-: Containers: Docker, Swarm mode :-

Containers are an operating system virtualization technology used to package applications and their dependencies and run them in isolated environments. They provide a lightweight method of packaging and deploying applications in a standardized way across many different types of infrastructure.

Containers are created from **container images**: bundles that represent the system, applications, and environment of the container. Container images act like templates for creating specific containers, and the same image can be used to spawn any number of running containers.

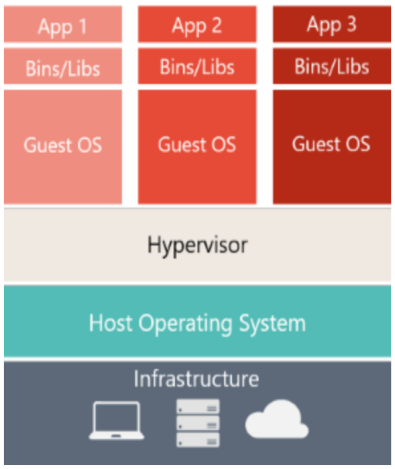
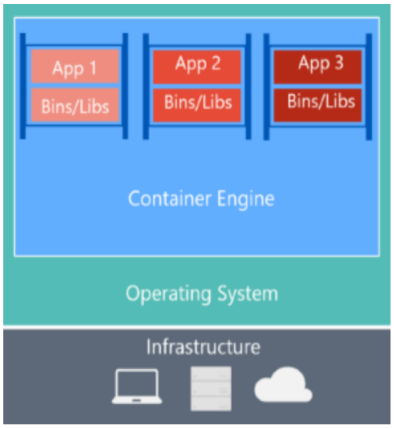
Users can download pre-built container from external sources or build their own images customized to their needs.

Containers are small, fast, and portable because unlike a virtual machine, containers do not need include a guest OS in every instance and can, instead, simply leverage the features and resources of the host OS.

Benefits of containers

The primary advantage of containers, especially compared to a VM, is providing a level of abstraction that makes them lightweight and portable.

- **Lightweight:** Containers share the machine OS kernel, making container files small and easy on resources. They can spin up quickly.
- **Portable and platform independent:** Containers carry all their dependencies with them, meaning that software can be written once and then run without needing to be re-configured across laptops, cloud, and on-premises computing environments.
- **Supports modern development and architecture:** 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—that are built as regular code deployments in small increments.
- **Improves utilization**: Like VMs before them, containers enable developers and operators to improve CPU and memory utilization of physical machines. They also enable microservice architectures, and application components can be deployed and scaled more granularly.

Virtual Machines	Docker Containers
	
Virtual machines include the application, the required libraries or binaries, and a full guest operating system. Full virtualization requires more resources than containerization.	Containers include the application and all its dependencies. However, they share the OS kernel with other containers, running as isolated processes in user space on the host operating system. (Except in Hyper-V containers, where each container runs inside of a special virtual machine per container.)

Use cases for containers

Containers are becoming increasingly prominent, especially in cloud environments. Many organizations are even considering containers as a replacement of VMs as the general purpose compute platform for their applications and workloads.

- **Microservices**: Containers are small and lightweight, which makes them a good for microservice architectures where applications are constructed of many, loosely coupled and independently deployable smaller services.
- **DevOps**
- **Hybrid, multi-cloud**
- **Application modernizing and migration**: Containerizing applications

Containerization

Software needs to be designed and packaged differently in order to take advantage of containers—a process commonly referred to as containerization.

When containerizing an application, the process includes packaging an application with its relevant environment variables, configuration files, libraries, and software dependencies. The result is a container image that can then be run on a container platform.

Docker

Docker is a tool that allows developers, sys-admins etc. to easily deploy their applications in a sandbox (called *containers*) to run on the host operating system i.e. Linux. The key benefit of Docker is that it allows users to **package an application with all of its dependencies into a standardized unit** for software development. Unlike virtual machines, containers do not have high overhead and hence enable more efficient usage of the underlying system and resources.

Docker is an open-source project based on Linux containers. It uses Linux Kernel features like namespaces and control groups to create containers on top of an operating system.

Docker is by far the most common way of running building and running containers. **Docker** is a set of tools that allow users to create container images, push or pull images from external registries, and run and manage containers in many different environments.

The `docker` command line tool plays many roles. It runs and manages containers, acting as a process manager for container workloads. It can create new container images by reading and executing commands from `Dockerfile` or by taking snapshots of containers that are already running. The command can also interact with Docker Hub, a container image registry, to pull down new container images or to push up local images to save or publish them.

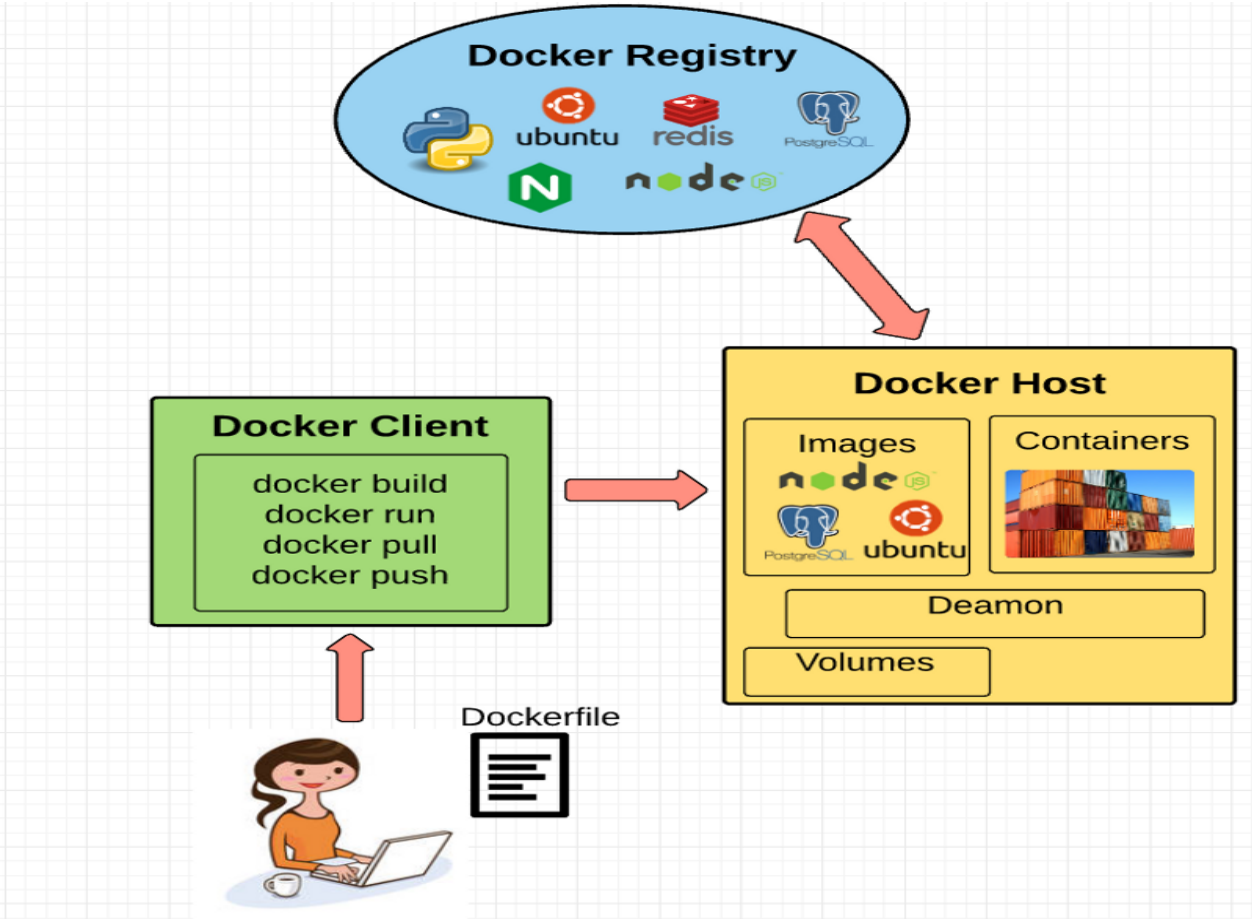
So why is Docker all of a sudden gaining steam?

1. **Ease of use:** Docker has made it much easier for anyone — developers, systems admins, architects and others — to take advantage of containers in order to quickly build and test portable applications. It allows anyone to package an application on their laptop, which in turn can run unmodified on any public cloud, private cloud, or even bare metal. The mantra is: "build once, run anywhere."

2. **Speed:** Docker containers are very lightweight and fast. Since containers are just sandboxed environments running on the kernel, they take up fewer resources. You can create and run a Docker container in seconds, compared to VMs which might take longer because they have to boot up a full virtual operating system every time.

3. **Docker Hub:** Docker users also benefit from the increasingly rich ecosystem of Docker Hub, which you can think of as an "app store for Docker images." Docker Hub has tens of thousands of public images created by the community that are readily available for use.

4. **Modularity and Scalability:** Docker makes it easy to break out your application's functionality into individual containers. For example, you might have your Postgres database running in one container and your Redis server in another while your Node.js app is in another. With Docker, it's become easier to link these containers together to create your application, making it easy to scale or update components independently in the future.



Docker Engine

Docker engine is the layer on which Docker runs. It's a lightweight runtime and tooling that manages containers, images, builds, and more. It runs natively on Linux systems and is made up of:

1. A Docker Daemon that runs in the host computer.
2. A Docker Client that then communicates with the Docker Daemon to execute commands.
3. A REST API for interacting with the Docker Daemon remotely.

Docker Client

The Docker Client is what you, as the end-user of Docker, communicate with. Think of it as the UI for Docker. For example, when you do…

you are communicating to the Docker Client, which then communicates your instructions to the Docker Daemon.

Docker Daemon

The Docker daemon is what actually executes commands sent to the Docker Client — like building, running, and distributing your containers. The Docker Daemon runs on the host machine, but as a user, you never communicate directly with the Daemon. The Docker Client can run on the host machine as well, but it's not required to. It can run on a different machine and communicate with the Docker Daemon that's running on the host machine.

Dockerfile

A Dockerfile is where you write the instructions to build a Docker image. These instructions can be:

- **RUN apt-get y install some-package**: to install a software package
- **EXPOSE 8000:** to expose a port
- **ENV ANT_HOME /usr/local/apache-ant** to pass an environment variable

and so forth. Once you've got your Dockerfile set up, you can use the **docker build** command to build an image from it. Here's an example of a Dockerfile:

Docker Image

Images are read-only templates that you build from a set of instructions written in your Dockerfile. Images define both what you want your packaged application and its dependencies to look like *and* what processes to run when it's launched.

The Docker image is built using a Dockerfile. Each instruction in the Dockerfile adds a new "layer" to the image, with layers representing a portion of the images file system that either adds to or replaces the layer below it. Layers are key to Docker's lightweight yet powerful structure. Docker uses a Union File System to achieve this:

Union File Systems

Docker uses Union File Systems to build up an image. You can think of a Union File System as a stackable file system, meaning files and directories of separate file systems (known as branches) can be transparently overlaid to form a single file system.

The contents of directories which have the same path within the overlaid branches are seen as a single merged directory, which avoids the need to create separate copies of each layer. Instead, they can all be given pointers to the same resource; when certain layers need to be modified, it'll create a copy and modify a local copy, leaving the original unchanged. That's how file systems can *appear* writable without actually allowing writes. (In other words, a "copy-on-write" system.)

Layered systems offer two main benefits:

1. **Duplication-free:** layers help avoid duplicating a complete set of files every time you use an image to create and run a new container, making instantiation of docker containers very fast and cheap.
2. **Layer segregation:** Making a change is much faster — when you change an image, Docker only propagates the updates to the layer that was changed.

Volumes

Volumes are the "data" part of a container, initialized when a container is created. Volumes allow you to persist and share a container's data. Data volumes are separate from the default Union File System and exist as normal directories and files on the host filesystem. So, even if you destroy, update, or rebuild your container, the data volumes will remain untouched. When you want to update a volume, you make changes to it directly. (As an added bonus, data volumes can be shared and reused among multiple containers, which is pretty neat.)

Docker Containers

A Docker container, as discussed above, wraps an application's software into an invisible box with everything the application needs to run. That includes the operating system, application code, runtime, system tools, system libraries, and etc. Docker containers are built off Docker images. Since images are read-only, Docker adds a read-write file system over the read-only file system of the image to create a container.

Moreover, then creating the container, Docker creates a network interface so that the container can talk to the local host, attaches an available IP address to the container, and executes the process that you specified to run your application when defining the image.

Once you've successfully created a container, you can then run it in any environment without having to make changes.

Docker Install

```
1) Go TO https://hub.docker.com/
2) Sign Up and create login
3) Login using credentials created above
4) Download and install docker  desktop application for your OS -
   GUI
5) Install docker, docker-compose, docker-machine from CLI using
   apt-get
6) Start the docker app
```

Docker Trial run

1) Docker login
2) docker run --interactive --tty alpine:3.10
    a. This will drop you into a Alpine ash shell inside of a container as the root user of that container. When you're done, just run exit or hit CTRL+D.
    docker run -it alpine:3.10 => Short form
3) docker run alpine:3.10 ls
    a. Run ls command inside docker
4) Clean up containers once done using
    a. docker rm <IDs>
    b. docker rm $(docker ps -a -q -f status=exited)

Docker File

A Dockerfile is a text document that contains the instructions to assemble a Docker image. When we tell Docker to build our image by executing the docker build command, Docker reads these instructions, executes them, and creates a Docker image as a result.

Sample Dockerfile content:

```
FROM python:3
# set a directory for the app
WORKDIR /usr/src/app
# copy all the files to the container
COPY . .
# install dependencies
RUN pip3 install --no-cache-dir -r requirements.txt
# define the port number the container should expose
EXPOSE 5000
# run the command
CMD ["python3", "./app.py"]
```

Now, Build, Run, Push Docker image

```
docker build -t <docker_user>/<build_tag> -f Dockerfile .
docker push <docker_user>/<build_tag>
docker run <docker_user>/<build_tag>
docker exec <docker_user>/<build_tag> ls
docker ps -a
```

Docker : Swarm mode

A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to join together in a cluster. The activities of the cluster are controlled by a swarm manager, and machines that have joined the cluster are referred to as nodes.

Docker Swarm mode is built into the Docker Engine.
a cluster of Docker Engines called a *swarm*.

Docker swarm is a container orchestration tool, meaning that it allows the user to manage multiple containers deployed across multiple host machines.

One of the key benefits associated with the operation of a docker swarm is the high level of availability offered for applications. In a docker swarm, there are typically several worker nodes and at least one manager node that is responsible for handling the worker nodes' resources efficiently and ensuring that the cluster operates efficiently.

Swarm Commands: docker swarm <swarm_command>

init, join, leave, ca, unlock, update

Docker Swarm Manager Node

The primary function of manager nodes is to assign tasks to worker nodes in the swarm. Manager nodes also help to carry out some of the managerial tasks needed to operate the swarm. Docker recommends a maximum of seven manager nodes for a swarm.

Docker Swarm Leader Node

When a cluster is established, the Raft consensus algorithm is used to assign one of them as the "leader node". The leader node makes all of the swarm management and task orchestration decisions for the swarm. If the leader node becomes unavailable due to an outage or failure, a new leader node can be elected using the Raft consensus algorithm.

Docker Swarm Worker Node

In a docker swarm with numerous hosts, each worker node functions by receiving and executing the tasks that are allocated to it by manager nodes. By default, all manager modes are also worker nodes and are capable of executing tasks when they have the resources available to do so.

Docker Swarm Benefits

Leverage the Power of Containers

Developers love using docker swarm because it fully leverages the design advantages offered by containers. Containers allow developers to deploy applications or services in self-contained virtual environments, a task that was previous the domain of virtual machines. Containers are proving a more lightweight version of virtual machines, as their architecture allows them to make more efficient use of computing power.

Docker Swarm Helps Guarantee High Service Availability

One of the main benefits of docker swarms is increasing application availability through redundancy. In order to function, a docker swarm must have a swarm manager that can assign tasks to worker nodes. By implementing multiple managers, developers ensure that the system can continue to function even if one of the manager nodes fails. Docker recommends a maximum of seven manager nodes for each cluster.

Automated Load-Balancing

Docker swarm schedules tasks using a variety of methodologies to ensure that there are enough resources available for all of the containers. Through a process that can be described as automated load balancing, the swarm manager ensures that container workloads are assigned to run on the most appropriate host for optimal efficiency.

https://www.simplilearn.com/tutorials/docker-tutorial/docker-swarm

Kubernetes

What is Kubernetes?

Kubernetes is developed by the community with the intent of addressing container scalability and management needs. In the early days of Kubernetes, the community contributors leveraged their knowledge of creating and running internal tools, such as Borg and Omega. With the advent of the Cloud Native Computing Foundation (CNCF) in partnership with the Linux Foundation, the community adopted Open Governance for Kubernetes. IBM, as a founding member of CNCF, actively contributes to CNCF's cloud-native projects, along with other companies like Google, Red Hat, Microsoft, and Amazon.

Kubernetes is an open source container-management tool for those important containers and their complex production workloads. With Kubernetes, developers and DevOps teams can schedule, deploy, manage, and discover highly available apps by using the flexibility of clusters. A Kubernetes cluster is made up of compute hosts that are called worker nodes. These worker nodes are managed by a Kubernetes master that controls and monitors all resources in the cluster. A node can be a virtual machine or physical, bare metal machine.

Pros of Kubernetes

- Open-source community that is very active in developing the code base
- Fast-growing KubeCon conferences throughout the year that are more than doubling attendance numbers
- Battle-tested by big players like Google and our own IBM workloads and runs on most operating systems
- Largest adoption in the market
- Available on the public cloud or for on-premises — managed or non-managed offerings from all the big cloud providers (IBM Cloud, AWS, Microsoft Azure, Google Cloud Platform, etc.)
- Broad Kubernetes support from an ecosystem of cloud tool vendors, such as Sysdig, LogDNA, and Portworx (among many others)
- Key functionalities include service discovery, ingress and load balancing, self-healing, storage orchestration, horizontal scalability, automated rollouts and rollbacks, and batch execution
- Unified set of APIs and strong guarantees about the cluster state

Kubernetes is an open-source platform created by Google for container deployment operations, scaling up and down, and automation across the clusters of hosts. This production-ready, enterprise-grade, self-healing (auto-scaling, auto-replication, auto-restart, auto-placement) platform is modular, and so it can be utilized for any architecture deployment.

Kubernetes also distributes the load amongst containers. It aims to relieve the tools and components from the problem faced due to running applications in private and public clouds by placing the containers into groups and naming them as logical units. Their power lies in easy scaling, environment agnostic portability, and flexible growth.