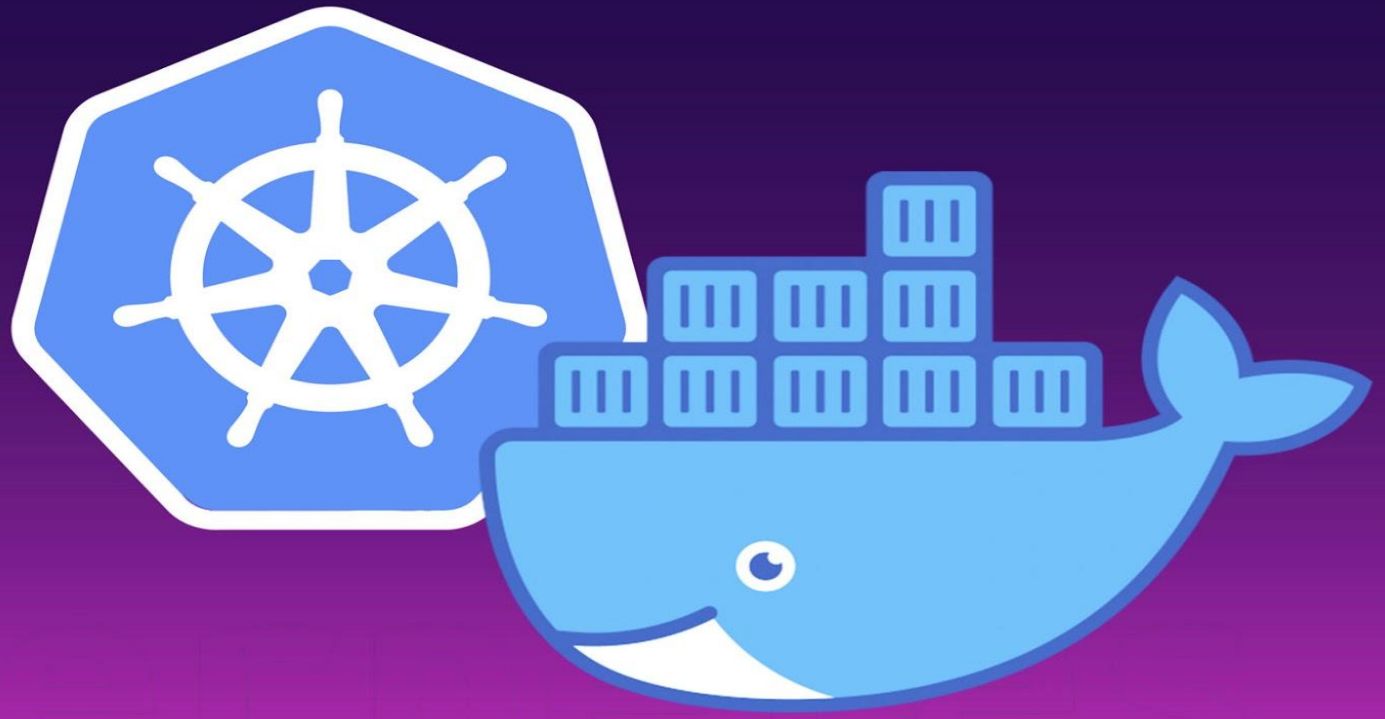
Container & Orchestration



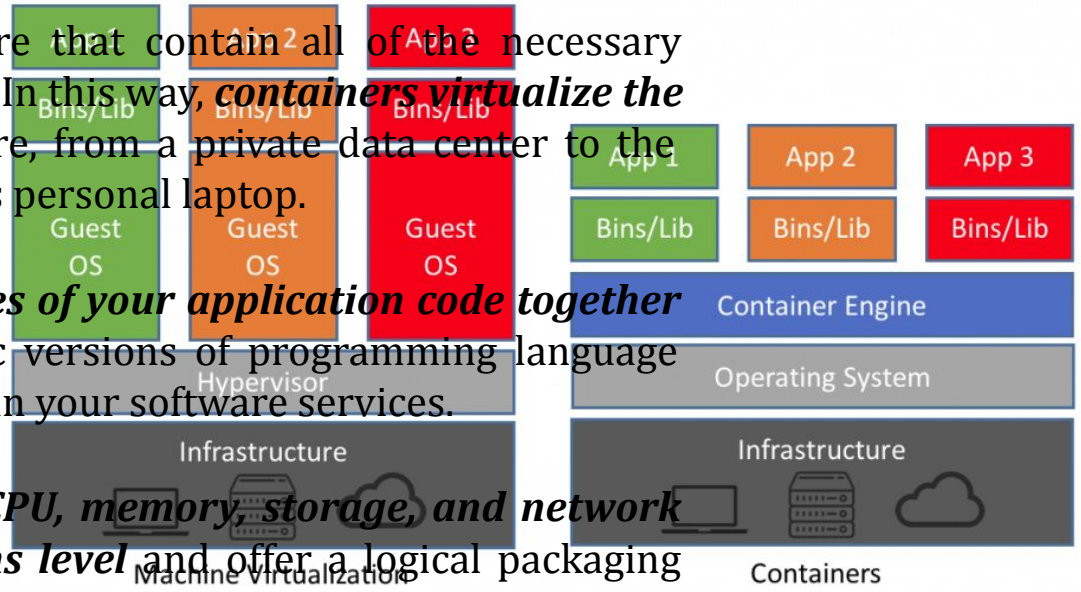
Build, Ship, Run

Container Technology ?

Container Technology ?

In software engineering, containerization is ***operating system-level virtualization*** or ***application-level virtualization*** over multiple network resources so that software ***applications can run in isolated user spaces*** called containers in any cloud or non-cloud environment, regardless of type or vendor.

Containers are packages of software that contain all of the necessary elements to run in any environment. In this way, ***containers virtualize the operating system*** and run anywhere, from a private data center to the public cloud or even on a developer's personal laptop.



Containers are ***lightweight packages of your application code together with dependencies*** such as specific versions of programming language runtimes and libraries required to run your software services.

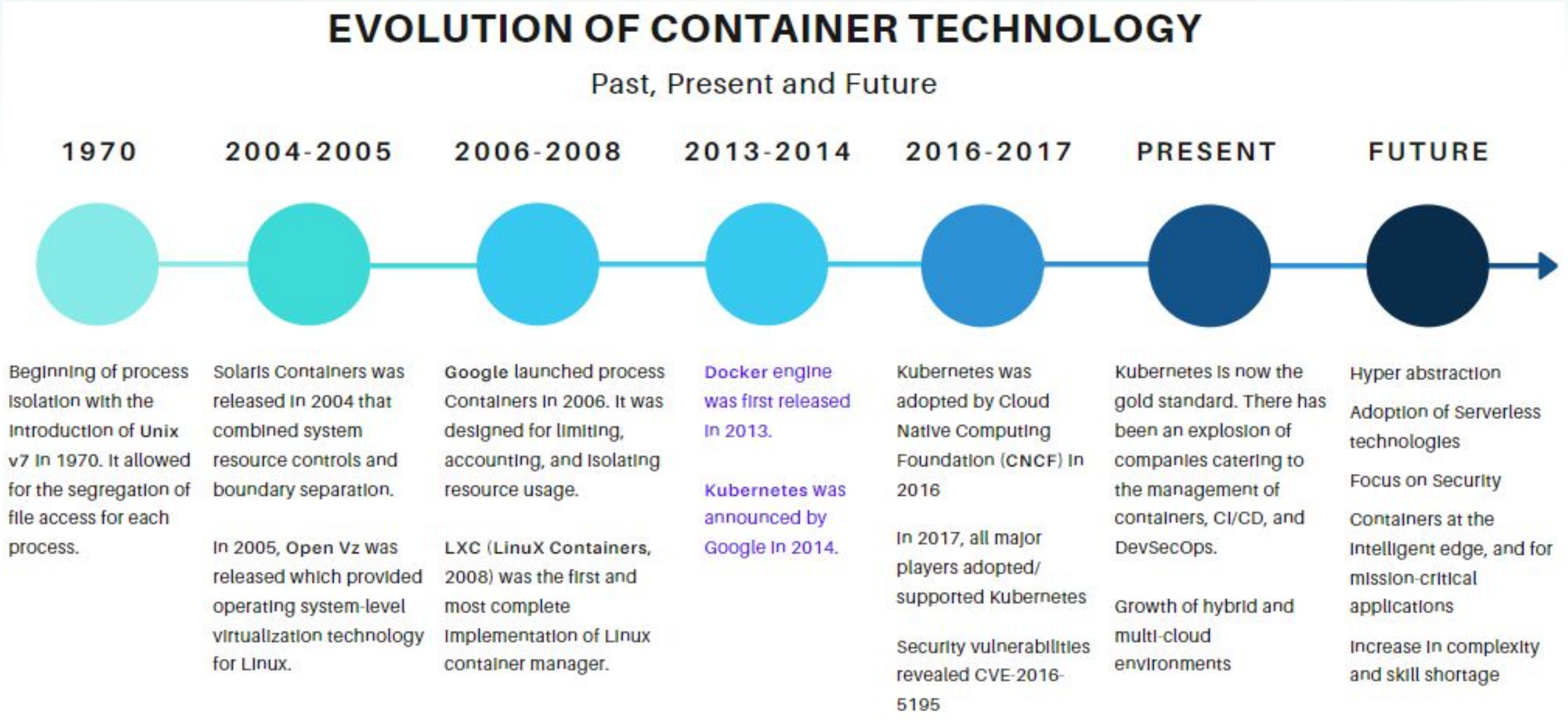
Containers make it ***easy to share CPU, memory, storage, and network resources at the operating systems level*** and offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run.

Containerization aims to provide a ***consistent and portable*** way to deliver software applications through containing the application software and its dependencies in "boxes" that can be rapidly copied and multiplied to scale up or down based on changing workload demands.

Container Technology ?

- ✔ Docker containers are the **lightweight *alternatives of the virtual machine***. It allows developers to package up the application with all its libraries and dependencies, and ship it as a single package.
- ✔ The advantage of using a ***Docker*** container is that you ***don't need to allocate any RAM and disk space*** for the applications.
- ✔ It ***automatically generates storage and space according to the application requirement***.

Container Technology ?



History of Containers

(Evolution of Containers. Past, Present, and Future)

- ✔ **1970:** saw the beginning of process isolation with the Introduction of Unix v7. It allowed for the segregation of file access for each process.
- ✔ **2004**: Solaris Containers was released. It combined system resource controls and boundary separation.
- ✔ **2005**: Open Vz was released. It provided operating system-level virtualization technology for Linux.
- ✔ **2006**: Google launched Process Containers. It was designed for limiting, accounting, and isolating resource usage.
- ✔ **2008**: LXC (LinuX Containers) was the first and most complete implementation of Linux container manager.
- ✔ **2013**: Docker engine was first released. Container use has since exploded in popularity.
- ✔ **2014**: Kubernetes was announced by Google in 2014. It is an open-source system for automating the deployment, scaling, and management of containerized applications.
- ✔ **2016**: Kubernetes was adopted by Cloud Native Computing Foundation (CNCF) in 2016. The first major security vulnerability CVE-2016-5195 was also revealed.
- ✔ **2017:** All major players such as Google, Docker, Red Hat, Microsoft, AWS, and VMware had adopted/ supported Kubernetes.

History of Containers

(Evolution of Containers. Past, Present, and Future)

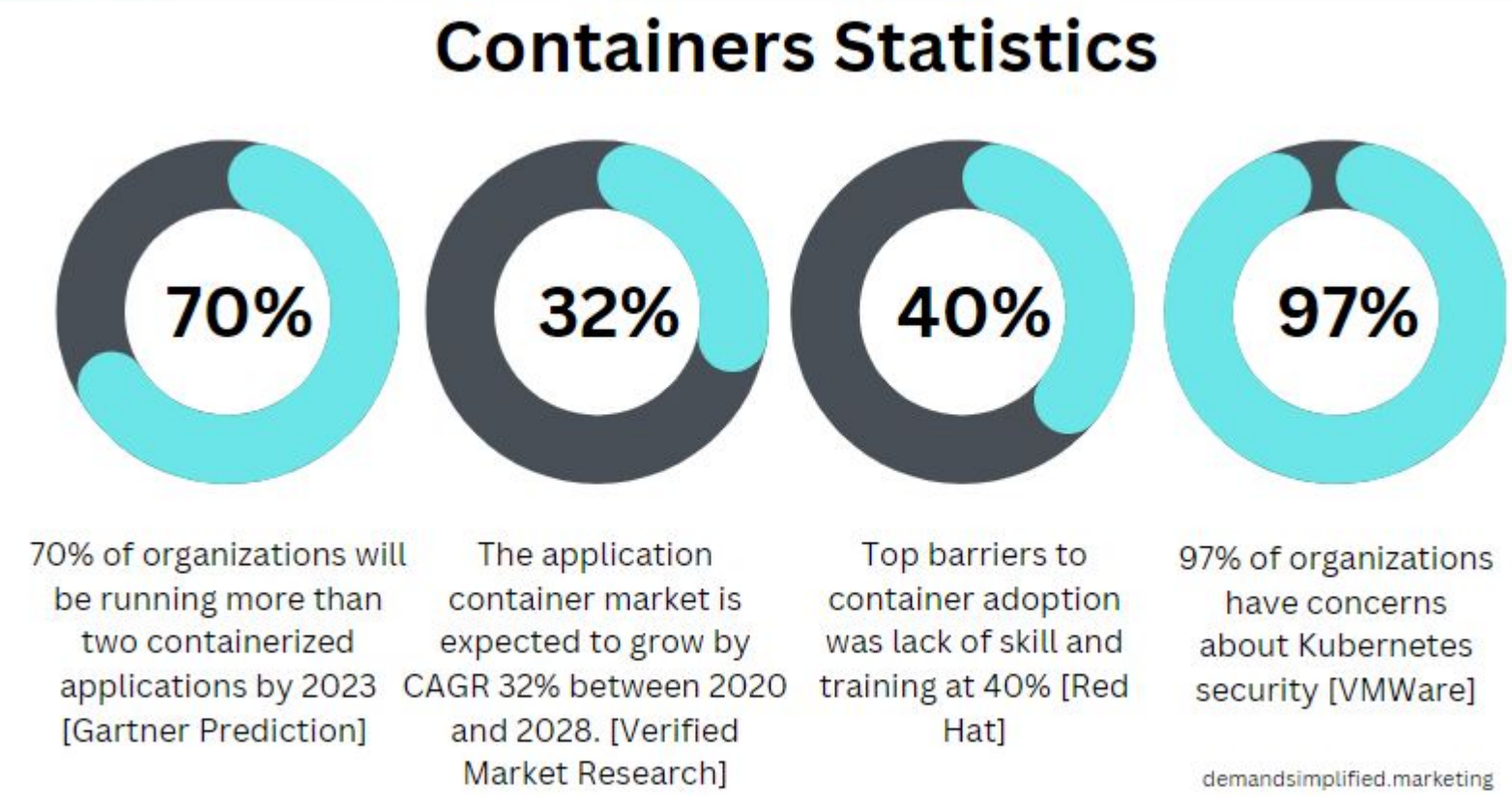
Present:

- ✔ Kubernetes is now the gold standard. There has been an explosion of companies catering to the management of containers, CI/CD, and DevSecOps.
- ✔ Growth of hybrid and multi-cloud environments

Future:

- ✔ Hyper abstraction
- ✔ Adoption of Server less technologies
- ✔ Focus on Security
- ✔ Containers at the intelligent edge, and for mission-critical applications
- ✔ Increase in complexity
- ✔ Skill shortage

Containers Statistics



Namespaces and cgroups

- ***cgroups and namespaces*** are powerful tools for managing resources and isolating processes
- They play a crucial role in ***system administration*** and ***containerization***
- *cgroups, short for control groups, allow* **administrators to limit and distribute resources** *among different groups of processes.*
- *Namespaces, on the other hand, create isolated environments for processes, separating them from the host system and other processes. Together, they provide* **a robust solution for resource management and isolation**.

Namespaces

- ✔ Namespaces are a Linux kernel feature that isolates various aspects of a process.
- ✔ They provide a process with its own isolated view of the system, such as its own file system, network, hostname, and more.
- ✔ Likewise, they allow us to create isolated environments for processes so that they can't access or affect other processes or the host system.

- ✔ **There are several types of them available in Linux, such as:**
  - ✔ **Mount**: isolates a process's view of the filesystem
  - ✔ **PID**: isolates a process's view of the process tree
  - ✔ **Network**: isolates a process's view of the network stack
  - ✔ **User**: isolates a process's view of user and group IDs

- ✔ They are ***often combined with cgroups*** to provide container isolation and resource management.

cgroups

- ✔ cgroups, or control groups, are a Linux kernel feature that enables the management and limitation of system resources like ***CPU, memory, and network bandwidth, among others***.
- ✔ We can use cgroups to set limits on these resources and distribute them among different groups of processes.
- ✔ cgroups have a hierarchical structure with root and child, each with resource limits set by controllers — for example, a CPU controller for CPU time or a memory controller for memory.
- ✔ We can use cgroups for various purposes, such as controlling resource usage in a multi-tenant environment, providing Quality of Service (QoS) guarantees, and running containers.

Conclusion

- ✔ They both serve different purposes, with cgroups providing resource management and namespaces providing isolation and security.
- ✔ In short, cgroups manage resources, and namespaces isolate and secure them.
- ✔ cgroups play a role in containerization solutions like Docker and Kubernetes, where they control container resource allocation and ensure isolation from the host and other containers.

Difference between Virtual Machines and Containers

- ✔ Virtual machines and Containers are two ways of deploying multiple, isolated services on a single platform.

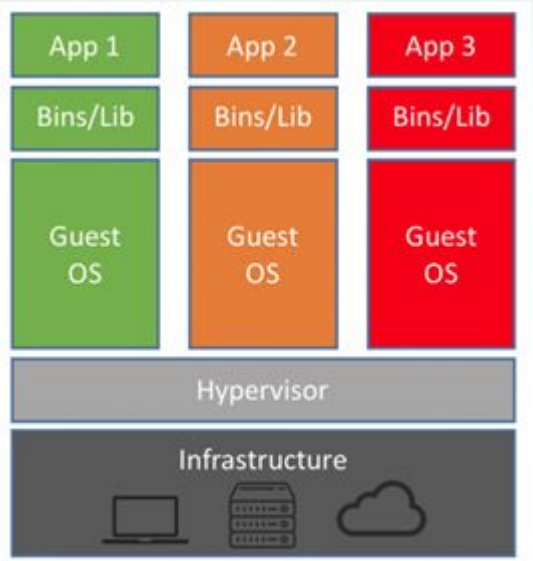
- ✔ **Virtual Machine:**

- ✔ It runs on top of an emulating software called the ***hypervisor*** which sits between the hardware and the virtual machine. ***The hypervisor is the key to enabling virtualization***. It ***manages the sharing*** of physical resources into virtual machines. Each virtual machine runs its guest operating system. They are less agile and have lower portability than containers.

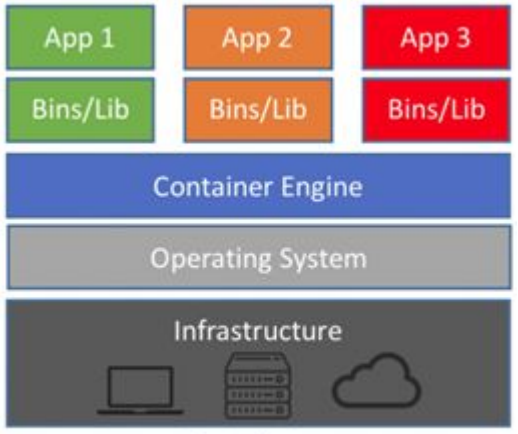
- ✔ **Container:**
- ✔ It sits on the ***top of a physical server and its host operating system***. They share a common operating system that requires care and feeding for bug fixes and patches. They are more agile and have higher portability than virtual machines.

Difference between Virtual Machines and Containers

S.No.	Virtual Machines(VM)	Containers
1	VM is a piece of software that allows you to install other software inside of it so you control it virtually as opposed to installing the software directly on the computer.	While a container is software that allows different functionalities of an application independently.
2.	Applications running on a VM system, or hypervisor, can run different OS.	While applications running in a container environment share a single OS.
3.	VM virtualizes the computer system, meaning its hardware.	While containers virtualize the operating system, or the software only.
4.	VM size is very large, generally in gigabytes.	While the size of the container is very light, generally a few hundred megabytes, though it may vary as per use.
5.	VM takes longer to run than containers, the exact time depending on the underlying hardware.	While containers take far less time to run.
6.	VM uses a lot of system memory.	While containers require very less memory.
7.	VM is more secure, as the underlying hardware isn't shared between processes.	While containers are less secure, as the virtualization is software-based, and memory is shared.
8.	VMs are useful when we require all of the OS resources to run various applications.	While containers are useful when we are required to maximize the running applications using minimal servers.

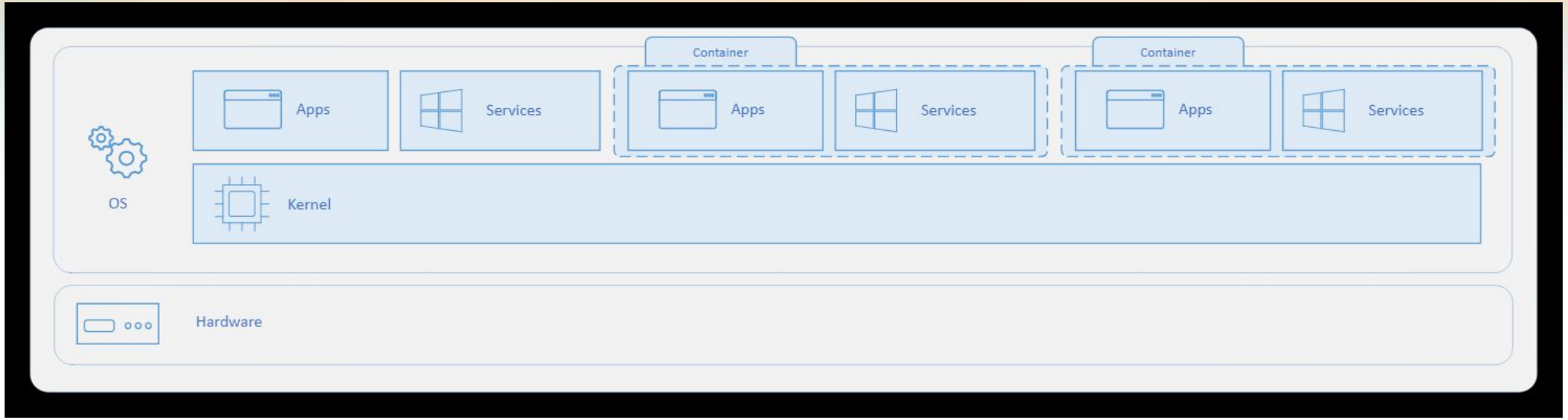


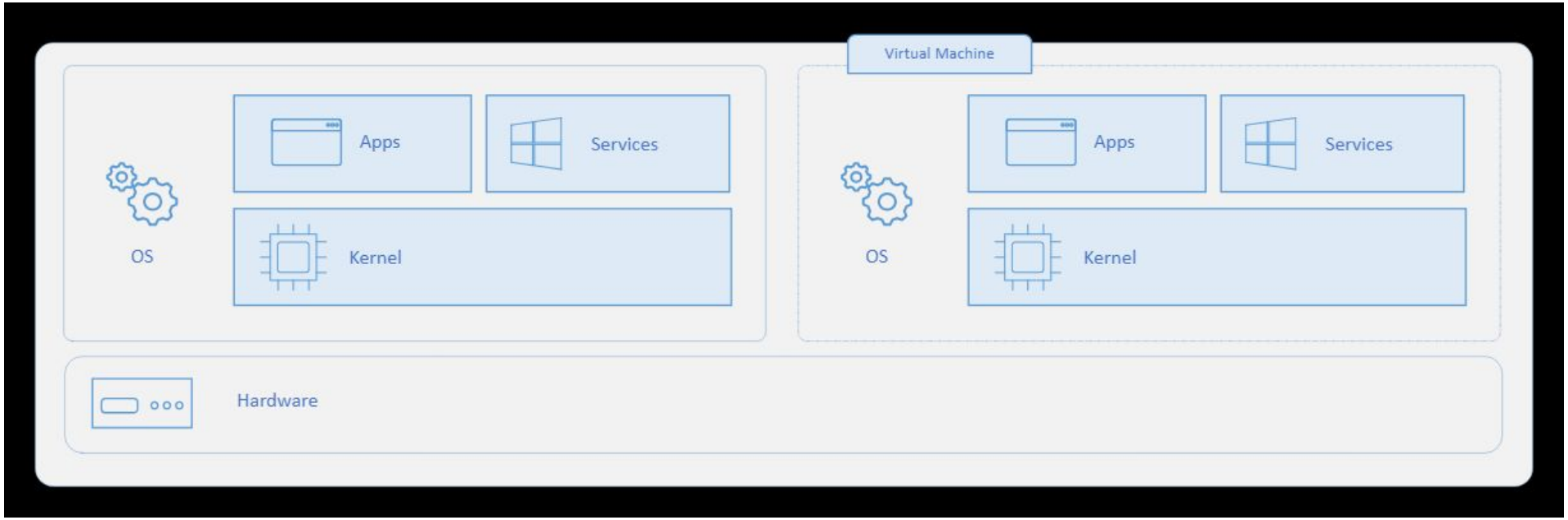
Machine Virtualization



Containers

Difference between Virtual Machines and Containers



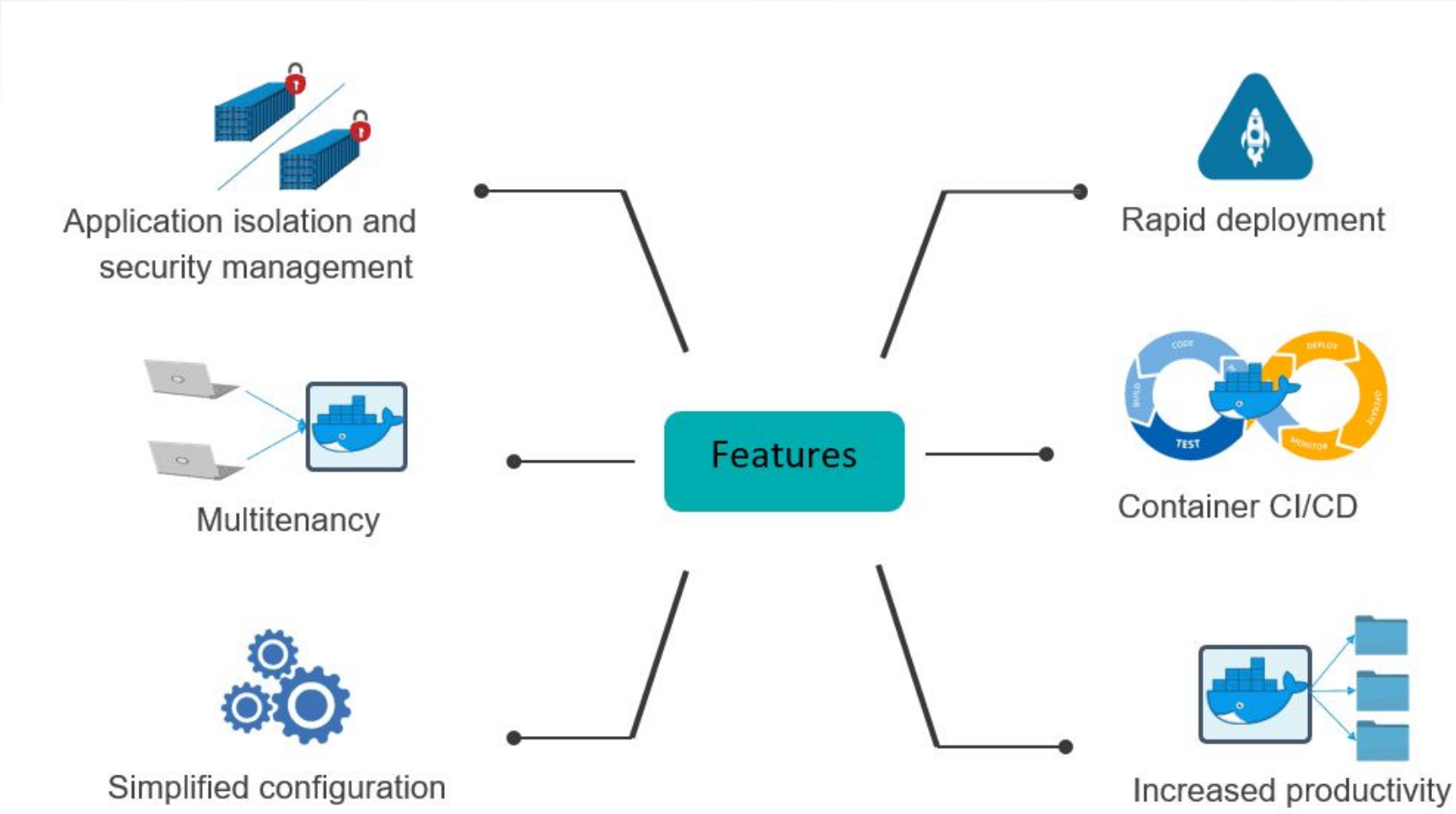


Why Do We Need Containers?

There are many reasons for using Containers but only some of them are listed below:

- **Lightweight:** Containers share the machine OS kernel and therefore they don't need a full OS instance per application. This makes the container files smaller and This is the reason why Containers are smaller in size, especially compared to virtual machines. As they are lightweight, thus they can spin up quickly and can be easily scaled horizontally.
- **Portable:** Containers are a package having all their dependencies with them, this means that we have to write the software once and the same software can be run across different laptops, cloud, and on-premises computing environments without the need of configuring the whole software again.
- **Supports CI/CD:** Due to a combination of their deployment portability/consistency across platforms and their small size, containers are an ideal fit for modern development and application patterns—such as DevOps, serverless, and microservices.
- **Improves utilization:** Containers enable developers and operators to improve CPU and memory utilization of physical machines.

Containers Features



Types of Containers

The very growth and expansion in container technology bring a large set of choices to choose from. Docker is the best known and most widely used container platform by far. But there are some more technologies on the container landscape, each with their own individual use cases and advantages.

Types of Containers

Docker

Docker is one of the most popular and widely used container platforms. It enables the creation and use of Linux containers. Docker is a tool which makes the creation, deployment and running of applications easier by using containers. Not only the Linux powers like Red Hat and Canonical have embraced Docker, but the companies like Microsoft, Amazon, and Oracle have also done it. Today, almost all IT and cloud companies have adopted Docker.

LXC

LXC is an open-source project of **LinuxContainers.org**. The aim of LXC is to provide isolated application environments that closely resemble virtual machines (VMs) but without the overhead of running their own kernel. LXC follows the Unix process model, in which there is no central daemon. So, instead of being managed by one central program, each container behaves as if it's managed by a separate program. LXC works in a number of different ways from Docker. For example, we can run more than one process in an LXC container, whereas Docker is designed in such a way that running a single process in each container is better.

CRI-O

CRI-O is an open-source tool which is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes. Its goal is to replace Docker as the Container engine for Kubernetes. It allows Kubernetes to use any OCI-compliant runtime as the container runtime for running pods. Today, it supports runc and Kata Containers as the container runtimes but any OCI-conformant runtime can be used.

Types of Containers

rkt

The rkt has a set of supported tools and community to rival Docker. rkt containers also known as Rocket, turn up from CoreOS to address security vulnerabilities in early versions of Docker. In 2014 CoreOS published the App Container Specification in an effort to drive innovation in the container space which produced a number of open-source projects. Like LXC, rkt doesn't use a central daemon and thereby provides more fine-grained control over your containers—at the individual container level. However, unlike Docker, they're not complete end-to-end solutions. But they are used with other technologies or in place of specific components of the Docker system.

Podman

Podman is an open-source container engine, which performs much of the same role as the Docker engine. But the difference between them is the way in which they work. Like rkt and LXC, Podman also does not have a central daemon but Docker follows the client/server model which is, using a daemon to manage all containers.

In Docker, if the daemon goes down, we also lose control over the containers. But in Podman, containers are self-sufficient, fully isolated environments, which we can manage independently of one another. In addition, Docker gives root permission to the container user by default, whereas non-root access is standard in Podman. Altogether, this isolation and user privilege features make Podman more secure by design.

Types of Containers

containerd

containerd is basically a daemon, supported by both Linux and Windows, that acts as an interface between your container engine and container runtimes. It provides an abstracted layer that makes it easier to manage container lifecycles, such as image transfer, container execution, snapshot functionality and certain storage operations, using simple API requests.

Similar to runC, containerd is another core building block of the Docker system that has been separated off as an independent open-source project.

Docker Hub

Sign up https://hub.docker.com/

Registry for Docker Images

What is Docker Images

- **A Stopped Container like VM Image.**
- **Consist of Multiple Layers.**
- **An app will be bundled in an Image.**
- **Container runs from images**
- **Images are called as Repositories in Registries**

- **Images become Containers when they run on Docker Engine.**

Working with Container

docker history
With this command, you can see all the commands that were run with an image via a container.

$docker history ImageID

ImageID – This is the Image ID for which you want to see all the commands that were run against it.

Eg: $docker history nginix

docker top
With this command, you can see the top processes within a container.

$docker top ContainerID

ContainerID – This is the Container ID for which you want to see the top processes.

$docker top 9f215ed0b0d3

Working with Container

docker stop
This command is used to stop a running container.

$docker stop ContainerID

ContainerID – This is the Container ID which needs to be stopped.

$docker stop 9f215ed0b0d3

The above command will stop the Docker container 9f215ed0b0d3.

docker rm
This command is used to delete a container.

$docker rm ContainerID

Eg. $docker rm 9f215ed0b0d3

The above command will remove the Docker container 9f215ed0b0d3.

Working with Container

docker stats
This command is used to provide the statistics of a running container.

$docker stats ContainerID

The output will show the CPU and Memory utilization of the Container.

$docker stats 9f215ed0b0d3 .

docker pause
This command is used to pause the processes in a running container.

$docker pause ContainerID

$sudo docker pause 07b0b6f434fe

The above command will pause the processes in a running container 07b0b6f434fe.

Working with Container

docker unpause
This command is used to unpause the processes in a running container.

$docker unpause ContainerID

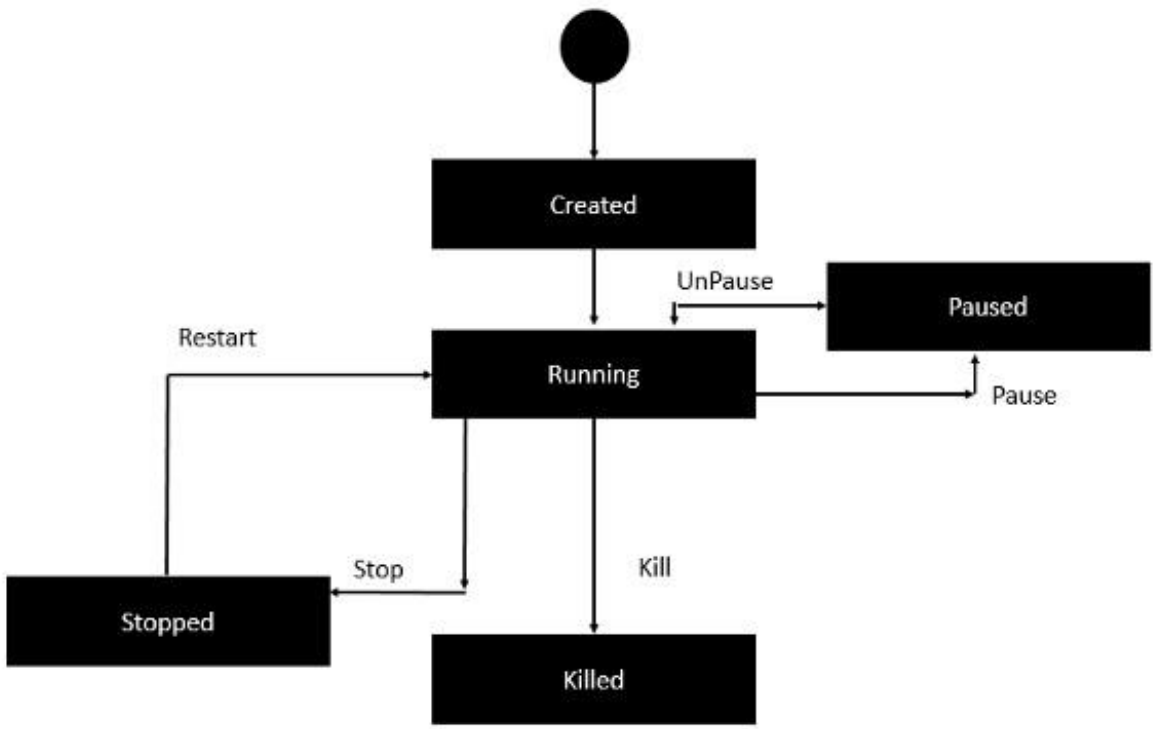
$docker unpause 07b0b6f434fe

docker kill
This command is used to kill the processes in a running container.

$docker kill ContainerID

$docker kill 07b0b6f434fe .

Docker – Container Lifecycle



- ✔ Initially, the Docker container will be in the **created** state.
- ✔ Then the Docker container goes into the running state when the Docker **run** command is used.
- ✔ The Docker **kill** command is used to kill an existing Docker container.
- ✔ The Docker **pause** command is used to pause an existing Docker container.
- ✔ The Docker **stop** command is used to pause an existing Docker container.
- ✔ The Docker **run** command is used to put a container back from a **stopped** state to a **running** state.

Docker - Configuring

service docker stop
This command is used to stop the Docker daemon process.

Syntax
$service docker stop

service docker start
This command is used to start the Docker daemon process.

Syntax
$service docker start

Docker - File

- Docker also gives you the capability to create your own Docker images, and it can be done with the help of **Docker Files**.
- A Docker File is a simple text file with instructions on how to build your images**.**
- The following steps explain how you should go about creating a Docker File.
- Step 1 − Create a file called Docker File and edit it using vim. Please note that the name of the file has to be "Dockerfile" with "D" as capital.

  $ sudo nano/vim Dockerfile
- Step 2 − Build your Docker File using the following instructions.

```
#This is a sample Image
FROM ubuntu
MAINTAINER mydypiu@gmail.com
RUN apt-get update
RUN apt-get install –y nginx
CMD [“echo”,”Image created”]
```

Docker - File

- The following points need to be noted about the above file −
- The first line "#This is a sample Image" is a comment. You can add comments to the Docker File with the help of the # command
- The next line has to start with the FROM keyword. It tells docker, from which base image you want to base your image from. In our example, we are creating an image from the ubuntu image.
- The next command is the person who is going to maintain this image. Here you specify the MAINTAINER keyword and just mention the email ID.
- The RUN command is used to run instructions against the image. In our case, we first update our Ubuntu system and then install the nginx server on our ubuntu image.
- The last command is used to display a message to the user.

Docker - File

- The following points need to be noted about the above file −
- The first line "#This is a sample Image" is a comment. You can add comments to the Docker File with the help of the # command
- The next line has to start with the FROM keyword. It tells docker, from which base image you want to base your image from. In our example, we are creating an image from the ubuntu image.
- The next command is the person who is going to maintain this image. Here you specify the MAINTAINER keyword and just mention the email ID.
- The RUN command is used to run instructions against the image. In our case, we first update our Ubuntu system and then install the nginx server on our ubuntu image.
- The last command is used to display a message to the user.

- **Step 3 − Save the file**.
- Press ctrl+o to save and for exit ctrl+x

Docker - File

docker build

- This method allows the users to build their own Docker images.
- Syntax

  $docker build -t ImageName:TagName dir

Options

- -t – is to mention a tag to the image
- ImageName – This is the name you want to give to your image.
- TagName – This is the tag you want to give to your image.
- Dir – The directory where the Docker File is present.

E.g.

$ sudo docker build –t myimage:0.1.

Here, **myimage** is the **name** we are giving to the **Image** and **0.1** is the **tag number** we are giving to our image.

Since the Docker File is in the present working directory, **we used "." at the end of the command to signify the present working directory.**

Docker - Public Repositories

Public repositories can be used to host Docker images which can be used by everyone else. An example is the images which are available in Docker Hub. Most of the images such as Centos, Ubuntu, and Jenkins are all publicly available for all. We can also make our images available by publishing it to the public repository on Docker Hub.

Step 1 – Log into Docker Hub and create your repository. This is the repository where your image will be stored. Go to https://hub.docker.com/ and log in with your credentials.

Step 2 – Click the button "Create Repository" on the above screen and create a repository with the name ***demorep***. Make sure that the visibility of the repository is public.

Once the repository is created, make a note of the ***pull*** command which is attached to the repository.

Step 3 – Now go back to the Docker Host. Here we need to tag our ***myimage*** to the new repository created in ***Docker Hub***. We can do this via the Docker tag command.

Docker - Public Repositories

Step 4 – Issue the Docker login command to login into the Docker Hub repository from the command prompt. The

Docker login command will prompt you for the username and password to the Docker Hub repository.

```
demo@ubuntudemo:~$ sudo docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: demousr
Password:
Login Succeeded
demo@ubuntudemo:~$
```

docker tag
This method allows one to tag an image to the relevant repository.

Syntax
$docker tag imageID Repositoryname
$docker tag ab0c1d3744dd demousr/demorep:1.0

Options
imageID – This is the ImageID which needs to be tagged to the repository.

Repositoryname – This is the repository name to which the ImageID needs to be tagged to.

Docker - Public Repositories

docker push

This method allows one to push images to the Docker Hub.

Syntax

$docker push Repositoryname

Options

Repositoryname − This is the repository name which needs to be pushed to the Docker Hub.

Return Value

The long ID of the repository pushed to Docker Hub.

Example

$docker push demousr/demorep:1.0

Docker - Public Repositories

docker push

This method allows one to push images to the Docker Hub.

Syntax

$docker push Repositoryname

Options

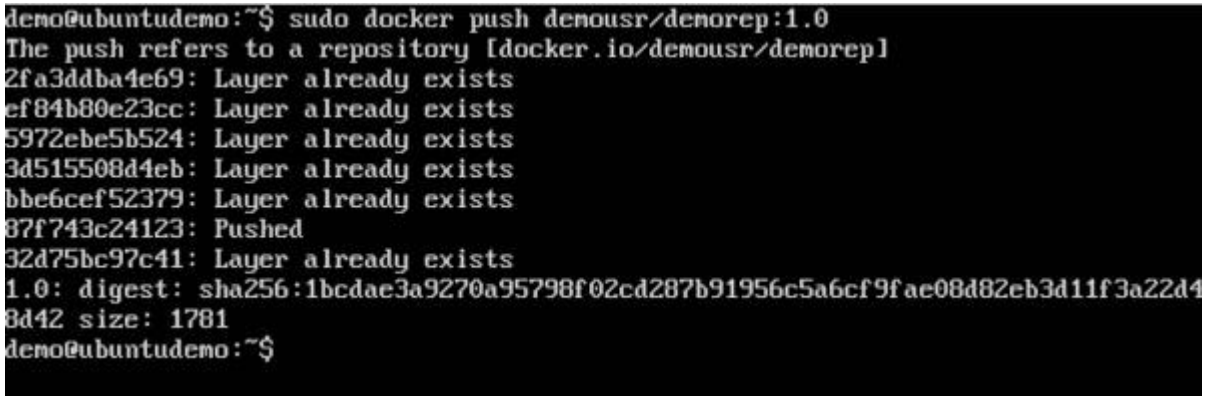
Repositoryname – This is the repository name which needs to be pushed to the Docker Hub.

Return Value

The long ID of the repository pushed to Docker Hub.

Example

$docker push demousr/demorep:1.0



Docker - Public Repositories

If you go back to the Docker Hub page and go to your repository, you will see the tag name in the repository.

Now let's try to pull the repository we uploaded onto our Docker host.

Let's first delete the images, myimage:0.1 and demousr/demorep:1.0, from the local Docker host.

($docker image rm [OPTIONS] IMAGE [IMAGE...])

Let's use the Docker pull command to pull the repository from the Docker Hub.

Docker - Public Repositories

If you go back to the Docker Hub page and go to your repository, you will see the tag name in the repository.

Now let's try to pull the repository we uploaded onto our Docker host.

Let's first delete the images, myimage:0.1 and demousr/demorep:1.0, from the local Docker host.

($docker image rm [OPTIONS] IMAGE [IMAGE...])

Let's use the Docker pull command to pull the repository from the Docker Hub.

Docker - Managing Ports

- In Docker, the containers themselves can have applications running on ports.
- When you run a container, if you want to access the application in the container via a port number, you need to map the port number of the container to the port number of the Docker host.
- We are going to download the Jenkins container from Docker Hub. We are then going to map the Jenkins port number to the port number on the Docker host.
- Step 1 – First, you need to do a simple sign-up on Docker Hub.
- Step 2 – Once you have signed up, you will be logged into Docker Hub.
- Step 3 – Next, let's browse and find the Jenkins image.
- Step 4 – If you scroll down on the same page, you can see the Docker ***pull*** command. This will be used to download the Jenkins Image onto the local Ubuntu server.
- Step 5 – Now go to the Ubuntu server and run the command – ***$sudo docker pull jenkins***
- ***Step 6*** – To understand what ports are exposed by the container, you should use the Docker inspect command to inspect the image. $docker inspect Container/Image

Docker - Managing Ports

- The output of the inspect command gives a JSON output. If we observe the output, we can see that there is a section of "ExposedPorts" .
- One is the ***data port*** of 8080 and the other is the ***control port*** of 50000.
- To run Jenkins and map the ports, you need to change the Docker run command and add the ‘p’ option which specifies the port mapping. So, you need to run the following command −

$ sudo docker run -p 8080:8080 -p 50000:50000 Jenkins

The left-hand side of the port number mapping is the Docker host port to map to and the right-hand side is the Docker container port number.

When you open the browser and navigate to the Docker host on port 8080, you will see Jenkins up and running.

Docker - Private Registries

- You might have the need to have your own private repositories.
- You may not want to host the repositories on Docker Hub.
- For this, there is a ***repository*** container itself from Docker.
- **Step 1** – Use the Docker run command to download the private registry. This can be done using the following command. ***$sudo docker run –d –p 5000:5000 –-name registry registry:2***
- The following points need to be noted about the above command –
- Registry is the container managed by Docker which can be used to host private repositories.
- The port number exposed by the container is 5000. Hence with the –p command, we are mapping the same port number to the 5000 port number on our localhost.
- We are just tagging the registry container as "2", to differentiate it on the Docker host.
- The –d option is used to run the container in detached mode. This is so that the container can run in the background
- **Step 2** – Let's do a ***docker ps*** to see that the registry container is indeed running. We have now confirmed that the registry container is indeed running.
- Step 3 – Now let's tag one of our existing images so that we can push it to our local repository. In our example, since we have the ***centos*** image available locally, we are going to tag it to our private repository and add a tag name of ***centos***.

Docker - Private Registries

- ***$sudo docker tag 67591570dd29 localhost:5000/centos***
- ***The following points need to be noted about the above command*** −
- 67591570dd29 refers to the Image ID for the centos image.
- localhost:5000 is the location of our private repository.
- We are tagging the repository name as centos in our private repository.
- ***Step 4*** − Now let's use the Docker push command to push the repository to our private repository.
- ***$sudo docker push localhost:5000/centos***
- Here, we are pushing the centos image to the private repository hosted at ***localhost:5000.***
- ***Step 5*** − Now let's delete the local images we have for centos using the ***docker rmi*** commands. We can then download the required centos image from our private repository.
- ***$sudo docker rmi centos:latest***
- ***$sudo docker rmi 67591570dd29***

Docker - Private Registries

- ***Step 6*** – Now that we don't have any **centos** images on our local machine, we can now use the following Docker **pull** command to **pull** the **centos** image from our private repository.
- **$sudo docker pull localhost:5000/centos**
- Here, we are pulling the **centos** image to the private repository hosted at **localhost:5000.**