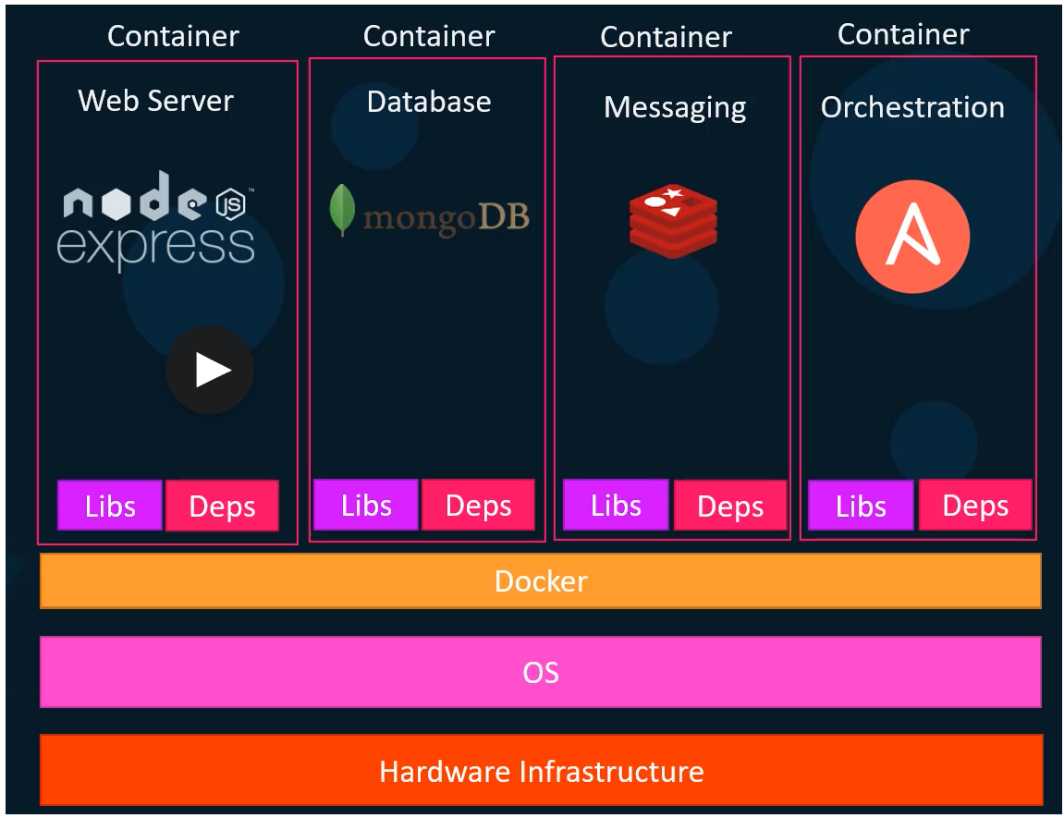
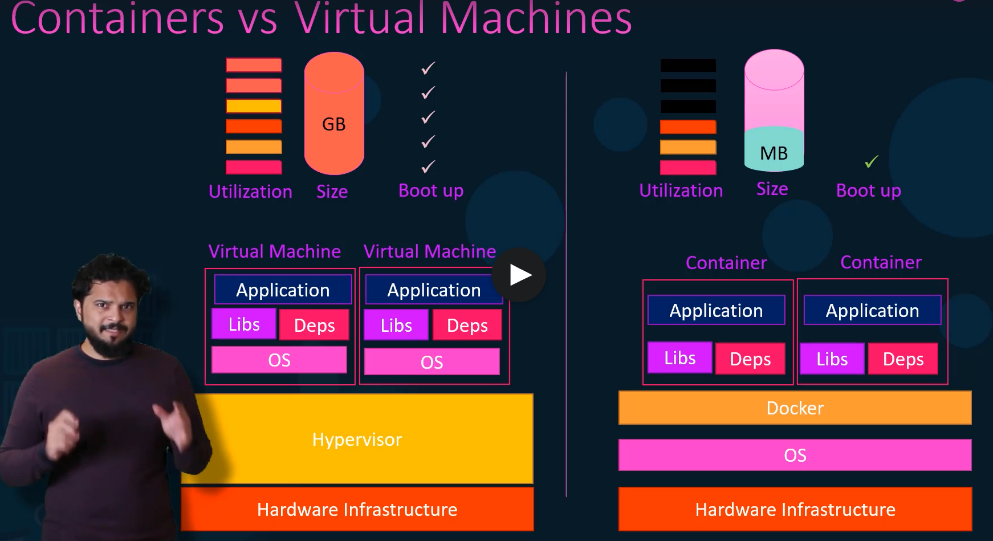
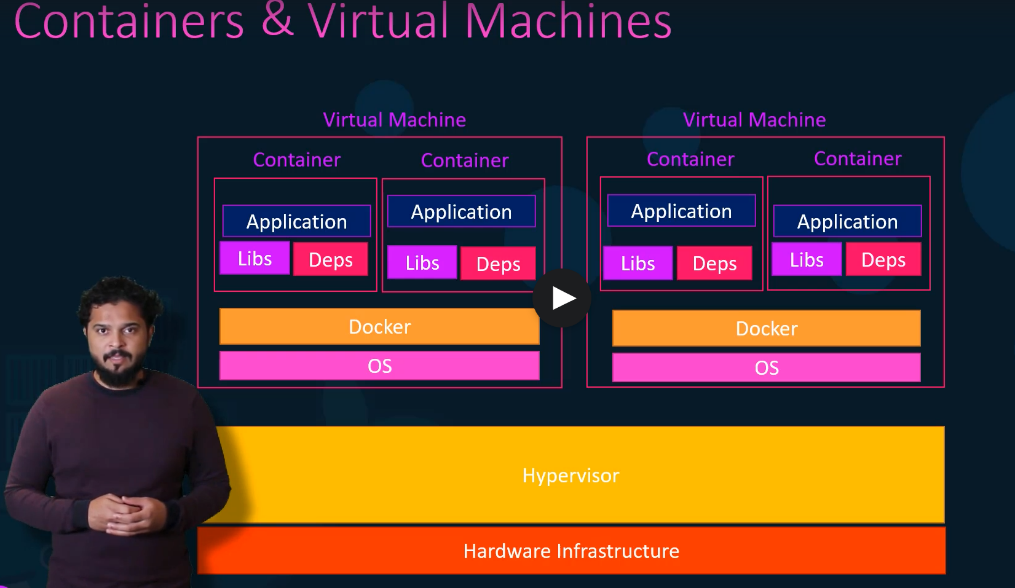
Docker allows you to run multiple different containers, each having its own set of libraries and dependencies. But then all containers can run on the same VM.



Docker run on any linux distribution can run a container using any other linux distribution but not something which doesn’t share the same OS kernel (in other words not a container running Windows)

But if you install Docker on Windows, you can run containers with linux OSs because Windows runs a linux virtual machine and that linux VM is the basis of the shared OS kernel for docker.





Docker has two different version including: a community edition and an enterprise edition

**Install on Windows:**

To install docker on windows I simply went to https://docs.docker.com/desktop/windows/install/ , didn’t even read about all of the possible dependency issues and clicked “docker desktop for windows”. I also clicked the WSL2 integration box so it would work with my WSL I have installed.

Then after my computer restarted docker was working fine from both its GUI and from the windows terminal.

I clicked on docker and settings in the system tray and checked the box with “Use the WSL2 based engine). After this I was able to run docker from my WSL terminal just fine.

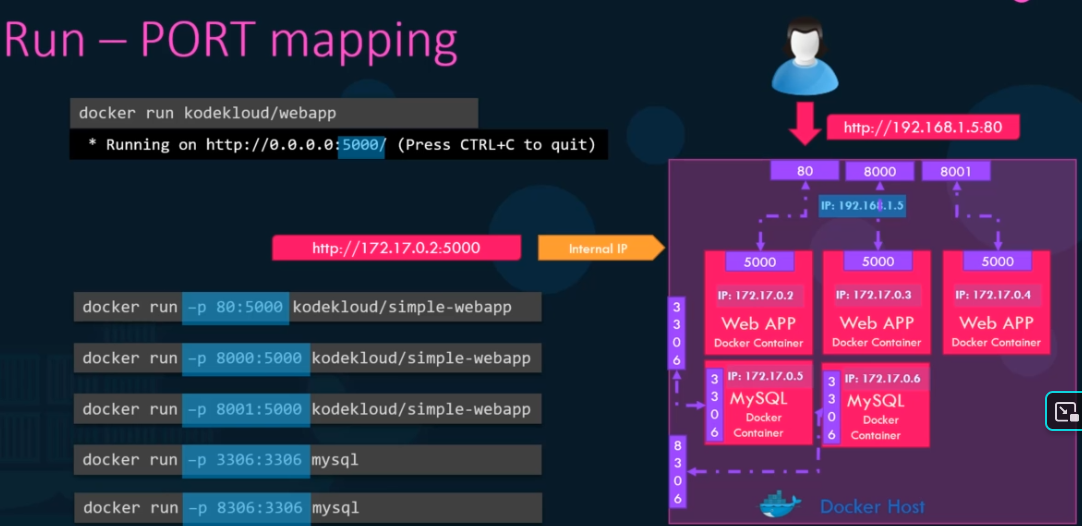
**Fundamental Notes:**

* A container only runs as long as the process inside it is running. If the process/service stops then the container turns off.
* If you don’t specify running in “detached-mode” then you will have to manipulate the container from a separate terminal. So use “-d” if you want to still use the same terminal
* The **order** of arguments DOES MATTER.
  + For example, I found this command had to have the image name trying to run at the end:
    - docker run -d --name webapp nginx:1.14-alpine

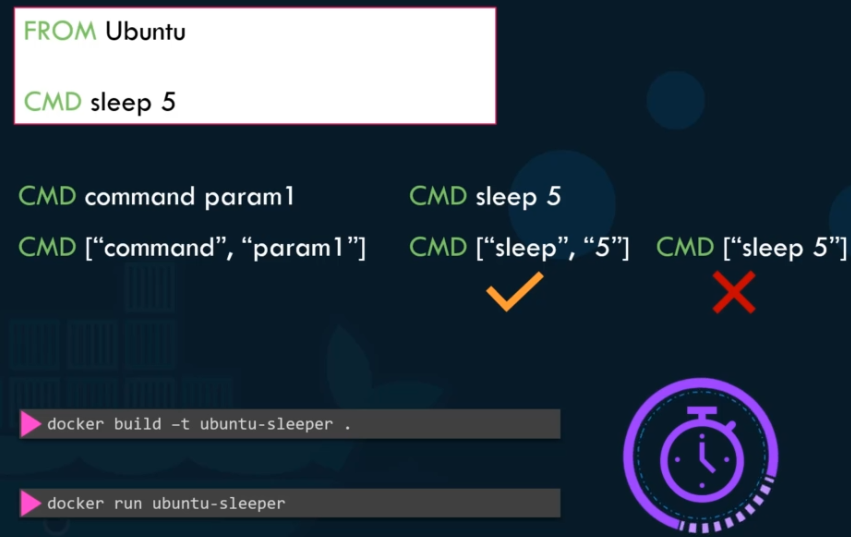
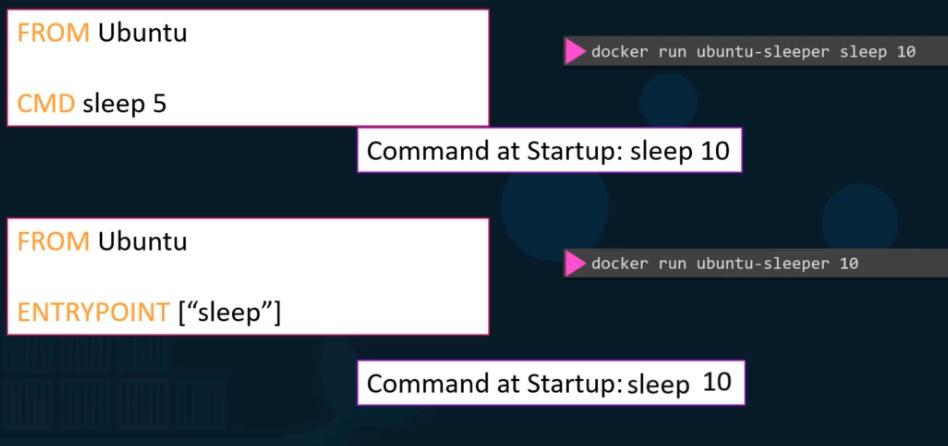
**Command Basics:**

|  |  |
| --- | --- |
| *docker ps* | lists information about all running containers |
| *docker ps -a* | lists information on currently running and previously run containers |
| *docker stop <name or id>* | stops a currently running container |
| *docker images* | displays all of the currently downloaded container images you have locally |
| *docker rmi <image>* | removes the identified docker image from your local repository |
| *docker rm <container>* | removes a specific container/instance from your history |
| *docker run ubuntu sleep 5* | will launch the container and keep it running for 5 seconds |
| *docker exec <name or id> <command>* | runs your command inside the specified container; if it is already running of course – so use detached mode or a separate terminal |
| *docker run -d ubuntu sleep 50* |  |
| *docker pull “<user>/<image\_name>* | pulls down a specific user’s named image |
| *docker run -it ubuntu bash* | opens a bash shell inside the ubuntu container |
| *docker rm $(docker ps -a -q)* | remove all docker images  (all containers based on these images must be stopped first) |
| *docker run -v /path/outputdir:/path/inside/container container\_name* | map a directory on the host to a directory inside a running container |
| *docker inspect container\_name* | provides json-formatted details on a container versus the simpler  *docker ps* command |
| *docker logs container\_name* | grabs container logs |
| *docker run -p hostport:contport image\_name* | port routing example |
| *docker run -e APP\_COLOR=blue image\_name* | example showing the passing of an environment variable to a container |
| *docker history image\_id* | can be a way to see some of the commands from the Dockerfile when you never saw that Dockerfile |
| *docker system df -v* | shows you the actual local disc usage on the docker host.  this will show you the total space of image components stored on local disc for when some of those images may share image layers but the command *docker ps* indicates the images are the same ***exportable*** size |

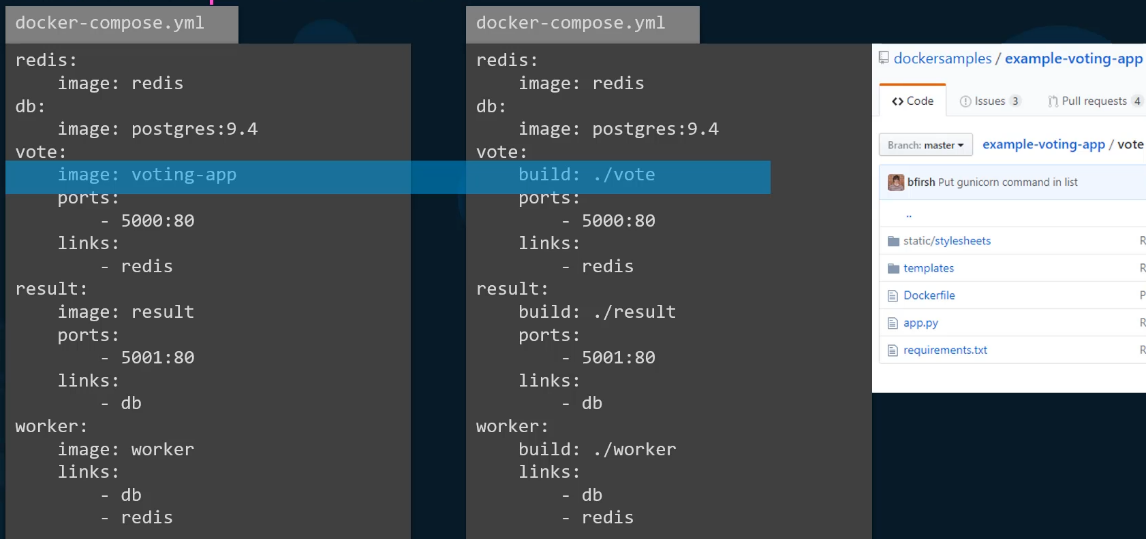
**More General Notes:**

* When you are pulling down images (like those you have made) that are NOT official docker repo images then running/pulling them, you need to identify them like “<user>/<image\_name >
* Mapping ports from the docker host to a specific docker container: 
* While attempting to follow the Jenkins port-mapping example I tried the following to establish the host IP of my machine running docker:
  + From WhatsMyIp utility I got [73.95.58.18](https://www.whatismyip.com/73.95.58.18/) which didn’t work
  + After installing net-tools on WSL ubuntu I ran both:
    - ifconfig
    - hostname -I
    - Both provided the working answer of 172.24.65.66 which I was able to paste into my browser using port 8080 after running the Jenkins with:
      * *docker run -d -p 8080:8080 jenkins/Jenkins*
  + Mapping ports using -p works like:
    - -p <host\_port>:<container\_port>

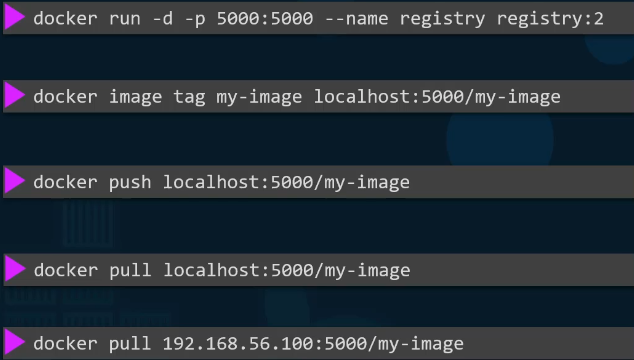
**Image Building Notes:**

* Every docker image must start from another docker image using the *FROM* instruction in the Dockerfile
  + Dockerfile’s first line must always start with this FROM
* Whenever docker builds an image, it builds and **caches each layer** so that all of the layers don’t have to be rebuilt each time your run a build, only the layers that have changed for the current build will be re-built for speed.
* Using environment variables:
  + If you inspect a docker image you can find the list of utilized environment variables under the Config—Env section
* The instruction *CMD* can define the command to be run once a container based on that images is initialized
  + Below shows different dockerfile variations on initial command overrides 
  + To then pass in the number of seconds you would use the *ENTRYPOINT* instruction instead: 
  + The key distinction here is that:
    - *CMD* replaces the initial starting command completed
    - *ENTRYPOINT* appends starting parameters to some partial command like “sleep” here
  + To supply a default value for the ENTRYPOINT instruction, also use a CMD instruction which only contains the default value.

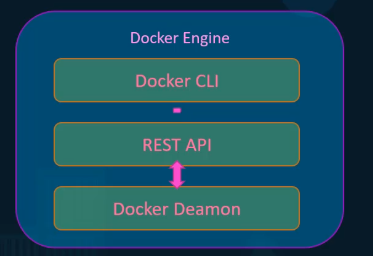
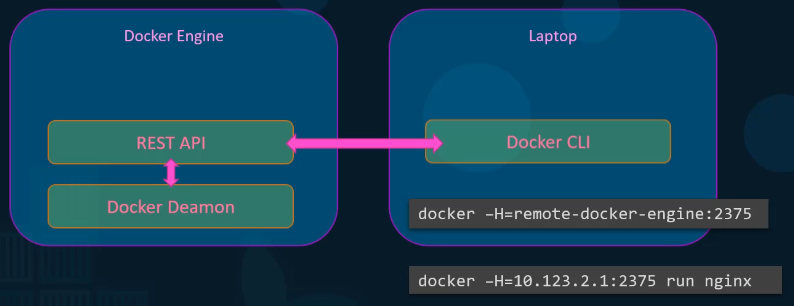
**Docker Compose Notes:**

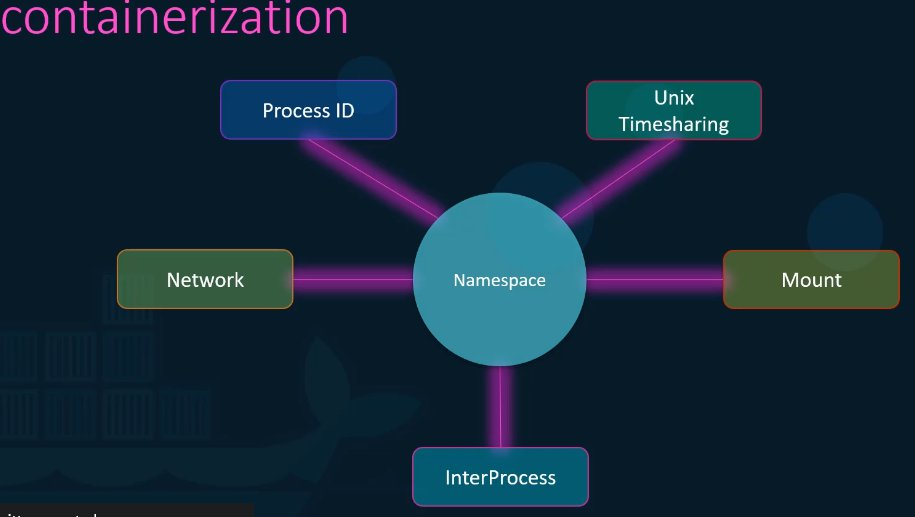
* Docker compose is meant to deploy an entire “stack” of containers on a single docker host.
* If you know that some of the containers you are going to include in your stack deployment are not already built-established images, then you can replace the *image*: keyword in the yaml file with the *build:* keyword as shown in the example below.
  + This of course assumes that the directories you specify under the *build* sections contain the application code(s) necessary as well as their own Dockerfiles governing those build(s)
  + 
* Docker compose has multiple **versions**
  + Version1 did not support deployment/container-up order
    - Order is specified using the “*depends\_on*:” qualifier
  + In Version1 “links” had to be used to connect up different containers on a stack.
    - In Version2/3 all containers are automatically connected to a dedicated “bridged network” that is instantiated when the stack is initialized.
      * No explicit linking is required, and the containers are able to communicate with each other using their corresponding service names
  + Version3 comes with support for Docker swarm

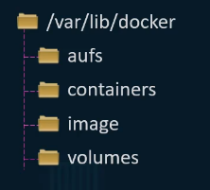
**Docker Registry Notes:**

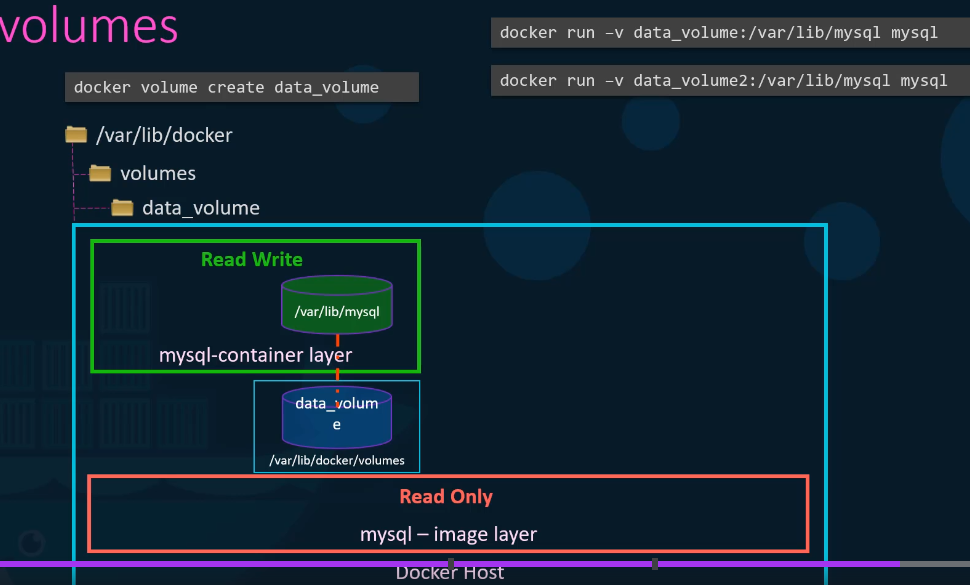
* When you say something like *docker pull nginx* this assumes lots of defaults
  + that you are using the default docker image registry really at docker.io
  + that the user/imageRepo names are both *nginx*
  + really the full path you are pulling is *docker.io/nginx/nginx*
* There are other image repos like googles for instance at *gcr.io*
* AWS/Azure/GCP all provide a private repo through them when you have a cloud account with them
* To pull down docker images that are part of a private repository you would run:
  + *docker login private-registry.io*
  + Then something like: *docker run private-registry.io/apps/internal-app*
* Example running a private-registry on-premise 

**Docker Engine/Storage/Networking Notes:**

 OR 



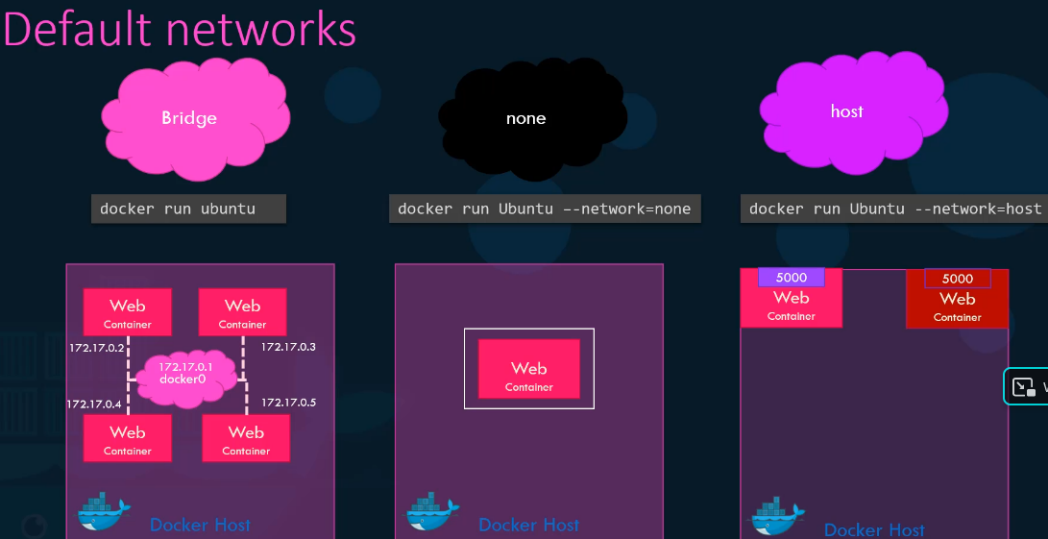
* Namespaces (for one at least) allow process ids inside a container to share the same id as a different process running on the host.
* By default there is **no** **restriction** on the amount of host resources (cpu power, memory, etc.) that an individual docker container may utilize/consume
  + cgroups are ways to restrict the amount of resources allocated to containers
    - *docker run --cpus=.5 ubuntu*
    - *docker run --memory=100m ubuntu*
* By default, docker installs under */var/lib/docker* on the docker host. 
* Diagram for “volume mounting” in docker:

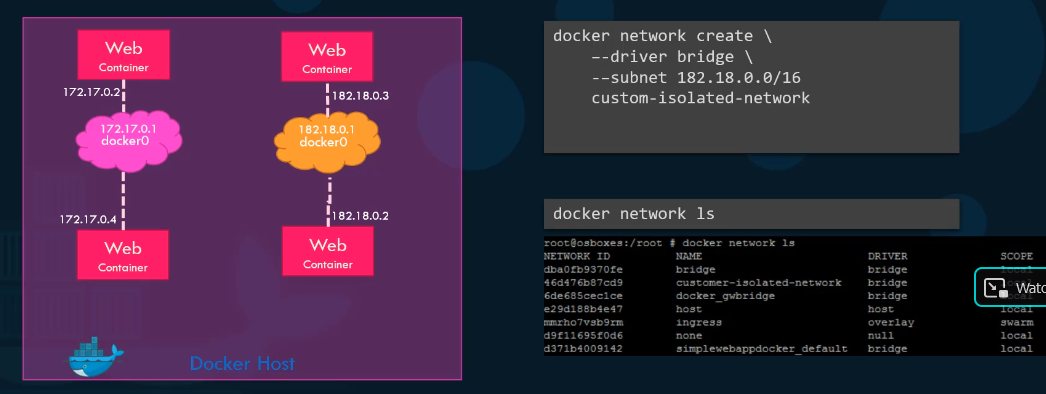
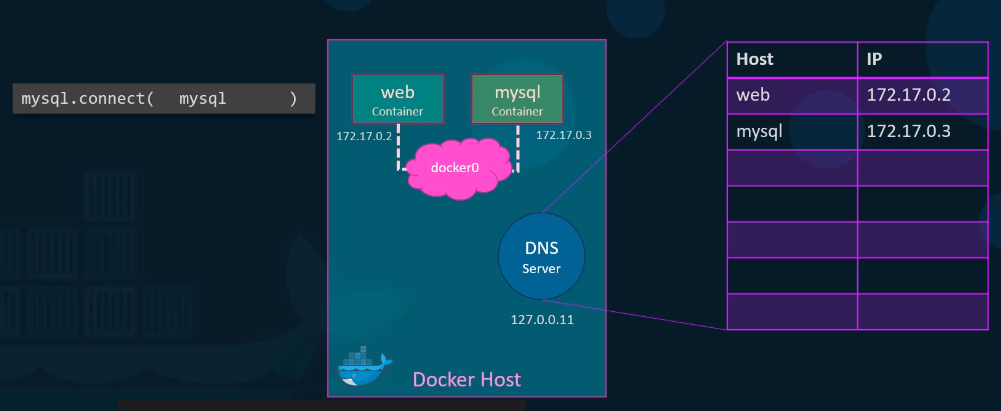


* A “bind-mounting” command works basically the same, but mounts another directory anywhere on the host system to the container instead of a directory inside of /var/lib/docker/volumes

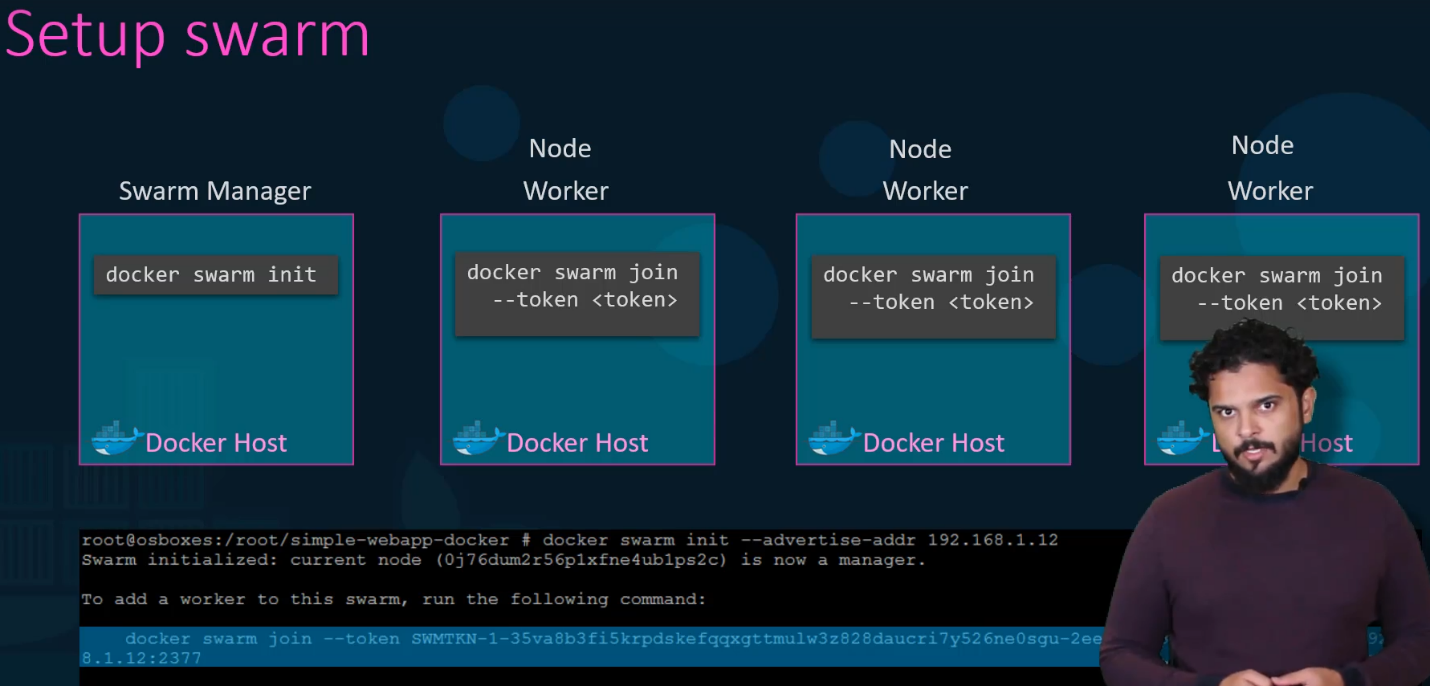
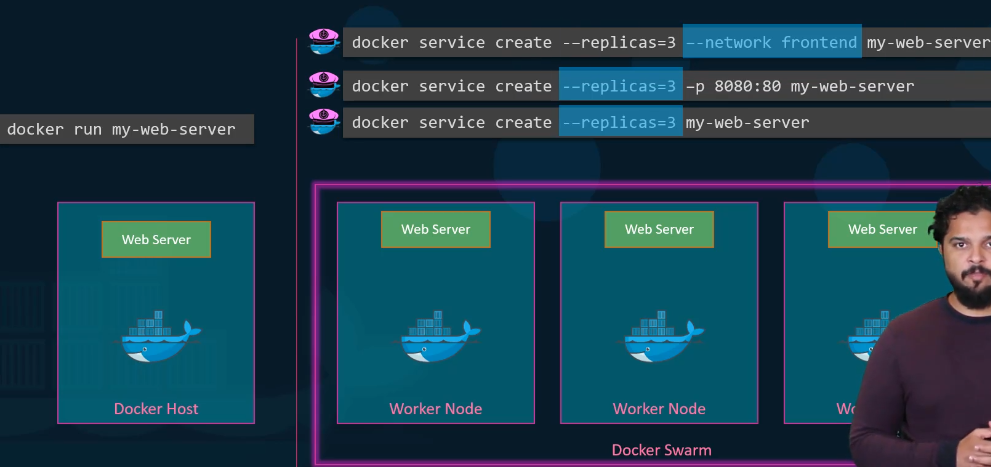


* This command format using *-v* is the older style versus *--mount* newer style commands like: 
* There are “storage drivers” underlying the various docker host OS possibilities which you might want to explore or optimize for certain applications
* Docker creates three networks automatically upon installation: **Bridge, none,** and **host** 
  + bridge is the default, and you can specify one of the others with:
    - *docker run ubuntu --network=host*



* Below is an example of overriding the default behavior of only generating one-bridged network and instead creating two separate bridged networks:
* Rather than using container-IPs for communicating between containers running on a single host, you should just use the container name.
  + Using the container IPs will work; so long as you are using the right ones at any given moment
  + Docker has a built-in DNS server for this purpose
  + This precludes the possibility that when a container reboots its IP changes – its name won’t change!
  + 

**Container Orchestration with Docker Swarm and Kubernetes:**

* Various container orchestration services are designed to manage multiple docker hosts each running multiple docker containers and performing tasks like:
  + Maintaining health of containers and hosts; replacing either when necessary
  + Scaling up and down based on user demand or other metrics
* 
* 
* Kubernetes basics: 