Introduction to Docker

Prepared by:
Ms. Avita Katal
Assistant Professor (SG)
School of Computer Science
UPES, Dehradun

Objectives

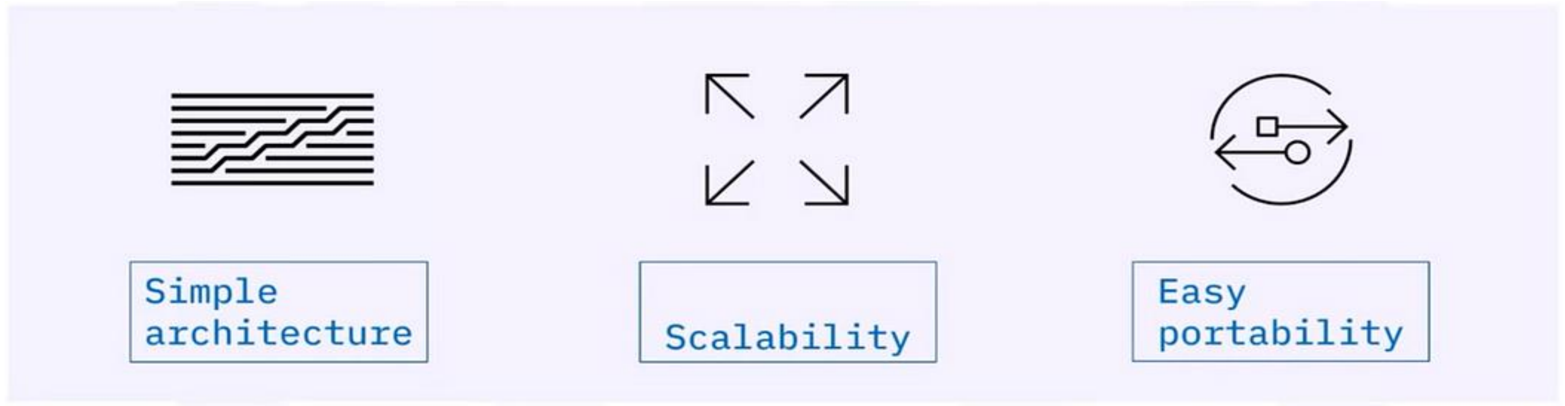
- Define Docker
- Describe the Docker architectures
- List the benefits of Docker containers
- Identify and understand Docker objects.

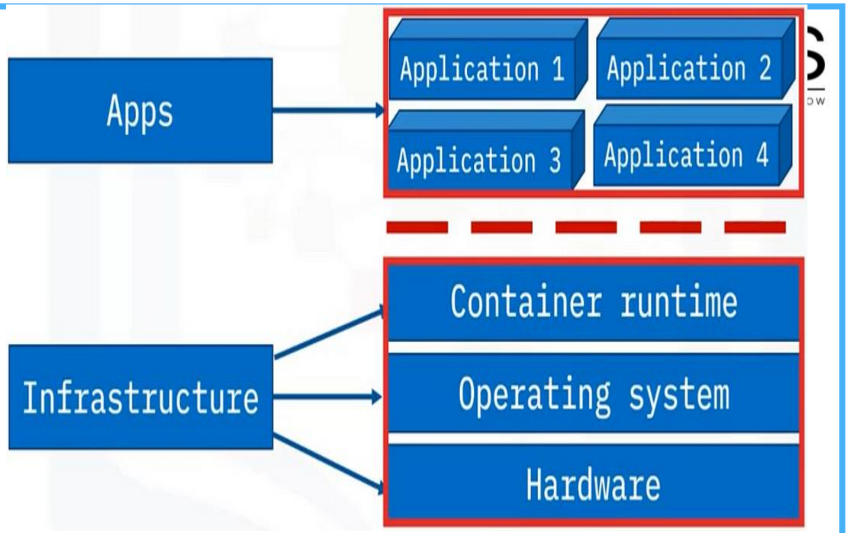
Docker Defined

Available since 2013, the official Docker definition, paraphrased, states that,

Docker provides **tooling and a platform to manage the lifecycle of your containers**:

- **Develop** your application and its supporting components using containers.
- The container becomes the **unit for distributing and testing your application.**
- When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your **production environment is a local data center, a cloud provider, or a hybrid of the two.**





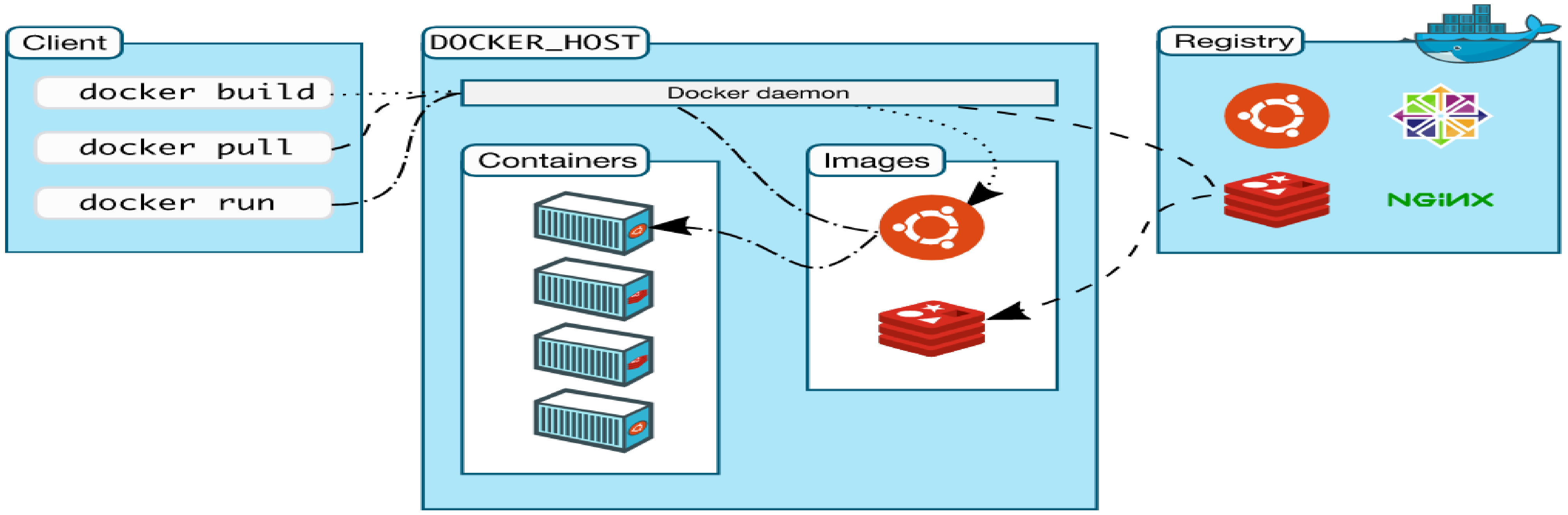
- Docker became popular with developers because of its **simple architecture, massive scalability, and portability on multiple platforms, environments, and locations.**
- Docker **isolates applications** from infrastructure, including the hardware, the operating system, and the container runtime.
- Docker is written in the **Go programming language and uses Linux kernel features** to deliver its functionality.
- Docker also uses **namespaces to provide an isolated workspace** called the container. It creates a set of namespaces for every container and each aspect runs in a separate namespace with access limited to that namespace.
- Docker inspired the development of complementary tools such as **Docker CLI, Docker Compose, and Prometheus, as well as various plugins for storage**. It also led to the creation of orchestration technologies like **Docker Swarm or Kubernetes** and development methodologies using **microservices and serverless.**

Docker offers the following benefits:

- Isolated and consistent environments result in **stable application deployments**.
- **Deployments occur in seconds** because Docker images are small, reusable, and can be used across multiple projects.
- Docker's **automation capabilities** help eliminate errors, simplifying the maintenance cycle.
- Docker supports **Agile and CI/CD DevOps practices**.
- Docker's **easy versioning speeds up testing, rollbacks, and redeployments**.
- Docker **helps segment applications for an easy refresh, cleanup, and repair**.
- Developers collaborate to **resolve issues faster** and **scale containers when needed.**
- Images are portable across platforms, giving developers and IT teams **greater flexibility**. Ultimately, this leads to **less downtime in the field and faster resolution of errors** — a win-win for everyone involved.

Docker is not a good fit for applications that require high performance or security, they're built around monolithic architectures, they use rich GUI features, or they perform standard desktop or limited functions.

Docker architecture



Docker uses a **client-server architecture.** The Docker client talks to the Docker daemon, which does the heavy lifting of **building, running, and distributing your Docker containers**. The **Docker client and daemon can run on the same system**, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a **REST API, over UNIX sockets or a network interface.** Another Docker client is **Docker Compose**, that lets you work with applications consisting of a set of containers.

The Docker daemon

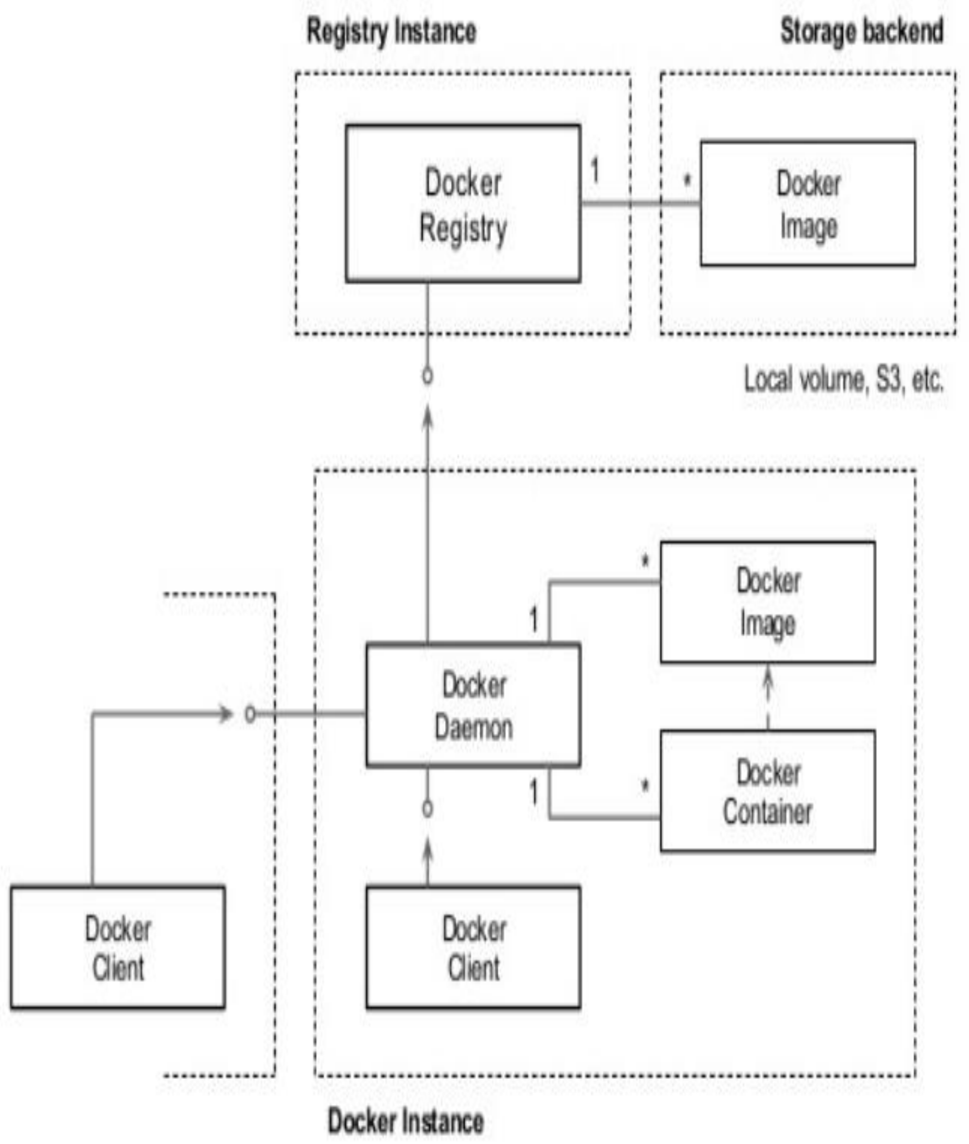
The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as **images, containers, networks, and volumes**. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the **primary way that many Docker users interact with Docker**. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The **Docker client can communicate with more than one daemon.**

Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to **build and share containerized applications and microservices**. Docker Desktop includes the **Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.**



Docker registries

A Docker *registry* stores **Docker images**. **Docker Hub is a public registry** that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run **your own private registry.**

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker objects

When you use Docker, you are creating and using **images, containers, networks, volumes, plugins, and other objects.**

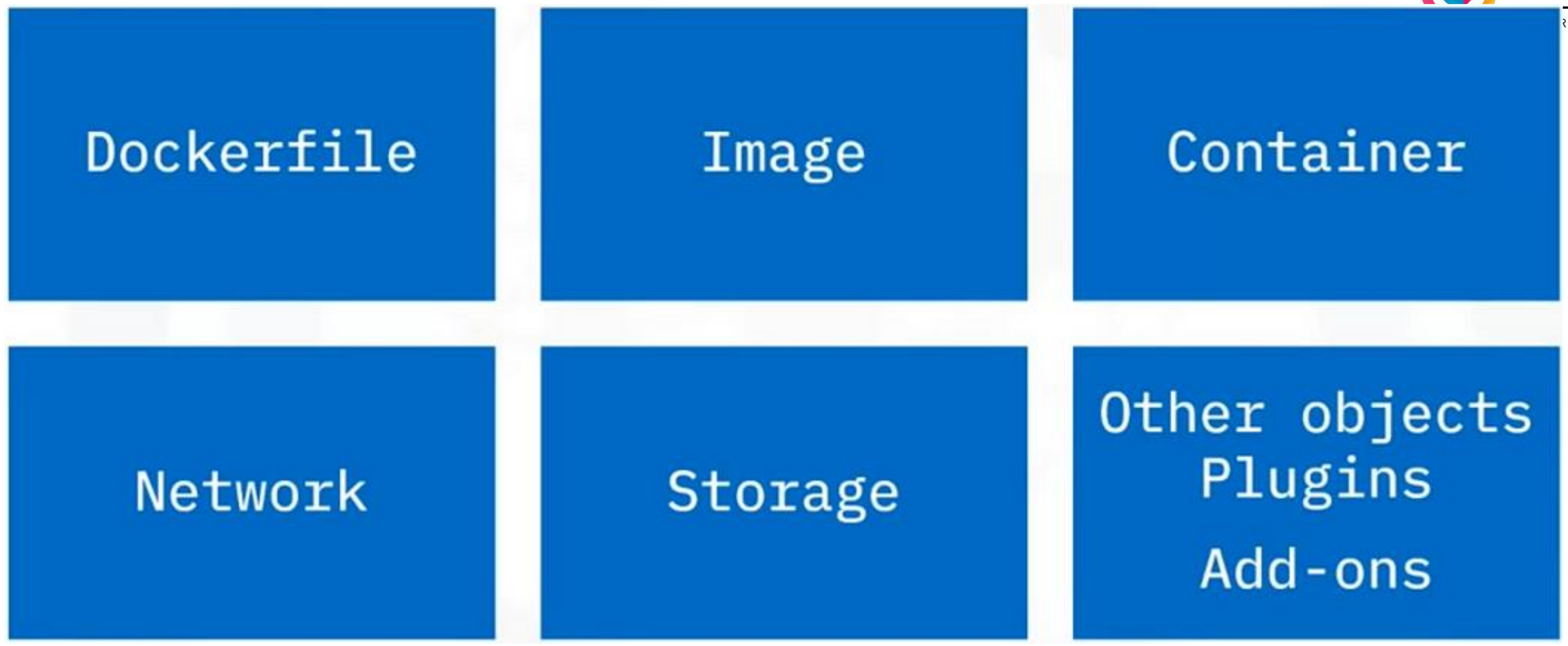
Docker Hub

- Docker Hub is a service provided by Docker for **finding and sharing container images**.
- It's the **world's largest repository of container images** with an array of content sources including **container community developers, open source projects, and independent software vendors (ISV)** building and distributing their code in containers.

What key features are included in Docker Hub?

Repositories: Push and pull container images.
Docker Official Images: Pull and use high-quality container images **provided by Docker**.
Docker Verified Publisher Images: Pull and use high-quality container images **provided by external vendors**.
Docker-Sponsored Open Source Images: Pull and use high-quality container images **from non-commercial open source projects.**
Builds: Automatically **build container images from GitHub and Bitbucket** and push them to Docker Hub.
Webhooks: **Trigger actions** after a successful **push to a repository to integrate Docker Hub** with other services.
Docker Hub CLI tool (currently experimental) and **an API that allows you to interact with Docker Hub.**

Docker Objects



- Docker is a tool that manages the **building, shipping, and running of applications inside software containers.**
- It consists of objects such as the **Dockerfile, images, container, network, storage volumes, and other entities such as plugins and add-ons.**

Images

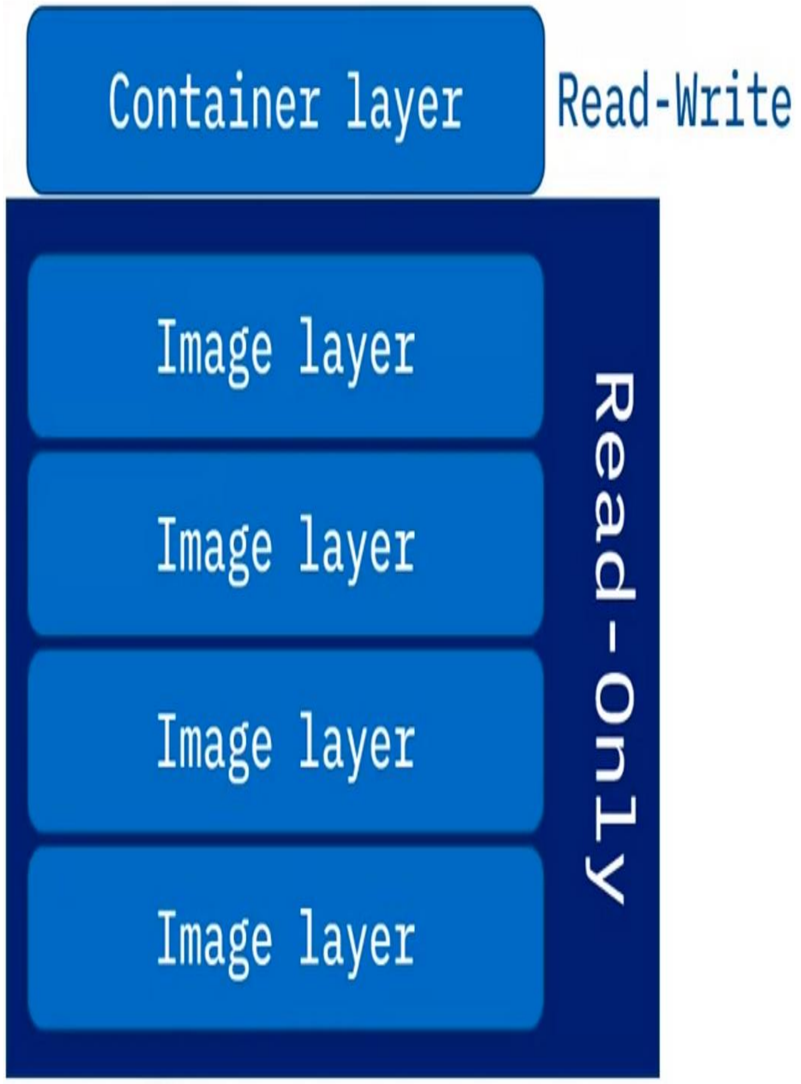
- An image is a **read-only template** with **instructions for creating a Docker container.**

- Often, an image is **based on another image, with some additional customization**. For example, you may build an image which is based on the **ubuntu image**, but installs the **Apache web server** and your **application, as well as the configuration details** needed to make your application run.

- You might **create your own images** or you might only use those created by others and published in a registry.

- To build your own image, you create a **Dockerfile with a simple syntax** for defining the steps needed to **create the image and run it**. Each instruction in a Dockerfile creates a layer in the image.

- When you change the **Dockerfile and rebuild the image, only those layers which have changed are rebuilt.** This is part of what makes images so **lightweight, small, and fast**, when compared to other virtualization technologies.



Containers

- A container is a **runnable instance of an image.**

- You can **create, start, stop, move, or delete** a container using the **Docker API or CLI**. You can **connect a container to one or more networks, attach storage to it, or even create a new image based on its current state**.

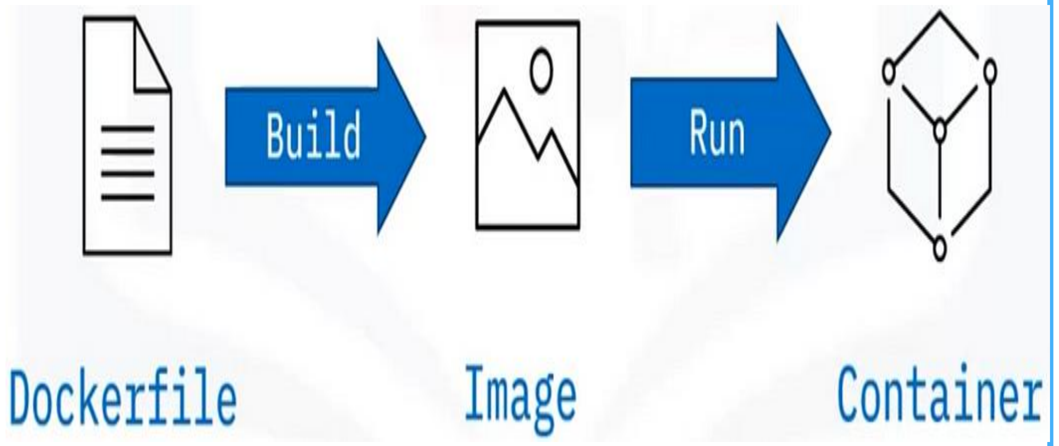
- By default, a container is relatively well isolated from other containers and its host machine.

- You can control how **isolated a container's network, storage, or other underlying subsystems are from other containers** or from the host machine.

- A container is defined by its image as well as any configuration options you provide to it when you create or start it. **When a container is removed, any changes to its state that are not stored in persistent storage disappear.** But you can use **volumes and bind mounts to persist data** even after a container stops. Plugins, such as storage plugins, provide the ability to connect to external storage platforms.

Dockerfile

A Dockerfile is a **text file that contains instructions needed to create an image**. You can create a Dockerfile using any editor from the console or terminal. Here are several of the essential education you can include in your Dockerfile.



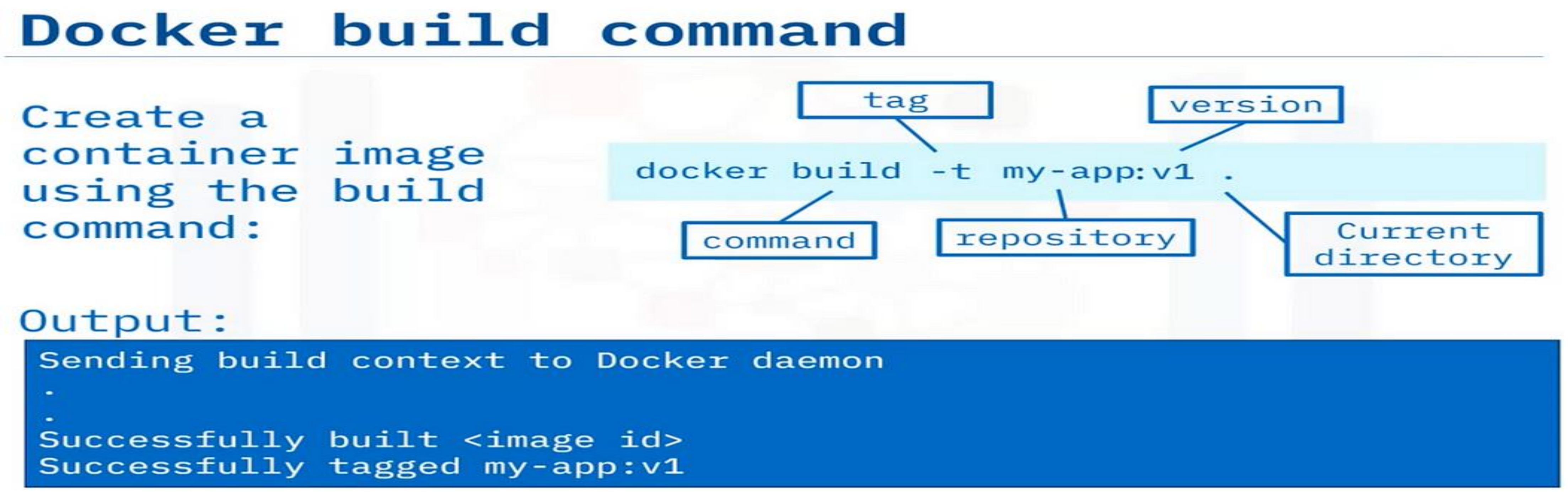
FROM
A Dockerfile must always begin with a FROM instruction **that defines a base image.** The most common base image is an **operating system** or a specific language like **Go or Node.js.**
RUN
The RUN command **executes commands.**
CMD
The CMD instruction sets the **default command for execution**. Therefore, a Dockerfile should have only one CMD instruction. If a Dockerfile **has several CMD instructions, only the last one will take effect.**

## Steps to create and run containers:
1. Create a Dockerfile
2. Use the Dockerfile to create a container image
3. Use the container image to create a running container

Dockerfile Example

```
FROM busybox:latest
MAINTAINER avita katal(avita207@gmail.com)
CMD ["echo", "HelloWorld"]
CMD ["date"] # only this one runs. Last CMD
CMD ["sh","-c","echo hello world;date;ls -l"]
# allows you to execute all the commands passes
#this is a comment
#ENTRY POINT
ENTRYPOINT ["/bin/cat"]
# used to configure the executables that will always run after the container is initiated.
CMD ["etc/psswd"]
```

```
mkdir images
cd image
notepad Dockerfile
docker build -t mybusybox:1.0 .
docker run mybusybox:1.0
docker run -it mybusybox:1.1 /bin/sh
```



Docker build command

Create a container image using the build command:

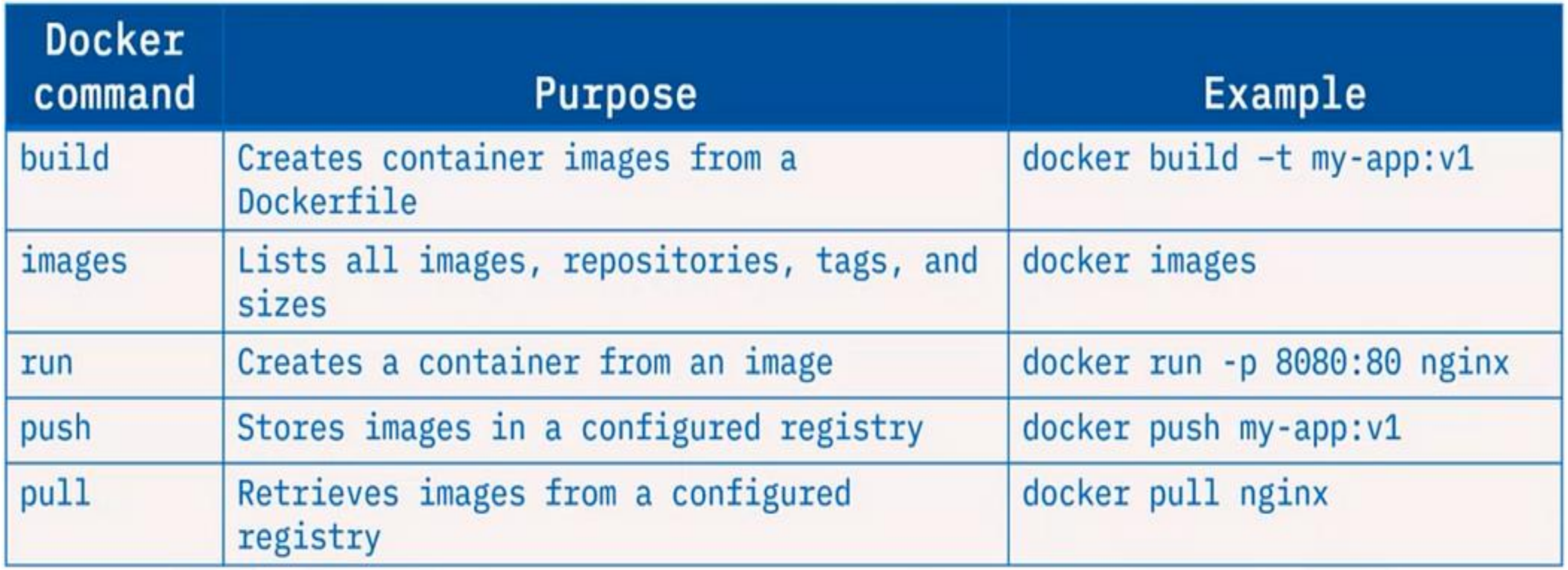
```
docker build -t my-app:v1 .
```

Output:

```
Sending build context to Docker daemon
.
.
Successfully built <image id>
Successfully tagged my-app:v1
```

- A Docker build command builds an image using the build command**, a tag, the repository, the version, and the current directory.**

- The output messages include: Sending build context to Docker daemon “Successfully built <image id>” and “Successfully tagged my-app:v1”



Docker command	Purpose	Example
build	Creates container images from a Dockerfile	docker build –t my-app:v1
images	Lists all images, repositories, tags, and sizes	docker images
run	Creates a container from an image	docker run -p 8080:80 nginx
push	Stores images in a configured registry	docker push my-app:v1
pull	Retrieves images from a configured registry	docker pull nginx

Let’s learn how images are named.

When naming images, the name has three parts: the hostname, repository, and tag.

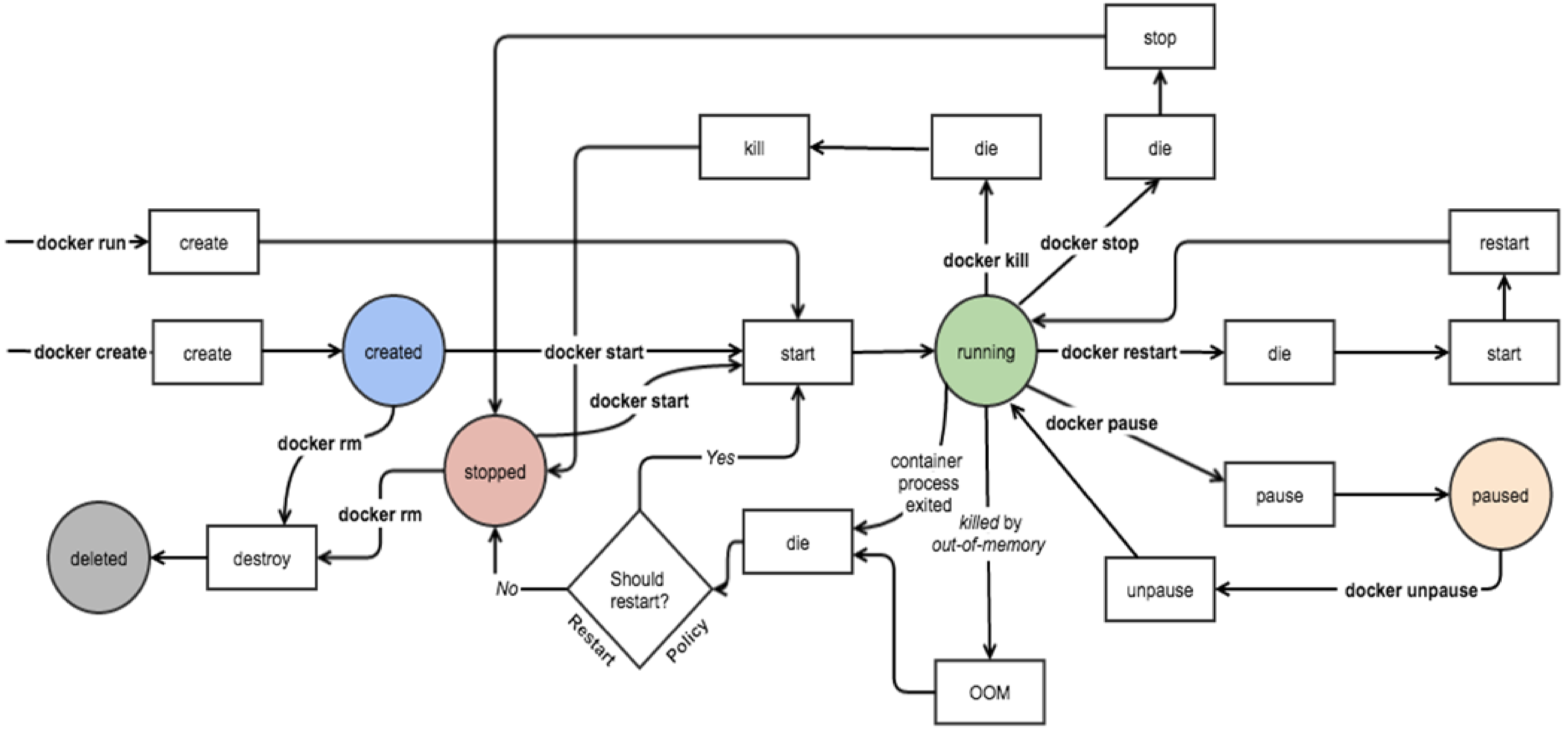
hostname/repository:tag

hostname: The hostname identifies the image registry.
repository: A repository is a group of related container images.
tag: The tag provides information about an image’s specific version or variant.
Consider the image name **docker.io/ubuntu:18.04**:

docker.io/ubuntu:18.04

hostname: The hostname **docker.io** refers to the Docker Hub registry. When using the **Docker CLI**, you can exclude the docker.io hostname.
repository: The repository name ubuntu indicates an Ubuntu image.
tag: Finally, the tag, shown here as 18.04, represents the installed Ubuntu version.

Docker Life cycle



Life cycle-Demo Create

```
#   docker create –t --name=ubucon ubuntu; docker ps -a
#   docker start ubucon;docker ps -a
#   docker attach ubucon;docker ps -a
#   Cont + c or cont+p+q;docker ps -a
#   docker kill ubucon;docker ps -a
#   docker start ubucon;docker ps -a
#   docker stop ubucon;docker ps -a
#   docker restart ubucon;docker ps -a
#   docker start ubucon;docker ps -a
#   docker pause ubucon;docker ps -a
#   docker unpause ubucon;docker ps -a
#   docker rm ubucon;docker ps –a
#   docker stats # Shows usage statistics for only running containers
#   docker stats -a # Shows usage for all containers
#   docker container prune
#   docker container inspect <container name or container id>
```

Life cycle-Demo Run

```
# docker run –dt --name=ubucon ubuntu; docker ps –a //start

   automatically

# docker attach ubucon;docker ps -a

# Cont + c or cont+p+q;docker ps -a

# docker kill ubucon;docker ps -a

# docker start ubucon;docker ps -a

# docker stop ubucon;docker ps -a

# docker restart ubucon;docker ps -a

# docker start ubucon;docker ps -a

# docker pause ubucon;docker ps -a

# docker unpause ubucon;docker ps -a

# docker rm ubucon;docker ps -a
```

UPES
UNIVERSITY OF TOMORROW

Thank You