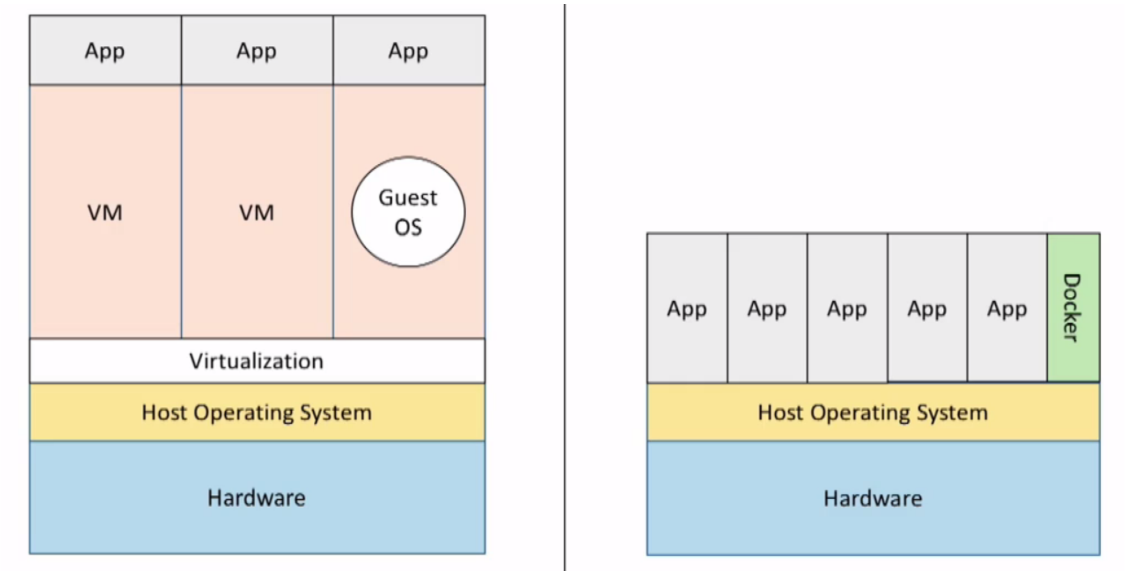
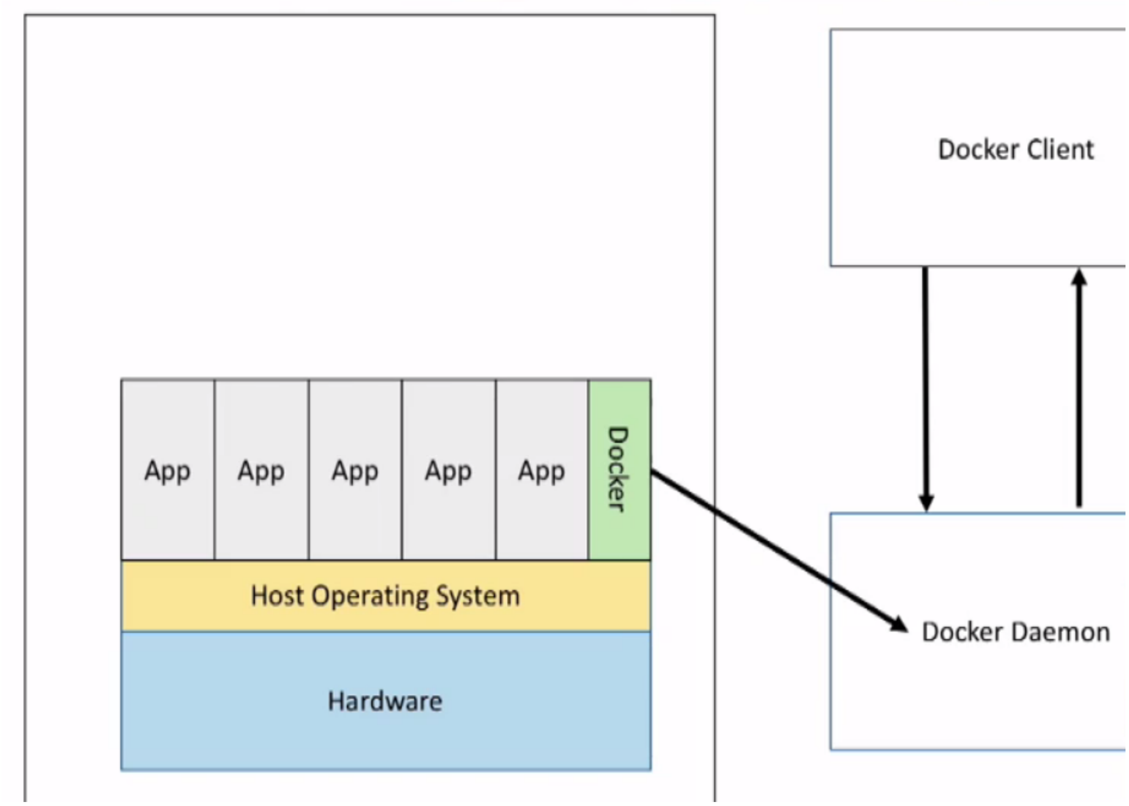
Docker

March 27, 2021

0.1 Overview





- `Docker Daemon` is like the server and you make requests to it to do things
- we mainly dont neeed to talk to the `Docker Daemon` because we have the `Docker Client` which we can talk to

```
docker run -it <CONTAINER> /bin/sh
```

- an docker `image` is like a class, a blueprint and you can create `containers` with it

- the `-d` tag is used to make the contaner a daemon
- `-p` to set the port

- docker kill `<container-id/container-name>`
- docker stop `<container-id>`
- docker rename `<previos-name>` `<new-name>`
- docker run –name `<new-name>` -itd `<running-worker>`
- docker container ls -a
- docker rm `<container-id>`
- docker run -itd –name `<new-container-name>` –restart=always `<container-name>`

- docker `rm` removes the image
- docker `rmi` removes the container

0.2 Inspecting Docker Images and Containers

- docker search `nginx`
- docker image ls
- docker top ‘

  - gives us a snapshot of whats running
- docker inspect worker
- docker stats worker
  - gives `CPU` usage or `MEMERY` usage
- docker log `<worker-name>`
- docker attach `<container-id>`
  - this lets you go into a container
- docker exec `<container-id>` ls
- docker run -it –name `<new-name>` `<container-id>`
  - lets you go inside of a container

0.3 Docker and Data

- data might not be safe inside of a container

- docker run -t -v `<local:/container>`
  - the `v` signifies a volume and links containers
  - docker run -t -v testData:/testData
  - make sure the folders are linked correctly

- docker create -v `<local-dir>` – name `<container-name>` `<image-name>`
  - docker create -v /data –name datastore busybox
  - this command creates a container with an associated volume
  - type `docker container ls --all` to see it

- docker run -it –volumes-from `<container-name>` – `<image-name>`
  - docker run -it –volumes-from datastore –name worker busybox
  - this command links the data connection created earlier to a new container
  - multiple reads are good but multiple writes are bad!

0.4 Docker and Data Use Case

- docker run -d -p 27017:27017 – `<new-name>` `<image-name>`
  - docker run -d -p 27017:27017 mongodb mongo

- sometimes data can be lost so you want to just have a data layer in a seprate container

- create a volumes conenction to
  - docker creater -v /data/db –name mongo_data mongo
- create a container which has access to that volume
  - docker run - -p 27017:27017 –name mongodb –volumes-from mongo_data mongo

0.5 Building our First Docker Image

- a `Dockerfile` is something that lets you wrap your application in its runtime and your container becomes the application
- each container should only do one thing

- first line starts with a `FROM`
  - basically asks where do you want to pull from
  - `FROM ubuntu:20.04`
- `CMD` lets you exectute a command once the docker file is executed

    - `CMD ["echo", "hello there"]`
- note double quotes matter for docker

- `docker build .`
    - builds the docker image in the directory
    - docker images can be used to see the docker file
- `docker build -t baseimage .`
    - `-t` allows you to rename or tage images
    - `-t` if you retag, you can use updated versions, etc.
- if you just change the `tag` name without any image changes, then you dont actually store the image multiple times, but just the refrence
- even if you change some code, only the part that changes takes up more space, the rest is just refrenced

- `RUN` is like a bash command
    - `RUN apt-get update -y`
    - `RUN apt-get nginx -y`
    - the `-y` asks you to confirm to build

- `CMD ["nginx", "-g", "daemon off;"]`

- `EXPOSE 80`
- docker run -d -p 80:80 baseimage

- `ENTRYPOINT` is simmilary to `CMD`
- ‘ENTRYPOINT [‘nginx’]
- ‘CMD [“-g”, “daemon off;”]
- from my estimation, `CMD` is used to pass arguments but the main feature is passed to `ENTRYPOINT`
- you can use `&&` or other boolean operators if you want
- try not to use `&&` because debugging becomes a nightmare

0.6 Storing Our Custom Docker Images

- you can try to move them to `tar` files, etc
- most common way people use are registeries
- if you are pushing to docker you need to rename to `<docker-username>/<image-name>`

- docker `commit` -m `"<message>" <container-id>`
    - can be used to take a snapshot of the container
- imagine you have a lightweight base container that you want to modify
- you want to have a simple server and add a `html` file to serve
- well you can ssh into the container, add files, and use `commit` to create a version of that container
- normally you might want to use the `Dockerfile` to build that

- docker image history `<container-id>`
- images have a `history` tag that keeps track of the comment messages
- if you mess up with you can do is, rename image, delete it locally using `docker rmi <container-id>` and then commit agian

- after you do a docker `commit`, you have to do a docker `push`

- if you want to not push to repository but want to export it for local use you can use export command
- docker export `<container-name>` - o `<container-name>`

- there is also a `save` paramater
- docker `save <container-name>` -o `<container-name.image.tar`

- `export` and `save` are the same
- probally better to `save` the image then have to run a container and `export` the container

0.7 Building an Application with Docker

- docker allows you to avoid having to set up server everytime you want to deploy
- `ADD <file-path>` is used to add a file in a directory
- `WORKDIR <file-path>` is used to decide where you container is going to start
  - `docker run ...` will put you in the he specified folder
- `ADD` vs `COPY` -`ADD` is for urls and stuff
  - `COPY` is for directories
  - `COPY` is the bare minimum and `ADD` has extra functionalities
  - for example, in some cases `ADD` will actually un-zip tar files, etc.

```
FROM ubuntu:20.04
RUN apt-get update -y;
RUN apt-get install curl -y;
RUN curl -sL https://deb.nodesource.com/setup_6.x | bash -;
RUN apt-get install nodejs -y;

COPY index.js /app/index.js
COPY package.json /app/package.json
WORKDIR /app
RUN npm install
CMD ["node", "index.js"]
```

- if you cant kill a docker container with `CTRL + C` just use `docker kill <container-id>`

- one neat thing you can do is simply create a docker image and then pull from that docker image
- if you see repeated code, you can just think of it as a function
- each docker file should do one thing

```
FROM `<prviously-built-image>`

COPY index.js /app/index.js
COPY package.json /app/package.json
COPY data.json /app/data.json

WORKDIR /app
RUN npm install

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

0.8 Multi-Container Apps with Docker

- `VOLUME` is creating a folder inside of a container
- `EXPOSE` is to expose a port

- old way of running two containers together is using `--link redis:redis`
- this is no longer reccommended

- the reccomended way to do this is to use `networks`
- docker network create `<network-name>`
- we simply add the apps to the network
- docker run -d -p 5000:5000 –name redis `--net <app-name> redis <web-app-name>`

- when adding a database remember you need to create a data volume link

- `docker create -v /data/db --name mongo_data mongo`
- `docker run -d --name mongo --net webapp --volumes-from mongo_data mongo`
- `docker run -d --name redis --net webapp redis`
- `docker run --net webapp --name app -p 5000:5000 <container-name>`

0.9 Docker Compose

- an orchestration management tool for docker
- for volumes you can use absolute paths, relative paths or named volumes
- images built with compose prepends the app name with parent folder
- `docker-compose up` to run
- `docker-compose stop` to gracefully stop
- `docker-compose up -d`

```
version: '3'
services:
    app:
        build: ./app
        ports:
            - "8080:8080"
         volumes:
             - '/nodecompose'
        networks:
            - webapp
    redis:
         image: redis
         posts:
             - "6379:6379"
         networks:
             - webapp
    mongo:
        image: mongo
        ports:
            - "27017:27017"
        volumes:
            - mongo_data:/data/db
        networks:
            - webapp

 volumes:
     mongo_data:
         driver: local

 networks:
     webapp:
         driver: bridge
```

0.10 Docker Machine

- a method of provisioning virtual machies
- simmilar to vegrant
- a container like a `virtual machine` but in reality its just a file system that shares underlying system processes
- docker machine comes installed in windows/mac but not linux
- this is a good way to simulate a cluster

- `docker machine create`provisions virtual machines with different drivers
  - drivers are like `virtual-box` or `vagrent` or `digital-ocean`
- `docker create` provisions new containers, new volumes

- benefit of using `docker machine` over just vagrent is that `docker-machine` sets up all of the docker stuff you need
- you can just ssh into it and start using it

- there is an `scp` command that lets you copy files between machines

- `docker-machine create -d virtualbox node-0`
- `docker-machine ssh node-0`
- `docker-machine scp hello.txt node-2:hello.txt`
- `docker-machine scp ssh node-2 ls`

0.11 Docker Machine with Docker

- export `DOTOKEN=<token>` to set enviorment
- `docker-machine create -d digitalocean --digitalocean-access-token=<DOTOKEN-ENV> node0`

- `docker machine regenerate-certs` for erros in ssh

0.12 Docker Swarm

- idea of a swarm is to provide a management tool for a cluster of compute nodes that you can distribute containers across
- enables you to horizontally scale your container across multiple servers
- you can make an `EC2` instance join a swarm

- you need to have multiple servers

- these can be multiple `EC2` instances or just can be created with `docker-machine`

- `docker swarm --init --advertise-addr <docker-machine-ip>`
  - other nodes can basically join the
  - this gives you a token to join
  - if you add workers, they get a `swarm` tag
- `docker-machine ssh node-1 <SWARM-TOKEN>`
- `docker-machine ssh node-2 <SWARM-TOKEN>`
- `docker-machine ssh node-3 <SWARM-TOKEN>`

- `docker node ls`
  - `node` is for managing swarm
  - gives you info about all your workers in the swarm
  - you can make anyone else a manager, remove them, etc

- `docker service --replicas 2 --name webserver nginx:apine`
- `docker service ls`
- `docker service scale webserver=9`

- docker swarm handles how the replicas are spread out
- you can try to set the policy on how the containers are distributed

0.13 Docker Swarm Digital Ocean

- `for i in 1 2 3; do docker-machine create -d digitalocean --digitalocean-access-token=<DOTOKEN-ENV> node<i>; done`
  - creates docker virtual instances
- `docker machine ssh node-1`
- `docker swarm init --advertise-adder <SWARM-QUEEN-IP>`
- `docker swarm join-token worker`
- `docker-machine ssh node-2 <SWARM-TOKEN>`
- `docker-machine ssh node-3 <SWARM-TOKEN>`
- `docker node ls`
- `docker service create --name webserver -p 80:80 --replicas 12 nginx:alpine`

0.14 Creating a Docker Swarm Application

- a docker swarm can be handled with docker-compose
  - the `deploy` tag used to handle swarm
  - you also have to set a `driver` which is sets a single point outsiders see and everything conencts through that
- `depends_on` is like a soft conditional statement that will just check that the container is running, not the application in the container

```
version: '3'
services:
    redis:
        image: redis:3.2-alpine
        ports:
            - "6379"
        networks:
            - webapp
         deploy:
             placement:
                 - constraints: [node.role == manager]
    mongo:
        image: mongo
        volumes:
            - mongo_data:/data/db
        networks:
            - webapp
        deploy:
            placement:
                constraints: [node.role == manager]
        app:
            image:<DOCKER-REGISTRY-NAME>
            ports:
                - "5000:5000"
            networks:
                - webapp
            depends_on:
                - redis
                - mongo
        deploy:
            mode: replicated
            replicas: 2

networks:
    webapp:
        driver: overlay

volumes:
    mongo_data:
```

- you have to use `docker-cloud` and not `docker swarm`
- the compose file we created above is actually a `stack` file
- its constructed as a compose file but its actually used to describe an application that is a stack of services
- you have to make sure the swarm is set up
- you need to transfer the docker-compose file we created above to go to the leader
- `docker-machine scp docker-comosoe.yml node-1:docker-compose.yml`
- ‘docker stack deploy –compose-file docker-compose.yml
- `docker node ls`