INTRODUCTION TO DOCKERS

Dockers:

* Run on Linux
* It is a client program
* It builds containers from code
* It distributes the containers through the internet

Container:

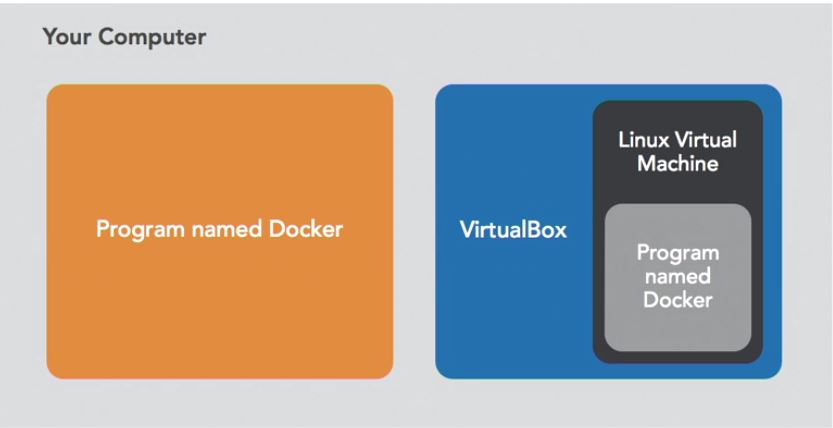
* Isolated,
* portable,
* self-contained unit of software,
* contains everything to run the code (Code, Configs, Processes, Networking, dependencies)
* Include the enough of the OS to run the code.

Community

* People built image that other can use.

Installation:

* Requires a Linux server (Bare Metal or VM)
* Docker Toolbox will install a program called Docker, and a program called Virtual Box
* Docker Toolbox will install a program called Docker Machine (it manages a Linux Virtual Machine on your computer)
* Inside that Linux Virtual Machine is the server side of the Docker program. (So when you type Docker at your Command Prompt, it will send that command into the Linux Virtual Machine over the network and to the Docker server running there)



Run Docker

* Run “Docker Quickstart Terminal”

This will start the “Docker Machine”

* $ docker run hello-world

It send a command to the “Docker Machine”

* $ docker-machine

Show help for available commands

Install Docker on Mac:

* Download and install “Docker toolbox” for Mac
* Start the “Docker Quickstart Terminal”

It creates a VM with ubuntu

* $ docker info

Show if docker status

* $ docker run -ti ubuntu bash

Run the shell on the “Docker Machine”

Install Docker on Windows:

* Enable Virtualization is enable on Windows (Hardware virtualization support)

See Task Manager\Performance Tab\Virtualization

* Download and install “Docker toolbox” for Windows

It installs “Git for Windows” (Utility that have bash, GUI, shell integration, etc)

* Start the “Docker Quickstart Terminal”

It creates a VM with ubuntu

* $ docker info

Show if docker status

* $ docker run –net=host -ti ubuntu:14.04 bash

Run the bash (terminal) of Ubuntu 14.04 on the “Docker Machine”

It will download the image if it does not exist.

* $ docker run -ti ubuntu bash

Run the bash (terminal shell) of the lastest ver of Ubuntu on the “Docker Machine”

It will download the image if it does not exist

Install Docker on Linux

* No need to setup Virtual Machine. You can run the Docker directly from your Linux
* Depending on your version of Linux, follow the instruction from https://docs.docker.com/engine/installation/linux/
* Allow secured connection to Docker’s archive via https
* Trust packages signed by Docker
* Allow app to download soft directly from Docker
* Install Docker
* Start docker

$ Sudo service docker start

* Add Docker to the sudo group

Sudo usermod -aG docker username

Docker Flow – Images to Containers

Image – a file that make ups the minimum for an specific app to run

$ docker images

Show a list available (already downloaded) images

$ docker run -ti “ImageName” command

It creates a container from an image. Any modification are not saved on the image, just on the

contrainer.

-ti: terminal interacted

“ImageName”: select an existing image or a new one to be downloaded

Command: what to run on the container

F.E. docker run -ti ubuntu:lastest bash

$ docker ps

Shows running images

$ docker ps -a

Shows all containers

$ docker ps -l

Shows the last container

$ docker commit “ContainerID”

It converts the container on a new image. It does not overwrite the original image

Changes in a container can be lost if not converted to an image

It returns a ImageID as a SHA256 key

“ContainerID”: it is the container name

$ docker tag “ImageName:”ImageTag” “NewImageName”:”NewImageTag”

$ docker tag “ImageID” “NewImageName”:”NewImageTag”

F.E. docker tag 258078f8a6cc0dae715c9f6a2b07e176dc7d17f13cc194a22e79b536a168a530 my-

image

Rename an image

$ docker commit “ContainerID” “ImageName”

Same as the 2 previous steps

$ docker run “Image Name or ImageID” command

F.E. docker run -ti my-image bash

Run Process in Containers

$ docker run –rm -ti ubuntu sleep 5

--rm: it deletes the container when you exit it.

Sleep 5: it exits after 5 seconds

$ docker run -ti ubuntu bash -c “sleep 3; echo all done”

-c: pass the instructions to bash

$ docker run -d -ti ubuntu bash

-d: run the container on the background and keep it Up

$ docker attach “NAMES of container”

Attach: it brings up a container running on the background

root@xxxxx:/# exit

exits the container and stop it

root@xxxxx:/# <Ctrl>-P <Ctrl>-Q

exits the container and keeps it up and running

$ docker exec -ti “NAME of container” command

run another process to a running container

Manager containers

$ docker logs “NAME of container”

Useful if a container stops, to see errors of command ran

$ docker kill “NAME of container”

It exits or close a running (up) container

$ docker rm “NAME of container”

Deletes a container

$ docker run -m “Memory Max-allowed” “ImageName” command

Limit the amount of memory used

$ docker run --cpu-shares X --cpu-quota “ImageName” command

Recommendations:

* Don’t let containers fetch dependencies when start. Keep dependencies inside the container.
* Don’t store important data on unnamed stopped container

Network between Containers:

Docker provide a private network between the containers

You can expose ports or linking containers

$ docker run -ti -p 45678:45679 --name “NAME of container”/”Protocol” “ImageName” command

Opens an inside port of the container that links to an outside port of the container.

-p InPort/OutPort

/“Protocol”: /udp or /tcp

It opens a port number OutPort on the Docker-Machine and forward it to the container

All containers share the Docker-Machine IP

$ docker-machine ip

Show the IP of the docker-machine (linux VM)

From a network PC, nc ip:outport

Ip: IP of the docker-machine

$ docker port “NAME of container”

Shows ports on the container

IntPort 🡪 OutPort

Link Containers

1. Connect 2 containers by their ports

$ docker-machine ip

192.168.99.100

Get the IP of the docker-machine. The same IPs of the containers.

$ docker run -ti --p 1234:1234 ubuntu:14.04 bash -c “nc -lp 1234”

Creates a container and make it listen to port 1234

$ docker run -ti –rm ubuntu:14.04 bash -c “nc 192.168.99.100 1234 hello”

1. Link containers

Data goes directly from client container to the server container.

All their ports are connected, from the client to the server

Server must run on the same machine

$ docker run -ti --rm --name server ubuntu:14.04 bash -c “nc -lp 1234”

$ docker run --rm -ti --link server --name client ubuntu:14.04 bash -c “nc server 1234”

Docker add the host of the server and client names on the containers

Each container is assigned an private IP

root@xxx:/# cat /etc/hosts

Links can be broken when containers restart, because their private IP changes

Dynamic and legacy linking

Dynamic Linking

To use links in such a way that they don't break when services come and go and restart,

By creating a network, you define a private network with built-in nameservers that fix the

change of IPs during containers’ restart. So links don’t break.

$ docker network create “Network Name”

$ docker run --rm -ti --net=”Network Name” --name server ubuntu:14.04 bash -c “nc -lp 1234”

$ docker run -rm -ti --link server --net=”Network Name” --name client ubuntu 14.04 bash -c “nc

server 1234”

Now even if any of the container is stopped, when restarted the link comes up.

Legacy Linking:

Similar to Dynamic Linking, but it is set up by setting environment variable inside the linked

containers for host and port

You need to bind the port (service) to 0.0.0.0, so the service listen to other containers.

You can setup Docker to only allow connection from connection from the host.

Images

$ docker images

List downloaded images

$ docker commint “ContainerID” “NameOf Image”:”TagOfImage”

Save an container as a image and gives it a name and a tag

$ docker pull Registry.example.com:port/organization/image-name:version-tag

Download an image, but does create a container

$ docker push Registry.example.com:port/organization/image-name:version-tag

Upload an image

$ docker rmi image-name:tag

Remove an image from the “Docker-Machine”

$ docker rmi “ImageID”

Same as above

Volumes

Volumes = virtual disk to store and share data between containers.

Type of Volumes:

1. Persistent

When the containers is stopped or removed, the volume stays

1. Ephemeral

When no container uses it, it is removed.

Sharing data between the Linux host and the containers running on it. The volume is persistent.

$ docker-machine ssh

docker@default:~$ mkdir example

docker@default:~$ docker run -ti -v /home/docker/example:/share-folder ubuntu bash

-c “cd/shared-folder; touch my-data; exit”

docker@default:~$ ls example/

my-data will show

Sharing data between containers. The volume is ephemeral.

$ docker run -ti -v /shared-data ubuntu bash -c “echo hello > /shared-data/data-file”

$ docker run -ti --volumes-from “Name of 1st Container” ubuntu bash -c “ls /shared-data/”

$ docker run -ti --volumes-from “Name of 2nd Container” ubuntu bash -c “ls /shared-data/”

A third container can access the share

Docker registries:

Registries manage and distribute images. It is like a library of images

Finding Images to Use:

$ docker search ubuntu

List images from the internet about ubuntu

The same can be done by going to <https://hub.docker.com> (this give more info)

$ docker login

Let access your docker registry on <https://hub.docker.com>

You can use pull and push

Building Docker Images

Dockerfiles:

It is a small script that describe how to build an image

$ docker built -t “name-of-result” .

It builds an image and save it on the docker registry.

Each line of the script takes the image from the previous line an makes a container, makes some

changes in the container and save it as a new image

to another image. It does not modify the previous line’s image.

When you re-run a dockerfile with some changes, only the changed lines are executed, saving

time.

Each line is a isolated from the rest of the lines. You can’t reference other lines or other lines’

images.

Environment variables are accessible from all lines.

Building a Basic dockerfile

* $ nano dockerfile1

FROM Image-Name

//\* create an image from Image-Name

RUN echo “Tex1”

Creates a container from thee created image and run echo “text1”

Creates a new image

Then removes the container

CMD echo “Text2”

Creates a container from the image created on the previous line, then it

does a command.

Creates a new image

Then removes the container

* $ docker built -f dockerfile1-t ImageName1 .

-t “ImageName1”: the name of the image that dockerfile1 will create

Creates an image from the script dockerfile1. It reads the local folder to find the

first dockerfile

* $ docker run ImageName1

Runs the image create by the dockerfile

The potential of dockerfiles exists on the ability to start FROM images created from other dockerfiles, automatizing processes.

Dockerfile Syntax

More information in https://docs.docker.com/engine/reference/builder/

FROM ImageName1: The FROM line must be the first of the dockerfile. It specifies the starting image.

MAINTAINER: XXX: list who is responsible for creating the end image.

RUN command1: it executes any commands in a new layer on top of the current image and commit the results.

It has 2 forms:

1. Shell form:

RUN command

RUN /bib/bash -c 'source $HOME/.bashrc; echo $HOME'

1. Exec form:

RUN [“executable”, “param1”, “param2”]

RUN ["/bin/bash", "-c", "echo hello"]

the exec form does not invoke a command shell.

RUN unzip install.zip /opt/install

ADD filename1 /path/filename1

Adds (copy) a file

ADD filename2.tar.gz /install/

Copies the content (decompress) of an archive

ADD [https://example.com/filename3.rpm /project/](https://example.com/filename3.rpm%20/project/)

Downloads a file to a directory

ENV DB\_HOST=db.example.com

Set environment variables, which are available during the build (run of the dockerfile) and also available when running the end image.

ENTRUPOINT

Command line arguments to docker run <image> will be appended after all elements in an exec form ENTRYPOINT, and will override all elements specified using CMD. This allows arguments to be passed to the entry point, i.e., docker run <image> -d will pass the -d argument to the entry point.

Only the last ENTRYPOINT instruction in the Dockerfile will have an effect.

It has 2 forms:

1. Shell form:

ENTRYPOINT command param1 param2

1. Exec form:

ENTRYPOINT ["executable", "param1", "param2"]

CMD

It provides defaults for an executing container. It does not execute anything at build time, but specifies the intended command for the image.

There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well.

When used in the shell or exec formats, the CMD instruction sets the command to be executed when running the image.

It has 3 forms:

1. Shell form:

CMD command param1 param2

1. Exec form:

CMD ["executable","param1","param2"]

the exec form does not invoke a command shell.

1. As default parameters to ENTRYPOINT:

CMD ["param1","param2"]

EXPOSE

EXPOSE <port> [<port>...]

Makes the container to listen on the specified network ports at runtime.

It does not make the ports of the container accessible to the host. You must use the

docker run dockerfile1-p 80

VOLUME

VOLUME ["/data"]

It creates a mount point (shares) with the specified name and marks it as holding

externally mounted volumes from native host or other containers.

VOLUME [“/host/path/” “/container/path/”]

WORKDIR

WORKDIR /install/

It sets the directory that the container starts in.

It is like cd.

USER

USER Arthur

It sets which user the container will run as.

Recommendations:

1. Include installers in your project
2. Have a canonical build system that builds your images from scratch. Don’t save images under the same name, keep
3. Use small images. Begin with small OS.
4. Build the images that you will share using Dockersfiles. They keep track of the changes.

Docker

It is written in Go language

It manages kernel features

It uses cgroups to contain process (provides container isolation)

It uses namespaces to contain networks

It uses copy-on-write filesystem to build images

Parts of the Docker program

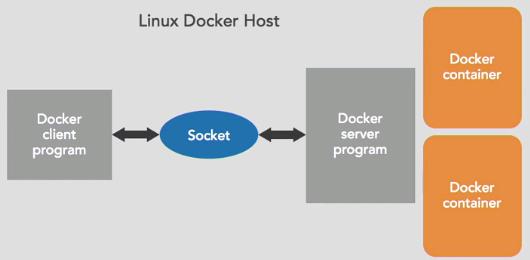
They can be run on the same PC, locally, or even remotely.

1. Client

The client sends commands to the server

1. Server

The server built images.



Docker Networking

It uses bridges to create virtual networks inside your computer. When you create a private

network in Docker, it creates a bridge. These function like software switches.

Bridges control the Ethernet layer.

$ docker run --net=host Ubuntu

--net==host give the container full network access to the host networking stack.

$ docker network create my-new-network

Creates a network bridge (virtual network)

Any container that uses a different network than “host” is insulated w/o access to the

host network

Docker Storage

Containers contains a layer of data that when applied (COW) to the source image (run) provides

the results. So the container is a supplement of the image.

COW: Copy on write.

Advantages of using Dockers:

It makes scripting distributed system much easier.

Saving docker images

$ docker saves -o my-images.tar.gz debian:sid busybox Ubuntu

Saves 3 images (debian:sid busybox Ubuntu) locally

-o : output

$ docker load -I my\_images.tar.gz

Loads images into the docker-machine

Orchestration

Facilitates the user of multiple containers that are interrelated.

It can start and restart containers

It provides service-discovery, allowing container to find each other.

It provides resource allocation.

Docker Compose:

It is an orchestration tool used for single machine coordination

It can bring all your containers, volumes, networks, etc. with one command.

Kurbernetes:

It is an orchestration tool used for single machine coordination

It has container that run programs

It has Pods that group container together, and provide them with resources.

It has Services that make pods available to others

It has Labels that are used for advanced service discovery.

It has a flexible overlay networking, that provides stability to virtual networks against changes

on the physical network infrastructure.

It runs on hardware, cloud or both.

Amazon ECS (EC2 Container Service)

It has “Task definition”, which are instructions that define who containers will run together.

Similar to Kubernetes Pods.

It runs Task

It has Service that exposes the Task to the Internet and ensures that the task runs all the time,

even if it has to split the task to several host)

It has a repo

Docker container management

$ docker run -ti “IMAGE NAME” “NEW CONTAINER NAME”

It creates a container and run it on the terminal

<Ctll>-P <Ctrl>-Q detach to the container. It leaves the container and console get back to the docker

host terminal

$ docker ps

Shows running containers with their IDs

$ docker commit “CONTAINER ID” “NEW IMAGE NAME”

$ docker attach “CONTAINER ID”

It returns the console (iterative) to a running container