**DOCKER**

**Have you worked on containers?**

Containers are form of lightweight virtualization, heavier than chroot but lighter than hypervisors. They provide isolation among processes while using same kernel as the host machine, and cgroups functionality within kernel. But container formats differ among themselves in a way that some provide more VM-like experience while other containerize only application.

LXC containers are most VM-like and most heavy weight, while Docker used to be more light weight and was initially designed for single application container. But in more recent releases Docker introduced whole machine containerization features so now Docker can be used both ways. There is also rkt from CoreOS and LXD from Canonical, which builds upon LXC.

Sonarcube, chef,aws testing,

**What is Kubernetes? Explain**

It is massively scalable tool for managing containers, made by Google. It is used internally on huge deployments and because of that it is maybe the best option for production use of containers. It supports self-healing by restating non-responsive containers, it pack containers in a way that they take less resources and has many other great features.

**VM VS DOCKER CONTAINERS**

Majority of people link Docker containers with VMs, they are two technologies which differ in the way they operate. They share some similarities like both are designed to provide an isolated environment in which to run an application. The key difference is that the underlying architecture is different. Virtual machines have a full OS with its own memory management. Every guest OS runs as an individual entity from the host system.

Whereas containers run on the host OS itself and use the ephemeral storage. Containers are therefore smaller than Virtual Machines and enable faster start up with better performance, less isolation and greater compatibility possible due to sharing of the host’s kernel. Docker is not a virtualization technology, it’s an application delivery technology.

Finally, the benefits of Docker containers include:

* They are lightweight in nature use less memory and
* Lesser time to start a container
* Comparably, applications hosted in containers offer superior performance (almost twice according to Docker estimate).

Future is about integrating the Docker containers with VMs which will provide them with pros like proven isolation, security properties, mobility, dynamic virtual networking, software-defined storage and massive ecosystem.

**DOCKERFILE**

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

The Docker daemon runs the instructions in the Dockerfile one-by-one, committing the result of each instruction to a new image if necessary, before finally outputting the ID of your new image.

**The format of Dockerfile is:**

#comment

INSTRUCTION/DIRECTIVES arguments

Although the instruction is not case-sensitive, it is convention to specify them in uppercase to distinguish them from arguments.

The instructions frequently used in a Dockerfile are:

**FROM**

The base image for building a new image. This command must be on top of the dockerfile.

**MAINTAINER**

Optional, contains the name of the maintainer of the image..

**RUN**

Used to execute a command during the build process of the docker image.

**ADD**

Copy a file from the host machine to the new docker image. There is an option to use an URL for the file, docker will then download that file to the destination directory.

**ENV**

Define an environment variable.

**CMD**

Used for executing commands when we build a new container from the docker image.

**ENTRYPOINT**

Define the default command that will be executed when the container is running.

**WORKDIR**

This is directive for CMD command to be executed.

**USER**

Set the user or UID for the container created with the image.

**VOLUME**

Enable access/linked directory between the container and the host machine.

• EXPOSE

• LABEL

Before the Docker CLI sends the context to the Docker daemon, it looks for a file named .dockerignore in the root directory of the context. If this file exists, the CLI modifies the context to exclude files and directories that match patterns in it. This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images using ADD or COPY.

**DOCKER COMMANDS**

* Docker build -t
* Docker run -d -it –rm -p –name -P(for all ports) -v --net --net=bridge –ip --link
* Docker ps & -a
* Docker images
* Docker commit -m -a container ID/ Image ID
* Docker rm & rmi
* Docker rm/stop $(Docker ps -a -q)
* Docker inspect
* Docker search
* Docker login and logout
* Docker pull & push
* Docker attach and Docker exec
* docker rmi $(docker images --quiet --filter "dangling=true")
* Docker system prune
* Docker --version – displays docker version
* Docker version – will display the details of client and server – version, api version, go version, built no etc.
* Docker info – detailed information about Docker engine
* Docker network ls – list the available docker networks
* Docker network inspect bridgename – info about docker bridge
* Docker network create --subnet <10.1.0.0./16> --gateway <10.0.0.1> --ip-range <10.1.4.0/24> --driver < > --label < >bridgename
* Docker network rm bridgename – caution never remove a default one u cannot retrieve it
* Docker

**Command Description**

* docker attach => Attach to a running container
* docker build => Build an image from a Dockerfile
* docker checkpoint => Manage checkpoints
* docker commit => Create a new image from a container’s changes
* docker container => Manage containers
* docker cp => Copy files/folders between a container and the local filesystem
* docker create => Create a new container
* docker deploy => Deploy a new stack or update an existing stack
* docker diff => Inspect changes to files or directories on a container’s filesystem
* docker events => Get real time events from the server
* docker exec => Run a command in a running container
* docker export => Export a container’s filesystem as a tar archive
* docker history => Show the history of an image
* docker image => Manage images
* docker images => List images
* docker import => Import the contents from a tarball to create a filesystem image
* docker info => Display system-wide information
* docker inspect => Return low-level information on Docker objects
* docker kill => Kill one or more running containers
* docker load => Load an image from a tar archive or STDIN
* docker login => Log in to a Docker registry
* docker logout => Log out from a Docker registry
* docker logs => Fetch the logs of a container
* docker network => Manage networks
* docker node => Manage Swarm nodes
* docker pause => Pause all processes within one or more containers
* docker plugin => Manage plugins
* docker port => List port mappings or a specific mapping for the container
* docker ps => List containers
* docker pull => Pull an image or a repository from a registry
* docker push => Push an image or a repository to a registry
* docker rename => Rename a container
* docker restart => Restart one or more containers
* docker rm => Remove one or more containers
* docker rmi => Remove one or more images
* docker run => Run a command in a new container
* docker save => Save one or more images to a tar archive (streamed to STDOUT by default)
* docker search => Search the Docker Hub for images
* docker secret => Manage Docker secrets
* docker service => Manage services
* docker stack => Manage Docker stacks
* docker start => Start one or more stopped containers
* docker stats => Display a live stream of container(s) resource usage statistics
* docker stop => Stop one or more running containers
* docker swarm => Manage Swarm
* docker system => Manage Docker
* docker tag => Create a tag TARGET\_IMAGE that refers to SOURCE\_IMAGE
* docker top => Display the running processes of a container
* docker unpause => Unpause all processes within one or more containers
* docker update => Update configuration of one or more containers
* docker version => Show the Docker version information
* docker volume => Manage volumes
* docker wait => Block until one or more containers stop, then print their exit codes

**DOCKER REGISTRY & DOCKER HUB**

The Registry is a stateless, highly scalable server-side application that stores and lets you distribute Docker images.

Docker Hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker Cloud so you can deploy images to your hosts. It provides a centralized resource for container image distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

**DOCKER ADD VS COPY:** the major difference is that ADD can do more than COPY:

• ADD allows <src> to be an URL

• If the <src> parameter of ADD is an archive in a recognized compression format, it will be unpacked

**DOCKER STOP VS KILL**

• docker stop: Stop a running container by sending SIGTERM and then SIGKILL after a grace period

• docker kill: Kill a running container using SIGKILL or a specified signal

**DOCKER DEBUGGING**

• Docker logs <container\_ID>

• Docker stats <container\_ID>

• Docker cp <container\_ID>:path\_to\_logs /local/path

• Docker exec -it <container\_ID> /bin/bash

• docker commit <container\_id> my-broken-container && docker run -it my-broken-container /bin/bash

**DOCKERFILE BEST PRACTICES**

• Containers should be ephemeral

* Use .dockerignore file
* Avoid installing unnecessary packages
* Each container should have only one concern
* Minimize the number of layers
* Sort multi-line arguments

**KUBERNETES**

**CHALLENGES ADDRESSED BY KUBERNETES**

• Service Discovery

• Load Balancing

• Secrets/configuration/storage management

• Health checks

• Auto-[scaling/restart/healing] of containers and nodes

• Zero-downtime deploys

Kubernetes is a system developed by google to manage containerized applications in a clustered environment. It is primarily meant to address the gap between the modern cluster infrastructure and the presumptions of majority of applications about the environments.

The controlling services in a Kubernetes cluster are called the **master, or control plane**, components. These operate as the main management contact points for administrators, and provide many cluster-wide systems for the relatively dumb worker nodes. These services can be installed on a single machine, or distributed across multiple machines.

**MASTER OR CONTROL PLANE COMPONENTS**

**Etcd:** Kubernetes uses etcd, which is a distributed key-value store that can be distributed across multiple nodes, to store configuration data that can be used by each of the nodes in the cluster. It can be configured on a single master server or distributed among various machines while ensuring its connectivity to each of kubernetes machines.

**API Server:** It is the management point of the entire cluster which allows for configuration of Kubernetes workloads and organizational units. Acts as a bridge between various components to maintain cluster health.

**Controller Manager Service:** It is the one which maintains the state of the cluster which reads the latest information and implements the procedure that fulfills the desired state. This can involve scaling an application up or down, adjusting endpoints, etc.

**Scheduler Service:** This assigns the workload to the nodes and tracks resource utilization on each host to ensure that they are not overloaded.

**NODE SERVER COMPONENTS**

Are the one on which actual work is done. They have the following requirements to communicate with the master components and configure networking for containers:

**Docker running as a Dedicated subnet**

**Kubelet service:** The main contact point with master components and is a service. It receives commands and work and interacts with etcd to read configuration details of the nodes.

**Proxy Service:** Used to deal with the individual host level subnetting and make the services available to external parties through forwarding the requests to the correct containers.

**Kubernetes Work Units:** While containers are the used to deploy applications, the workloads that define each type of work are specific to Kubernetes.

**Pods:** The basic unit which generally represents one or more containers that should be controlled as a single "application". It models an application-specific "logical host" and can contain different application containers which are relatively tightly coupled. In Pod, Horizontal scaling is generally discouraged on the pod level because there are other units more suited for the task.

**Services:** A service, when described this way, is a unit that acts as a basic load balancer and ambassador for other containers. A service groups together logical collection of pods that perform the same function to present them as a single entity. Services are an interface to a group of containers so that consumers do not have to worry about anything beyond a single access location.

**Replicated Controllers:**

A more complex version of a pod is a replicated pod. These are handled by a type of work unit known as a replication controller.

A replication controller is a framework for defining pods that are meant to be horizontally scaled. The work unit is a nested unit. A template is provided, which is basically a complete pod definition. This is wrapped with additional details about the replication work that should be done.

Kubernetes for creating new Projects, Services for load balancing and adding them to Routes to be accessible from outside, Creation of Pods through new application and control the scaling of pods, troubleshooting pods through ssh and logs, writing/modification of Buildconfigs, templates, Imagestreams etc.

**KUBERNETES IN A NUTSHELL**

It is one of the most popular and stable management platforms for Docker containers – it powers Google Containers Engine (GCE) on the Google Cloud platform.

In Kubernetes, a group of one or more containers is called a ***pod***. Containers in a pod are deployed together, and are started, stopped, and replicated as a group. A **pod** could represent e.g. a web server with a database that run together as a microservice including shared network and storage resources. **Replication controllers** manage the deployment of pods to the cluster nodes and are responsible for creation, scaling and termination of pods. For example, in case of a node shutdown, the replication controller moves the pods to other nodes to ensure the desired number of replicas for this pod is available. Kubernetes **services** provide the connectivity with a load balancing proxy for multiple pods that belong to a service. This way clients don’t need to know which node runs a pod for the current service request. Each pod could have multiple **labels.** These labels are used to select resources for operations. For example, a replication controller and services discover pods by label selectors for various operations.

**CONFIGURING MASTER AND MINIONS:**

• The first pre-requisite is to install ntpd package on master and as well as minions and we need to enable and start this service to ensure that all the servers are time synchronized.

• Name the master and minions accordingly and save that in /etc/hosts file in order to refer to them based on those names rather than public ips.

• Next is to create a repo to pull latest docker package in /etc/yum.repos.d/virt7-docker-common-release and add the content with name and base url of the repo along with gpgcheck=0 and then run yum update to pull packages on to all the servers.

• Note: For our lab, we just need to ensure iptables and firewall.d services are disabled.

• Now we need to install two packages on all the servers docker and kubernetes by **yum install -y –enablerepo=virt7-docker-common-release kubernetes docker**

• Now we need to configure master server. The first step is to edit the config file in /etc/kubernetes/ where we edit the KUBE\_MASTER part to bind it to an interface that we can communicate with and we change its value to the master name that we have changed in /etc/hosts file and leave the default port to 8080. And we will add KUBE-ETCD-SERVERS=”--etcd-servers=http://master\_name:2379”

• Next step is to configure etcd in master by editing config file in the /etc/etcd/ by changing the listen client and advertise client urls to listen to all the servers on port number 2379.

• The third step is to edit the api server file in /etc/kubernetes/ and change the kube api address to bind to all servers and ensure that port on the local server is listening on 8080 and the kubelet port is 10250 which is default and we can edit admission control to restrict addition kubes and kublets entering our environment.

• Finally, we need to enable the services etcd, kube-apiserver, kube-controller-manager and kube-scheduler.

• For configuring minions, the first step is to edit the kubernetes config file by changing the kube master to look out for the name rather than ip and add etcd servers value to interact with the master etcd server.

* Next is to edit the kubelet config file where we change the kubelet address to bind all addresses, enable the kubelet port, set the kubelet hostname to bind to the minion name, map the kubelet api server to interact with the one in master

Now enable and start the services: kube-proxy, kubelet and docker.

The kubectl is the cli used to work with k8s:

* **Kubectl get nodes** – to get registered nodes with master, use -o flag to define the output path and filter it out
* **Kubectl describe nodes –** details about nodes
* **Kubectl get pods –** to list out pods on cluster

**Sample POD definition:**

apiVersion: 1

kind: pod

metadata:

name: ngnix

spec:

containers:

* name: ngnix

image: ngnix: 1.7.9

ports:

containerPort: 80

Now use **kubectl create -f file\_path** to create a pod. We will find a interim container google container which is required for kubernetes to run.

We need to do port-forward in order for the master to interact with the containers.

We can add a label (key/value pair) field to the metadata portion of yaml file to define labels which can be used to filter the pod details by providing the -l flag to list certain pods which match our defined key-value pair (label).

At times we might use an extension api version to support particular tasks like deployments.

We will definitely have a docker container in the name of google pause which is part of the implementation of pods.

We will add replicas in the spec portion and mention our desired number and we will add metadata under the spec again to add labels along with spec again, while using replicas.

* We can update a deployment by just using **kubectl apply -f file\_path**
* Autoscale command – **kubectl autoscale deployment <deployment\_name> --min=n --max=n –cpu-percent=x%**
* To edit an autoscaled deployment - **Kubectl scale –current-replicas=n –replicas=n deployment/<deployment\_name>**
* To create a deployment – **kubectl run <some\_name> --port=x --replicas=x --labels=a=b**

**Docker from Docker Documentation**

**Image:**

An image is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.

**Container:**

A container is a runtime instance of an image—what the image becomes in memory when actually executed. It runs completely isolated from the host environment by default, only accessing host files and ports if configured to do so.

Containers run apps natively on the host machine’s kernel. They have better performance characteristics than virtual machines that only get virtual access to host resources through a hypervisor. Containers can get native access, each one running in a discrete process, taking no more memory than any other executable.

**Components of docker:**

* Daemon
* Client
* Docker I/O registry

**Installation:** Docker is available in two versions, i.e., Community Edition and Enterprise Edition

Docker Community Edition (CE) is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker CE has two update channels, stable and edge:

* **Stable** gives the reliable updates every quarter
* **Edge** gives you new features every month

Docker Enterprise Edition (EE) is designed for enterprise development and IT teams who build, ship, and run business critical applications in production at scale.

Docker --version = to check whether docker is installed or not.

Yum install docker-engine => to install the docker engine

**Containers:**

App (containers) -> service (behavior in production) -> stack (top level) => hierarchy of docker

**Defining Container with Docker:**

Create a **Dockerfile** in the location or directory you want to create the container

* docker build -t name (repo name) => to build a docker image
* docker images => to list all the docker images
* docker run -p 4000:80 repo name => to run the docker container (-p maps the port 4000 to the containers port 80)
* docker run -d -p 4000:80 repo name => runs the application in background by using -d option
* docker ps -a => lists all the containers that are run previously and presently in stopped state
* docker ps => displays the current running containers
* docker search word => searches the list of images in the docker hub from the command line. (word = search pattern i.e., centos, ubuntu, java, etc.)
* docker pull image name => pulls the requested image from the docker hub
* docker run image name: => latest to run the image or container with specified tag names
* docker run image id => to run a docker image by mentioning its id)
* docker inspect image name or id => to view the more info about the container image
* docker stop container name or id => to stop the process using the container id or name
* docker login => to login to public registry on your local machine.
* docker tag image username/repository:tag => to tag the image ( docker tag image helloworld akiti1m/myrepo:1)
* docker push username/repository:tag => to upload the image to the repository or docker hub
* docker rmi imagename:tag => removes a docker image
* docker rm image id or container id => removes the image
* docker rm ‘docker ps -a -q’ = > to remove all the containers that stopped by the system at a time.
* Docker history imagename:version => To view the history of the image.
* Docker history -quite imagename

**Services:**

In a distributed application, different pieces of the app are called “services.” For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

Luckily, it’s very easy to define, run, and scale services with the Docker platform – just write a **docker-compose.yml** file.

A **docker-compose.yml** file is a YAML file that defines how Docker containers should behave in production.

**Packaging a customized docker container:** This can be done in 2 ways

* from the existing the container
* using the docker file

**Swarms:**

**Understanding Swarm clusters**

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you’re used to, but now they are executed on a cluster by a swarm manager. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as nodes.

swarm managers can use several strategies to run containers, such as “emptiest node” – which fills the least utilized machines with containers. Or “global”, which ensures that each machine gets exactly one instance of the specified container. You instruct the swarm manager to use these strategies in the Compose file, just like the one you have already been using.

Swarm managers are the only machines in a swarm that can execute your commands, or authorize other machines to join the swarm as workers. Workers are just there to provide capacity and do not have the authority to tell any other machine what it can and cannot do.

Up until now, you have been using Docker in a single-host mode on your local machine. But Docker also can be switched into swarm mode, and that’s what enables the use of swarms. Enabling swarm mode instantly makes the current machine a swarm manager. From then on, Docker will run the commands you execute on the swarm you’re managing, rather than just on the current machine.

**Set up your swarm**

A swarm is made up of multiple nodes, which can be either physical or virtual machines. The basic concept is simple enough:

* run docker swarm init => to enable swarm mode and make your current machine a swarm manager
* run docker swarm join => on other machines to have them join the swarm as workers. Choose a tab below to see how this plays out in various contexts.

Creating VM’s using the docker-machine (using VirtualBox driver)

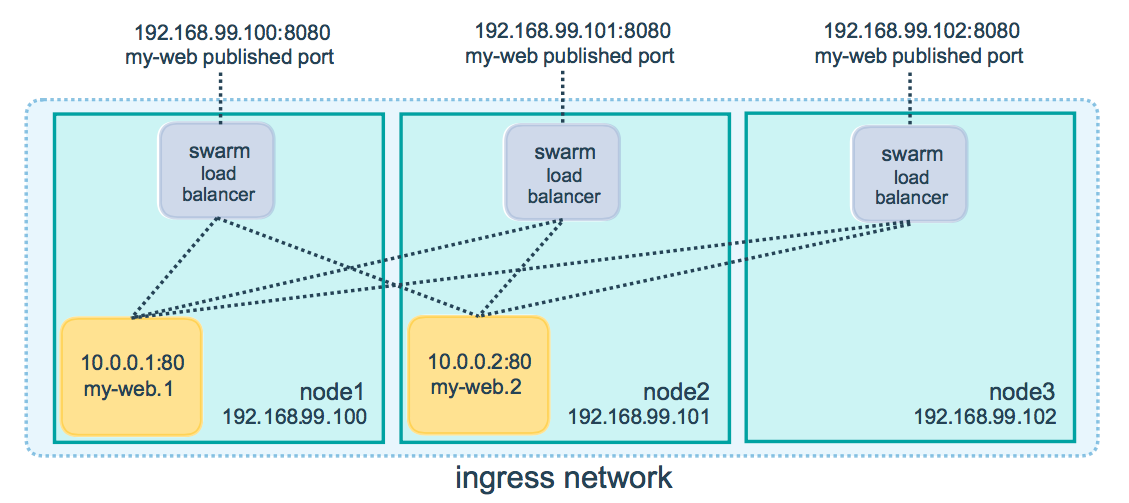
* docker-machine create --driver virtualbox vm1
* docker-machine create --driver virtualbox vm2
* docker-machine ls => list the all the docker machines
* docker node ls => lists the nodes in the swarm
* docker stack deploy => to deploy that are made to the swarm (docker stack rm => to clean up the docker stack docker-machine ssh myvm1 "docker stack rm getstartedlab")
* docker-machine ssh nodename "docker swarm leave" => to remove the node (worker) form the swarm cluster.
* docker-machine ssh managername "docker swarm leave --force" => to remove the swarm manager

2377 => port for swarm joins

### Accessing your cluster

You can access your app from the IP address of **either** myvm1 or myvm2. The network you created is shared between them and load-balancing. Run docker-machine ls to get your VMs’ IP addresses and visit either of them on a browser, hitting refresh (or just curl them). You’ll see five possible container IDs all cycling by randomly, demonstrating the load-balancing.

The reason both IP addresses work is that nodes in a swarm participate in an ingress **routing mesh**. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container. Here’s a diagram of how a routing mesh for a service called my-web published at port 8080 on a three-node swarm would look:



Keep in mind that in order to use the ingress network in the swarm, you need to have the following ports open between the swarm nodes before you enable swarm mode:

* Port 7946 TCP/UDP for container network discovery.
* Port 4789 UDP for the container ingress network.

**Stacks:**

A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together. A single stack is capable of defining and coordinating the functionality of an entire application (though very complex applications may want to use multiple stacks).

**Connecting Docker Cloud:** We can run the docker cloud in two modes i.e., standard mode and swarm mode

* Standard mode:

<https://docs.docker.com/docker-cloud/cloud-swarm/link-aws-swarm/#add-your-aws-account-credentials-to-docker-cloud>

* Swarm mode:

<https://docs.docker.com/docker-cloud/cloud-swarm/link-aws-swarm/#create-a-dockercloud-swarm-role-role-with-an-embedded-policy>

for the docker overview, follow the below given link:

<https://docs.docker.com/engine/docker-overview/>

**Docker RUN, CMD and ENTRYPOINT:**

* RUN executes command(s) in a new layer and creates a new image. E.g., it is often used for installing software packages.
* CMD sets default command and/or parameters, which can be overwritten from command line when docker container runs.
* ENTRYPOINT configures a container that will run as an executable.

**RUN:**

RUN instruction allows you to install your application and packages requited for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

RUN has two forms:

* RUN <command> (shell form)
* RUN ["executable", "param1", "param2"] (exec form)

**CMD:**

CMD instruction allows you to set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.

CMD has three forms:

* CMD ["executable","param1","param2"] (exec form, preferred)
* CMD ["param1","param2"] (sets additional default parameters for ENTRYPOINT in *exec* form)
* CMD command param1 param2 (shell form)

Again, the first and third forms were explained in *Shell and Exec forms* section. The second one is used together with ENTRYPOINT instruction in *exec* form. It sets default parameters that will be added after ENTRYPOINT parameters if container runs without command line arguments. See ENTRYPOINT for example.

**ENTRYPOINT**

ENTRYPOINT instruction allows you to configure a container that will run as an executable. It looks similar to CMD, because it also allows you to specify a command with parameters. The difference is ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters. (There is a way to ignore ENTTRYPOINT, but it is unlikely that you will do it.)

ENTRYPOINT has two forms:

* ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)
* ENTRYPOINT command param1 param2 (shell form)

Be very careful when choosing ENTRYPOINT form, because forms behaviour differs significantly.

##### **Exec form**

Exec form of ENTRYPOINT allows you to set commands and parameters and then use either form of CMD to set additional parameters that are more likely to be changed. ENTRYPOINT arguments are always used, while CMD ones can be overwritten by command line arguments provided when Docker container runs. For example, the following snippet in Dockerfile

ENTRYPOINT ["/bin/echo", "Hello"]

CMD ["world"]

when container runs as docker run -it <image> will produce output

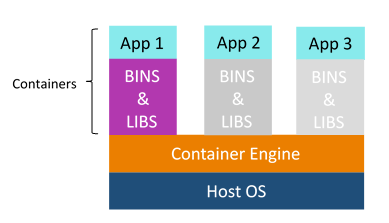
Hello world

but when container runs as docker run -it <image> John will result in

Hello John

**What are containers?**

containers are used to provide consistent computing environment from a developer’s laptop to a test environment, from a staging environment into production. A container consists of an entire runtime environment: an application, plus all its dependencies, libraries and other binaries, and configuration files needed to run it, bundled into one package. Containerizing the application platform and its dependencies removes the differences in OS distributions and underlying infrastructure.



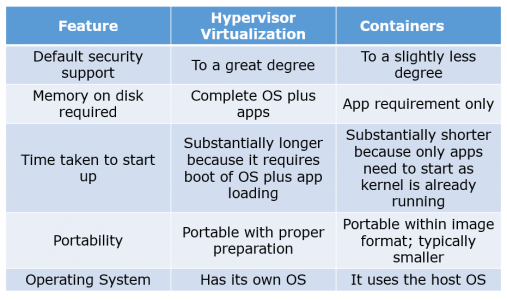
**What are the advantages that Containerization provides over virtualization?**

Below are the advantages of containerization over virtualization:

* Containers provide real-time provisioning and scalability but VMs provide slow provisioning
* Containers are lightweight when compared to VMs
* VMs have limited performance when compared to containers
* Containers have better resource utilization compared to VMs

**How exactly are containers (Docker in our case) different from hypervisor virtualization (vSphere)? What are the benefits?**

Given below are some differences. Make sure you include these differences in your answer:



**What is Docker image?**

Docker image is the source of Docker container. In other words, Docker images are used to create containers. Images are created with the build command, and they’ll produce a container when started with run. Images are stored in a Docker registry such as registry.hub.docker.com because they can become quite large, images are designed to be composed of layers of other images, allowing a minimal amount of data to be sent when transferring images over the network.

**What is Docker container?**

Docker containers include the application and all of its dependencies but share the kernel with other containers, running as isolated processes in user space on the host operating system.

Docker containers are not tied to any specific infrastructure:

They run on any computer, on any infrastructure, and in any cloud.

Docker containers can be created by either creating a Docker image and then running it or you can use Docker images that are present on the Dockerhub.

Docker containers are basically runtime instances of Docker images.

**What is Docker hub?**

Docker hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker cloud so you can deploy images to your hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

**How is Docker different from other container technologies?**

Docker containers are easy to deploy in a cloud. It can get more applications running on the same hardware than other technologies, it makes it easy for developers to quickly create, ready-to-run containerized applications and it makes managing and deploying applications much easier.

You can even share containers with your applications.

**What is Docker Swarm?**

It is native clustering for Docker which turns a pool of Docker hosts into a single, virtual Docker host. Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

I will also suggest you to include some supported tools:

* Dokku
* Docker Compose
* Docker Machine
* Jenkins

**What is Dockerfile used for?**

Docker can build images automatically by reading the instructions from a Dockerfile.  
A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

**Can I use json instead of yaml for my compose file in Docker?**

You can use json instead of yaml for your compose file, to use json file with compose, specify the filename to use for ex:  
**docker-compose -f docker-compose.json up**

**Tell us how you have used Docker in your past position?**

Explain how you have used Docker to help rapid deployment. Explain how you have scripted Docker and used Docker with other tools like Puppet, Chef or Jenkins. If you have no past practical experience in Docker and have past experience with other tools in similar space, be honest and explain the same. In this case, it makes sense if you can compare other tools to Docker in terms of functionality.

**How to create Docker container?**

I will suggest you to give a direct answer to this. We can use Docker image to create Docker container by using the below command:

**docker run -t -i <image name> <command name> =>** This command will create and start container.

You should also add, If you want to check the list of all running container with status on a host use the below command:  
**docker ps -a**

**How to stop and restart the Docker container?**

In order to stop the Docker container, you can use the below commands:  
**docker stop <container ID>**  
Now to restart the Docker container you can use:  
**docker restart <container ID>**

**How far do Docker containers scale?**

Large web deployments like Google and Twitter, and platform providers such as Heroku and dotCloud all run on container technology, at a scale of hundreds of thousands or even millions of containers running in parallel.

**What platforms does Docker run on?**

Docker runs on only Linux and Cloud platforms and then I will mention the below vendors of Linux:

* Ubuntu 12.04, 13.04 et al
* Fedora 19/20+
* RHEL 6.5+
* CentOS 6+
* Gentoo
* ArchLinux
* openSUSE 12.3+
* CRUX 3.0+

Cloud:

* Amazon EC2
* Google Compute Engine
* Microsoft Azure
* Rackspace

**Note that Docker does not run on Windows or Mac.**

**Do I lose my data when the Docker container exits?**

No, we won’t lose my data when Docker container exits. Any data that your application writes to disk gets preserved in its container until you explicitly delete the container. The file system for the container persists even after the container halts.

**What is Vagrant and what is it used for?**

Vagrant is a tool that can create and manage virtualized (or containerized) environments for testing and developing software. At first, Vagrant used VirtualBox as the hypervisor for virtual environments, but now it supports also KVM.

**Difference between containers and virtual machines?**

Each VM instantiation requires starting a full OS. VMs take up a lot of system resources. This quickly adds up to a lot of RAM and CPU cycles. Container host uses the process and file system isolation features of the linux kernel.

**What is Juju?**

Juju is orchestration tool primarily for ubuntu for management, provision and configuration on Ubuntu systems. It is being initially written in Python and since have been rewritten in Go.