Docker File , Networking & Swarm

1. Docker file

Dockerfile defines what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you need to map ports to the outside world, and be specific about what files you want to “copy in” to that environment

(both Dockerfile and dockerfile would be accepted, but not DockerFile, DOckerfilE)

Docker file Component:

FROM
The base image for building a new image. This command must be on top of the dockerfile.

MAINTAINER
Optional, contains the name of the maintainer of the image.

RUN
Used to execute a command during the build process of the docker image.

ADD
Copy a file from the host machine to the new docker image. There is an option to use an URL for the file, docker will then download that file to the destination directory.

```
# fetch node v4 LTS codename argon
FROM node:argon

# Request samplename build argument
ARG samplename

# Create app directory
RUN mkdir -p /usr/src/spfx-samples
WORKDIR /usr/src/spfx-samples

#Install app dependencies
RUN git clone https://github.com/SharePoint/sp-dev-fx-webparts.git .
WORKDIR /usr/src/spfx-samples/samples/$samplename

# install gulp on a global scope
RUN npm install gulp -g

# RUN ["npm", "install", "gulp"]
RUN npm install
RUN npm cache clean

# Expose required ports
EXPOSE 4321 35729 5432

# Run sample
CMD ["gulp", "serve"]

```

ENV
Define an environment variable.

CMD
Used for executing commands when we build a new container from the docker image.

ENTRYPOINT
Define the default command that will be executed when the container is running.

WORKDIR
This is directive for CMD command to be executed.

USER
Set the user or UID for the container created with the image.

VOLUME
Enable access/linked directory between the container and the host machine.

Best practices for writing Dockerfiles:

Docker can build images automatically by reading the instructions from a Dockerfile, a text file that contains all the commands, in order, needed to build a given image. Dockerfiles adhere to a specific format and use a specific set of instructions

- **Containers should be ephemeral:** mean that it can be stopped and destroyed and a new one built and put in place with an absolute minimum of set-up and configuration
- **Use a .dockerignore file:** To exclude files which are not relevant to the build, without restructuring your source repository, use a .dockerignore file.
- **Avoid installing unnecessary packages:** To exclude files which are not relevant to the build
- **Each container should have only one concern:** Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers.
- **Minimize the number of layers**

Some important links:

https://docs.docker.com/v17.09/engine/userguide/eng-image/dockerfile_best-practices/#build-cache

https://docs.docker.com/v17.09/engine/reference/builder/

Push images to Docker Hub:

```
docker login --username=<yourhubusername>
docker tag <imageid> yourhubusername/<reponame>:<tagname>
docker push yourhubusername/<reponame>
docker pull yourhubusername/<reponame>:<tagname>
```

Docker Hub Account: push docker images to the hub

Docker File Practice:

Create an empty directory **app**. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it.

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Create two more files called requirements.txt and app.py (When the above Docker file is built into an image, app.py and requirements.txt is present because of that Docker file's ADD command, and the output from app.py is accessible over HTTP thanks to the EXPOSE command)

requirements.txt

```
Flask
Redis
```

app.py

```
from flask import Flask
from redis import Redis, RedisError
import os
import socket

# Connect to Redis
redis = Redis(host="redis", db=0, socket_connect_timeout=2, socket_timeout=2)

app = Flask(__name__)

@app.route("/")
def hello():
    try:
        visits = redis.incr("counter")
    except RedisError:
        visits = "<i>cannot connect to Redis, counter disabled</i>"

    html = "<h3>Hello {name}!</h3>" \
           "<b>Hostname:</b> {hostname}<br/>" \
           "<b>Visits:</b> {visits}"
    return html.format(name=os.getenv("NAME", "world"), hostname=socket.gethostname(), visits=visits)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80)
```

How to build docker images from Dockerfile

```
docker build -t myfirstdocker .
docker images
docker run -itd -p 4000:80 myfirstdocker
docker ps
docker ps -a
docker start 03e9a3566b60
```

---

2. Docker Networking:

When we install docker by default it will create 3 networks.

docker network ls

```
[root@ip-172-31-30-165 ec2-user]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
b469e26c310e        bridge              bridge              local
f940a7666910        host                host                local
8bb20e494c5b        none                null                local
```

Bridge: Bridge plays very important role for whole networking systems in docker container.

Host: Host has very important role for only for docker host machine. It doesn't have responsibility to do networking with containers.

Null: Null network can be used when we don't want to allow any container to use network. Container cannot do any communication with any devices.

Default Network: The Bridge network represents the docker0 network present in all Docker installations. To change the network run below command

```
docker run --network=<NETWORK>

Syntax: docker run -it -d --network=none --name node6 ubuntu:latest /bin/bash
```

```
[root@ip-172-31-30-165 ec2-user]# docker run -it -d --network=none --name node
ubuntu:latest /bin/bash
293d188457175a7426ec1c38e8e83ac7c8859d8a29c7066d4cf16cda89f99b81
[root@ip-172-31-30-165 ec2-user]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
293d18845717        ubuntu:latest       "/bin/bash"         22 seconds ago
Up 22 seconds                           node6
```

When you go inside your container and try to install anything, it will not work because here we have attached null network to out container.

```
[root@ip-172-31-30-165 ec2-user]# docker exec -it 293d18845717 /bin/bash
root@293d18845717:/# apt-get update
Err:1 http://security.ubuntu.com/ubuntu xenial-security InRelease
  Temporary failure resolving 'security.ubuntu.com'
Err:2 http://archive.ubuntu.com/ubuntu xenial InRelease
  Temporary failure resolving 'archive.ubuntu.com'
Err:3 http://archive.ubuntu.com/ubuntu xenial-updates InRelease
  Temporary failure resolving 'archive.ubuntu.com'
Err:4 http://archive.ubuntu.com/ubuntu xenial-backports InRelease
  Temporary failure resolving 'archive.ubuntu.com'
Reading package lists... Done
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/xenial/InRelease  Temp
orary failure resolving 'archive.ubuntu.com'
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/xenial-updates/InRelea
se  Temporary failure resolving 'archive.ubuntu.com'
W: Failed to fetch http://archive.ubuntu.com/ubuntu/dists/xenial-backports/InRel
ease  Temporary failure resolving 'archive.ubuntu.com'
W: Failed to fetch http://security.ubuntu.com/ubuntu/dists/xenial-security/InRel
ease  Temporary failure resolving 'security.ubuntu.com'
W: Some index files failed to download. They have been ignored, or old ones used
 instead.
root@293d18845717:/#
```

==========

Create Custom (User Defined) Bridge Network:

1. **User-defined bridges provide better isolation and interoperability between containerized applications**.

2. **User-defined bridges provide automatic DNS resolution between containers**:

- Containers on the default bridge network can only access each other by IP addresses, unless you use the --link option, which is considered legacy. On a user-defined bridge network, containers can resolve each other by name or alias.

```
docker network create my-bridge-net
docker network ls
```

we also can define subnet ranges for user defined bridge network

```
docker network create --driver=bridge --subnet=192.168.0.0/16 my-bridge-net2
```

Delete custom default network

```
Docker network rm my-bridge-net2
```

If you want to run new container to custom bridge network

```
docker create --name my-nginx --network my-bridge-net --publish 6080:80 nginx:latest
```

Login to your existing ubuntu container and Install ping

```
apt-get install iputils-ping
```

Now try to ping any container which is on different network and same bridge network (EX: nginx container which we created in previous step and old ubuntu containers. And verify ping)

Note: Only Bridge can be created.
We CAN'T create host and none network like bridge network
`docker network create --driver=host myhost`

(If we try to create then Docker will throe error:
Error response from daemon: only one instance of "host" network is allowed)

Docker Network Linking: By default, two containers can't not talk to each other if they reside on two different networks.

Create first network "my-network-prakash-green"

```
docker network create my-network-prakash-green
```

Creation of container "c1" on network "my-network-prakash-green"

```
docker run -itd --net my-network-prakash-green  --name c1 busybox sh
```

Create second network "my-network-prakash-blue"

```
docker network create my-network-prakash-blue
```

Creation of container "c2" on network "my-network-prakash-blue"

```
docker run -itd --net my-network-prakash-blue  --name c2 busybox sh

docker exec -it c2 /bin/sh
```

ping c1 ------> packets dropped

How to connect two containers if they exist on 2 different networks?
docker network connect my-network-prakash-green c2

```
docker exec -it c2 /bin/sh
```

ping c1 -----> package passing

Docker log directory:

```
/var/lib/docker
```

Docker troubleshooting:

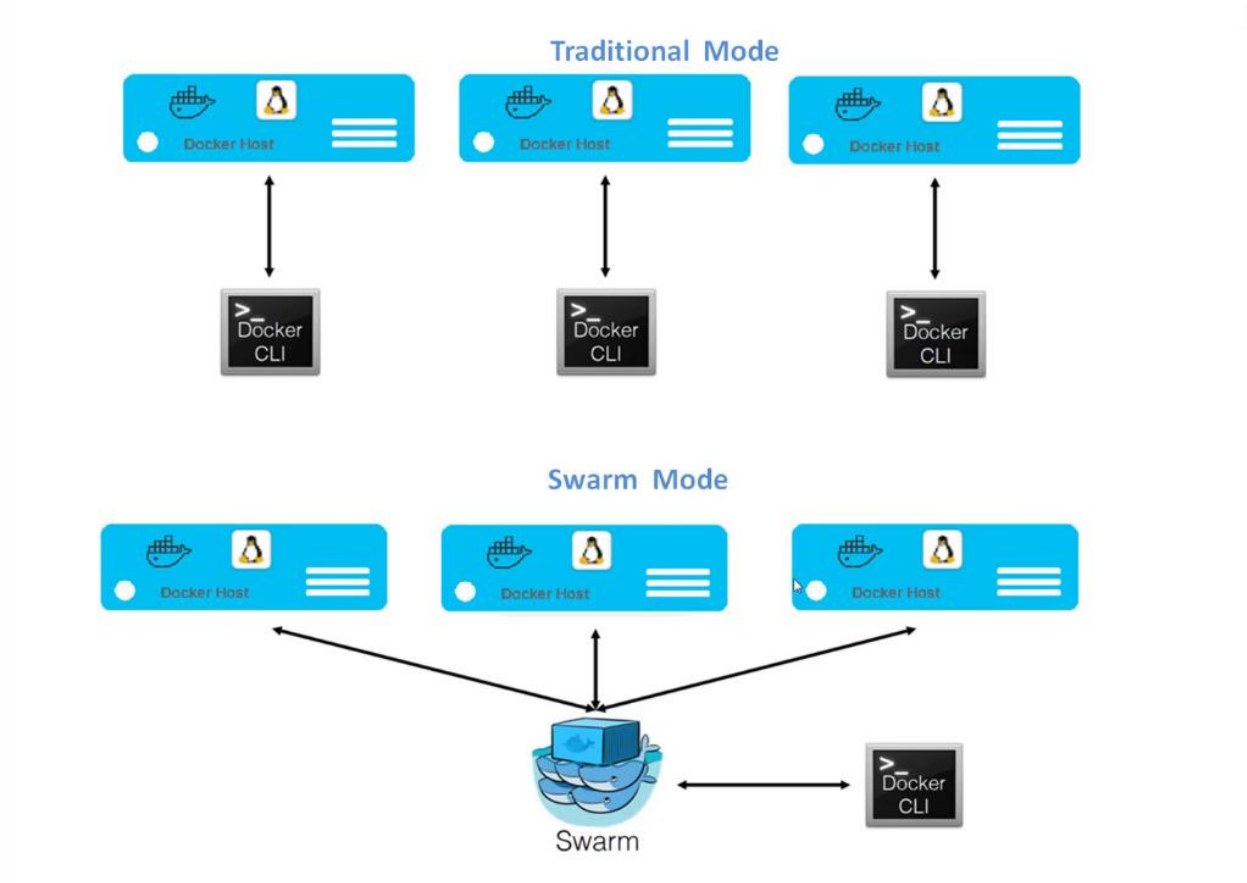
```
docker inspect <container Id>
docker inspect <image Id>

docker logs <container id>
docker logs –since 2018-05-25 <container id>
```

3. Docker Swarm

Swarm is native clustering for the Docker. In the context of swarm, a cluster is a pool of Docker hosts that acts as a bit like a single large docker host. You can also run swarm services and standalone containers on the same Docker instances.

The cluster management and orchestration features embedded in the Docker Engine are built using swarmkit. Swarmkit is a separate project which implements Docker's orchestration layer and is used directly within Docker.

A swarm consists of multiple Docker hosts which run in **swarm mode** and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles



Features of Swarm:

- **Cluster management integrated with Docker Engine:** Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don't need additional orchestration software to create or manage a swarm.
- **Decentralized design:** Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.

- **Declarative service model:** Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application comprised of a web front end service with message queuing services and a database backend.
- **Scaling:** For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.
- **Desired state reconciliation:** The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.
- **Multi-host networking:** You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.
- **Service discovery:** Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.
- **Load balancing:** You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.
- **Secure by default:** Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.
- **Rolling updates:** At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployments to different sets of nodes. If anything goes wrong, you can roll-back a task to a previous version of the service.

On Manager Node

1. Execute the following command with the manager nodes IP for initializing the swarm cluster.

```
docker swarm init
```

OR

```
docker swarm init --advertise-addr MANAGER
```

Run output command to your swarm nodes (to add all the nodes to cluster mode)

To know the swarm cluster info

```
docker info
```

To know the information about all the nodes in the cluster,

```
docker node ls
```

---IMP commands

docker swarm leave --- leave the swarm

docker swarm update –update the swarm

Promote a node (worker become manager) - docker node promote docker03

Demote a node (manager become worker) - docker node demote docker03

Docker swarm join-token worker --- to get worker token