POD 7: CONTAINERIZATION

By: Duy Nguyen and Iker Santana

WHAT IS CONTAINERIZATION

Containerization is a technology that involves packaging an application and its dependencies into a container, allowing it to run consistently across different computers.

A container is a lightweight, stand-alone, and executable package that includes everything needed to run an application: code, runtime, libraries, and system settings.

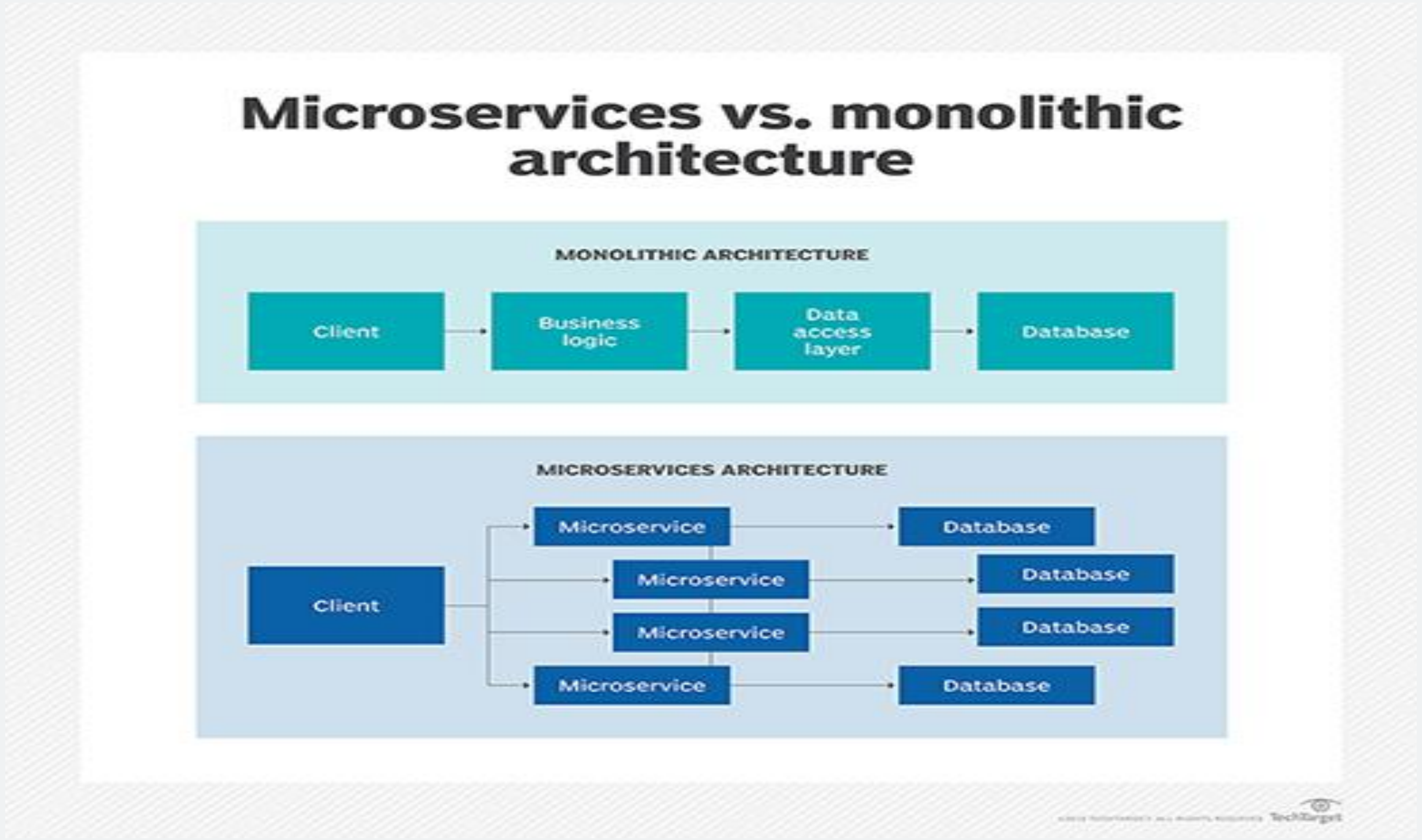
BENEFITS OF CONTAINERS

- Isolation: Containers run isolated from the host system and other containers. They have their own file system, processes, and network configurations.
- Consistency: Since the environment is bundled with the container, applications behave the same way, regardless of where they are deployed (e.g., local machines, servers, or cloud environments).
- Efficiency: Containers are lightweight compared to virtual machines because they share the host system’s kernel, which reduces overhead (data usage).

USES OF CONTAINERIZATION

- Microservice architecture vs Monolithic architecture
- Continuous Integration / Continuous Development
- Software as a Service
- Safety while working with bigger groups

MONOLITHIC VS MICROSERVICES

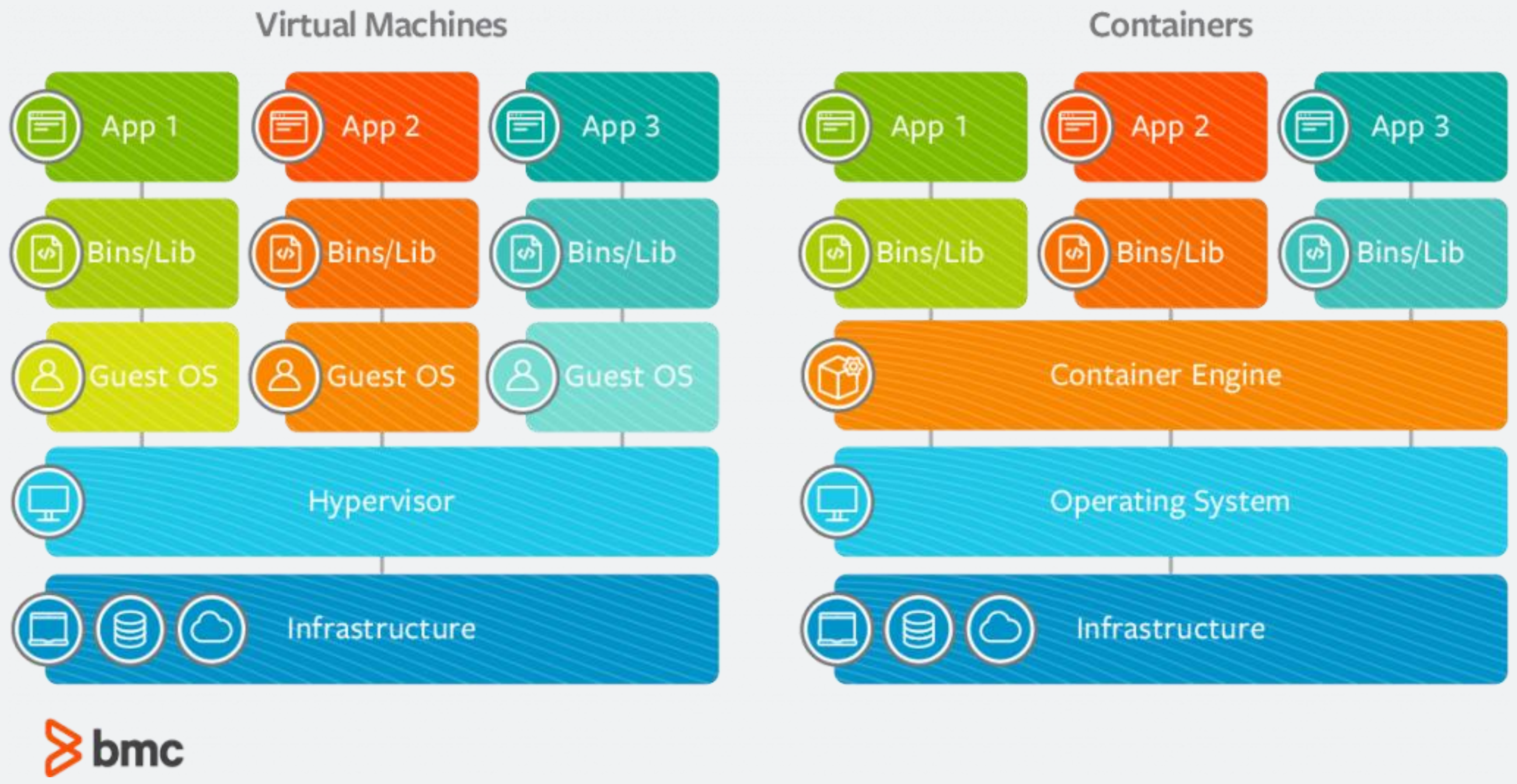


VIRTUAL MACHINES VS CONTAINERS

- Each VM runs a full operating system (OS), which could be different from the host OS (e.g., running Linux on a Windows host or vice versa).More flexibility in terms of the OS you can run, but comes with increased resource overhead.

- Containers share the **same OS kernel** as the host. VMs can only run applications that are compatible with the host OS. While containers share the host OS, they are isolated from one another and have their own user space.

VIRTUAL MACHINES VS CONTAINERS



DOCKER

- **Docker** is a platform that simplifies the process of developing, shipping, and running applications in containers.
- **Isolates applications** in lightweight, portable containers.
- Ensures **consistent behavior** across different systems (development, testing, production).
- Allows for **fast startup** and **resource-efficient** execution of applications.
- Works well with **microservices architectures** and is highly suitable for **cloud-native** applications.
- Integrates with orchestration tools like **Docker Swarm** and **Kubernetes** for managing containers at a large scale.

CONTAINER ORCHESTRATION

- **Container orchestration** is the automated management of containerized applications and maintains containers across multiple hosts. Manual management of containers becomes more difficult as the container architecture grows into a larger scale.
- **Deployment**: Automatically starting containers on various hosts.
- **Scaling**: Adjusting the number of running containers based on demand.
- **Self-healing**: Automatically restarting failed containers or migrating them if needed.
- **Updates and Rollbacks**: Ensuring seamless updates to applications while minimizing downtime.

TOOLS FOR CONTAINER ORCHESTRATION

- **Kubernetes** is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications.
- **AWS Fargate** automates deployment of containers and deals with CI/CD pipelines.
- **AWS Lambda** is an event-driven program.

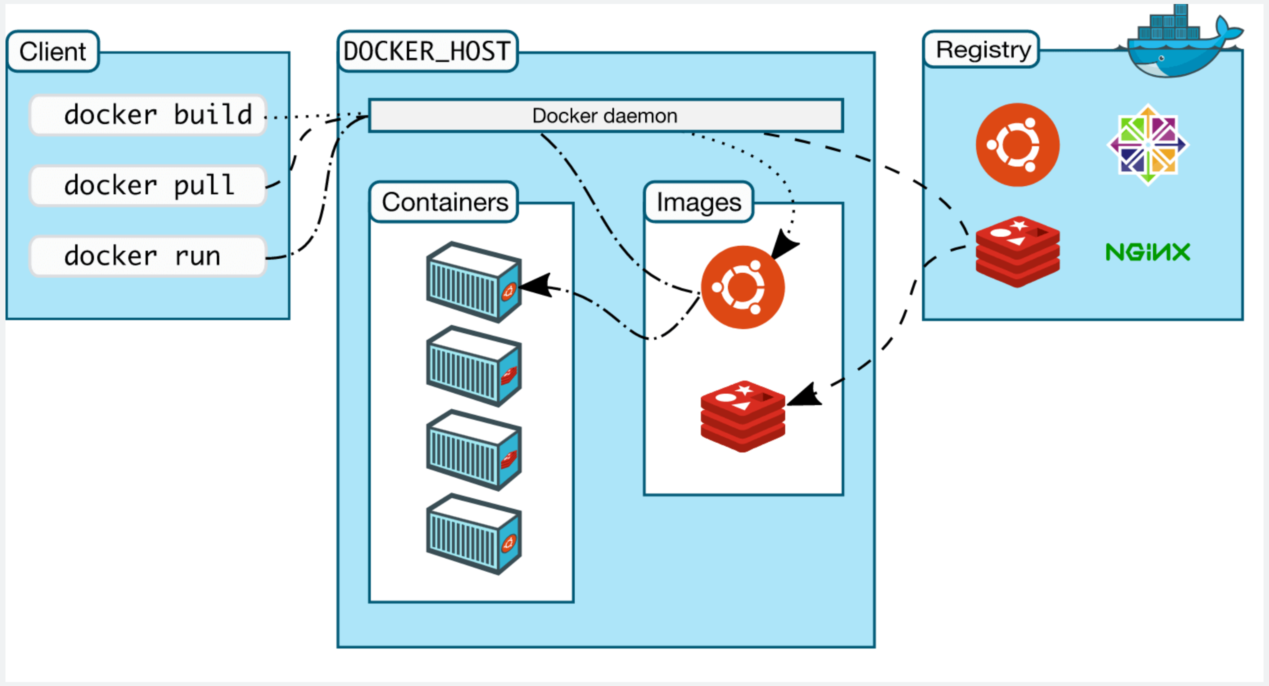
List of Container Orchestration Tools

INTRODUCTION TO DOCKER

- Open platform for developing, shipping, and running applications.
- Allows us to manage our infrastructure in the same ways we manage our application.
- Packages and runs applications in a loosely and isolated environment (Container).

DOCKER ARCHITECTURE

- Client-server architecture.
- **Docker Client:** Primary way that Docker users interact with Docker. Client use commands such as **docker run** to communicate with the Docker Daemon.
- **Docker Daemon:** Listens for Docker API request and manages Docker Objects such as images, containers, networks, and volumes.
- The Docker Client talks to the Docker Daemon. They both (client and daemon) run on the same system. There are cases where you can connect your docker client to a remote docker daemon.



DOCKER ARCHITECTURE

- Docker Registries: A storage and distribution system for Docker images. A docker registry is organized into Docker repositories. It allows users to pull/push images to/from the registry. It also works as a version control tool for the images. By default, docker interacts with DockerHub, but there are other public registries such as Amazon Elastic Container Registry (ECR), Google container Registry (GCR), etc.
- Docker Desktop: An easy-to-install application for Linux, Mac and Windows that allows you to build and share containerized applications. It includes the Docker Daemon, the Docker Client, Docker Compose, and more components.

DOCKER IMAGES

- An image is a read-only **template** (or "prototype/blueprints").
- Includes all the files, env variables, binaries, libraries and configurations or create and run a container.
- Images can be based on another image with additional customization.
- Make use of a **Dockerfile** to build your own images. In this file we explicitly define the steps needed to create and run the image.

DOCKER IMAGES

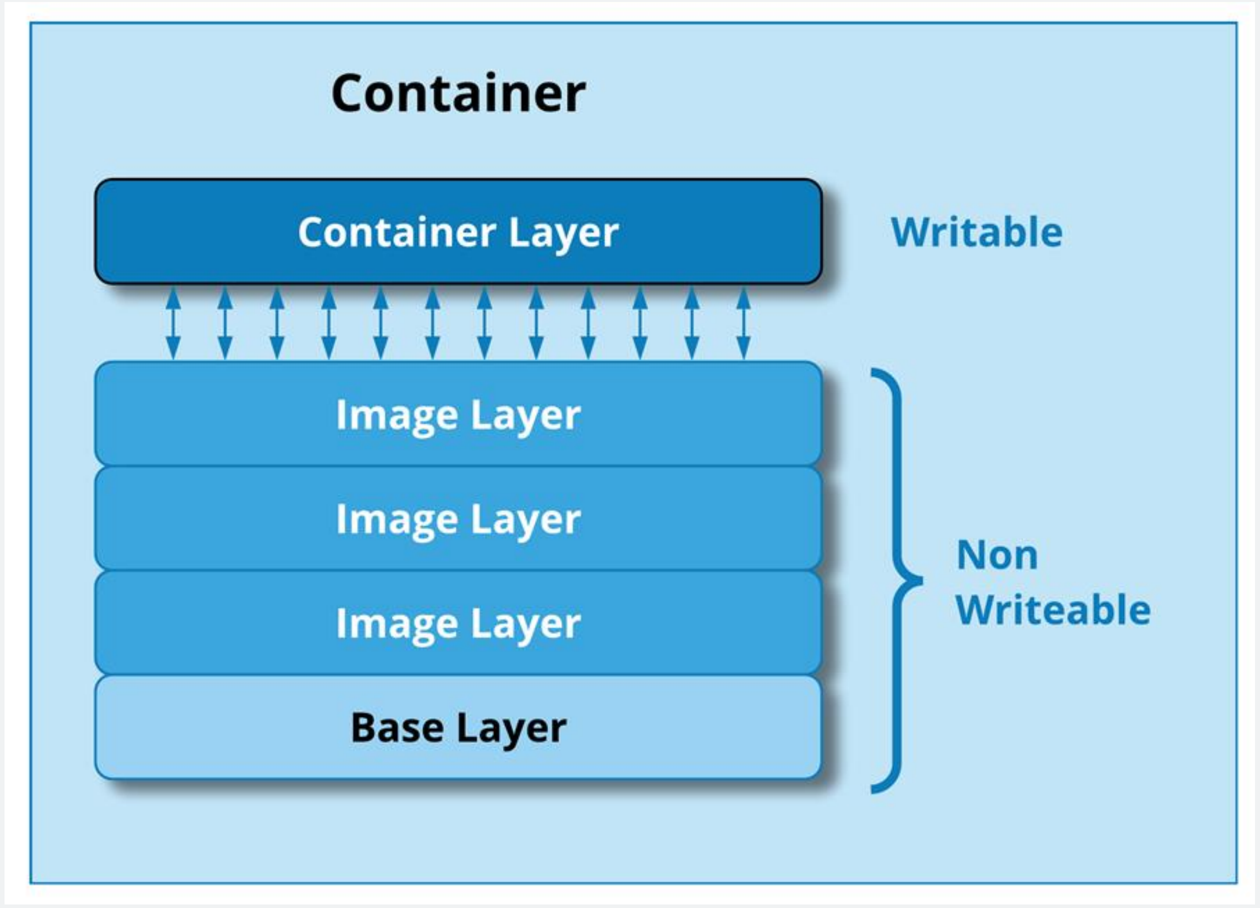
Two important principles:

1. **Immutable.** We are not allowed to modify an image once the image is created. We can only make a new image or add changes on top of it.
2. **Composed of layers.** Each layer represents modifications in the image.

DOCKER IMAGES

Layers

- Layers stacked on top of each other.
- Layers let you extend images of others by reusing their base layers, allowing you to add only the data that you application needs.



BUILDING IMAGES WITH DOCKERFILE

- **Dockerfile** - "the recipe" that docker uses to build images.
- Instructions are not case-sensitive but **uppercase** for convention.

`FROM <baseImage>:<version>` **->** Starts with a base image - the foundation of our application.

`RUN <command>` **->** Break up large commands with "\"

`ENV <name> <value>` **->** Set environment variables.

`WORKDIR <path>` **->** Set working directory where files be copied and commands will be executed.

`EXPOSE <portNumber>` **->** Expose ports.

`ARG <name> <defaultValue>` **->** Variable that users can pass at build time.

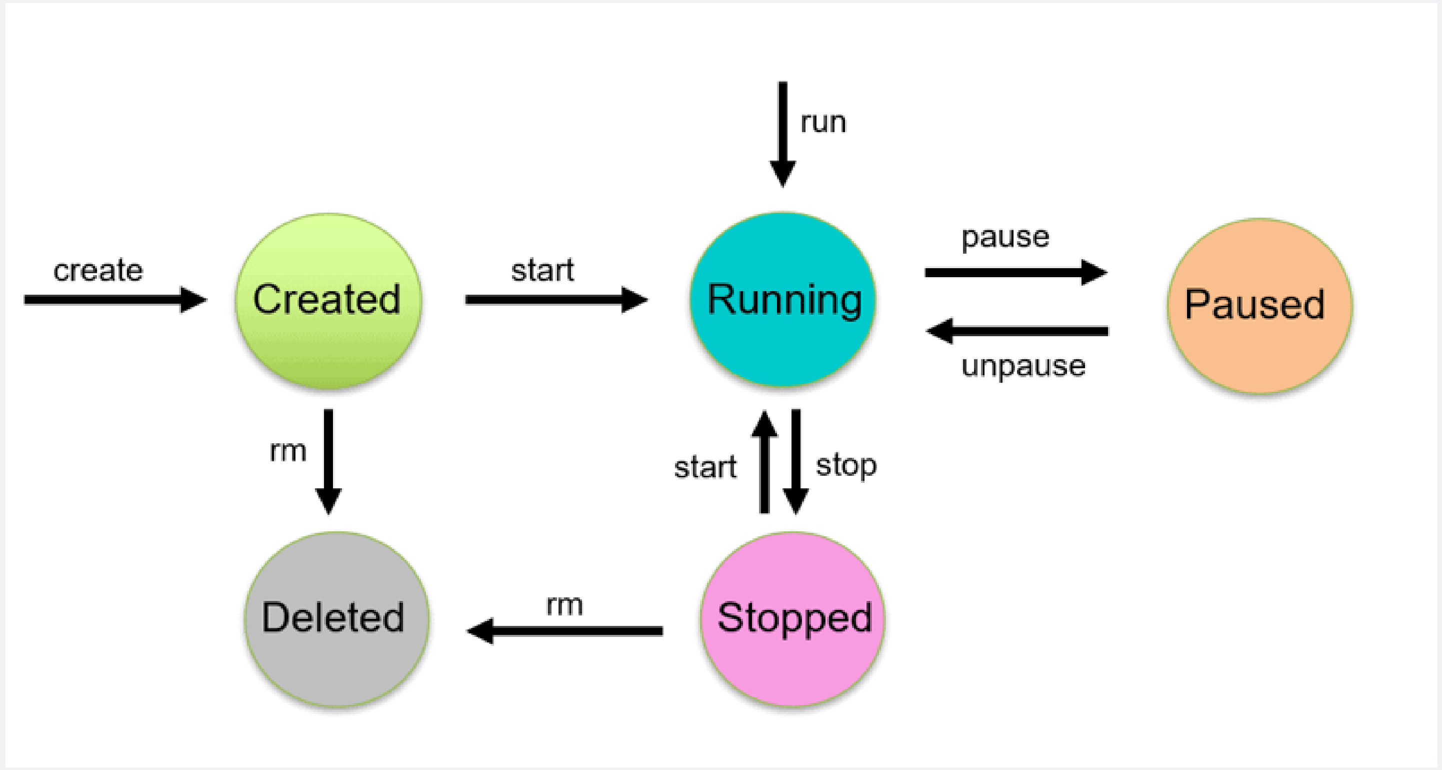
Specify what command should be executed when we start the container

1. `CMD <command or executable> <arg>` Easy to override. Used to provide default arguments for ENTRYPOINT
2. `ENTYRPOINT <command> <arg>` Hard to override. Consider this when a container should run the same executable every time. More preference.

DOCKER CONTAINERS

- A container is a runnable, portable and isolated instance of an image.
- They are defined by its image as well as any configuration option that we provide to it when we create or start it.
- Create, start, stop or delete a containers.
- When a container is removed, any changes to its state that are not stored in persistent storage disappear.

DOCKER CONTAINER LIFECYCLE



CONTAINER AND IMAGE MANAGEMENT

- Make use of the Docker CLI

```
docker build [OPTIONS] dockerfilePath
docker run [OPTIONS] image [COMMAND] [ARGS...]
docker create [OPTIONS] image [COMMAND] [ARGS...]
docker start [OPTIONS] container
docker rm [OPTIONS] container
docker stop [OPTIONS] container
docker pause [OPTIONS] container
docker unpaused [OPTIONS] container
```

DOCKER COMPOSE

- A tool for defining and running multi-container applications.
- Simplifies the control of your entire application stack.
- Make use of a YAML configuration file. Known as the Compose file.
- Docker compose for small scale.

COMPOSE APPLICATION MODEL(LEVELS)

- Service: An abstract concept of a container that you want to run as a part of your application. Each service corresponds to a containerized application and its configuration.
- Networks: Services make use of networks to communicate with each other. Capable of establishing an IP route between containers within services connected together.
- Volumes: Services store and share persistent data in volumes.
- Configs: Configuration of data that is dependent on the runtime or platform.
- Secret: Configuration of sensitive data.

SERVICES TOP-LEVEL ELEMENTS ATTRIBUTES

- **`build:`** specifies the build configuration for creating a container image from source. (path for the dockerfile).
- **`container_name`**
- **`depends_on:`** control the order of service startup and shutdown.
- **`env_file:`** attribute used to specify one or more files that contain environment variables for the container.
- **`environment:`** defines environment variables set in the container.
- **`image:`** specifies the image to start the container from.
- **`networks:`** defines the networks that service containers are attached to.
- **`ports:`** define port mapping between the host machine and the containers. Crucial for allowing external access. (similar to -p of docker run command)
- **`restart:`** defines the policy that the platform applies on container termination (no, always, on-failure, unless-stopped).

KEY COMMANDS

- `docker compose up:` to start all the services defined in your compose.yaml.
- `docker compose down:` to stop and remove the running services.
- `docker compose logs:` to monitor the output of your running containers.
- `docker compose ps:` list all the services.