Commands

1. Docker Machine (DM) Commands. (All at https://docs.docker.com/machine/reference/)

a. Commands (DM = docker-machine)
   a. User runs commands below from outside the Linux VM (ie, OsX terminal)
      a. Host machine (Unix in OsX case) w/docker installed, can create a docker-machine which creates the Docker_Host, Docker Server/Engine/Daemon & configures a Docker-Client

   b. Create a docker-machine
      $ docker-machine create --driver virtualbox dev
         i. --driver (or just -d)
         ii. provider machine is created on: digitalocean, aws, virtualbox (local)
         iii. dev name of machine
         iv. Examples
            1. $ docker-machine create -d digitalocean consul
               a. name is “consul”
            2. $ docker-machine create --driver digitalocean --digitalocean-access-token <xxxxx> docker-app-machine
               a. name is docker-app-machine
   c. $ docker-machine → list all commands
   d. $ docker-machine ls → list all DMs
   e. $ docker-machine active → list active DM
   f. $ docker version → displays versions of client and servers

   g. $ docker-machine start [machine name]
   h. $ docker-machine stop [machine name]

   i. $ docker-machine env [machine name] → connect term & virt box
   j. $ docker-machine ip [machine name] → ip address of machine

   k. $ docker-machine status [machine name] → show virt box stat
   l. $ docker-machine ssh consul ifconfig
      a. Above runs command ifconfig from within the docker-machine named “consul”

2. Images Commands

(note if commands not working, terminal likely connect to OsX. Run Quickstart Terminal)

a. $ docker - lists all available commands
b. $ docker build -t <i-name> **.** - create image (w/name i-name) from Dockerfile in cwd
c. $ docker pull [image name] - downloads from docker.hub
d. $ docker run [image name] - runs the image (& pulls if not local)
e. $ docker inspect [image name] - shows info on container (JSON) or image
f. $ docker run <image-name> sleep 1000 - stop container after 1,000 seconds
g. $ docker run -t -i busybox:1.24 - runs image (busybox:1.24) in interactive mode
h. $ docker images - lists all images
i. $ docker history <image-name> - see layers in image
j. $ docker build -t dougwells/debian **.** - **.** = cwd (works if Dockerfile is in cwd
k. $ docker -t <image_id> <new_name>:tag - Rename image,
l. $ docker rm**i** 198 - removes **image** (only need first few parts of identifier)
m. $ docker rmi $(docker images -q -f dangling=true) - delete all untagged image
n. $ docker rmi $(docker images -q) - delete ALL images
o.

2. Container Commands

a. $ docker - lists all available commands
b. $ docker run [image name] - runs image (pulls if not local), creates/starts container
c. $ docker run -it -p **8080**:**3000** tomcat:8.0 -create & start container from image
d. $ docker start [cont ID] - starts an existing container
e. $ docker stop [cont ID] - stops a container (does NOT delete)
f. $ docker rm [cont ID] - deletes. Add -f tag if need to force
g. $ docker ps - lists all containers running
h. $ docker ps -a - list all containers (running & stopped)
i. $ docker logs [cont ID] - show logs of running container
j. $ docker inspect <cont> - shows info on container (JSON) or image
k. $ docker run <image-name> sleep 1000 - stop container after 1,000 seconds
l. $ docker network ls - list docker networks
m. $ docker network inspect [ID] - network info (including containers in it)
n. $ docker exec [cont ID] xyz - execute command in running container
o. $ docker exec -it <cont_id> command - “ “ & enter interactive mode
p. $ docker exec d64 node seed.js - as if in console (d64 $ node seed.js)
q. $ docker exec d64 bash - gain access to container’s bash shell
r. $ docker exec -it d64 /bin/bash - enter container d64 & attach bash shell
s. $ docker rm 198.. - deletes container (only if it is stopped)
t. $ docker rm -f 198 - removes/force deletes container (even if running)
u. docker stop $(docker ps -a -q) - stop ALL containers
v. $ docker rm $(docker ps -a -q) - delete ALL stopped container
w. $ docker system prune - delete ALL unused data
(stopped cont, vol w/o container & images w/ no cont)
x. --rm → remove container when stopped
y. --name <contName> → name the container once it is built from image

1. Volume Commands

a. $ docker rm [container id] - removes container, not volume (b/c mult cont)
b. $ docker rm **-v** [container id] - removes container AND volume (but local file remains) (only delete volume when ALL containers are done using it)
c. $ docker volume - list all volume commands
d. $ docker volume ls - lists all volumes
e. $ docker volume rm `docker volume ls -q -f dangling=true` - remove all dangling volumes

2. Docker Network/Bridge/Linking Commands

a. $ docker network ls - list docker networks
b. $ docker network inspect [ID] - network info (including containers in it)
c. $ docker disconnect [cont ID} - disconnect a container from a network
d. $ docker network rm [ntwk ID] - remove entire network

1. Docker Compose Commands

a. docker-compose build → When need new images. ALL IMAGES NEW
b. docker-compose build mongo → just build one service. In this case, mongo
c. docker-compose up -d → create & start containers as running services
("up" does NOT rebuild images. Use "build")
d. docker-compose up --no-deps node → can add additional command lines
e. docker-compose down → stops AND REMOVES all containers/services
f. docker-compose down -rmi all --volumes → also removes all images & volumes

g. docker-compose stop → stop services/containers. Does NOT remove
h. docker-compose start → start the services/containers back up

i. docker-compose logs -f → view the logs. -f = follow (outputs new logs)
j. docker-compose cont-name -f → views logs of just that container

k. docker-compose ps → all containers managed by docker compose
l. docker-compose rm → remove ALL containers managed by docker-compose
m.

n. Differences between start & up
    1. $ docker-compose start
        a. simply starts existing containers for a service.

    2. $ docker-compose up
        a. Builds, (re)creates, starts, and attaches to containers for a service. (NOT images)
        b. Linked services will be started, unless they are already running.
        c. By default, docker-compose up will aggregate the output of each container and, when it exits, all containers will be stopped. Running docker-compose up -d, will start the containers in the background and leave them running.
        d. By default, if there are existing containers for a service, docker-compose up will stop and recreate them (preserving mounted volumes with volumes-from), so that changes in docker-compose.yml are picked up. If you do not want containers stopped and recreated, use docker-compose up --no-recreate. This will still start any stopped containers, if needed.
        e. Source: https://docs.docker.com/v1.5/compose/cli/

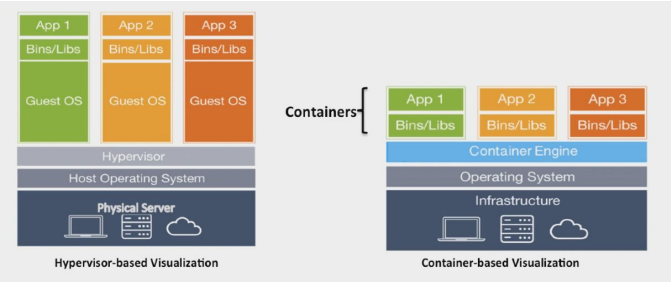
        f.

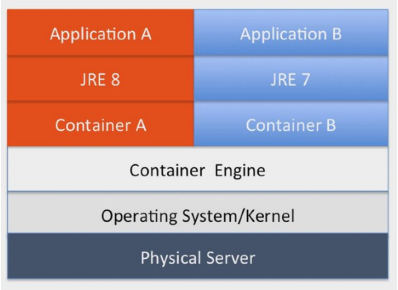
1. What is Docker?

a. (nice intro article at: https://www.jayway.com/2015/03/21/a-not-very-short-introduction-to-docker/)
b. Java's promise: Write Code Once. Run Anywhere
c. Docker's promise: Configure Runtime Once (per image). Run it Anywhere
    i. Instead of code, you can configure your runtime exactly the way you want them (pick the OS, tune the config files, install binaries, etc.) and you can be certain that your runtime template will run exactly the same on any host that runs a Docker server.
    ii. Docker's collaboration features (docker push and docker pull) are one of the most disruptive parts of Docker. The fact that any Docker image can run on any machine running Docker is amazing. But The Docker pull/push are the first time developers and ops guys have ever been able to easily collaborate quickly on building infrastructure together. The developers can focus on building great applications and the ops guys can focus on building perfect service containers. The app guys can share app containers with ops guys and the ops guys can share MySQL and PosgreSQL and Redis servers with app guys.
    iii. This is the game changer with Docker. That is why Docker is changing the face of development for our generation. The Docker community is already curating and cultivating generic service containers that anyone can use as starting points. The fact that you can use these Docker containers on any system that runs the Docker server is an incredible feat of engineering.

2. Docker utilizes container handling features built into Linux & Windows Server 2016
    a. On Mac or PC, we need a Linux Virtual Box machine to access these capabilities

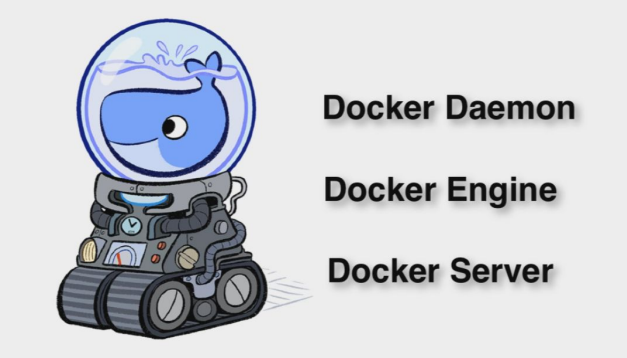
3. Container model is much more efficient than virtual machine model for achieving better resource utilization and portability
    a. Operating System/Kernel - Docker-Machine (gives us a lightweight linux VM)
    b. "Container Engine" - Docker Daemon/Server/Engine - Supports multiple containers. One engine per docker-machine
    c. Containers - Each container can have its own runtime environment. Multiple containers OK
    d. Swarm - multiple Docker-Machines, each w/1 docker-engine. Each docker-engine can have multiple containers.





1. Explaining Docker imagery (Whales, Containers, Octopuses … Oh my !!)

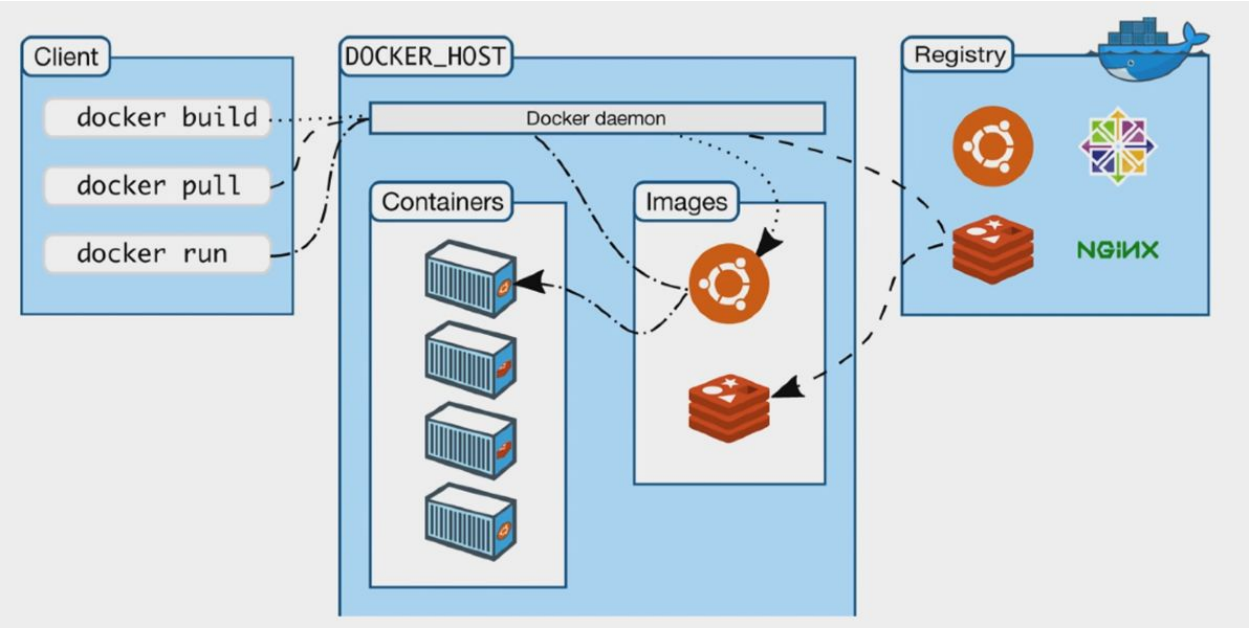
a. Fishbowl - Docker-Machine. (linux OS itself or docker-machine if in cloud, Mac or Windows)
b. Docker Whale = Docker Engine / Docker Server / Docker Daemon
c. Octopus = Docker-Compose
d. Containers on Whale = Containers themselves (images → container)
    i. Spun up container is called a “service”
e. Multiple Fishbowls = Swarm. Each “fishbowl” has just 1 whale inside.
    i. Multiple containers per whale



Docker Client
(Communicates w/Docker daemon)

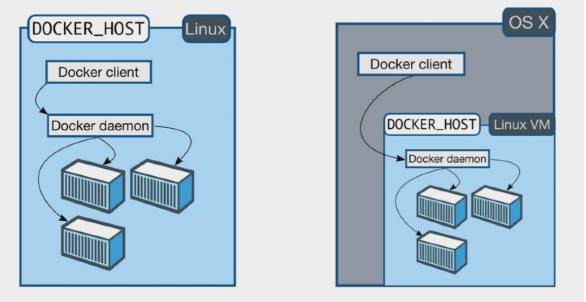
Docker Host (active Docker-Machine)
(Lightweight Linux Environment. Docker daemon runs in)

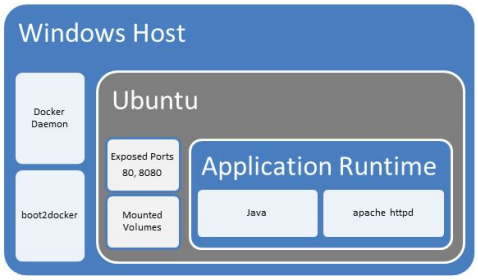
Docker Hub
(Images)


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

1. Docker Toolbox - Installing Docker on your computer gives you all 3 ...
    a. 3 “Tools”
        i. Docker-Machine → creates lightweight linux box. Docker Dameon runs in here
        ii. Docker Client → humans use Docker Client to communicate w/Docker Daemon
        iii. Docker Compose → tools to handle multiple containers
        ---
        iv. Docker Swarm → not in toolbox. Tools to help manage multiple Docker Daemons

2. Docker Client/Server setup





    a. Docker Machine (aka DOCKER_HOST environment. Lightweight Linux VM)
        i. docker-machine IS the DOCKER_HOST. One & the same.
        ii. The Docker-Machine (Docker_Host) is a lightweight Linux VM
        iii. Docker-Machine: Linux environment in which docker server/daemon/engine runs inside
            1. Docker CAN run natively on Linux (no docker-machine needed)
            2. Docker CAN NOT run natively on OSX/Win (needs a Linux VM)
        iv. Can see this Linux VM by opening Oracle VM VirtualBox Manager
            1. OSX → use Spotlight Search “Virtual”
        v. Two popular Linux distributions for desktop
            1. Ubuntu - Most user friendly. African word for “community spirit”
            2. Debian - Also very popular. Came first. Slower release cycle

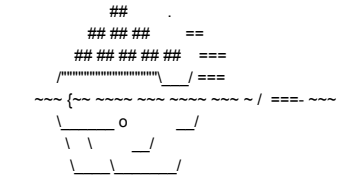
    b. Docker Server - Builds, runs & distributes docker containers.
        i. AKA: Docker Daemon, Docker Server, Docker Engine
        ii. Can run in daemon mode or interactive mode
            1. Daemon simply means program runs in the background.
            2. Interactive simply means human user is inputting each command

c. Docker Client - communicates with Docker Server/Engine/Daemon

i. Communicates with Docker Engine (Daemon/Server) which manages containers & images

ii. Can use CLI (docker quickstart terminal) or Kitematic (GUI) to interface with docker client. The docker client is the "terminal" for how we communicate with Docker Engine/Daemon/Server

1. Docker Quickstart Terminal
    a. Provisions a Docker-Machine (a Linux VM. Docker_Host)
    b. Each Docker-Machine is a lightweight Linux Virtual Machine/Box
    c. opens docker client terminal which is communicating w/docker-server (CLI prints out "whale" icon)



docker is configured to use default machine w/ IP 192.168.99.100
$

1. IP 192.168.99.100 = ip address of linux virtual machine

iii. Active docker-machine is called Docker_Host. This is the environment which hosts the Docker Server that docker client is communicating with

1. To see which docker-machine's daemon that the client is attached to
    a. $ docker-machine ls → (* = "active")

2. Connect/change client to the daemon in any docker-machine
    a. $ docker-machine env [docker-machine-name]
        i. List environment variables of DM.
        ii. Gives you IP address & command to point Docker Client to that IP
        iii. Pointing Docker Client to a DM's IP makes it the DOCKER_HOST
    b. $ eval $(docker-machine env name-of-docker-machine)

3. Connect docker client to swarm:

$ eval $(docker-machine env -swarm name-of-swarm-master)

iv. Terminal: Docker Quickstart Terminal: Returns a docker client terminal that is connected to daemon inside active docker-machine (indicated by *)

a. Manually connect docker client to desired docker-machine
   i. $ eval $(docker-machine env name-of-docker-machine)
b. $ docker-machine ls will show “active”
   i. * = docker client is currently communicating with this docker-machine/server

v. Can run docker server in daemon mode or interactive mode

1. Daemon simply means program runs in the background.
2. Interactive simply means human user is inputting each command
3. Framework for thinking about interactive vs daemon
   a. YES: Docker Server is currently running a program loaded into it. How do I interact directly w/docker server to manually give it some commands to run
   b. YES: "How do I enter/exit interactive mode
   c. YES: “How do I return to daemon mode"
      i. ctrl-p & then ctrl-q
      ii. http://stackoverflow.com/questions/25267372/correct-way-to-detach-from-a-container-without-stopping-it
   d. NO "How do I enter/exit the container"

4. Interactive flags

a. -i → interactive
b. -t → creates psuedo TTY with stdin & stdout
   1. TTY = TeleType Writer (originally for deaf people) `tele` comes from a Greek root meaning far or distant. TTY's provide a virtual interface similar to what the physical machines provided. This is the origin of the 80 char width and carriage return

5. Examples

a. $ docker run -t -i busybox:1.24
   i. Gives you terminal inside the container. Prompt looks like / #
      1. / # exit → returns to Daemon mode & SHUTS DOWN CONTAINER

      OR:

2. ctrl-p & then ctrl-q → returns to daemon & KEEPS CONTAINER running.

b. To enter interactive mode once container running …
   i. $ docker exec -it <cont_id> command
      1. exec lets you run a command inside a running container
      2. -it → enter interactive mode
      3. bash → run bash shell
   ii. Example → $ docker exec -it 175c bash
      1. See Running programs → $ ps axu
      2. (ps axu is just a shell command)
   iii. Exit interactive mode (& return to daemon mode) → ctrl-p & ctrl-q

c. "Daemon" Terminology
   i. In multitasking computer operating systems, a daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive user. Traditionally, the process names of a daemon end with the letter d. For example, syslogd is the daemon that implements the system logging facility, and sshd is a daemon that serves incoming SSH connections.

ii. Why called daemon?

1. The term was coined by the programmers of MIT's Project MAC. They took the name from Maxwell's demon, an imaginary being from a thought experiment that constantly works in the background, sorting molecules. Unix systems inherited this terminology. Maxwell's Demon is consistent with Greek mythology's interpretation of a daemon as a supernatural being working in the background, with no particular bias towards good or evil. However, BSD and some of its derivatives have adopted a Christian demon as their mascot rather than a Greek daemon.

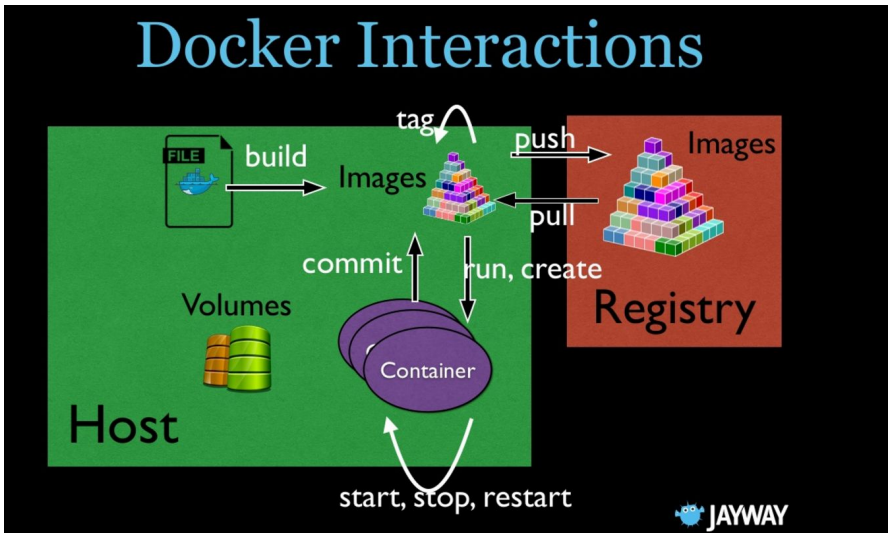
2. The word daemon is an alternative spelling of demon and is pronounced /ˈdiːmən/ dee-mən.

3. Alternate terms for daemon are service

4. After the term was adopted for computer use, it was rationalized as a "backronym" for Disk And

Execution MONitor.

3. Docker Interactions



4. What is difference between Docker-Machine and Docker Engine?

https://docs.docker.com/machine/overview/

a. Docker Engine runs on Linux. If you are not running natively on Linux, Docker-Machine gives your a lightweight Linux VM to serve as Docker_Host. Docker Engine and containers reside INSIDE the Docker_Host.

b. If you have a Linux box as your primary system, and want to run docker commands, all you need to do is download and install Docker Engine. However, if you want an efficient way to provision multiple Docker hosts on a network, in the cloud or even locally, you need Docker Machine.

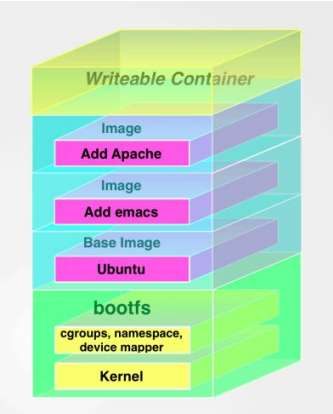
c. Whether your primary system is Mac, Windows, or Linux, you can install Docker Machine on it and use docker-machine commands to provision and manage large numbers of Docker hosts. It automatically creates hosts, installs Docker Engine on them, then configures the docker clients. Each managed host (“machine”) is the combination of a Docker host and a configured client.

d. When people say “Docker” they typically mean Docker Engine, the client-server application made up of the Docker daemon, commands (via APIs) for interacting with the daemon, and a command line interface (CLI) client that talks to the daemon (through the REST API wrapper). Docker Engine accepts docker commands from the CLI, such as docker run <image>, docker ps to list running

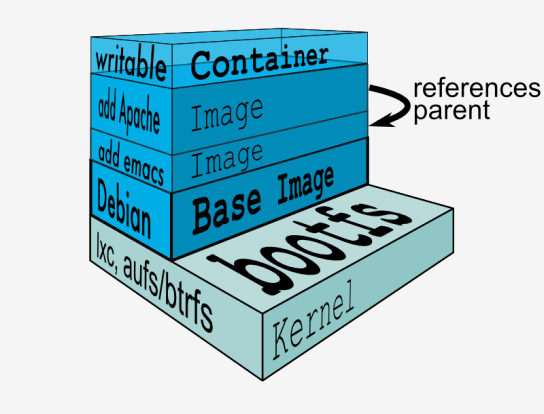
containers, docker images to list images, and so on.

e. Comments from Udemy Instructor James Lee
    i. Multiple hosts means multiple VMs.
    ii. Each docker machine is effectively a VM.
    iii. Docker_Host points to the IP of your active VM.
    iv. If you look at the droplets section under your digital ocean account, you should see multiple droplets, each droplet is a an individual VM.
    v. So yes Digital Ocean creates different hosts for you and docker-machines run on multiple hosts on Digital Ocean. So we need a swam manager to manage all the hosts which will operate transparently to the end users. This is the whole point of Docker Swarm.

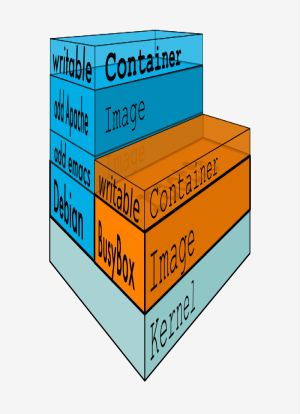
1. Docker Images



Bootfs - Linux VM (Docker-Machine)



Single Container (Container merely a r/w layer on image)



2 Containers (1 Server in DM/Linux/Kernel)

a. Images are read-only templates that are used to create containers
   i. Created with $ docker build command
   ii. Composed of "layers" of other images
   iii. Can see these layers $ docker history <image-name>

b. Stored in a Docker registry
   i. Registry is where we store our images (ie, DockerHub.com, Github)
   ii. Repository - same image but different versions & tags
      1. Repository is just like a github repository
      2. versions / tags - like git commits and their associated messages

c. DockerHub.com
   i. Registry for Docker images
   ii. Note the "Official" images
      1. Reviewed by Docker Team. Created for general use cases
      2. Updated with Security patches as they become known
      3. Good place to start is with "Official" Docker images

d. Two ways to create an image

i. Commit changes made to a container (but not pushed yet to dockerhub.com)

1. Example: Create Debian Linux base with git installed
2. $ docker run -it debian:jessie (-it = interactive so get terminal)
3. /# apt-get update && apt-get install -y git (apt-get is package installer like npm)
4. Commit changes to your DockerHub account
   a. $ docker commit <cont_ID> <user_Name>/<image_Name>

b. $ docker commit [OPTIONS] <cont_ID> [REPOSITORY/img_name:TAG
   i. -m, --message string Commit message

ii. From a Dockerfile

1. Dockerfile is merely a text file w/ instructions for building an image
   a. Dockerfile must start with capital "D" (or end in .dockerfile)
2. Great tutorial on Dockerfile commands by Digital Ocean
   a. https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images

3. Example: Dockerfile

```
FROM debian:jessie
RUN apt-get update
RUN apt-get install -y git
RUN apt-get install -y vim
```

e. Build image from Dockerfile

i. $ docker build -t <image_name> <Dockerfile relative location . = cwd>
$ docker build -t dougwells/debian . → . = cwd (works if Dockerfile is in cwd)
$ docker build -t dougwells/debian Dropbox/1-Code/Docker/DockerBasics2
   1. Since image built with Dockerfile, no version #. Instead tagged w/"latest"
      a. Docker images → dougwells/debian:latest
   2. Avoid temptation to use :latest. Rather, give version #

ii. Rename image, use $ docker tag
$ docker tag <image_id> <new_image_name>:tag
   1. Remember, try to avoid leaving off tag (defaults to "latest")
   2. But, if building new image with this image as a layer, if your existing layer in cache already has "latest", it won't update b/c docker thinks you have the latest (which you might or might not)

f. Image Naming cautions:

i. Build manually (ie, $ docker build -t dougwells/debian .)
   1. Image name = Image name given
   2. Example node or dougwells/node

ii. Build image locally with docker-compose.yml
   1. Image name = <folder of docker-compose.yml>_<service name given in docker-compose-yml>

iii. Build image in CircleCI - (built from github repo)
   1. Image name = <gh repo name (with dashes [-]removed) >_<service name given in docker-compose.yml

g. Pushing image to DockerHub.com (James Lee)

1. Make sure image has format <DockerHub_Username>/<image_Name>
    a. If not, rename it using “docker tag” (‘tagged” w/ a name)
    b. $ docker tag <current_img_id> <new_img_name>:tag
2. Login to Docker Hub from CLI
    a. $ docker login --username=dougwells -- or --
    b. docker login
3. Then push to Docker Hub
    a. $ docker push <repository_name>/<image_name>:tag
    b. Example: $ docker push dougwells/debian:1.01

h. Publishing images to Docker Hub (Dan Wahlin)

i. $ docker login → must be logged in first
ii. $ docker push <your username>/<image-name>
    1. Must use this format <username>/<image-name> or will not push
    2. If need to rename image to follow naming convention
        a. docker tag <image_id> <new_image_name>:tag
            i. Remember, try to avoid leaving off tag (defaults to “latest”)
            ii. But, if building new image with this image as a layer, if your existing layer in cache already has “latest”, it won’t update b/c docker thinks you have the latest (which you might or might not)
iii. $ docker pull <filename>
iv. $ docker run <filename> → (docker will download image if not on machine yet)

Dockerfiles Commands (James Lee)

1. RUN - commands you want ran to BUILD IMAGE
    a. Any linux command. Same syntax as if from terminal
    b. Each RUN command builds a unique layer in the image
    c. Suggest chaining Run commands so fewer layers
        i. Put in alphanumerical order so easy to find
        ii. Chaining also ensures all updates are checked
            1. Otherwise cache is used for all but "new" commands" (when chained, entire command is modified so all pkgs (git, python, etc) updated
        iii. Example (reduces 3 RUN from above to 1 RUN)

```
RUN apt-get update && apt-get install -y \
git \
python \
Vim
```

2. CMD - instructions ran when CONTAINER is built
    a. If no COMMANDs given in Dockerfile, Docker uses default command defined in the base image
    b. COMMAND can be given in "execute form" (preferred) or shell
    c. Users can override the default CMD by specifying an argument after the image name when starting the container
    d. COMMAND ["/bin/bash"]
        i. to open shell when container is run (& user tags -it)

3. COPY - copies folder or files from build context & adds to file system of container
    a. COPY <path_to_file_from_Dockerfile> <path_in_Container>
    b. COPY temp.txt src/temp.txt
    c. NOT live-linked. Once image is made, files copied over are set. If change source code, must rebuild image to see changes. Use "volume" when build container to live-link code (see below)

4. ADD - similar to COPY but can also download source files from internet

5. USER - user whose access rights used to run image's resultant container
    a. Best practice - set to "unpriviledged" user (w/o admin rights) b/c of fear container breached and user gains access to host (OsX). Example USER sierrawells or USER admin

6. WORKDIR - directory from which Dockerfile commands run
    a. CMD, RUN COPY ADD ENTRYPOINT

1. More Docker file - Dan Wahlin

i. (simply docker image building instructions)
ii. Can name this file anything you want. Convention is simply 'Dockerfile or, if multiple docker files, use .dockerfile as extension (ie, 'node.dockerfile')
iii. Simply a text file with instructions to build an image
iv. Common Dockerfile commands
    1. "FROM" - base image (ie, node) to start from
    2. "MAINTAINER" - who maintains the image (you)
    3. "RUN" - define actions that are done as image is created. (ie, npm install)
        a. (grab file from internet, npm start, etc). Typically to install pkgs
        b. Each RUN commands creates a new image layer
    4. "COPY" - move files into container (ie, source code. "Baked" into image).
    5. "ENTRYPOINT" - command that gets exec when container is spun up
        a. (ie, 'npm start'). Makes container an executable -like an .exe file
        b. ENTRYPOINT and CMD are different but interchangeable
            i. Over-ride default CMD by specifying an argument after the image name when starting the container:
            ii. ENTRYPOINT can be similarly overridden but it requires the use of the --entrypoint flag
    6. "WORKDIR" - defines work dir. Location that commands will run from (ie, npm install, npm start). RUN, ENTRYPOINT, CMD, ADD & Copy
    7. "EXPOSE" - ie, expose a port
    8. "ENV" - set environment variables
    9. "VOLUME - define volume for container -see volume discussion above

b. Dockerfile example

i. FROM node:latest → gives latest version of node image
ii. FROM node:5.5.0 → gives you the specified version (5.5.0)
iii. ENV NODE_ENV=production
iv. ENV PORT=3000 → note no spaces between =
(or)
ENV NODE_ENV=production PORT=3000
v. MAINTAINER Doug W → just metadata (information others might want to know)
vi. COPY . /var/www → copy entire cur dir (.) into container folder /var/www ("bakes" a file layer into the image Docker builds up). Will NOT live update
vii. WORKDIR /var/www → sets work dir (where commands RUN ie, 'npm install')
viii. Volume ["/var/www", "/logs"] → Docker set sources on localhost. This way logs stick around on local host even if container is deleted (unless use rm -v)
    1. CHALLENGE: Volume lives outside of the container. Therefore when image is built, the volume does not yet exist. Docker creates the Volume when CONTAINER is made (not when image is made). When container is made (and Volume is set up), Dockerfile's "Run npm install" command has already ran. The files are in /var/www BUT Dockerfile's Volume command kind of wipes out those files (node_modules). Therefore /var/www no

longer has node_module files to share with host. Solution, delete Volume command from Dockerbuild. Can do this because we don't need to persist anything in this example (no need for Volume).

ix. RUN npm install

x. EXPOSE 3000 → intern cont port 'exposed' (can still assign via $ docker run)
-or- EXPOSE $PORT → since port is defined as environmental variable
1. Note: did not NEED to expose port since Express setup did (3000)

xi. ENTRYPOINT ["npm", "start"] → JSON array. Need double quotes
(or ENTRYPOINT npm start → works but best practice is JSON array- above

c. "Building" an image from the Dockerfile (custom image). Layers

i. $ docker build --help → shows all build commands

ii. Pretty easy

1. $ docker build -f <fileName> -t <your username>/<imageName> .
   a. -f → don't need if named Dockerfile. If not, give actual name here (ie, node.dockerfile would need -f)
   b. -t → short for --tag, name for your custom image
   c. -d → runs process in daemon mode (so console stays free)
   d. . → folder in which Dockerfile exists (. = curr dir)
   e. If don't have required images already downloaded, FROM statement in Dockerbuild will take some time since it will download all required images first
   (specified in FROM in Dockerfile)

2. Every instruction in Dockerbuild leads to an intermediate container being built & cached locally. Future builds will look here and if the intermediate container already exists, will just use it. Layers all these containers & makes image w/ name specified above (<username>/<imageName>). Can build with no cache by adding tag $ docker build --no-cache
   i. -or- $ docker build --no-cache=true

d. Create and run a container from that image

a. Remember to assign ip port!
$ docker run -d -p 8080:3000 dougwells/node

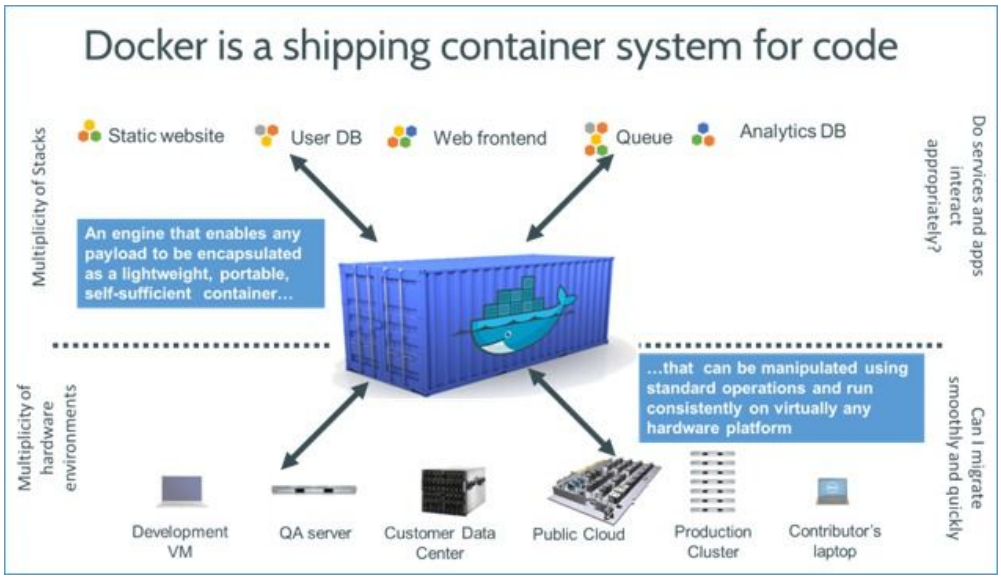
b. -d → daemon mode (keeps console free)

c. -p → ports external:internal(linux vm)
→ ports host_port:container_port
→ port on OsX:containerPort
→ localhost:8080:localhost:3000

d. dougwells/node → image name

e. Note, because we copied over files from ExpressSite on image build, any changes to files inside ExpressSite will NOT be reflected on any container ran off of that image (need to create new image

1. Containers



a. Containers are why we use Docker
b. Simply puts a thin read-write layer on top of the images (see figure below)
c. Lightweight & portable encapsulation of runtime environment
    i. environment in which to run our application
    ii. Our container, has all the binaries & dependencies needed to run our application
    iii. Isolation - docker uses functionality in Linux to achieve isolation
        1. Name Space: Information from one container not viable in others
        2. Control Groups or "C Groups": (CPU, memory, I/O, network, etc)

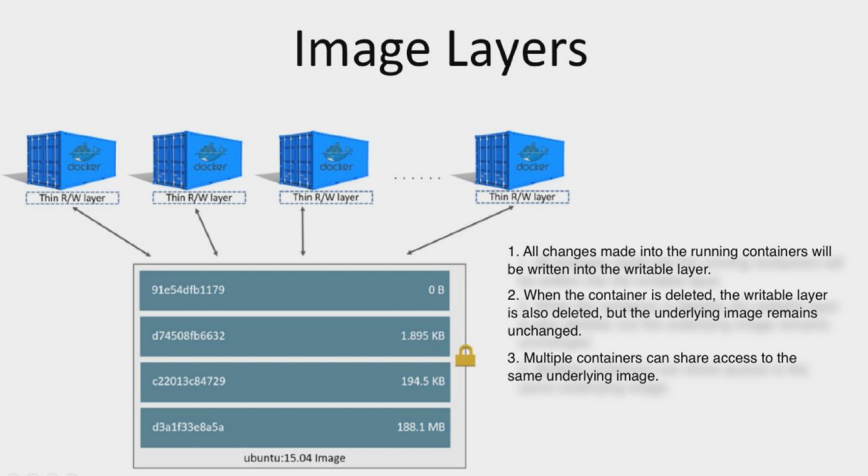
d. metaphor: Class/instance → Image/Container

i. an image is an "instance" of an image" or "a runtime object"
    1. a runtime library is a special program library used by a compiler, to implement functions built into a programming language, during the runtime (execution) of a computer program

e. Container: simply a thin read/write layer on top of image

i. Multiple containers can simultaneously share access to same image
    1. Each container will have its own data state
ii. All data is shared in each containers discrete r/w layer
iii. When delete container, r/w layer deleted but NOT the image
iv.

2. How to run a container



a. $ docker run repository/(imageName):tag any-command-to-run-when-spun-up
   i. Creates the container, spins it up and runs it
   ii. :1.24 → how you specify a version of the image
   iii. Example: $ docker run busybox:1.24 echo “hello world”
   iv. Example2: $ docker run busybox:1.24 ls /dev (list files in /dev)
   v. Above command puts you inside containers console.
      1. To free up console, run -d flag (“detached” mode)

3. Flags - Runs container w/specified parameters
   a. Interactive
      i. -i → interactive
      ii. -t → creates psuedo TTY with stdin & stdout
         a. TTY = TeleType Writer (originally for deaf people) `tele` comes from a Greek root meaning far or distant. TTY's provide a virtual interface similar to what the physical machines provided. This is the origin of the 80 char width and carriage return

      2. Example: $ docker run -t -i busybox:1.24
         a. Gives you terminal inside the container. Prompt looks like / #
         b. Exit terminal in container
            i. / # exit → exits terminal & SHUT DOWN CONTAINER

   b. Detached
      i. -d → detached mode. Frees up console
   c. Interactive mode
      i. $ docker exec -it <cont_id> command
         1. exec lets you run a command inside a running container
            a. -it → enter container in interactive mode
            b. bash → command (enter bash mode)
         2. To enter detached container once running …

3. Example → $ docker exec -it 175cfac1b48a bash
4. See Running programs → $ ps axu (just a shell command)
5. Exit interactive mode (& return to daemon mode) → ctrl-p & ctrl-q

d. "Hitting" IP address of a container

i. We linked external port to the DM's internal port when we created the container

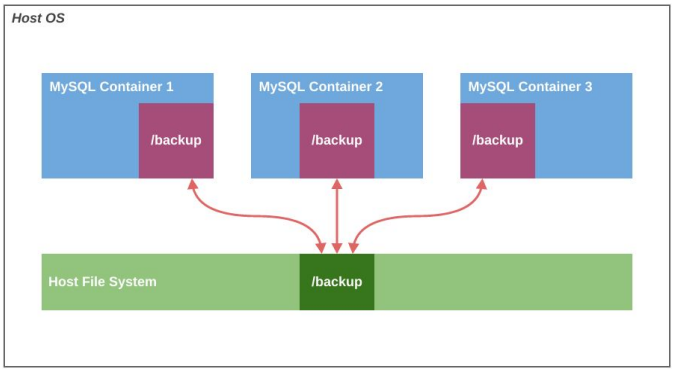
ii. docker run -p **8080**:**3000** imageName - (machine port:container port)
(DM external port:DM internal port)

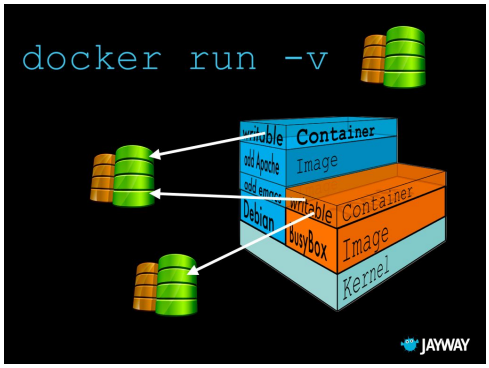
1. $ docker-machine ls to get ip of docker_host environment
2. $ docker ps to get ip of specific container
   a. dockerapp_1 | * Running on http://0.0.0.0:5000/
      i. This is the ip WITHIN container. Need to get to where container is running (which is the ip of docker_host)
   b. Therefore ip of running container is:
      i. Docker_host_ip:external port exposed by container

iii. Examples of finding docker_host IP:container_external_port

1. Node example → $ docker run -p **8080**:**3000** **node**
   a. Remember, whale icon: "docker is configured to use the default machine with IP 192.168.99.100". This is host ip
   b. Port can be 'hit' outside linux VM at this ip + port #
   c. Can hit port within Linux VM at http://localhost:3000
   d. To hit port from outside the Linux VM, we set external port to **8080** so container will export on Linux VM's ip address on port 8080 http://192.168.99.100**:8080**
2. Tomcat example → $ docker run -p 8888:8080 tomcat:8.0
   a. http://host-ip:8888
      i. $ docker-machine ls → Lists DMs & host-ip
      ii.

1. Docker Volumes - Hooking up your source code to a docker container





a. Volumes exist OUTSIDE of the docker container (on host machine)
b. Volumes can be “connected” to more than 1 container
c. See if any volumes are “mounted” (aka connected) to a specific container
    i. $ docker inspect [container ID] → ie, $ docker inspect 901
        1. Look at JSON output for “Mounts”
        2. Source = file location of source code (outside of container)
            a. Source is “source” for volume (on host of docker machine, OSX).
            b. Destination is w/in container
        3. Destination = file location w/in this container

```
"Mounts": [
    {
        "Source":
        "/Users/dougwells/Dropbox/1-Code/Docker/DockerBasics1/ExpressSite",
        "Destination": "/var/www",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
    }]
```

d. How connect container to local code (how set “Source” and “Destination” in Mounts?

i. You create a volume when you first create the container from its image
    1. In this example, we create a container ($ docker run) from image ‘node’
ii. MUST RUN THIS COMMAND FROM W/IN DESIRED LOCAL DIRECTORY
(kinda true but only because use $(pwd) … print working directory)

$ docker run -p 8080:3000 -v $(pwd):/var/www -w "/var/www" node npm start

1. node: simply the name of image you want to run ($ docker run node)
2. -v Source (external to container) **:** destination (internal to container)
3. -v → tells docker you want to connect this container to external volume
4. -w → sets working directory.
Where all commands, ie “npm start”, are run from
5. -d → not show. Runs process in “detached mode” so console stays free
6. $(pwd):/var/www
    a. $(pwd)→ print working dir (dir where you are typing command)

b. : → separates source from destination
c. /var/www → destination WITHIN container

7. npm start → command you want to run when container starts (commands run from working directory called out in -w "xxxx/xxx"

e. Can also create a volume without a source.

i. Simply name the volume (ie var/www) and docker sets it (volume) up with a docker specified source.
$ docker run -p 8080:3000 -v /var/www node npm start.

docker run -p 8080:3000 -v /var/www -w "/var/www" node npm start
(Note: no "source")

ii. Docker creates that folder (/var/www) and mounts/connects it to host machine
iii. Careful, when do $ docker rm -v [cont id], this folder (var/www) will be DELETED
iv. But, this folder will NOT be deleted if just delete container ($ docker rm [cont id])

f. High level concepts

i. External source file/"volume" is connected to machine's "Destination"
1. Note: volume is OUTSIDE of container. It is connected to the machine aka linux virtual box
2. If delete container, volume persists (b/c other containers might use it)
ii. Container looks at machine's "Destination" location for its code
iii. If Delete container, volume persists

g. Start & Stop container

i. Cool thing is once container is connected to source code, next time it is started, it (container) remains hot linked to source code
1. $ docker start [cont id] $ docker stop [container id]

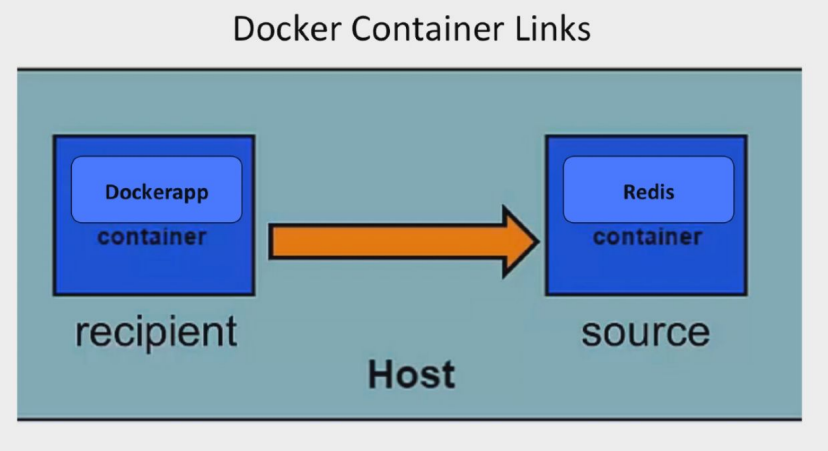
h. Remove volume

i. $ docker rm [container id] → removes container, not volume (b/c mult cont)
ii. $ docker rm **-v** [container id] → removes container AND volume
1. Does not delete local folder (source code)
a. CAUTION: it would delete Source if it set by default by Docker
2. Only worry about removing volume if source was created by docker (clutters hard drive if not needed). If you specified source on your local machine, you can clean up or delete this file as you normally do.
Reminder, only delete volume when ALL containers are done using it.

i. Managing volumes

i. $ docker volume → list all volume commands
ii. $ docker volume ls → lists all volumes

1. Docker Container Links (James Lee)



a. how to connect services that run in different containers
b. (in Docker Image section, wrote Dockerfile that had Flask (aka dockerapp) service in one container using local memory). What if want 2+ services? Use multiple containers & link
c. Recipient container can get information FROM source container
    i. Links established using --link flag AND container names
    ii. Docker creates a secure tunnel that doesn't need to expose any ports externally on the containers (note, do not need to use -p flag when spinning up source container - redis)

d. Steps to set up a link

i. Modify dockerapp's Dockerfile so that dockerapp installs not just Flask library but also redis client library (note, this is just the library. NOT the images). Redis service will be built from its own image. We need this so Dockerapp has a redis client (can talk to redis service/container).

Dockerapp Dockerfile:

```
FROM python:3.5                          → base image is python:3.5
RUN pip install Flask==0.11.1 redis 2.10.5   → installs Flask libr & redis client libr (NOT img)
RUN useradd -ms /bin/bash admin          → adds new user, admin.
USER admin                               -ms: adds default home dir for new user
WORKDIR /app                             /bin/bash: sets default shell to "bash"
COPY app /app
CMD ["python", "app.py"]
```

ii. Start redis service by downloading existing "official" redis image from Docker Hub
    1. $ docker run -d --name redis redis:3.2.0 → note named container "redis"

iii. Build dockerapp image from Dockerfile.
    1. $ docker build -t dougwells/dockerapp:v0.3.1 .

2. Note name it “tag it” dougwells/dockerapp:v0.3.1

iv. Run dockerapp container & link to redis (note: redis container is already running)
1. docker run -d -p 3000:5000 --link redis dougwells/dockerapp:v0.3.1
a. (note: secure tunnel between containers. No external ports exposed to link containers. -p flag is URL for app. No -p when spun up redis container)

v. Look up docker-machine’s IP ($ docker-machine ls). Can visit dockerapp at that IP on port 3000 (set above)

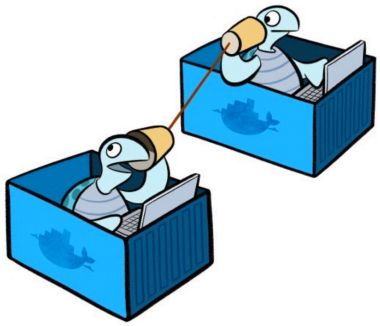
e. How does container link work?

i. To see IP of containers, log into the running recipient container (dockerapp). Mapping of host IP addresses is in /etc/hosts file
1. $ docker exec -it <cont_id> bash
#/: $ cd .. → move up to root directory
#/: ls → see all directories (want etc/hosts
#/ more etc/hosts → open hosts file inside etc directory
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3 redis 7007a28ed576 → container_id
172.17.0.2 7a7bb4e21e6a

2. Inspect redis container
a. $ docker inspect redis
"Networks": {
"bridge": {
"Gateway": "172.17.0.1",
"IPAddress": "172.17.0.3",

3. Can also verify redis ip or 172.17.03 by logging into running dockerapp container & pinging redis
a. $ docker exec -it 7a7 bash
i. /# ping redis note: 172.17.0.3
→ 64 bytes from 172.17.0.3: icmp_seq=0 ttl=64 time=0.106 ms

1. More on Linking Containers (Dan Wahlin)



a. Communicating between containers

i. High Level, 2 methods

1. Legacy Linking - via naming convention
2. Bridge Network - container w/in bridge network can talk w/each other

ii. Legacy Linking

1. 3 steps
   a. Each container to be linked must have a name

$ docker run -d --name <contName> <imageName>

Example: $ docker run -d --name my-mongodb mongo

   b. Link to running container by name

$ docker run -p 3000:3000 --link my-mongodb:mongodb danwal/node

      i. This command runs danwal/postgres
      ii. Links danwal/node to my-mongodb
      iii. <contName>:<contAlias> - gives alias ('mongodb')
         1. can use alias internally by linked containers (node)
         2. Optional to give a named container an alias

   c. Repeat for additional containers

2. Example: Linking Node.js container to MongoDB container
   a. Run mongodb from image AND NAME THE CONTAINER
      i. $ docker run -d --name my-mongodb mongo
   b. Run node from image & link to mongodb container ("my-mongodb")

$ docker run -p 3000:3000 --link my-mongodb:mongodb dougwells/node

   c. Another example

$ docker run -p 3000:5000 -d --link redis dougwells/dockerapp:v0.3

d. That's it …

3. Testing networks
   a. Ping one container from server of another container in same network
      $ docker exec -it python-app bash
      /# ping redis (name or ip address of other container)

iii. Network Linking (Creating an isolated network).

1. AKA "Container Networking with a Bridge Driver"
2. NOTE: if connecting more than 2 or 3 containers, likely better to use Docker Compose (see below)
3. Note: Docker docs refers to networking as "Communicate between containers"
4. Steps
   a. Create a bridge network
   b. Run containers w/in that network
   c. All containers within bridge network can "talk" with one another
   d. Note: containers can belong to more than one network
5. Docker commands for creating bridge network & running containers in it
   a. Create network

```
$ docker network create --driver bridge isolated_network
create network          bridge network    network name
```

   b. Run containers from within the network

```
$ docker run -d --net=isolated_network --name mongodb mongo
$ docker run -d --net=isolated_network --name nodeapp -p 3000:3000 dougwells/node
```

6. Docker commands that help with networks

a. $ docker network ls - list docker networks
b. $ docker network inspect [ID] - network info (including containers in it)
c. $ docker disconnect [cont ID} - disconnect a container from a network
d. $ docker network rm [ntwk ID] - remove entire network

iv. Benefits of linked containers

1. Microservices
2. Docker creates secure tunnel between containers
   a. No need to expose external ports to link services

2. Deploying on Digital Ocean

a. Sign up at Digital Ocean and get an API Token
    a. Note - you can also access droplet via console w/in Digital Ocean
        1. https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-16-04
            a. users: root/pw=address, doug/pw=social, dougwells/pw=social

b. Provision a VM in Digital Ocean
    a. $ docker-machine create --driver digitalocean --digitalocean-access-token <API-Token-AlphaNum> <desired-name-of-VM-on-Digital-Ocean>
    b. (Actual Code w/o Token) $ docker-machine create --driver digitalocean --digitalocean-access-token abcFakexyz docker-app-machine
        1. --driver
            a. Docker has drivers for all of the large cloud providers (DO, AWS, Azure, Linode, etc).
        2. --digitalocean
            a. Each provider has their own way of sharing API Token
        3. Name of VM (we use docker-app-machine)

c. Set up Docker Client to communicate with docker-machine/server on Digital Ocean
    a. When we finished provisioning VM on DO (above), got the following message:
        1. > *To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run*: $ docker-machine env docker-app-machine
        2. > # Run this command to configure your shell: $ eval $(docker-machine env docker-app-machine)
        3. Do exactly that. Steps 1 & 2. Run env command (sets environment var)
        4. Run $ docker-machine ls to see all VM and to make sure DO is active (*)
        5. Run $ docker info to display info on Dig Ocean VM

d. Let Digital Ocean know how to get app image (that it will run to spin up containers)
    a. Image that passed CircleCI is on Docker Hub. How point DO to Docker Hub?
        1. Copy docker-compose.yml into new file called prod.yml
            a. In prod.yml
                i. change line **build: .**
                    1. Builds new image w Dockerfile in CWD
                ii. to → image: dougwells/dockerapp
                    1. Use existing image from Docker Hub

e. Time to Deploy our dockerapp!!
    a. Make sure docker VM is active → $ docker-machine ls
        1. If not, follow steps in section C above
    b. With docker vm active
        1. $ docker-compose -f prod.yml up -d
            a. docker-compose up → regular command that builds/gets images and spins them up
            b. -f prod.yml → what file to use (otherwise defaults to docker-compose.yml)

c. -d → free up console (“detached” mode)

c. Now when run $ docker-machine ls

1. → get name of app (docker-app-machine) w/IP

a. Name we gave VM in first step above (step “b” - when we provisioned VM on DO)

f. Digital Ocean has lots of options for provisioning a VM

a. Google “docker digital ocean driver”

g. High level summary of steps

a. Create Servers

$ docker-machine create --driver digitalocean --digitalocean-access-token <API-Token-AlphaNum> <desired-name-of-VM-on-Digital-Ocean>

b. “Provision” Servers

1. Once the server has been set up, it needs to be "provisioned", meaning that the all software and configuration settings are being applied. Other tools beside Docker include Chef, Puppet and Ansible for this.

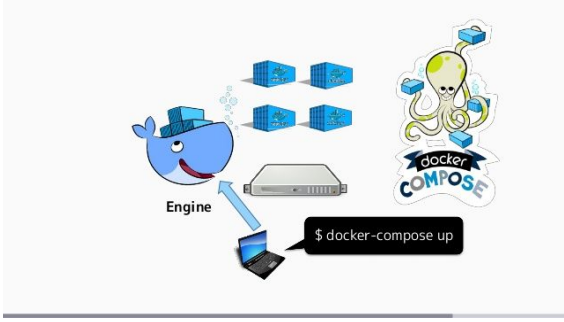
$ docker-compose -f prod.yml up -d

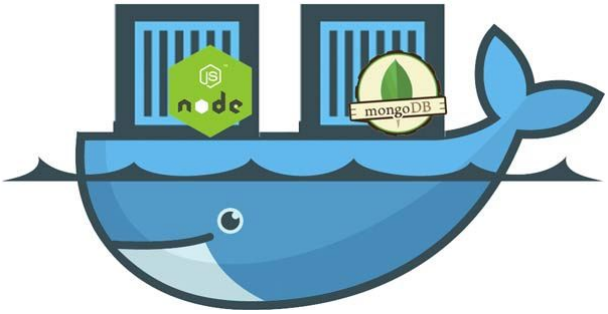
c. To take to Swarm, need tool like Consul (see below)

3. Docker Compose (Part of Docker Toolbox).

a. One Docker-Machine, One Docker-Server, multiple containers

a. Use Docker Swarm when managing multiple Docker-Machines





b. Create docker-compose.yml file to tell docker what to do when build services & bring up
c. Docker Compose automatically supports communications between services/containers using the names you give the services in yaml file (docker-compose.yml in our case)

d. docker-compose.yml → pronounced “yamel”

a. yml is similar to JSON but indentations instead of brackets. No commas
b. “Extends” keyword
    1. Enables sharing of common configurations amongst diff files & projects
    2. Useful if several different files which use common set of configurations

3. Example of using “extends” in yaml files

```
Common.yml
----
version: '2'
services:
  dockerapp:
    ports:
      - "5000:5000"

  redis:
    image: redis:3.2.0

docker-compose.yml
-----
version: '2'
services:
  dockerapp:
    extends:
```

```
      file: common.yml
      service: dockerapp                //use common.yml
  dockerapp code
    build: .                    //dockerapp's Dockerfile in cwd

  redis:
    extends:
      file: common.yml                  //use common.yml redis
  code
      service: redis
```

c. Example docker-compose files

```
-------------
Example 0:  github.com/jleetutorial/dockerapp
-------------
        version: '2'
        Services:       //either "build" image with Dockerfile or from image
          dockerapp:    //local-image-name →
        (local-image-name_repository_name)
            build: .
            ports:
              - "5000:5000"     external host(OsX):internal container
            volumes:
              - ./app:/app      //host:container (ie OsX:container)
                                //no longer need COPY command in Dockerfile

          redis:
            image: redis:3.2.0          //naming image



-------------
Example 1:  github.com/DanWahlin/NodeExpressMongoDBDockerApp
-------------
        version: '2'            // '2' supports volumes & networks (v1 does
        NOT)
        services:

          Node:
            container_name: ex1Name
            build:
              context: .                // "." - look in curr dir(of .yml) for
        dockerfile
              dockerfile: node.dockerfile
            ports:
              - "3000:3000"             //leading "-" b/c can have multiple
            networks:
              - node-app-network                //leading "-" b/c can have
        multiple

          mongodb:
            image: mongo
            networks:
```

```
      - node-app-network                //leading “-” b/c can have
multiple

networks:
  node-app-network:
    driver: bridge
```

```
        ------------
Example 2:
github.com/DanWahlin/CodeWithDanDockerServices/blob/master/docker-compose.yml
        ------------

# 1. Update config values (localhost --> mongo and localhost --> redis) in
config.development.json
    # 2. Set APP_ENV environment variable to proper value in Dockerfile-redis (default
is "development")
    #    export APP_ENV=development
    #    export DOCKER_ACCT=<yourHubUserName>
    # 3. Run docker-compose build
    # 4. Run docker-compose up
    # 5. Live long and prosper

version: "2"

services:
    nginx:                              //reverse proxy server. Balance loads node1,2 &
3
      container_name: nginx             // name container. OK: “-” & CAPS. NO: “ “ & /
      image: ${DOCKER_ACCT}/nginx
      build:
        context: .
        dockerfile: .docker/nginx.${APP_ENV}.dockerfile
      links:
        - node1:node1                   //links: important here since feeds 3 servers
        - node2:node2
        - node3:node3
      ports:
        - "80:80"
        - "443:443"
      Env_file:
        - ./.docker/env/app.${APP_ENV}.env
      networks:
        - codewithdan-network

    node1:
      container_name: node-codewithdan-1
      image: ${DOCKER_ACCT}/node-codewithdan
      build:
        context: .
        dockerfile: .docker/node-codewithdan.${APP_ENV}.dockerfile
      ports:
      - "8080"
      volumes:
        - .:/var/www/codewithdan
      working_dir: /var/www/codewithdan
      env_file:
        - ./.docker/env/app.${APP_ENV}.env
      networks:
        - codewithdan-network

    node2:
      container_name: node-codewithdan-2
      image: ${DOCKER_ACCT}/node-codewithdan
```

```
    build:
      context: .
      dockerfile: .docker/node-codewithdan.${APP_ENV}.dockerfile
    ports:
    - "8080"
    volumes:
      - .:/var/www/codewithdan
    working_dir: /var/www/codewithdan
    env_file:
      - ./.docker/env/app.${APP_ENV}.env
    networks:
      - codewithdan-network

  node3:
    container_name: node-codewithdan-3
    image: ${DOCKER_ACCT}/node-codewithdan
    build:
      context: .
      dockerfile: .docker/node-codewithdan.${APP_ENV}.dockerfile
    ports:
    - "8080"
    volumes:
      - .:/var/www/codewithdan
    working_dir: /var/www/codewithdan
    env_file:
      - ./.docker/env/app.${APP_ENV}.env
    networks:
      - codewithdan-network

  mongo:
    container_name: mongo
    image: ${DOCKER_ACCT}/mongo
    build:
      context: .
      dockerfile: .docker/mongo.dockerfile
    ports:
    - "27017:27017"
    env_file:
      - ./.docker/env/mongo.${APP_ENV}.env
    networks:
      - codewithdan-network

  redis:
    container_name: redis
    image: ${DOCKER_ACCT}/redis
    build:
      context: .
      dockerfile: .docker/redis.${APP_ENV}.dockerfile
    ports:
      - "6379"
    networks:
      - codewithdan-network

networks:
  codewithdan-network:
    driver: bridge
```

4. Docker Cloud (www.cloud.docker.com)

a. Link to cloud provider (AWS, Google Cloud, Azure, etc)

b. Fire up a node (linux virtual machine). IP address is what you "hit" to see webpage
c. Create a Stack
    a. Similar to Docker Compose & .yml file above
    b. Networks not supported yet so you "link" containers
d. To hit IP for your stack
    a. Go to "Nodes" tab (in Docker Cloud header)
    b. IP address is what you go to on internet to see your web app

```
mongo:
  image: '<yourHubAccount>/mongo:latest'
  environment:
    - MONGODB_DBNAME=codeWithDan
    - MONGODB_PASSWORD=password
  ports:
    - '27017:27017'
nginx:
  image: '<yourHubAccount>/nginx:latest'
  links:
    - node1
    - node2
  ports:
    - '80:80'
    - '443:443'
node1:
  image: '<yourHubAccount>/node-codewithdan:latest'
  environment:
    - NODE_ENV=production
  links:
    - mongo
    - redis
  ports:
    - '8080'
  working_dir: /var/www/codewithdan
node2:
  image: '<yourHubAccount>/node-codewithdan:latest'
  environment:
    - NODE_ENV=production
  links:
    - mongo
    - redis
  ports:
    - '8080'
  working_dir: /var/www/codewithdan
redis:
  image: '<yourHubAccount>/redis:latest'
```

```
    ports:
      - '6379'
```

5. CircleCI: Testing on Docker (Udemy. James Lee Docker Hub Course. Section 4)

a. Continuous Integration

b. Pulls files from your github repo
   a. circle.yml → tells circle what steps to take
c. Uses docker compose to build images & spin up containers
   a. Caution, resultant image names will have specific format
      1. When using docker-compose, image naming format is <folderName><serviceName (given in docker-compose.yml)>

Image name = <gh repo name (with dashes [-]removed) >_<service name given in docker-compose.yml

d. Once tests pass, can push passing image to DockerHub.com

```
github.com/dougwells/docker-web-app
273ada2377b5750622e8f23df5506d236115f9fcf40c2434abab73f034c2ba5c
File: docker-compose.yml
-----
version: '2'
services:
  Dockerapp:
    build: .
    ports:
      - "5000:5000"
    Volumes:            //Delete Volumes if deploy to cloud service
      - ./app:/app      //Cannot guarantee access to Volumes
                        //Rather-COPY in Dockerfile. Source files baked in
image
  redis:
    image: redis:3.2.0


File: circle.yml
--------
machine:
  pre:
    - curl -sSL
https://s3.amazonaws.com/circle-downloads/install-circleci-docker.sh | bash -s
-- 1.10.0
  services:
    - docker

dependencies:
  pre:
    - sudo pip install docker-compose

test:
  override:
    - docker-compose up -d
```

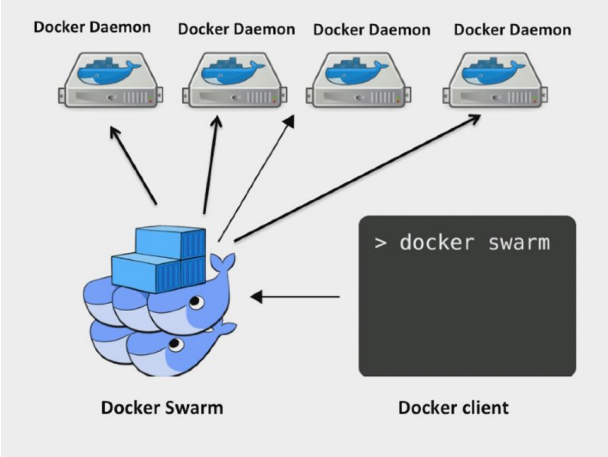
```
    - docker-compose run dockerapp python test.py

deployment:
  hub:
    branch: [circle_ci_publish, master]
    commands:
      - docker login -e $DOCKER_HUB_EMAIL -u $DOCKER_HUB_USER_ID -p
$DOCKER_HUB_PWD
      - docker tag dockerwebapp_dockerapp
$DOCKER_HUB_USER_ID/dockerapp:$CIRCLE_SHA1
      - docker tag dockerwebapp_dockerapp $DOCKER_HUB_USER_ID/dockerapp:latest
      - docker push $DOCKER_HUB_USER_ID/dockerapp:$CIRCLE_SHA1
      - docker push $DOCKER_HUB_USER_ID/dockerapp:latest
```

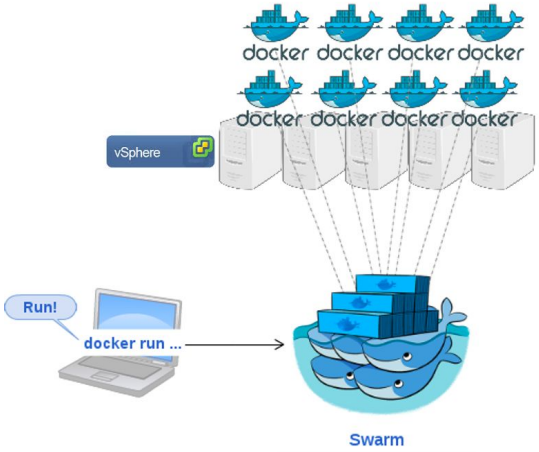
6. Docker Swarm

a. Docker Swarm is for deploying multiple Docker-Machines (Linux VMs).

a. Swarm helps manage this complexity

b. Developer connects docker-client to Swarm (group of whales) instead of one whale (aka a docker-machine)
c. See Udemy course by James Lee.
    1. See figure below
    2. whale = docker-machine, docker daemon = spun up container





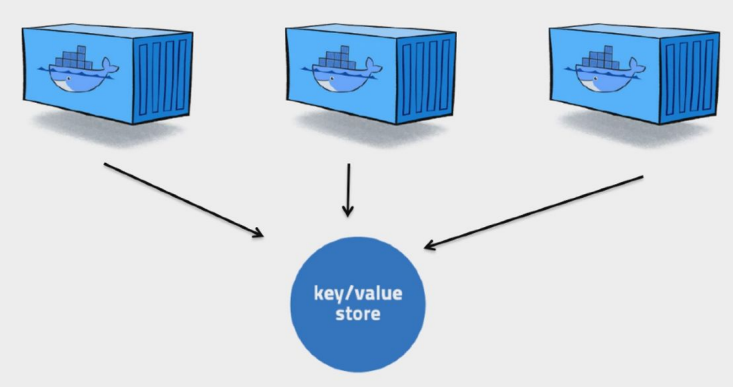
b. Steps to setting up Docker Swarm

a. Create docker-machine in cloud (ie, Digital Ocean) to be key/value store
b. Create docker-machine in cloud to be Swarm Master. Only ONE
c. Create Slaves (simply docker-machines). Multiple OK
d. All swarm nodes communicate on private network, eth1 (same data center)
e. All Docker-Machines know about one another b/c of key/value store (consul)
f. Hook up Docker Client to the Swarm to connect to it and control the swarm

c. First, Must set up key/value store (Consul or ZooKeeper)

a. Key/Value Store gives Docker Swarm information on how each node (docker server is running).
b. Also called Service Discovery

1. how Swarm Manager keeps track of the state of all nodes in the cluster
    a. One way to track state is with a “key/value store”
        i. Swarm supports Consul and Apache ZooKeeper



ii. More info on Hashicor’s Consul (https://www.consul.io/)

1. Monitors how each server is running
    a. Once all the servers have been set up and provisioned, they need to know about each other, and they need to be monitored. A tool that is well suited to do this in cloud environments is Consul.
        i. It runs as a sidecar next to each service and monitors the service and the host it is running on.
        ii. Using Consul, the load balancer (nginx) will be aware of the different web servers it can forward requests to.
        iii. https://ahus1.github.io/saltconsul-examples/tutorial.html

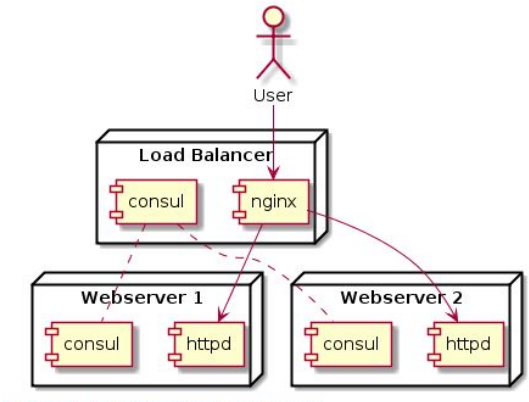


Figure 1. Infrastructure Big Picture

c. Make our lives easier by first setting ENV variables from w/in terminal
    1. If new terminal session, likely have to redo this step
        a. To see ENV, run $ printenv VARIABLE_NAME
        b. To see all ENVs, run $ env
    2. $ export DIGITALOCEAN_ACCESS_TOKEN=<YOUR_DO_TOKEN>
    3. $ export DIGITALOCEAN_PRIVATE_NETWORKING=true

4. $ export DIGITALOCEAN_IMAGE=debian-8-x64

d. 3 Steps to Set up Service Discovery / Consul

1. Create a host for running key/value store, Consul

a. $ docker-machine create -d digitalocean consul
   i. -d = --driver = provider (do, AWS, virtualbox)
   ii. consul - name we give this docker-machine
   iii. This machine is NOT part of the Swarm
      1. Rather, consul service is used by Swarm to bootstrap and communicate shared state of cluster across nodes
      2. Before a Consul cluster can begin to service requests, a server node must be elected leader (in image below, this is the consul node in the load balancer). Thus, the first nodes that are started are generally the server nodes. Bootstrapping is the process of joining these initial server nodes into a cluster.

b. Look at the network configuration of consul server
   i. $ docker-machine ssh consul ifconfig
      1. $ docker-machine ssh → run command on a docker-machine that uses ssh
         a. Docker0 - bridge network for container
         b. eth0 - public network interface. Allows inbound/outbound to entire internet
         c. eth1 - private network interface. Keeps our information private. NOT accessible over internet. Only hosts in the same data center (digital ocean in this case) can communicate.
         d. We use eth1 since private interface. If use eth0, expose key/value store to entire internet

c. Pinging eth0 (public) & eth1(private) from w/in consul docker-machine
   i. `$(docker-machine ssh consul 'ifconfig eth0 | grep "inet addr:" | cut -d: -f2 | cut -d" " -f1')`
      1. Ping results in 2 way packet traffic
   ii. `$(docker-machine ssh consul 'ifconfig eth1 | grep "inet addr:" | cut -d: -f2 | cut -d" " -f1')`
      1. Ping results in NO return packet
   iii. Use -c flag to limit # of pings
      1. `$ ping 0 -c 2  //0=localhost`
         a. Returns stats

b. TTL = # of routers packet passed through. Higher # = fewer routers

iv.

d. Use ehth1 from step above for consul's private network
   i. Set eth1's ip to an env variable so don't have to retype it
   ii. Code below, simply returns an ip address. Example 10.132.70.113
      1. $ export KV_IP=$(docker-machine ssh consul 'ifconfig eth1 | grep "inet addr:" | cut -d: -f2 | cut -d" " -f1')

2. Connect docker client to docker server consul

a. $ eval $(docker-machine env consul)

3. From the docker-machine named "consul", Create consul container from image on DockerHub.

a. Creates consul's key/value store/service

$ docker run -d -p ${KV_IP}:8500:8500 -h consul --restart always gliderlabs/consul-server -bootstrap

1. -p ${KV_IP}:8500:8500 → eth1 private network
   a. -p takes an optional parameter that specifies network interface that container's external port should be mapped to. KV_IP is env var for priv network (in our case, it is 0.132.70.113)
2. --restart always → restart if container crashes
3. gliderlabs/consul-server → Docker Hub image
4. -bootstrap
   a. Before a Consul cluster can begin to service requests, a server node must be elected leader (in image below, this is the consul node in the load balancer). Thus, the first nodes that are started are generally the server nodes. Bootstrapping is the process of joining these initial server nodes into a cluster.

   b.

d. Ready to setup Docker Swarm now that we have a key/value store (Consul)

a. Swarm "Cluster" consists of:
   1. Swarm Master - The "master node" that acts as the Swarm manager
      a. Responsible for the entire cluster and manages all the nodes
   2. An arbitrary number of "ordinary nodes"
   3. Good article at

https://blog.nimbleci.com/2016/08/17/how-to-set-up-and-deploy-to-a-1000-node-docker-swarm/

b. Create Docker Swarm "Master"

1. Only ONE SWARM MASTER

```
$ docker-machine create -d digitalocean --swarm \
--swarm-master \
--swarm-discovery="consul://${KV_IP}:8500" \
--engine-opt="cluster-store=consul://${KV_IP}:8500" \
--engine-opt="cluster-advertise=eth1:2376" \
Master

>>> Output >>>
Configuring swarm...
Checking connection to Docker...
Docker is up and running!          → Tells us Docker Swarm is running
```

2. Just a simple "docker-machine create" command w/lots of flags
   a. -d → sets driver (digitalocean aws virtualbox/local)
   b. --swarm
   c. --swarm-master
      → identifies this node (docker-machine) as Swarm Master
   d. --swarm-discovery → sets discovery (aka key/value store)
      i. Gives instances of this Swarm the ability to find & communicate w/each other
   e. --eng-opt → allows us to set the docker daemon (server) flags for this created docker-machine (called "master")
      i. --cluster-store flag
         → what key/value store to use for cluster coordination
      ii. --cluster-advertise flag
         1. Address master "advertises" to cluster as connectable. Tells Consul to do this …

2. If run $ docker-machine ls, see all digital ocean docker-machines (nodes) are on port 2376

f. master → name of this newly created docker-machine

c. Creating Docker Swarm "Slaves"

1. Can have as many slaves as you want
2. Command to create slave node is very similar to master node
   a. Once again, simply creating a docker-machine
   b. (but no --swarm-master)

```
$ docker-machine create \
-d digitalocean \
--swarm \
--swarm-discovery="consul://${KV_IP}:8500" \
--engine-opt="cluster-store=consul://${KV_IP}:8500" \
--engine-opt="cluster-advertise=eth1:2376" \
slave

>>> Output >>>
(slave) Creating SSH key...
(slave) Creating Digital Ocean droplet...
(slave) Waiting for IP address to be assigned to the Droplet...
Docker is up and running!
```

d. Connect docker client to the swarm // note "-" in -swarm

1. $ eval $(docker-machine env -swarm master)
2. $ docker-machine ls // note: "* swarm" instead of just "*"
3. $ docker info
   a. 3 containers but only 2 nodes. Why?
      i. $ docker-machine ps -a //to find out more
         1. slave/swarm-agent running on slave host
         2. master/swarm-agent running on master host
         3. master/swarm-agent-master on master host
            a. Master host is a slave node as well as a master node
               i. As Master → responsible for which host to run containers
               ii. As Slave → can run containers on itself
   b. Also note "strategy" is spread
      i. Master runs containers in nodes with least load

e. Changes to dockerapp prod.yml file to run app container in Swarm
   1. Add networks section to prod.yml
      a. b/c in default docker, multiple containers can only communicate intra-host (all containers in same host/docker-machine). Adding networks: to yaml file allows inter-host communications.

   2. Add Environment constraint so that dockerapp always runs on master node (otherwise Swarm can assign nodes based on load)

```
version: '2'
services:
  dockerapp:
    extends:
      file: common.yml
      service: dockerapp
    image: jleetutorial/dockerapp
    Environment:                //ensures dockerapp always runs on
master node
      - constraint:node==master
    Depends_on:         // ensures redis container is up & running 1st
      - redis           // b/c dockerapp needs redis
    networks:
      - mynet           // allows inter-host container communications

  redis:
    extends:
      file: common.yml
      service: redis
    Networks:           // allows inter-host container communications
      - mynet

Networks:               // overlay supports multiple host networking
  Mynet:                        // natively (works "out of the box")
    driver: overlay
```

3. Spin up dockerapp & redis containers

   a. Make sure docker client is connected to the Swarm
      i. $ eval $(docker-machine env -swarm master)
   b. (from dockerapp directory) $ docker-compose -f prod.yml up -d
   c. Spins up the dockerapp & redis containers
      i. $ docker ps
         1. see running containers
            a. Also show's docker-app's ip address
            b. 104.131.52.1:5000
            c. dockerapp runs on master
            d. "Master" runs as master & slave node

```
CONTAINER ID    IMAGE               COMMAND                  CREATED          STATUS          PORTS                          NAMES
b1fd2f59e7b9    dougwells/dockerapp "python app.py"          14 hours ago     Up 14 hours     104.131.52.1:5000->5000/tcp    master/dockerapp_dockerapp_1
aa213126c684    redis:3.2.0         "docker-entrypoint.sh"   14 hours ago     Up 14 hours     6379/tcp                       slave/dockerapp_redis_1
94afbfc1ed91    swarm:latest        "/swarm join --advert"   15 hours ago     Up 15 hours     2375/tcp                       slave/swarm-agent
```

```
ad18199d27ab  swarm:latest  "/swarm join --advert"  18 hours ago  Up 18 hours  2375/tcp                                  master/swarm-agent
3927e6c7ed0e  swarm:latest  "/swarm manage --tlsv"  18 hours ago  Up 18 hours  2375/tcp, 104.131.52.1:3376->3376/tcp  master/swarm-agent-master
```

ii. $ docker-machine ls

1. See VM's & network ip's they communicate on
    a. Note all ip ports are :2376 which we set

```
NAME                ACTIVE      DRIVER       STATE     URL                         SWARM
consul              -           digitalocean Running   tcp://45.55.201.124:2376
default             -           virtualbox   Running   tcp://192.168.99.100:2376
docker-app-machine  -           digitalocean Running   tcp://45.55.85.230:2376
master              * (swarm)   digitalocean Running   tcp://104.131.52.1:2376     master (master)
slave               -           digitalocean Running   tcp://104.131.163.64:2376   master
```

7. Courses

a. Dan Wahlin

a. → https://github.com/DanWahlin/NodeExpressMongoDBDockerApp

b. James Lee

a. → https://github.com/jleetutorial/dockerapp