**Applications & machines**

Back in the day, an application would be created and run on a machine. There was a ratio of 1:1. One application, one machine.

When wondering what type of a machine to purchase for their application, developers always purchased the most powerful machine possible. There was no certainty as to how much an application might be in demand. To cover their asses, they errored on the side of too much power instead of not enough.

The result of this: many machines were greatly underutilized. Many machines only ran at a small fraction of their capacity.

**Virtual machines**

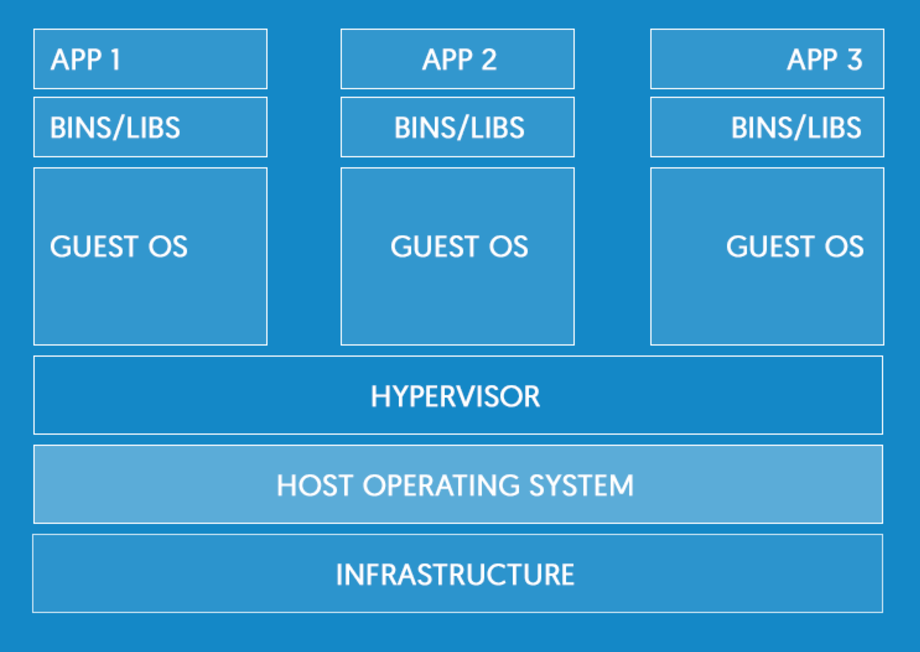
There were many machines with unused capacity.

Virtual machines found a way to use that capacity.

One physical machine could host several virtual machines.

Each virtual machine appeared just like a real machine to its user.

Here is an image from [docker's website](https://www.docker.com/what-docker) that shows the architecture of virtual machines:



There was a downside to this, though: each virtual machine had its own operating system. This created license and maintenance costs, as well as using up resources on the physical machine.

The question arose: \*\*Is there a way to divide up a physical machine, but only have one operating system on that machine?\*\*

**Containers**

Linux is built in such a way that you can create separate user spaces.

Each user space is in a separate isolated "sandbox" area. Each user space has its own file system and processes. These user spaces are all segregated and separate from each other.

Containers leverage this technology of the Linux operating system.

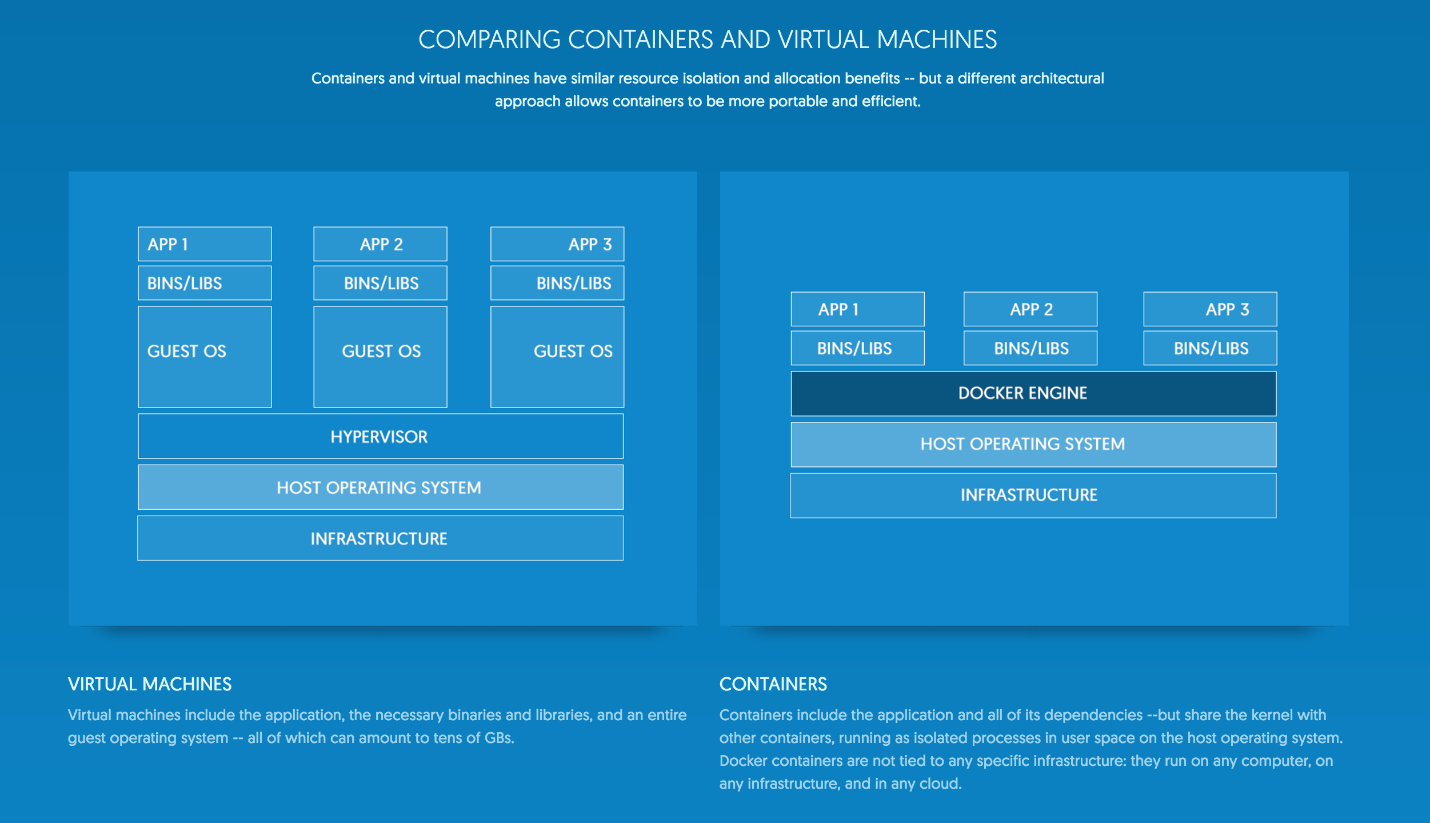
A container is just like a virtual machine WITHOUT THE OPERATING SYSTEM.

Each PHYSICAL MACHINE has ONE LINUX OPERATING SYSTEM.

Containers use the Linux OS of the physical machine.

To the user, each container appears just like a real machine.

Here is an image from [docker's website](https://www.docker.com/what-docker) that shows a comparison between virtual machines and containers:



Containers and virtual machines have similar resource isolation and allocation benefits -- but a different architectural approach allows containers to be more portable and efficient.

**Virtual machines**

Virtual machines include the application, the necessary binaries and libraries, and an entire guest operating system -- all of which can amount to tens of GBs.

**Containers**

Containers include the application and all of its dependencies --but share the kernel with other containers, running as isolated processes in user space on the host operating system. \*\*Docker containers are not tied to any specific infrastructure: they run on any computer, on any infrastructure, and in any cloud.\*\*

**Docker**

Docker is the most popular container system.

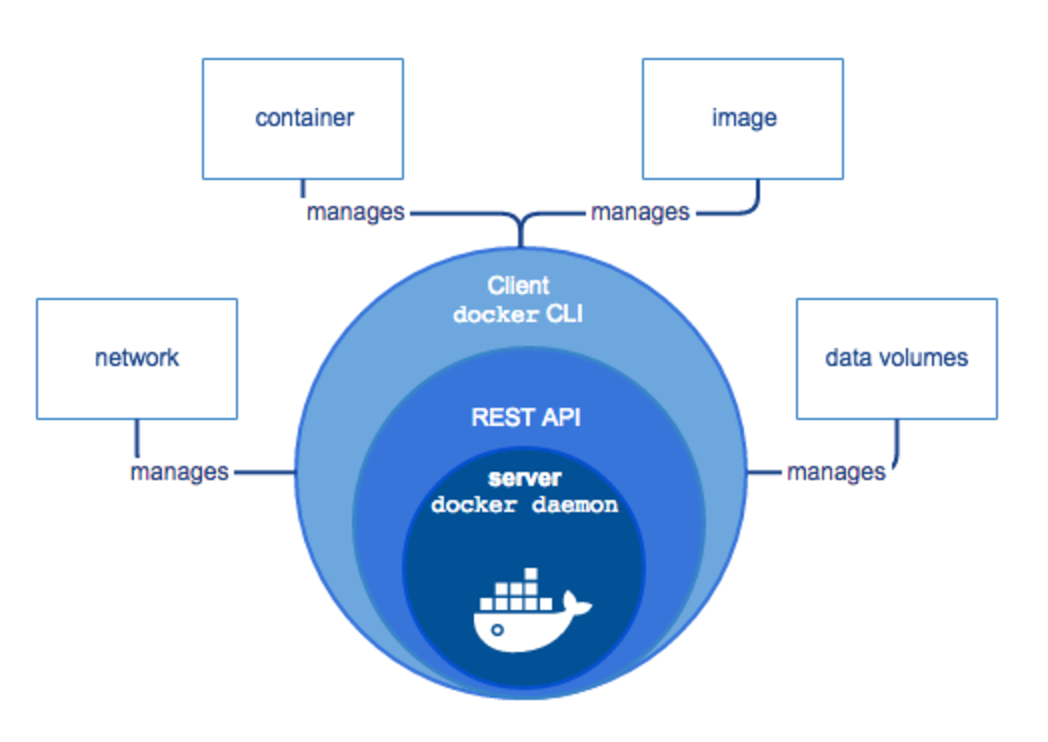
You build an \*\*IMAGE\*\* and then create as many \*\*CONTAINERS\*\* as you would like from that image.

You can then run those containers on physical machines.

Docker containers run on Linux, Mac OS, and Windows.

The official description: Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

[source](https://docs.docker.com/engine/understanding-docker/)



**Docker benefits**

Run anywhere, easily

Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.

Docker containers are based on open standards, enabling containers to run on all major Linux distributions and on Microsoft Windows -- and on top of any infrastructure.

**Secure**

Containers isolate applications from one another and the underlying infrastructure

**Lightweight**

Containers running on a single machine share the same operating system kernel; they start instantly and use less RAM than VMs.

**Accelerate developers**

Stop wasting hours setting up developer environments, spinning up new instances, and making copies of production code to run locally. With Docker, you simply take copies of your live environment and run them on any new endpoint running a Docker engine.

**Focus on microservices**

The isolation capabilities of Docker containers free developers from constraints: they can use the best language and tools for their application services without worrying about causing internal tooling conflicts.

**Collaborate**

Docker creates a common framework for developers and sysadmins to work together on distributed applications

**Docker repositories (like Docker hub)**

Store, distribute, and manage Docker images in Docker Hub with your team. Image updates, changes, and history are automatically shared across your organization.

**SHIP 7X MORE**

On average, Docker users ship 7X more software after deploying Docker in their environment. More frequent software updates provide added value to customers.

**Scale**

Docker containers spin up and down in seconds, making it easy to scale application services to satisfy peak customer demand, and then reduce running containers when demand ebbs.

**Debugging**

Docker makes it easy to identify issues, isolate the problem container, quickly roll back to make the necessary changes, and then push the updated container into production. Isolation between containers makes these changes less disruptive than in traditional software models.

**Orchestration**

Automate deployment, scaling, and management of containerized applications with \*\*Docker Swarm\*\* or \*\*Kubernetes\*\*.

**Overview**

This is what we will do:

1. install Docker

1. run a software image in a container

1. locate an interesting image on Docker Hub

1. run the image on your own machine

1. modify an image to create your own and run it

1. create a Docker Hub account and repository

1. push your image to Docker Hub for others to share

**Install**

[Install docker](https://docs.docker.com/)

Choose \*\*Docker engine / install\*\*

**Verify installation**

Check the docker version

```

docker version

```

Run a sample "hello world" application

```

docker run hello-world

```

**Images, Repository, Containers**

You use an \*\*image\*\* to make a \*\*container\*\*.

You can make an unlimited number of \*\*containers\*\* from one \*\*image\*\*.

An \*\*image\*\* is stored in a docker image \*\*repository\*\*.

\*\*Docker hub\*\* is one docker image repository you can use.

See all containers (processes) on your machine

```

docker ps -a

```

See all running containers on your machine

```

docker ps

```

We will search docker hub for an image. We will then run that image.

When you run an image, that image is copied to your local machine.

When you run an image in a container, Docker downloads the image to your computer. This local copy of the image saves you time. Docker only downloads the image again if the image’s source changes on the hub. You can, of course, delete the image yourself. You’ll learn more about that later.

**Locate whalesay**

Locate the \*\*docker/whalesay\*\* image on [Docker Hub](https://hub.docker.com/)

You can also do this to search docker hub:

```

docker search <search term>

```

Caveat: This is code. When you grab someone else's code to use with your code, make sure you know what's in it.

**Run the program**

```

docker run docker/whalesay cowsay boo

```

# See the images on your machine

```

docker images

```

# See all containers (processes) on your machine

```

docker ps -a

```

```

docker ps

```

**Reflection**

You searched for an image on Docker Hub.

You used your command line to run an image. Effectively you ran a piece of Linux software on whatever machine you're using (Windows, Mac, Linux).

You learned that running an image copies it on your computer.

**Main points**

Dockerfile --> Docker image --> Docker containers

A Dockerfile builds a docker image. From a docker image, you can run an unlimited number of containers.

Your Dockerfile must be named Dockerfile

Include stuff in your image

Everything in the folder with your Dockerfile, and all subsequent folders down the directory structure, are included in your image.

Dockerfile always starts with "**FROM**"

Specify some image unless you're creating from scratch.

You can also have a comment on the first line. Dockerfile comments are ```#``` hash symbols

**Build an image**

1. Create a Dockerfile

1. Build an image from the Dockerfile

**Create Dockerfile**

Create a directory

```

mkdir mydockerbuild

```

Enter directory

```

cd mydockerbuild

```

Create a docker build file

```

nano Dockerfile

```

Add these lines to your build file

```

FROM docker/whalesay:latest

RUN apt-get -y update && apt-get install -y fortunes

CMD /usr/games/fortune -a | cowsay

```

**FROM**

The FROM keyword tells Docker which image your image is based on.

**RUN**

The RUN statement will install the fortunes program into the image.

The whalesay image is based on Ubuntu, which uses apt-get to install packages.

These two commands refresh the list of packages available to the image and install the fortunes program into it.

The fortunes program prints out wise sayings for our whale to say.

**CMD**

The CMD statement tells the image the final command to run after its environment is set up.

This command runs fortune -a and pipes its output to the cowsay command.

**Build an image from the Dockerfile**

**Docker build**

Build the image using the docker build command.

The -t parameter gives your image a tag, so you can run it more easily later.

Don’t forget the . command, which tells the docker build command to look in the current directory for a file called Dockerfile.

```

docker build -t docker-whale .

```

The above command reads the Dockerfile in the current directory and processes its instructions one by one to build an image called \*\*docker-whale\*\* on your local machine.

**Run your new docker-whale**

See what images you have on your machine:

```

docker images

```

Run your image

```

docker run docker-whale

```

Run it again, and again, and again to see different output.

**Notes from "Docker RUN vs CMD vs ENTRYPOINT"**

[Yury Pitsishin - 02 APRIL 2016](http://goinbigdata.com/docker-run-vs-cmd-vs-entrypoint/)

Some Docker instructions look similar and cause confusion among developers who just started using Docker or do it irregularly. In this post I will explain the difference between CMD, RUN, and ENTRYPOINT on examples.

## RUN

executes command(s) in a new layer and creates a new image. E.g., it is often used for installing software packages.

## CMD

sets default command and/or parameters, which can be overwritten from command line when docker container runs.

## ENTRYPOINT

configures a container that will run as an executable.

If it doesn't make much sense or you after details, then read on.

**Docker images and layers**

When Docker runs a container, it runs an image inside it. This image is usually built by executing Docker instructions, which add layers on top of existing image or OS distribution. OS distribution is the initial image and every added layer creates a new image.

**Shell and Exec forms**

All three instructions (RUN, CMD and ENTRYPOINT) can be specified in shell form or exec form. Let's get familiar with these forms first, because the forms usually cause more confusion than instructions themselves.

Shell form

```<instruction> <command>```

Examples:

```

RUN apt-get install python3

CMD echo "Hello world"

ENTRYPOINT echo "Hello world"

```

When instruction is executed in shell form it calls /bin/sh -c ```<command>``` under the hood and normal shell processing happens. For example, the following snippet in Dockerfile

```

ENV name John Dow

ENTRYPOINT echo "Hello, $name"

```

```docker run -it <image>```

will produce

```

Hello, John Dow

```

Note that variable name is replaced with its value.

**Exec form**

This is the preferred form for CMD and ENTRYPOINT instructions.

```<instruction> ["executable", "param1", "param2", ...]```

Examples

```

RUN ["apt-get", "install", "python3"]

CMD ["/bin/echo", "Hello world"]

ENTRYPOINT ["/bin/echo", "Hello world"]

```

When instruction is executed in exec form it calls executable directly, and \*\*shell processing does not happen.\*\* For example, the following snippet in Dockerfile

```

ENV name John Dow

ENTRYPOINT ["/bin/echo", "Hello, $name"]

```

```docker run -it <image>```

will produce output

```

Hello, $name

```

Note that variable name is not substituted.

How to run bash?

If you need to run bash (or any other interpreter but sh), use exec form with /bin/bash as executable. \*\*In this case, normal shell processing will take place.\*\* For example, the following snippet in Dockerfile

```

ENV name John Dow

ENTRYPOINT ["/bin/bash", "-c", "echo Hello, $name"]

```

```

docker run -it <image>

```

will produce output

```

Hello, John Dow

```

**RUN**

RUN allows you to install your application and packages. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

RUN has two forms:

```

RUN <command> (shell form)

RUN ["executable", "param1", "param2"] (exec form)

```

(The forms are described in detail in Shell and Exec forms section above.)

A good illustration of RUN instruction would be to install multiple version control systems packages:

```

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion

```

\*\*Note that apt-get update and apt-get install are executed in a single RUN instruction. This is done to make sure that the latest packages will be installed. If apt-get install were in a separate RUN instruction, then it would reuse a layer added by apt-get update, which could had been created a long time ago.\*\*

**CMD**

CMD instruction allows you to \*\*set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.\*\*

CMD has three forms:

```

CMD ["executable","param1","param2"] (exec form, preferred)

CMD ["param1","param2"] (sets additional default parameters for ENTRYPOINT in exec form)

CMD command param1 param2 (shell form)

```

Again, the first and third forms were explained in Shell and Exec forms section. The second one is used together with ENTRYPOINT instruction in exec form. It sets default parameters that will be added after ENTRYPOINT parameters if container runs without command line arguments. See ENTRYPOINT for example.

Let's have a look how CMD instruction works. The following snippet in Dockerfile

```

CMD echo "Hello world"

```

when container runs as ```docker run -it <image>``` will produce output

```

Hello world

```

but when container runs with a command, e.g., ```docker run -it <image> /bin/bash```, CMD is ignored and bash interpreter runs instead:

```

root@7de4bed89922:/#

```

**ENTRYPOINT**

ENTRYPOINT instruction allows you to \*\*configure a container that will run as an executable.\*\* It looks similar to CMD, because it also allows you to specify a command with parameters. The difference is \*\*ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters.\*\* (There is a way to ignore ENTTRYPOINT, but it is unlikely that you will do it.)

ENTRYPOINT has two forms:

```

ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)

ENTRYPOINT command param1 param2 (shell form)

```

Be very careful when choosing ENTRYPOINT form, because forms behavior differs significantly.

**Exec form**

Exec form of ENTRYPOINT allows you to set commands and parameters and then use either form of CMD to set additional parameters that are more likely to be changed. ENTRYPOINT arguments are always used, while CMD ones can be overwritten by command line arguments provided when Docker container runs. For example, the following snippet in Dockerfile

```

ENTRYPOINT ["/bin/echo", "Hello"]

CMD ["world"]

```

when container runs as ```docker run -it <image>``` will produce output

```

Hello world

```

but when container runs as ```docker run -it <image> John``` will result in

```

Hello John

```

**Shell form**

Shell form of ENTRYPOINT ignores any CMD or docker run command line arguments.

**The bottom line**

Use RUN instructions to build your image by adding layers on top of initial image.

Prefer ENTRYPOINT to CMD when building executable Docker image and you need a command always to be executed. Additionally use CMD if you need to provide extra default arguments that could be overwritten from command line when docker container runs.

Choose CMD if you need to provide a default command and/or arguments that can be overwritten from command line when docker container runs.

**Launching a container running Curl**

Build an image

1. Create a Dockerfile

1. Build an image from the Dockerfile

Create Dockerfile

Create a directory

```

mkdir mydockerbuild2

```

Enter directory

```

cd mydockerbuild2

```

Create a docker build file

```

nano Dockerfile

```

Add these lines to your build file

```

# Creates an ubuntu image with curl installed

FROM ubuntu:latest

RUN apt-get -y update && apt-get install -y curl

```

**Build an image from the Dockerfile**

Docker build

Build the image using the docker build command.

The -t parameter gives your image a tag, so you can run it more easily later.

Don’t forget the ```.``` command, which tells the docker build command to look in the current directory for a file called Dockerfile.

```

docker build -t <give it an image name> .

```

The above command reads the Dockerfile in the current directory and processes its instructions one by one to build an image called <whatever name you gave it> on your local machine.

**Run your new image**

See what images you have on your machine:

```

docker images

```

Run your image

Mac / Linux / maybe Windows

```

MAC / LINUX: docker run -it <image name> /bin/bash

```

Windows: if you get this error ```the input device is not a TTY. If you are using mintty, try prefixing the command with "winpty" ``` use the below code

```

WINDOWS: winpty docker run -it <image name> bash

```

**run a curl command**

```

curl --head www.google.com

```

**Exit**

Exit your image by typing ```exit```

**Running a Go web App in a Docker Container**

**Create Go app**

Create a directory

Create main.go

Create code for a "hello world" web app

**Create .dockerignore file**

If you want to ignore any files or directories in your build, add a .dockerignore file

```

\*.md

```

**Create Dockerfile**

The Dockerfile must be named Dockerfile.

The Dockerfile will include EVERYTHING in the current directory, and descendent directories, in the image which is built (unless told to ignore something by the .dockerignore file)

The Dockerfile may start with a comment

```

# Yo, this is my Dockerfile, Yo

```

The Dockerfile must have FROM as the first instruction

The FROM says what image your are building this image from.

In most cases, you will start with an image to build your image.

You then later new images on top of your starting FROM image, and that all finally becomes your finished image.

Images are made out of images layered on top of images.

You can add in a MAINTAINER instruction if you'd like and say who built this image

We are going to build our image FROM a golang image so ...

Go to docker hub

Search for golang

Find the golang image you want

We will use: golang:1.8-onbuild

So our Dockerfile will be

```

# Some comment

FROM golang:1.8-onbuild

MAINTAINER youremail@gmail.com

```

**Now build our image**

```

docker build -t my-app .

```

-t means "tag" or "give it a name"

The name we gave it is "my-app"

The dot "." means the code for this image is in this current directory

Make sure you are in the correct directory when you run this

**Now create a container from your image and run it**

```

docker run -d -p 80:80 my-app

```

-d means run this detached, as a daemon, eg, not dependent on the terminal session

-p means map ports; mapping ```<host machine port>:<to docker container port>```

**Verify it's running**

Go to your browser and see if it's running – http://localhost

**Pushing and Pulling to Docker Hub**

See what images you currently have

```

docker images

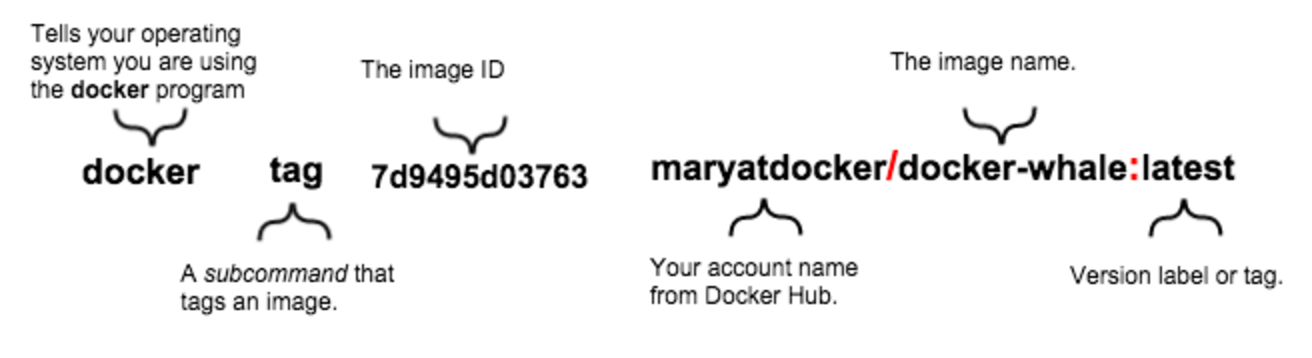
```

Add a docker tag

```

docker tag <image ID> <docker hub username>/<image name>:<version label or tag>

```



Give terminal your docker hub credentials

Create a docker hub account, if necessary

```

docker login

```

Push your image

```

docker push <docker hub username>/<image name>

```

Look at your docker hub account

Your image should be there.

See images on your machine

```

docker images

```

Remove all "docker-whale" images

```

docker rmi -f <image ID or image name>

```

-f is "with force"

See docker help

```

docker --help

docker <COMMAND> --help

docker rmi --help

```

Pull from your repo

run it this way if it's our go web app from previous step

```

docker run -d -p 80:80 <yourusername>/<app-name>

```

for some other image you might use this

```

docker run <yourusername>/<app-name>

```

Stop your container from running

```

docker ps

docker stop <container id>

docker ps

docker images

```

Look at what you've done

1. gained an understanding of containers & Docker

1. gained an understanding of Docker, images, containers, image repositories

1. installed Docker

1. run a software image in a container

1. located an interesting image on Docker Hub

1. run the image on your own machine

1. modified an image to create your own and run it

1. built a docker image with a Go web app

1. run a docker container with a Go web app

1. create a Docker Hub account and repository

1. pushed your image to Docker Hub for others to share

**Go, Docker and AWS**

**Create a new EC2 instance**

Use the Amazon Linux AMI

[EC2 Docker Instructions](http://docs.aws.amazon.com/AmazonECS/latest/developerguide/docker-basics.html)

change the permissions on your pem if you created a new one

```

sudo chmod 400 your.pem

```

**Connect to your machine**

ssh into your instance

```

ssh -i /path/to/[your].pem ec2-user@[public-DNS]

```

**Update the installed packages and package cache on your instance**.

```

sudo yum update -y

```

**Install Docker.**

```

sudo yum install -y docker

```

**Start the Docker service.**

```

sudo service docker start

```

**Add the ec2-user to the docker group so you can execute Docker commands without using sudo.**

```

sudo usermod -a -G docker ec2-user

```

Log out and log back in again to pick up the new docker group permissions.

Verify that the ec2-user can run Docker commands without sudo.

```

docker info

```

In some cases, you may need to reboot your instance to provide permissions for the ec2-user to access the Docker daemon. Try rebooting your instance if you see the following error:

```

Cannot connect to the Docker daemon. Is the docker daemon running on this host?

```

Run the Go app from the previous example

```

docker run -d -p 80:80 toddmcleod/golang-hello-world

```

Check that it's running

```

docker ps

```

View the web app from a browser

```

Use the IP address of your instance

```