Docker Notes
including Compose, Swarm, & Stack

Notes open for creative commons use by Kyle Miskell at https://unfoldkyle.com and kmiskell@protonmail.com

Learning Resources
Docker Official Docs: https://docs.docker.com/
Docker Master Udemy Course: https://www.udemy.com/course/docker-mastery/

Version
-These notes are from 08/2020 using version 19.03.12

Intro to Docker

Why Docker
-Allows uniform standardization of test, dev, prod, etc. environments and across machines, with quick setup, as opposed to complicated, version dependent, command driven, hard to easily replicate nightmare

-Containers run through Docker engine, which runs on top of OS layer, each container managing their own dependencies and libraries, with cross-OS compatibility of containers

-Different parts of stack in diff containers (ex. Node, mogoDB, Redis) can interact together. Containers each have own running processes, network interface, mount, etc..

Containers vs Virtual Machines
-Virtual machine contains it all, the OS, the libraries, the dependencies, everything, and is managed on top of the hardware via the hypervisor, given an user interface on the software level on the OS.

-Docker runs *on top* of the OS, and then using its engine, manages the containers, which also run on the software layer, as simple processes on the native OS. For example, start nginx via Docker, then run linux command *ps -ef | grep nginx* and you will see nginx running native on the host, just as a standard, non-Docker Nginx process would. It uses what exists on the existing OS and adds what the container needs to function, but in a segregated environment, communicating with the native OS through the docker API.

-Linux' namespaces allow you to create environments that isolate everything; process table, filesystem, networking, etc. Docker doesn't facilitate communication, it tells linux to connect host environment's port x to guest environment's port y. Docker's whole job is essentially telling linux to create environments in a certain way and keeping track of image and container files.

-By interacting with the OS layer, instead of the hardware layer Docker has less processing and memory overhead that VMs. It also uses less disc space (MB vs GB).

-Docker daemon (*dockerd*) manages containers and communicates with host via CLI (*docker*), which uses a REST API. Docker is client-server, with CLI as client and docker daemon communicating with

containers, images, and docker registry as server.

-Docker daemon can listen for unix, tcp, and fd requests, can interact with local storage drives, and can isolate in unique user namespaces.

-The Docker registry stores Docker images and acts as a public repo

-On the downside, Docker is less isolated than VMs, as containers share resources, vs stand alone planet of VM. VM also allows you to run full GUI OS on top of OS.

-For huge hostings, often see servers containing many VMs, with diff OS installs, each running Docker, managing containers

-On Linux and Mac, Docker creates a linux VM

Images
-A read-only image, where a container is an instance of an image (as object is to class)

-Docker Hub is a pub registry (repo) that contains many images that can be user uploaded, including many standard official containers for stack, such as postgres, ubuntu, redis, nginx, etc.

-Custom images often created using existing image as base

Containers
-Runnable instance of image. Containers can be created, started, stopped, moved, and deleted. Multi containers of same image can run or exist on host at same time.

-Run container on Docker via: *docker container run [imageName]*

-Can connect to networks and attach storage to containers

Services
-Allows containers to work across multiple Docker daemons as a "swarm." Each daemon is a swarm member and communicates via the Docker API.

Basic Container Creation & Use

- *docker version* – shows current local version as well as latest Docker Hub server version.

- *docker container pull [imageName]* – downloads image to local host, with Docker Hub as default pull registry (repo)

-Do not need to ref containers by full ID, and instead can do so by unique start of ID.
-Ex. If containers, 00928301JD and 0092AAFJD exist, could ref by 0092A or 00928, as long as unique ID sections across all containers

Running Containers

- *docker container run [imageName]* – starts container from local image instance or pulls from Docker Hub (or specified repo), then runs, if image not local
  -ex. *docker container run nginx*

-option: *-d* - running container without options will result in container output being showed in terminal. If want to run silently (detached, without occupying terminal), run with *-d*

-If run detached with *-d*, then later want to attach, with container output shown in host terminal, run *docker container attach [containerName or containerID]* to attach to terminal

-If want to give container specific name, instead of auto-gen one, add option --*name [someName]*

-Can pass in env variables for container via *-e [envVariable]* option.
  -Ex. *-docker container run mysql --env MYSQL_RANDOM_ROOT_PASSWORD=yes*

-Run specific version: *docker container run [imageName]:version* – runs (and pulls, if needed) specified ver
  -ex. *docker container run redis:4.0*

  -*:version* is a "tag" given to the image name

-To see all versions/tags available for local install, check Docker Hub under "Tags" for image

-Can also see details on local via *docker inspect [imageName or containerName]*

-For effective cleanup, run containers you don't need later with *--rm* option, to remove as soon as stopped

Stopping and Pausing Containers

- *docker container stop [containerName or containerID]* – stops running container

- *docker container kill [containerName or containerID]* – force stops running container

- *docker container pause [containerID/name]* – pauses the container. Resume with *unpause*.

Removing Containers & Images

- *docker container rm [containerName or containerID]* – each time run image, new container with new ID and name started, but on *stop*, old container file still exists. *rm* to remove these old, essentially un-needed temp files.
 -On debian, containers stored in /var/lib/docker/containers
 -Can also restart created containers, which will maintain state at *stop* time
  -Can remove multi by specifying IDs separated with spaces

- *docker image rmi [imageName]* – removes docker image from local. No instances of must be running and no old containers must be referencing image (must delete all old containers created from). This also removes run history from l*s -a*
  -Can force quit all running containers, then remove image with option *-p*

Managing Containers

- *docker image ls* – list all currently local docker images

- *docker container ls* – list all currently running containers
  -run with *-a* option to see stopped containers for local images, as well as running

- *docker container top [containerID/name]* – shows top processes running within that container

- *docker container stats [containerID/name]* – shows stats of containers resource on host machine, such as how much mem, cpu, etc. Docker engine is using for container from host

- *docker container inspect [containerID/name]* – shows config for container

- *docker container logs [containerID/name]* – will show standard logs for specified container

Container Lifetimes

-A container runs as long as it has active processes running for it. For example, running Ubuntu will cause it to close as soon as it starts. That is because as soon as Ubuntu boots, it's processing is done. In order to make it run longer, it would have to be run with a command, like *sleep 5*

-Containers start and stop as needed, and if no processes are running for a container, it will stop.

Running Containers Interactively

-These commands replace the standard SSH interaction like you would normally use when logging into a web server and setting up it's config files, etc.

-Interactive mode – *docker container run -it [imageName]* runs container in interactive terminal mode, allowing container to accept input from host terminal while running.

-Can run container with starting command via *docker container run [imageName] [someCommand]*

-Common to run *bash* command with container, which gives shell access to container, to interact with files in container, run commands on it, etc..
-ex. *docker container run -it nginx bash*, then from terminal, *sudo nano /etc/nginx/conf.d/default* to config web server

-As docker containers have persistent state on change, if start up again after pausing or exiting, state changes will persist
-To start exited container interactively again: *docker start -ai [containerName/ID]*

-Can run commands in already running container via *docker container exec [options] [name/ID] [cmd]*
-ex. start container without interactive terminal and bash with *docker container start my_container*, but then want to interact with, so run *docker container exec -it my_container bash*

-When exit running container started with *bash*, etc., container also stops. When exit from terminal started via *exec*, however, container keeps running in BG.

Docker Networking

Basic Concepts

-Review: A server machine provides a variety of ports, each for different services hosted by the machine. For example a web server is usually accessed on the host via port 80, while FTP is via port 21. A server machine can host a variety of services simultaneously, accessed via different ports.

The Public IP of a web server is the unique ID to access it (typically via a domain name). Via the request-response procedure, the server listens for client requests over it's specified port (ex. 80), then returns the requested files, data, etc. to the requesting client, also via that port.

-Bridge – creates a single aggregate network from multiple communication networks or network segments

-For containers requiring outside port access, Docker gives containers a virtual IP (each container gets it's own IP) on a virtual private network created by the docker engine, that runs on the OS. Docker then communicates with host network from container port X, through NAT firewall, and out of host network through host port Y. Virtual network is bridge network between docker container and host network.

-Essentially, since docker is on its own network, need "bridge" network to connect ports on docker container to host ports, for listening, etc.

-Set up port mapping via: *docker run -p [hostPort]:[dockerPort] [imageName]*

-ex. *docker run -p 8080:80 httpd*

-By default, each new docker container is addded to the existing virtual network. Containers on the same virtual network can communicate with each other without port mapping. Multiple virtual networks can also exist and run simultaneously on the host. Best practice is each app gets it's own virtual network.

-A clear benefit of this is easy install of multiple web servers, etc. on the same host, each on their own network, all being set to serve web via port 80, etc.

-Containers can talk to 0 to many networks. Can also use host network, with no virtual network.

-To see current port mapping for container: *docker container port [containerName/ID]*

-Docker *inspect* cmd gives container virtual IP. Search for it in output with:
docker container inspect --format '{{ .NetworkSettings.IPAddress }}' [containerName/ID]

-Security benefit as not all containers on virtual network need to have port mapping set. Ex. Could put nginx and mysql on virtual network, then port map nginx to host port 80, but give mysql no port mapping. Nginx can now talk to mysql, but mysql cannot be accessed via any host port.

-Containers can only map to unique ports. Ex. Nginx could map to host port 80, and Apache port 81, but not both host port 80 (muh networking basics).

Network Commands
-Show Docker networks: *docker network ls*

-View network details: *docker network inspect [networkName/ID]*
  -Will show all containers connected to it and their IP addresses

-Create network: *docker network create [networkName]*
  -option --driver to create using specific available network drivers. Default is *bridge* driver.

-Connect container to network: *docker network connect [networkName/ID] [containerName/ID]*
  -connecting to a new network does not remove existing network connections

-Disconnect container from network: *docker network disconnect [netName/ID] [containerName/ID]*

-Prune unused networks: *docker network prune*

-Remove network: *docker network rm [netName/ID]*

-Can add option *--network [netName]* during container creation to attach to network at creation

DNS
-Docker IPs very dynamic, thus using IP as a unique reference ID for network comms of containers talking to each other not good. Instead, docker uses container names as unique domain names.

CLI APP Testing
-Docker useful for testing apps via CLI, as can immediately set up via images, then run tests on through interactive terminal mode

-Ex. could run an Ubuntu container and CentOS container, then check the curl version on both

DNS Round Robins
-Multi hosts, with multi IPs responding to same DNS name. Useful for load balancing. Docker can also have multi containers respond to the same DNS name.

-Can do this by giving containers additional DNS names, same name for multi containers with *run* option *--network-alias=[someName]*

-To do this with new containers, create a new network, then containers, all on that network, all with the same network alias. To test, can run *nslookup [aliasName]* from a linux container on the same network and should return virtual IPs of all containers using that alias name.

Cleanup
-*docker image prune* – removes all dangling images

-*docker system prune* – removes all stopped containers, all networks without containers, all dangling images, and all dangling build cache

Images in Depth

What is an Image?
-Essentially, the binaries and dependencies needed for the process, in a segregated namespace, as well as metadata about it, and how to run it

-Unlike a virtual machine, which will contain the kernel, kernel modules, etc., these already exist on the host OS Docker is running on, and are thus not part of images. The only thing the image has is the binaries and dependencies needed for the process to run on the existing OS kernel, using existing kernel modules, etc..

Docker Hub
-Docker Hub is a registry (repo) for hosting images. If downloading a common image, use an official image. Official images are tested, well documented, etc., coming from the official software team.

-For official images, good docs, so check ‘em.

-See “tags” under images for latest and available older versions.

-Pull specific version: *docker container pull [imageName]:[tag]*
  -ex. *docker container run redis:4.0*

-Best practice for production, use specific version, and not just “latest,” which will change over time. For example, nginx 1.19.0 might be the latest during app creation, so you create the container with that, but even when 1.19.1 comes out, you stick with 1.19.0 on prod.

-Common to download official image, do a bunch of mods on it, then re-upload it as an altered version.

-After create account on Docker hub, login using DH user/pass on local Docker via: *docker login*
- *docker logout*

Image Layers
-Images are made of layers, where each layer has its own SHA. As the image is built, on each change, another layer is added.

-Ex. Apt is added to Deb on one layer, then env vars set on another.

-View image layer history: *docker history [imageName]*

-Since layers have unique SHA, and layers stored locally in cache, multiple images can reference same layer. For example, Ubuntu and Debian images might use same apt layer. This saves space and speeds things up, requiring less layers to exist, be pulled, etc..

-Container states are layers on top of the image. For example, if Apache is the image and there are two apache containers, both have separate layers on top of the read-only image for write data.

-Additions of layers similar to git commits

-For metadata details and configs of image: *docker inspect [imageName]*

Image Tags
-Docker images contain three parts: *[organization]/[repo]:tag*

-When pulling official, don't need to pull using org name, but when using non-offical will pull with both. ex. *docker image pull tomjones/nginxmini*

-A tag is often a version, but really its just a label referencing a specific docker image commit. For example, for myApp v1.2, could have tags *myApp:1.2* and *myApp:1.2.0*, which both refer to the same image commit, at v1.2. This is why you can have tags like *latest* and *mainline*.

-Add tag to image: *docker image tag [imageName] [tag]*

-Giving image a new tag creates a new image with the repo name the same as that tag. This can then be pushed to Docker Hub, etc., as a new image.

-After logged in to DH acct on host via *docker login*, can push image to DH via: *docker image push [tag]*

-If want to push private repo to DH, create repo on DH first, set as private, then push to from host

Dockerfiles

Dockerfiles Intro
-Similar to a build script, that contains details on how to build the Docker image. Each command, etc. in the Docker file adds another layer to the image as it builds it. Can build in part using existing code (ex. some app in dev), that becomes part of the file.

-Common to chain commands in Docker file via &&, to put related commands on the same layer

-Contains details such as start command, log setup, ports to expose, etc.

-Basic statement format:
 # Comment
INSTRUCTION [arguments]

-First instruction in Dockerfile must be *FROM*

-Contents example:
-*FROM* creates a layer from the ubuntu:18.04 Docker image.
-*COPY* adds files from your Docker client's current directory.
-*RUN* builds your application with make.
-*CMD* specifies what command to run within the container.

-To build from Dockerfile: *docker image build -t [tag] [dirToPlaceImage]*
-This only works if Dockerfile is named *Dockerfile*. If named something else, need to add option *-f /path/to/someDockerfile*.

-Build steps cached, so after initial build, later builds are fast, with rebuild only occurring from changed lines down. Thus should put least changing lines in Dockerfile first, and most changing at bottom of Dockerfile.

-Typically build app, then Dockerize it by setting up Dockerfile, then build Dockerfile, then check that app via Docker.
-Ex. If app is a Node app, base image would also be Node, and *RUNs* would npm install, etc., with *COPY*, copying over code files, etc.

Env Variables
-A variable whose values is set outside the program, often through a microservice or by the OS
-In Dockerfile, first declare variable + value, then can ref value by name later in file

-Declare: *ENV [VAR_NAME] [value]*
ENV [VAR_NAME]=[value]

-Reference variable: *${[VAR_NAME]} or $VAR_NAME*

-Variable ref modifier: if *VAR_NAME* is set, use that, else use *value*: *${VAR_NAME:-value}*
-Variable ref modifier: if *VAR_NAME* is set, use *value*, else use empty string: *${VAR_NAME:-value}*

-ex. *FROM busybox*
ENV foo /root/is/here/bar
WORKDIR ${foo} #WORKDIR set to /root/is/here/bar

-Can set env var to ref another env var with: *ENV abc hello*
ENV ghi $abc #ghi value is hello

.dockerignore
-Same as .gitignore. Looks for it in root dir, and files and dirs listed in it are excluded from build *COPY*, etc.

-Can set to match like regex using Go's filepath.match syntax
-Can include exceptions to exclusion statements using this with *!*
-ex. **.md #ignore all md files*
!README.md #except this one

FROM
-Initials a build layer and sets the base image. Must be first instruction in Dockerfile.

-*FROM [ImageName]*

-Image can have tag. Also can be follow by *AS [alias]*

RUN

-Executes all specified commands in new layer, then commits the result.

-*RUN [command]*

-Each *RUN* gets it's own layer so common to chain multi commands across multi lines by using *&&* and putting \ on each line end in this chain

-*RUN* instructions are cached, so if the same *RUN* is called in a later build, cache, speed, etc.

CMD
-Main purpose is to provide default commands for when container runs. Can only be one per Dockerfile, with only last one taking effect. If command passed in during run, then this will be run instead of *CMD* specified commands.

-*CMD [[executable], [param1], [param2], …*
-ex. *CMD ["/usr/bin/wc", "--help"]*

LABEL
-Adds metadata to image

-*LABEL key="value"*
-ex. *LABEL version="1.0"*

-Can have multi key/value pairs on single *LABEL* line, separated by spaces

EXPOSE
-Informs Docker that the container listens on the specified network ports at runtime

-Note that this does not actually publish the port at runtime, but more so is documentation for which port should be exposed, as specified by *-p*

-Can use TCP or UDP, and if don't specify, defaults to tcp:
EXPOSE 80/tcp
EXPOSE 80/udp

ADD
-Copies host files from host source to docker image destination

-*ADD [source] [destination]*
-option: *--chown=[user]:[group]*

-Can have multiple sources, with directories separated by spaces. If dir has space in name, sources must be wrapped in " ".

-Sources may contain wildcard chars, to pull in all matching dirs, etc. Source can also be a file instead of directory.

-Can also copy from locations besides a local directory, including a URL or a TAR file that is extracted into directory

COPY
-Copies host files from host source to docker image destination. Pretty much same rules as *ADD*, except only can copy local files, not URL or TAR unzip copy. This should be preferred over *ADD*.

-*COPY [source] [destination]*
  -option: *--chown=[user]:[group]*

ENTRYPOINT
-Allows you to configure a container that will run as an executable. Does so by always running specified commands, unlike *CMD*, which only runs it's commands if other commands are not passed into the container on *RUN*.

-*ENTRYPOINT [["executable"], ["param1"], ["param2"]] #preferred form*
-*ENTRYPOINT [command] [param1] [param2] #shell form*

-ex. of use: *docker run -i -t --rm -p 80:80 nginx #runs, then removes container*

RUN vs CMD vs ENTRYPOINT
-*RUN* executes given commands in a new layer, the commits

-*CMD* sets default commands for when container runs, but will be overriden if user passes in cmds

-*ENTRYPOINT* does the same as *CMD* but cmds cannot be overriden w/ runtime commands. Use when image is an executable with one purpose only.

VOLUME
-Creates a mount point and marks it as holding persistent data that exists past the deletion of a container that uses it, specifying its filepath in the container

-*VOLUME [filepath]* – filepath can be json array of "" filepaths or just a list of filepaths, separated by spaces

USER
-Sets username (or UID) and optional group id (or GID) to use when running image

-*USER [userOrUID]:[groupOrGID]*

WORKDIR
-Sets the working dir for any *RUN*, *CMD*, *ENTRYPOINT*, *COPY*, and *ADD* instructions that follow. Can be used multi times in Dockerfile, to changing working dirs, as needed.

-*WORKDIR [path]*

ONBUILD
-Adds a command to run immediately after *FROM* when the image is used as a base image for another build

-*ONBUILD [instruction]*

-ex. *ONBUILD ADD . /app/src*
ONBUILD RUN /usr/local/bin/python-build --dir /app/src

STOPSIGNAL
-Sets system call signal will be used when container exits

-*STOPSIGNAL [signal]*

HEALTHCHECK
-Tells how Docker should test a container to ensure it is still working. Ex. Could detect infinite loop occurring on web sever.

-*HEALTHCHECK [options] CMD [command]*

-*Can take options:*
--interval=DURATION (default: 30s)
--timeout=DURATION (default: 30s)
--start-period=DURATION (default: 0s)
--retries=N (default: 3)

-Can take in shell command or an *exec* json array w/ command and params

-ex. HEALTHCHECK --interval=5m --timeout=3s \ #every 5 mins w/ a 3 second timeout
CMD curl -f http://localhost/ || exit 1 #as long as index accessible, return 1 if not

-exit codes:
0: success - the container is healthy and ready for use
1: unhealthy - the container is not working correctly
2: reserved - do not use this exit code

SHELL
-Changes default shell (bash) to be replaced with specified shell, after which shell commands will be sent to it

-*SHELL [[“shellToUse”], [params]]*
-ex. *SHELL ["powershell", "-command"]*

Dockerfile Example
#comments are developer instructions used to build

 # - you should use the 'node' official image, with the alpine 3.10 branch
FROM node:14.5.0-alpine3.10

 # - this app listens on port 3000, but the container should launch on port 80
 # so it will respond to http://localhost:80 on your computer
EXPOSE 3000

```
# - then it should use alpine package manager to install tini: 'apk add --update tini'
RUN apk add --update tini

# - then it should create directory /usr/src/app for app files with 'mkdir -p /usr/src/app'
RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

# - Node uses a "package manager", so it needs to copy in package.json file
COPY package.json package.json

# - then it needs to run 'npm install' to install dependencies from that file
# - to keep it clean and small, run 'npm cache clean --force' after above
RUN npm install \
    && npm cache clean --force

# - then it needs to copy in all files from current directory
COPY . .

# - then it needs to start container with command '/sbin/tini -- node ./bin/www'
CMD ["tini", "--", "node", "./bin/www"]

# once built, test by building, then accessing app on http://localhost
```

Container Lifetimes and Persistent Data

Intro
-Containers typically immutable and temporary

-Docker generally deployed as “immutable infrastructure.” Containers can be re-deployed, but if their config, etc. must be modified, a new container is built and deployed.

-Some containers (ex. DB full of data) can obviously not adhere to this. In such case, these are separated from immutable containers, the same way you would with components requiring state from stateless components in React, via “separation of concerns.”

-Note that in general containers, data is persistent across starts and stops after the container is created, but if a container is deleted, that data is also gone. This persistent data problem is solved by volumes and mounts

-Volumes – a special location outside the docker container file system space, within docker
-Bind mount – link container to host path

Data Volumes
-Volumes exist past the deletion of any container that may be using them and must be deleted in an additional step

-Data lives on host (under /var/lib/docker/volumes/ in Debian), but is mapped to another location within the container (ex. /var/lib/mysql)

-*docker volume ls* – list docker volumes on host

-*docker volume inspect [ID]* – show details on vol. Cannot abbreviate ID. Note, that this does not show containers the vol is connected to. For that, must *inspect* containers.

-*docker volume prune* - removes all local volumes not used by at least one container

-*docker container run -v* – option allows multi potentials: specify the location for the volume in the container (ex. */var/lib/mysql*). " " and give it a name (ex. *my-name:/var/lib/mysql*). You can also mount to an existing vol this way, by having it ref the same name and location. Multi containers can ref same vol. Note that *-v* overrides the default vol location a container would have, if one is specified in the Dockerfile used to build it, etc.

-Suggested to name volumes starting with project name

-*docker container create* – Usually create vol either at *run* or in Dockerfile, but creating this way allows you to specify driver type and driver options, options

-*VOLUME [filepath]* – Dockerfile instruction that creates a volume and specifies it's mount point in the container. filepath can be json array of "" filepaths or just a list of filepaths, separated by spaces. If want to mount container volume at default path, but with unique name, check to see what default Dockerfile *VOLUME* is.

Bind Mounts
-Maps a host file or directory into a container file or directory. Essentially two locations pointing to same file.

-Like volumes, deletion of container does not delete bound data

-Cannot specify in Dockerfile and must be done so at run time: *docker container run -v* [/local/path]:[/container/path]

-Binding mounts is useful for dev environments, as it lets you change files on the localhost, then have the changes immediately reflected in the container.

Real World Examples
-Project on prod is using postgres 9.6.1 with a named volume, default directory. Want to upgrade postgres to 9.6.2 patch version. DevOps stops 9.6.1 container, then replaces with 9.6.2 container, telling new container to use the same named volume with *-v.*

-Have created simple static site via Jekyll, which also runs a local dev server. Run Jekyll container with bind mount set to local site files directory. When change local files via text edit, files also changed on Jekyll container, as seen by refreshing localhost in browser for Jekyll.

Docker Compose

Intro
-Defines and run multi-container applications in a YAML file, then runs via *docker-compose* CLI tool

-Basic steps
1) Define the app's environment with a Dockerfile, so it can be reproduced anywhere.
2) Define the services of the app in a *docker-compose.yml* file, which allows them to work together in an isolated environment.
3) Run *docker-compose up* to run the app

-Allows for multiple isolated environments on a single host. Preserves volume data when containers are created. Only recreates containers that have changed. Allows variables and moving a composition between environments.

-Useful for dev env, as can run multi versions of same project on dev host. Also, can have one instance of project for dev & another instance for testing.

docker-compose.yml
-Used with *docker-compose* command for local dev automation and with Docker Swarm.

-Largely a set of key-value pairs used to declare images and define config for them

-Default filename name is *docker-compose.yml*, but can name anything, then ref via *docker-compose –[filename.yml]*

Template
```
# yaml is nested/blocked by spaces, like Python, with equally spaced indents of any size

version: '3.1'              # if no version is specified then v1 is assumed. Recommend v2 minimum

services:                   # containers. same as docker run
  servicename:              # a friendly name. this is also the DNS/host name inside network
    image:                  # Optional if you use build:
    command:                # Optional, replace the default CMD specified by the image and run cmd
    environment:            # Optional, same as -e in docker run
    volumes:                # Optional, same as -v in docker run
  servicename2:
                            # ….

volumes:                    # Optional, same as docker volume create

networks:                   # Optional, same as docker network create
```

Example
```
version: '2'

services:
```

```
  wordpress:
    image: wordpress
    ports:
      - 8080:80
    environment:
      WORDPRESS_DB_PASSWORD: examplePW
    volumes:
      - ./wordpress-data:/var/www/html

  mysql:
    image: mariadb
    environment:
      MYSQL_ROOT_PASSWORD: examplerootPW
    volumes:
- mysql-data:/var/lib/mysql
```

-See docker docs for details on all keys for setting up compose files. Also, see individual image docs on Docker Hub, which often give examples and details on image specific compose set ups.

docker-compose CLI
-Must be installed separately from Docker on Linux

-Not a production-grade tool, but great for local dev and test env

-*docker compose up* - sets up vols and networks, then starts all containers
-By default, logs all containers in terminal out, but can run in background with *-d* option
-If using compose file named anything but *docker-compose* (ex. *my-compose.yml), specify after -f option*
-Many similar commands to what use with *docker container*, except start, pause, rm, top etc. whole compose app, instead of just solo container

-*docker compose down* - stops all containers and removes all un-needed containers, vol, and networks

Building Images
-Like when starting individual containers with *run*, docker compose will first try and build with cached image, and if not present, then try and pull from Docker Hub

-Can pass in environment variables to *build* to create custom, complex builds

-Useful to separate build logic from *docker-compose.yaml* and instead put build details for each container in a dockerfile, then telling the services to use these files via:

```
services:
  nginx:
    context: /path   #if dockerfile named 'Dockerfile'
```

or

```
services:
  nginx:
   context: /path/Dockerfile-cust-name  #if dockerfile not named 'Dockerfile'
```

-When building via *context* Dockerfile reference, also useful to include instruction *image: somename* to save a local copy of this image with the specified name. This will cause future builds to use this image, instead of building step by step from the Dockerfile, saving time.

-If don't use *image*, will name image in name format *projectname_imagename*

Docker-compose Configuration
-See details for all commands, like *build, environment, depends_on, volumes,* etc. at:

https://docs.docker.com/compose/compose-file/#service-configuration-reference

-For the most part, the basic configuration is very similar to instructions in Dockerfiles

Intro to Swarm

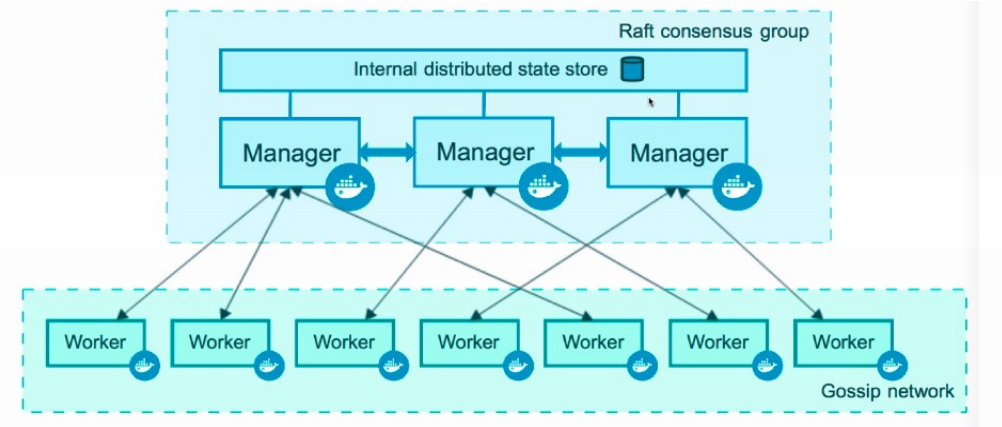
Intro
-Containers make deployment easy through a "one container, many hosts and OS" model.

-Swarm automates the container lifecyle, handling things like handling container failures, replacing containers without downtime (a blue/green deploy), managing nodes running round robins, creating cross-node virtual networks, and security concerns, including PW storage

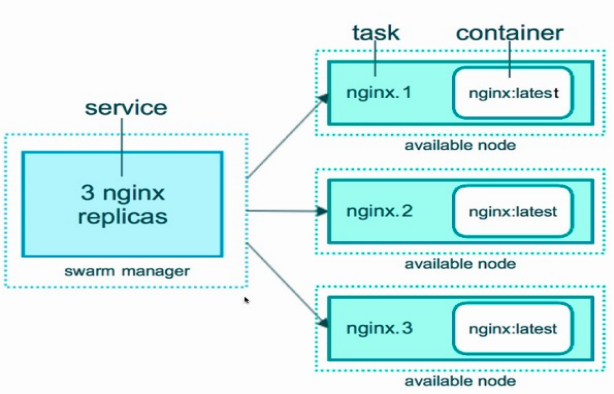
-Swarm Mode is server clustering solution that brings the "swarm" of nodes together, into a single manageable unit

-In addition to providing redundancy, in case one container fails, since swarm managers also allocate work of the app to different workers in the swarm, this also provides automated load balancing, and more efficient processing.

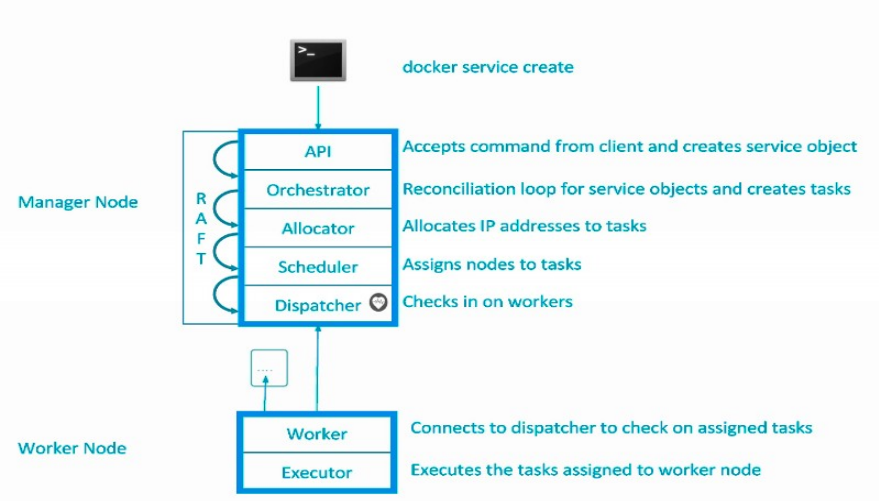
-Part of the base Docker CLI and server, but not enabled by default



-Overview: Internal distributed state store DB exists. Managers control different nodes on the swarm and communicate with workers over “control plane.” All communication encrypted. Managers control the swarm. Workers handle processes, etc..



-Above shows scaling up, where swarm is managing three nginx replica containers. Swarm uses the *service* command instead of *run*. A task is a duplicate container, and each task launches and instance of the container.



-Overview of Swarm API work flow

-logs replicated amongst nodes via encrypted TLS control plane

Basic Use

-To tell if swarm is enabled: *docker info*

-enable swarm: *docker swarm init*

-*init* creates a single node swarm with root signing certificate for the swarm, a certificate for the first node (which is a manager), and then creates join tokens, which are used for connecting nodes. It also builds the encrypted Raft DB, which stores root CA, node configs, and secrets.

-*docker swarm ls* – list Swarm nodes. First manager node in a Swarm created is marked as *leader* node

-*docker swarm demote [nodeid]* – demotes node from manager to worker

-*docker swarm promote [nodeid]* – promotes node from worker to manager

-*docker swarm rm [nodeid]* – removes node from swarm

-*docker swarm ps [nodeid1] [nodeid2]* - list processes running in one or more nodes

service
-Replaces *run* command to start cluster

-*docker service create [imageName]* – creates a single node service using the specified image name

-*docker service ls* – list all services created by swarm. Shows service name, id, and how many replicas of service task running.

-*docker service ps [serviceIDorName]* – lists tasks (containers) running for service, with each task showing the node name it is on, in addition to standard container ID and name

-*docker service update [serviceIDorName] --replicas [someNum]* – Replicates the tasks of the service to the number specified. Ex. Turns service with 1 nginx task into one with 3 tasks. Has a variety of options (ex. *--env-add*, *--host-add*, *--limit-memory*, etc.) that allow to update service w/o stopping it.

-If a task fails, a new task will launch after a few seconds to replace it. This is in opposite to *docker run*, which would not restart if the container fails.

-*docker service rm [serviceIDorName]* – stops service and stops and removes all tasks (containers) for service. Note there is a few seconds delay in this command and containers being removed.

-A cluster requires multiple host machines, with one machine per node. Multiple ways to do this:

Docker-machine Cluster
1) Install *docker-machine* (see Docker site install instructions), then install Virtual Box.

2) Create nodes by running *docker-machine create [nodeName]* once for each desired node. This creates a lightweight linux virtualbox machine for the node.

3) With machine created, can ssh into via *docker-machine ssh [nodeName]* or run *docker-machine env [nodeName]* , which gives various info for connection to, like TCP ip.

-List created docker machines with *docker-machine ls* , remove machine via *rm*, stop via *stop*, etc.

Digital Ocean Cluster
1) Create a linux droplet with an SSH key, at the bottom selecting a number of droplets under “how many droplets,” with names like node1, etc. Suggest using $10/month plan for basic small scale 3-4 node projects.

2) Connect to node via *sudo ssh -i my-private-keyfile ip-address*

3) On each server, run *sudo apt-get update -y && apt-get upgrade -y && apt-get install -y curl && curl -fsSL https://get.docker.com -o get-docker.sh && sudo sh get-docker.sh && sudo usermod -aG docker root*, then check installed properly with *docker -v*. After install, reboot with *sudo reboot*

4) On each server, init swarm first with *docker swarm init --advertise-addr [serverIP]*. On the server you want to hold the leader node, copy the returned *docker swarm join* command.

5) On the non-leader nodes, run *docker swarm leave --force*. This removes the nodes from their individual swarms. Then, on each of these no-swarm nodes, run the copied *join* command to connect them to the leader node swarm. Lastly, on the leader node, run *docker node ls* to get the node ids for the newly joined nodes, and run *docker node update --role manager [nodeNameorID]* on the leader node for each newly joined node to promote these nodes back to managers.

More on Nodes
-If want to add node to swarm as manager by default, get manager join token on node adding nodes to via *docker swarm join-token manager*

-Note that join tokens can be flushed and re-generated, in case a swarm is hacked, etc.

-When services on a swarm are created, Docker automatically chooses which node the service, and its replicates, should be running on. For example, even if you run *service create* for an image on node1. The swarm may very well create and run it on node2. Likewise, if you create a service with 3 replicates on node1 in a swarm with 3 nodes, Swarm will likely choose to run one replicate on each node1, node2, and node3.

Overlay Networking
-A swarm wide bridge network which allows container-to-container communication across all nodes in the swarm

-Create via: *docker network create --driver overlay [networkName]*

-By default, all swarm management data is encrypted, and communication of such data through manager nodes, etc. is also encrypted. To also encrypt network data, add the option *--opt encryption* when creating and overlay network. Note this might result in a performance cost.

-Services can be connected to multiple overlay networks. Ex (DB on a back-end network and web servers on front-end network with API between the two on both networks)

Overlay Network Setup
1) Create network via: *docker network create --driver overlay [networkName]*

2) Create service(s) and set run on network:
docker service create --name [myServiceName --network [networkName] [imageName]

-When creating services via docker-compose, DNS/host name of service is name given to service. Same applies for services created with Swarm, through name given by *--name* option on creation, etc.

Routing Mesh

-For a service running with public facing ports (ex. Drupal on 80), this service can be accessed via that port on any one of the swarm's node's IPs. For example, even though nginx might be running on just node2, it can be accessed by the IP of node2, as well any other node IP on the swarm. This is due to Swarm's routing mesh.

-Routing mesh – routes ingress (incoming) packets for a service to the proper task on the proper node. Spans all nodes on the swarm and uses IPVS from the Linux kernel for routing. Listens on all nodes for traffic, then load balances swarm services across their tasks.

-Swarm builds a virtual IP for it's service. When the front end references services, it talks to this VIP. If a service has replicates across the swarm, Swarm then load balances between the replicates, and chooses which service on which Swarm IP (node) to handle the processing.

-Request with replicates: Front-end request sent to one node IP → request routed to VIP → Swarm chooses which Node replicate should handle request → sends to that node's IP → responds to client

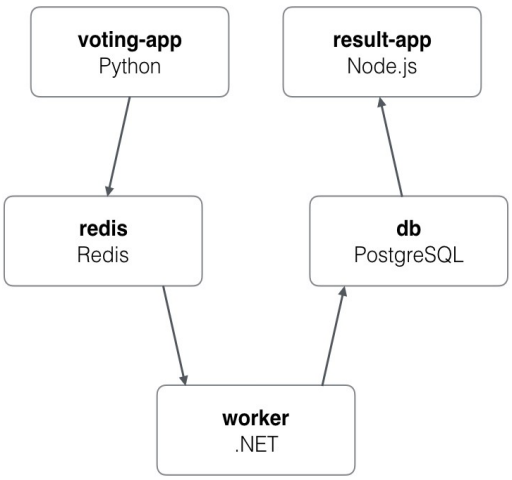
-Request without replicates: Front-end request sent to one node IP → request routed to VIP → Swarm determines which node is holding service for required task → sends to that node's IP → responds to client

-Swarm's routing mesh is stateless. This can result in problems with things like session cookies, which will require additional configuration to work.

-Swarm's routing mesh is on OSI layer, not DNS layer. This means additional config if running multiple websites on the same port, on the same server.

-Both of the above configs can be solved via an nginx proxy, which acts as a stateful load balancer for the mesh

Network Overlay App Example



required front and back end networks

docker network create --driver overlay backend
docker network create --driver overlay frontend

overlay network linked services with requested volume and auth config
docker service create --name vote --network frontend -p 80:80 --replicas 3 bretfisher/examplevotingapp_vote

docker service create --name redis --network frontend redis:3.2

docker service create --name worker --network frontend --network backend bretfisher/examplevotingapp_worker:java

docker service create --name db --network backend --mount type=volume,source=db-data,target=/var/lib/postgresql/data -e POSTGRES_HOST_AUTH_METHOD=trust postgres:9.4

docker service create --name result -p 5001:80 --network backend bretfisher/examplevotingapp_result

Prod Deploy: Stack
-Stacks - an additional abstraction Docker layer on top of Swarm. Accept compose files, which hold a declarative definition for services, networks, and volumes. Similar to the functioning of *docker-compose*, but for Swarm.

-First, build stack compose file, then, deploy using: *docker stack deploy*

-Comparison:
  various *docker container run* commands executed together → single *docker-compose*
  various *docker service create* commands executed together → single *docker stack deploy*

-Stack auto manages services similar to compose. Compose usually used for dev and test sides and Stack used for prod deploy.

-Instead of *build:* instruction, Stack has *deploy:*. Swarm will ignore *build* and keep running. Compose will ignore *deploy* and keep running. Other instructions are the same. Result is can write a single compose file to be used with both Stack and Compose, with *build* and *deploy* commands, where Compose will run *build* and ignore *deploy*, and vice versa.

-Stack also adds secrets, which is Dockers solution to securing information like passwords, etc., that would be stored in environment variables in Compose, etc.