DOCKER Tutorials

VMs are nothing but a physical server being divided in separate VMs which act as independent servers for us, this helps in better resource utilization of the physical server and it is highly cost optimized.

Issue with VMs –

* OS exists to facilitate the application
* OS consumes CPU RAM and Disk space
* In a physical machine we can have multiple VMs, each VM can host one application, each application needs a full blown OS.
* OS may also need individual licensing
* Each VM or OS acts as a security boundary

Containers –

* are a bit like VMs, they are application runtime environment
* way lighter than VMs, they consume less CPU, RAM or disk space
* on one physical machine, we have one OS installed on it, on top of that we can create multiple containers, each container can then work as an instance, independent of other containers and we can install an app on it.
* An OS container has a lightweight OS
* Containers are faster and more portable than VMs
* One container should be able to have its own root file system, its own process tree ie; each container should be able to have its own PID 0 and so on
* The PID tree isolation can be achieved by kernel namespaces, it helps us in isolation of PID tree, networking, file system mounts, user namespaces /accounts
* Container also uses cgroups or the control groups, it helps in grouping together resources and then isolate with each other; so we will have one cgroup mapped to one container and we can set limits on how much CPU, memory, block IO, that the container has access to.
* Docker supports cgroups, if we wish to have multiple containers on same system we should have a system that supports cgroup in order to manage server resource distributions.

Docker --

* It’s a runtime container
* Using docker, we create containers that can have code, application or anything and once that container is created it can run on any system with any platform as far as docker is installed on it.
* Libcontainer are the prerequisites for docker installation, they are controlled by docker themselves,
* Container shares a common kernel can run on machines with that kernel, so a unix container cannot run on a windows machine and vice versa.
* Containers encourages a micro services architecture, so we can have each of the services running in its own containers then a lot of these small containers together can form a package, which then can act as one container.
* Docker is a client server architecture wherein docker client sends commands to docker demon running on a remote machine, demon creates the containers or does the actual working for docker
* Client and the demon comes in one package so we can run client and demon on the same server.

Docker installation on Ubuntu:

#Docker installation guide for centOS: https://docs.docker.com/engine/installation/linux/centos/

* Sudo su
* Cd
* Check if docker is running on system
  + Service docker.io status
    - Why we have this Io in the name is because Ubuntu already had a package named docker
* Uname -a #to check the version of Ubuntu installed
* Apt-get update #to sync up already running packages from registered repository
* Apt-get install -y docker.io
* Service docker.io status

Now here docker client and the demon both are available and we have access to both

* Docker -v #this will tell the version of the docker demon, so firing this is like demon is responding to the client
* Docker version #this will give details of the client and demon both running on the VM
* Docker info #it will give details of images and containers we have in docker

Updating docker installation

# one must backup data before doing the upgrade, backup all the containers and images created in docker and the file system.

* Adding the docker repo key to the local apt key chain of our system
  + Wget -q0- [https://get.docker.com/gpg | apt-key](https://get.docker.com/gpg%20|%20apt-key) add -
* Echo deb <https://get.docker.com/ubuntu> docker main > /etc/apt/sources.list.d/docker.list #this is adding docker repo to our apt sources.
* Apt-get update
* Apt-get install lxc-docker #press yes when it asks for, if an updated version is available to update
* Docker version
  + Docker needs root, to work
  + Docker run -it Ubuntu /bin/bash
    - -I interactive
    - -t is make it tty
    - This is command to create basic containers
    - Ubuntu is to say make it an Ubuntu container
    - This command will be throw error of permission denied if run with any user other than root, as we do not always want to be root we need to give users permissions to docker group, as the docker.sock file in /var/run is group owned by docker group
  + Checking the docker group:
    - Check if group exists: cat /etc/group
    - Add user to the group if it exists, sudo gpasswd -a mgupta docker
      * This will add (-a) mgupta to docker group
    - Now running the container create command after logging out and back in
      * Once container is created, it takes you directly in the container
      * Every container is given a unique ID
      * Type “exit” to come back to the Ubuntu command line, It also terminates the container ie makes the container to be in stop state

Configuring docker to communicate over network

* Check version: docker -v
  + 1.11.2
* Ifconfig details
  + Ubuntu-2: 172.17.0.1
* Docker service stop
  + Service docker stop/start
* Did not run Docker -H <ip address>:2375 -d & #2375 is the default docker port
  + Setup port on first machine using the command mentioned above. Then set DOCKER\_HOST on the other machine to talk to your first machine
* Export DOCKER\_HOST=”tcp://<ip add>:2375”

Docker commands now:

* + Create a container: Docker run -it Ubuntu /bin/bash
  + Check docker processes: docker ps
  + Check all docker process including stop: docker ps -a
  + Start the docker container in stopped state: docker start <short container ID>
  + Docker attach <container short ID> #why do you need to attach a container, when we start container on a machine the container is initialized but the system is not yet aware that it needs to start using this container or behave like one, so to make it understand that we need to attach the container
  + Exit, while you are in the container will stop the container
  + Once your container is stopped you can still see the files of your container in: ls -l /var/lib/docker/aufs/diff/docker full id
    - Docker full iD is docker short container ID + some more characters, list can be found at /var/lib/docker/aufs/diff
    - Go here, match the short container ID with the starting phase of the long container IDs mentioned there.
  + Docker pull -a Ubuntu/fedora/centOS #this command will pull **all** the available images of Ubuntu
    - Once downloaded these images are stored on /var/lib/docker/aufs (or any storage driver depending on the container)
  + Docker images Ubuntu/fedora/centOS #this will give you list of all the images that are downloaded on your system
    - We can have one container image comprising of multiple container images inside of it, for example an Ubuntu image be used and then ngnix is installed on it that becomes image 2, now in this we apply patches and that becomes our image 3. Now they will all have their configuration files and mismatching configs, in this case the config file on the top most container image that is image 3 in our case will be considered.
    - These multiple images are brought together using union mount using which it will mount all the file systems on top of each other and not like one is mounted on root and other is mounted on any secondary file system or mount point.
    - Images can be listed in /var/lib/docker/aufs/layers
      * In these images you can cat the content of any of the listed images and they will list the underlying images it is created from.
  + Docker run Ubuntu /bin/bash -c “echo ‘cool-content’ > /tmp/cool-file”
    - If we wish to run a specific version of Ubuntu we will write docker run Ubuntu: 14.04 or whatever version we want, then rest of the command
    - If we want a container to run in the background, we will run “docker run -d Ubuntu”, this will fire a container and it will run in the background until the processing within the container is going on.
    - Docker run –cpu-shares=256 #to control how many cpu shares this container gets
    - Docker run memory=1g
    - Docker run -d Ubuntu:14.04.1 -c “ping 8.8.8.8” #here “-c” is to specify the command it needs to run once the container is up
    - Docker inspect <container ID> #this command gives all the details about the container
  + Docker commit <container ID> fridge #this creates new image from the change that we have done on the existing image and FRIDGE here is the name of the image that we have created.
  + Docker history <image-name> #this will tell all the commands ran on the image, all the images or container it is part of
  + Docker save -o /tmp/fridge.tar fridge
    - This will save the image in the tar file, if we do not specify “.tar” then also docker by default creates a tar file
    - Now this tar file can be transferred to any of the systems that have docker on it.
  + Once we have the image on the system, to load the image we run: docker load -i /tmp/fridge.tar
    - Once you do this you can run “docker images” to see the image you have just added being listed
    - After this fire a run command “docker run -it fridge /bin/bash to load container from your image.
  + Docker top <container id> #it shows the top running commands inside of our containers

Docker containers, images, docker engine, registry and repositories

* + Docker engine is docker demon also called docker runtime
  + Docker image is used to launch a docker container

Container management:

* + Creating/running a new container:
    - Docker run -it Ubuntu:14.04 /bin/bash
      * This was created as an interactive container so when it started we directly got attached to this container and opened for us to use
    - Exit; will stop the container and bring you back to the shell prompt. In order to come out of container without stopping it we can fire control + P +Q . this is referred to as detaching from a container.
    - Docker start <container id>
    - Docker attach <container ID> #this command attaches us to the first process running in the container (PID 1) so if we wish to have a shell prompt for our container we need to have bash/sh or any shell running as the PID 1 in our container.
    - Docker stop <container ID/name>
    - Docker kill -s <signal> #using this we can send any kill signal to the process ID 1 of the container. Actually when we fire a “docker stop” command it sends a kill -1 signal to the PID 1 of the container, using which the processes running inside container are killed thus making the container stop as when the process inside container is completed or killed the container is functioned to stop.
    - Docker restart <container ID/name>
    - Docker rm <container ID/name> #to remove a container; you cannot remove a running container
    - Docker rm -f <container id> #to delete a running container
    - Docker top <container id> #list of all the processes running on the container
    - Docker inspect container-id/name or even image-name
    - Having a shell prompt in the container if it doesn’t already have it:
      * Find out the process ID of the container on your system
        + Docker inspect container-id |grep Pid
        + Nsenter -m -u -n -p -t -i Pid

-m: mount namespace

-u: uts namespace

-n: network namespace

-p: process namespace

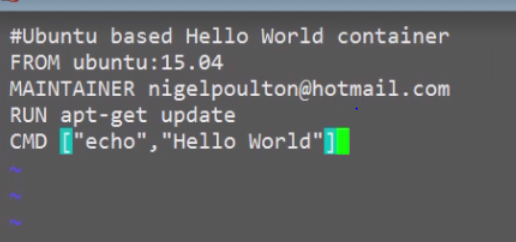
-i: ipc namespace

* + - * + Docker-enter container-id #this is a docker command for nsenter and takes us directly to container shell, also if we “exit” from this shell the container does not stop
        + Docker exec -it container-id /bin/bash #this is recommended way of having a terminal in the container
        + To install these modules in the container

Docker run -v /usr/local/bin: /target jpetazzo/nsenter

Dockerfile

* it is spelled as “Dockerfile”
* docker build
* Dockerfile is a plain text file and it can be created anywhere
* any line with “#” is treated as comment line
* FROM instruction is the first instruction in the docker files, it tells us which image are we going to use to build this container from
* MAINTAINER person who is going to create the dockerfile so [mansi.gupta@gmail.com](mailto:mansi.gupta@gmail.com)
* RUN apt-get update or any command #each run commands adds a new layer to our image, so it is preferred to use minimum run commands to have less layers
* CMD [“echo”,”Hello World”] #thr could be only one CMD in the dockerfile, if we have multiple the last one will get executed ignoring all other . This CMD works as run command or any command that can be run on the server once it has come up.
* VOLUME /data #this we usually write after the run command and it is used to create volumes in the containers
* When we create a dockerfile, any other file in the same directory will get included in the docker build
* Number of layers in an image is restricted to 127 for any image, so we must try to reduce the number of layers we create in our dockerfile. Every instruction we write in the Dockerfile acts as a layer for the image.
* Docker built -t image-name:version <directory path>
  + This will build the docker image and gives you the ID in the end
  + Once you have the image you can create containers from that image using docker run image-id
* EXPOSE 80 #this will ensure port 80 is exposed to any container created from the image of this dockerfile
* Docker built
* ENTRYPOINT [“echo”] #now when we create a container from the image (X) created from the Dockerfile here the container is created as “docker run X Hello Mansi!” it will return Hello Mansi! As the entrypoint will give this value to “echo” command mentioned in the Dockerfile
* ENV var1=Mansi var2=Gupta # this can be used to specify variables while creating an image
* Sample Dockerfile:





Working with registries:

* + Registries 🡪 repos
  + In order to push an image to the docker hub, hub.docker.com you need to create an account
  + Once account is created you need to create a new repository using the hub.docker.com account
  + Then you need to tag the image you have created as:
    - Docker tag <image-id> mgupta/repo-name:1.0 #here mgupta is docker hub account and then the repo created on the docker hub account
  + Then you can upload the image:
    - Docker push mgupta/repo-name:1.0
  + In order to delete an image, we need to delete the containers connected to those images
    - So first fire a “docker rm container-ids”
    - “docker rmi <images-ids>” #this is to delete the images
  + Pulling image from docker hub:
    - Docker pull mgupta/image-name:1.0
  + Pulling the registry image locally to our system:
    - Docker run -d -p 5000:5000 registry #this will create our own private registry on docker hub
      * Once it is created, we can push our images to it to be able to download it later.

Volumes in Docker:

* Used to decouple data from containers and also to share same data between multiple containers
* Create volume based containers:
  + Docker run -it -v /test-vol –name=voltainer Ubuntu:15.04 /bin/bash
* Create a container that can use this volume:
  + Docker run -it --volumes-from=voltainer –name=voltainer2 Ubuntu:15.04 /bin/bash
* Using volumes we can have one file system available to all the containers, we want to be
* Delete volumes, it can only be done from within the container
  + Docker rm -v <container>
    - Using this container and the volume gets deleted, if we do not specify ‘-v” then the volume doesn’t get deleted

Docker troubleshooting:

* Starting the docker demon and specify the log levels while we do that:
  + Service docker stop
  + Docker -d -l debug &
    - “-d” is used to specify it’s the demon we are running
    - “-l” to specify log levels: debug, info, error, fatal
    - Once we do that , for ever command we fire we start to see the logs on screen
    - To avoid that we can redirect the logs to a file:
      * Docker -d >> <file> 2>&1
    - Vim /etc/default/docker
      * Here come to the bottom and specify:
        + DOCKER\_OPTS=”—log-level=fatal”
        + Once this is done, we need to restart the service
* Docker logs:
  + Docker logs <container-id/name>
  + Docker logs <container-d> -f #continues logs
  + One way to have the logs being available to other containers is having the logs written to a common volume which can be accessed in other containers for monitoring purposes.
* Testing the Dockerfile
  + Spin a container using the base image we wish to use in Dockerfile and then run all the commands inside the container to see how it works
  + Another way is to run the Dockerfile to spin up and image, as the dockerfile is executed line by line it will spin up an image until it encounters an error. Once image is created it will have no name or tag, use that image to spin up a container. Once you have the container and the dockerfile you can check each command to see if it has run or where it got stuck.

DOCKER SWARM:

* Service used to manage large number of docker containers
* SWARM is the native clustering of the docker, ie it is created by docker community, keeping docker containers in mind.
* It is for clustering of docker engines and demons
* Every swarm cluster needs a discovery manager, its mandatory.
  + This discover service keeps track of the state and config of the cluster
* Also the cluster needs Swarm manager, in order to manage the cluster
* It has filters and schedulers – so when we need to have a new container, it find the list of containers that matches the requirement and then picks one instance from it to schedule the creation of a new container from it.
* Filter currently has these filters: affinity, constraint, resource
* Scheduling is also of three kinds: random, spread(default), binpack
  + Random: randomly selects the node on which it can create the new container
  + Spread: it selects the node which has the least containers and then creates a new container on that node.
  + Binpack: its opposite of spread, it creates container on the node which has most containers. This is useful to ensure the least resources are being used.
  + These scheduling takes ram and cpu in consideration while making decision, so if we have nodes with varied CPU and RAM, the binpack will start creating containers on the smallest available node.
* Swarm does not have any defined ports for communication so any port can be used, people usually user 4000/4001 or 3375/3376 for insecure and secure.
* Port between swarm manager and docker demon is 2375/2376(TLS/SSL)