Docker:

* Released in 2013 as an open source project.
  + Mainframe to PC in 90s
  + Baremetal to Virtual in 00s
  + Datacenter to Cloud in 10s
  + Host to Containers (serverless)
* Containers are the fastest growing cloud enabling technology. By 2020, more than 50 % of global organizations will be running containers in production.
* Docker is all about speed.
  + Develop, build, test, deploy, update, recover – all can be faster
* <https://github.com/BretFisher/udemy-docker-mastery>

Installation:

* Docker has Community edition CE (free) and Enterprise Edition EE (paid)
* Docker release edge version like Beta that changes every month
* And also stable version which comes every quarter.
* EE version will have longer support and features.
* We have a script from docker to installer automatically based on our OS.
* <https://get.docker.com/> open this and run the curl command mentioned in the page.
* curl -fsSL https://get.docker.com -o get-docker.sh
* sh get-docker.sh
* this doesn’t work on Amazon Linux distribution.
* Sudo yum update –y
* Sudo yum install docker –y
* Sudo service docker status
* Sudo usermod –a –G docker ec2-user
* Docker –v
* To install docker machine, go through the below link.
  + <https://github.com/docker/machine/releases>
* To install docker compose, go through the below link.
  + <https://github.com/docker/compose/releases>

A registry is a place where we store our images. You can host your own registry or use docker’s public registry called DockerHub.

Inside a registry, images are stored in a repository. Docker repository is a collection of different docker images with the same name, that have different tags, each tag usually represents a different version of image.

Docker Simple commands:

* Docker version
* Docker info
* Docker
* In the above command output, we can see the management commands. It’s the new way of organizing all the commands.
* The format is “docker management-cmd subcommand”
* Example is “docker container run” and it was previously running as “docker run”
* Docker Imange is the application that we want to run
* A container is an instance of image running as a process.
* You can have many containers running off the same image.
* Docker’s default image registry is called as Docker hub.
* Docker container run –publish 80:80 nginx
* Here it will first download the image for nginx and then start a container with that image.
* After the above command, try to access the nginx server at localhost or public IP or that instance where docker is running.
* Here
  + Downloaded image nginx from docker hub
  + Started a new container from that image.
  + Opened port 80 on the host IP.
  + Routes that traffic to the container IP, port 80
* To run this container in the backgroup
  + Docker container run –publish 80:80 –detach nginx
  + This will give you an unix number for your container
* To list the containers
  + Docker container ls
  + Docker ps
* Docker container stop container-id
  + Docker container stop 3d0
  + First 3 unique digits of container id is enough
* Docker ps or docker container ls will show only running containers.
* To show all the containers “docker ps –a” or docker container ls –a
* Next time, instead of running, we can start the existing container.
* Docker container id and Name are unique
* You can also specify the name of docker container while running.
  + Docker container run –publish 80:80 –detach –name webhost nginx
* Docker container logs “container-name”
  + This will display the logs for that container
* Docker container top webhost
  + To see info of running container
* Docker container –help
* Docker container rm id’s
  + Docker container rm 3d0 d94
  + Here we are deleting two containers
* You can’t delete a running container without stopping.
  + Ofcourse you can force delete
  + Docker container rm –f 3d0

What happens in background for docker run:

1. Looks for image locally in image cache, doesn’t find anything
2. Then looks for image in remote repository which is docker hub
3. Downloads the latest version if you don’t specify any version (nginx:latest )
4. Creates a new container based on that image and prepares to start
5. Give it a virtual ip on the private network inside docker engine
6. Opens up port 80 on host and forwards to port 80 on container.
7. Start the container by using CMD in the image dockerfile.

Docker containers are just a process:

* Docker top “container-id” will give the top process running on the container
* Docker container run –name mongo –d mongo
  + Running mongo db in detached mode
* Ps aux | grep mongo
* To start a mysql container with an environmental variable set in advance
  + Docker container run –p 3306:3306 –d –name db –e MYSQL\_RANDOM\_ROOT\_PASSWORD=yes mysql
  + ohxooNgaengae4mi7Quu0reimoChai3E
* To check the random generated password
  + Docker container logs db
* Docker container run –name webhost –d –p 8080:80 httpd
  + Host 8080 is forwarded to container port 80
  + 3.232.107.62:8080 will give output in browser
* Docker container top – process list in one container
* Docker container inspect – details of one container config
* Docker container stats – performance stats for all containers
* Docker container run –it – start a new container interactively
  + –I means interactively and –t means tty terminal
* Docker container exec –it – run additional command on existing container
* Docker container start –ai –to start an existing docker container again.
* Docker container run –name ubuntu –it Ubuntu bash
  + Here the last bash means the command to run on the container. Now you will be entering bash screen of new Ubuntu container
  + Use exit to exit from that bash mode
* If you exit on the above command, container will stop automatically.
* To exit without stopping the container use docker container exec –it command
* Docker pull alpine
  + To pull the alpine linux image from docker hub
  + Alpine will not contain bash also by default
* So use docker container run –it alpine sh

Docker Networking:

* Each container is 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
* “Batteries included, but Removable”
  + Defaults work well in many cases, but easy to swap out parts to customize it.
* Docker container run – 80:80 –name webhost –d nginx
* Docker container port webhost
  + This will give from which host port to container port it is forwarding
* To know the container IP address
  + Docker container inspect –format ‘{{ .NetworkSettings.IPAddress }}’ webhost
* If you create two or more containers in same network, they can talk to each other without opening any port separately.
* To show networks – “docker network ls”
* To inspect a network – “docker network inspect bridge”
* To create a network – “docker network create –driver”
* Attach a network to a container – “ docker network connect”
* Detach a network from a container – “docker network disconnect”
* We have three network types – **Bridge, host and none**
* **Bridge is** the default virtual network, which is NAT’ed behind the host IP.
* “Docker network inspect bridge “gives the containers that are attached to it along with some other info.
* **Host network** gains performance by skipping virtual network but sacrifices the security of the container model.
* **None network** type removes the eth0 network and only leaves you with localhost interface in the container
* “docker network create my\_app\_net”
  + Creating a new network
* A network driver Is a built in or 3rd party extensions that give you virtual network features.
* Create a new container with new network
  + Docker container run –d –name new\_nginx –network my\_app\_net nginx
* If we have two containers running on same network, they can be easily talk to each or ping with their **container id or container Name instead of IP address**.
  + Docker container exec –it my\_nginx ping new\_nginx
  + But this feature not supported in default bridge network
  + You can link the two containers in bridge network with –link option but creating new network will be better option.

DNS Round Robin Test:

* “docker network create dude”
* “docker container run –d –net dude –net-alias search elasticsearch:2”
* Create another container with the same command as above
* “docker container run –rm –net dude alpine nslookup search”
  + Use nslookup command to search in alpine image
* “docker container run –rm –net dude centos curl –s search:9208”
* Do the same with other port as well
* “docker container run –rm –net dude centos curl –s search:9209”

Docker image:

* App binaries and dependencies
* Metadata about the image data and how to run the image
* An image is an ordered collection of root file system changes and corresponding execution parameters for use within a container runtime.
* Not a complete OS. No kernel, no kernel modules (drivers)
* Small as one file (your app binary) like a golang static binary
* Big as Ubuntu distro with apt, and Apache, PHP and more installed.
* <https://hub.docker.com>

**Docker Commit:**

Docker commit command would save the changes we made to the Docker container’s file system to a new image.

Docker commit container\_ID repository\_name:tag

Docker images:

* Images are made as union file system where changes made on file system level and cache them locally.
* “docker image ls”
* “docker history nginx:latest” or “docker image history nginx:latest”
  + Shows layers of changes made in image
* For the first time, we pull the image from the docker hub is called base image and It will be the first layer.
* Each layer is uniquely identified and only stored once on a host. This saves storage space on host and transfer time on push/pull.
* Every time you make any change to that image like installing new packages, changing env variables, opening ports etc. each change considered as a new layer on that image.
* “docker image inspect nginx” returns JSON metadata about the image.
* “docker image tag –help” – assign one or more tags to an image.
* Official Repositories live at the “root namespace” of the registry, so they don’t need account name in front of repo name.
* The repository name in the docker comes in format “owner-account-id/name” but the official repositories come without owner-account-id. Ex: mysql/mysql-server
* TAGs are the pointers to the specific image commit. We can specify the versions with the Tags.
* If you want to pull specify version of nginx “nginx:version” will give you the specific version of nginx. Here the version is the tag that we specify. Without tag, it will take latest as default.
* We can tag our own images and publish them to github.
* “docker image tag nginx sprudhivi/nginx”
  + This will tag the nginx with my account name sprudhivi and I can publish this now.
* Latest is just the default tag, but image owners should assign it to the newest stable version.
* After you tag your image, you will push with “docker image push sprudhivi/nginx” which uploads the changed layers to a image registry.
* Before pushing, you need to login to docker hub from console.
  + Docker login
  + This will ask username and password. After providing, you will see a message Login succeeded.
* Once logged in, you can push your image.
* You can see the login authentication details in “cat .docker/config.json” also session details.
* Once you push, it’s good to logout. “docker logout”

Docker files:

* The default file name is “Dockerfile”
* To build the image from docker file,
  + “docker build -f some-dockerfile”
* Package manager is like apt and yum are one of the reasons to build containers FROM Debian, Ubuntu, Fedora or CentOS.
* Docker file starts with FROM to get the default OS to build the image up on.
* Next thing is ENV to setup the environment variables in the build.
* Once reason ENV Variables were chose as preferred way to inject key/value is they work everywhere, on every OS and config.
* You use RUN command to execute any shell commands or install any packages.
* We can use EXPOSE command to open ports inside the container.
  + EXPOSE 80 443
* At the end, we have one mandatory command
  + CMD [“nginx”, “-g”, “daemon off;]
* To build the docker file,
  + “docker image build -t customnginx .
    - Here -t is to add the tags and “.” At end is to run docker file in current directory.
* If you edit the docker file and run the build command again, only the changed line execution happens, not the entire build again.
* The sample docker is

FROM nginx:latest

WORKDIR /usr/share/nginx/html

COPY index.html index.html

* WORKDIR is like change directory command cd.
* If you build the above docker file and run a container with that image, you will see different index.html output that we copied rather than the default nginx home.

**Dockerfile Assignment:**

FROM node:6-alpine

EXPOSE 3000

RUN apk add –update tini

RUN mkdir -p /usr/src/app

WORKDIR /usr/src/app

COPY packge.json package.json

RUN npm install && npm cache clean

COPY . .

CMD [“tini”, “--”, “node”, “./bin/www”]

* Docker build -t testnode .
* Once the image is build, run the container.
* “docker container run –rm -p 80:3000 testnode
* Check the execution in browser.
* Once done, change the tag. “docker tag testnode sprudhivi/testing-node”
* Docker push sprudhivi/testing-node.
* We can use “prune” commands to clean up the images, volumes, build cache, and containers.
* “docker image prune” to clean up just dangling images.
* “docker system prune” will cleanup everything.
* “docker image prune -a” will remove all images you’re not using.

Container Lifetime:

* Containers are usually immutable and ephemeral.
* “Immutable infrastructure”: only re=deploy containers, never change
* This is the ideal scenario, but what about databases, or unique data?
* Docker gives us features to ensure these “separation of concerns”
* This is known as “persistent data”
* Two was: Volumes and Bind Mounts
* Volumes: make special location outside of container UFS
* Bind Mounts: link container patch to host path

**Volumes**:

* If we check mysql default image dockerfile, we can see a line “VOLUME /var/lib/mysql” which means it will create a local mount of volume to store the files related to the image in this location.
* Even the container is removed, this location volume will not be removed.
* You can check these volume details in “docker image inspect mysql” under volume secion.
* “docker container run –name mysql mysql”
* Docker container ls
* Docker container inspect mysql
* Now you can see the volume and mount section having the volume details.
* “docker volume ls” to see the volumes created so far.
* “docker volume inspect uniq-id” will give same details.
* The volume names are assigned with a unique string and very long length.
* So we have named volumes concept to name our volume.
* While creating a container we can specify -v command
  + To attach it to a new volume
    - Docker container run -d –name mysql -v /var/lib/mysql mysql
  + Or to attach it to an existing volume
    - Docker container run -d –name mysql -v mysql-db:/var/lib/mysql mysq
    - Here we are giving a name to volume mysql-db
* Now if you do “docker volume ls” we can see a new volume with a name that we specified instead of long unique name
* “docker volume create” required to do this before “docker run” to use custom drivers and labels.

Bind Mounting:

* Maps a host file or directory to a container file or directory
* Basically, just two locations pointing to the same file(s)
* Again, skips UFS, and host files overwrite any in container.
* Can’t use in Dockerfile, must be at container run
* … run -v /Users/bret/stuff:/path/container
* “docker container run -d –name nginx -p 80:80 -v $(pwd):/usr/share/nginx/html nginx
  + Now the files in pwd will be synced to the volume /usr/share/nginx/html on the container.

Assessment:

* Docker container run -d –name psql -v psql:/var/lib/postgresql/data postgres:9.6.1
* Docker container logs -f psql
  + To see the logs of container for initial setup. Once this logging is done, we can proceed to next step.
* Docker container stop psql
* Docker container run -d –name psqls2 -v psql:/var/lib/postgresql/data postgres:9.6.2
* Docker ps -a
* Docker volume ls
* Now here we did upgrade of postgre sql and used the same volume to restore the previous data.

Docker Compose:

* Why: configure relations between the containers.
* Why: Save our docker container run settings on easy-to-read file.
* Why: create one liner developer environment startups.
* Comprised of 2 separate but related things
* 1. YAML – formatted file that describes our solution options for
  + Containers
  + Networks
  + Volumes
* 2. A CLI tool docker-compose used for local dev/test automation with those YAML files.
* Compose YAML 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.
* With docker directly in our production with Swarm (as of v1.13)
* Docker-comose –help
* Docker-compose.yml is the default file name otherwise you can use the –f to specify file name.

Sample compose files:

Version: ‘2’

Services:

Jelly:

Image: sprudhivi/mysql-server

Volumes:

* + - .:/site

Ports:

* + - ’80:4000

Another example:

Version: ‘2’

Services:

Wordpress:

Image: wordpress

Ports:

* + - 8080:80

Environment:

WORDPRESS\_DB\_PASSWORD: example

Volumes:

* + - ./wordpress-data:/var/www/html

Mysql:

Image: mariadb

Environment:

MYSQL\_ROOT\_PASSWORD: example

Volumes:

* + - ./mysql-data:/var/lib/mysql

CLI:

* Docker CLI tools comes with default in docker for windows/Mac, but separate download for Linux
* Not a production grade tool but ideal for local level environments.
* Two most common use commands
  + Docker-compose up #setup volumes, networks and start all containers
  + Docker-compose down #stop all containers and remove cont/vol/net
* Docker-compose up
* Docker-compose up –d #run in backgroup
* Docker-compose logs
* Docker-compose ps –a
* Docker-compose top

Assignment:

Version: ‘2’

Services:

Drupal:

Image: drupal

Ports:

* + - ‘8080:80’

Volumes:

Copy from official page

Postgres:

Image: postgres

Environment:

* + - POSTGRES\_PASSWORD=mypassword

Compose to build:

* Compose can also build your custom images
* Will build them with docker-compose up if not found in cache
* Also rebuild with docker-compose build
* Great for complex builds that have lots of vars or build args

Version: ‘2’

Services:

Proxy:

Build:

Context: .

Dockerfile: nginx.Dockerfile

Image: nginx-custom

Ports:

* + - * ’80:80’

Web:

Image: httpd

Volumes:

* + - * ./html:/usr/local/apache2/htdocs
* In the above code, we are giving image as nginx-custom. If that image is already in cache, it will proceed. If not, it will build it with the details mentioned in build section.
* In build section context is the directory which is pwd and Dockerfile is the name of file in the context.
* We can also delete the local images creating while removing the containers.
  + “docker-compose down –rmi local”

Docker SWARM:

* Docker swarm comes with default docker installation, but we need to enable it separately.
* To see the status of swarm, “docker info” and find the swarm row.
* Run “docker swarm init” for initialization.