Docker is an open source software development platform which runs software packages called "**containers**". A container is a standard unit of software that packages up all the dependencies of an application so that the application runs quickly and reliably from one computing environment to another.

Docker is an open source container platform, from Docker, Inc launched in 2013. Docker enables operating **system level virtualization**. Docker is available as free and enterprise version. Docker containers, unlike Virtual Machines, **share the same host operating system**. It has less overhead compared to VM's. Docker created the industry guidelines for containers, making it more portable and secure. Docker is having the best isolation capability that is beneficial when more containers run on the same host system. Docker could be run on any system on-premises or cloud. Using docker we can create, start, stop, move, pause, unpause or delete a container.

A Docker container can connect to one or more networks, attach storage to it, and even create a new image depending on the current state of the container. Docker also helps in controlling how the containers are isolated from other containers or host machine. Dockerfile in docker platform helps to achieve infrastructure as code thus enabling consistent and reliable computing environment. This reduces a lot of configuration and infrastructure related defects that would improve the quality of applications.

Docker is supported on all the latest technologies like Google Cloud Platform and by Google Kubernetes Engine. Docker was primarily developed for Linux systems but now also supports Windows and Mac Os.

1.

How is Docker different from standard virtualization using VMs?

Virtual Machines (VMs) virtualize the underlying hardware. They run on physical hardware via an intermediation layer known as a hypervisor. They require additional resources are required to scale-up VMs.

They are more suitable for monolithic applications. Whereas, Docker is operating system level virtualization. Docker containers userspaceace on top the of host kernel, making them lightweight and fast. Up-scaling is simpler, just need to create another container from an image.

Generally, Docker is more suitable for Microservices based cloud applications.

2.

What are the major components of Docker Architecture?

The four major components of Docker are daemon, Client, Host, and Registry

- **Docker daemon:** It is also referred to as 'dockerd' and it accepts Docker API requests and manages Docker objects such as images, containers, networks, and volumes. It can also communicate with other daemons to manage Docker services.

- **Docker Client:** It is the predominant way that enables Docker users to interact with Docker. It sends the docker commands to docked, which actually executes them using Docker API. The Docker client can communicate with more than one daemon.
- **Docker Registry:** It hosts the Docker images and is used to pull and push the docker images from the configured registry. Docker Hub is the public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. However, it is always recommended for organizations to use own private registry.
- **Docker Host:** It is the physical host (VM) on which Docker Daemon is running and docker images and containers are created.

3.
What is a Volume in docker?

A data volume is a specially-designated directory that is located outside of the root filesystem of a container (i.e. created on the host), designed to persist data, independent of the container's life cycle. This allows sharing data within containers by importing volume directory in other containers.

Data volumes provide several useful features:

- Data volumes persist even if the container itself is deleted.
- Data volumes can be shared and reused among containers.
- Changes to a data volume can be made directly.
- Volumes can be initialized when a container is created.

4.
When is .dockerignore file used?

A typical Dockerfile contains one or more COPY commands to copy files and/or folders from the developer machine to the docker image, which eventually become part of the container. While copying folders to a docker image, it is quite possible that some unwanted files are also copied to the image. This may create a bulky image and hence cause performance issues in the container.

In order to avoid this, we can create a file named ***.****dockerignore* along with Dockerfile in the same directory. This file is used to list all the files and directories that need to be excluded while copying folders onto the image. It contains a pattern and none of the files matching it is added to the image. This helps to avoid unnecessarily sending large or sensitive files and directories to the daemon and potentially adding them to images.

5.

What is docker-compose?

Compose is a tool provided by Docker for defining and running multi-container applications together in an isolated environment. Either a YAML or JSON file can be used to configure all the required services like Database, Messaging Queue along with the application server. Then, with a single command, we can create and start all the services from the configuration file.

It comes handy to reproduce the entire application along with its services in various environments like development, testing, staging and most importantly in CI as well.

Typically the configuration file is named as docker-compose.yml. Below is a sample file:

```
version: '3'
services:
  app:
    image: appName:latest
    build: .
    ports:
    - "8080"
    depends_on:
      - oracledb
    restart: on-failure:10
  oracledb:
    image: db:latest
    volumes:
      - /opt/oracle/oradata
    ports:
      - "1521"
```

6.

What is Docker Hub?

Docker Hub is a service provided by Docker for finding and sharing container images. The default version of Hub is the cloud-based registry that hosts all the public docker images like Ubuntu, Linux, etc.

We need to create repositories to push and pull the docker images, allowing us to share container images within our team, organization, customers. In the case of public repositories, we can share the images with the entire Docker community.

Docker images are pushed to Docker Hub through the 'docker push' command. A single Docker Hub repository can hold many Docker images.

It also allows you to link repositories with GitHub in order to automate building, testing and deploying of our application images. It provides a centralized resource for container image discovery, distribution and change management, collaboration and workflow automation throughout the development pipeline.

We can also use third-party Repository tools like Nexus and JFrog Artifactory to store and manage docker images.

7.

Explain basic Docker usage workflow.

Everything starts with the Dockerfile. The Dockerfile is the source code of the Image.
Once the Dockerfile is created, you build it to create the image of the container. The image is just the "compiled version" of the "source code" which is the Dockerfile.

Once you have the image of the container, you should redistribute it using the registry. The registry is like a git repository -- you can push and pull images.

Next, you can use the image to run containers. A running container is very similar, in many aspects, to a virtual machine (but without the hypervisor).

8.

What are the steps involved while using Docker for application development?

All the steps below are based on the prerequisite that Docker is already installed on the machine:

- **Create Dockerfile:**

The initial step is to create a Dockerfile file using a suitable base image along with all the required steps/commands, like setting environment variables, adding application jar, etc. This creates several layers on the existing base image.

- **Build image:**

Once the Dockerfile is ready, we can either use docker command or via a Gradle task to generate a docker image. This image contains all the application dependencies required to run the application in a container.

```
docker build -t-test/security tool.
```

- **Run the image:**

Once the docker image is built, we can create and start the container using command.

```
docker run --name rest_tool test/security tool.
```

- **Start Containers using Compose:**

In case we have multiple containers constituting an application like database, messaging queue, etc.; then it is advisable to use docker-compose to run multiple containers simultaneously. It is also useful in the CI pipeline for running the application and performing tests.

```
docker-compose up
```

- **Test the Application:**

After the containers are up and running, the application is ready for Integration or Acceptance tests to be performed. Ideally, it is integrated into the CI pipeline for determining if the new code changes are affecting the existing flows.

- **Push image:**

Typically in a container-based development environment, the deliverable artifact is a docker image. This image needs to be published to an internal Registry like Artifactory so that it can be propagated to next levels like Continuous Delivery and Deployment pipelines.

```
docker push test/securityTool
```

- **Production orchestration:**

Ideally, organizations need to use orchestration tools like Kubernetes to run the containers in a Pod to perform load balancing, service discovery, etc in a production environment. It also helps in providing scalability and high availability of the application.

9.
What are containers?

A container is a standalone unit of software that packages together applications with all its dependencies and configurations. Containers help in abstracting applications from the computing environment in which they run. Containers help in running the applications correctly in various computing environments. This helps to resolve the biggest problem " It doesn't work in my system. Multiple containers in an isolated manner can co-exist in a host system. All containers share the common host operating system. Thus making the containers lightweight and more preferable than virtual machines.

Detaching applications from its environment helps container-based applications to be deployed faster and on any on-premise systems or cloud. Developers with the help of containers can now concentrate on how to develop an application rather than how to run an application on different environments.IT operations teams focus on

deployment and management rather than with application details such as specific software versions and configurations. Main competitors for containers are the virtual machines. Virtual machines are not lightweight compared to containers.

Containerization as a strategy is used as a practice across digital transformation due to its capability to bring a DevOps culture. The Dockerfile which defines the configuration of the image also acts as Infra as code up to some extent. Thus enabling teams to abstract their infrastructure in the form of code.

10.
Why do we need containers? Give at least 3 reasons.

Containers is a virtualization technique which is based on the operating system resources. Multiple containers share the same host operating system. The main advantages are:

- **Consistent and Reliable Environment**

Containers help to achieve a stable environment that helps the application to run reliably. All dependencies like libraries etc needed for the applications are packaged into the containers. Containers can also include dependencies like specific versions of programming languages and other software libraries. Containers, in turn, increases the productivity of the developers and operations teams as they can concentrate more on the actual application rather than debugging different computing environments. This also creates fewer bugs for the developers and IT operations teams.

- **Run Anywhere**

Containers can run anywhere-

1. Across different operating system whether it be a Linux, Windows or Mac system. or a developer
2. Available across different service providers like AWS, Azure, GCP, etc.
3. Can be used across different environments like development, test, pre-production, production, etc.

- **Isolation**

Since the containers virtualize OS resources like file systems developers get an isolated sandbox from other applications using containers. The container itself acts as a full-fledged system providing this isolation.

- **From Code to Applications (Infrastructure As Code)**

Applications along with its dependencies are packaged into the containers which could be versioned. This helps easy management and leading to greater agility and productivity.

11.

Tell me the differences between virtual machines and containers?

Sl No:	Virtual Machine	Containers
1.	A virtualization technique where each VM has an individual operating system.	A virtualization technique where all containers share a host operating system.
2.	Virtual machines are isolated at the hardware level	Each container is isolated at the operating system level.
3.	Virtual machines take time to create	Containers are created fast
4.	Increased management overhead	Decreased management overhead as only one host operating system needs to be cared for.
5.	Takes minutes to boot up	Boots up in seconds
6.	Full isolation of VM's thus providing high security	Decreased security compared to VM's
7.	VM's are huge as it has the overhead of the separate OS	Lightweight as multiple containers share the same host operating system.
8.	VM's are mostly application-centric as making it service-centric is not a cost viable option.	Containers can be designed to be service-centric i.e a container for every microservice. Hence it is highly customizable for the performance of that particular microservices. Service-centric is the capability to configure a single service to run in a single container.
9.	Higher Cost	Less costly compared to VM's
10.	More effort required to make VM's versionable.	Easily Versionable
11.	The marketplace in VM's are segmented and not well known as Docker Hub	Availability of a central, trusted marketplace where we can pull the images for the containers

12.

What are the basic actions performed on the docker container?

Basic actions on Docker containers are:

- **Create a docker container**

Following command creates the docker container with the required images.

```
docker create --name <container-name> <image-name>
```

- **Run docker container**

Following command helps to execute the container

```
docker run -it -d --name <container-name> <image-name> bash
```

Main steps involved in the run command is

1. Pulls the image from Docker Hub if it's not present in the running environment
2. Creates the container.
3. The file system is allocated and is mounted on a read/write layer.
4. A network interface is allocated that allows docker to talk to the host.
5. Finds an available IP address from pool.
6. Runs our application in our case "*bash*" shell.
7. Captures application outputs

- **Pause container**

Processes running inside the container is paused. Following command helps us to achieve this.

```
docker pause <container-id/name>
```

Container can't be removed if in a paused state.

- **Unpause container**

Unpause moves the container back to run the state. Below command helps us to do this.

```
docker unpause <container-id/name>
```

- **Start container**

If container is in a stopped state, container is started.

```
docker start <container-id/name>
```

- **Stop container**

Container with all its processes is stopped with below command.

```
docker stop <container-id/name>
```

To stop all the running Docker containers use the below command

```
docker stop $(docker ps -a -q)
```

- **Restart container**

Container along with its processes are restarted

```
docker restart <container-id/name>
```

- **Kill container**

A container can be killed with below command

```
docker kill <container-id/name>
```

- **Destroy container**

The entire container is discarded. It is preferred to do this when the container is in a stopped state rather than do it forcefully.

```
docker rm <container-id/name>
```

13.
What are the disadvantages of Docker?

Like any other technologies, Docker has its own advantages and disadvantages. Docker is not a silver bullet and needs care in architecting and orchestrating docker containers keeping below points in mind.

- Docker is not so fast compared to bare metal servers. Even though containers are lightweight and so easy to boot up it is subjected to decreased network performance. Performance of the containers are affected as a single operating system caters to multiple containers.
- **Integrability:** Even though Docker is open source some of the container products don't work with all. This may be due to the competition prevailing in the market. For eg: OpenShift which is a container-as-a-service platform from Red Hat, only works with the Kubernetes.
- **Data loss in containers:** When the container exits the data will be lost. Data could be saved through volume mounting like Docker Data Volumes but more hard work needs to be done in this space.
- **Poor or no GUI:** Containers were used mainly for deploying server application that doesn't require GUI. There are some methods to run GUI app inside containers but it's somewhat clumsy these days.

- **More suitable for applications that use microservices:** Generally the docker's benefit is to ease the application delivery by providing a packaging mechanism but the true benefit comes when we use microservices.
- There are other virtualization techniques like unikernels which is based on library operating systems. Unikernels provide improved security, more optimisation and boots faster.

14.
What is docker image?

A Docker image is an executable file, that creates a Docker container. An image is built from the executable version of an application together with its dependencies and configurations. Running instance of an image is a container.

Docker image includes system libraries, tools, and other files and dependencies for the application. An image is made up of multiple layers. Layered structure helps the developers to reuse already available static image layers from the Docker Hub, for different projects. This saves the developers time. Each image has a base layer which could be already available in Docker Hub or built from scratch. Then a readable/writable layer over the static layer is created that helps to customise the container. Each layer of the docker image could be verified in /var/lib/docker/aufs/diff, or via the Docker history command.

Storage drivers are used to managing the image layers. A writable layer created while creating the container is called the container layer. Multiple containers can share the same base layer and have their own writable layer. Few commands that can be used with images are:

- *docker history* shows the history of an image and its layers.
- *docker update* helps to update the container configurations.
- *docker tag* creates a tag for the container and organizes container images.
- *docker search* looks for the image in Docker Hub
- *docker save* allows saving of images.
- *docker rmi* removes one or multiple images.

15.
What is the difference between ADD and COPY in Dockerfile?

The ADD instruction is used to copy files from build context to the image. Apart from regular files, it also allows URL and archive (tar, gzip, etc) files as the <source> parameter.

When a URL is provided, a file is downloaded from the URL and copied to the <destination>.It automatically unpacks compressed files, if the <source> argument is a local file in a recognized compression format (tar, gzip, bzip2, etc).

Whereas COPY does a straight-forward, as-is copy of files and folders from the build context into the container. It doesn't support URLs or gives any special treatment to archives.

Anything that you want to COPY into the container must be present in the local build context. The recommendation from the Docker team is to use COPY in almost all cases.

16.

What is the difference between ENTRYPOINT and CMD in Dockerfile?

ENTRYPOINT is a definition in Dockerfile that specifies a command that will always be executed when the container starts. It allows us to configure a command along with its parameters that will run as an executable in a container. Even if we do not specify any ENTRYPOINT, we may inherit it from the base image specified using the FROM keyword in your Dockerfile. Most of the official Docker base images have an ENTRYPOINT of /bin/sh or /bin/bash.

To override the ENTRYPOINT at runtime, we can use *--entrypoint.*

Whereas, the main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable or for executing an ad-hoc command in a container. It can omit the executable as well, in which case we must specify an ENTRYPOINT instruction as well specified with the JSON array format.

The thumb rule is that every Dockerfile should specify at least one of CMD or ENTRYPOINT commands.

17.

How is Kubernetes related to Docker?

Though Docker is a great tool for quick development and environment setup, it cannot be used in production directly. It needs a layer of orchestration for container management.

That’s where Kubernetes comes into the picture. Kubernetes is an open source orchestration system for Docker containers. It manages containerized applications across multiple hosts and provides basic mechanisms for deployment, maintenance, and scaling of applications.

In a Kubernetes environment, one or more docker containers are run inside a ‘pod’. A pod is a basic unit that Kubernetes deals with and it represents one or more containers that should be controlled as a single "application".

18.

What is Docker Swarm?

Swarm is the native way provided by docker for implementing clusters for Docker containers. Basically, it allows a group of machines that are running Docker to join as a single cluster. Docker commands executed on a cluster are implemented on all the hosts by the swarm manager. Machines in a swarm can either be physical or virtual. After joining a swarm, they are referred to as nodes.

It also provides built-in load balancing and Service Discovery for the applications running as containers.

However, a more popular way of implementing clustering in production is achieved using Kubernetes. It is technically superior to Docker Swarm as it provides better health checking provisions at the pod level, better container scheduling and scaling.

19.

How does Docker play a role in Continuous Integration?

Continuous Integration is a mechanism used to enable DevOps methodology in development process i.e. helps to obtain immediate feedback on the changes made in the code by executing unit and acceptance tests, as soon as the code changes are checked-in to the repository.

A typical process involves developers merging their changes to the branch as often as possible and they are validated by building, running and executing automated tests against the application.

Docker plays a key role in enabling CI, as it can seamlessly integrate with any popular CI tools like Jenkins or TeamCity. Also, several Gradle plugins like Palantir, trans mode are available for automating the docker build and publish images by creating respective Gradle tasks.

We can then define stages in Jenkins to perform tasks like building Docker images, starting containers using compose and executing Acceptance tests to validate the application changes.

20.

What is Docker architecture?

Docker is having a client-server architecture. The Docker client talks to the Docker daemon. Docker daemon helps in building, running, and distributing your Docker containers. The Docker client and daemon can co-exist on the same system or on different systems. This client-daemon connection is done via REST API.

- **The Docker daemon**

The Docker daemon is also known as docker engine manages docker objects such as images, containers, networks, and volumes. Communication with other docker daemons also happens through this. Docker daemon is started by the command "*dockerd*".Docker daemon talks with the kernel manages system calls for creation, running, destroying, etc of containers.

- **The Docker client**

The Docker client is the utility we use when we use commands such as docker run. The Docker client sends these requests to Docker daemon, which does the rest. One Docker client can connect with more than one Docker daemon.

- **Docker registries**

A Docker registry is where the Docker images are stored. Docker Hub is a public registry and docker searches for images on Docker Hub by default. Other registries are Docker Datacenter (DDC), which also includes the Docker Trusted Registry (DTR). Whenever *docker pull* or *docker run* commands are executed, the required images are pulled from the registry. We can also push the newly created docker images to Docker registry.

21.
What is developer workflow of docker?

The workflow starts with:

- **Attaining the docker image**

If the docker image is present in Docker Hub or that kind of Docker registries the images are selected from them. Developers can also create a great number of images if need be. Images have a layered architecture. Mostly a combination of a base layer and then a customised layer is used to create a docker image. There are several types of files that help to build images and they are kept in respective repositories. Some of the important files to consider to create docker images are docker files, docker-compose.yml files, etc. Multiple containers could be run together using docker-compose.yml files. Some of the types of repositories where the images are stored are

1. **Version Control** - This is used mainly used to store text-based files such as Dockerfiles, docker-compose.yml, and configuration files. Some of the version control systems are git, svn, Team Foundation Server, VSTS, and Clear Case.
2. **Repository Manager** - Large binary files such as that of Maven/Java, npm, NuGet, and RubyGems are stored in repository managers.Whereas smaller binary files could be saved to version control. Examples include Nexus, Artifactory, and Archiva.
3. **Package Repository** - Packaged applications for example like an operating system such as CentOS, Ubuntu, and Windows Server are stored in the package repository. Examples are yum, apt-get, and package management. They can also run several containers together described in a docker-compose.yml file and test the application.

- **Deploy on UCP(Docker Universal control plane)**

A CI/CD system runs unit tests, builds the applications, and create a Docker image on the Docker Universal Control Plane (UCP). If the image passes all tests they are signed using Docker Content Trust and shipped to Docker Trusted Registry (DTR). The developer could also do testing on an integration environment on UCP in cases where the developer's machine does not have access to all the resources to run the entire application.

- **Push Images**

Once the image is ready newer images created are pushed into Docker Trusted Registry. The developer can use the command

```
docker push < image name>
```

The developer account in DTR is a must have to do this action

- **Commit**

Once the application is tested on UCP files used to create the application, its images, and its configuration is committed to version control. The commit triggers could be set up to trigger the CI/CD workflow. The images could now be used to create containers.

22.
What are Dockerfiles?

Dockerfile is a text file that has instructions to build a Docker image. All commands in dockerfile could also be used from the command line to build images.

docker build command creates an image from the dockerfile and its contexts. Contexts are a set of files at a PATH/URL. URL could be any repository location of any version control system. PATH could also be a location in your local system. The following command shows building docker image from the contexts in the current directory and send the context to the docker daemon

```
$ docker build .
Sending build context to Docker daemon  6.51 MB
...
```

Root directory or path is not preferred as context path because the entire contents of your hard drive will be transferred to the Docker daemon.

Docker daemon performs a validation of the dockerfile before running instructions in dockerfile. Docker daemon takes on the instructions one by one creating an updated image each time.

Sample Dockerfile :

```
FROM ubuntu:16.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Each instruction in a dockerfile creates one read-only layer:

- FROM creates a layer from the ubuntu:16.04 Docker image as the base image.
- COPY adds files or contexts from your Docker client's current directory.
- RUN builds your application. Here we use make.
- CMD specifies the command to be run inside the container.

23.

What are docker-compose.yml files?

Docker compose is a tool that helps to run multi-container docker applications. Compose uses docker-compose.yml which is a YAML file to configure application's services. Compose workflow is a three-step process. First is to create a dockerfile to build the image, secondly define the services to be run on a container in the docker-compose.yml file and thirdly *docker-compose up* command starts up your entire applications. dockerfiles and docker-compose.yml files are similar but the main difference is that docker compose file manages multi-container architecture rather than a single container.

Sample docker-compose.yml file looks like:

```
version: "7"
services:
  testservice:
    # replace username/repo:tag with your name and image details
    image: <image name>
    deploy:
      replicas: 10
      resources:
        limits:
          cpus: "0.2"
          memory: 70M
      restart_policy:
        condition: on-failure
    ports:
      - "9080:80"
    networks:
      - webnet
```

This docker-compose.yml is reflected upon below:

- Pull the image from Docker registry .

- Run 10 instances of the pulled image as a service called testservice, with setting limits on each one as, 20% of a single core of CPU time and 70 MB of RAM.
- Immediately restart containers on failure.
- Map port 9080 on the host system to test services port 80.
- test services containers share port 80 through a load-balanced network called webnet.

24.

What is Docker volume mounting?

Data is not persistent inside a container. One of the most used methods is Volume mounting. Another method for persisting data is to include data in the writable layer but that layer is tightly coupled with the host machine where the container is running, thus the data couldn't be managed easily. Also writing into the writable layer is complex and needs to use a driver.

To retain data, the best-preferred way is to use a Docker volume mount where it mounts another directory into your container. This directory is on the host which could be seen within the container. The main advantage of volume mounting is

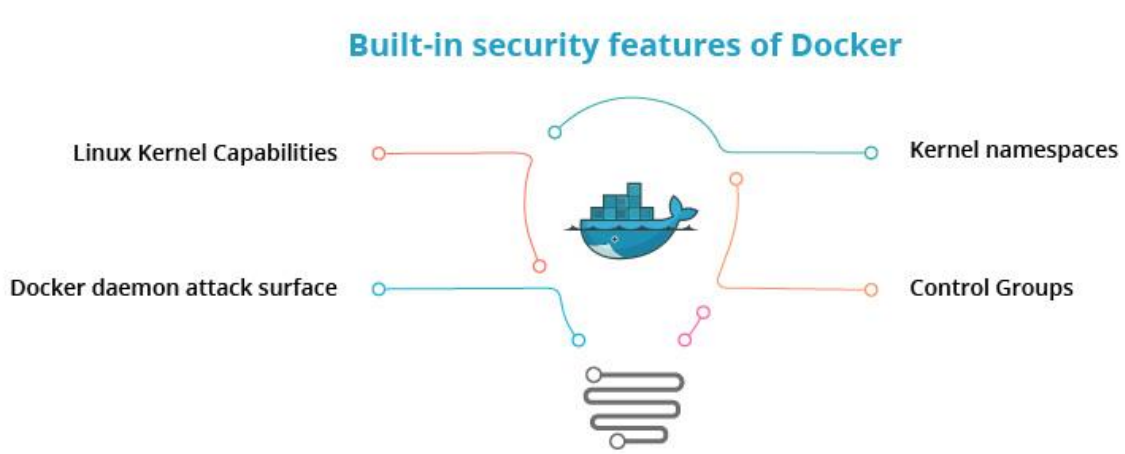
- Easy to manage using Docker CLI commands.
- We have support for both Windows and Linux systems
- Multiple containers can share the volumes.
- Volumes are easier to back up
- Volume drivers are present to encrypt volume contents and to store on cloud systems etc
- The container can pre-populate the new volumes with its contents
- Volumes don't increase the size of the containers.

Following commands could be used to manage volumes:

- docker volume create <name> to create the volume
- docker volume ls to list the volumes available to you
- docker volume inspect <name> to get more details about the volume like path, drivers etc.
- docker volume rm <name> to remove the container.
- docker volume prune to remove unused volumes

25.

What are the basic built-in security features of Docker?



- **Kernel namespaces**

Namespaces define the context in which names are defined whether it be variable names or function names. In other words, namespace defines the scope of the names. For eg: namespaces are like surnames. Suppose we have two people with the same name "John " their surname differentiates them.

Each container in docker creates a set of namespaces specific to the container. Hence is the first and a great method of security between containers.

- **Control Groups**

Control groups facilitate resource accounting and limiting. Control Groups doesn't allow a container to exhaust the host system's CPU, memory, disk I/O, etc. It also doesn't allow data and processes of container to be accessed by another container.

- **Docker daemon attack surface**

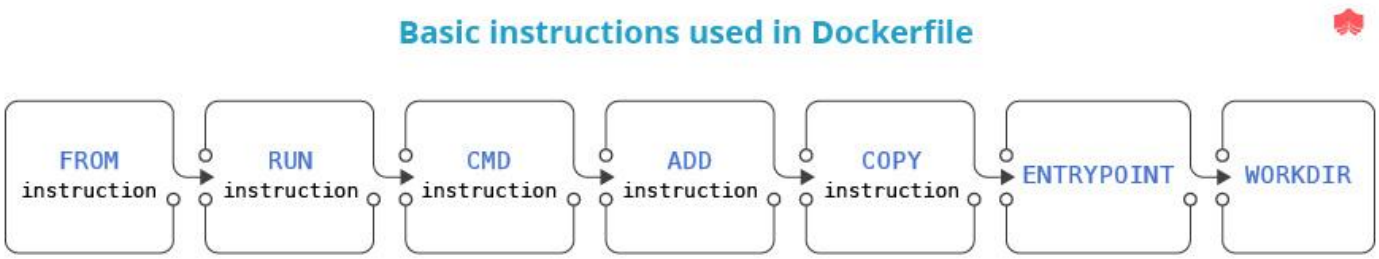
When a "*docker run* " command is performed docker client speaks to docker daemon who manages the images and containers. Docker daemon needs root privileges. Extra precaution must be taken to give access only to trusted users to control docker daemon. A container could even be started from the root directory on your host and the container can alter your host filesystem without any restriction.

- **Linux Kernel Capabilities**

Containers could be started with a reduced set of capabilities. This would mean that "root" within a container has fewer privileges than the real "root". This, in turn, reduces the damage by an intruder with root privileges.

26.

What are the basic instructions used in Dockerfile and how they differ from each other?



- **FROM instruction**

Usage :

```
FROM <image> [AS <name>]
```

Or

```
FROM <image>[:<tag>] [AS <name>]
```

Or

```
FROM <image>[@<digest>] [AS <name>]
```

FROM helps to set the base image for the following instructions in the Dockerfile. The base image could be one pulled from the remote public repository or a local image built by the user. FROM can appear multiple times in Dockerfile and each time it appears it clears any stage created from previous instructions. Hence usually an AS <NAME> is added to the FROM instruction to identify the last image created just before the FROM instruction.

- **RUN instruction**

Usage:

- RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
- RUN ["executable", "param1", "param2"] (exec form)

RUN instruction creates a new layer on the current image which is used for the next step in Dockerfile.RUN cache is valid for the next build provided "*docker build*" command is not run with "*--no-cache"* flag.

RUN in the executable form doesn't do shell processing like variable substitution.

- **CMD instruction**

Usage:

- CMD ["executable","param1","param2"] (exec form, this is the preferred form)

- CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
- CMD command param1 param2 (shell form)

CMD provides defaults for executing container. There could only be one CMD instruction but if there are many, only the last one has the effect. If CMD is providing default arguments for the ENTRYPOINT instruction, then the CMD and ENTRYPOINT instructions should be specified with the JSON array format.

- **ADD instruction**

Usage:

- ADD [--chown=<user>:<group>] <src>... <dest>
- ADD [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

- **COPY instruction**

Usage:

- COPY [--chown=<user>:<group>] <src>... <dest>
- COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.Main difference between COPY and ADD is that ADD supports the source to be a URL or tar file.

- **ENTRYPOINT**

Usage:

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

An ENTRYPOINT allows you to configure a container that will run as an executable. As a best practice ENTRYPOINT should be defined when using the container as an executable and CMD should be used as a way of defining default arguments for an ENTRYPOINT command.CMD could be overridden while running a container with arguments.

- **WORKDIR**

Usage:

- WORKDIR /path/to/workdir
- The WORKDIR instruction sets the working directory for RUN, CMD, ENTRYPOINT, COPY and ADD instructions in the Dockerfile.

27.

What are the networks used in Docker Swarm?

When a Docker Swarm is created or a docker host joins the swarm two networks are created.

- an overlay network called ingress, which manages control and data traffic related to swarm services. This is the default network unless the user specifies other user-defined overlay networks. User-defined overlay networks could be created by using the command "docker network create"ingress network facilitates load balancing among a service's nodes.
- To encrypt the application data traffic on a given overlay network, use the *--opt* encrypted flag on *docker network create.*
- To attach a service to an existing overlay network, use the *--network* flag to *docker service create*, or the *--network-add* flag to *docker service update* command. Service containers connected to an overlay network can communicate with each other through it.
- a bridge network called docker_gwbridge, which connects the Docker daemon to the other docker daemons in a swarm. docker_gwbridge connects the overlay networks (including the ingress network) to an individual Docker daemon's physical network. It is a virtual bridge that exists in the kernel of the docker host. For customising the bridge network we have to do that before docker host joins the swarm or by temporarily removing the host from the swarm.

28.

Explain different types of volume mounts in docker.

There are three mount types available in Docker

- Volume mounts are the best way to persist data in Docker. Data are stored in a part of the host filesystem which is managed by Docker containers. (/var/lib/docker/volumes/ on Linux).Main advantages of using volume mounts are

- Easy to use and backup
- Docker CLI commands or the Docker API can be used to manage volume mounts.
- Linux and Windows containers support volume mounts.
- Volumes could be shared securely among multiple containers.
- New volumes can use pre-populated content by a container.

Starting with Docker 17.06, *-v* or *--volume* flag and *--mount* flag could be used for docker swarm services and standalone containers. To create a docker volume. For eg:

"*docker volume create my-vol*" creates new volume "my-vol".

We can inspect a volume with the command "*docker volume inspect*"

For eg:

"*docker volume inspect my-vol* " gives the output


```
[
{
    "Driver": "local",
    "Labels": {},
    "Mountpoint": "/var/lib/docker/volumes/my-vol/_data",
    "Name": "my-vol",
    "Options": {},
    "Scope": "local"
}
]
```


If we need to start a container with "my-vol"

- With -v flag

"*docker run -d --name devtest -v my-vol:/app nginx:latest*". Here nginx images with the latest tag are executed with using volume mount "my-vol"

- With --mount flag

"docker run -d --name devtest --mount \ source=my-vol,target=/app nginx:latest"

- Bind mounts may be stored anywhere on the host system. A file or directory on the host machine is mounted into a container unlike volume mounts where a new directory is created within Docker's storage directory

on the host machine, and Docker manages that directory's contents. Non-Docker processes on the Docker host or a Docker container can modify them at any time.

- tmpfs mounts are stored in the host system's memory only and are never written to the host system's file system. When the container stops, the tmpfs mount is removed, and files won't persist.

29. What is multistage dockerfile, why we use it ?

30. Security measures while building the docker image/ Image harding

31. How to take a image/container backup

32. How to clean up dangling image

33. how to commit a container?