Docker Certified Associate

Exam Info

55 questions - 13 multiple choice + 42 discrete option multiple choice (DOMC) questions.

Duration: 90 mins

Study Guide - https://docker.cdn.prismic.io/docker/3f8ef3b3-87f1-47c7-b820-0e0a8bb308d4_DCA_study_guide_+v1.1+pdf.pdf

Basics

An image is an executable package that includes everything needed to run

an application - the code, a runtime, libraries, environment variables,

and configuration files.

A container is a runtime instance of an image.

Images are made of multiple read-only layers. When an image is instantiated a top writable layer is created (which is deleted when container is removed).

docker image pull alpine

docker image ls or docker images

docker image inspect

docker image rm <image ID>

Docker (image and registry specs) now supports multi-architecture images. This means a single image (repository:tag) can have an image for Linux on x64, Linux on PowerPC, Windows x64, ARM etc

Shortcut to delete all images:

docker image rm $(docker image ls -q) -f ==> deletes all images

docker image prune -a ==> deletes dangling and images not used by any containers.

Containers:

Start a new container.

docker container run --name percy -it ubuntu:latest /bin/bash

Cntl + P + Q - to exit the container without killing it.

In a standard, out-of-the-box Linux installation, the Docker daemon implements the Docker Remote API on a local IPC/Unix socket at **/var/run/docker.sock .**

List all running Containers:

docker container ls

or docker ps

docker container exec -it <containerId/name> bash -- reattach to the container.

docker container stop <containerID/Name>

docker container ls -a -- will list even stopped containers.

docker container start <containerID/Name>

Remove the container:

docker container rm <containerID/Name>

Restart policies:

always, unless-stopped, on-failed

Running container in interactive mode with restart policies:

$ docker container run --name neversaydie -it --restart always alpine sh

-d - detached mode (as daemon)

Run container as a demon with -d option:

docker container run -d --name webserver -p 80:8080 nigelpoulton/pluralsight-docker-ci

docker container inspect <containerID/Name>

To delete all containers:

docker container rm $(docker container ls -aq) -f

Swarm:

Engines in a swarm run in swarm mode.

Manager nodes maintain swarm - only one is leader but there could be more than 1 manager nodes

Raft consensus algorithm is used to decide which of the manager nodes is the leader.

Worker nodes execute tasks

Services - declarative way of running and scaling tasks

E.g. docker service create --name web-fe --replicas 5

Docker Trusted Registry (DTR) - private registry that comes with Docker EE

Docker hub is default for docker login command unless we specify another registry.

Images in the registry are referenced as:

<registry>/<repo>:<image or tag>

For e.g. docker.io/redis:latest

Top level image: docker.io/redis or just redis

docker image pull docker.io/redis:latest

is same as

docker image pull docker.io/redis

is same as

docker image pull redis

where,

docker.io - registry

redis - repo

latest - image (or tag)

docker image pull nigelpoulton/tu-demo -a

-a is to pull all images (or tags) in the repo

Unofficial images: docker.io/repo/<image-tag>

docker.io/nigelpoulton/tu-demo:v1

where,

docker.io - registry

nigelpoulton/tu-demo - repo

v1 - image tag

Same image can have multiple tags.

latest - is a conventional (manually created) tag which does not always mean latest version of an image.

When we push images to registry we compress each layer. This changes the SHA IDs of the layers. So each layer has 2 SHA ids:

- **content hash -** when layer is uncompressed

- **distribution hash** - when layer is compressed before being pushed to registry

Best Practice:

- smaller image is better

- prefer offical repo as base

- dont use latest tag - use exact version tag instead.

Images are immutable.

If we ssh to container and write into a file, there is a read-writable layer created (copy-on-write) on the fly. This layer is not part of the image and hence if we restart the container, the changes will be lost.

Containerizing an Application

App -> Dockerfile -> docker image build -> image

Dockerfile:

FROM - base image (always first instruction)

LABEL is a config

RUN, COPY, ADD - create layers

EXPOSE, WORKDIR, ENTRYPOINT are metadata that go in image config JSON

ENTRYPOINT = default app to run when container is run.

2 ways to specify default processes for new containers:

CMD - runtime args override other CMD instructions (so last one in the Dockerfile is what gets executed)

ENTRYPOINT - runtime args are appended to ENTRYPOINT

E.g. Dockerfile

```
FROM java:8-jdk-alpine
COPY . .
WORKDIR .
RUN javac Hello.java

ENTRYPOINT ["java", "Hello"]
```

Hello.java:

```
public class Hello {
	public static void main(String[] args) {
```

```
        System.out.println("Hello, World");
    }
}
```

docker image build -t wrajneesh/javatest .

docker container run wrajnees/javatest

o/p: Hello, World

docker login -u wrajneesh --- to login to docker hub

docker image push wrajneesh/javatest:latest

https://hub.docker.com/repository/docker/wrajneesh/javatest

Multi-stage build:

```
FROM openjdk:8 AS build_stage
COPY . .
WORKDIR .
RUN javac Hello.java

FROM openjdk:8-jre-alpine AS production_stage
COPY --from=build_stage Hello.class .
ENTRYPOINT ["java", "Hello"]
```

The resultant image is ~85MB in size (uncompressed).

Image = Layers (bunch of files) + Manifest files

Manifest file of an image defines how to stack the layers to build the image.

Layers are independent and have no dependency on other layers.

Manifest defines how the layers are stacked together.

docker login [<registry>]

docker image pull <image-name>

e.g. docker image pull redis

docker image ls

docker image ls --digests

Steps for image pull:

0. Get fat manifest - which will have references for different architecture dependent image manifests.

1. Get image manifest applicable to the current system architecture (docker system info returns this info) by default from hub.docker.com

2. Pull layers defined in the manifest in order.

Content addressable storage - SHA id associated with each Layer.

docker system info

--

Server Version: 19.03.8

Storage Driver: overlay2

--

We are using overlay2 driver that stores all immutable docker layers.

Layering in-order:

1. Base Layer - OS files and objects (it could be ubuntu linux files and objects but the kernel part of the OS will come from host system's OS kernel and that may be a different Linux distro like SUSE or Fedora). So even though base layer is the OS files layer it does not include the Linux kernel and so it is very slim.

2. App code copied to the image

3. Updates or patches to App

Storage driver overlay2 squashes all layers together into one docker image.

root@rwatsh-ThinkPad-T490:/var/lib/docker# ls

builder containers network plugins swarm trust

buildkit image **overlay2** runtimes tmp volumes

Under /var/lib/docker/overlay2 directory we see various layers. We cannot tell from the SHA ids of the layers which one contains the OS files (or is the Base Layer).

When we pull an image it gets stored under:

On Linux under /var/lib/docker

On Windows under C:/ProgramData/docker/windowsfilter

The above directories are like local registries.

So if we look under each layer's UID/diff directory we can find the root file system of the base layer under one of them.

IMPORTANT: These UIDs are randomly generated on file system and are not layer's content SHA ids.

root@rwatsh-ThinkPad-T490:/var/lib/docker/overlay2# ls -al

total 36

drwx------ 9 root root 4096 Apr 24 09:42 .

drwx--x--x 14 root root 4096 Apr 21 05:41 ..

drwx------ 3 root root 4096 Apr 24 09:42 137c1609daab9dfefb137a1dd15ffc5df55e1c8e766eb02ce90a1283c662c76e

drwx------ 4 root root 4096 Apr 24 09:42 6106f0e3c61f3f66a3d76acfa84361a273b95a7c5aecd11cbe32b74836101f00

drwx------ 4 root root 4096 Apr 24 09:42
927f0472c5a76ef6642d3ef616208152150c30ad896ef92152749a24babe7bc3

drwx------ 4 root root 4096 Apr 24 09:42
b7c8c26c461e42cc657e7070c187dab1abe6026a2e2fdc9f280223e95e46057c

drwx------ 4 root root 4096 Apr 24 09:42
d60d3043515be5798778f02c8b90f38d701a635e2b89e1c55f01bfce6cdf640b

drwx------ 4 root root 4096 Apr 24 09:42
d903cebd39580b1c316cfb257d88710d274252556acc859a15821624cc400620

drwx------ 2 root root 4096 Apr 24 09:42 l

root@rwatsh-ThinkPad-T490:/var/lib/docker/overlay2# ls
137c1609daab9dfefb137a1dd15ffc5df55e1c8e766eb02ce90a1283c662c76e

committed diff link

root@rwatsh-ThinkPad-T490:/var/lib/docker/overlay2# ls
137c1609daab9dfefb137a1dd15ffc5df55e1c8e766eb02ce90a1283c662c76e/diff/

bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

Running the docker history on image will show the layers that make that image.

docker image history redis

IMAGE	CREATED	CREATED BY	SIZE	COMMENT
a4d3716dbb72	35 hours ago	/bin/sh -c #(nop) CMD ["redis-server"]	0B	
<missing>	35 hours ago	/bin/sh -c #(nop) EXPOSE 6379	0B	
<missing>	35 hours ago	/bin/sh -c #(nop) ENTRYPOINT ["docker-entry…	0B	
<missing>	35 hours ago	/bin/sh -c #(nop) COPY file:df205a0ef6e6df89…	374B	
<missing>	35 hours ago	/bin/sh -c #(nop) WORKDIR /data	0B	
<missing>	35 hours ago	/bin/sh -c #(nop) VOLUME [/data]	0B	
<missing>	35 hours ago	/bin/sh -c mkdir /data && chown redis:redis …	0B	
<missing>	35 hours ago	/bin/sh -c set -eux; savedAptMark="$(apt-m…	24.6MB	

```
<missing>      35 hours ago    /bin/sh -c #(nop)  ENV REDIS_DOWNLOAD_SHA=53…
0B
<missing>      35 hours ago    /bin/sh -c #(nop)  ENV REDIS_DOWNLOAD_URL=ht…   0B
<missing>      35 hours ago    /bin/sh -c #(nop)  ENV REDIS_VERSION=5.0.9      0B
<missing>      35 hours ago    /bin/sh -c set -eux;  savedAptMark="$(apt-ma…   4.15MB
<missing>      35 hours ago    /bin/sh -c #(nop)  ENV GOSU_VERSION=1.12        0B
<missing>      35 hours ago    /bin/sh -c groupadd -r -g 999 redis && usera…   329kB
<missing>      2 days ago      /bin/sh -c #(nop)  CMD ["bash"]                 0B
<missing>      2 days ago      /bin/sh -c #(nop) ADD file:9b8be2b52ee0fa31d…   69.2MB
```

The commands shown in history o/p that have non-zero size have contributed to a layer. **The commands with 0B size made an entry into image config JSON file (like ENV, CMD, ENTRYPOINT, EXPOSE etc).**

RUN, COPY and ADD commands in Dockerfile make layers.

Each layer has a hash id.

Image ID is the hash of the image manifest file.

We can see the image config and layers using inspect command:

docker image inspect redis

Delete image: docker image rm redis

Image can be shared across multiple containers. **A running container adds a read-writable layer on top of the immutable layers during runtime by making a copy of the layer in the image that needs to be modified. This is called copy-on-write.**

Containers virtualize the OS unlike the VMs which virtualize the hardware running the OS. **A linux container will only run on a linux host OS as it needs to use the linux kernel from the host OS.**

docker container run -it alpine sh

/ # ps

PID USER TIME COMMAND

1 root 0:00 sh

7 root 0:00 ps

CNTL + P + Q = to come out of the interactive mode without killing the container.

docker container run -d alpine sleep 1d

docker container ls

docker container stop <container-id>

docker container ls -a -- lists even exited containers

docker container start <container-id>

Though the data written within the container before it was stopped will remain there after it is started, for production deployments if we need to persist data across container restarts then we we should use volumes to write the data to.

To remove all containers:

docker container rm $(docker container ls -aq) -f

To know the exposed ports of a container:

docker port <container name>

docker container run -d -p 80:80 --name web1 nginx

$ docker port web1

80/tcp -> 0.0.0.0:80

Orchestration (25%)

- Complete the setup of a swarm mode cluster, with managers and worker nodes

Refer https://www.tecmint.com/network-between-guest-vm-and-host-virtualbox/ to create virtualbox VMs and setup network between them so we can ssh to them from host system and ping other VMs from each VM. This requires setting up 2 network adapters - one host-only network adapter and one NAT.

A typical setup with 1 manager VM and 2 worker VM running ubuntu 19.10 is described in my blog post at - http://rwatsh.blogspot.com/2020/04/creating-swarm-cluster-with-virtualbox.html

On Manager:

docker swarm init --advertise-addr 192.168.56.101:2377 --listen-addr 192.168.56.101:2377

On 2 Workers:

docker swarm join --token SWMTKN-1-4azauf7ujxp711zzwb1ihfluqjy5om6pa4zlicmm7pq3l0svcp-d2bgy5mtsl0w82n99suvb1uu6 192.168.56.101:2377 --advertise-addr 192.168.56.102:2377 --listen-addr 192.168.56.102:2377

To get the above commands again anytime following commands can be used on the manager node to get it:

docker swarm join-token manager

docker swarm join-token worker

To join another manager:

docker swarm join --token <token> mgr1ip:2377 --advertise-addr mgr2ip:2377 --listen-addr mgr2ip:2377

docker node ls -- command will show which nodes are managers and which node is the leader among the manager nodes.

```
$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER
STATUS      ENGINE VERSION
vy17sxvahg6nerrmfi7lc2v9q *   manager             Ready               Active              Leader
18.09.9
0gi39018c92nqtcuq3iatsxma     worker1             Ready               Active
18.09.9
nci485cg5vcbtuu4yy412k38u     worker2             Ready               Active
18.09.9

* indicates the current node.
```

docker info -- will have a section on swarm showing number of managers and total nodes.

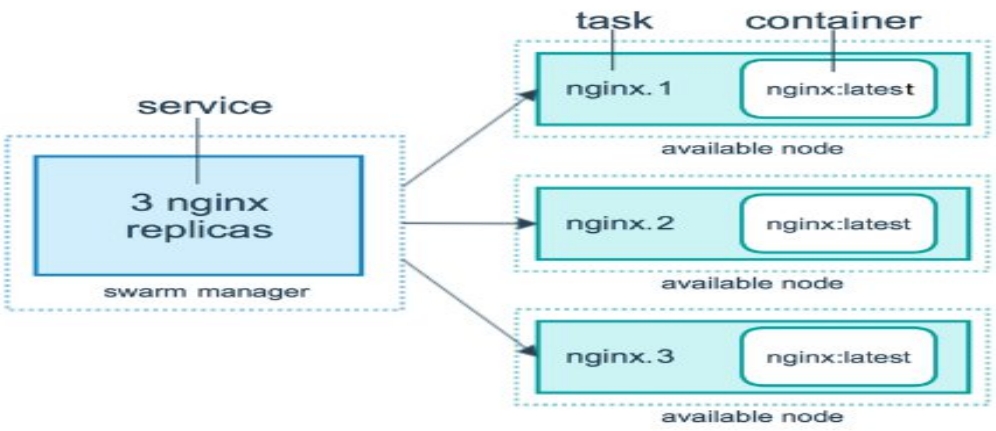
To promote a worker node to manager node:

docker node promote <node id>

- Describe and demonstrate how to extend the instructions to run individual containers into running services under swarm

Service:

When you deploy the service to the swarm, the swarm manager accepts your service definition as the desired state for the service. Then it schedules the service on nodes in the swarm as one or more replica tasks. The tasks run independently of each other on nodes in the swarm.



A container is an isolated process. In the swarm mode model, each task invokes exactly one container. A task is analogous to a “slot” where the scheduler places a container. Once the container is live, the scheduler recognizes that the task is in a running state. If the container fails health checks or terminates, the task terminates.

Service:

docker service create --name psight1 -p 8080:8080 --replicas 5 nigelpoulton/pluralsight-docker-ci

-p maps service replica port 8080 to swarm cluster node port 8080 (each node's port 8080 will be used up for this service psight1).

docker service ls

docker service ps psight1 -- shows which service replica is running on which node in the cluster.

docker service inspect psight1

Routing mesh - docker networking does load balancing between different replicas of app. We can access the service/app running in swarm cluster from <any-node-ip>:<app-port>. Similar to k8s ingress.

We can front the swarm nodes with an external LB that will load balance traffic between different swarm cluster nodes and then swarm's routing mesh will do the load balancing between service replicas.

To scale service:

docker service scale psight1=7

docker service update --replicas 10 psight1

Running docker ps on a node can show how many replicas of a service are running on the current node.

We can also do:

docker node ps <node-name> - to get the replicas on the node.

docker service ps psight1 -- shows the same across all nodes.

Shutting down a node in swarm re-balances the service replicas across the remaining nodes.

Adding back the node which was shutdown does not cause rebalancing though.

Rolling updates:

```
docker service rm psight1
docker service ls

docker network create -d overlay ps-net
docker network ls

docker service create --name psight2 --network ps-net -p 80:80 --replicas 12 nigelpoulton/tu-demo:v1

docker service inspect --pretty psight2

Update service image to tu-demo:v2, update-parallelism to 2 and delay to 10s:

docker service update --image nigelpoulton/tu-demo:v2 --update-parallelism 2 --update-delay 10s psight2

docker service ps psight2 | grep :v2


https://hub.docker.com/r/nigelpoulton/tu-demo
```

Uninstall and update docker version:

For Ubuntu:

```
sudo snap remove docker
sudo apt-get remove docker docker-engine docker.io containerd runc
sudo apt-get update
sudo apt-get install \
```

```
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88

sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"

sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io

# If get error - Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
docker daemon running?

sudo service docker restart

# If get error - -bash: /snap/bin/docker: No such file or directory
env | grep PATH
and remove /snap/bin from PATH
test running - docker version
```

 # To leave swarm cluster

sudo docker swarm leave

Stacks and DBA (Distributed Application Bundles):

Bundle - group services together.

Stack - application made up of multiple services - DAB file is stack definition.

docker stack - sub-command to create stack of services

docker compose v2+ YAML file is used to compose a stack of services in YAML compose file.

Swarm mode needs the docker compose YAML with stack definition converted to DAB format.

Install docker-compose:

sudo apt install docker-compose

docker-compose bundle - sub-command is used to generate a DAB file for consumption in swarm mode. Creates a file with .dab extension - its a JSON file. It uses the docker-compose.yaml file in current directory as input.

docker stack deploy <name-of-.dab file>

This will create all services and overlay networks defined in the .dab file.

docker stack services <stack-name>

docker stack rm <stack-name>

- Describe the importance of quorum in a swarm cluster

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

When the Docker Engine runs in swarm mode, manager nodes implement the Raft Consensus Algorithm to manage the global cluster state.

Raft tolerates up to (N-1)/2 failures and requires a majority or quorum of (N/2)+1 members to agree on values proposed to the cluster. This means that in a cluster of 5 Managers running Raft, if 3 nodes are unavailable, the system cannot process any more requests to schedule additional tasks. The existing tasks keep running but the scheduler cannot re-balance tasks to cope with failures if the manager set is not healthy.

To start swarm in locked mode:

docker swarm init —autolock

or enable/disable it at any time:

docker swarm update —autolock=true/false

This requires swarm manager to be unlocked upon manager restart we need to run admin command that can be run only on a manager node for example - docker node ls.

To unlock:

docker swarm unlock

docker swarm unlock-key — this can be run on any other manager nodes that remained up while the other manager node was restarted.

Stack

Defines the spec for all services, networking, secrets etc. in one yaml file.

https://raw.githubusercontent.com/dockersamples/example-voting-app/master/docker-stack.yml

A stack yaml has 4 top level elemets:

1. version

2. services

3. networks

4. volumes

rwatsh@manager:~$ **docker stack deploy --compose-file docker-stack.yml mystack**

Creating service mystack_db

Creating service mystack_vote

Creating service mystack_result

Creating service mystack_worker

Creating service mystack_visualizer

Creating service mystack_redis

docker stack ls

docker stack ps mystack

- Describe the difference between running a container and running a service

Docker container can run on a single node docker engine.

Docker service runs on a swarm cluster. The ports exposed on service will be available for access via any node in swarm cluster. They can also have a rolling update policy (how many replicas to update at a time in parallel during update) and unlike containers, services can have replicas distributed across nodes in swarm cluster.

https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#services-tasks-and-containers

- Interpret the output of “docker inspect” commands.

```
$ docker service inspect --pretty ping

ID:             2627bxkqap7mz6xjkr87hla0r
Name:           ping
Service Mode:   Replicated
 Replicas:      3
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:
        alpine:latest@sha256:9a839e63dad54c3a6d1834e29692c8492d93f90c59c978c1ed791
09ea4fb9a54
 Args:          sleep 1d
 Init:          false
Resources:
Networks: myovernet
Endpoint Mode:         vip
```

Running docker ps on a node can show how many replicas of a service are running on the current node.

We can also do:

docker node ps <node-name> - to get the replicas on the node. (can be run from manager)

docker service ps psight1 -- shows the same across all nodes.

docker container ls or docker ps - can show the containers on that node.

- Convert an application deployment into a stack file using a YAML compose file with "docker stack deploy"

docker service create --name registry --publish published=5000,target=5000 registry:2

curl http://localhost:5000/v2/

Create the example app as described here - https://docs.docker.com/engine/swarm/stack-deploy/

docker-compose.yaml:

```
version: '3'

services:
  web:
    image: 127.0.0.1:5000/stackdemo
    build: .
    ports:
      - "8000:8000"
  redis:
    image: redis:alpine
```

Dockerfile:

```
FROM python:3.4-alpine
ADD . /code
WORKDIR /code
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

app.py

```
from flask import Flask
from redis import Redis

app = Flask(__name__)
redis = Redis(host='redis', port=6379)

@app.route('/')
def hello():
    count = redis.incr('hits')
    return 'Hello World! I have been seen {} times.\n'.format(count)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, debug=True)
```

```
$ cat requirements.txt
flask
redis
```

docker stack deploy --compose-file docker-compose.yml stackdemo

docker stack services stackdemo

docker stack rm stackdemo

- Manipulate a running stack of services

https://docs.docker.com/engine/reference/commandline/stack_services/#related-commands

docker stack deploy

docker stack ls

docker stack ps — list tasks in stack

```
$ docker stack ps stackdemo
ID             NAME               IMAGE                          NODE          DESIRED STATE  CURRENT STATE
ERROR          PORTS
qzku3mzt898q   stackdemo_redis.1  redis:alpine                   worker2       Running        Running 5
minutes ago
tkkr3o2a46r8   stackdemo_web.1    127.0.0.1:5000/stackdemo:latest  worker1       Running
```

docker stack rm

docker stack services — list services in stack

```
$ docker stack services stackdemo
ID             NAME             MODE         REPLICAS       IMAGE                          PORTS
2rrjit1ishes   stackdemo_web    replicated   1/1            127.0.0.1:5000/stackdemo:latest
*:8000->8000/tcp
lqldj1tp2u1s   stackdemo_redis  replicated   1/1            redis:alpine
```

- Describe and demonstrate orchestration activities
  - Increase the number of replicas

To scale service:

docker service scale psight1=7

or

docker service update --replicas 7 psight1

Important: Scale can only be used with replicated mode service (and not with global mode service).

  - Add networks, publish ports

Bridge network - bridge driver (linux) or NAT (windows) - default

Isolated network scoped to a single node. So containers cannot talk directly to other containers on a different node.

Bridge is the default network driver.

docker network create <network name> -- encrypts control plane

docker network create -o encrypted <network name> -- also encrypts data plane

docker network inspect <network name>

Overlay network -

docker network create -d overlay myovernet

rwatsh@manager:/var/lib/docker$ docker network ls

NETWORK ID NAME DRIVER SCOPE

9c14fcfda1e7 bridge bridge local

...

8fd8acf145d4 mynet bridge local

r6c1vxekl1wx myovernet overlay swarm

The bridge network is scoped to local node.

docker container run -d --rm -p 8080:80 --name web1 nginx

The above command will run the container web1 on default bridge network (named bridge)

Running docker network inspect bridge -- shows web1 container in its o/p:

"Containers": {

"75c8b80145f708158fd35c69bad01a7e64cefff19cc9893963e481840d1e0761": {

"Name": "web1",

Since the container port is exposed on the local node port 8080

docker port web1

80/tcp -> 0.0.0.0:8080

so to access the page we need to run curl http://<node IP or hostname>:8080

To make it available across swarm cluster (so we can access it via any node in the cluster) we can use overlay network.

docker service create -d --name web1 --replicas 2 --network myovernet nginx

To create an overlay network which can be used by swarm services or standalone containers to communicate with other standalone containers running on other Docker daemons, add the **--attachable** flag:

```
$ docker network create -d overlay --attachable my-attachable-overlay
```

To encrypt traffic on an overlay network use —opt encrypted.

```
$ docker network create --opt encrypted --driver overlay --attachable my-attachable-multi-host-network
```

Map UDP port 80 on the service to port 8080 on the routing mesh.

docker service create —name web -d -p 8080:80/udp nginx

- Mount volumes

```
$ docker volume create my-vol
```

```
docker run -d \
  --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
```

is equivalent to:

$ docker run -d \

  --name devtest \

  **-v myvol2:/app **

  nginx:latest

For service:

$ docker service create -d \

  --replicas=4 \

  --name devtest-service \

  **--mount source=myvol2,target=/app **

  nginx:latest

Backing up a container: https://docs.docker.com/storage/volumes/#backup-restore-or-migrate-data-volumes

docker run -v /dbdata --name dbstore ubuntu /bin/bash

docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

Restore container from backup:

docker run --rm --volumes-from dbstore -v $(pwd):/backup ubuntu bash -c "cd /dbdata && tar xvf /backup/backup.tar --strip 1"

- Describe and demonstrate how to run replicated and global services

https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#replicated-and-global-services

2 types of service deployments:

1. Replicated service
2. Global service - is a service that runs one task on every node.

docker service create —name monitoringservice —mode global nginx

- Apply node labels to demonstrate placement of tasks

```
$ docker node update --label-add foo worker1
```

or

```
$ docker node update --label-add type=queue worker1
```

- Describe and demonstrate how to use templates with “docker service create”

You can use templates for some flags of service create, using the syntax provided by the Go’s text/template package.

```
$ docker service create \
  --name hosttempl \
  --hostname="{{.Node.Hostname}}-{{.Node.ID}}-{{.Service.Name}}"\
  busybox top
```

Set the template of the created containers based on the service’s name, the node’s ID and hostname where it sits.

- Identify the steps needed to troubleshoot a service not deploying

https://success.docker.com/article/swarm-troubleshooting-methodology

- Physical node failures
- Local storage volume filling to capacity
- OS kernel panics
- Exhaustion of file descriptors

- Network partitions
- DNS resolution
- Distributed multi-container application issues
- Loss of quorum for consensus-based systems

Flow for triaging will be:

$ docker service ls — list all services in swarm cluster and their replicas is all are running and ready or not.

$ docker service ps <service> — use --format "{{json .}}" and use jq for parsing

```
$ docker service ps ucp-agent --format "{{json .}}" --filter "desired-state=running" | jq -r .ID
```

$ docker service inspect <service> — created at, updated at, labels

$ docker inspect <task> — can be used to get the container id from task

$ docker inspect pw6u97k0th7q | jq -r '.[].Status.ContainerStatus.ContainerID'

$ docker inspect <container> — can be used to get all details for the container. Look for - startedAt, FinishedAt, ExitCode, Error, OOMKilled, node IP, CPUs, Memory, IPAddress, Ports, Networks.

docker inspect 90cbc98062c8 | jq '.[].State'

docker inspect 90cbc98062c8 | jq '.[].Node'

$ docker inspect 90cbc98062c8 | jq '.[].NetworkSettings'

$ docker logs <container>

- Describe how a Dockerized application communicates with legacy systems

- Describe how to deploy containerized workloads as Kubernetes pods and deployments
- Describe how to provide configuration to Kubernetes pods using configMaps and secrets.

Image Creation, Management and Registry (20%)

- Describe the use of Dockerfile
- Describe options, such as add, copy, volumes, expose, entry point

https://docs.docker.com/engine/reference/builder/#from

FROM:

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app
```

RUN:

```
RUN /bin/bash -c 'source $HOME/.bashrc; \
echo $HOME'

RUN ["/bin/bash", "-c", "echo hello"]
```

CMD: Only the last CMD in Dockerfile takes effect. Ideally there should be only one CMD statement. Main purpose is to provide defaults for an executing container.

The CMD instruction has three forms:

- CMD ["executable","param1","param2"] (exec form, this is the **preferred** form)
- CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
- CMD command param1 param2 (shell form)

FROM ubuntu **CMD ["-al"]** **ENTRYPOINT ["ls"]**
docker image build -t test_cmd:v1 .
docker container run —rm test_cmd:v1 This will output ls -al on root directory in ubuntu.
docker container run —rm test_cmd:v1 -ltr this will override the default CMD -al and pass -ltr as argument to ls.

If CMD is used to provide default arguments for the ENTRYPOINT instruction, both the CMD and ENTRYPOINT instructions should be specified with the JSON array format.

CMD can be used by itself in which case only one CMD should be present in Dockerfile or the last one will only take effect.

FROM ubuntu CMD ["wromg option"] ENTRYPOINT ["ls"] CMD ["-al"]
docker build -t test_cmd:v2 .
docker container run --rm test_cmd:v2 ignores the first CMD

EXPOSE - is documentation in Dockerfile that the container using the image will need that port published/exposed for external access. **To actually publish the port use -p option with docker container run.**

EXPOSE 80/tcp

tcp is default anyway.

VOLUME - creates a mount point and marks it as holding externally mounted volume from native host or other containers. Also see https://docs.docker.com/storage/volumes/

VOLUME /myvol

or VOLUME [“/myvol”]

docker volume create myvol

docker container run **-v myvol:/u01/shared** nginx

or

docker container run **—mount source=myvol,target=/u01/shared** nginx

3 types of volumes:

- **volume** - created under /var/lib/docker/volumes. The files under this directory are managed by docker. We can also create named volumes using docker volume CLI.
- **bind** - a file or directory on host system is mounted in container. This is outside of docker control and there is no named volume of bind mount type.

  docker container run -v /u01/shared:/u01/shared nginx

  where, /u01/shared is a directory on the host system that gets mounted inside container under /u01/shared directory.

- **tmpfs** - behaves like a volume but is in-memory on host. This type of volume cannot be shared between containers. Only available on Linux. When container stops tmpfs mount is removed unlike bind and volume types where the data exists even after container is stopped or removed.

- Identify and display the main parts of a Dockerfile

https://docs.docker.com/engine/reference/builder/#dockerfile-examples

```
FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox
RUN apt-get update && apt-get install -y x11vnc xvfb firefox
RUN mkdir ~/.vnc
# Setup a password
RUN x11vnc -storepasswd 1234 ~/.vnc/passwd
# Autostart firefox (might not be the best way, but it does the trick)
RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900
CMD    ["x11vnc", "-forever", "-usepw", "-create"]
```

A multi-stage dockerfile:

```
FROM golang:1.11-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

- Describe and demonstrate how to create an efficient image via a Dockerfile
  - Create ephemeral containers - containers should be stateless so even if they are destroyed a new container can be started to replace the old without any loss in functionality.
  - Keep a smaller docker build context - this results in faster builds of images and less image size (as we don't include unnecessary content in the image).
  - Pipe Dockerfile through stdin - can be used when Dockerfile is generated on-the-fly and need not be persisted.

```
docker build -<<EOF
FROM busybox
RUN echo "hello world"
EOF
```

  - Build an image using a Dockerfile from stdin without sending build context - when we dont need to add any files on the container (as shown in the example above) we can specify it with docker build - . This speeds up the docker build as there is no build context files to be sent to the dockerd.
  - Exclude with .dockerignore

- Use multi-stage builds - drastically reduces image size. Order the layers in the image from less frequently changed to the more frequently changed. See example above.
    - Install tools layer
    - install or update library dependencies
    - generate application
- Dont install unnecessary packages
- Decouple applications - each container should have only one concern (may not be just one process).
- Minimize the number of layers -
    - **RUN, COPY and ADD statements are the only ones that create layers**.
- Sort multi-line args alphanumerically - for ease of maintaining the Dockerfile

- Describe and demonstrate how to use CLI commands to manage images, such as list, delete, prune, rmi

https://docs.docker.com/engine/reference/commandline/image/#usage

docker image ls

docker image prune - removes unused images

docker image rm == docker rmi - deletes a specific image.

- Describe and demonstrate how to inspect images and report specific attributes using filter and format
- Get image OS

  $ docker image inspect test_cmd:v2 --format {{.Os}}

  linux

  $ docker image inspect test_cmd:v2 --format {{.ContainerConfig.Cmd}}

  [/bin/sh -c #(nop) CMD ["-al"]]

- Describe and demonstrate how to tag an image

```
docker tag 0e5574283393 myregistryhost:5000/fedora/httpd:version1.0
```

- Describe and demonstrate how to apply a file to create a Docker image

docker image load -i <tar file>

or

cat image.tar | docker image load

Also:

docker image save <image>

$ docker image save test_cmd:v2 -o test_cmd_v2.tar

$ docker image load -i test_cmd_v2.tar

Loaded image: test_cmd:v2

- Describe and demonstrate how to display layers of a Docker image

docker image history <image> —no-trunc

- Describe and demonstrate how to modify an image to a single layer

See example of multi-stage build above.

- Describe and demonstrate registry functions
    - Deploy a registry
    - Log into a registry
    - Utilize search in a registry
    - Push an image to a registry
- Sign an image in a registry
- Pull and delete images from a registry

Registry is an instance of registry image and runs as container.

docker run -d -p 5000:5000 --restart=always --name registry registry:2

docker tag test_cmd:v2 localhost:5000/test_cmd:v3

docker push localhost:5000/test_cmd:v3

docker pull localhost:5000/test_cmd:v3

docker container stop registry && docker container rm -v registry

To sign an image in registry we need to use DTR which is available only with Docker EE:

```
# Pull NGINX from Docker Hub
docker pull nginx:latest

# Re-tag NGINX
docker tag nginx:latest dtr.example.org/dev/nginx:1

# Log into DTR
docker login dtr.example.org

# Sign and push the image to DTR
export DOCKER_CONTENT_TRUST=1
docker push dtr.example.org/dev/nginx:1
```

docker image prune - deletes all dangling images (or untagged images not referenced by any container)

docker image prune -a - deletes all images that are not in use by any container.

Installation and Configuration (15%)

- Describe sizing requirements for installation

https://docs.docker.com/datacenter/ucp/2.2/guides/admin/install/system-requirements/#hardware-and-software-requirements

Enterprise tooling

Docker EE has:

Docker Engine EE

Ops UI - UCP

Secure Registry - DTR

UCP:

UCP Manager + UCP Workers= UCP Swarm Cluster

UCP Manager is control plane

UCP workers is where you will run apps.

Questions:

1. Order of backup recommended by Docker is:

    1. Swarm - backing up a single manager node should suffice as it stores the entire state of the cluster in etcd.

    2. UCP - stores config, access control, certs and metrics data.

3. DTR - stores images and config relevant to images.

2. COPY - lets you copy in a local file or directory from host into Docker image.

ADD - can do what COPY does plus it also supports source as URL and can auto-extract a tar file from source into docker image location.

ADD mytar.tar.gz /work

COPY is generally preferred over ADD. Only if tar file auto-extraction is required use ADD otherwise use COPY.

3. Ports to ppen to ensure traffic between swarm nodes are not blocked.

TCP/2377 - cluster management communication

TCP/7946 and UDP/7946 - communication among nodes.

UDP/4789 - for overlay network traffic.

- Describe and demonstrate the setup of repo, selection of a storage driver, and installation of the Docker engine on multiple platforms
- Describe the use of namespaces, cgroups, and certificate configuration

On Windows:

Download and install docker for windows.

Docker for Windows can do both Native Windows and Linux containers.

Activate windows containers support in powershell as:

- Install-Module DockerProvider -Force
- Install-Package Docker -Provider-Name DockerProvider -Force

On Linux:

wget -q0- https://get.docker.com/ | sh

To install on ubuntu:

sudo apt install docker

sudo apt install docker-compose

To add current user to docker group so we dont need to use sudo to run docker commands:

sudo groupadd docker

sudo usermod -aG docker $USER

Now we can run docker without using sudo. On Ubuntu we needed to reboot for it to tkae effect.

docker container run hello-world

docker version

Architecture:

Containers building blocks:

- Namespaces - isolation

- Control groups - grouping objects and setting limits.

Each container gets its own:

- process IDs - each container has its isolated process tree including its own init process pid 1

- network - so no port collision across containers running on same node, own network stack, NICs, ips etc.

- File system

- IPC - processes within a container can share memory but it remains isolated from other container's or host node's shared memory space.

- UTS - each container gets its own host name

- user - lets each container's users to be mapped to users on the host - for e.g. we can map container's root user to a non-privileged user on the host.

2 specs (as part of OCI - Open Container Initiative)

- - Image spec
- - Runtime spec - runc is ref. implementation.

Docker engine - split into:

- docker client - Docker Inc does this

- docker daemon (or API service) - this layer also implements swarm orchestration, builds, stacks, overlay networking etc. - Docker Inc does this too.

  - Windows has docker client (docker.exe) and docker daemon (dockerd.exe) ported and implementing same APIs and features, including swarm orchestration.

- containerd - handles lifecycle operations, like start container, stop, pause etc. - containerd and GRPC are projects owned by CNCF.

  - On windows this is currently called "Compute Services" and has a different implementation thank containerd on Linux.

- Runtime - OCI Layer that interfaces with the kernel - runc on Linux

Flow is:

1. Docker client sends a command to run container (docker container run ...)

2. Docker API service (or docker daemon) invokes the containerd (REST POST/vx.x/containers/create HTTP1.1 endpoint invoked).

3. **containerd** is also a daemon process that invokes runc (via a shim process) to start container (GRPC API call to containerd - client.NewContainer(context, ...)) - creates a shim process for every container which then calls runc. The shim process created by containerd sticks around as long as the container is running.

4. **runc** - forks a container and exits leaving every container associated with its shim process- (system call to LXC/Kernel)

The docker API daemon and containerd can be stopped or upgraded without affecting the running containers. The running containers are only associated with their shim process. When the docker engine is upgraded (or restarted)when it comes back up it discovers the running container shim processes and connects to them.

On Windows:

- A container has multiple processes (smss is windows version of initd process on Linux) as windows processes require other system processes to be available on bootup unlike in linux where we have just one initd process on bootup.

Windows supports 2 different types of containers:

- **Native win32 containers** - namespace isolation for containers much like Linux

docker container run (creates a native win32 container)

- **Hyper-V containers** - launches a Hyper-v light-weight VM with full windows OS kernel and a single container running within it (so its a VM + container combination to give better isolation than namespace isolation)

  - in this model, it is also possible to run a Linux hyper-v VM and a Linux container within it.

docker container run --isolation=hyperv (creates a Hyper-v container)

- Describe and demonstrate configuration of logging drivers (splunk, journald, etc.)

Container log:

All logs from app running in container that is done to stdout and stderr is captured by logging drivers (syslog, gelf, splunk, fluentd etc.)

Default logging driver is set in **/etc/docker/daemon.json** file (on windows it is **%programdata%\docker\config\daemon.json**). This file does not exist by defualt so we need to create it first time configuring the daemon. More at https://docs.docker.com/config/daemon/

Sample:

```
{
  "debug": true,
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem",
  "hosts": ["tcp://192.168.59.3:2376"]
}
```

After making any changes to the config, we can run the following command to reload configuration:

sudo systemctl daemon-reload

For container specific logging driver we can use the following option when starting dockerd or use the dockerd config json and reload it:

--log-driver string **Default driver for container logs (default "json-file")**

--log-opt map **Default log driver options for containers (default map[])**

docker logs <container-name> == used to view container logs.

E.g.

$ docker logs web1 -f

172.17.0.1 - - [26/Apr/2020:20:21:59 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.113 Safari/537.36" "-"

To check the status of dockerd:

sudo systemctl is-active docker

or

sudo service docker status
Example o/p:

```
● docker.service - Docker Application Container Engine
    Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
    Active: active (running) since Tue 2020-04-21 05:41:10 CDT; 5 days ago
TriggeredBy: ● docker.socket
      Docs: https://docs.docker.com
  Main PID: 3140 (dockerd)
     Tasks: 49
    Memory: 1013.4M
    CGroup: /system.slice/docker.service
            ├─  3140 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
            └─3206703 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 80 -container-ip
```

172.17.0.2 -container-port 80

- Describe and demonstrate how to set up swarm, configure managers, add nodes, and setup the backup schedule

Swarm has 2 pieces to it:

1. Secure cluster - this is central to docker's strategy

2. Orchestrator - this may get replaced by k8s in production

Running k8s workload on docker is only supported in Docker EE

Swarm mode is powered by https://github.com/docker/swarmkit

Swarmkit is integrated in docker engine in v1.12+

Docker engine has 2 modes:

1. Single engine mode

2. Swarm mode

docker swarm init - command run on first node which becomes manager and leader node for the swarm cluster.

Swarm uses etcd as distributed cluster store where it stores TLS certs for managers and worker nodes (these are generated by the leader manager node). The leader manager is by default the root CA. We can specify --external-ca to use another root CA. There is only one leader manager at any given time. When leader goes down one of the follower manager nodes will become the new leader (uses Raft concensus algorithm to select the new leader)

A new node can be joined to the swarm cluster as manager or worker:

docker swarm join <manager | worker>

-- gives the command to be executed on manager or worker node.

Best practice is to have 3,5 or 7 managers for manager HA. That forms a **quorum** (or majority) across ADs even when one AD goes down.

docker node ls -- shows all nodes in the cluster (can only be run from manager node)

We can have a hybrid windows and linux nodes in a swarm cluster.

To see the contents of swarm node certificate:

root@manager:**/var/lib/docker/swarm/certificates**# ls

swarm-node.crt swarm-node.key swarm-root-ca.crt

root@manager:/var/lib/docker/swarm/certificates# **openssl x509 -in swarm-node.crt -text**

A manager node can be **autolocked** so running docker node ls command will fail stating - Swarm is encrypted and needs to be unlocked before it can be used. This ensures that is join token is compromised and a rogue node becomes swarm manager it will still be unable to perform any management actions unless it has the unlock key.

docker swarm [init|update] --autolock=true

>> Will generate an unlock key

sudo service docker restart

docker swarm unlock -- to unlock the swarm

To change certificate expiration:

docker swarm update --cert-expiry 48h -- this will change expiration to 2 days (verify with docker info). Default expiration is 3 months.

CA Configuration:

Expiry Duration: 3 months

- Describe and demonstrate how to create and manage user and teams.

https://docs.docker.com/datacenter/dtr/2.4/guides/admin/manage-users/create-and-manage-teams/

- Describe and demonstrate how to configure the Docker daemon to start on boot

sudo systemctl enable docker — for systemd

or

echo manual | sudo tee /etc/init/docker.override — for upstart

or

sudo chkconfig docker on — for chkconfig

- Describe and demonstrate how to use certificate-based client-server authentication to ensure a Docker daemon has the rights to access images on a registry

https://docs.docker.com/engine/security/certificates/

- Describe and interpret errors to troubleshoot installation issues without assistance

https://docs.docker.com/config/daemon/#troubleshoot-the-daemon

Docker Engine Logs:

Linux:

systemd:

journalctl -u docker.service

non-systemd:

/var/log/messages

Windows:

~/AppData/Local/Docker

Also consider:

- Enabling debug in /etc/docker/docker.json (debug = “true”)
- Send a HUP signal to daemon to cause it to reload its config without restart.

  sudo kill -SIGHUP $(pidof dockerd)

- Describe and demonstrate the steps to deploy the Docker engine, UCP, and DTR on AWS and on-premises in an HA configuration

https://docs.docker.com/datacenter/dtr/2.3/guides/admin/install/

https://docs.docker.com/ee/ucp/

https://docs.docker.com/engine/swarm/admin_guide/#add-manager-nodes-for-fault-tolerance

- Describe and demonstrate how to configure backups for UCP and DTR

https://docs.docker.com/datacenter/ucp/2.2/guides/admin/backups-and-disaster-recovery/

Networking (15%)

- Describe the Container Network Model and how it interfaces with the Docker engine and network and IPAM drivers

https://success.docker.com/article/networking/

Bridge network - bridge driver (linux) or NAT (windows) - default

Isolated network scoped to a single node. So containers cannot talk directly to other containers on a different node. But on the same host, containers can talk to each other.

Bridge is the default network driver.

docker network create <network name> -- encrypts control plane

docker network create -o encrypted <network name> -- also encrypts data plane

docker network inspect <network name>

Overlay network -

docker network create -d overlay myovernet

rwatsh@manager:/var/lib/docker$ docker network ls

NETWORK ID NAME DRIVER SCOPE

9c14fcfda1e7 bridge bridge local

...

8fd8acf145d4 mynet bridge local

r6c1vxekl1wx myovernet overlay swarm

The bridge network is scoped to local node.

docker container run -d --rm -p 8080:80 --name web1 nginx

The above command will run the container web1 on default bridge network (named bridge)

Running **docker network inspect bridge** -- shows web1 container in its o/p:

"Containers": {

"75c8b80145f708158fd35c69bad01a7e64cefff19cc9893963e481840d1e0761": {

"Name": "web1",

Since the container port is exposed on the local node port 8080

docker port web1

80/tcp -> 0.0.0.0:8080

so to access the page we need to run curl http://<node IP or hostname>:8080

To make it available across swarm cluster (so we can access it via any node in the cluster) we can use overlay network.

docker service create -d --name web1 --replicas 2 --network myovernet nginx

We cannot attach docker container run to a swarm scoped overlay network... we need to create a service.

Service discovery -

1. Every service gets a name

2. Names are registered with Swarm DNS

3. Every service uses swarm DNS to lookup other services by name.

Demo service ping by name:

```
$ docker service create -d --name ping --network myovernet --replicas 3 alpine sleep 1d

$ docker service create -d --name pong --network myovernet --replicas 3 alpine sleep 1d

docker container exec -it <container-id ping service> sh
#/ ping pong
```

Load Balancing:

1. Ingress load balancing - a service replica can be accessed via any node in the swarm cluster.

2. Internal load balancing - swarm will balance the load across all service replicas across nodes in the cluster. So we can have an external load balancer to load balance between swarm nodes and regardless of which node the request arrives at from external load balancer, swarm will load balance it across the service replicas.

```
docker service create --network myovernet -d -p 8080:80 --replicas 1 --name web2 nginx

docker service inspect web2
...
Ports:
 PublishedPort = 8080
  Protocol = tcp
  TargetPort = 80
  PublishMode = ingress
...
```

Every node in swarm cluster is attached to ingress network.

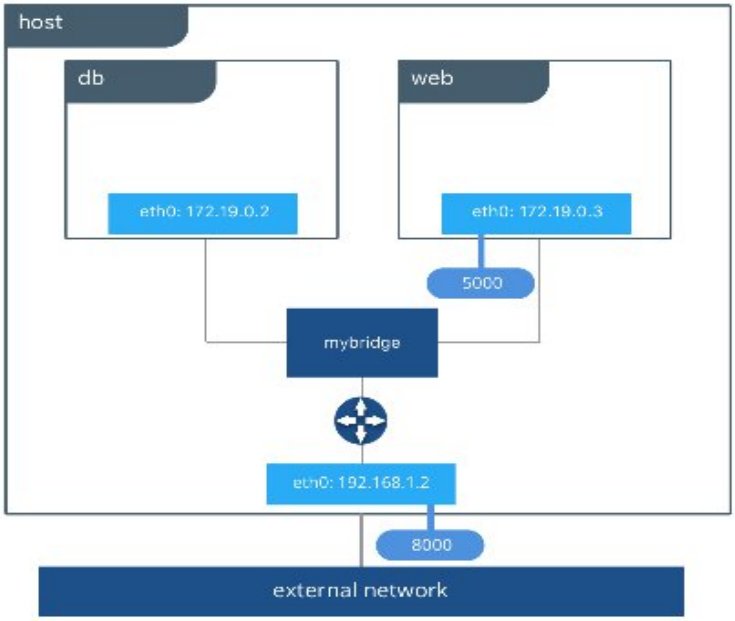
```
docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
...
ztbhs71r01wk        ingress             overlay             swarm
```

- Describe the different types and use cases for the built-in network drivers.

- Describe and demonstrate how to create a Docker bridge network for developers to use for their containers.
- Describe and demonstrate how to publish a port so that an application is accessible externally
- Identify which IP and port a container is externally accessible on.

https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/

– Bridge Network Driver - creates a private network internal to the host so container on this network can communicate. External access granted by exposing ports to containers. Docker secures network by managing iptables rules that block connectivity between different Docker networks.



To create user-defined bridge network:

docker network create -d bridge mybridge

docker run -d --net mybridge --name db redis

docker run -d --net mybridge -e DB=db -p 8000:5000 --name web chrch/web

The application web is being serviced at host port 8000.

The db is accessible by its container name from application container web.

The bridge driver is a **local scope** driver, which means it only provides service discovery, IPAM, and connectivity on a single host.

– Overlay network driver

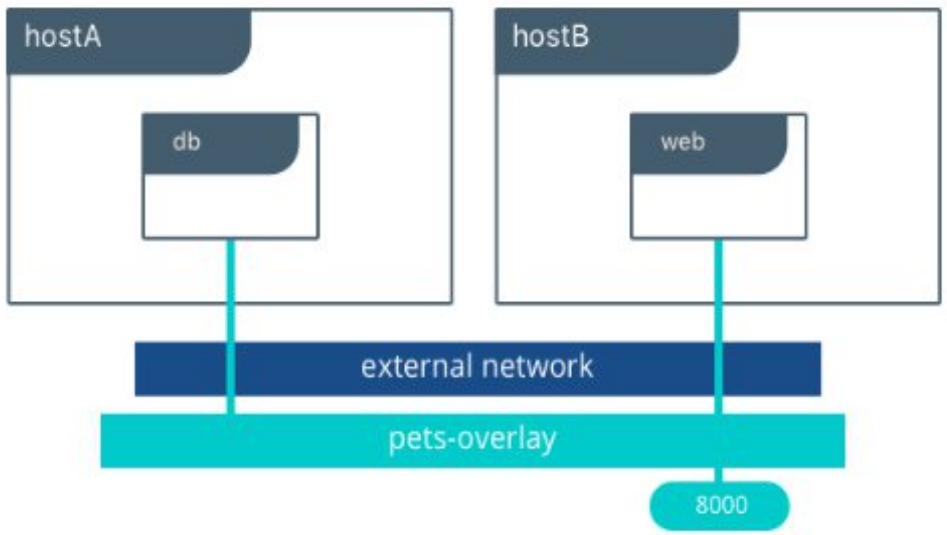
It is swarm scope driver so it spans across all nodes in the swarm cluster.

IPAM, service discovery, multi-host connectivity, encryption, and load balancing are built right in.

docker network create -d overlay --opt encrypted pets-overlay

docker service create --network pets-overlay --name db redis

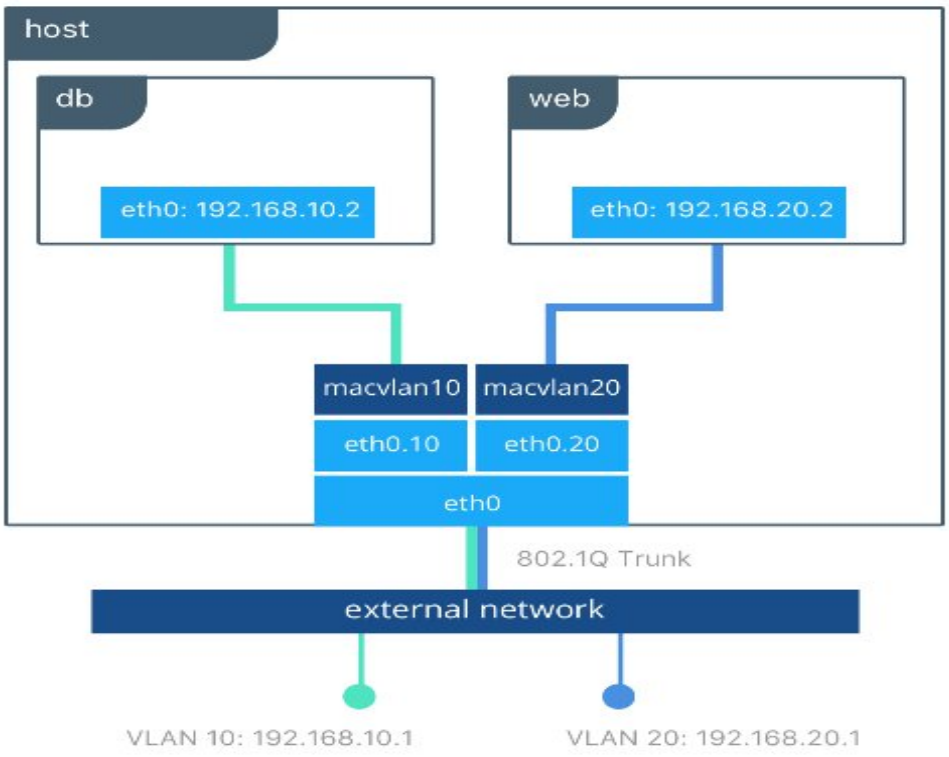
docker service create --network pets-overlay -p 8000:5000 -e DB=db --name web chrch/web



– Macvlan network driver

Lets each container have a virtual NIC and MAC address. It connects container's vNIC directly to host's NIC so containers are addressed with routable IP addresses that are on the subnet of the external network.

As a result of routable IP addresses, containers communicate directly with resources that exist outside a Swarm cluster without the use of NAT and port mapping. This can aid in network visibility and troubleshooting.



docker inspect <container-name> — shows the IP and port published by container and mapped to host port (if applicable)

- Describe the types of traffic that flow between the Docker engine, registry and UCP controllers.

https://success.docker.com/article/networking/

- Compare and contrast “host” and “ingress” publishing modes

The following example runs nginx as a service on each node in your swarm and exposes nginx port 80 locally on each swarm node.

```
$ docker service create \
  --mode global \
  --publish mode=host,target=80,published=8080 \
  --name=nginx \
  nginx:latest
```

You can reach the nginx server on port 8080 of every swarm node. If you add a node to the swarm, a nginx task is started on it. You cannot start another service or container on any swarm node which binds to port 8080.

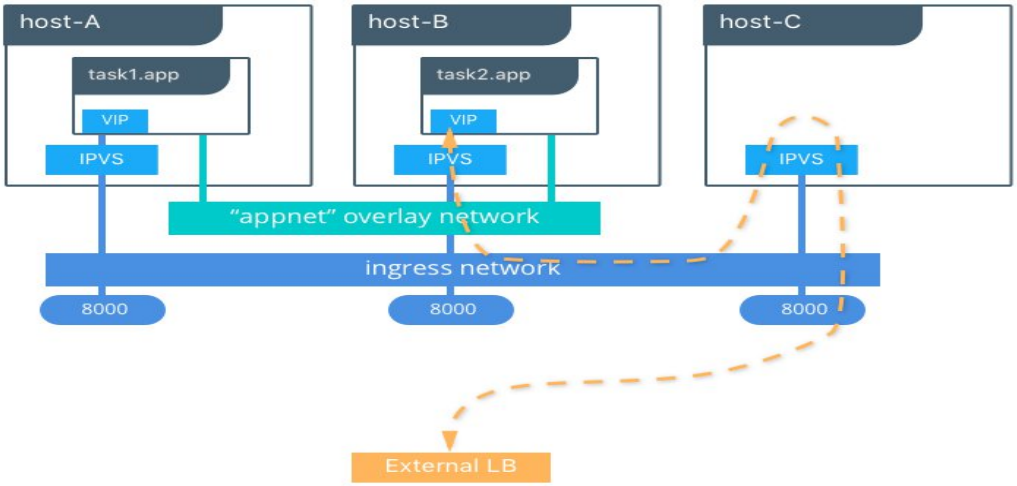
For example, the following command publishes port 80 in the nginx container to port 8080 for any node in the swarm:

```
$ docker service create \
  --name app \
  --publish published=8080,target=80 \
  --replicas 2 \
  nginx
```

When you access port 8080 on any node, Docker routes your request to an active container.

This diagram illustrates how the routing mesh works.

- A service is created with two replicas, and it is port mapped externally to port `8000`.
- The routing mesh exposes port `8000` on each host in the cluster.
- Traffic destined for the `app` can enter on any host. In this case the external LB sends the traffic to a host without a service replica.
- The kernel's IPVS load balancer redirects traffic on the `ingress` overlay network to a healthy service replica.



Routing mesh leverages operating system primitives (IPVS+iptables on Linux and VFP on Windows) to create a powerful cluster-wide transport-layer (L4) load balancer. It allows the Swarm nodes to accept connections on the services' published ports. When any Swarm node receives traffic destined to the published TCP/UDP port of a running `service`, it forwards it to service's VIP using a pre-defined overlay network called `ingress`. The `ingress` network behaves similarly to other overlay networks but its sole purpose is to provide inter-host transport for

mesh routing traffic from external clients to cluster services. Once you launch services, you can create an external DNS record for your applications and map it to any or all Docker Swarm nodes. You do not need to know where the container is running as all nodes in your cluster look as one with the routing mesh routing feature.

https://success.docker.com/article/networking#swarmexternall4loadbalancingdockerroutingmesh

You can publish a port for an existing service using the following command:

```
$ docker service update \
  --publish-add published=<PUBLISHED-PORT>,target=<CONTAINER-PORT> \
  <SERVICE>
```

You can use docker service inspect to view the service's published port. For instance:

```
$ docker service inspect --format="{{json .Endpoint.Spec.Ports}}" my-web

[{"Protocol":"tcp","TargetPort":80,"PublishedPort":8080}]
```

- Describe and demonstrate how to configure Docker to use external DNS.

```
Either via
$ docker run --dns 10.0.0.2 busybox nslookup google.com


or edit your /etc/docker/daemon.json to have something like:


{
    "dns": ["10.0.0.2", "8.8.8.8"]
}


then restart docker service
$ sudo systemctl docker restart
```

- Describe and demonstrate how to use Docker to load balance HTTP/HTTPs traffic to an application (Configure L7 load balancing with Docker EE).

https://success.docker.com/article/networking#ucpexternall7loadbalancinghttproutingmesh

UCP provides built-in L7 HTTP/HTTPS load balancing. URLs can be load balanced to services and load balanced across the service replicas.

- Understand and describe the types of traffic that flow between the Docker engine, registry, and UCP controllers

https://docs.docker.com/ee/docker-ee-architecture/#docker-enterprise-components

- Describe and demonstrate how to deploy a service on a Docker overlay network

```
docker network create -d overlay my-overlay
```

You can use the overlay network feature with both **--opt encrypted --attachable** and attach unmanaged containers to that network:

```
$ docker network create --opt encrypted --driver overlay --attachable my-attachable-multi-host-network
```

- Describe and demonstrate how to troubleshoot container and engine logs to resolve connectivity issues between containers.

https://success.docker.com/article/troubleshooting-container-networking

Container - container networking issues:

1. On the host where the frontend container is running, start a netshoot container reusing the network namespace affected container:

```
docker run -it --rm --network container:<container_name> nicolaka/netshoot
```

Once inside container,

- nslookup tasks.<service_name>
- nc -zvw2 <task_ip> <service 's listening port>
- nslookup <IP> == to get the service name and task id.
- docker network inspect -v <network-name>
- docker service ps <service_name>
- docker inspect —type task <task_id>
- Docker daemon logs: journalctl -u docker —no-pager —since "YYYY-MM-DD 00:00" --until "YYYY-MM-DD 00:00"

2. Ingress network troubleshooting:

- docker service inspect <service_name> — find the ingress VIP

- docker run -it --rm -v /var/run/docker/netns:/netns --privileged=true nicolaka/netshoot nsenter --net=/netns/ingress_sbox sh
- iptables -nvL -t mangle |awk '/<vip>/ {printf("%d", $NF)}'; echo
- ipvsadm -l -f <fwmark>
-

- Describe how to route traffic to Kubernetes pods using ClusterIP and NodePort services

https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/

- Describe the Kubertnetes' container network model

https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model

Security (15%)

Secrets

String <= 500K

docker secret create sec1 ... creates a secret and registers it with the manager in swarm cluster.

docker service create --name green --secret sec1 --replicas 2 ...

Manager sends the secret to those worker nodes that are running one of the service replicas to which sec1 secret access has been granted.

The secret sec1 is mounted in its unencrypted form at **/run/secrets** (Linux) in-memory (and never written to disk).

For windows the in-memory location is **%ProgramData%\Docker\Secrets**.

When service is deleted, worker nodes flushes the secret.

```
rwatsh@manager:~$ echo "secret stuff" > mysec1
rwatsh@manager:~$ docker secret create sec1 ~/mysec1
b37isgvr41xumynloxdv5o8ob
```

```
rwatsh@manager:~$ docker secret ls
ID                          NAME                DRIVER              CREATED             UPDATED
b37isgvr41xumynloxdv5o8ob   sec1                                    4 seconds ago       4 seconds ago

rwatsh@manager:~$ docker service create -d --name secservice --secret sec1 alpine sleep 1d
weyizascrwza929nezd3wegoy

rwatsh@manager:~$ docker container exec -it 0de297c8fbbe sh
/ # cat /run/secrets/sec1
secret stuff
```

A secret in use cannot be deleted. So first delete the service and then secret can be deleted.

rwatsh@manager:~$ docker service rm secservice

secservice

rwatsh@manager:~$ docker secret rm sec1

sec1

- Describe security administration and tasks.
- Describe the process of signing an image

Docker Content Trust (DCT) provides the ability to use digital signatures for data sent to and received from remote Docker registries. Through DCT, image publishers can sign their images and image consumers can ensure that the images they pull are signed.

https://docs.docker.com/engine/security/trust/content_trust/#signing-images-with-docker-content-trust

[REGISTRY_HOST[:REGISTRY_PORT]/]REPOSITORY[:TAG]

eg. `dtr.example.com/admin/demo:1`

DCT is associated with the  portion of an image. Each image repository has a set of keys that image publishers use to sign an image tag. Image publishers have discretion on which tags they sign.

An image repository can contain an image with one tag that is signed and another tag that is not.

Image consumers can enable DCT to ensure that images they use were signed. If a consumer enables DCT, they can only pull, run, or build with trusted images. Enabling DCT is a bit like applying a “filter” to your registry. Consumers “see” only signed image tags and the less desirable, unsigned image tags are “invisible” to them.

$ docker trust key generate jeff

$ docker trust key load key.pem --name jeff

$ docker trust signer add --key cert.pem jeff dtr.example.com/admin/demo

$ docker trust sign dtr.example.com/admin/demo:1

or alternatively, we can enable DCT and each push with automatically sign and push the image to registry.

$ export DOCKER_CONTENT_TRUST=1

$ docker push dtr.example.com/admin/demo:1

- Describe default engine security

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

- Describe swarm default security

https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/

- Describe MTLS

The nodes in a swarm use mutual Transport Layer Security (TLS) to authenticate, authorize, and encrypt the communications with other nodes in the swarm.

- Describe identity roles

https://docs.docker.com/datacenter/ucp/2.2/guides/access-control/permission-levels/#roles

A role is a set of permitted API operations on a collection that you can assign to a specific user, team, or organization by using a grant.

- Compare and contrast UCP workers and managers

https://docs.docker.com/datacenter/ucp/2.2/guides/architecture/

If nodes are not already running in a swarm when installing UCP, nodes will be configured to run in swarm mode.

When you deploy UCP, it starts running a globally scheduled service called ucp-agent. This service monitors the node where it's running and starts and stops UCP services, based on whether the node is a manager or a worker node.

If the node is a:

Manager: the ucp-agent service automatically starts serving all UCP components, including the UCP web UI and data stores used by UCP. The ucp-agent accomplishes this by deploying several containers on the node. By promoting a node to manager, UCP automatically becomes highly available and fault tolerant.

Worker: on worker nodes, the ucp-agent service starts serving a proxy service that ensures only authorized users and other UCP services can run Docker commands in that node. The ucp-agent deploys a subset of containers on worker nodes.

- Describe the process to use external certificates with UCP and DTR

https://success.docker.com/article/how-do-i-provide-an-externally-generated-security-certificate-during-the-ucp-command-line-installation

To specify an externally-generated and signed certificate for the UCP controller during a command line installation, use the `--external-server-cert` option.

```
docker run --rm -it \
  --name ucp \
  -v /var/run/docker.sock:/var/run/docker.sock \
  docker/ucp \
  install \
  --external-server-cert \
  [command options]
```

- Describe and demonstrate that an image passes a security scan

https://docs.docker.com/datacenter/dtr/2.5/guides/admin/configure/set-up-vulnerability-scans/

- Describe and demonstrate how to enable Docker Content Trust

https://docs.docker.com/engine/security/trust/content_trust/#enabling-dct-within-the-docker-enterprise-engine

The content-trust flag is based around a mode variable instructing the engine whether to enforce signed images, and a trust-pinning variable instructing the engine which sources to trust.

```
# Retrieving Root ID
$ grep -r "root" ~/.docker/trust/private
/home/ubuntu/.docker/trust/private/0b6101527b2ac766702e4b40aa2391805b70e5031c04714c748f914e89014403.key:role: root

# Using a Canonical ID that has signed 2 repos (mydtr/user1/repo1 and mydtr/user1/repo2).
```

Note you can use a Wildcard.

```
{
  "content-trust": {
    "trust-pinning": {
      "root-keys": {
         "mydtr/user1/*": [
           "0b6101527b2ac766702e4b40aa2391805b70e5031c04714c748f914e89014403"
         ]
      }
    },
    "mode": "enforced"
  }
}
```

- Describe and demonstrate how to configure RBAC with UCP

https://docs.docker.com/datacenter/ucp/2.2/guides/access-control/

If you're a UCP administrator, you can create *grants* to control how users and organizations access swarm resources.

A grant is made up of a *subject*, a *role*, and a *resource collection*. A grant defines who (subject) has how much access (role) to a set of resources (collection).

- Subject = user, team or organization
- Role = set of permitted API operations (Full control, View only, etc)
- Resource collections = any swarm resource (physical or virtual nodes, containers, services, networks, volumes, secrets and application configs)

- Describe and demonstrate how to integrate UCP with LDAP/AD

https://docs.docker.com/datacenter/ucp/2.2/guides/admin/configure/external-auth/

Docker UCP integrates with LDAP directory services, so that you can manage users and groups from your organization's directory and it will automatically propagate that information to UCP and DTR.

You control how UCP integrates with LDAP by creating searches for users. You can specify multiple search configurations, and you can specify multiple LDAP servers to integrate with. Searches start with the Base DN, which is the distinguished name of the node in the LDAP directory tree where the search starts looking for users.

- Describe and demonstrate how to create UCP client bundles

https://www.docker.com/blog/get-familiar-docker-enterprise-edition-client-bundles/

A client bundle is a group of certificates downloadable directly from the Docker Universal Control Plane (UCP) user interface within the admin section for “My Profile”. This allows you to authorize a remote Docker engine to a specific user account managed in Docker EE, absorbing all associated RBAC controls in the process. You can now execute docker swarm commands from your remote machine that take effect on the remote cluster. (Similar to running kubectl against a remote k8s cluster by setting the local KUBECONFIG to point to the remote k8s master).

Storage and Volumes (10%)

Volumes and Persistent Data

/var/lib/docker (or %ProgramData%\Docker\windowfilter) is the ephemeral storage location for docker container.

docker volume create ...

creates volumes that are outside of container space and are not impacted by lifecycle of containers.

The volumes can be backed by SAN or NFS storage.

```
rwatsh@manager:/var/lib/docker$ docker volume create myvol
myvol
rwatsh@manager:/var/lib/docker$ docker volume ls
DRIVER              VOLUME NAME
local               myvol
root@manager:/var/lib/docker/volumes# ls
metadata.db  myvol

root@manager:/var/lib/docker/volumes# docker volume rm myvol

docker container run -dit --name voltest --mount source=ubervol,target=/vol alpine

If ubervol does not exist, docker will create it.

rwatsh@manager:/var/lib/docker$ docker volume ls
DRIVER              VOLUME NAME
local               ubervol

$ docker container exec -it voltest sh
```

#/ echo "some data" > /vol/newfile

root@manager:/var/lib/docker/volumes/ubervol/_data# cat newfile
some data

As long as volume is in use we cannot delete it.

rwatsh@manager:/var/lib/docker$ docker volume rm ubervol
Error response from daemon: remove ubervol: volume is in use -
[102fc4c13246a14c6ce7166d79b149dd943bd58a5123b4bacc28f53309d671f3]

- Identify the correct graph drivers to uses with various operating systems

https://docs.docker.com/storage/storagedriver/select-storage-driver/

- overlay2 (default) - on Linux (uses xfs and ext4 type backing filesystem)
- aufs - default on older docker (18.06 or older) on Linux with kernel 3.13 or older
- devicemapper - requires direct-lvm backing filesystem for production. Older RHEL/Centos which did not support overlay2 required this.
- btrfs and zfs - backed by btrfs and zfs filesystem on host respectively
- vfs - only for testing purposes (where no copy-on-write filesystem can be used). This can be backed by any filesystem.

* Note: Backing filesystem - is where **/var/lib/docker** directory is located.

- Describe and demonstrate how to configure devicemapper.

https://docs.docker.com/storage/storagedriver/device-mapper-driver/#configure-docker-with-the-devicemapper-storage-driver

```
$ sudo systemctl stop docker
```

Edit /etc/docker/daemon.json

```
{
  "storage-driver": "devicemapper"
}
```

```
$ sudo systemctl start docker
```

```
$ docker info
```

...

`Storage Driver: devicemapper`

...

- Compare and contrast object and block storage and when they should be used.

https://rancher.com/block-object-file-storage-containers/

File system storage is probably the most awkward match for containers because file systems were not originally designed with portability in mind. As I've noted, however, there are ways to implement container-friendly file storage systems; this is usually done by distributing a file system across multiple servers

Block storage is more flexible than file system storage, which makes it easier to adapt block storage for container environments. The only big challenge is making sure that block storage data is available across an environment composed of multiple hosts. This can be resolved through distributed storage.

Object storage can be more complex to implement because it relies on REST calls, but the scalability that object storage provides makes it a good choice for container environments where massive scalability is a priority.

- Describe how an application is composed of layers and where these layers reside on the filesystem

https://docs.docker.com/storage/storagedriver/#images-and-layers

When you use docker pull to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually **/var/lib/docker/<storage-driver>** on Linux hosts.

```
$ ls /var/lib/docker/overlay2
16802227a96c24dcbeab5b37821e2b67a9f921749cd9a2e386d5a6d5bc6fc6d3
377d73dbb466e0bc7c9ee23166771b35ebdbe02ef17753d79fd3571d4ce659d7
3f02d96212b03e3383160d31d7c6aeca750d2d8a1879965b89fe8146594c453d
```

- Describe the use of volumes are used with Docker for persistent storage

https://docs.docker.com/storage/volumes/

Also covered in Basics section above.

- Identify the steps to take to clean up unused images on a filesystem and DTR

https://docs.docker.com/engine/reference/commandline/image_prune/

```
$ docker image prune -a
```

```
$ docker system prune
```

Deleting images in DTR:

There are three steps to delete a signed image:

1. Find which roles signed the image.
   notary delegation list dtr-example.com/library/wordpress

```
ROLE              PATHS          KEY IDS                                                    THRESHOLD
----              -----          -------                                                    ---------
targets/releases  "" <all paths>
c3470c45cefde5447cf215d8b05832b0d0aceb6846dfa051db249d5a32ea9bc8    1
targets/qa        "" <all paths>
c3470c45cefde5447cf215d8b05832b0d0aceb6846dfa051db249d5a32ea9bc8    1
```

2. Remove the trust data for each role.
   notary remove dtr-example.com/library/wordpress <tag> --roles <role-name> --publish
3. The image is now unsigned, so you can delete it.

- Describe and demonstrate how storage can be used across cluster nodes

https://docs.docker.com/engine/extend/legacy_plugins/#volume-plugins

Use NFS shared file system across nodes in a cluster to maintain the backing files for the docker storage driver layers.

- Describe how to provision persistent storage to a Kubernetes pod using persistentVolumes
- Describe the relationship between container storage interface drivers, storageClass, persistentVolumeClaim and volume objects in Kubernetes

https://kubernetes.io/docs/concepts/storage/persistent-volumes/

1. Create a Persistent Volume named pv, access mode ReadWriteMany, storage class name shared, 512MB of storage capacity and the host path /data/config.

```
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
  name: pvspec
  capacity:
    storage: 512m
  accessModes:
    - ReadWriteMany
  storageClassName: shared
  hostPath:
    path: /data/config


k create -f pv.yaml
persistentvolume/pv created

k get pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS
REASON   AGE
pv     512m       RWX            Retain           Available           shared
64s
```

2. Create a Persistent Volume Claim named pvc that requests the Persistent Volume in step 1. The claim should request 256MB. Ensure that the Persistent Volume Claim is properly bound after its creation.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 256m
  storageClassName: shared

k create -f pvc.yaml
persistentvolumeclaim/pvc created

 k get pvc
NAME   STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc    Bound    pv       512m       RWX            shared         33s

k get pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM          STORAGECLASS
REASON   AGE
pv     512m       RWX            Retain           Bound    app-stack/pvc   shared
3m55s
```

3. Mount the Persistent Volume Claim from a new Pod named app with the path /var/app/config. The Pod uses the image nginx.

```
➜  learn_k8s k create -f app.yaml
pod/app created
```

```
➜  learn_k8s cat app.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: app
  name: app
spec:
  containers:
  - image: nginx
    name: app
    volumeMounts:
      - mountPath: "/var/app/config"
        name: configpvc
    resources: {}
  volumes:
    - name: configpvc
      persistentVolumeClaim:
        claimName: pvc
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

Practice Questions

1. Command that places image into registry? Ans: docker push
2. **Which network allows Docker Trusted Registry components running on different nodes to communicate and replicate DTR data?** Ans: dtr-ol
3. **Endpoint exposed by DTR can be used to assess the health of a DTR replica?** Ans: /health
4. **By default where do Docker manager nodes store the swarm state and manager logs?** Ans: /var/lib/docker/swarm
5. Deploy 4 replicas of nginx with single command? Ans: docker swarm create —replicas 4 —name myservice nginx
6. **You are using self-signed UCP certs and have a second DNS name that points to your internal controllers. When installing UCP, which flag should you use to add this additional name?** Ans: —san
7. .dockerignore - can be used to exclude files and directories from docker build context, can help in reducing image size. Needs to be present in root directory of context.
8. Docker stack can only run on docker swarm.

9. Running multiple containers from same image is not same as running multiple replicas of a service as in case of service there is fault-tolerance guaranteed by Swarm such that if one node goes down swarm will start the replicas on that node on the remaining nodes.
10. —no-cache is to not use cache for docker image build.
11. Entrypoint in docker file is executed everytime a container is created using the image.
12. docker image inspect <image> can be used to check the CMD and ENTRYPOINT in the image.
13. docker image rm or docker rmi — delete image.
14. json-file is default logging driver.
15. **sudo kill -SIGHUP $(pidof dockerd)** — makes the dockerd reload its configuration without killing the dockerd.
16. On Linux docker manipulates iptables rules to provide network isolation. https://docs.docker.com/network/iptables/
17. Containers running on same Docker network are part of same default “bridge” network and have connectivity to all ports of each other.
18. overlay networks can only be created in swarm cluster and they can only be created from manager node.
19. Default address pool for global scope (overlay) networks is 10.0.0.0/8.
20. If —default-addr-pool option specifies 10.0.0.0/16 as the address pool to use for the overlay network then if —default-addr-pool-mask-len is not specified it defaults to 24 and each subnet will be assigned sequentially a CIDR block of 10.0.x.0/24.
21. UCP - provides dashboard for swarm. It has **grants** (akin to ACLs) that are made up of **subject**, **role** and **resource set**. Grant defines which user can access what resources in what way.
22. By default all processes executed inside docker container run as root user in the container which is mapped to root user on the host. But this root user in container has limited privileges compared to root user on host and hence is safe. It does not have SYS_ADMIN capability that is needed for mount for example. We can use docker container run —user option to specify a different user than root user in the container. Alternatively the root user on container can be mapped to a non-root user on host using user namespace —userns (https://docs.docker.com/engine/security/userns-remap/).
23. **Secrets are immutable in docker swarm**. So in order to update a secret, we need to create a new secret, update the service to use this new secret (docker service update command ... this will cause service to be restarted by swarm), and then delete the old secret. Default location for secrets is /run/secrets and for configs it is /. Both config and secrets are encrypted in transit but only secrets are encrypted at rest also (not config). Secrets use ramdisk to mount the volume so they are never written to disk.
24. bind mount and tmpfs mount. tmpfs mount is only available on Linux.
25. LVM storages - loop-lvm is used for testing only(allows files on local disk to be treated as actual physical disk or block device**). direct-lvm is recommended for production use.**
26. Configs and secrets are not backed up when UCP is backed up.
27. To remove a manager node from swarm, first demote the manager node to a worker node and then remove it from swarm with docker swarm leave command.
28. Swarm autolock feature was created to encyrpt secrets in the Raft logs securely.

29. DCT = Docker Content Trust (to do with locking docker swarm). It provides the abilty to use “digital signatures” for verifying integrity and the publisher of all the data received from a registry over any channel.
    https://docs.docker.com/engine/security/trust/content_trust/
30. If we have 7 manager nodes and 3 ADs best way to distribute manager nodes across ADs will be 2-2-3 as to attain quorum (majority) we need 4 managers to be available even when one AD goes down. The other possible option is 3-1-3. But between 2-2-3 and 3-1-3:
    a. For 2-2-3: If AD1 or AD2 goes down, we have a fault tolerance of 1, If AD3 goes down we have fault tolerance of 0.
    b. For 3-1-3: If AD1 or AD3 goes down, we have a fault tolerance of 0, If AD2 goes down we have fault tolerance of 2 (6-4).
    c. Hence 2-2-3 is a better distribution as we have more cases with +ve fault tolerance.
31. DTR requires UCP to run. You need to install UCP on all nodes where you plan to install DTR.
32. **docker ps -s** — command used to get the disk space used by a running container.
33. To cleanup all dangling images — **docker image prune —filter dangling=true**
34. **overlay2** is the preferred storage driver on Linux. It is also the default.
35. If container1 is part of network net2 and we need to connect it to net1 then use — **docker network connect net1 container1**
36. To deploy nginx service only on worker nodes in swarm such that it is accessible externally on port 8000 use: **docker service create -p 8000:80 —constraint node.role=worker nginx**
37. To assign static ip to a container:

    docker network create —subnet 172.18.0.0/16 mynet123

    docker container run —net mynet123 —ip 172.18.0.22 -it ubuntu bash

38. To limit max memory used by container: docker container run **-m 512m** ....
39. —privileged flag enables all kernel capabilities for the container. A process running inside container can bypass most of controls such as kernel namespaces isolation and cgroups limitations. It is not deemed secure.
40. Mutual TLS (MTLS) = both server and client verify each other’s identity. It is used by swarm for data transmission between nodes in the cluster.

References

1. https://github.com/Evalle/DCA - A guide that maps each exam objective to official docker documentation.
2. Docker Deep Dive by Nigel Poulton - Course on PluralSight
3. Docker certified associate practice tests at Whizlabs - https://www.whizlabs.com/docker-certified-associate/

4. Raft consensus algorithm - http://thesecretlivesofdata.com/raft/
5. Go package template - https://golang.org/pkg/text/template/