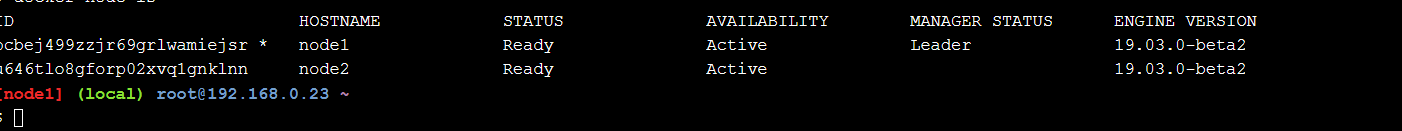
Swarm manger is created and is listening on the IP address returned by the (hostname –i) command. In the 2nd screen, run add worker node.



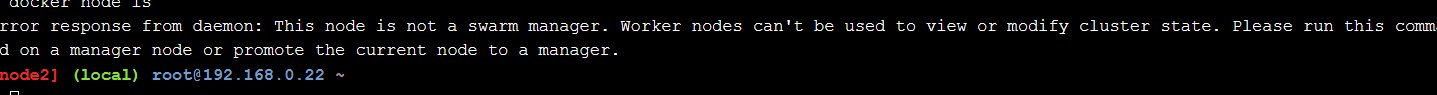
Now we have one manger and one worker node.

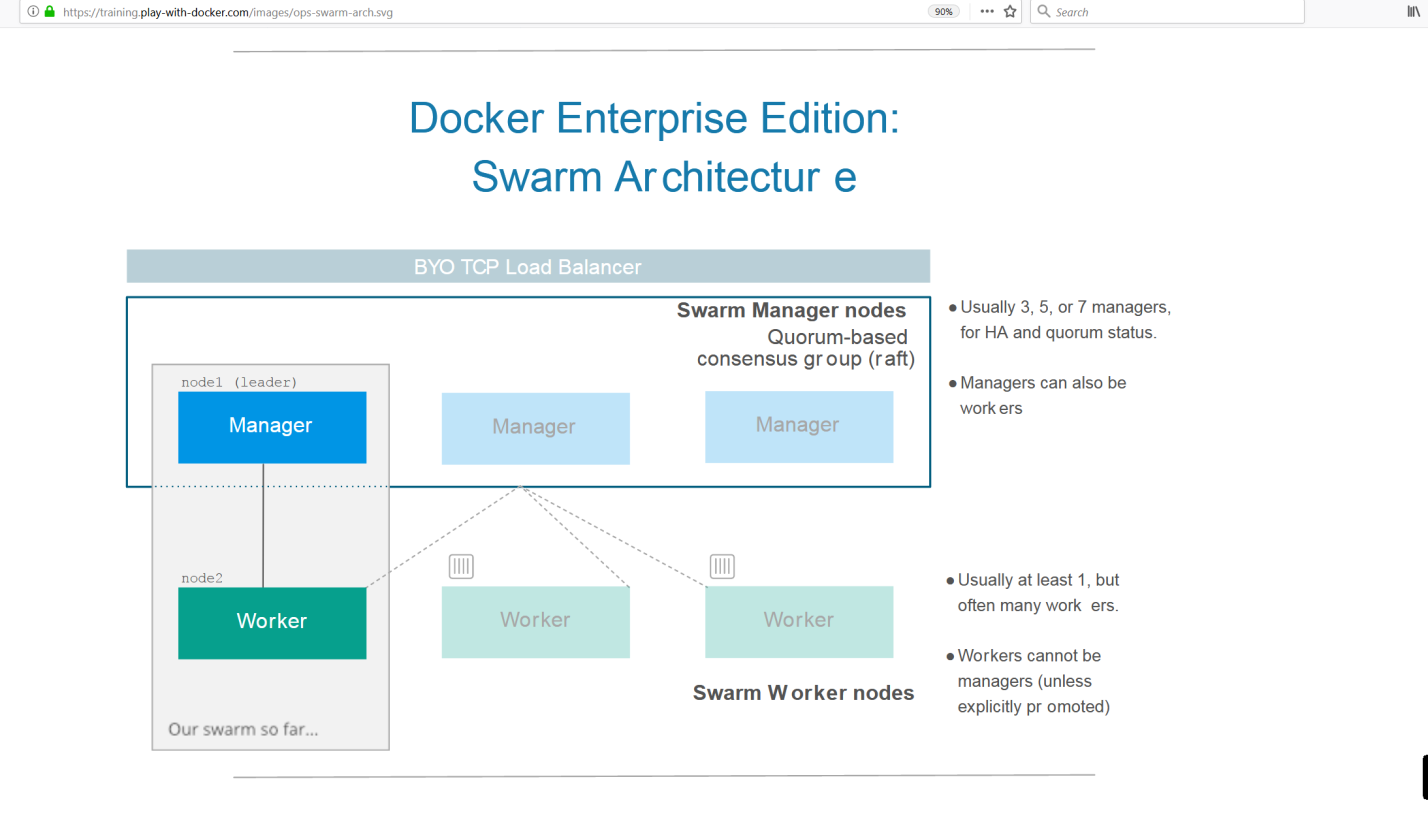
## Show swarm members

From manager node , to run $docker node ls (can’t run on worker node), the manager node is ‘Leader’ (the main control node for all the managers, if the leader goes down for some reason, the other manager nodes will elect a new leader, why we need multiple manager nodes) ,as we only have one manager node.



Run from worker node:





## Clone the Voting App

$git clone <https://github.com/docker/example-votinh-app>

#cd example-voting-app

## Deploy a Stack

A stack is a group of services that are deployed together: multiple containerized components of an application that run in separate instances. Multiple services app running on a swarm.

Each individual service can actually be made up of one or more containers, called tasks and then all the tasks & services together make up a stack

The file that defines a stack is a plain text file that is easy to edit and track.

$cat docker-stack.yml

The YAML file defines the entire stack(kind of source code for our application design):

* The architecture of the services: services, these are the individual application components, in this application , we have redis, db, vote, result, worker, and visualizer as our services, beneath each service are lines that specify how that service should run
  + Image: this is the container image to use for a particular service
  + Ports and Networks , can be privately used within the stack or they can allow external communications to and from a stack
  + Replicas: Indicate the number of instances, or tasks, of this service that the Swarm managers should start when the stack is brought up. The Docker engine is intelligent enough to automatically load balance between multiple replicas using built-in load balanceers.
* Number of instances
* How everything is wired together
* How to handle updates to each service

|  |
| --- |
| version: "3"  services:  redis:  image: redis:alpine  networks:  - frontend  deploy:  replicas: 1  update\_config:  parallelism: 2  delay: 10s  restart\_policy:  condition: on-failure  db:  image: postgres:9.4  volumes:  - db-data:/var/lib/postgresql/data  networks:  - backend  deploy:  placement:  constraints: [node.role == manager]  vote:  image: dockersamples/examplevotingapp\_vote:before  ports:  - 5000:80  networks:  - frontend  depends\_on:  - redis  deploy:  **replicas: 2**  update\_config:  parallelism: 2  restart\_policy:  condition: on-failure  result:  image: dockersamples/examplevotingapp\_result:before  ports:  - 5001:80  networks:  - backend  depends\_on:  - db  deploy:  replicas: 1  update\_config:  parallelism: 2  delay: 10s  restart\_policy:  condition: on-failure  worker:  image: dockersamples/examplevotingapp\_worker  networks:  - frontend  - backend  deploy:  mode: replicated  replicas: 1  labels: [APP=VOTING]  restart\_policy:  condition: on-failure  delay: 10s  max\_attempts: 3  window: 120s  placement:  constraints: [node.role == manager]  visualizer:  image: dockersamples/visualizer:stable  ports:  - "8080:8080"  stop\_grace\_period: 1m30s  volumes:  - "/var/run/docker.sock:/var/run/docker.sock"  deploy:  placement:  constraints: [node.role == manager]  networks:  frontend:  backend:  volumes:  db-data: |

In node1:

To deploy stack from a command line, using the compose file format, be used to manage services over multiple nodes

$ docker stack deploy --compose-file=docker-stack.yml

Creating network voting\_stack\_backend

Creating network voting\_stack\_frontend

Creating network voting\_stack\_default

Creating service voting\_stack\_worker

Creating service voting\_stack\_visualizer

Creating service voting\_stack\_redis

Creating service voting\_stack\_db

Creating service voting\_stack\_vote

Creating service voting\_stack\_result

[node1] (local) root@192.168.0.12 ~/example-voting-app

6 services of the voting app’s stack have been deployed:

$docker stack ls

$ docker stack ls

NAME SERVICES ORCHESTRATOR

voting\_stack 6 Swarm

[node1] (local) root@192.168.0.12 ~/example-voting-app

Get detail of services within the stack

$docker stack services voting\_stack

$ docker stack services voting\_stack

ID NAME MODE REPLICAS IMAGE PORTS

5cx9z9iq0s0y voting\_stack\_result replicated 1/1 dockersamples/examplevotingapp\_result:before \*:5001->80/tcp

dc606g5tf09i voting\_stack\_db replicated 1/1 postgres:9.4

nof8wiktsoj2 voting\_stack\_worker replicated 1/1 dockersamples/examplevotingapp\_worker:latest

v28pj4108zub voting\_stack\_redis replicated 1/1 redis:alpine

vja6iwv3xzvy voting\_stack\_vote replicated 2/2 dockersamples/examplevotingapp\_vote:before \*:5000->80/tcp

vks8n46c3969 voting\_stack\_visualizer replicated 1/1 dockersamples/visualizer:stable \*:8080->8080/tcp

[node1] (local) root@192.168.0.12 ~/example-voting-app

$ docker service ps voting\_stack\_vote : list the tasks of the vote service

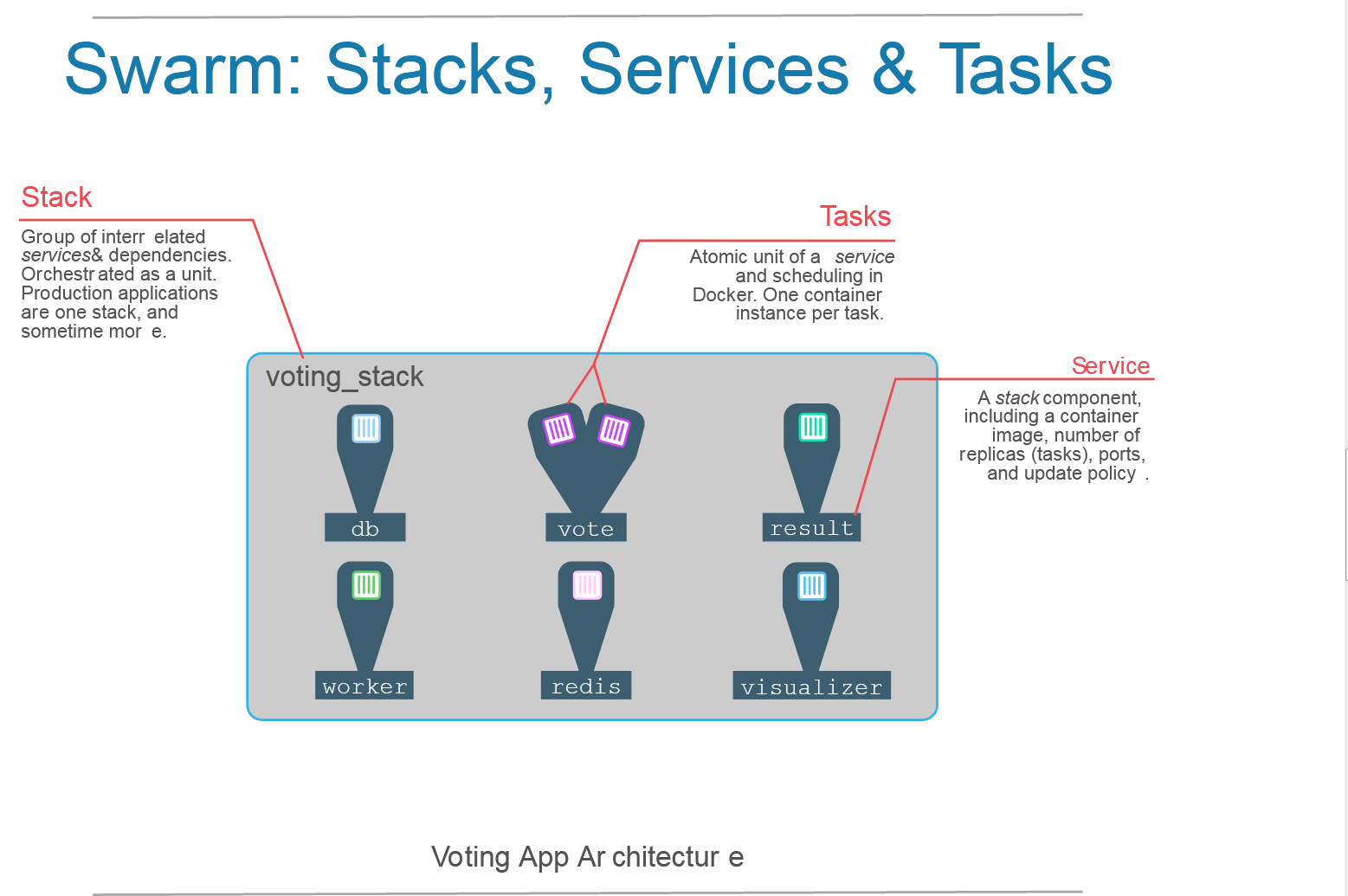
$ docker service ps voting\_stack\_vote

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

ouzkofnv834b voting\_stack\_vote.1 dockersamples/examplevotingapp\_vote:before node2 Running Running 8 minutes ago

n13nn7ltdpmo voting\_stack\_vote.2 dockersamples/examplevotingapp\_vote:before node1 Running Running 8 minutes ago

SWARM visualizer to show you how the app is setup and running. Stacks, Services and Tasks



## Scaling An Application

$docker service scale voting\_stack\_vote=5

$ docker service scale voting\_stack\_vote=5

voting\_stack\_vote scaled to 5

overall progress: 2 out of 5 tasks

1/5: pending

2/5: running

3/5: running

4/5: pending

5/5: pending

[node1] (local) root@192.168.0.12 ~/example-voting-app

$docker service ps voting\_stack\_vote

$ docker service ps voting\_stack\_vote

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

ouzkofnv834b voting\_stack\_vote.1 dockersamples/examplevotingapp\_vote:before node2 Running Running 17 minutes ago

n13nn7ltdpmo voting\_stack\_vote.2 dockersamples/examplevotingapp\_vote:before node1 Running Running 17 minutes ago

ijuw4y9lnm9p voting\_stack\_vote.3 dockersamples/examplevotingapp\_vote:before node2 Running Running about a minute ago

ybcdxwyyatpz voting\_stack\_vote.4 dockersamples/examplevotingapp\_vote:before node1 Running Running about a minute ago

y9hbx4i2quy6 voting\_stack\_vote.5 dockersamples/examplevotingapp\_vote:before node2 Running Running about a minute ago

[node1] (local) root@192.168.0.12 ~/example-voting-app

## Summary:

Docker swarm can easily and quickly scale your application’s ervices up and down as needs require. Again , in many situations you would probably want to automate this rather than manually scaling, which is easy through the Docker APIs. Can also swap out the built-in load balance for something with additional controls.