Docker

. Batteries are available but also replacable.
. Docker run and Docker Commit are complimentary to each other.
    -Run command Converts images to containers.
    -Commit command converts containers to images, it doesn’t overwrites the image from which the container was made from.
    $docker commit <name> <name the you want>
. Docker run starts a container by giving it an image name and a process to run in that container(This is the main process. One main process).
. Started a container attached and want to move away from it - Sequence Control+p,Control+q. It lets you detach yourself from the container but leaves it running and get attached and working back anytime you need.
. Add another process to the same container - docker exec -ti <container-name> <process name say bash.
. docker logs is an important command for debugging.
. Resource constraints. Ex :
    - docker run —memory maximum-allowed-memory image-name
    - docker run —cpu-shares (relative to other containers)
    - docker run —cpu-quota (to limit it in general)
. Don’t let your containers fetch dependencies when they start.
. Docker allows publishing a port.
. Communication between docker container and others through published port.
    - docker run —rm -ti -p 45678:45678 -p 45679:45679 —name echo-server ubuntu:14.04 bash -> terminal 1

    - nc -lp 45678 | nc -lp 45679 -> terminal 1
    - nc localhost 45678 -> terminal 2 send data here
    - nc localhost 45679 ->terminal 3 data received here

    - nc -lp 45678 | nc -lp 45679 -> terminal 1
    - docker run —rm -ti ubuntu:14.04 bash -> terminal 2
    - nc host.docker.internal 45678 -> terminal 2 send data here
    - docker run —rm -ti ubuntu:14.04 bash -> terminal 3
    - nc host.docker.internal 45679 ->terminal 3 data received here

. Expose port dynamically.
    - docker run —rm -ti -p 45678 -p 45679 —name echo-server ubuntu:14.04 bash -> terminal 1
    - nc -lp 45678 | nc -lp 45679 -> terminal 1
    - docker port echo-server -> terminal 2
        #45678/tcp -> 0.0.0.32777
        #45678/tcp -> 0.0.0.32776
    - nc localhost 32777 -> terminal 2 send data here
    - nc localhost 32776-> terminal 3 data received here

. Docker network
- docker network create learning -> terminal 1
- docker run —rm -ti —net learning —name catserver ubuntu:14.04 bash -> terminal 2
- docker run —rm -ti —net learning —name dogserver ubuntu:14.04 bash -> terminal 3
- ping dog server -> terminal 2
- nc -lp 1234 -> terminal 3
- nc dogserver 1234 -> terminal 2

. Legacy linking (Only one way)
- docker run —rm -ti -e SECRET=theinternetlovescat —name catserver ubuntu:14.04 bash -> terminal 1
- docker run —rm -ti —link catserver —name dogserver ubuntu:14.04 bash -> terminal 2
- nc -lp 1234 -> terminal 1 (both way)
- nc catserver 1234 -> terminal 2 (both way)
- nc -lp 4321 -> terminal 2
- nc dogserver 4321 ->terminal 1
    #nc: getaddrinfo: Nam or service not known
- env -> terminal 1
    #SECRET=theinternetlovescat
- env -> terminal 2
    #CATSERVER_ENV_SECRET=theinternetlovescat

. Images
- docker images
- docker commit <id> <name>- tags image for you
- docker commit <id> <name>:<tag>

. Complete name of image - registry.example.com:port/organization/image-name:version-tag
- docker rmi <id/name>

. Volumes - Sharing data between containers, container and hosts. Not a part of image. 2 variety.
1. Persistent
2. Ephemral

. Sharing data with host
- mkdir example
- docker run -ti -v /entire path/example:/path-of-shared-folder ubuntu bash
- touch /path-of-shared-folder/abc.txt
    #shared with host

. Sharing between containers

- docker run -ti -v /shared-data ubuntu bash -> terminal 1
- echo hello > /shared-data/data-file -> terminal 1
- docker ps -l —format=$FORMAT -> terminal 2
- docker run -ti —volume-from <name of the container sharing the folder> ubuntu bash -> terminal 2
- ls /shared-data -> terminal 2

#data is there
#when exited that data are not saved

. Docker Images

# Registries manage and distribute images
- docker search ubuntu -> search from command line
- docker login
- docker pull debian:sid
- docker tag debian:sid pratikghose/test-image-01:99.9
- docker push pratikghose/test-image-01:99.9

. Docker files are small programs to create an image

- docker build -t name-of-results . -> the . Is file's path

#each step of running a docker file is cached

. Build a docker file

```
{
    FROM busybody
    RUN echo “building simple docker image”
    CMD echo “Hello Container”
}
```

- docker build -t hello .
- docker run -rm hello

```
{
    FROM debian:sid
    RUN apt-get -y update
    RUN apt-get install nano
    CMD “nano” “/tmp/notes” #or CMD [“/bin/nano”, “/tmp/notes”]
}
```

- docker build -t example/nanoer .
- docker run - rm -ti example/nanoer

```
{
    FROM example/nanoer
    ADD notes.txt /notes.txt
    CMD “nano” “/notes.txt” #or CMD [“/bin/nano”, “/notes.txt”]
}
```

- nano notes.txt
  #write somethings
- docker build -t example/notes
- docker run -rm -ti example/notes

. FROM statement : what image do you start running from. Always be first expression of docker file. Can be multiple.
- FROM java:8

. MAINTAINER : Defines the author of the docker file
- MAINTAINER Firstname Lastname <example@email.com>

. RUN : run a command from the shell, waits for it to finish and saves the result
- RUN unzip install.zim /opt/install/

. ADD : Add local files, Add contents from archives, works with urls
- ADD run.sh /run.sh
- ADD project.tar.gz /install/
- ADD https://project.example.com/download/1.0/project.rpm /project/

. ENV : sets environment variables both during the build and when running the results
- ENV DB_HOST=db.production.example.com
- ENV DB_PORT=5432

. ENTRYPOINT : specifies the start of the command to run. It specifies the beginning of the expression to use when starting your container and lets you tack more on the end. If your container has an ENTRYPOINT of ls, then anything you type when you say - docker run myimagename would be treated as arguments to ls command. It gets added to when arguments are added to the container.

. CMD : specifies the whole command run and if the person when they're running the container types something after - docker run myimagename that will be run instead of CMD. It get replaced when arguments are added to the container.

. ENTRYPOINT and CMD can use either form:
1. Shell form:
   - nano notes.txt
2. Exec form: Slightly more efficient
   - ["/bin/nano", "notes.txt"]

. EXPOSE : maps port into the container
- EXOSE 8080

. VOLUME : Defines shared or ephemeral volumes depending upon you having one or two arguments

- VOLUME [“/host/path/” “/container/path/”]
- VOLUME [“/shared-data”]

#avoid defining shared folders in DockerFile with the host

. WORKDIR : sets the directory for the remainder of the docker file and the resulting container when you run it. Its like running the command cd for all the RUN statements.

- WORKDIR /install/

. USER : sets which user the container will run as.

- USER Pratik
- USER 1000

. MultiStage Builds

```
{
	FROM ubuntu:16.04
	RUN apt-get update
	RUN apt-get -y curl
	RUN curl https://google.com | wc -c > google-size
	ENTRYPOINT echo google is this big; cat google-size
}
```

- docker build -t too-big .
- docker run too-big
- docker images

#too-big - 171MB
#try to make its size smaller

```
{
	FROM ubuntu:16.04 as builder
	RUN apt-get update
	RUN apt-get -y curl
	RUN curl https://google.com | wc -c > google-size

	FROM alpine
	COPY —from=builder /google-size /google-size
	ENTRYPOINT echo google is this big; cat google-size
}
```

- docker build -t google-size .
- docker run google-size
- docker images

#google-size - 4.41MB

. What kernels do..??

#Runs Directly on the hardware.

#Recieve and responds to messages from hardware
#Start and Schedule programs
#Control and organise storage
#Pass messages between programs
#Allocate resources, memory, CPU, network, etc.
#Create containers by Docker configuring the kernel.

. What Docker Does..??
#Program written in Go
#Manages Kernel features
#Uses “cgroups” to contain processes
#Uses “namespaces” to contain network
#Uses “copy-on-write” filesystems to build images
#Makes scripting distributed systems “easy”

. The Docker Control Socket
#Docker is two programs: a client and a server
#The server receives commands over a socket (either over a network or through a “file”)
#The client can even run inside docker itself
- ls /var/run/docker.sock
- docker run -ti —rm -v /var/run/docker.sock:/var/run/docker.sock docker sh
- docker info
- docker run -ti —rm ubuntu bash
#This is a client within a docker container controlling a server that’s outside that container.

. Networking in a brief
. Bridging
#Docker uses bridges to create virtual networks in your computer
#These are software switches
#They control the Ethernet layer

- docker run -ti --rm —net=host ubuntu:16.04 bash -> terminal 1
- apt-get update && apt-get install all bridge-utils -> terminal 1
- brctl show -> terminal 1
- docker network create my-new-network -> terminal 2
- brctl show -> terminal 1
#A new network will show up

. Routing
#Creates “Firewall” rules to move packets between networks
#NAT
#Change the source address on the way out
#Change the destination address on the way back

- sudo iptables -n -L -t nat
#"Exposing" a port really is "port forwarding"

- docker run -ti —rm —net=host —privileged=true ubuntu bash -> terminal 1
- apt-get update -> terminal 1
- apt-get install iptables -> terminal 1
- iptables -n -L -t nat -> terminal 1
- docker un -ti —rm -p 8080:8080 ubuntu bash -> terminal 2
- iptables -n -L -t nat -> terminal 1
#Now we see a port forward rule for port 8080 and the ip of docker created

. Namespaces
#They allow processes to be attached to private network segments
#These private networks are bridged into a shared network with the rest of the containers
#Containers have virtual "Cards"
#Containers get their own copy of the networking stack

. Primer on Linux processes
#In docker your container starts with an init process ad vanishes when that process exits
- docker run -ti —rm —name hello ubuntu bash -> terminal 1
- docker inspect —format '{{.State.Pid}}' hello -> terminal 2
#7368 - process identification number
- docker run -ti —rm —privileged=true —pid=host ubuntu bash -> terminal 2
- kill 7368
#Container in terminal 1 is done

. Docker COWs : Copy on Write
#The contents of layers are moved between containers in gzip files
#Containers are independent of the storage engine
#Any container can be loaded (almost) anywhere
#It is possible to run out of layers on some of the storage engines

. Volumes and Bind Mounting
#The Linux VFS
#Mounting devices/directories on the VFS
- docker run -ti —rm —privileged=true ubuntu bash
- mkdir example
- cd example
- mkdir work
- cd work
- touch a b c d e f

- cd ..
- mkdir other-work
- cd other-work
- touch other-a other-b other-c other-d
- cd ..
- ls -R
- mount -o bind other-work work
- ls -R

#work now has the content of other-work layered onto it

- umount work
- ls -R

#Everything is back to the way they are
#Mounting volumes - always mounts host's filesystem over the guest

. Registries in detail
. What is Docker Registry?

#Its just a program which stores layers of Images and it's just a service usually on port 5000
#Docker makes network services run easier and Registry is a network service so we can run it on docker

- docker run -d -o 5000:5000 —restart=always —name registry registry:2
- docker tag ubuntu:14.04 localhost:5000/mycompany/my-ubuntu:99
- docker push localhost:5000/mycompany/my-ubuntu:99

. Storage Options

#Local storage
#Docker trusted repo
#AWS Repo
#Google Container repo
#Azure Container repo

. Saving and Loading Containers

# docker save
# docker run

- docker save -o my-image.tar.gz debian:sid busybox ubuntu:14.04
- docker rmi ebian:sid busy box ubuntu:14.04

#gone

- docker load -I my-image.tar.gz
- docker images

#They are all back

. Introduction to Orchestration

. One container = one hand clapping

#Many options

#Starts containers - Keeps them running - restart them if they fail
#Service discovery - allow them to find each other
#Resource allocation - match containers to computer

. Docker Compose
#Single machine coordination - for smaller systems
#Designed for testing and development
#Brings up all your containers, volumes, networks, etc., with one command.

#Larger Systems
. Kubernetes
#Containers run programs
#Pods are a group of containers that are intended to be run together in the same system
#Services makes pods available to others
#Labels are used for describing every aspect of your system. Labels are used for very advance service discovery

Advantages
#Make scripting large operations possible with the kubectl command
#Very flexible overlay of networking
#Runs equally well on your hardware or a cloud provider
#Build-in service discovery

. EC2 Container Service(ECS)
#Uses analogous vocabulary but with a slight different twist to each part
#Task Definitions : provide all the information required to start a container and a set of container that are designed to run together.
#Tasks : is when a Task Definition is actually running. Actually makes a container run right now
#Services and expose it to Net : Services takes Tasks, expose them to the Net as a whole and ensure they stay up all the time.

Advantages
#Connects load balancer(ELBs) to services
#Can create your own host instances in AWS
#Make your instances start the agent and join the cluster
#Pass the docker control docker into the agent
#Provides docker repos - and it’s easy to run your own repo
#Note that containers(tasks) can e a part of CloudFormation stack!

. AWS Fargate
#It is more automated version of AWS ECS
. Docker Swarm
. Google Kubernetes Engine

. Microsoft Kubernetes Service

.Kubernetes in AWS
- kubectl get services

#Everything in Kubernetes has a replication controller and a service. The Replication controller's job is to make sure that this service is up to when it needs to be.

- kubectl apply -f https://raw.githubusercontent.com/kubernetes/v1.10.3/examples/guestbook-go/redis-master-controller.json
- kubectl apply -f https://raw.githubusercontent.com/kubernetes/v1.10.3/examples/guestbook-go/redis-master-service.json
- kubectl apply -f https://raw.githubusercontent.com/kubernetes/v1.10.3/examples/guestbook-go/redis-slave-controller.json
- kubectl apply -f https://raw.githubusercontent.com/kubernetes/v1.10.3/examples/guestbook-go/redis-slave-service.json
- kubectl apply -f https://raw.githubusercontent.com/kubernetes/v1.10.3/examples/guestbook-go/guestbook-controller.json
- kubectl apply -f https://raw.githubusercontent.com/kubernetes/v1.10.3/examples/guestbook-go/guestbook-service.json
- kubectl get services
- kubectl delete rc/redis-master rc/redis-slave rc/guestbook svc/redis-master svc/redis-slave sac/guestbook
- kubectl get services

*****DOCKER ESSENTIAL TRAINING*****
____Docker Swarm____
- docker swarm
- docker swarm init - swarm initialise, you'll get a command with url, paste it on another docker node to create a worker
- docker swarm join-token manager - to add a redundant joint token node
- docker swarm join-token worker - get a worker token, copy it and paste it into another node to use it as worker

. Node : Physical host that's running docker, the docker engine.
. Service : Automatically run across however many nodes are needed, Highly availability built in, Scalability of app is easy, and Exposing networks and storage is easy.
#task - WebApp1, Container - NGINX
- docker service create —name webapp1 —replicas=6 nginx
- docker service ls

. Container : Runs on one host, Must share networks and storage per container when run and tough to scale and make highly available.

. Lock Your Swarm Cluster
- docker swarm init —autolock=true
- docker swarm update —autolock=true - gives you a key to unlock which you have to store.
- docker node ls
- sudo systemctl restart docker
- docker node ls
- docker swarm unlock - paste the key copied above
- docker node ls
- docker swarm unlock-key - get back the key if not locked
- docker swarm unlock-key —rotate - gives a new key so save it

. Visualising Swarm
- docker info | more

#Docker swarm visualizer
- docker run -it -d -p 8080:8080 -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer
- docker ps
- docker service ls - webapp1 10 replication of nginx
- ipadder - get ip of the host and add port 8080 on browser you'll get the visualizer

. Analysing Services
- docker inspect webapp1 | more

. Understanding Stacks and stack file

#Stacks are used to define complete multitiered, highly available, highly scalable applications to a swarm cluster. Multiple services running Multiple containers are defined as Docker stack with the stack file.

#Stack File is used to define complete, multiple-tiered, highly available, highly scalable application to a swarm cluster. Created in YAML format. Portable from desktop to swarm cluster.
- docker-stack.yml -> Docker stack file
- docker stack deploy —compose-file docker-stack.yml mystack
- docker stack ls
- docker stack ps
- docker stack services

. Manipulating a running stack of services
- docker service ls
- docker service update —replicas=20 mystack_web
- cat docker-stack.yml
- gedit docker-stack.yml -> make changes that you need
- docker stack deploy —compose-file docker-stack.yml mystack

. Modifying network ports

```
- docker service update —publish-add published=8080, target=8080 mystack_web
```

. Mounting Volumes

```
- docker service update —mount-add type =volume,source=web-vol, targer=/web-vol-dir mystack_web
```

. Replicated vs Global services

#Replicated services are default services in docker swarm.

#Global service, it will run exactly one task, which is typically one container, on every single node in the cluster. There is no set number of clusters. It's just going to run one on every single cluster.

```
- docker service create —mode global —name global-service nginx
- docker service ls
```

. Trouble shooting a service

```
- docker service create —name webserver2 -p8080:80 httpd
- docker service logs webserver2
- docker service logs -f webserver2
```

. Node Labels

```
- docker node update —label-add prio1 DC2-N1
- docker node inspect DC2-N1 | more
```

____Docker Image Creation, Management, and Registry____

. Docker Image

#An image is an executable package that includes everything needed to run an application - the code, a runtime, libraries, environment variables ,and configuration files. A container isa runtime instance of an image.

. Layering with Docker Images

#The layered file system allows Docker to be really flexible and efficient. Images are made up of multiple read-only ;ayers. Multiple containers are typically based on the same image. When an image is instantiated into a container, a top veritable layer is created (which is deleted when the container is removed). Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all the drivers use stackable image layers and the copy-on-write(CoW) strategy.

. DockerFile

#The configuration that builds the Docker Image is called the Dockerfile. It is a text file that contains all the commands, in order, needed to build a given image. A docker file is executed by the docker build command.

- man docker file
- man docker build

. Docker Registry
#It is stateless, highly scalable application that stores and lets you distribute Docker images. Registries could be local(private) or cloud-based(private or public).
- docker run -d -p 5000:5000 —restart=always —name myRegistry registry:2

____Docker Storage and Volumes____

. Issues with Storing data in containers
#Containers are design to be ephemeral
#When containers are stopped, data is not accessible
#Containers are typically stored on each host
#Containers filesystems wasn't designed for high performance I/O

. Options for Data storage
#Volume
#Bind Mount
#tmpfs Mounts

. Block storage
#Fixed chunks of data
#No metadata is stored
#Beest for I/O intensive apps
#SAN storage uses block storage protocols like iSCSI

. Object Storage
#Data is stored with metadata and a unique identifier
#There is no organisation to the objects
#Scalability is limitless
#Accessed with Restful HTTP calls

. Default Docker Engine Security

. Kernel namespaces
#Created when a container is run
#Container processes cannot see or affect other processes
#Each container has its own network stack

. Control Groups
#Created when container is run
#Provide resource accounting and limiting
#Ensure no containers exhaust host resources

. Docker Daemon attack surface

#Attack surface is small, but take appropriate security
#Only run Docker on a host
#Only let trusted users control the Docker Daemon

. Linux Kernel Capabilities
#Docker runs containers with restricted capabilities
#For example, even OS containers don't get full root privileges

. Mutually Authenticated TLS(MTLS)
#Manually rotating certificates is painful. Rotting security certificates without downtime is even harder. MTLS is built into Docker Swarm and solves these challenges.

. Swarm Managers vs Swarm Workers
#Maintain the cluster state #Sole purpose is to execute container workloads
#Schedules services
#Service as Swarm HTTP API endpoints

. Initiating key rotation
- docker swarm ca —rotate

. Docker content trust
#It's important to know what the image you wanted is the image you received especially when pulling images from internet sites or on large organisations.
#Docker Notary us an open source project that provides the ability to digitally sign content and is also included with DTR.
#Enforces a policy that any operations with a remote registry require signed images
- export DOCKER_CONTENT_TRUT=1 -> Enable
#Once enabled, the docker push will try to sign an image
#Once enabled, a docker pull will only download signed images
#With UCP, you can ensure that only signed images can be run.

. Docker Access Control Model
#Subject
#Roles
#Resources
#Grants

. Subjects
#Ex : User, Team, Organisation, Service account
#Subjects are granted roles
. Roles

#Defines what can be done
#Ex : View Only, Restricted Control, full control

. Resources Sets
#Swarm Collections or Kubernetes namespaces
#Collections and namespaces together are called resource sets

. Grant
#It is a combination of subject, role and resource set
#These are the "ACL" of Docker UCP

——NETWORKING——

. Basic things needed to be done
#Types of network
#Publish ports
#DNS
#Load glancing
#Traffic flow
#Logging

Use Cases for Docker Networking

. Bridge
#Connects container to the LAN and other containers
#The default network type
#Great for most use cases

. Host
#Remove network isolation between container and host - Connected directly to the host network
#Only one container (or application on the host) can use a port at the same time
#Useful for specific applications, such as a management container that you want to own on every host.

. Overlay
#Connect multiple Docker hosts (and their containers) together and enable swarm
#Only available with Docker EE and Swarm enabled
#Multihost networking using VXLAN

.Macvlan
#Assign a MAC address, appears as physical host
#Clones host interfaces to create virtual interfaces, available in the container
#Supports connecting to VLANs

.None

#Connects the container to an isolated network with only that container on it
#Container cannot communicate with any other networks or networked devices

*****Creating a bridge network*****
It is a default network. We’d create a user defined bridge network.

- docker network ls
- docker network inspect bridge
- docker network create —driver bridge app-net
- docker network ls
- docker network inspect app-net
- docker run -dit --name app1 --network app-net alpine ash
- docker run -dit --name app2 —network app-net alpine ash
- docker ps
- docker ps | grep alpine
- docker network inspect app-net
- docker container attach app1
    - ping app2
    - ping www.docker.com
    - control + p + q
- docker container attach app2
    - ping app1
    - ping www.docker.com
    - control + p + q
- docker network ls
- docker container stop app1 app2
- docker container rm app1 app2
- docker network rm app-net
- docker network ls

*****Creating a overlay network*****
It is created by default if you have created a swarm or initialised swarm on a node.

- docker network ls
- docker network create --driver overlay app-overlay
- docker network ls
- docker service create --network app-overlay --name app1 --replicas=6 nginx
- docker service ls
- docker ps | grep app1
- docker service inspect app1 | more
- docker ps | grep app1
- docker container inspect fca | more
- docker network inspect app-overlay

*****Publishing ports*****
By default,
1. Containers are connected to bridge network.
2. Containers have outbound network access but no inbound network access.

Ports must be published to allow inbound network access.
- -p host port:container port
- -P exposed ports in expose statement inside of docker file

- docker container run -dit ip 8080:80 nginx
- docker ps
- docker stop mys

- docker container run -dit -P nginx

. Host and Ingress Port Publishing
. Host Port Publishing
- Typically used with global mode service
- Used to publish a single port on each host
- When publishing ports, specify mode=host

. Ingress Port Publishing
- Used with replicated mode services
- Used to publish a single port access all hosts that goes to a pool of containers

*****Configuring DNS*****
#Using an internal DNS Server
- docker container run -it --dns 192.168.1.254 centos /bin/bash
- ping esx3.company.pri -> Successful
- nslookup www.google.com -> fails
- cat /etc/resolv.conf -> name server 192.168.1.254
- exit

- docker container run -it centos /bin/bash
- ping esx3.company.pri -> Fails
- ping www.google.com -> Succeed
- cat /etc/resolv.conf -> Shows default DNS

- sudo nano /etc/docker/daemon.json
```
{
"dns":["192.168.1.254]
}
```
- sudo systemctl restart docker
- docker container run -it centos /bin/bash

- cat /etc/resolv.conf -> name server 192.168.1.254
- ping esx3.company.pri -> Successful

*****Configuring Load Balancing*****
. Load Balancing
	#Docker Swarm provides load balancing of workloads as they are instantiated.
	#Scaling of new workloads is very easy.
	#However Swarm hasn’t traditionally scaled up or scaled down containers, based on demand nor has it load balanced layer 7 web based application based on domain names

*****Configuring Host Networking*****
#Instead of container running and having some sort of network address translation that you may or may not configure between the host and the container’s inside network, with host netwroking the container runs and it’s utilising the physical interface of the host network.

- docker run --rm -d --network host --name my_nginx nginx
- docker ps
- docker container port relaxed_lumiere -> relaxed_lumiere is ports
- ip addr
- docker network ls
- docker network inspect host
- go to web browser and try the IP address.

. How Does Docker EE Traffic flow?
	#Easy answer is API
	#Whether you are using the UCP GUI, the Docker CLI, DTR, or other Docker-compatible tools, management traffic flows via API

*****Identifying external network ports*****
- docker version
- docker ps
- docker container port cup-proxy
- docker container port cup-metrics
- docker container port ucp-kv
- docker container port ucp-controller
- docker container inspect usp-controller

*****Using logs to analyse networking issues*****
- docker container start web1 -> nginx web server with 80/tcp -> 0.
0.0.0:8080
- docker container port web1
- docker container logs web1

- docker container logs --follow web1