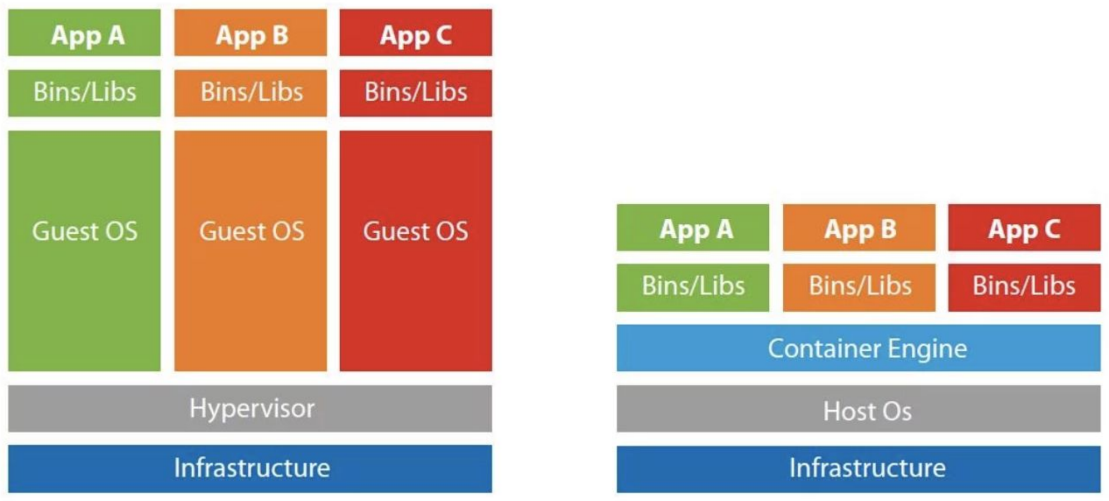
Topics Covered:

- Virtualisation
- Containerization
- Creating Docker images
- Docker compose
- Docker orchestration using Swarm or Kubernetes

Virtualisation

A virtual machine is a computer file, typically called an image, that behaves like an actual computer. In other words, a computer is created within a computer. It runs in a window, much like any other program, giving the end user the same experience on a virtual machine as they would have on the host operating system itself. The virtual machine is sandboxed from the rest of the system, meaning that the software inside a virtual machine can't escape or tamper with the computer itself. This produces an ideal environment for testing other operating systems including beta releases, accessing virus-infected data, creating operating system backups and running software or applications on operating systems they weren't originally intended for.

Virtualization technology has serious drawbacks, such as performance degradation due to the heavyweight nature of virtual machines(VM), the lack of application portability, slowness in provisioning of IT resources



A Docker container is a software bucket comprising everything necessary to run the software independently. There can be multiple Docker containers in a single machine and containers are completely isolated from one another as well as from the host machine.

So essentially the main difference between virtualisation and containerisation is that with containers everything is faster and more lightweight. This is because when you are creating a virtual machine, you need to create a guest OS and have an underlying hypervisor to orchestrate all the guest OS’s on the same hardware. However with containers, you are essentially reducing this down to just having the software bucket without the need for a guest OS, meaning that it is far more lightweight.

Docker

https://medium.com/@kelvin_sp/docker-introduction-what-you-need-to-know-to-start-creating-containers-8ffaf064930a

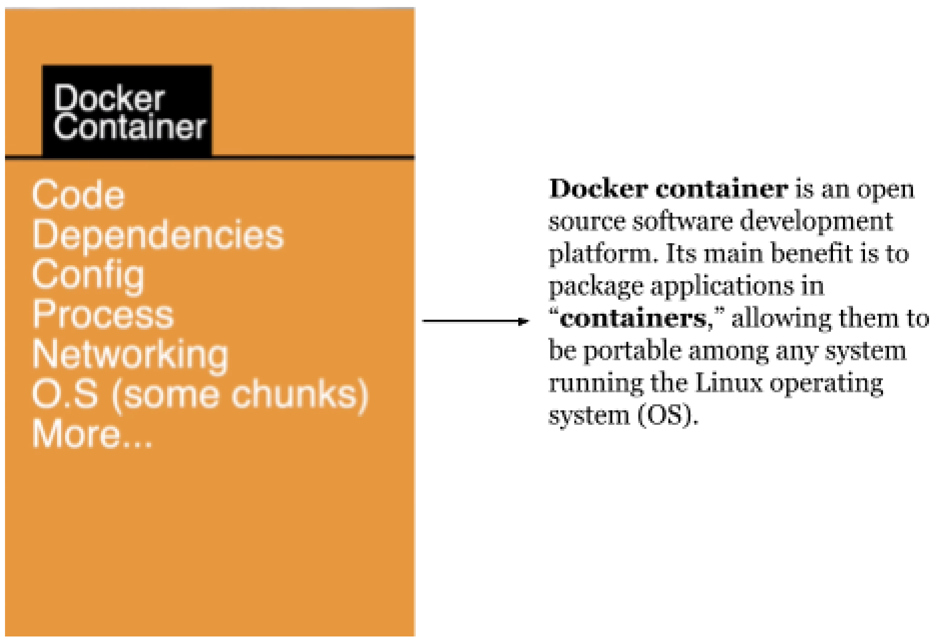
https://blog.usejournal.com/what-is-docker-in-simple-english-a24e8136b90b

https://www.docker.com/resources/what-container

When you are developing software, there comes a point when you want to ship the product. Some people say things like “it works on my machine” however this is not enough. The app needs to be able to work on all machines regardless of the platform or whatever.

So Docker is used to address the problem of “it works on my machine…”. How it does this is by containerisation.

Docker basically provides you with containers that are portable. So if you take this container from your machine to some other machine, it will work exactly how it worked on your machine. Docker also allows for social containers which means that you can share your docker containers with other people easily.



Docker container is an open source software development platform. Its main benefit is to package applications in “**containers**,” allowing them to be portable among any system running the Linux operating system (OS).

So to summarise, docker is a client side application. You install it on your machine and if you want to create a container, you can do so using a terminal command. Once you create a container, you can give this container to anyone else that has the docker client installed who can then take your container and run your program the exact way that it ran on your computer and the way you intended for it to be run.

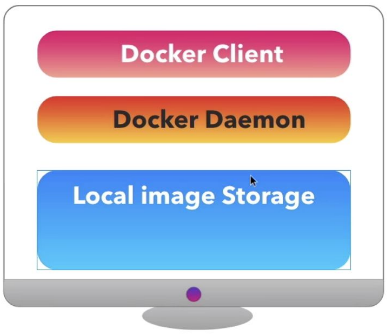
This is helpful in production because you can install the docker client on your server and it will run the code EXACTLY how you ran it on your dev machine.

Real life example:

Imagine you have a class of students and you want to teach them how to use templates in django. You can make them each install django on their respective platforms and run into various problems OR you could make use of docker containers and create a docker container for the project yourself beforehand. Then all the students can just use this socially shared docker container to get up and running without having to worry about long installations etc.

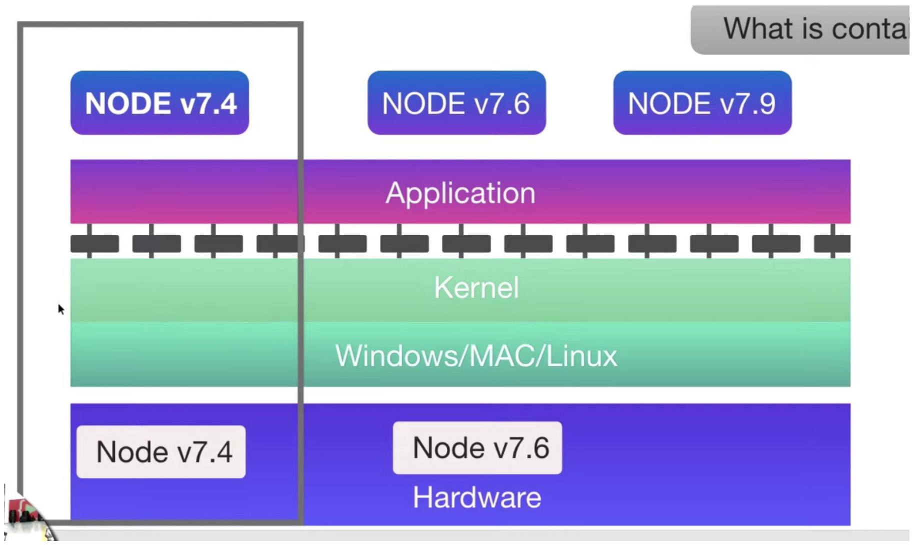
Docker in Action

Following this series :: here

1. Install docker by going on their website and signing up
2. When you do something like docker run hello-world heres what happens
    a. 

    b. When you run the command on the docker client, the docker daemon checks to see if you have the image you wanted, stored locally in the local image storage. If it is, then it retrieves it else it will retrieve it from the dockerhub online.

Now lets understand exactly what a container is:

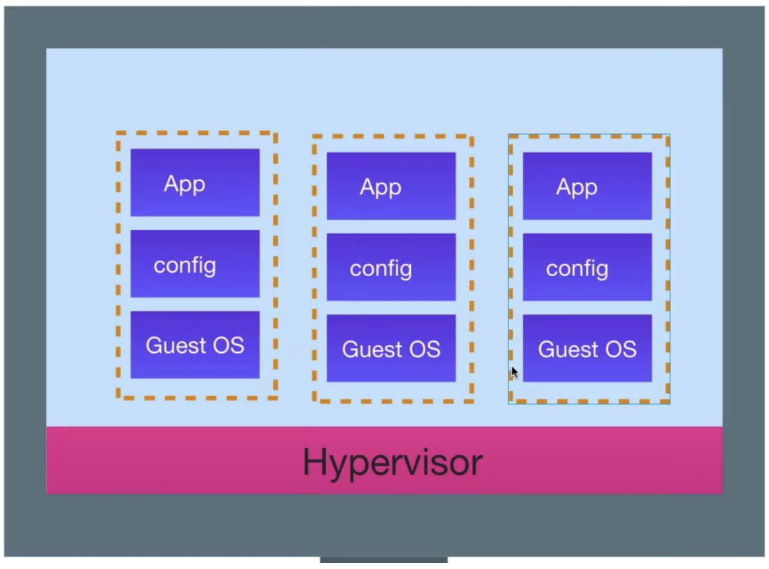


Imagine you have installed one version of node (7.4) on the hardware. You have multiple applications running different versions of node like 7.4, 7.6, 7.9 etc. The only one that will work with the hardware is 7.4. This is a problem. Because we can't just upgrade the version of node at the hardware level because this will mean that other applications will break. So how can we solve this problem? An immediate suggestion could be to store multiple versions of node at the hardware level. This is called segmentation or namespacing which allows you to dedicate specific disk space for different versions of the same program.

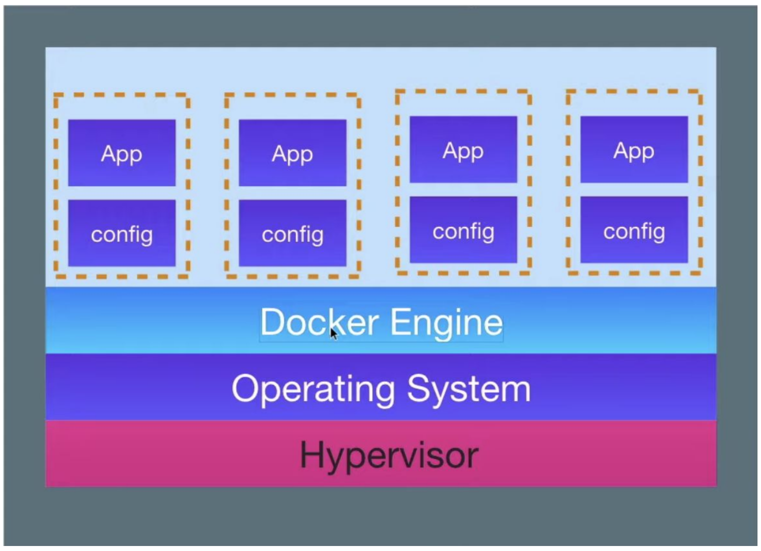
So now we can create a container that includes the following:

- The state of the application that you are trying to run
- Abit of the OS, system calls and kernel
- And the hardware specifications such that wherever this container is deployed, you can use the exact same segmentation on there. I.e take up some hardware space and install the version of the program you need right there.

This sounds a lot like a VM. What’s the difference?



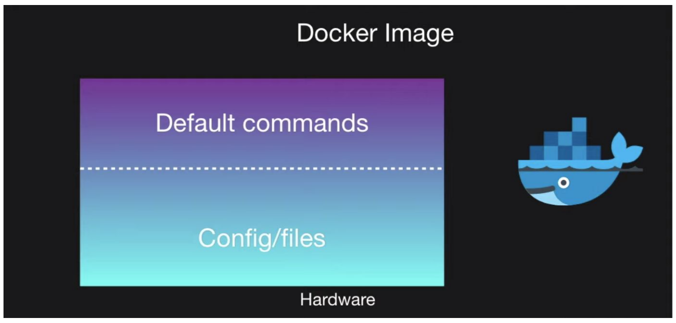
This is a VM. It has a layer known as the hypervisor which allows the host machine OS to be used by all the other guest OS's. So imagine in an industry grade application where you want to run your app on a server using a VM. Your app may be 300mb however with the guest OS, the size of your VM is much greater! This means that it will consume more resources hence running up your tab with the cloud providers.



This is a container. You see that the host machine now has an OS layer on top of the hypervisor as well as the docker engine layer. The containers now dont each have their own OS hence they are much more lightweight and are all sharing the OS of the host by going through the Docker Engine. The docker engine is clever because even if it is always working on linux, you can still use it when you are on a different OS.

Now let's dive into the practical side of things:

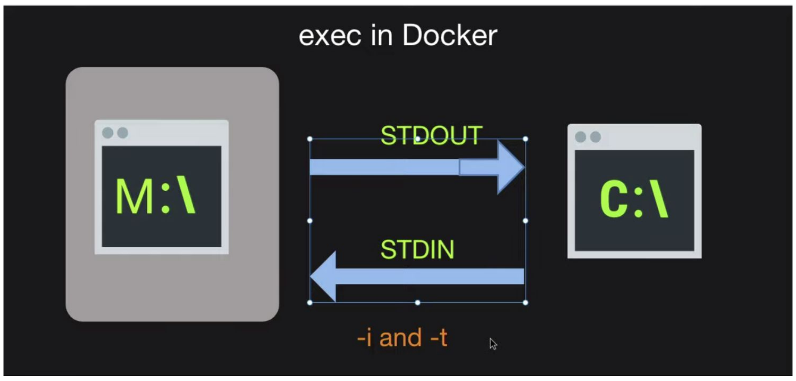
Docker image



A Docker image is a file, comprised of multiple layers, used to execute code in a Docker container. An image is essentially built from the instructions for a complete and executable version of an application, which relies on the host OS kernel. When the Docker user runs an image, it becomes one or multiple instances of that container. So it's essentially like a snapshot of the system of whomever created the image and then everyone can pull this image and run the container on their docker client

Exploring the exec command

https://docs.docker.com/engine/reference/commandline/exec/



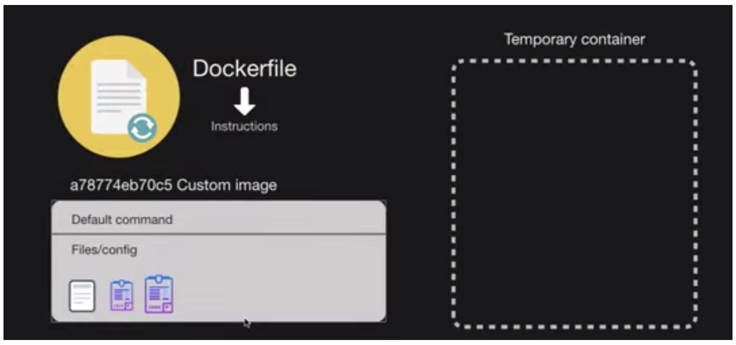
When we use the command docker exec -it container_id bash , we are trying to run the command line inside the container. How this is working can be seen by the diagram above. We are changing the standard input for the container's bash to map directly to what we as the user types into the command line on our physical machine. The standard output from the container's bash then also gets mapped onto the output of our physical command line.

Custom docker images

1. Create a file called Dockerfile , pass it to the docker client and it spits out the image!

But what do you actually write in your Dockerfile:
1. Get the base image. The base image is like the operating system
2. Install a software and configure it
3. Set default commands

You then run docker build. In the terminal and this will create your custom image. But how does it actually do this :

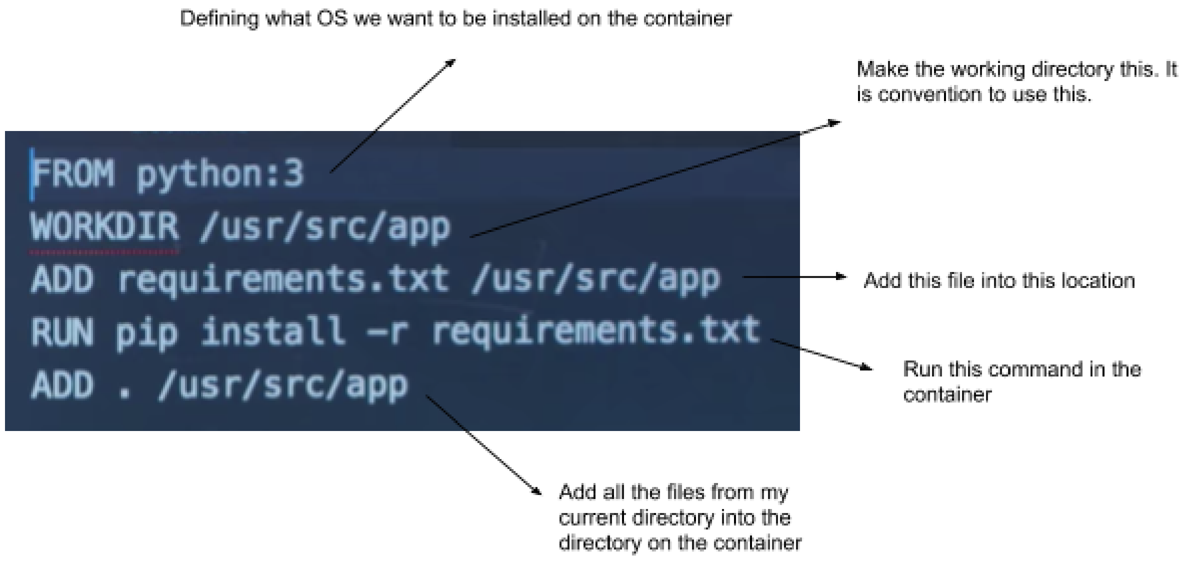


So we have our dockerfile. We have run the docker build command. What happens is that we first install the new OS into the image file. Then we create a temporary container to do all the software installs and configs. Once all of that is done, they are copied over to the custom image and the temporary container gets deleted. We then get our image ID.

<u>Docker python examples from cs50</u>

- He created a Dockerfile, which defines an image which defines how to create a container.

Dockerfile:

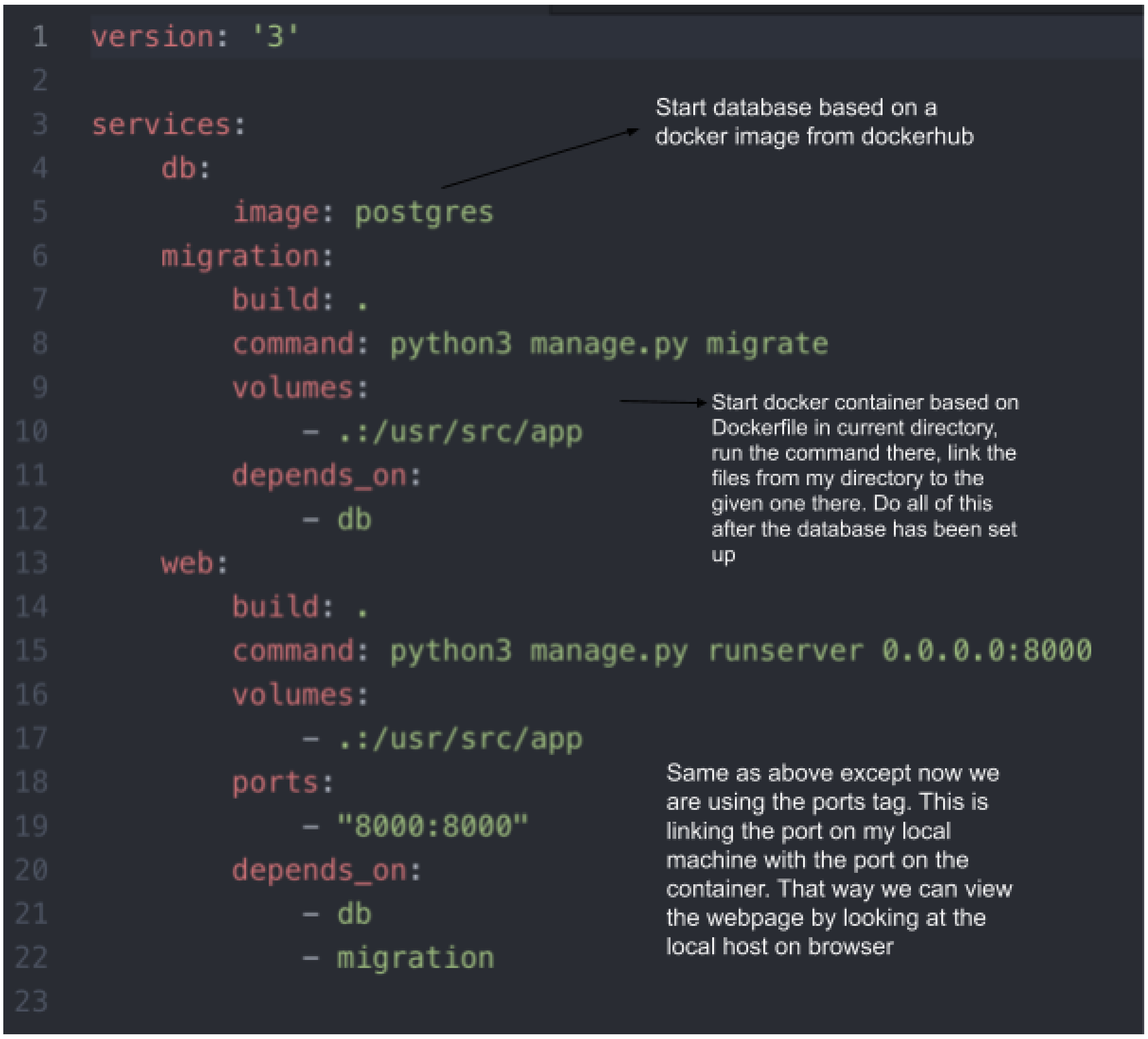


```
FROM python:3
WORKDIR /usr/src/app
ADD requirements.txt /usr/src/app
RUN pip install -r requirements.txt
ADD . /usr/src/app
```

Defining what OS we want to be installed on the container
Make the working directory this. It is convention to use this.
Add this file into this location
Run this command in the container
Add all the files from my current directory into the directory on the container

So this file can be run to create an image which can then be run to create a container

The next step is to create a docker-compose.yml file which will allow us to build these containers and do things with them.



```yaml
version: '3'

services:
    db:
        image: postgres
    migration:
        build: .
        command: python3 manage.py migrate
        volumes:
            - .:/usr/src/app
        depends_on:
            - db
    web:
        build: .
        command: python3 manage.py runserver 0.0.0.0:8000
        volumes:
            - .:/usr/src/app
        ports:
            - "8000:8000"
        depends_on:
            - db
            - migration

```

Start database based on a docker image from dockerhub

Start docker container based on Dockerfile in current directory, run the command there, link the files from my directory to the given one there. Do all of this after the database has been set up

Same as above except now we are using the ports tag. This is linking the port on my local machine with the port on the container. That way we can view the webpage by looking at the local host on browser

So then to actually get this up and running you do :

docker-compose up

This will start the server. You can then make changes locally and because of the volumes line in the docker compose, any changes you make will automatically go to the container.

Docker commands cheatsheet

docker ps --all : lists out all containers you are running
Docker run image_name : creates container from image if found locally or pulls it from docker hub
docker start container_id : start up a specific container by ID
docker exec -it container_id bash : run a command inside the container. In our case, we are choosing to run the bash command which gives us access to the command line inside the container

To exit the container use control + d
To stop a container from running you can use docker stop container_id or docker kill container_id
To create a custom docker image, call the file Dockerfile
Enter commands into the dockerfile then to create it run docker build . in the directory
docker compose up : this starts up the docker-compose file
docker run -d image_name : this will run the image and start up the container however this is in development mode so you can still use the terminal
docker inspect container_id : use this command to inspect everything about the container including things like environment variables

Docker notes from https://www.youtube.com/watch?v=fqMOX6JJhGo&t=489s

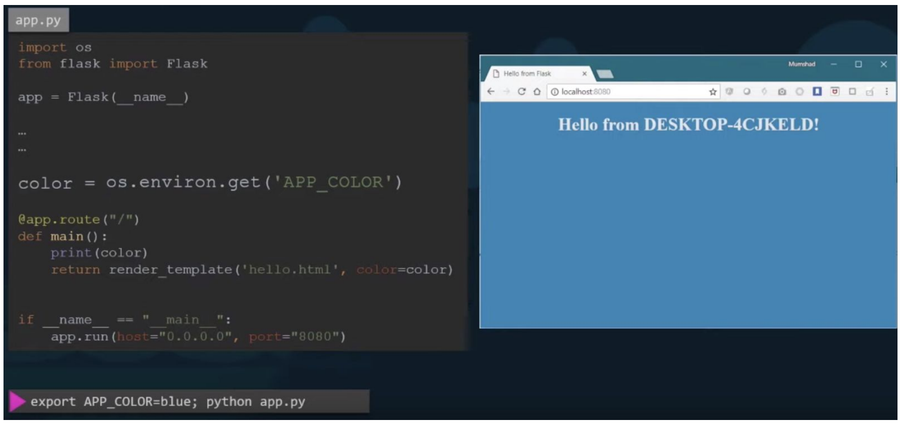
Containers vs VM

Once again, containers are usually in MB size range. Whereas VM's are usually in GB. This means that containers boot up faster (in a matter of seconds) whereas VM's usually take longer (several minutes) to boot up.

Docker in real life

In the past, the developers and operation teams used to work seperately. The developers would create the app and then give it to the operations team that would then take the app with all its requirements and then put it into production. This would usually take a long time due to lack of communication between the developers and ops teams which would cause lengthy issues when the ops teams faced issues. Now things are slightly different. The developers and operations teams work hand in hand. The developers create the app then they both create the Dockerfile which is then used to create a docker image. With the docker image, you can run the application on any machine that has docker installed on it. This means it will run exactly the same as it was running on your local machine, on the production server! Happy Days!!!

Environment Variables



```
app.py
import os
from flask import Flask

app = Flask(__name__)

...
...

color = os.environ.get('APP_COLOR')

@app.route("/")
def main():
    print(color)
    return render_template('hello.html', color=color)


if __name__ == "__main__":
    app.run(host="0.0.0.0", port="8080")
```

Hello from DESKTOP-4CJKELD!

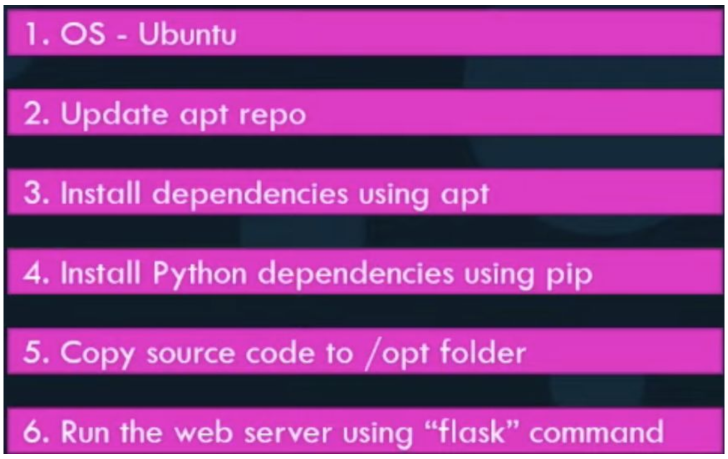
```
export APP_COLOR=blue; python app.py
```

With environment variables, you can use them in your code. Then set them when running your app. See above for more details.

Creating custom images

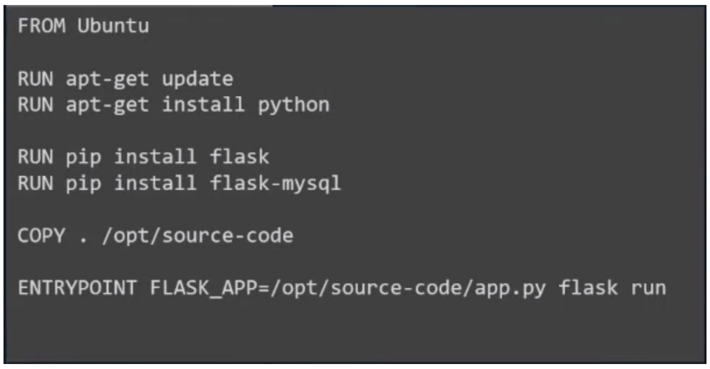
You would want to create a custom image if the image you want cannot be found on docker hub or your team decides they want to do it in a custom manner for ease of shipping and deployment.

If you wanted to do the steps manually it would look something like this:



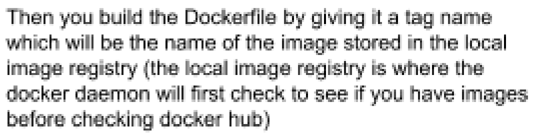
1. OS - Ubuntu
2. Update apt repo
3. Install dependencies using apt
4. Install Python dependencies using pip
5. Copy source code to /opt folder
6. Run the web server using "flask" command

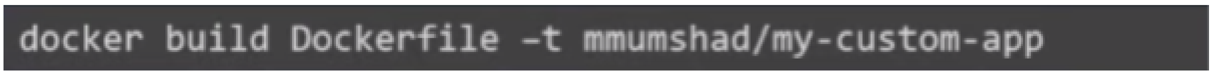
This can be translated into the Dockerfile to look something like this:



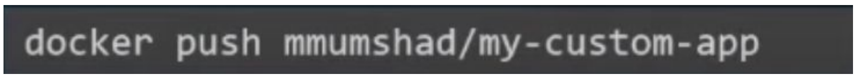
```
FROM Ubuntu

RUN apt-get update
RUN apt-get install python

RUN pip install flask
RUN pip install flask-mysql

COPY . /opt/source-code

ENTRYPOINT FLASK_APP=/opt/source-code/app.py flask run
```

The dockerfile is written in instruction and argument format.


Then you build the Dockerfile by giving it a tag name which will be the name of the image stored in the local image registry (the local image registry is where the docker daemon will first check to see if you have images before checking docker hub)


```
docker build Dockerfile -t mmumshad/my-custom-app
```

To then make the image available on the docker hub registry you do the following:


```
docker push mmumshad/my-custom-app
```

* where you provide it with your username and app name

CMD and Entrypoint

In the dockerfile, you can specify the command to be run when you run the container. This can be done by using the command CMD like this:

- CMD function param1

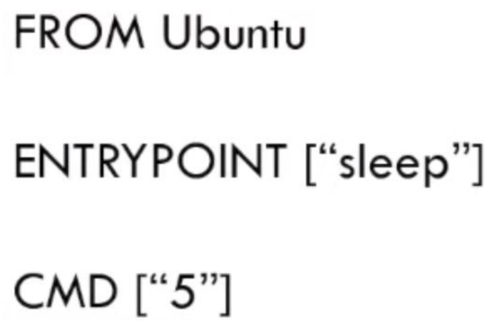
However, what if you want to specify the parameters when you are actually running the container rather than hard coding it.In that case you use the entrypoint command like this:

- Entrypoint function

Then when you run the container, you pass the parameter like this:

Docker run image_name parameter

If you want to have a default value in case the user does not provide a parameter, then you can use the CMD function to specify it:


```
FROM Ubuntu

ENTRYPOINT [“sleep”]

CMD [“5”]
```

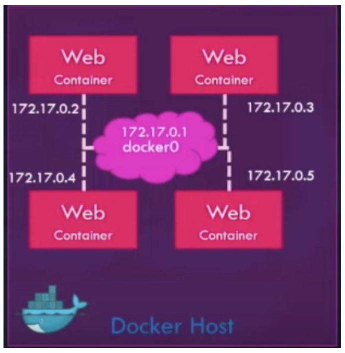
Networking with containers

There are 3 main options for networking:

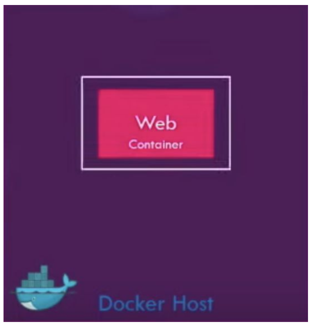
- Bridge
- None

- Host

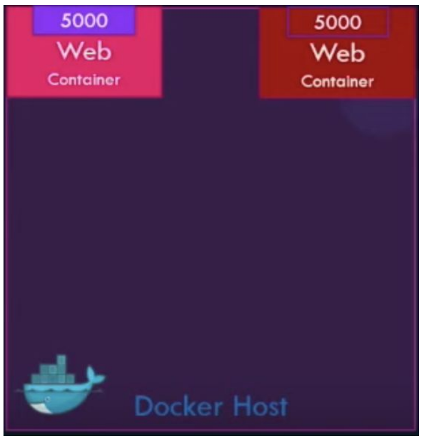
If you set up your container with the network parameter as **bridge** like this "docker run image", then all of the containers will be able to communicate with each other on that network.



If you set up the network as **none** like this : docker run image --network=none then the container will run in an isolated environment. It won't be able to communicate with other containers.



Running the container with network parameter as **Host**, you are essentially, removing any isolation from the docker host and docker container. This means if you were to run a web server on say port 5000, you wouldn't need to do any port mapping because they are common to the host network. docker run image --network=host



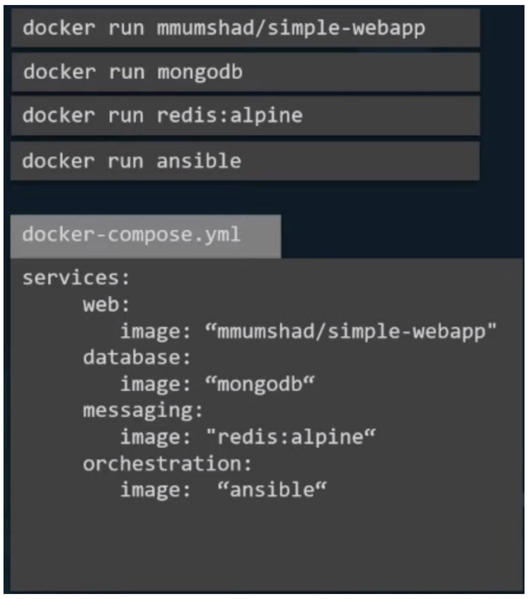
To find out the details of which network the container is in: run the inspect command like this:
Docker inspect container_id
Then you can find out which network the container is in and also other properties like the IP address, mac address and more.

Embedded DNS

How can you connect to containers together. So imagine you had a web server and a database in two different containers. You could connect to the database by using the IP address of the database however this is unreliable as the IP address of the database may change every time you run the container. So what docker has embedded internally is a DNS system. So we can refer to the other containers by their name and docker will resolve this internally.

Docker Compose

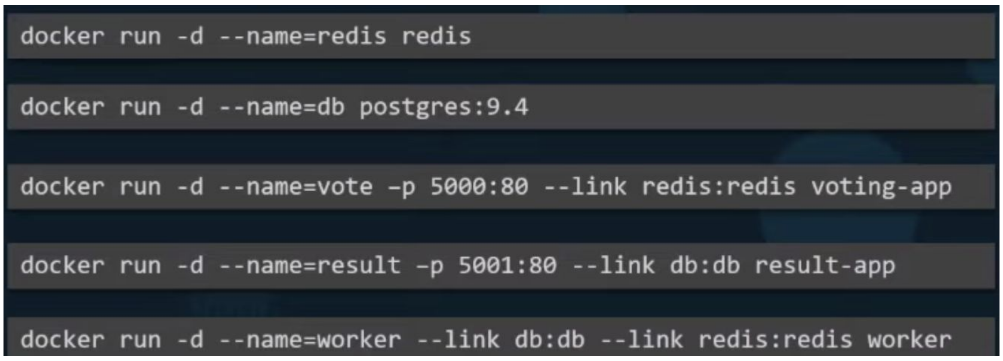
The compose files are written in YAML



```
docker run mmumshad/simple-webapp
docker run mongodb
docker run redis:alpine
docker run ansible
```

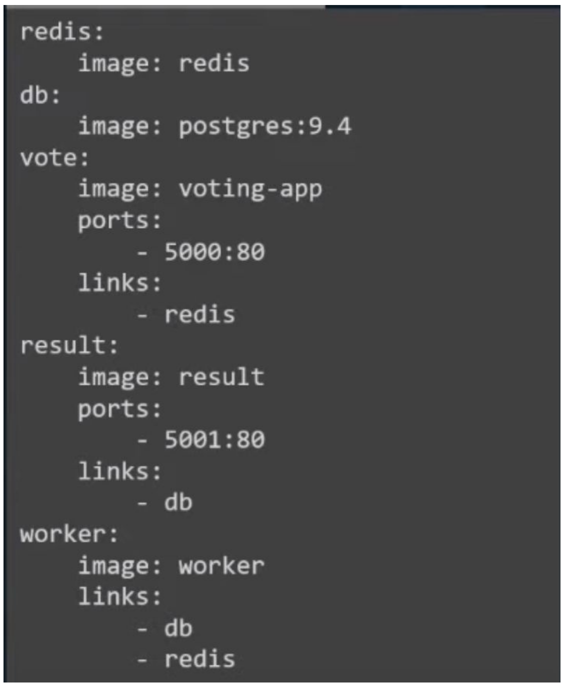
```
docker-compose.yml
services:
     web:
        image: “mmumshad/simple-webapp"
     database:
        image: “mongodb“
     messaging:
        image: "redis:alpine“
     orchestration:
        image:  “ansible“
```

Then to run all the containers, you just do “docker-compose up”
This is much easier to maintain and much faster to run.



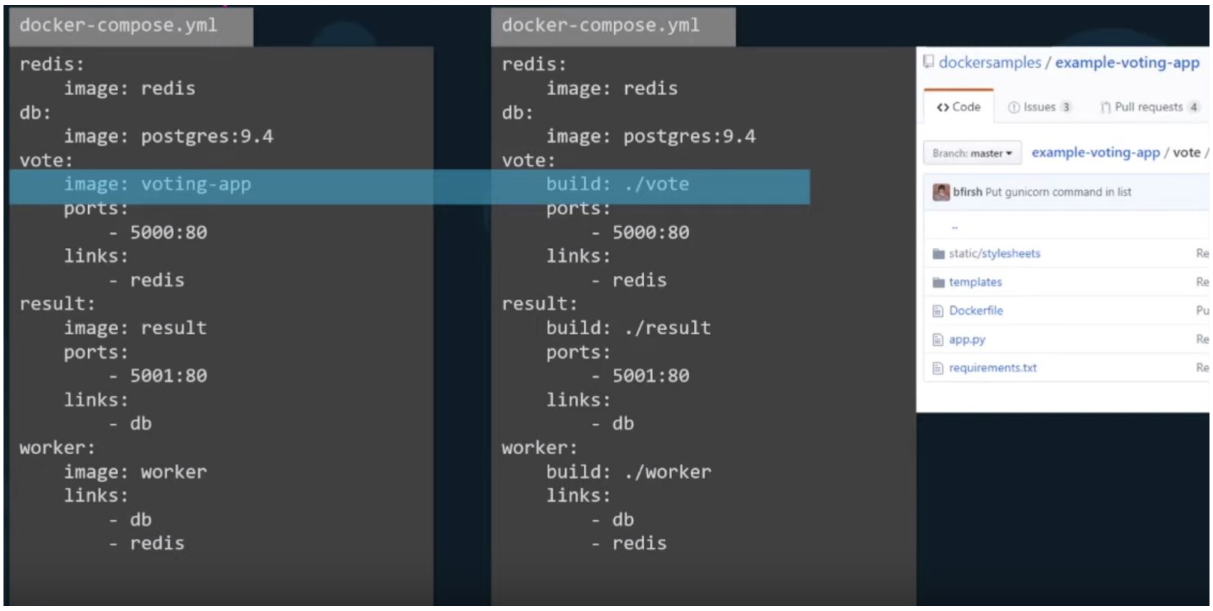
```
docker run -d --name=redis redis

docker run -d --name=db postgres:9.4

docker run -d --name=vote –p 5000:80 --link redis:redis voting-app

docker run -d --name=result –p 5001:80 --link db:db result-app

docker run -d --name=worker --link db:db --link redis:redis worker
```

The docker run commands above include a lot of tags. For example the name tags are there to make use of the dns system which is embedded into the docker host. The -p tag is the port tag and is specifying the port mappings to use. The link tag is used so that in source code, you can use the name of the service and it will link to it directly. (The link tag is deprecated and in version 2 and 3 of docker compose it is not needed). These docker run files can be translated into the docker compose file that looks like this:



```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
result:
    image: result
    ports:
        - 5001:80
    links:
        - db
worker:
    image: worker
    links:
        - db
        - redis
```

As you can see, we are specifying image tags. This will only work if the image is stored locally or is in the docker hub registry. If we have the dockerfile in a directory, we can choose to just build the images directly in our docker compose file like this…



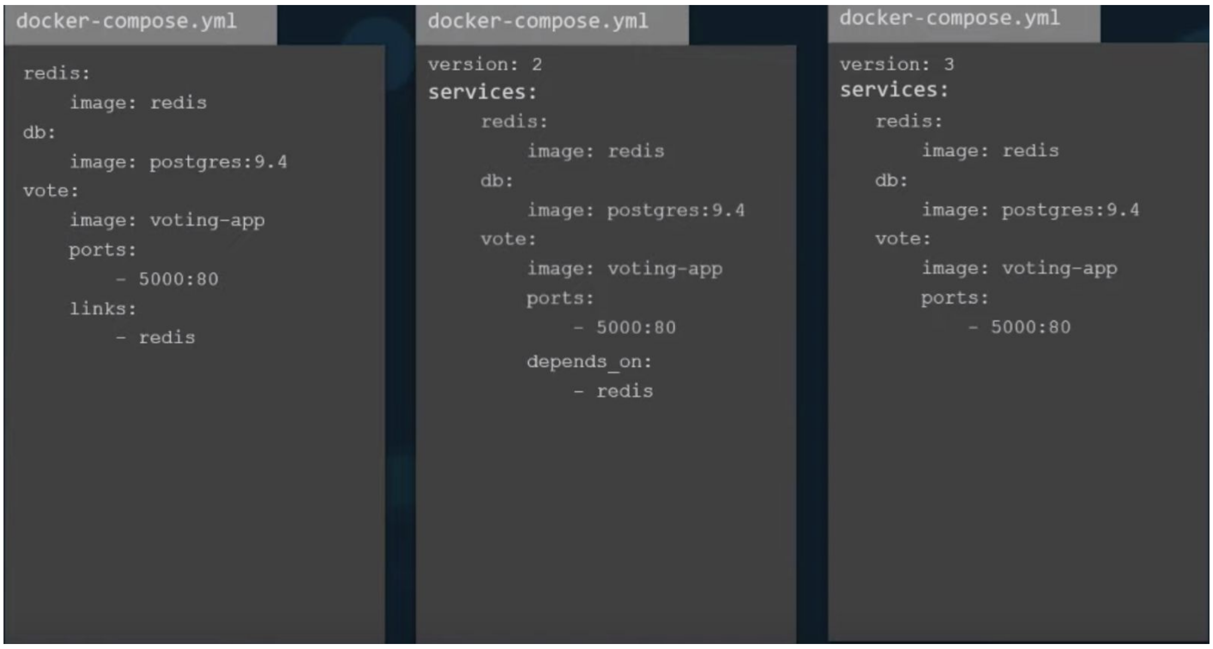
docker-compose.yml

```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
result:
    image: result
    ports:
        - 5001:80
    links:
        - db
worker:
    image: worker
    links:
        - db
        - redis
```

docker-compose.yml

```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    build: ./vote
    ports:
        - 5000:80
    links:
        - redis
result:
    build: ./result
    ports:
        - 5001:80
    links:
        - db
worker:
    build: ./worker
    links:
        - db
        - redis
```

Below is an image showing different docker compose files as they progress from version 1 to version 3:



docker-compose.yml

```
redis:
    image: redis
db:
    image: postgres:9.4
vote:
    image: voting-app
    ports:
        - 5000:80
    links:
        - redis
```

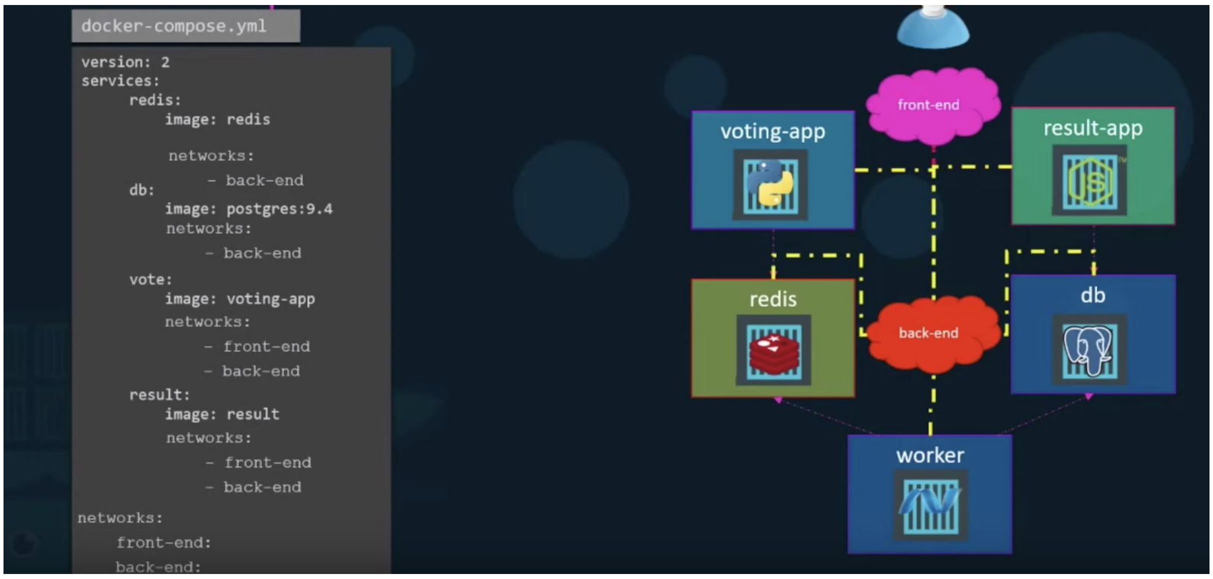
docker-compose.yml

```
version: 2
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
        depends_on:
            - redis
```

docker-compose.yml

```
version: 3
services:
    redis:
        image: redis
    db:
        image: postgres:9.4
    vote:
        image: voting-app
        ports:
            - 5000:80
```

So far we have been deploying our containers on the default bridge networks. Imagine we have designed an architecture where we would want to separate our network traffic. To do this we define our networks, then for each of the services, we define what network they will be on.



```
docker-compose.yml
version: 2
services:
     redis:
         image: redis

         networks:
             - back-end
     db:
         image: postgres:9.4
         networks:
             - back-end
     vote:
         image: voting-app
         networks:
             - front-end
             - back-end
     result:
         image: result
         networks:
             - front-end
             - back-end
networks:
     front-end:
     back-end:
```

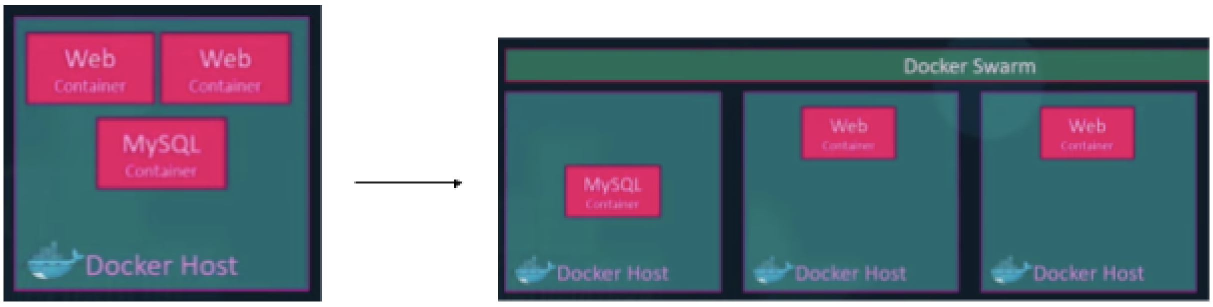
How do containers work on Mac if it is only using linux?

When you install the docker package, you are installing various packages : a virtual machine program to create virtual linux machines , docker engine and many more. So when you want to run a docker image, you are running it on a linux virtual machine because all images are linux based.

Docker orchestration

Docker Swarm

With docker swarm, you can have multiple docker hosts running your services and swarm will orchestrate everything from load balancing and availability. Transforming architectures like this into this.



To set up a docker swarm, do the following:

- Have multiple hosts with docker installed on them
- Designate one to be the swarm manager and the others as the swarm workers
- On the swarm manager, run docker swarm init this will then print out a token which needs to be copied
- Then on all of the swarm workers, run docker swarm join --token <token here>

- The workers are now referred to as nodes

Now we have a swarm, how can we use it to our advantage to run multiple web servers. We could simply run the docker run command on each node but this has a runtime of O(n) which is shit. So instead, we use the power of orchestration. On the manager node, we run :

- docker service create --replicas=3 my_web_server

This will run three instances of our web server across all the nodes in the swarm. This is good because now if a web server crashes, the swarm manager will automatically set up a new one.

Kubernetes

Kubernetes allows you to cluster together groups of hosts running Linux containers, and Kubernetes helps you easily and efficiently manage those clusters.

Joe Beda, CTO at Heptio (and one of Kubernetes' original developers, along with Craig McLuckie and Brendan Burns, at Google): "Kubernetes is an open source project that enables software teams of all sizes, from a small startup to a Fortune 100 company, to automate deploying, scaling, and managing applications on a group or cluster of server machines. These applications can include everything from internal-facing web applications like a content management system to marquee web properties like Gmail to big data processing."

Kimoon Kim, senior architect at Pepperdata: "Kubernetes is software that manages many server computers and runs a large number of programs across those computers. On Kubernetes, all programs run in containers so that they can be isolated from each other, and be easy to develop and deploy."