Docker 19.03.8

Docker is an open-source project that automates the deployment of applications inside software containers. These application containers are similar to lightweight virtual machines, as they can be run in isolation to each other and the running host.

Introduction

- Docker requires features present in recent Linux kernels to function properly, therefore on Mac OSX and Windows host a virtual machine running Linux is required for docker to operate properly.
- Currently the main method of installing and setting up this virtual machine is via Docker Toolbox that is using VirtualBox internally, but there are plans to integrate this functionality into docker itself, using the native virtualization features of the operating system.
- On Linux systems docker run natively on the host itself.
- Docker containers are stateless. So, if you use a Containerized app, then you will lose all your saved Data once you restart the container

Installation

- Since version 1.12 you don't need to have a separate VM to be installed, as Docker can use the native Hypervisor framework functionality of OSX to start up a small Linux machine to act as backend.

To install docker follow the following steps:

1. Go to Docker for Mac/Windows

2. Download and run the installer.

Continue through installer with default options and enter your account credentials when requested.

sudo apt install docker.io

Prerequisites

- Docker only works on a 64-bit installation of Linux.
- Docker requires Linux kernel version 3.10 or higher (Except for Ubuntu Precise 12.04, which requires version 3.13 or higher).
- Check current kernel version with the command uname -r. You need to update your Ubuntu Precise (12.04 LTS) kernel by scrolling further down.
- Log into your machine as a user with sudo or root privileges. Open a terminal window.
- Update package information, ensure that APT works with the https method, and that CA certificates are installed.
- docker --version

Installation

- sudo apt-get update
- sudo apt-get install apt-transport-https ca-certificates curl software-properties-common
- curl --version
- curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
- sudo apt-key fingerprint 0EBFCD88
- echo "" | sudo tee /etc/apt/sources.list.d/docker.list
- sudo apt-get update
- sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual

Installation

- sudo apt-get install linux-image-generic-lts-trusty
- sudo reboot
- sudo service docker start
- sudo docker run hello-world
- docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -d mysql:tag
- docker container run -d --name cassandra -p 9042:9042 cassandra:latest
- ./mvnw spring-boot:build-image

Manage Docker as a non-root use

If you don’t want to use sudo when you use the docker command, create a Unix group called docker and add users to it. When the docker daemon starts, it makes the ownership of the Unix socket read/writable by the docker group.

To create the docker group and add your user:

1. Log into Ubuntu as a user with sudo privileges.

2. Create the docker group with the command sudo groupadd docker.

Add your user to the docker group. $ sudo usermod -aG docker $USER

4. Log out and log back in so that your group membership is re-evaluated. Verify that you can docker commands without sudo permission.

$ docker run hello-world

Docker Troubleshooting

sudo apt-get install docker-engine

sudo service docker start

sudo docker run hello-world

Adding a .dockerignore file to the build directory is a good practice. Its syntax is similar to .gitignore files and will make sure only wanted files and directories are uploaded as the context of the build.

Create a docker container in Google Cloud

You can use docker, without using docker daemon (engine), by using cloud providers. You should have a gcloud (Google Cloud util), that connected to your account

```
docker-machine create --driver google --google-project ‘your-project-name’ google-machine-type f1-large fm02
```

This example will create a new instance, in your Google Cloud console. Using machine time f1-large

Building images

sudo docker pull mysql/mysql-server:latest

sudo docker images

sudo docker run --name=mysqldemo -d mysql/mysql-server:latest

docker ps

Sudo apt-get install mysql-client

sudo docker logs mysqldemo

Sample Dockerfile

```
FROM ubuntu
RUN apt update
RUN apt -y install default-jdk
WORKDIR /home/myapp
COPY pom.xml ./
COPY . ./
COPY HelloWorld.java /home/myapp
RUN javac HelloWorld.java
RUN java HelloWorld
```

Building an image from a Dockerfile

you can build an image from it using docker build. The basic form of this command is:

docker build -t image-name path

If your Dockerfile isn't named Dockerfile, you can use the -f flag to give the name of the Dockerfile to build.

docker build -t image-name -f Dockerfile2 .

To build an image named dockerbuild-example:1.0.0 from a Dockerfile in the current working directory:

$ ls

Dockerfile Dockerfile2

$ docker build -t dockerbuild-example:1.0.0 .

$ docker build -t dockerbuild-example-2:1.0.0 -f Dockerfile2 .

A simple Dockerfile

```
FROM node:5
```

The FROM directive specifies an image to start from. Any valid image reference may be used.

```
WORKDIR /usr/src/app
```

The WORKDIR directive sets the current working directory inside the container, equivalent to running `cd` inside the container. (Note: RUN cd will not change the current working directory.)

```
RUN npm install cowsay knock-knock-jokes
```

RUN executes the given command inside the container.

```
COPY cowsay-knockknock.js ./
```

COPY copies the file or directory specified in the first argument from the build context (the path passed to docker build path) to the location in the container specified by the second argument.

```
CMD node cowsay-knockknock.js
```

CMD specifies a command to execute when the image is run and no command is given. It can be overridden by passing a command to docker run.

Difference between ENTRYPOINT and CMD

There are two Dockerfile directives to specify what command to run by default in built images. If you only specify **CMD** then docker will run that command using the default **ENTRYPOINT**, which is /bin/sh -c. You can override either or both the entrypoint and/or the command when you start up the built image. If you specify both, then the ENTRYPOINT specifies the executable of your container process, and CMD will be supplied as the parameters of that executable.

```
FROM ubuntu:16.04

CMD ["/bin/date"]
```

Difference between ENTRYPOINT and CMD

Then you are using the default ENTRYPOINT directive of /bin/sh -c, and running /bin/date with that default entrypoint. The command of your container process will be /bin/sh -c /bin/date. Once you run this image then it will by default print out the current date

```
$ docker build -t test .
```

```
$ docker run test
```

Tue Jul 19 10:37:43 UTC 2016

Difference between ENTRYPOINT and CMD

You can override CMD on the command line, in which case it will run the command you have

specified.

```
$ docker run test /bin/hostname
```

bf0274ec8820

If you specify an ENTRYPOINT directive, Docker will use that executable, and the CMD directive specifies the default parameter(s) of the command. So if your Dockerfile contains:

```
FROM ubuntu:16.04

ENTRYPOINT ["/bin/echo"]

CMD ["Hello"]
```

Difference between ENTRYPOINT and CMD

Then running it will produce

```
$ docker build -t test .
```

```
$ docker run test
```

Hello

You can provide different parameters if you want to, but they will all run /bin/echo

```
$ docker run test Hi
```

Hi

Difference between ENTRYPOINT and CMD

If you want to override the entrypoint listed in your Dockerfile (i.e. if you wish to run a different command than echo in this container), then you need to specify the --entrypoint parameter on the

command line:

```
$ docker run --entrypoint=/home test_image
```

b2c70e74df18

Generally you use the ENTRYPOINT directive to point to your main application you want to run, and CMD to the default parameters.

Exposing a Port in the Dockerfile

The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. EXPOSE does not make the ports of the container accessible to the host.

To do that, you must use either the -p flag to publish a range of ports or the -P flag to publish all of the exposed ports. You can expose one port number and publish it externally under another number.

Inside your Dockerfile:

```
EXPOSE 8765
```

To access this port from the host machine, include this argument in your docker run command:

```
-p 8765:8765
```

ENTRYPOINT and CMD seen as verb and parameter

Suppose you have a Dockerfile ending with

```
ENTRYPOINT [ "nethogs"] CMD ["wlan0"]
```

if you build this image with a `docker built -t inspector` . launch the image built with such a Dockerfile with a command such as

```
docker run -it --net=host --rm inspector
```

,nethogs will monitor the interface named wlan0, Now if you want to monitor the interface eth0 (or wlan1, or ra1...), you will do something like

```
docker run -it --net=host --rm inspector eth0
```

```
docker run -it --net=host --rm inspector wlan1
```

Removivng Images

sudo docker system prune -a // All Images

sudo docker system prune // Only running Images

Pushing and Pulling an Image to Docker Hub or another Registry

Locally created images can be pushed to Docker Hub or any other docker repo host, known as a

registry. Use docker login to sign in to an existing docker hub account.

```
docker login
```

A different docker registry can be used by specifying a server name. This also works for private or

self-hosted registries. Further, using an external credentials store for safety is possible.

```
docker login quay.io
```

Pushing and Pulling an Image to Docker Hub or another Registry

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Different tags can be used to represent different versions, or branches, of the same image. An image with multiple different tags will display each tag in the same repo. Use docker images to see a list of installed images installed on your local machine, including your newly tagged image. Then use push to upload it to the registry and pull to download the image.

```
docker push quay.io/cjsimon/mynginx:latest
```

All tags of an images can be pulled by specifying the -a option

```
docker pull quay.io/cjsimon/mynginx:latest
```

Pushing and Pulling an Image to Docker Hub or another Registry

```
docker tag mynginx quay.io/cjsimon/mynginx:latest
```

Different tags can be used to represent different versions, or branches, of the same image. An image with multiple different tags will display each tag in the same repo. Use docker images to see a list of installed images installed on your local machine, including your newly tagged image. Then use push to upload it to the registry and pull to download the image.

```
docker push quay.io/cjsimon/mynginx:latest
```

All tags of an images can be pulled by specifying the -a option

```
docker pull quay.io/cjsimon/mynginx:latest
```

Building using a proxy

Often when building a Docker image, the Dockerfile contains instructions that runs programs to fetch resources from the Internet. It is possible to instruct Docker to pass set set environment variables so that such programs perform those fetches through a proxy:

```
$ docker build --build-arg http_proxy=http://myproxy.example.com:3128 \
 --build-arg https_proxy=http://myproxy.example.com:3128 \
 --build-arg no_proxy=internal.example.com \
 -t test .
```

build-arg are environment variables which are available at build time only.

Checkpoint and Restore Containers

1. Make sure git and make is installed

```
sudo apt-get install make git-core -y
```

2. install a new kernel (at least 4.2)

```
sudo apt-get install linux-generic-lts-xenial
```

3. reboot machine to have the new kernel active

```
sudo reboot
```

4. compile criu which is needed in order to run docker checkpoint

```
sudo apt-get install libprotobuf-dev libprotobuf-c0-dev protobuf-c-compiler protobufcompiler python-protobuf libnl-3-dev libcap-dev -y
wget http://download.openvz.org/criu/criu-2.4.tar.bz2 -O - | tar -xj
cd criu-2.4
make
make install-lib
make install-criu
```

Checkpoint and Restore Containers

5. check if every requirement is fulfilled to run criu

```
sudo criu check
```

6. compile experimental docker (we need docker to compile docker)

```
cd ~
wget -qO- https://get.docker.com/ | sh
sudo usermod -aG docker $(whoami)
```

.At this point we have to logoff and login again to have a docker daemon. After relog continue with compile step

```
git clone https://github.com/boucher/docker
cd docker
git checkout docker-checkpoint-restore
make #that will take some time - drink a coffee
DOCKER_EXPERIMENTAL=1 make binary
```

Checkpoint and Restore Containers

7. We now have a compiled docker. Lets move the binaries. Make sure to replace <version> with the version installed

```
sudo service docker stop

sudo cp $(which docker) $(which docker)_ ; sudo cp ./bundles/latest/binary-client/docker-<version>-dev $(which docker)

sudo cp $(which docker-containerd) $(which docker-containerd)_ ; sudo cp./bundles/latest/binary-daemon/docker-containerd $(which docker-containerd)

sudo cp $(which docker-containerd-ctr) $(which docker-containerd-ctr)_ ; sudo cp./bundles/latest/binary-daemon/docker-containerd-ctr $(which docker-containerd-ctr)

sudo cp $(which docker-containerd-shim) $(which docker-containerd-shim)_ ; sudo cp ./bundles/latest/binary-daemon/docker-containerd-shim $(which docker-containerd-shim)

sudo cp $(which dockerd) $(which dockerd)_ ; sudo cp ./bundles/latest/binarydaemon/dockerd $(which dockerd)

sudo cp $(which docker-runc) $(which docker-runc)_ ; sudo cp ./bundles/latest/binarydaemon/docker-runc $(which docker-runc)

sudo service docker start
```

Checkpoint and Restore a Container

```
# create docker container
export cid=$(docker run -d --security-opt seccomp:unconfined busybox /bin/sh -c 'i=0; while true; do echo $i; i=$(expr $i + 1); sleep 1; done')
# container is started and prints a number every second display the output with
docker logs $cid
# checkpoint the container
docker checkpoint create $cid checkpointname
# container is not running anymore
docker np
# lets pass some time to make sure # resume container
docker start $cid --checkpoint=checkpointname
# print logs again
docker logs $cid
```

Checkpoint and Restore a Container

```
# create docker container
export cid=$(docker run -d --security-opt seccomp:unconfined busybox /bin/sh -c 'i=0; while true; do echo $i; i=$(expr $i + 1); sleep 1; done')
# container is started and prints a number every second display the output with
docker logs $cid
# checkpoint the container
docker checkpoint create $cid checkpointname
# container is not running anymore
docker np
# lets pass some time to make sure # resume container
docker start $cid --checkpoint=checkpointname
# print logs again
docker logs $cid
```

Concept of Docker Volumes

- Docker containers are stateless. to avoid the issue is to create a docker volume and attach it to your MySQL container. Here are the commands to create a MySQL container including attached volume in your local machine:
- Docker filesystems are temporary by default. If you start up a Docker image you'll get a container that on the surface behaves much like a virtual machine.
- You can create, modify, and delete files. However, unlike a virtual machine, if you stop the container and start it up again, all your changes will be lost -- any files you previously deleted will now be back, and any new files or edits you made won't be present. Volumes in docker containers allow for persistent data, and for sharing host-machine data inside a container.

Launch a container with a volume

[root@localhost ~]# docker run -it -v /data --name=vol3 8251da35e7a7 /bin/bash

root@d87bf9607836:/# cd /data/

root@d87bf9607836:/data# touch abc{1..10}

root@d87bf9607836:/data# ls

Now press [cont +P+Q] to move out from container without terminating the container checking for container that is running

[root@localhost ~]# docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES d87bf9607836

8251da35e7a7 "/bin/bash" About a minute ago Up 31 seconds vol3 [root@localhost ~]#

Run 'docker inspect' to check out more info about the volume

[root@localhost ~]# docker inspect d87bf9607836

You can attach a running containers volume to another containers

[root@localhost ~]# docker run -it --volumes-from vol3 8251da35e7a7 /bin/bash

root@ef2f5cc545be:/# ls

bin boot data dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var

root@ef2f5cc545be:/# ls /data abc1 abc10 abc2 abc3 abc4 abc5 abc6 abc7 abc8 abc9

You can also mount you base directory inside container

[root@localhost ~]# docker run -it -v /etc:/etc1 8251da35e7a7 /bin/bash

Here: /etc is host machine directory and /etc1 is the target inside container

Connecting Containers

The host and bridge network drivers are able to connect containers on a single docker host. To

allow containers to communicate beyond one machine, create an overlay network. Steps to create

the network depend on how your docker hosts are managed.

Docker network

Containers in the same docker network have access to exposed ports.

```
docker network create sample
```

```
docker run --net sample --name keys consul agent -server -client=0.0.0.0 -bootstrap
```

Consul's Dockerfile exposes 8500, 8600, and several more ports. To demonstrate, run another container in the same network:

```
docker run --net sample -ti alpine sh
```

Here the consul container is resolved from keys, the name given in the first command. Docker provides dns resolution on this network, to find containers by their --name

Docker-compose

Networks can be specified in a compose file (v2). By default all the containers are in a shared network. docker-compose.yml:

```
version: '2'
Services:
    keys:
        image: consul
        command: agent -server -client=0.0.0.0 -bootstrap
   test:
        image: alpine
        tty: true
        command: sh
```

Starting this stack with **docker-compose up -d** will create a network named after the parent directory, in this case **example_default**. Check with **docker network ls**

```
> docker network ls
```

Docker-compose

Connect to the alpine container to verify the containers can resolve and communicate:

> docker exec -ti example_test_1 sh

/ # nslookup keys

...

/ # wget -qO- keys:8500/v1/kv/?recurse

...

A compose file can have a networks: top level section to specify the network name, driver, and other options from the docker network command

Container Linking

The docker --link argument, and link: sections docker-compose make aliases to other containers.

docker network create sample

docker run -d --net sample --name redis redis

With link either the original name or the mapping will resolve the redis container.

> docker run --net sample --link redis:cache -ti python:alpine sh -c "pip install redis && python"

>>> import redis

>>> r = redis.StrictRedis(host='cache')

>>> r.set('key', 'value')

True

Creating a service with persistence

docker volume create --name <volume_name> # Creates a volume called <volume_name>

docker run -v <volume_name>:<mount_point> -d crramirez/limesurvey:latest # Mount the <volume_name> volume in <mount_point> directory in the container They persist even when the container is removed using the -v option.

- The only way to delete a named volume is doing an explicit call to docker volume rm

The named volumes can be shared among container without linking or --volumes-from option.

- They don't have permission issues that host mounted volumes have.
- They can be manipulated using docker volume command.

Persistence with named volumes

Persistence is created in docker containers using volumes. Let's create a Limesurvey container and persist the database, uploaded content and configuration in a named volume:

```
docker volume create --name mysql
```

```
docker volume create --name upload
```

```
docker run -d --name limesurvey -v mysql:/var/lib/mysql -v upload:/app/upload -p 80:80 crramirez/limesurvey:latest
```

Backup a named volume content

We need to create a container to mount the volume. Then archive it and download the archive to our host.

Let's create first a data volume with some data:

```
docker volume create --name=data
echo "Hello World" | docker run -i --rm=true -v data:/data ubuntu:trusty tee /data/hello.txt
```

Let's backup the data:

```
docker run -d --name backup -v data:/data ubuntu:trusty tar -czvf /tmp/data.tgz /data
docker cp backup:/tmp/data.tgz data.tgz
docker rm -fv backup
```

Let's test:

```
tar -xzvf data.tgz
cat data/hello.txt
```

Data Volumes and Data Containers

Many resources on the web from the last couple of years mention using a pattern called a "dataonly container", which is simply a Docker container that exists only to keep a reference to a data

volume around.

Remember that in this context, a "data volume" is a Docker volume which is not mounted from the

host. To clarify, a "data volume" is a volume which is created either with the VOLUME Dockerfile

directive, or using the -v switch on the command line in a docker run command, specifically with

the format -v /path/on/container. Therefore a "data-only container" is a container whose only

purpose is to have a data volume attached, which is used by the --volumes-from flag in a docker run

command.

Data Volumes and Data Containers

```
docker run -d --name "mysql-data" -v "/var/lib/mysql" alpine /bin/true
```

When the above command is run, a "data-only container" is created. It is simply an empty container which has a data volume attached. It was then possible to use this volume in another

container like so:

```
docker run -d --name="mysql" --volumes-from="mysql-data" mysql
```

The mysql container now has the same volume in it that is also in mysql-data.

Creating a data volume

```
docker run -d --name "mysql-1" -v "/var/lib/mysql" mysql
```

creates a new container from the mysql image. It also creates a new data volume, which it then mounts in the container at /var/lib/mysql.

This volume helps any data inside of it persist beyond the lifetime of the container. That is to say, when a container is removed, its filesystem changes are also removed.

If a database was storing data in the container, and the container is removed, all of that data is also removed. Volumes will persist a particular location even beyond when its container is removed

Creating a data volume

It is possible to use the same volume in multiple containers with the --volumes-from command line option:

```
docker run -d --name="mysql-2" --volumes-from="mysql-1" mysql
```

The mysql-2 container now has the data volume from mysql-1 attached to it, also using the path

`/var/lib/mysql`.

Debugging a container

To execute operations in a container, use the docker exec command. Sometimes this is called "entering the container" as all commands are executed inside the container.

```
docker exec -it container_id bash
```

or

```
docker exec -it container_id /bin/sh
```

And now you have a shell in your running container. For example, list files in a directory and then leave the container:

```
docker exec container_id ls -l
```

Monitoring resource usage

Inspecting system resource usage is an efficient way to find misbehaving applications. This

example is an equivalent of the traditional top command for containers:

```
docker stats
```

```
docker stats 7786807d8084 7786807d8085
```

Monitoring processes in a container

Inspecting system resource usage is an efficient way to find misbehaving applications. This

example is an equivalent of the traditional top command for containers:

```
docker stats
```

```
docker stats 7786807d8084 7786807d8085
```

Monitoring processes in a container

Inspecting system resource usage is an efficient way to narrow down a problem on a live running application. This example is an equivalent of the traditional ps command for containers.

```
docker top 7786807d8084
```

To filter of format the output, add ps options on the command line:

```
docker top 7786807d8084 faux
```

Or, to get the list of processes running as root, which is a potentially harmful practice:

```
docker top 7786807d8084 -u root
```

Attach to a running container

'Attaching to a container' is the act of starting a terminal session within the context that the container (and any programs therein) is running. This is primarily used for debugging purposes, but may also be needed if specific data needs to be passed to programs running within the container.

The attach command is utilized to do this. It has this syntax:

```
docker attach <container>
```

<container> can be either the container id or the container name. For instance:

```
docker attach c8a9cf1a1fa8
```

To detach from an attached container, successively hit Ctrl-p then Ctrl-q

Printing the logs

docker logs --follow --tail 10 7786807d8084

start the failing container with docker run ... ; docker logs $(docker ps -lq)

- find the container id or name with

docker ps -a

docker logs container-id or

docker logs containername

as it is possible to look at the logs of a stopped container

Docker container process debugging

sudo ps aux

Any currently running Docker containers will be listed in the output. This can be useful during application development for debugging a process running in a container.

As a user with appropriate permissions, typical debugging utilities can be used on the container process, such as strace, ltrace, gdb, etc.

Docker Data Volumes

Docker data volumes provide a way to persist data independent of a container's life cycle.Volumes present a number of helpful features such as:

Mounting a host directory within the container, sharing data in-between containers using the filesystem and preserving data if a container gets deleted

Mounting a directory from the local host into a container

to mount a host directory to a specific path in your container using the -v or –volume command line option. The following example will mount /etc on the host to /mnt/etc in the container:

```
(on linux) docker run -v "/etc:/mnt/etc" alpine cat /mnt/etc/passwd
```

```
(on windows) docker run -v "/c/etc:/mnt/etc" alpine cat /mnt/etc/passwd
```

The default access to the volume inside the container is read-write. To mount a volume read-only inside of a container, use the suffix :ro:

```
docker run -v "/etc:/mnt/etc:ro" alpine touch /mnt/etc/passwd
```

Creating a named volume

```
docker volume create --name="myAwesomeApp"
```

Using a named volume makes managing volumes much more human-readable. It is possible to create a named volume using the command specified above, but it's also possible to create a

named volume inside of a docker run command using the -v or --volume command line option:

```
docker run -d --name="myApp-1" -v="myAwesomeApp:/data/app" myApp:1.5.3
```

Docker Engine API

An API that allows you to control every aspect of Docker from within your own applications, build tools to manage and monitor applications running on Docker, and even use it to build apps on Docker itself.

Edit **/etc/init/docker.conf** and update the **DOCKER_OPTS** variable to the following:

```
DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
```

Restart Docker deamon

```
service docker restart
```

Verify if Remote API is working

```
curl -X GET http://localhost:4243/images/json
```

Enable Remote access to Docker API on Linux running systemd

create a file called /etc/systemd/system/docker-tcp.socket to make docker available on a TCP socket on port 4243:

```
[Unit]
Description=Docker Socket for the API
[Socket]
ListenStream=4243
Service=docker.service
[Install]
WantedBy=sockets.target
```

Enable Remote access to Docker API on Linux running systemd

Then enable the new socket:

```
systemctl enable docker-tcp.socket
systemctl enable docker.socket
systemctl stop docker
systemctl start docker-tcp.socket
systemctl start docker
```

Now, verify if Remote API is working:

```
curl -X GET http://localhost:4243/images/json
```

Docker events

docker events provides details, but when debugging it may be useful tolaunch a container and be notified immediately of any related event:

```
docker run... & docker events --filter 'container=$(docker ps -lq)'
```

In docker ps -lq, the l stands for last, and the q for quiet. This removes the id of the last container launched, and creates a notification immediately if the container dies or has another event occur

Docker in Docker

set up a Docker Container with Jenkins inside, which is capable of sending Docker commands to the Docker installation of the Host. Effectively using Docker in Docker. We have to build a custom Docker Image which is based on an arbitrary version of the official Jenkins Docker Image.This Dockerfile builds an Image which contains the Docker client binaries this client is used to communicate with a Docker Daemon.

Docker in Docker

```
FROM jenkins
USER root
RUN cd /usr/local/bin && \
curl https://master.dockerproject.org/linux/amd64/docker > docker && \
chmod +x docker && \
groupadd -g 999 docker && \
usermod -a -G docker jenkins
USER Jenkins
```

Docker Machine

The **docker-machine** command line tool manages the full machine's life cycle using provider specific drivers. It can be used to select an "active" machine. Once selected, an active machine can be used as if it was the local Docker Engine.

docker-machine env to get the current default docker-machine configuration

eval $(docker-machine env) to get the current docker-machine configuration and set the current shell environment up to use this docker-machine with .

Create a Docker machine

Using docker-machine is the best method to install Docker on a machine. It will automatically apply the best security settings available, including generating a unique pair of SSL certificates for mutual authentication and SSH keys.

To create a local machine using Virtualbox:

```
docker-machine create --driver virtualbox docker-host-1
```

To install Docker on an existing machine, use the generic driver:

```
docker-machine -D create -d generic --generic-ip-address 1.2.3.4 docker-host-2
```

The --driver option tells docker how to create the machine

```
docker-machine ls
```

Upgrade a Docker Machine

Upgrading a docker machine implies a downtime and may require planing. To upgrade a docker machine, run:

```
docker-machine upgrade docker-machine-name
```

This command does not have options Get the IP address of a docker machine To get the IP address of a docker machine, you can do that with this command :

```
docker-machine ip machine-name
```

Docker --net modes

Bridge Mode It's a default and attached to docker0 bridge. Put container on a completely separate network namespace.

Host Mode When container is just a process running in a host, we'll attach the container to the host NIC.

Mapped Container Mode This mode essentially maps a new container into an existing containers network stack. It's also called 'container in container mode'.

None It tells docker put the container in its own network stack without configuration

Bridge Mode

```
$ docker run –d –-name my_app -p 10000:80 image_name
```

Note that we did not have to specify --net=bridge because this is the default working mode for docker. This allows to run multiple containers to run on same host without any assignment of dynamic port. So BRIDGE mode avoids the port clashing and it's safe as each container is running its own private network namespace.

Host Mode

```
$ docker run –d –-name my_app –net=host image_name
```

As it uses the host network namespace, no need of special configuraion but may leads to security issue.

Mapped Container Mode

This mode essentially maps a new container into an existing containers network stack. This implies that network resources such as IP address and port mappings of the first container will be shared by the second container.

This is also called as 'container in container' mode. Suppose you have two contaienrs as web_container_1 and web_container_2 and we'll run web_container_2 in mapped container mode.

Let's first download web_container_1 and runs it into detached mode

Mapped Container Mode

```
$ docker run -d --name web1 -p 80:80 USERNAME/web_container_1
```

Once it’s downloaded let’s take a look and make sure its running. Here we just mapped a port into a container that's running in the default bridge mode. Now, let’s run a second container in mapped container mode. We’ll do that with this command.

```
$ docker run -d --name web2 --net=container:web1 USERNAME/web_container_2
```

Now, if you simply get the interface information on both the contaienrs, you will get the same network config. This actually include the HOST mode that maps with exact info of the host.

Docker network

```
$ docker-machine ls
$ docker-machine ip default
$ docker network create app-backend
$ docker network ls
$ docker network connect app-backend myAwesomeApp-1
$ docker network disconnect app-backend myAwesomeApp-1
$ docker network rm app-backend
$ docker network inspect app-backend
```

Docker Registry

Do not use registry:latest! This image points to the old v1 registry. That Python project is no longer being developed. The new v2 registry is written in Go and is actively maintained. When

people refer to a "private registry" they are referring to the v2 registry, not the v1 registry!

```
docker run -d -p 5000:5000 --name="registry" registry:2
```

The above command runs the newest version of the registry

Docker stats all running containers

Docker stats all running containers

```
sudo docker stats $(sudo docker inspect -f "{{ .Name }}" $(sudo docker ps -q))
```

Shows live CPU usage of all running containers

Docker swarm mode

A swarm is a number of Docker Engines (or nodes) that deploy services collectively. Swarm is

used to distribute processing across many physical, virtual or cloud machines.

Features of swarm

Cluster management integrated with Docker Engine

- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure design by default
- Rolling updates

Swarm Mode CLI Commands

docker swarm init [OPTIONS]

docker swarm join [OPTIONS] HOST:PORT

docker service create [OPTIONS] IMAGE [COMMAND] [ARG...]

docker service inspect [OPTIONS] SERVICE [SERVICE...]

docker service ls [OPTIONS]

docker service rm SERVICE [SERVICE…]

docker service scale SERVICE=REPLICAS [SERVICE=REPLICAS…]

docker service ps [OPTIONS] SERVICE [SERVICE…]

docker service update [OPTIONS] SERVICE

Persistence

Database needs persistence, so we need some filesystem which is shared across all the nodes in

a swarm. It can be NAS, NFS server, GFS2 or anything else. Setting it up is out of scope here.

Currently Docker doesn't contain and doesn't manage persistence in a swarm. This example

assumes that there's /nfs/ shared location mounted across all nodes.

Sample Dockerfile

```
# Base image
FROM python:2.7-alpine
# Metadata MAINTAINER is depricated
MAINTAINER John Doe <johndoe@example.com>
LABEL Maintainer="John Doe johndoe@example.com"
# System-level dependencies
RUN apk add --update  ca-certificates  && update-ca-certificates  && rm -rf /var/cache/apk/*
# App dependencies
COPY requirements.txt /requirements.txt
RUN pip install -r /requirements.txt
# App codebase
WORKDIR /app
COPY . ./
# Configs
ENV DEBUG true
EXPOSE 5000
CMD ["python", "app.py"]
```

Dockerfiles

Dockerfiles are files used to programatically build Docker images. They allow you to quickly and reproducibly create a Docker image, and so are useful for collaborating. Dockerfiles contain instructions for building a Docker image. Each instruction is written on one row, and is given in the form <INSTRUCTION><argument(s)>. Dockerfiles are used to build Docker images using the docker build command.

When building a Dockerfile, the Docker client will send a "build context" to the Docker daemon.

The build context includes all files and folder in the same directory as the Dockerfile. COPY and ADD

operations can only use files from this context.

Sample Dockerfile

FROM alpine

CMD ["echo", "Hello StackOverflow!"]

This will instruct Docker to build an image based on Alpine (FROM), a minimal distribution for

containers, and to run a specific command (CMD) when executing the resulting image

docker build -t hello .

docker run --rm hello

COPY localfile.txt containerfile.txt

COPY ["local file", "container file"]

COPY *.jpg images/

EXPOSE 8080 8082

RUN apt-get -qq update

WORKDIR Instruction

WORKDIR /path/to/workdir

The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.

If the WORKDIR doesn’t exist, it will be created even if it’s not used in any subsequent Dockerfile instruction. It can be used multiple times in the one Dockerfile. If a relative path is provided, it will be relative

to the path of the previous WORKDIR instruction. For example:

```
WORKDIR /a
WORKDIR b
RUN pwd
ENV DIRPATH /path
WORKDIR $DIRPATH/$DIRNAME
```

WORKDIR Instruction

FROM ubuntu

RUN mkdir /myvol

RUN echo "hello world" > /myvol/greeting

VOLUME /myvol

This Dockerfile results in an image that causes docker run, to create a new mount point at /myvol and copy the greeting file into the newly created volume.

Note: If any build steps change the data within the volume after it has been declared, those changes will be discarded.

docker run -p 2500:80 <image name>

Dockerfile

```
# escape=`

FROM windowsservercore

SHELL ["powershell","-command"]

RUN New-Item -ItemType Directory C:\Example

ADD Execute-MyCmdlet.ps1 c:\example\

RUN c:\example\Execute-MyCmdlet -sample 'hello world'
```

Dockerfile

```
FROM debian
RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y \
 git \
 openssh-client sudo \
 vim \
 wget \
 && apt-get clean \
 && rm -rf /var/lib/apt/lists/*
```

Debugging when docker build fails

When a **docker build -t mytag .** fails with a message such as ---> Running in d9a42e53eb5a The command '/bin/sh -c returned a non-zero code: 127 (127 means "command not found, but

1) it is not trivial for everybody

2) 127 may be replaced by 6 or anything) it may be non trivial to find the error in a long line

You just launch the last created image with a shell and launch the command, and you will have a

more clear error message

docker run -it d9a42e53eb5a /bin/bash

How to Setup Three Node Mongo Replica using Docker Image and Provisioned using Chef

1) Generate a Base 64 keyfile for Mongo node authentication. Place this file in chef data_bags
2) Go to chef suppermarket and download docker cookbook. Generate a custom cookbook (e.g custom_mongo) and add depends 'docker', '~> 2.0' to your cookbook's metadata.rb
3) Create an attributes and recipe in your custom cookbook
4) Initialise Mongo to form Rep Set cluster

Step 1: Create Key file

create data_bag called mongo-keyfile and item called keyfile. This will be in the data_bags directory in chef. Item content will be as below

```
openssl rand -base64 756 > <path-to-keyfile>
```

keyfile item content

```
{
 "id": "keyfile",
 "comment": "Mongo Repset keyfile",
 "key-file": "generated base 64 key above"
}
```

Step 2: Download docker cookbook from chef supper market and then createcustom_mongo cookbook

knife cookbook site download docker

knife cookbook create custom_mongo

in metadat.rb of custom_mongo add

depends 'docker', '~> 2.0’

Step 3: create attribute and recipe

Attributes

default['custom_mongo']['mongo_keyfile'] = '/data/keyfile'

default['custom_mongo']['mongo_datadir'] = '/data/db'

default['custom_mongo']['mongo_datapath'] = '/data'

default['custom_mongo']['keyfilename'] = 'mongodb-keyfile'

Step 4: Initialise the three node Mongo to form repset

```
docker run --name mongo -v /data/db:/data/db -v /data/keyfile:/opt/keyfile --hostname="mongo01.example.com" -p 27017:27017 -d mongo:3.4.2 --keyFile /opt/keyfile/mongodb-keyfile --auth
```

Access the interactive shell of running docker container on node 01 and Create admin user

```
docker exec -it mongo /bin/sh
 mongo
 use admin
 db.createUser( {
 user: "admin-user",
 pwd: "password",
 roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
 });
```

Inspecting a running container

To get all the information for a container you can run:

```
docker inspect <container>
```

You can get an specific information from a container by running:

```
docker inspect -f '<format>' <container>
```

For instance, you can get the Network Settings by running:

```
docker inspect -f '{{ .NetworkSettings }}' <container>
```

You can also get just the IP address:

```
docker inspect -f '{{ .NetworkSettings.IPAddress }}' <container>
```

Iptables with Docker

To limit access to your docker containers from outside world using iptables.

iptables -I DOCKER [RULE ...] [ACCEPT|DROP] // To add a rule a the top of the DOCKER table

iptables -D DOCKER [RULE ...] [ACCEPT|DROP] // To remove a rule from the DOCKER table

ipset restore < /etc/ipfriends.conf // To reconfigure your ipset ipfriends

Configuring iptables rules for Docker containers is a bit tricky. At first, you would think that "classic" firewall rules should do the trick

Iptables with Docker

```
$ iptables -A INPUT -i eth0 -p tcp -s XXX.XXX.XXX.XXX -j ACCEPT
$ iptables -P INPUT DROP
$ iptables -L
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

adding a rule at the top of the DOCKER table is a good idea. It does not interfere with the rules automatically configured by Docker, and it is simple.

```
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
$ iptables -I DOCKER -i ext_if -m set ! --match-set my-ipset src -j DROP
// Then Accept rules for established connections
$ iptables -I DOCKER -i ext_if -m state --state ESTABLISHED,RELATED -j ACCEPT
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 3rd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 2nd custom accept rule
$ iptables -I DOCKER -i ext_if ... ACCEPT // Then 1st custom accept rule
```

Logging

Docker's approach to logging is that you construct your containers in such a way, so that logs are written to standard output (console/terminal). If you already have a container which writes logs to a file, you can redirect it by creating a symbolic link:

```
ln -sf /dev/stdout /var/log/nginx/access.log
ln -sf /dev/stderr /var/log/nginx/error.log
```

After you've done that you can use various log drivers to put your logs where you need them

Managing containers

docker rm [OPTIONS] CONTAINER [CONTAINER...]

docker attach [OPTIONS] CONTAINER

docker exec [OPTIONS] CONTAINER COMMAND [ARG...]

docker ps [OPTIONS]

docker logs [OPTIONS] CONTAINER

docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE…]

docker ps

docker ps -a

docker ps -a -f status=exited

ocker ps -aq

ocker ps -f name=mycontainer1

Managing images

docker images [OPTIONS] [REPOSITORY[:TAG]]

docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

docker rmi [OPTIONS] IMAGE [IMAGE...]

docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

docker pull ubuntu

docker pull ubuntu:14.04

docker pull registry.example.com/username/ubuntu:14.04

docker images

docker rmi <image name>

ocker rmi registry.example.com/username/myAppImage:1.3.5

Multiple processes in one container instance

To run multiple processes e.g. an Apache web server together with an SSH daemon inside the same container you can use supervisord. Create your supervisord.conf configuration file like:

```
[supervisord]
nodaemon=true
[program:sshd]
command=/usr/sbin/sshd -D
[program:apache2]
command=/bin/bash -c "source /etc/apache2/envvars && exec /usr/sbin/apache2 -DFOREGROUND"
```

Then create a Dockerfile like:

```
FROM ubuntu:16.04
RUN apt-get install -y openssh-server apache2 supervisor
RUN mkdir -p /var/lock/apache2 /var/run/apache2 /var/run/sshd /var/log/supervisor
COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
CMD ["/usr/bin/supervisord"]
```

Multiple processes in one container instance

Then you can build your image:

```
docker build -t supervisord-test .
```

Afterwards you can run it:

```
$ docker run -p 22 -p 80 -t -i supervisord-test
```

passing secret data to a running container

`docker run`

such as

`docker run -e password=abc`

or in a file

`docker run --env-file myfile`

where myfile can contain

`password1=abc password2=def`

it is also possible to put them in a volume

`docker run -v $(pwd)/my-secret-file:/secret-file`

some better ways, use

Restricting container network access

docker network create -o "com.docker.network.bridge.enable_ip_masquerade"="false" lanrestricted

docker network create -o "com.docker.network.bridge.enable_icc"="false" icc-restricted

iptables -I INPUT -i docker0 -m addrtype --dst-type LOCAL -j DROP

docker network create --subnet=192.168.0.0/24 --gateway=192.168.0.1 –ip-range=192.168.0.0/25 local-host-restricted

iptables -I INPUT -s 192.168.0.0/24 -m addrtype --dst-type LOCAL -j DROP

Running Simple Node.js Application

```
{
"name": "docker_web_app",
"version": "1.0.0",
"description": "Node.js on Docker",
"author": "First Last <first.last@example.com>",
"main": "server.js",
"scripts": {
"start": "node server.js"
},
"dependencies": {
"express": "^4.13.3"
}
}
```

Running Simple Node.js Application

```
var express = require('express');
 var PORT = 8080;
 var app = express();
 app.get('/', function (req, res) {
 res.send('Hello world\n');
 });
 app.listen(PORT);
 console.log('Running on http://localhost:' + PORT);
```

Running Simple Node.js Application

```
FROM node:latest
RUN mkdir -p /usr/src/my_first_app
WORKDIR /usr/src/my_first_app
COPY package.json /usr/src/my_first_app/
RUN npm install
COPY . /usr/src/my_first_app
EXPOSE 8080
```

security

```
FROM node:latest
RUN mkdir -p /usr/src/my_first_app
WORKDIR /usr/src/my_first_app
COPY package.json /usr/src/my_first_app/
RUN npm install
COPY . /usr/src/my_first_app
EXPOSE 8080
```

Create Dockerfile

- Create a file with the name **Dockerfile** under the directories **src/main/docker** with the contents shown below. Note that this file is important to create a Docker image.

```
FROM java:8

VOLUME /tmp

ADD dockerapp-0.0.1-SNAPSHOT.jar app.jar

RUN bash -c 'touch /app.jar'

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

Docker plugin in pom.xml

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>1.0.0</version>
    <configuration>
      <imageName>${docker.image.prefix}/${project.artifactId}</imageName>
      <dockerDirectory>src/main/docker</dockerDirectory>
      <resources>
        <resource>
          <directory>${project.build.directory}</directory>
          <include>${project.build.finalName}.jar</include>
        </resource>
      </resources>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
  </plugin>
```

Application Execution

you can run your application by using the Maven command **mvn package docker:build**

Note − Enable the Expose daemon on **tcp://localhost:2375** without TLS.

After build success, see the Docker images by the command using docker images and see the image info on the console.