Docker Toolbox IP address : 192.168.99.100

**Docker Command Format:**

docker <command> <sub-command> <options> Image [COMMAND] [ARG..]

**Image vs Container:**

An Image the application we want to run. It is the Binary, Library and Source Code that all make up our application.

A Container is an instance of that image running as a Process.

We can have many containers running off the same image.

Docker’s default image “registry” is called Docker Hub (hub.docker.com)

**docker container run –-publish 80:80 nginx**

**docker container run –-publish 80:80 –-detach nginx (**run in background**)**

**docker container run –-publish 80:80 –-detach –-name pankaj nginx (**give container name by ourself**)**

1. Download image ‘nginx’ from Docker Hub
2. Started a new container from that image
3. Opened port 80 on the host IP
4. Routes that traffic to the container IP, port 80. (You will get a “bind” error if the left number [host port] is being used by anything else, even another container. You can use any port you want on the left, like 8080:80 or 8888:80, then use localhost:8888 while testing)

**docker container ls**

Show all running container

**docker container ls -a**

Show all container

**docker container stop <container ID>**

Stop running container

**Run vs Start**

Run always start a new Container

Start always start an existing Stopped Container

**docker container logs <container-name>**

To see logs of container

**docker container top < container-name>**

Display running processes of a Container

**docker container –-help**

See all command

**docker container rm <container-id> <container-id> <container-id> <container-id> <container-id>**

Remove docker Container normally. This command can delete only stopped container, it can’t delete running container.

**docker container rm -f <container-id> <container-id> <container-id> <container-id> <container-id>**

Remove docker Container Forcefully.

**What happens in “docker container run”**

1. Looks for that image locally in image cache, doesn’t find anything
2. Then looks in remote image repository (default to Docker Hub)
3. Download the latest version (nginx: latest by default)
4. Creates new container based on that image and prepare to start
5. Gives it a virtual IP on a private network inside docker engine
6. Opens up port on host and forwards to port in container
7. Starts container by using the CMD in the image Dockerfile

**Container vs VM**

1. Containers aren’t Mini-VM’s
2. They are just process
3. Limited to what resources the can access
4. Exit when process Stops

**ps aux**

Show all running process

**ps aux | grep ‘pan’**

Show all running process and filter also

**docker container run –d –p 3306:3306 –name db –e MYSQL\_RANDOM\_ROOT\_PASSWORD=yes mysql**

Run MYSQL container on port 3306 and setup the Environment path also

**docker container inspect mysql**

show metadata about the container(startup config, volumes, networking, etc)

**docker container stats**

Show live performance data for all containers

**docker container stats mysql**

Show live performance data of “mysql” container

**docker container run –t**

Simulate a real terminal, like what SSH does

**docker container run –i**

Keep session open to receive terminal input, it’s like interactive mode

**docker container run –it**

Start new container interactively

**docker container run –it –-name proxy nginx bash**

Get a shell inside Container without using SSH

**docker container exec**

Run additional process in running process

**docker container exec –it mysql bash**

Get a shell inside a running container Container like MYSQL. If we exit from this command, then see that our MYSQL container would be running, because bash was an additional process in running container MYSQL.

**docker container port <container>**

Quick port check

**Docker Networks Defaults**

Each Container connected to a private virtual network “Bridge”

Each virtual network routes through NAT firewall on host IP

All containers on a virtual network can talk to each other without –p

Best practice is to create a new virtual network for each app:

Network “web\_app” for mysql and php/apache containers

Network “my\_api” for mongo and nodejs containers

Batteries included, but Removable

Defaults work well in many cases, but easy to swap out parts to customize it

Make new virtual networks

Attach containers to more then one virtual network (or none)

Skip virtual networks and use host IP (--net=host)

Use different Docker network drivers to gain new abilities

**docker**  **container run –p 80:80**

-p is publish. Publishing port is always in HOST:CONTAINER format

**docker container inspect –-format “{{ .NetworkSettings.IPAddress }}” <container name>**

Find IP of running container

**IP of host and IP of container will not be same**

**docker network ls**

Show Networks

**docker network inspect <network name>**

Inspect a network. Will show attached container, IP, gateway to this network

**docker network create <network name>**

Create a virtual network for you to attach containers to. Bridge is the default network driver.

**docker network connect <network id> <container id>**

Attach a network to container. Dynamically creates a NIC in a controller on an existing virtual network.

**docker network disconnect connect <network id> <container id>**

Detach a network from container. Dynamically removes a NIC from a container on a specific virtual network.

**--network host**

It gain performance by skipping virtual networks but sacrifices security of container model

**--network none**

Remove eth0 and only leaves you with localhost interface in container

**Network driver**

Built-in or 3rd party extension that give you virtual network features.

**docker container run –d –-name new\_nginx –-network <network name> nginx**

Run nginx container and attach to the network named as <network name>

**Docker Networks: Default Security**

Create you apps so frontend/backend sit on same Docker network

Their inter-communication never leaves host

All externally exposed ports closed by default

You must manually expose via –p, which is better default security.

This gets even better later with Swarm and Overlay Networks.

**Docker DNS**

Docker daemon has a built-in DNS server that containers use by default.

Docker defaults the hostname to the container’s name, but you can also set alias

**Section 4**

**Image**

An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime.

It is app binaries and dependencies.

It is not a complete OS. No Kernel, kernel module. Because host machine provides Kernel.

**Image and their layers**

Images are made up of file system changes and metadata

Each layer is uniquely identified and only stored once on a host

This saves storage space on host and transfer time on push/pull

A container is just a single read/write layer on top of image

**docker image inspect <image name>**

Returns JSON metadata about the image.

**docker image history <image name>**

Show layers of changes made in image

**docker image tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]**

Assign one or more tag to an image.

If we will not specify TAG, then it will take default TAG. Default TAG is LATEST

**docker image push IMAGE\_NAME[:TAG]**

Uploads changed layers to image registry (default is docker hub)

**docker login <server>**

Defaults to logging in Hub, but you can override by adding server url. To push image, we must login

**docker logout**

Always logout from shared machines or servers when done, to protect your account.

To create a image, we have to make a docker file. The name of that file will be “Dockerfile”.

**docker build –f <file name>**

Create a docker file through command line and give another name apart from default name.

**docker image build –t <new image name> <Dockerfile path>**

Build image by using Dockerfile. It will run each command from Dockerfile and build image.

**Section 5**

**docker volume create**

Required to do this before “docker container run” to use custom drivers and lables

**Persistent Data: Bind Mounting**

Maps a host file or directory to a container file or directory

Basically just two locations pointing to the same file

Again, skip UFS, and host file overwrite any in container

Can’t use in Docerfile, must be in “container run”

… run –v /users/bret/stuff:/path/container (mac/linux)

… run –v //c:/users/bret/stuff:/path/container (windows)

**Section 6**

**Docker-Compose**

Why: configure relationships b/w containers

Why: Save our docker container run settings in easy-to-read file

Why: Create on-liner developer environment startups

Comprised of 2 separate but related things

1. YAML- formatted file that describes our solution options for :
   1. Containers
   2. Networks
   3. Volumes
2. A CLI tool docker-compose used for local dev/test automation with those YAML files

Docker-compose is talking to docker API in the background on behalf of docker CLI.

**docker-compose.yml**

Compose YAML format has it’s own version: 1, 2, 2.1, 3, 3.1

YAML file can be used with docker-compose command for local docker automation or..

With docker directly in production with Swarm

docker-compose –-help

docker-compose.yml is default filename, but any can be used with docker-compose –f

**docker-compose CLI**

CLI tool comes with Docker for Windows/Mac, but separate download for linux

Not a production-grade tool but ideal for local dev and test

Two most commons are

“docker-compose up“ #setup volumes/network and start all containers

“docker-compose down“ # stop all containers and remove cont/vol/net

If all your project had a “Dockerfile” and “docker-compose.yml” then “new developer onboarding” would be:

git clone github.com/some/software

docker-compose up

**Section 7: Swarm intro**

**Swarm Mode: Built-In Orchestration**

Swarm Mode is a clustering solution built inside Docker

Not related to Swarm “classic” for pre-1.12 versions

Added in 1.12 via SwarmKit toolkit

Enhances in 1.13 via Stacks and Secrets

Not enabled by default, new commands once enabled

1. docker swarm
2. docker node
3. docker service
4. docker stack
5. docker secret

**docker swarm init**

Initialize Swarm

**Section 8: Swarm Basic features and how to use them in workflow**

**Overlay multi-host networking**

Just choose **--driver overlay** when creating network

For container-to-container traffic inside a single Swarm

Optional IPSec(AES) encryption on network creation

Each service can be connected to multiple networks (like: backend, frontend)

**Routing Mesh**

Routes ingress(incoming) packets for a Service to proper Task

Spans all nodes in Swarm

Uses IPVS from Linux Kernel

Load balances Swarm Services across their tasks

Two ways this works:

1. Container-to-Container in a Overlay network (use VIP)
2. External traffic incoming to published ports (all node listen)

**Docker swarm join-token worker**

Get token for worker from master host to join worker, if you lost token.

**Docker swarm join-token manager**

Get Token for manager from master host to make a system as master

**Docker swarm leave**

Run this command on worker to leave the worker from swarm cluster

**Docker node rm -f <hostname/hostid>**

Run this command on manager to remove a worker from the cluster.

**Docker node inspect <nodename/nodeid>**

Show all info of a node

**Docker node promote <worker1name, worker2name , worker3name, ….>**

Run this command on manager to promote specified worker as manager.

**Docker node demote < manager1name, manager2name , manager3name, ….>**

Run this command on manager to demote specified manager as worker.

**Docker service create –-replicas 4 <image name>**

Create container in swarm cluster. This command will run only on manager/master. –-replica will create specified number of replicas.

**Docker service ls**

It will list created services.

**Docker service inspect <service id>**

Inspect a service

**Docker service logs**

Display logs of service

**Docker service scale <service id>=no\_of\_replicas**

This command used to create number of replicas of a service.

**--mode=global**

This option is specified with “docker service create” command. It will create instance of the service on each node. If we add a node later in swarm cluster, then also it will automatically create an instance of that service on newly added node.

**--constraint=”node.role==manager”**

This option is specified with “docker service create”. If we want to create service only on the nodes which are “master/manager”, then we will use this command.

**--constraint=”node.role==worker”**

This option is specified with “docker service create”. If we want to create service only on the nodes which are “slave/worker”, then we will use this command.

**Docker node update –-label-add=” custom\_label\_name=custom\_label\_value” <worker name>**

This command is used to create custom label on a worker. We should define these custom label to all the workers before we start any services. Suppose we create a label as “ssh=true” and created a service with replica 8. It will create all replicas on node which has label as “ssh=true”. If we add “ssh=true” label to another worker then docker will not distribute the task between these two nodes, which has label name as “ssh=true”.

This command will run only on master/manager.

**--constraint=”node.labels.custom\_label\_name==custom\_label\_value”**

This option is specified with “docker service create”. Here we can specify condition according to custom label. If we specify “ssh==true”, means this service will be created only to the nodes where custom label name is “ssh” and it’s value is “true”.

This is **Node label.**

There are 2 ways to create custom label in node:

1. We have already seen using command. This is called as **Node label**
2. Another one is called as **Engine label.** Create a file on worker system in “/etc/docker/” path with the name as “daemon.json” with value as

{“labels”: [“label\_name=label\_value”]}

After creating this file, restart the service on worker system using “service docker restart”

**--constraint=”engine.labels.custom\_label\_name==custom\_label\_value”**

This option is specified with “docker service create”. Here we can specify condition according to custom label. If we specify “ssh==true”, means this service will be created only to the nodes where custom label name is “ssh” and it’s value is “true”.

This is **Engine label.**

Engine label are used because sometime in production we don’t have permission to use Master node. And Node label are created using Master node only. So in production, we will create Engine label.