**Understanding Docker**

***Docker Domain Early competitors***

* KVM, Xen, OpenStack, Mesos, Capistrano, Fabric, Ansible, Chef, Puppet, SaltStack, and so on.

***Solving the below problems***

* Shipping software at the speed expected in today’s world is hard to do well

***Highlight***

When Docker released their libswarm development library at DockerCon 2014, an engineer from Orchard demonstrated deploying a Docker container to a heterogeneous mix of cloud providers at the same time. This kind of orchestration had not been easy before because every cloud provider provided a different API or toolset for managing instances, which were usually the smallest item you could manage with an API.

***Important Terminologies***

**Docker client** - This is the ***docker command*** used to control most of the Docker workflow and talk to remote Docker servers.

**Docker server** - This is the ***dockerd command*** that is used to start the Docker server process that builds and launches containers via a client.

**Docker images -** Docker images consist of one or more filesystem layers and some important metadata that represent all the files required to run a Dockerized application. A single Docker image can be copied to numerous hosts. An image typically has both a name and a tag. The tag is generally used to identify a particular release of an image.

**Docker container -** A Docker container is a Linux container that has been instantiated from a Docker image. A specific container can exist only once; however, you can easily create multiple containers from the same image.

**Atomic host -** An atomic host is a small, finely tuned OS image, like Fedora CoreOS, that supports container hosting and atomic OS upgrades.

***If all of your tooling is around Docker, your applications can be deployed in a cloud-agnostic manner, allowing a huge new flexibility not previously possible***

***Docker Server (daemon) can run on***

* Linux ( Mostly used)
* Windows

***Docker Client***

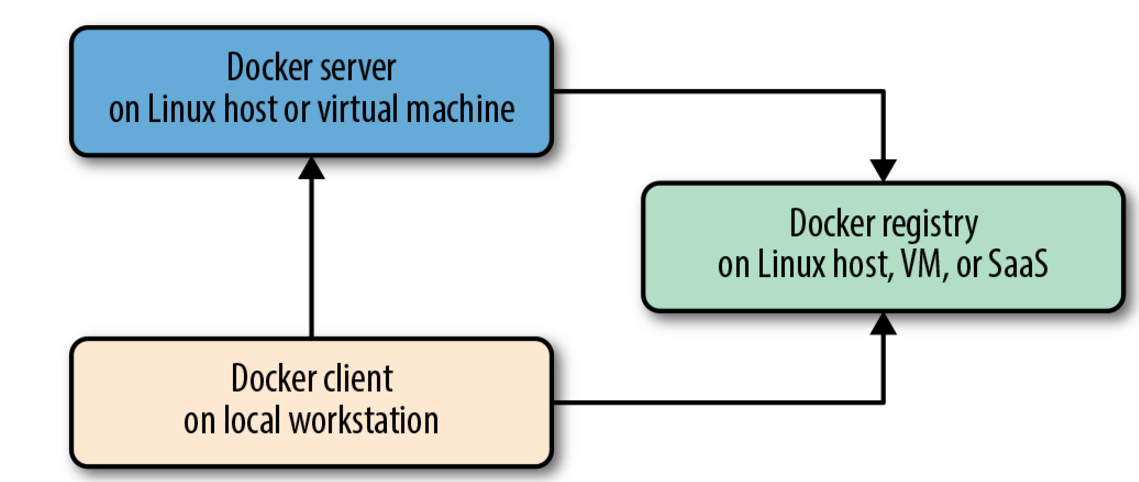
can run on different Operating Systems

***Open container initiative***

Previously docker images where their own proprietary . After public concerns, open container initiative was launched and the all are open source which should follow certain standards (demanded by oci spec)

***Docker written in go programming language***

***Docker Architecture (Client Server Model)***



***Confusion - Docker Swarm and Docker which has a Swarm Mode***

There was an older, now deprecated project that was a standalone application, called “Docker Swarm,” and there is a newer, built-in Swarm that we refer to here as “Swarm mode.” You’ll need to carefully look at the search results to identify the relevant ones. Over time this conflict will hopefully become moot as the older product fades into history.

***Dockers own orchestration tool set***

* Docker compose 🡪 works well in local deployment
* Swarm
* Machine

***Popular docker orchestration products***

* Kubernetes
* Mesos

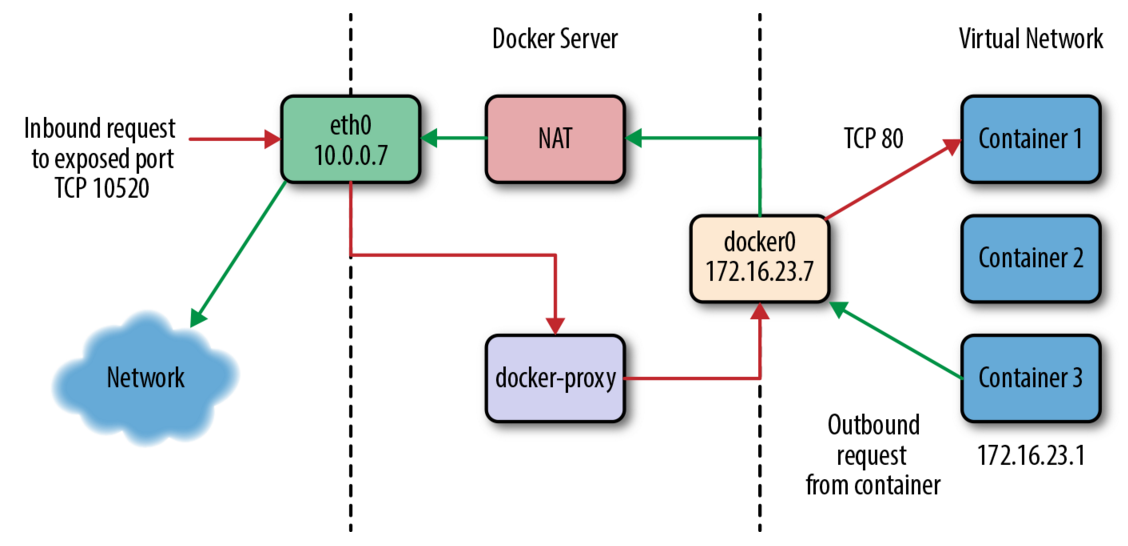
***Docker command – supports lot of functionalities***

* Build a container image
* Pull an image from registry ( Local, Docker Hub etc.)
* Start a docker container
* Retrieve docker logs
* CLI into container
* Stats and Process listing inside container

***Docker command – All the functionalities of CLI is available as API ( Python, go SDK)***

***Docker networking***

Docker 0 acts as virtual bridge between the containers . This can help interacting with the outside world



***Good use cases***

Some good applications for beginning with Docker include web frontends, backend APIs, and short-running tasks like maintenance scripts that might normally be handled by cron.

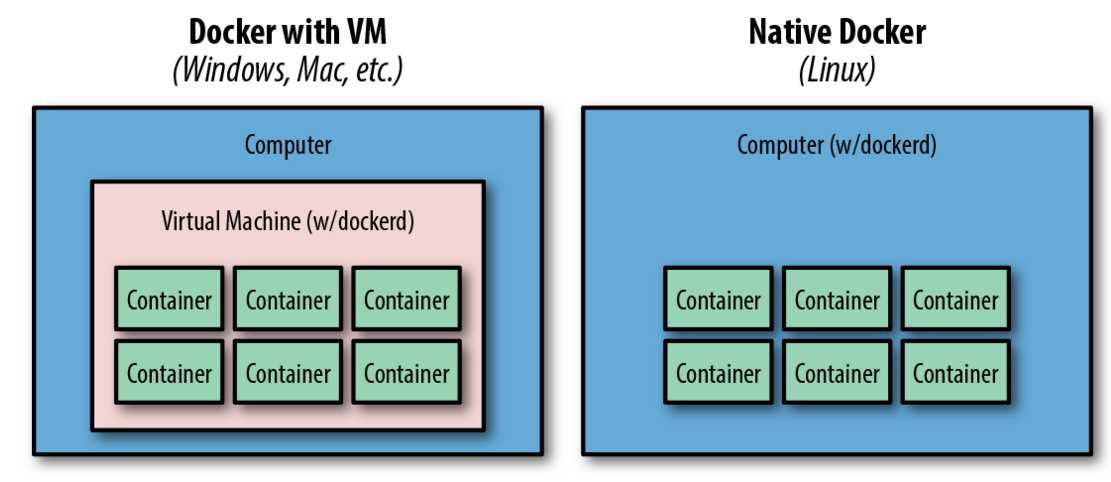
***Bad use cases for docker***

Doing things like putting a database engine inside Docker is a bit like swimming against the current. It’s not that you can’t do it, or even that you shouldn’t do it; it’s just that this is not the most obvious use case for Docker

***Use case selection***

starting with stateless applications and learning from that experience before tackling other use cases

***Docker Server or Docker Daemon can be run in a host server or in a virtual machine***



***Container isolation***

* *Each container is isolated from one another. However, still they shared the CPU and memory of the host computer*
* *Containers are nothing but the processes of the docker daemon (server)*
* *A quick test reveals that a newly created container from an existing image takes a whopping 12 kilobytes of disk space - On the other hand, a new virtual machine created from a golden image might require hundreds or thousands of megabyte*

***Stateless scenario***

Containers – should be typically used in stateless scenarios. ( Ex: no need maintain the data (state) , huge data etc. ) -- > Because when you spawn a new container, no need to copy all its data. Such scenarios should be chosen when deploying containerization

***Suppose state has to be stored***

Databases are often where scaled applications store state, and nothing in Docker interferes with doing that for containerized applications. Applications that need to store files, however, face some challenges. Storing things to the container’s filesystem will not perform well, will be extremely limited by space, and will not preserve state across a container lifecycle. If you redeploy a stateful service without utilizing storage external to the container, you will lose all of that state. Applications that need to store filesystem state should be considered carefully before you put them into Docker

***Docker workflow***

***When there are changes to the existing code ? ( containers )***

Remember that a Docker image contains everything required to run your application. If you change one line of code, you certainly don’t want to waste time rebuilding every dependency your code requires into a new image. Instead, Docker will use as many base layers as it can so that only the layers affected by the code change are rebuilt.

***Docker image tagging***

Docker has a built-in mechanism for handling this: it provides image tagging at deployment time. You can leave multiple revisions of your application on the server and just tag them at release.

***Build Procedure***

The Docker command-line tool contains a build flag that will consume a *Dockerfile* and produce a Docker image. Each command in a *Dockerfile* generates a new layer in the image, so it’s easy to reason about what the build is going to do by looking at the *Dockerfile* itself. The great part of all of this standardization is that any engineer who has worked with a *Dockerfile* can dive right in and modify the build of any other application

***Docker Images***

Docker container is built from docker image. Images are the core component of docker

Docker Container

Docker Container

Docker Container

Uses underlying Linux kernel for filesystem storage ( Multiple layers exists)

Docker Image

Container depends on Image for its file system. Container and image are closely related

Unable to delete the image since container is using the image

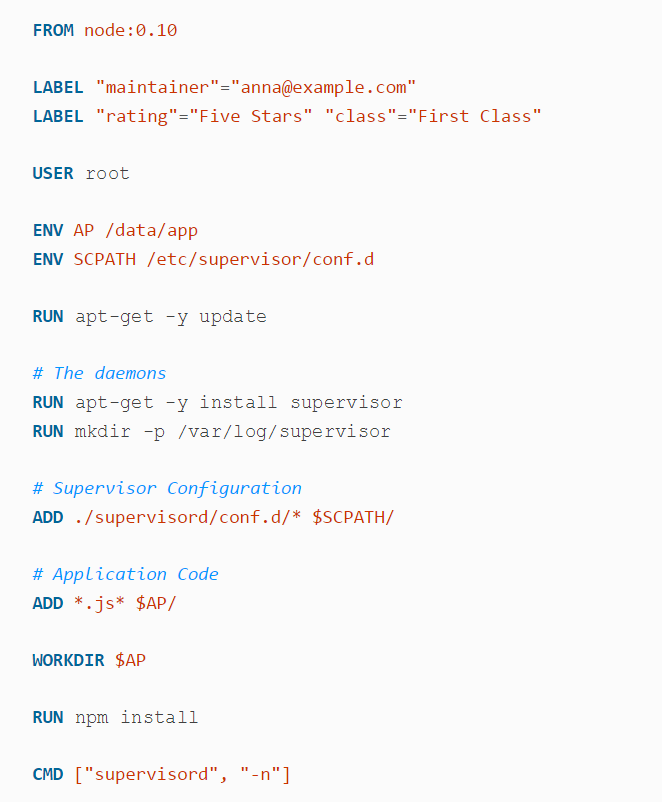


Docker image can be pulled from docker hub etc. or we can build our own image

***Docker Image Creation***

*DockerFile 🡪 Tool to create Docker Images*

*Sample Docker File*



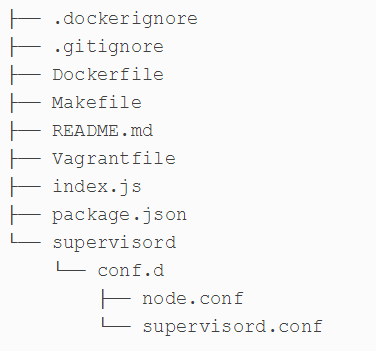
*Important thing to be noted here is ,*

* *Each line builds a layer on the image*
* *Order is very important*
* *In future, if there is a change – a layer will be built on top of these commands*

***Building a Sample Docker Image***

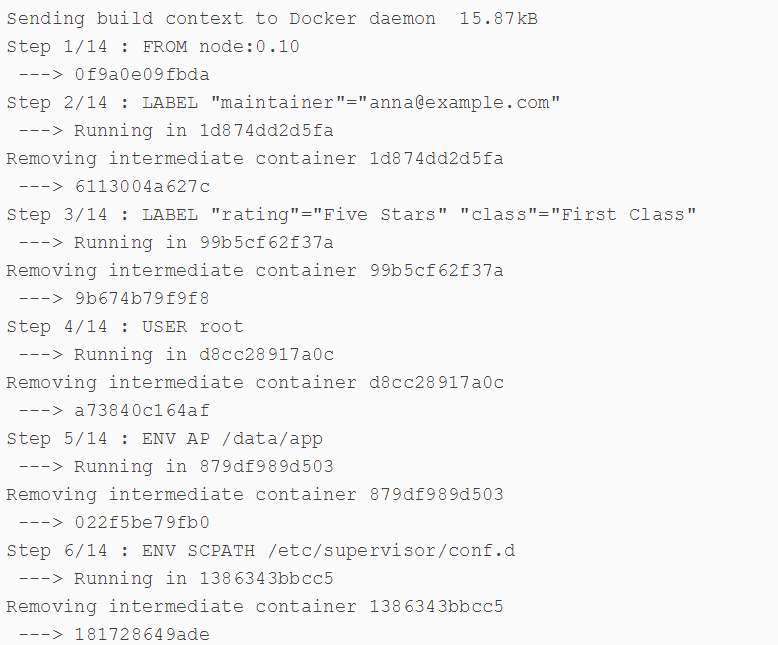
Sample git repo is cloned in Local - This builds a local NODE JS image

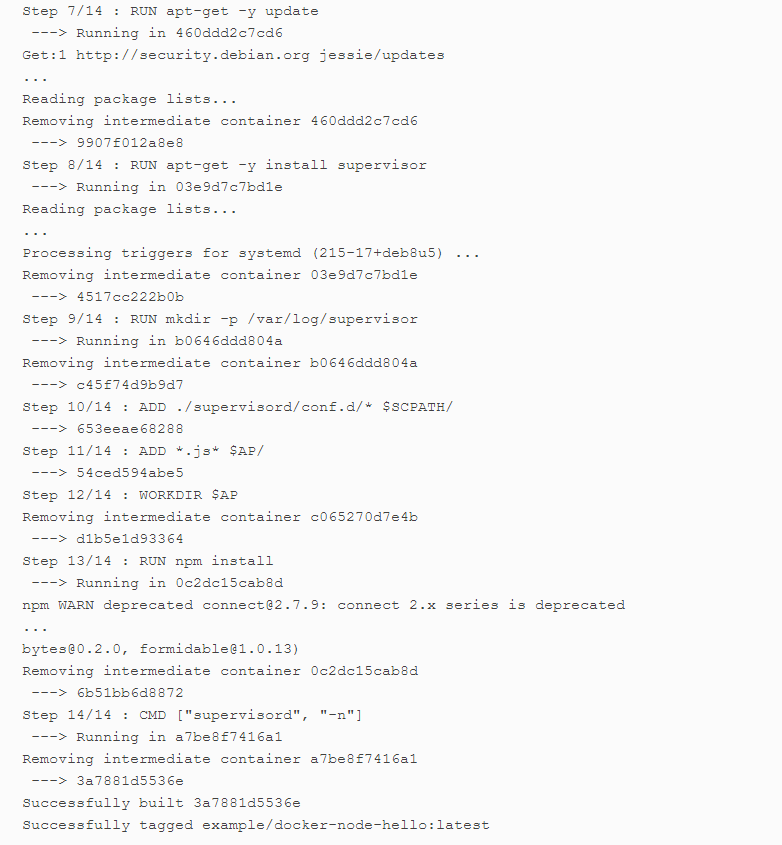
*Docker*



* *index.js and package.json will be used to build a Hello World Node APP*
* *Dockerfile is the important file which will spin up an image*
* *Supervisord – typically used to monitor the node app ( any container in fact)*
* docker build -t example/docke-hello:latest

All the instructions inside the dockerfile will be used to spin up





Notice each line in docker file is used for building the image ( order is very important and also each command builds a file system )

***Running the Docker Image – which typically starts a container in the docker host (dockerd)***

docker run -d -p 8080:8080 example/docker-node-hello:latest -e WHO="Sunil"

* This command spawns a docker container which listens in port 8080 and it is redirected to port 8080 of the Node APP
* Environment variables can be passed via doc-e option

***Image Storing repository***

***Public ( Like github)***

* [Docker](http://bit.ly/2N1GdRi) Hub and the [Quay](http://bit.ly/2Nw9UHj)

***Private***

* [Docker Trusted Registry](http://bit.ly/2N1GdRi) and the [Quay](http://bit.ly/2Nw9UHj) ( Enterprise version )

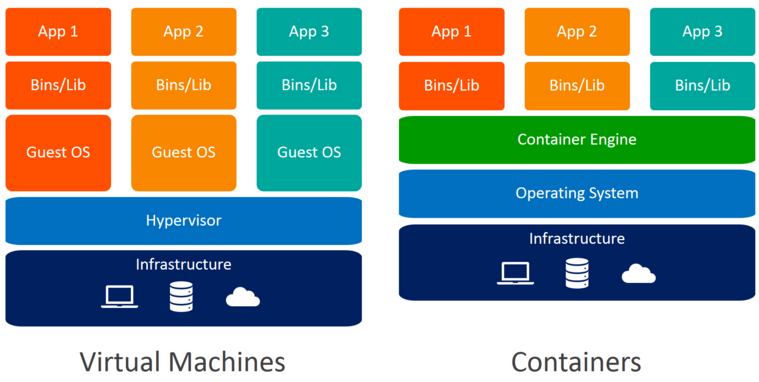
***Designing Images – Techniques***

*< refer book “Docker up and running” and other materials - care should be taken cared about the size of the container, necessary files >*

***Docker Container***

Kernel

( All Containers C1, C, C3 shares the same Kernel)



*Complete isolation because Shares the same Kernel. Hence, isolation is implemented using Kernel*

*Each VM(guest OS is a*

*Separate kernel) No need of one complete OS for a small task ( CPU and memory can be utilized well)*

***Container creation commands***

$ docker create --name="awesome-service" ubuntu:latest sleep 120

Name of the container else it will be some famous personality name

Ubuntu:latest 🡪 Image and its associated tag – latest

Docker create , docker start -🡪 docker run

$ docker run --rm -ti ubuntu:latest /bin/bash

The --rm argument tells Docker to delete the container when it exits, the -t argument tells Docker to allocate a pseudo-TTY, and the -i argument tells Docker that this is going to be an interactive session, and we want to keep STDIN open. The final argument in the command is the executable that we want to run within the container, which in this case is the ever-useful /bin/bash

***Setting Hostname for the container***

docker run --rm -ti --hostname="mycontainer.example.com" \

ubuntu:latest /bin/bash

***Explicit DNS setting***

docker run --rm -ti --dns=8.8.8.8 --dns=8.8.4.4

***MAC Address setting***

docker run --rm -ti --mac-address="a2:11:aa:22:bb:33" ubuntu:latest /bin/bash

***Mounting drives for containers***

$ docker run --rm -ti **-v** /mnt/session\_data:/data ubuntu:latest /bin/bash

root@0f887071000a:/# mount | grep data

/dev/sda9 on /data type ext4 (rw,relatime,data=ordered)

***Resource allocation – Update***

Use the docker container update command or deploy a new container with the adjustments

***CPU SHARES, CPU Pinning , CPU Quotas***

Go through the book

***MEMORY***

docker run --rm -ti --memory 512m progrium/stress \ --cpu 2 --io 1 --vm 2 --vm-bytes 128M --timeout 120s

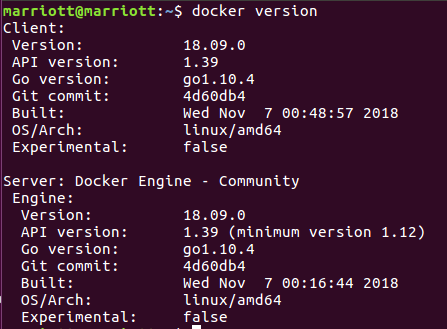
***BLOCK I/O***

Many containers are just stateless applications and won’t have a need for I/O restrictions. But Docker also supports limiting block I/O in a few different ways via the cgroups mechanism.

***ULIMITS***

***Exploring more useful Docker commands***

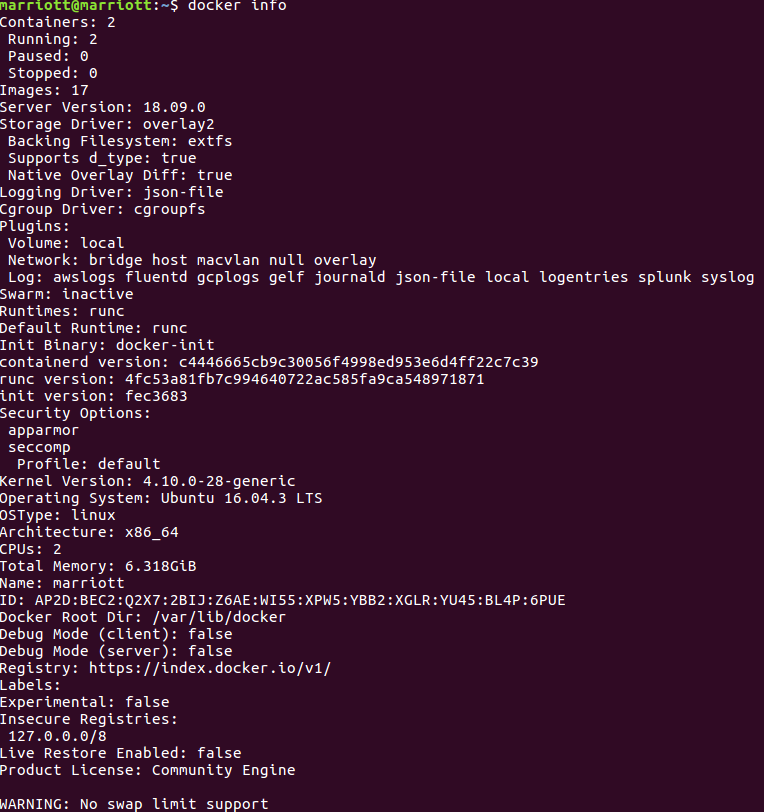
$ docker version



***Default Images & root directory***

In most installations**, /var/lib/docker** will be the default root directory used to store images and containers. If you need to change this, you can edit your Docker startup scripts to launch the daemon, with the --data-root argument pointing to a new storage location.

$ docker info



***Pulling the existing docker image from the repository***

$ docker pull ubuntu:latest

We’re going to use an Ubuntu base image for the following examples. Even if you have already grabbed the Ubuntu base image once, you can pull it again and it will automatically pick up any updates that have been published since you last ran it. That’s because latest is a tag that, by convention, is always moved to the most recent version of the image that has been published to the image registry

That command pulled down only the layers that have changed since we last ran the command. You might see a longer or shorter list, or even an empty list, depending on when you ran it and what changes have been pushed to the registry since then.

***Safest way to pull the image – because the hash is the entire image hash value***

docker pull ubuntu@sha256:2f9a...82cf

***Inspecting docker container***

$ docker inspect <Container ID>

Very useful information like hostname, command, args , image used, created time can all be fetched from here

***Starting the docker container in shell prompt***

$ docker run -i -t ubuntu:16.04 /bin/bash

-i denotes interactive,

/bin/bash 🡪 starts with bash shell

That will run an Ubuntu 16.04 LTS container with the bash shell as the top-level process. By specifying the16.04 tag, we can be sure to get a particular version of the image. So, when we start that container, what processes are running?

$ ps -ef

UID PID PPID C STIME TTY TIME CMD

root 1 0 0 22:12 ? 00:00:00 /bin/bash

root 12 1 0 22:16 ? 00:00:00 ps -ef

Wow, that’s not much, is it? It turns out that when we told docker to start bash, we didn’t get anything but that. We’re inside a whole Linux distribution image, but no other processes started for us automatically. We only got what we asked for. It’s good to keep that in mind going forward.

***Returning a result from a container***

Suppose create a container and execute some instructions inside that container and you expect the result in your terminal

$ docker run 8d12decc75fe /bin/cat /etc/passwd

root:x:0:0:root:/root:/bin/bash

bin:x:1:1:bin:/bin:/sbin/nologin

Here we executed /bin/false on the remote server, which will always exit with a status of 1. Notice how *docker* proxied that result to us in the local terminal. Just to prove that it returns other results, we also run/bin/true, which will always return a 0. And there it is.

Then we actually ask docker to run cat /etc/passwd on the remote container. What we get is a printout of the */etc/passwd* file contained inside that container’s filesystem. Because that’s just regular output on stdout, we can pipe it into local commands just like we would anything else

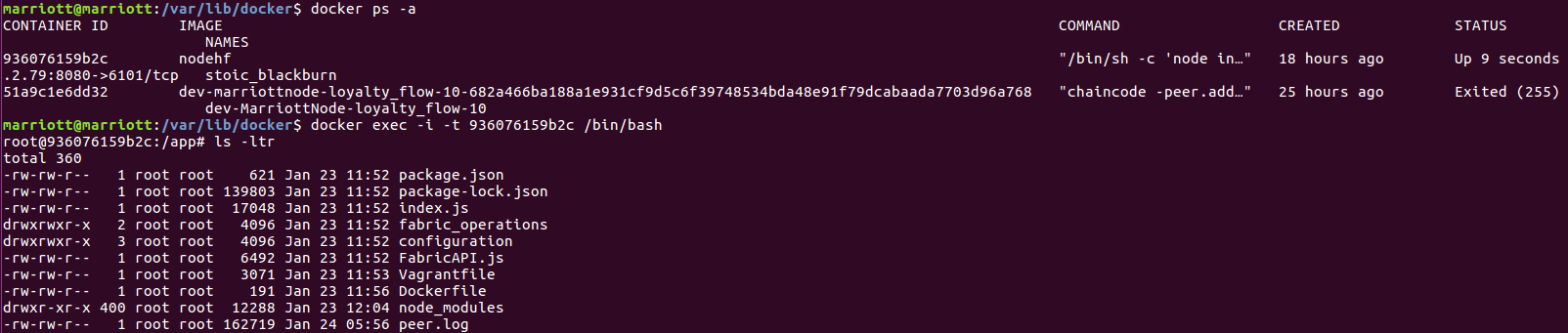
***Getting Inside a Running Container***

Two ways

* docker exec 🡪 Simpler and easy to use
* nsenter -🡪 Linux native way ( This can be used even if the dockerd is down)

In the below way, we are using the command to enter inside the container and looking at all the files present

$ docker exec –I –t <hash> /bin/bash



***List docker volumes – if something external has been bounded***

$ docker volume ls

***Docker Logging***

Docker makes logging easier in a few critical ways.

1. Anything sent to stdout or stderr in the container is captured by the Docker daemon
2. Other way is to use docker plugins

In fact, 1st method uses a default in built JSON plugin in docker server ( execute docker info command and look at the default plugin)

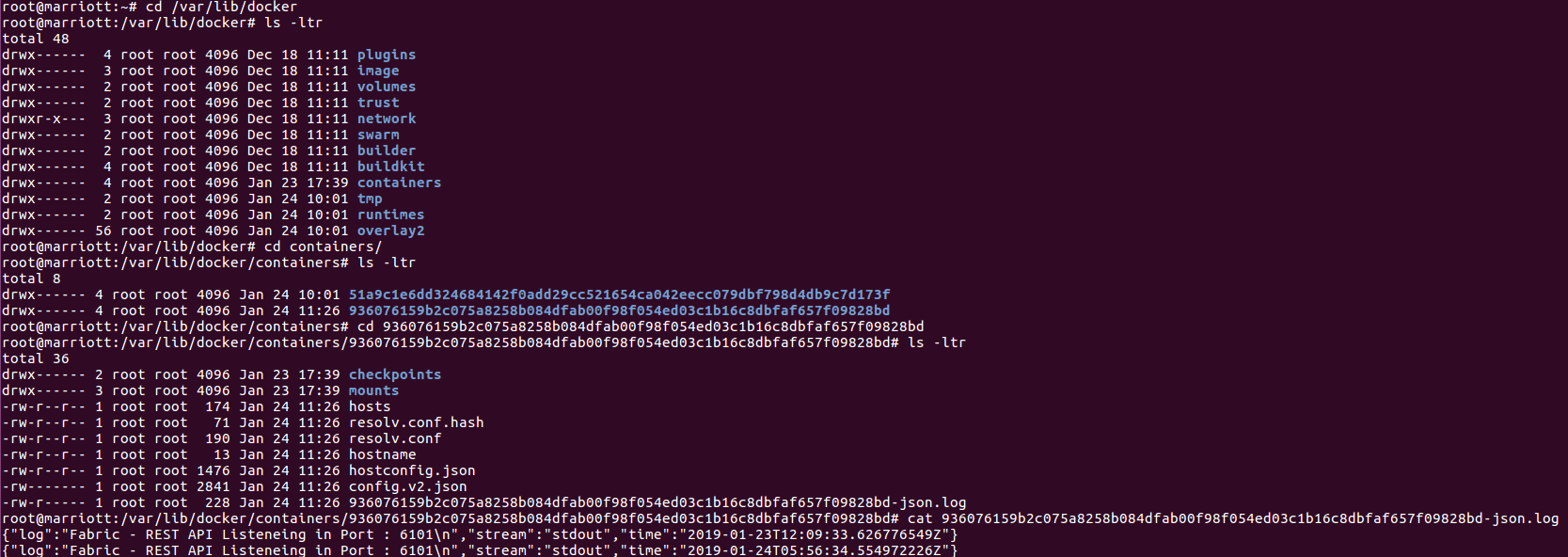
$ docker logs 3c4f916619a5

2017/11/20 00:34:56 [notice] 12#0: using the "epoll" ...

2017/11/20 00:34:56 [notice] 12#0: nginx/1.0.15

Docker logs in fact on the server itself

*/var/lib/docker/containers/*<container\_id>*/* where the <container\_id> is replaced by the actual container ID. If you take a look at one of those files, you’ll see it’s a file with each line representing a JSON object. It will look something like this:



***More Advanced Logging***

Refer book

***Docker Stats***

There are many ways to check and monitor the docker statistics but a simple way is used to docker stat command

$ docker stats <CONTAINER ID>

For more look and feel, stats refer google’s open source cAdvisor

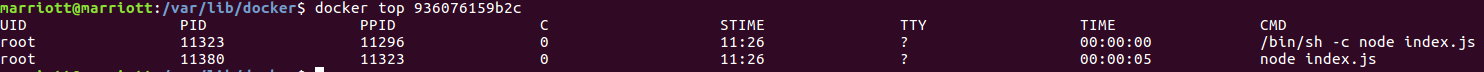
***Docker Health Check***

*Refer the book*

***Debugging Docker Containers***

$ docker top <CONTAINER ID>

docker top and get a nice listing of what is running inside our container, ordered by PID



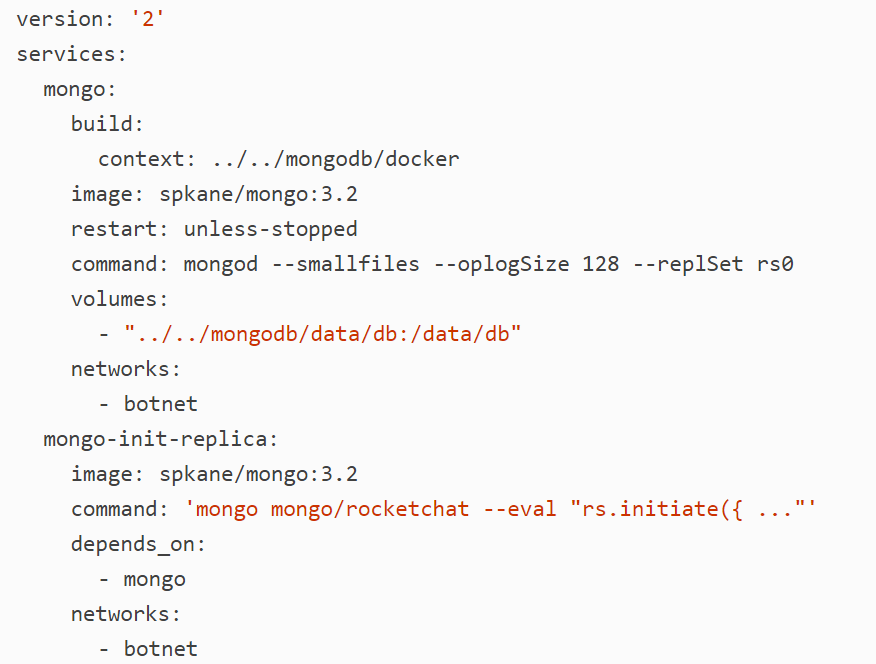
***Detailed Process output & Inspection , Monitoring Process***

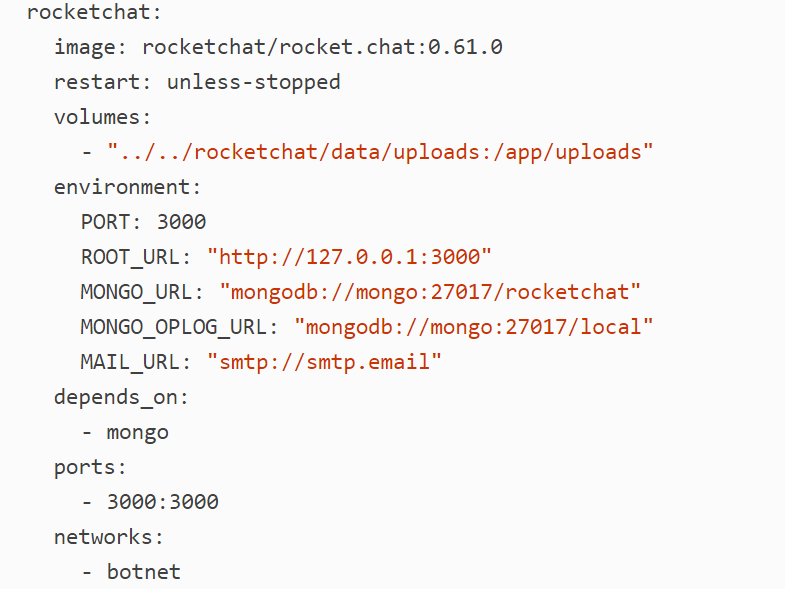
*Refer book*

***Docker Compose***

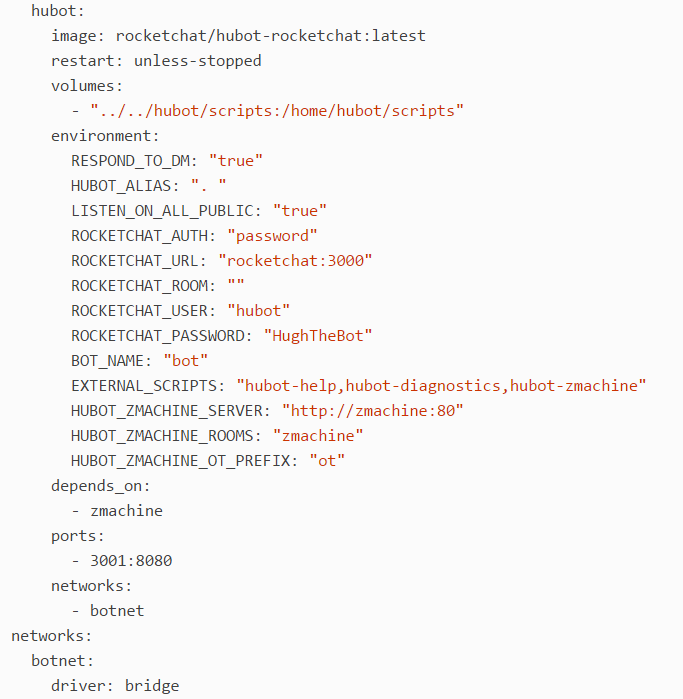
Configuration toll which is used to spin up multiple containers

Sample docker-compose.yaml file









***Decoding docker\_compose YAML***

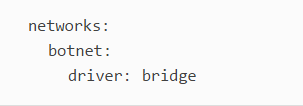


denotes the version definition

***Other Sections***

* Services
* Networks

***Networks section***



botnet 🡪 denotes single network and the name of the network is botnet

driver 🡪 bridge ( This is nothing but a bridge between the host network and the containers )

***Services section***

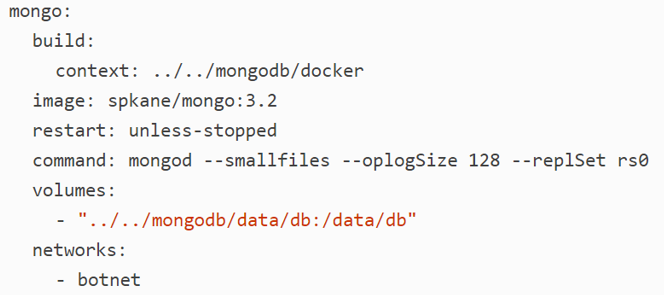
This section denotes what all services are needed as part of the container

There are five named services

* mongo
* mongo-init-replica,
* rocketchat
* zmachine
* hubot

Each named service then contains sections that tell Docker how to build, configure, and launch that service

***Mongo Service***



***Build*** – denotes build the necessary image using docker build ( Files necessary are present in the context location mentioned )

< This step is not mandatory if you already have the image>

***Image*** – Name of the image for the build

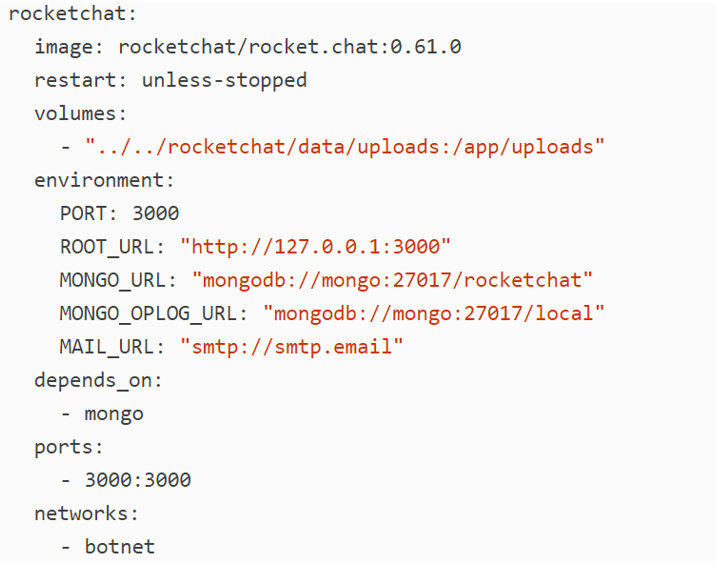
***Restart*** – restart the container unless it is stopped state

***Command –*** Command which needs to be executed during the startup procedure of the container

***Volumes -*** Data that needs to be persisted during development like files etc

***Networks –*** Which network this one should belong to ?

***Rocket Chat Service***



As like above except

***Environment –*** Which denotes all the environment variables

***Depends on –*** denotes mongo service should be running before starting rocket chat service

***Ports –*** 3000 : 3000

docker daemon port 🡪 docker container port



***Expose –*** 80 🡪 This section allows us to tell Docker that we want to expose this port to the other containers on the Docker network, but not to the underlying host. This is why you do not provide a host port to map this port to.

You might notice at this point that, while we expose a port for zmachine, we didn’t expose a port in the mongo service. It wouldn’t have hurt anything to expose the mongo port, but we didn’t need to because it is already exposed by the upstream [mongo *Dockerfile*](http://bit.ly/2okkFBn).

***With this docker-compose YAML configuration is complete***

***Launching Services using Docker Compose Command***

***Important –*** While running docker compose command, make sure docker-compose.YAML is in the same directory from where you are running

***Command to inspect docker compose file***

$ docker-compose config

***Command – Builds a image by inspecting the docker compose file***

$ docker-compose build

***Command – to Bring up all the services***

$ docker-compose up –d

Check the services, status using $ docker-compose logs