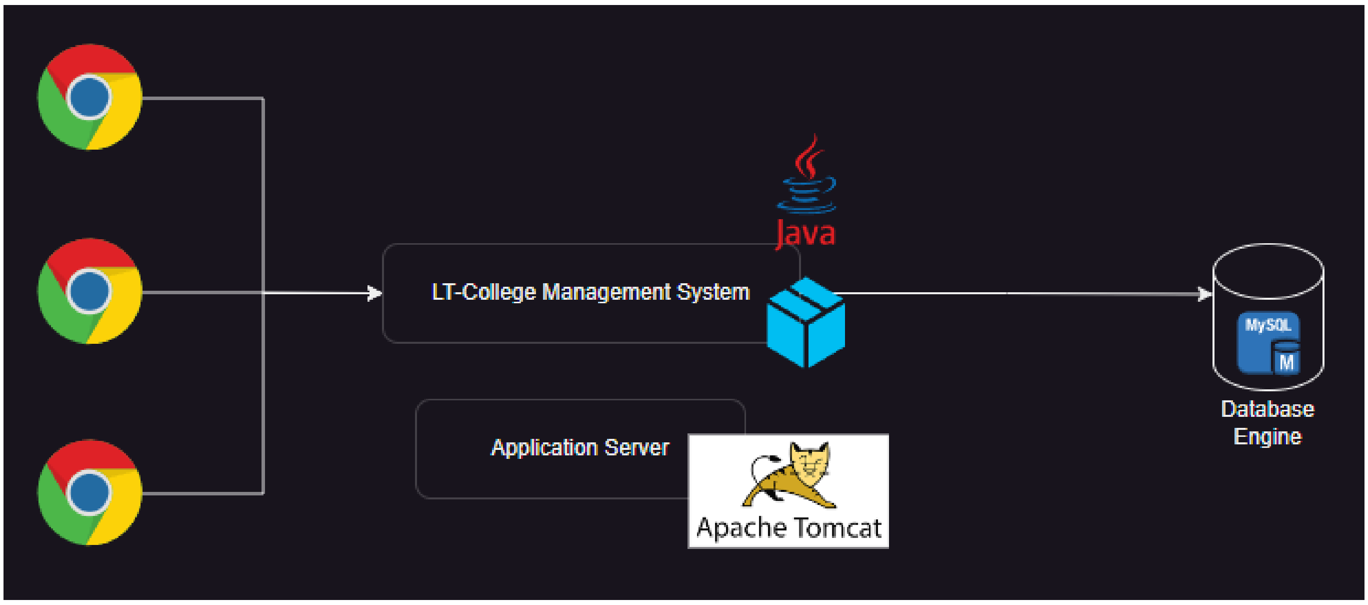
Docker

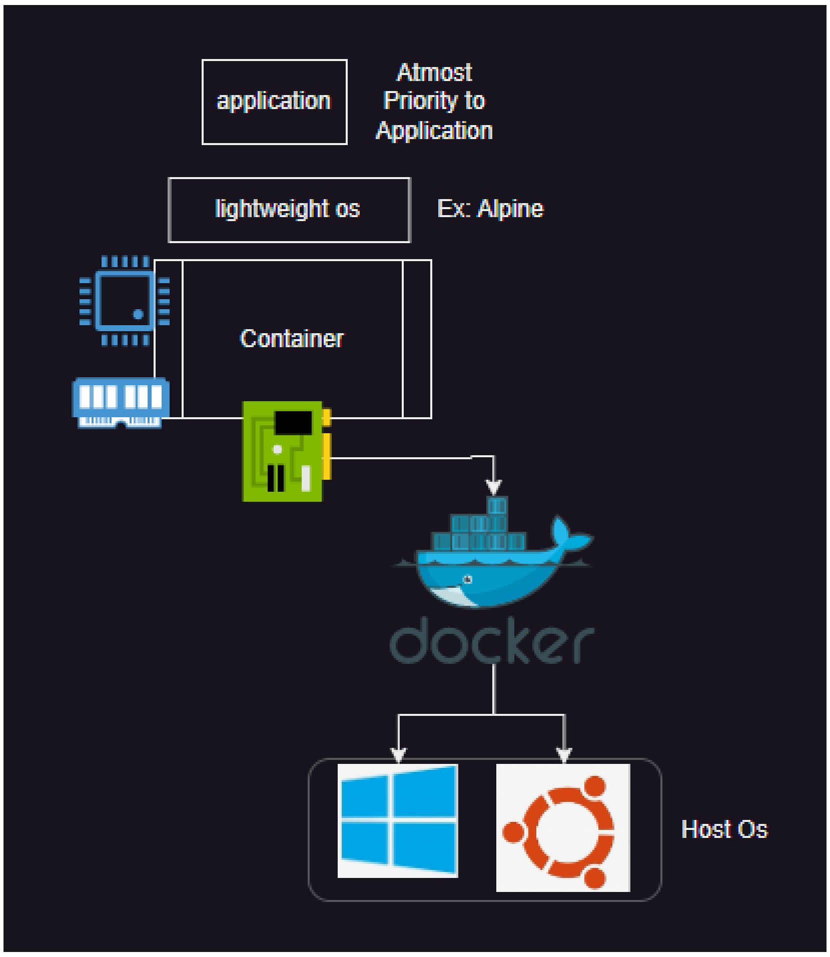
Post Docker Evalution



- This Architecture is Refered as **Monolith**

Overview of Server Generations:

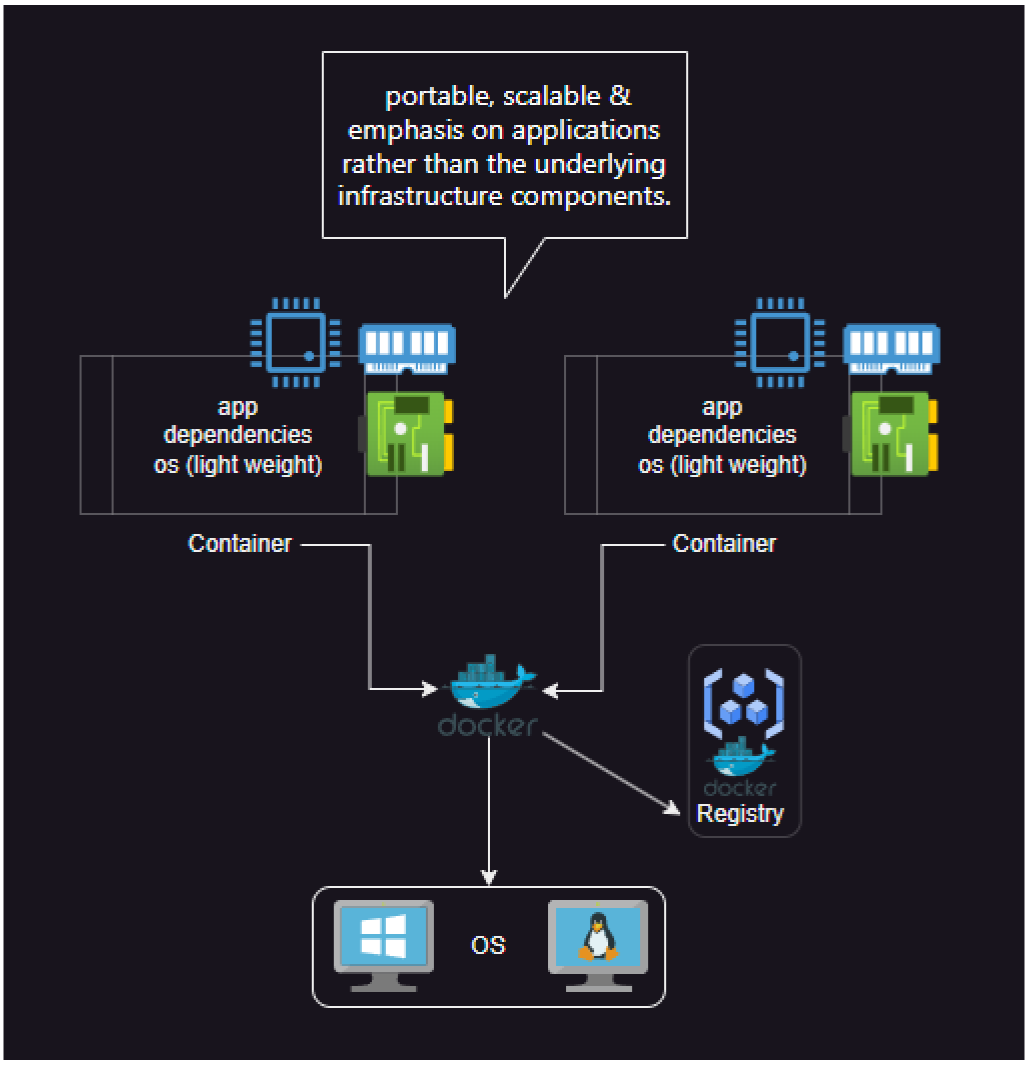
- Generation 1: **Physical server deployment**
- Generation 2: **Hypervisors**
  - The second generation introduced virtualization through hypervisors, allowing multiple virtual machines (VMs) to run on a single physical server.
  - Hypervisors can be classified into two types:
    - Type 1 (Bare Metal): Runs directly on the hardware (e.g., VMware ESXi, Microsoft Hyper-V).
    - Type 2 (Hosted): Runs on top of an existing operating system (e.g., VMware Workstation).
- Generation 3: **Container**
  - Containers allows applications to run in isolated environments called containers.
  - Unlike VMs, containers share the host OS kernel but remain isolated from each other, making them lightweight and fast to deploy.
  - the emphasis is increasingly placed on applications rather than the underlying infrastructure components.
  - Portability
  - EX: LXC (Linux Containers), Docker, Podman, rkt (Rocket)



Feature	Hypervisors (Virtual Machines)	Containers
Definition	Software that creates and manages virtual machines on a physical host.	Lightweight, portable packages that include applications and their dependencies.
Virtualization Level	Virtualizes the underlying hardware, allowing multiple OS instances.	Virtualizes the operating system, sharing the host OS kernel among containers.
Isolation	Strong isolation; each VM runs a separate OS, providing high security.	Less isolation; containers share the host OS, which can lead to security risks.
Resource Usage	Higher resource overhead; each VM requires its own OS and resources.	Minimal resource usage; containers are lightweight and share the host OS kernel.
Portability	Limited portability; VMs can be cumbersome to migrate between environments.	Highly portable; containers can run consistently across different platforms and environments.
Use Cases	Ideal for running multiple OS environments and applications requiring full isolation.	Best suited for microservices, rapid deployment, and environments needing scalability.
Examples	VMware, Hyper-V, KVM, Xen	Docker, Podman, Kubernetes, OpenShift

What is a container

- Container is a isolated area. This has everything required to run the application



- To run a docker container we need docker image
- Docker image contains all the necessary dependencies to run your application (Ghost Image)
- All the images which can be used are stored in Registry (repo)
- Docker will need a registry to get the images, Default Registry is docker hub which is public
- Organizations will have private Registries where they will be storing application images

Installing Docker

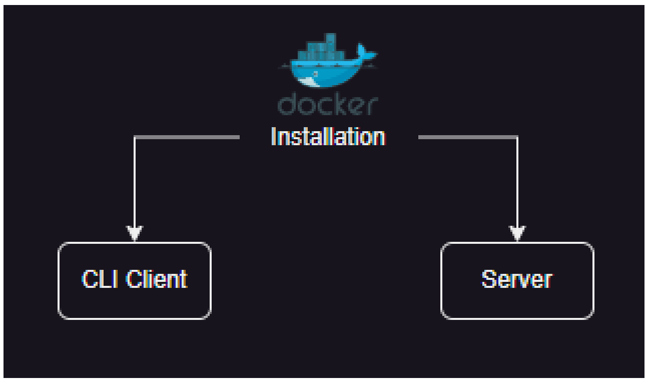
```
curl -fsSL https://get.docker.com -o install-docker.sh
sudo sh install-docker.sh
sudo usermod -aG docker $USER
```

- When we install docker the following components will be installed
  - docker ce (docker daemon)
  - docker ce cli (docker client)
  - containerd
  - docker compose
  - docker buildx

Checking installation

`docker info`

- you should get both cli client and server up and running



Docker Images

- Every Docker image will have Unique Id "SHA 256 based"
- Hash of Contents of Image
    - Contents
    - metadata
    - Build date & time

```
docker pull <imagename:<tag>
docker image inspect <image_id>
```

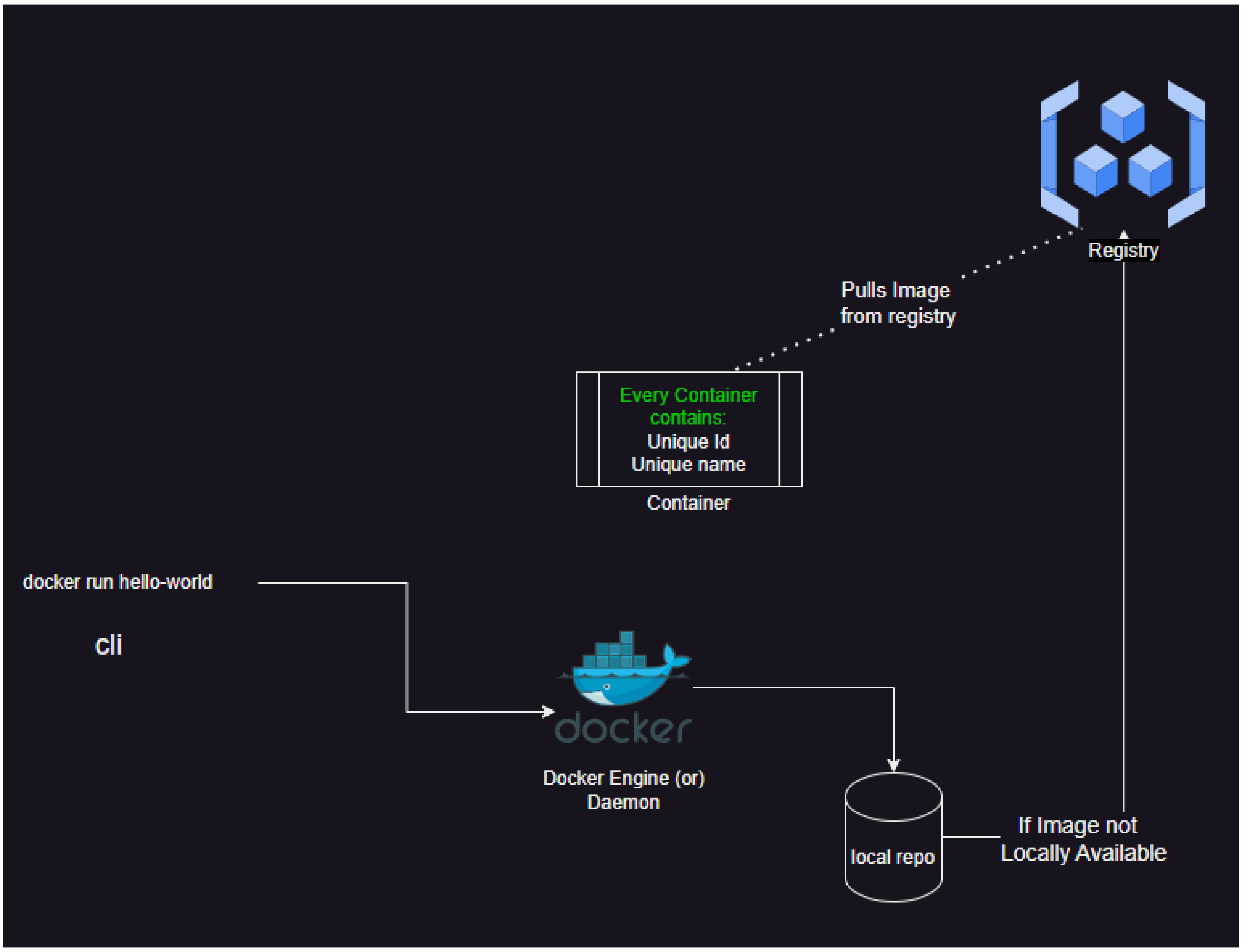
Tags

- Docker images have tags, **tags represent version**.
- If you don’t pass tag, docker will assume it to be latest
- Docker images from docker hub Tag Syntax

    ```
    official images <application>:<tag>
    community images <organization_name>/<application>:<tag>
    private images <username>/<application>:<tag>
    ```

```
docker image build -t my-nginx:1.0 .
# This command will read your Dockerfile, execute its instructions, and create an image tagged as my-nginx:1.0
```

Docker Containers


Registry
Pulls Image from registry
Every Container contains:
Unique Id
Unique name
Container
docker run hello-world
cli
docker
Docker Engine (or) Daemon
local repo
If Image not Locally Available

- If we don't give name docker will generate a name.
  - It is good idea to pass a name
  - `docker container run --name my-nginx nginx:latest`
- Unique Id will be generated by docker

Mode's of running Docker Containers

- Docker containers can be run in three primary modes:
  1. Detached Mode
  2. Attached Mode (Foreground Mode)
  3. Interactive Mode

Running container's

```
docker container run nginx:latest # attached mode to cli
docker container run --name my-nginx nginx:latest # passing name to container
docker container run -d --name my-nginx nginx:latest # detached mode from cli
docker container run -it --name my-nginx nginx:latest /bin/sh # Interactively
running Containers
docker exec -it my-nginx /bin/sh # Runs a Interactive command inside an already
running container
docker container exec my-nginx whoami # Runs a command inside an already running
container
```

List container's

```
docker container ls # (or)
docker ps
docker container ls -a # list all
```

Container Removal

```
docker container rm -f $(docker ps -a -q)
# -q : container id's of all running containers.
# -a: container id's of all containers.
docker image rm $(docker image ls -q)
# -q: There is no concept of running images, By default shows all images
```

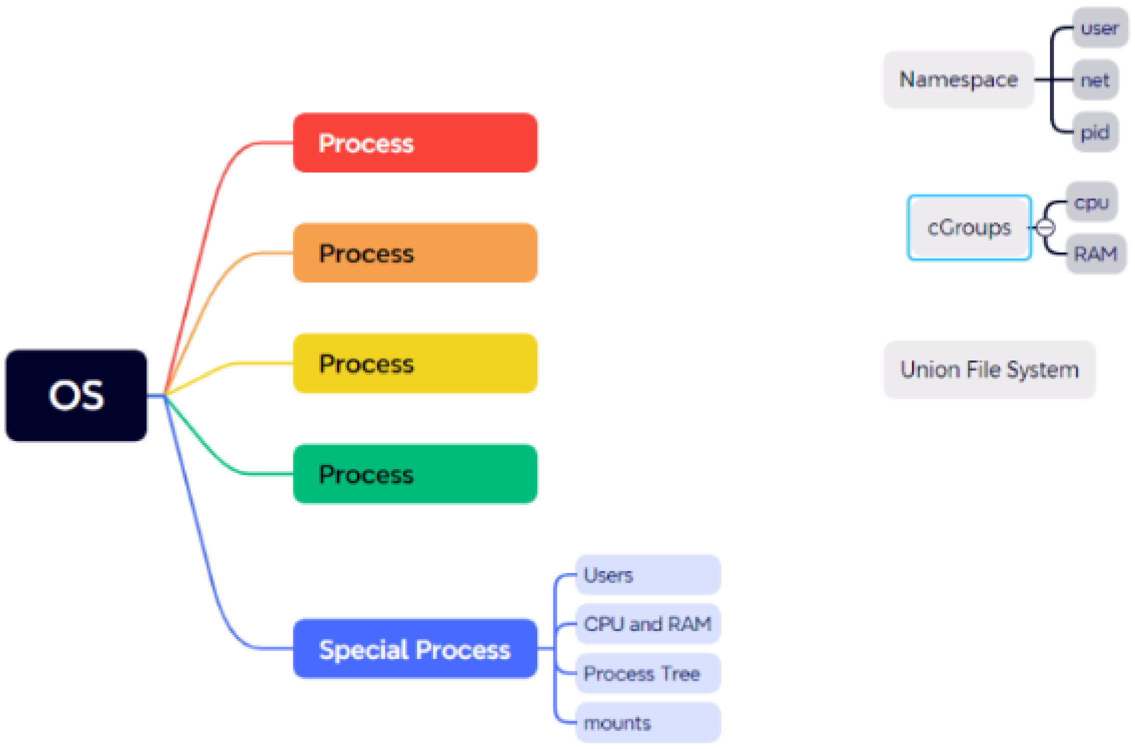
Port Mapping (or) Port forwarding

- Static Port Mapping
  - `-p <host-port>:<container-port>`
- Dynamic Port Mapping
  - `-P`

```
docker container run -d -P --name my-nginx nginx:latest
# Publishes all exposed ports of the Nginx container to random ports on the host
docker container run -d -p 36000:80 --name my-nginx nginx:latest
# Preferred when you need to expose a specific service on a known port
# -p <host-port>:<container-port> (or) -p <node-port>:<Application-port>
```

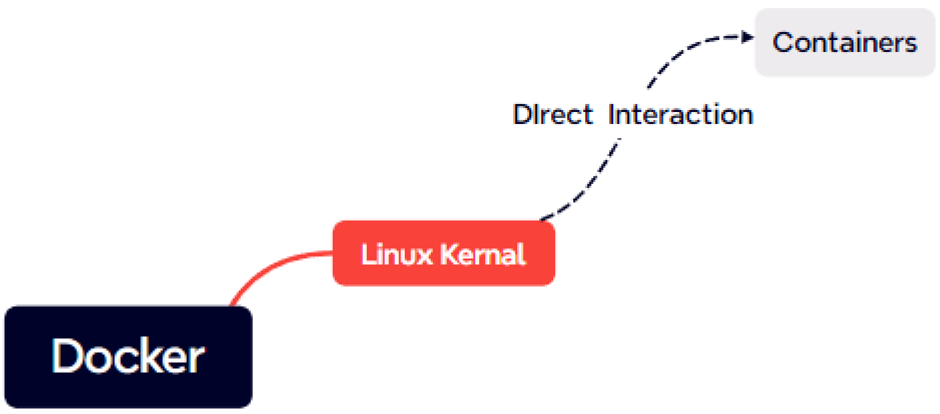
Key Points

- Inside containers, the PID 1 (first process that starts on startup) is the application
- Container will have its own process tree
- Contianer will have its own network, diskmounts, CPU and RAM allocated
- Restrictions can be placed on RAM and CPU on Containers
  `docker run --name nginx-1 --cpus 1 --memory 250m -d nginx:latest`
- Inside containers we have its own set of users



Container is an isolated space which will have its own
Refer Here

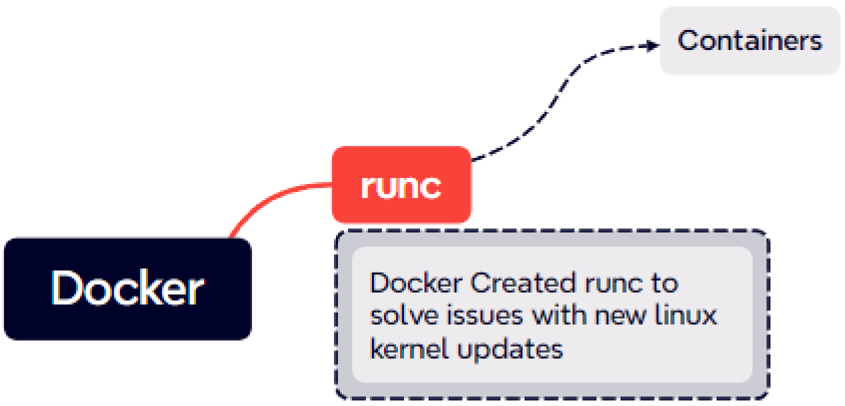
- process tree with the help of pid namespace
- disk mounts with the help of mnt namespace
- users with the help of usr namespace
- network with the help of net namespace
- CPU and RAM : with the help of cGroups
- Earlier Generation of Container creation incuded
  - docker engine directly interacting wiht linux kernel to create containers



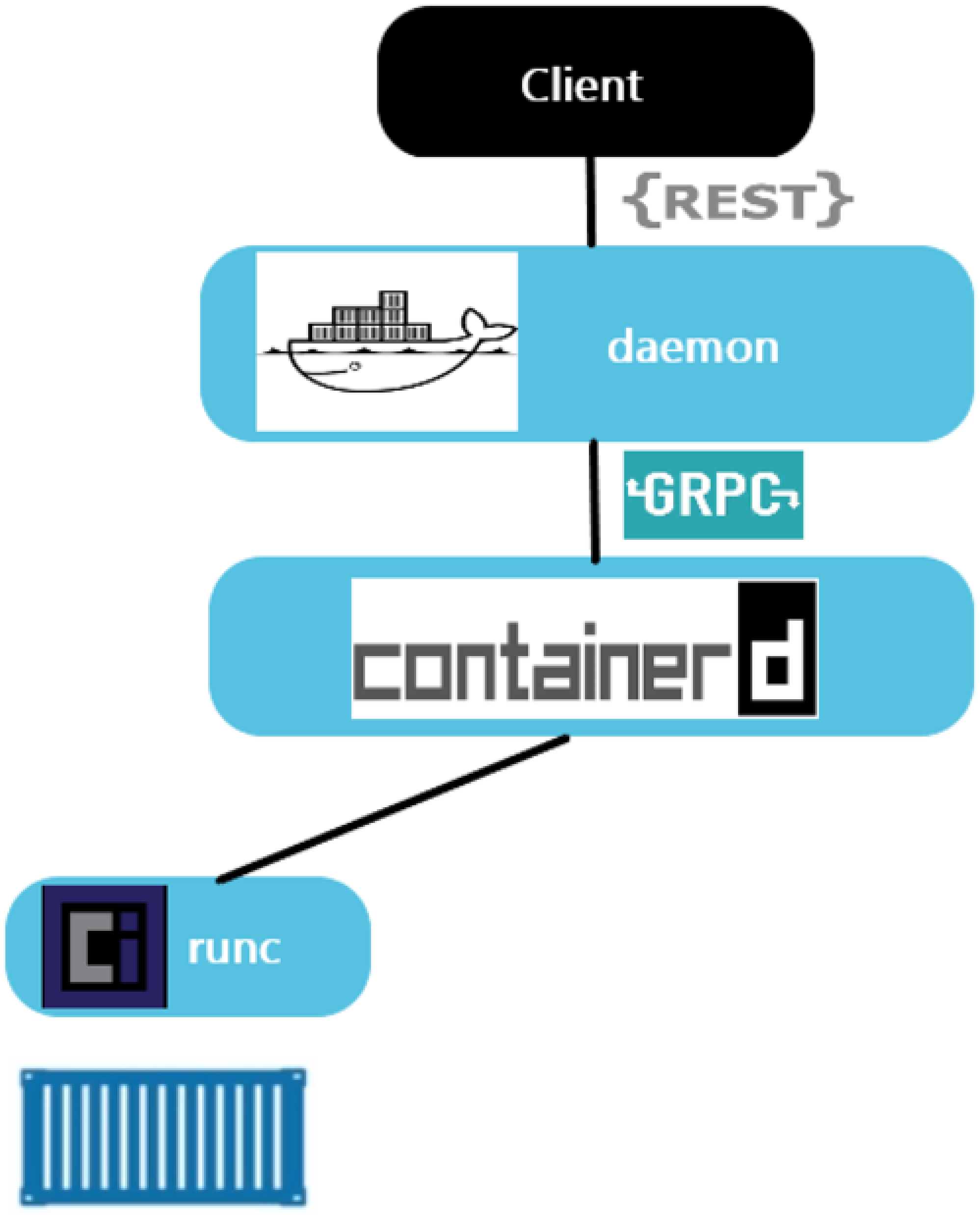
- This caused issues when ever linux got new Kernel update

solution

- Docker has implemented runc [`project Design document: lib container --> Implementation: runc`] to create containers


Containers
runc
Docker
Docker Created runc to solve issues with new linux kernel updates

- OCI (Open container initiative was formed) and the architecture of Docker has changed from monolith to layered
    - To establish common standards for container image formats(image Specification) and runtimes(container runtime Specification). This initiative aims to promote consistency and portability across different container technologies



Component	Functionality
Docker CE (dockerd)	- Manages containers (creation, execution, management).
	- Handles images (pulling, building, storing).
	- Provides networking for containers.
	- Exposes an API for interaction with clients.
Docker CLI	- Command-line tool for interacting with Docker.
	- Runs commands to manage containers and images (e.g., `docker run`, `docker build`).
containerd	- Lightweight runtime for managing container execution and storage.
	- Integrates with orchestration tools like Kubernetes.
Docker Compose	- Defines and runs multi-container applications using a YAML file.

Component	Functionality
	- Simplifies the management of complex applications.
Docker Buildx	- Enhances image building with advanced features (e.g., multi-platform support).
	- Uses BuildKit for efficient builds and caching.

Containerize Applications

- Making applications run in the container is referred as containerization.
  i.e. Building images with our app in it, For doing this we have two approaches
  - Manual Process
  - Dockerfile Approach

```
# Example
FROM openjdk:17-jdk-slim
LABEL app="spc"
COPY spring-petclinic-3.3.0-SNAPSHOT.jar /tmp/spring-petclinic-3.3.0-SNAPSHOT.jar
EXPOSE 8080
WORKDIR /tmp/
CMD ["java", "-jar", "spring-petclinic-3.3.0-SNAPSHOT.jar"]
```

```
docker build -t spc:1.0 .
docker run --name spc-1 -d -p 8600:8080 spc:1.0
```

Dockerfile

- docker image is made up of layer
- RUN, CMD, ADD or COPY --> each step adds a layer to base image
- A Layer is created during image creation by any instruction which leads to changes in the disk contents (RUN, ADD, COPY)

Refer here for Documentation to write Dockerfile

- Dockerfile Contains set of Instructions to Build the Image
  - `<instruction> <arguments>`
    - Ex: `<FROM> <alpine:1.27>`

FROM

- sets the base image, a valid Dockerfile must start with a FROM instruction

LABEL

- LABEL instruction adds metadata to an image.

RUN

- execute any commands to create a new layer on top of the current image, while building the image.
- shell form is recommended

EXPOSE

- publishes the port information

ADD, COPY

- Both COPY and ADD are instructions used in Dockerfiles to transfer files from the host machine into a Docker image

Key Differences

Feature	COPY	ADD
Source Types	Local files only	Local files, URLs, tar files
Extraction of Archives	No	Yes
Remote File Handling	No	Yes
Use Case	Preferred for simple copies	Use for URLs or extraction

CMD, ENTRYPOINT

- In Dockerfiles, both CMD and ENTRYPOINT are used to specify the command that runs when a container is started from an image.
- `This will be executed as pid1`
- Exec form (recommended):
  - CMD ["executable", "param1", "param2"]
  - ENTRYPOINT ["executable", "param1", "param2"]
- Shell form:
  - CMD executable param1 param2
  - ENTRYPOINT command param1 param2

Key Difference :

`CMD`:

- If you pass additional arguments when running the container, the CMD instruction is completely overridden by the new arguments.

`ENTRYPOINT`:

- If you pass additional arguments when running the container, those arguments are appended to the command defined in ENTRYPOINT.

- When `CMD ENTRYPOINT` both are used, Whatever is present in entrypoint remains intact and what ever has been written in CMD will be arguments to ENTRYPOINT

Examples

CMD Example

- CMD ["echo", "Hello, World!"]
  - docker run my-image ls
- Result:
  - The CMD `(echo "Hello, World!")` is replaced by ls.

ENTRYPOINT Example

- ENTRYPOINT ["echo", "Hello,"]
  - docker run my-image "Docker"
  - The ENTRYPOINT (echo "Hello,") remains intact, and "Docker" is appended as an argument. `Output: Hello, Docker`

CMD, ENTRYPOINT Example

- ENTRYPOINT ["echo"]
- CMD ["hello"]
  - docker run -d --name my-web alpine:1.27 ram
  - `Output: ram`

Note

- IF **We donot write entrypoint or CMD** the base images entrypoint or CMD will be considered
- Ideally anything can go in CMD or entrypoint, But in practice we execute the commands that start our application and wait till the application is executing.

ARG

- The ARG instruction defines variables that users can pass at build-time to the Dockerfile.

WORKDIR

- WORKDIR instruction **sets the working directory** for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.
- The WORKDIR instruction can be used multiple times in a Dockerfile.
  If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

USER

- USER instruction in a Dockerfile is used to specify the user that the container will run as.
- This is an important security feature, as it allows you to run applications within a container under a non-root user, **reducing potential vulnerabilities**.
- sets the user name (or UID) and optionally the user group (or GID)

```
FROM eclipse-temurin:17-jre
LABEL app='spc'
RUN useradd -m -s /bin/sh lt # user creation
# -m: This option creates a home directory for the new user.
# -s /bin/sh: This option sets the user's default shell to /bin/sh.
USER lt # user selection
WORKDIR /app # Set working directory (creates /app if it doesn't exist)
COPY spring-petclinic-3.3.0-SNAPSHOT.jar . # Copy application JAR to /app
EXPOSE 8080
ENTRYPOINT ["java", "-jar"]
CMD ["spring-petclinic-3.3.0-SNAPSHOT.jar"]
```

VOLUME

- The VOLUME instruction in a Dockerfile specifies a directory in the container that will be used as a mount point for a volume.
- This creates a persistent storage area outside of the container's writable layer, ensuring that data remains intact even if the container is stopped or deleted.
- Note: if the user gives the volume it will use that, otherwise it will create one

ARG and ENV

- ARG can be used as a parameter while building images
- ENV can be used as a parameter while running containers
  ENV instruction **sets the environment variable**

Refer below for further details |__>

Factors to Consider While Choosing Base image

- Size of Image
- vulnerability's

Typical startup commands

```
# java:
spring boot: java -jar <jarfile path>
app server: if your app runs on some external server then take the base image of
the app server and dont write CMD or ENTRYPOINT
# react js/angular js/vue js
first option: npm run start or equivalent
best `npm run build` option is to create a html site and run in nginx
# dotnet (asp.net core)
dotnet <dll path>
# Python
flask `python app.py` or server based startup
```

```
django `python app.py` or server based startup
fastapi uvicorn main:app --host 0.0.0.0 ...
```

Multi Stage Dockerfile

- Multi-stage Dockerfiles allow you to use multiple `FROM` statements in a single Dockerfile, enabling separate build and runtime environments.
  - **Reduced Image Size**: By excluding unnecessary files and dependencies from the final image, multi-stage builds create smaller images that are faster to transfer and deploy.
  - **Improved Security**: Smaller images have a reduced attack surface, minimizing potential vulnerabilities and security risks.
  - **Faster Build Times**: Changes in application code only require rebuilding affected layers, not the entire image, leading to quicker build times.
  - **Easier Maintenance**: A single Dockerfile with multiple stages simplifies management compared to having separate Dockerfiles for different environments (development, staging, production).
  - **Layer Caching**: Docker caches each layer, so if a stage hasn't changed, it can be reused in subsequent builds, speeding up the process.
  - **User Management**: You can create non-root users in the runtime stage to enhance security by running applications with limited permissions.
  - **Copying Artifacts**: Use COPY --from=<stage_name> to selectively copy only the necessary artifacts (like binaries or JAR files) from previous stages.

Examples

Python - Multi Stage, Dockerfile

```
# stage 1
FROM python:3.9-alpine AS builder
# metadata
LABEL app='sample-api'
WORKDIR /app
COPY requirements.txt .
# Install system dependencies required for building Python packages
RUN apk add --no-cache libffi libgcc libstdc++ musl openssl
# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt
COPY . .

FROM python:3.9-alpine AS runner
LABEL app='sample-api'
WORKDIR /app
# Copy installed packages from the builder stage
COPY --from=builder /usr/local/lib/python3.9/site-packages/
/usr/local/lib/python3.9/site-packages/
COPY --from=builder /usr/local/bin/ /usr/local/bin/
# Copy application files from the builder stage and change ownership
COPY --chown=nobody:nobody . .
# Switch to the non-root user
```

```
USER nobody
#if user not avilable, create: `RUN adduser -m -s <user> /bin/sh`
# Non-Root user: `RUN adduser -D <user>`
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Java - Multistage Dockerfile

```
FROM maven:3.9.9-eclipse-temurin-17 AS builder
LABEL stage='build' app='spc'
WORKDIR /app
COPY . .
RUN mvn clean package

FROM eclipse-temurin:17-jre AS runner
LABEL stage='run' app='spc'
WORKDIR /app
# RUN adduser -D spc
RUN adduser spc
# COPY --chown=spc:spc --from=builder /app/target/*.jar spring-petclinic.jar
COPY --from=builder /app/target/*.jar spring-petclinic.jar
RUN chown spc spring-petclinic.jar
USER spc
EXPOSE 8080
CMD ["java", "-jar", "spring-petclinic.jar"]
```

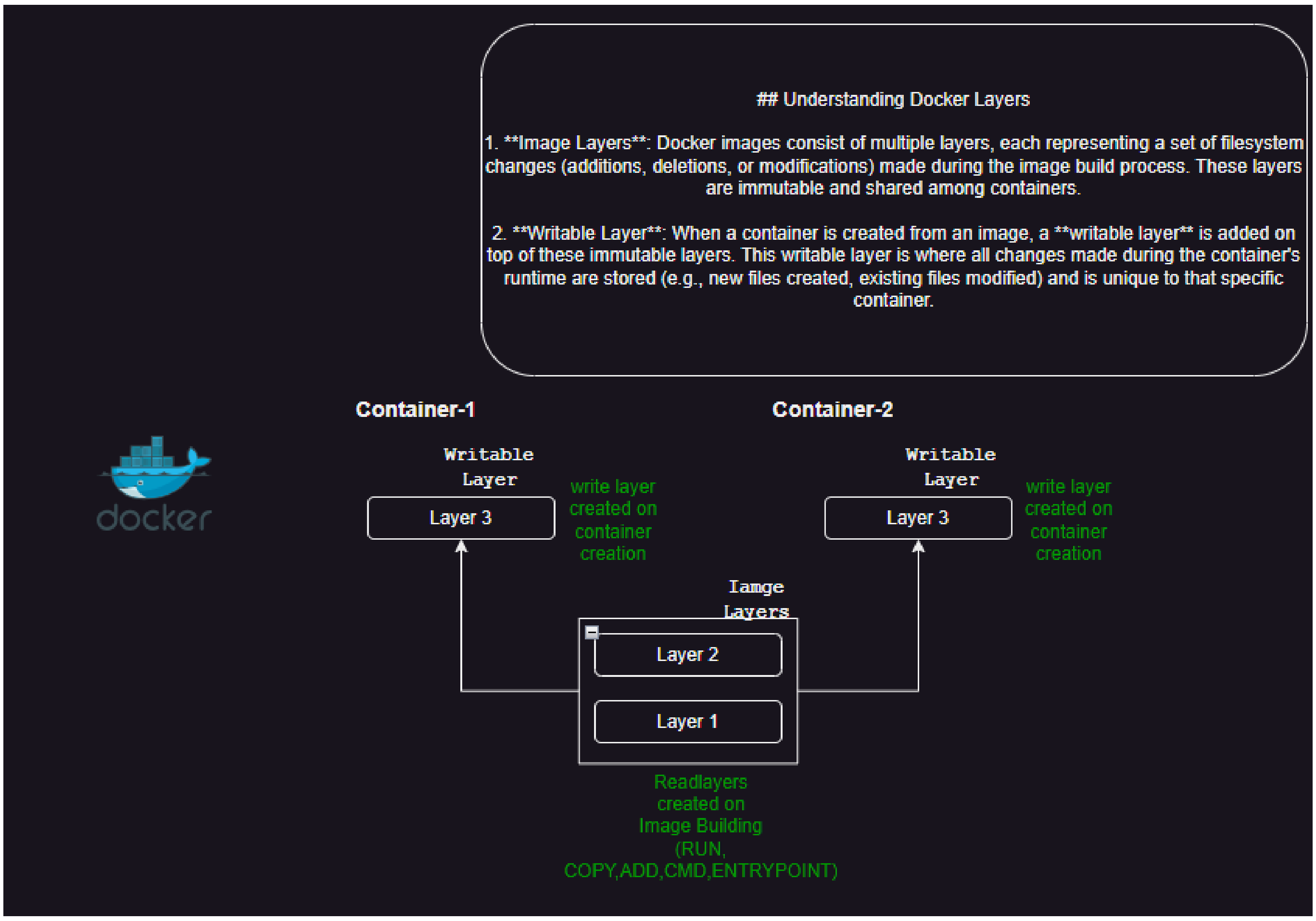
Reactjs - Multistage Dockerfile

```
# Stage 1: Build the application
FROM node:16-alpine AS builder
LABEL stage="build"
WORKDIR /app
# Copy package.json and package-lock.json first (to cache dependencies layer)
COPY package*.json ./
# Install dependencies
RUN npm install
# Copy the entire application source code
COPY . .
# Build the application (creates static HTML/CSS/JS in the "build" or "dist"
folder)
RUN npm run build
# npm run start
# Stage 2: Serve the static files with Nginx
FROM nginx:stable-alpine AS runner
LABEL stage="serve"
# Copy the build output to the Nginx HTML directory
COPY --from=builder /app/build /usr/share/nginx/html
# Expose the port Nginx will listen on
```

```
EXPOSE 80
# Start Nginx
CMD ["nginx", "-g", "daemon off;"]
```

Docker Layers:

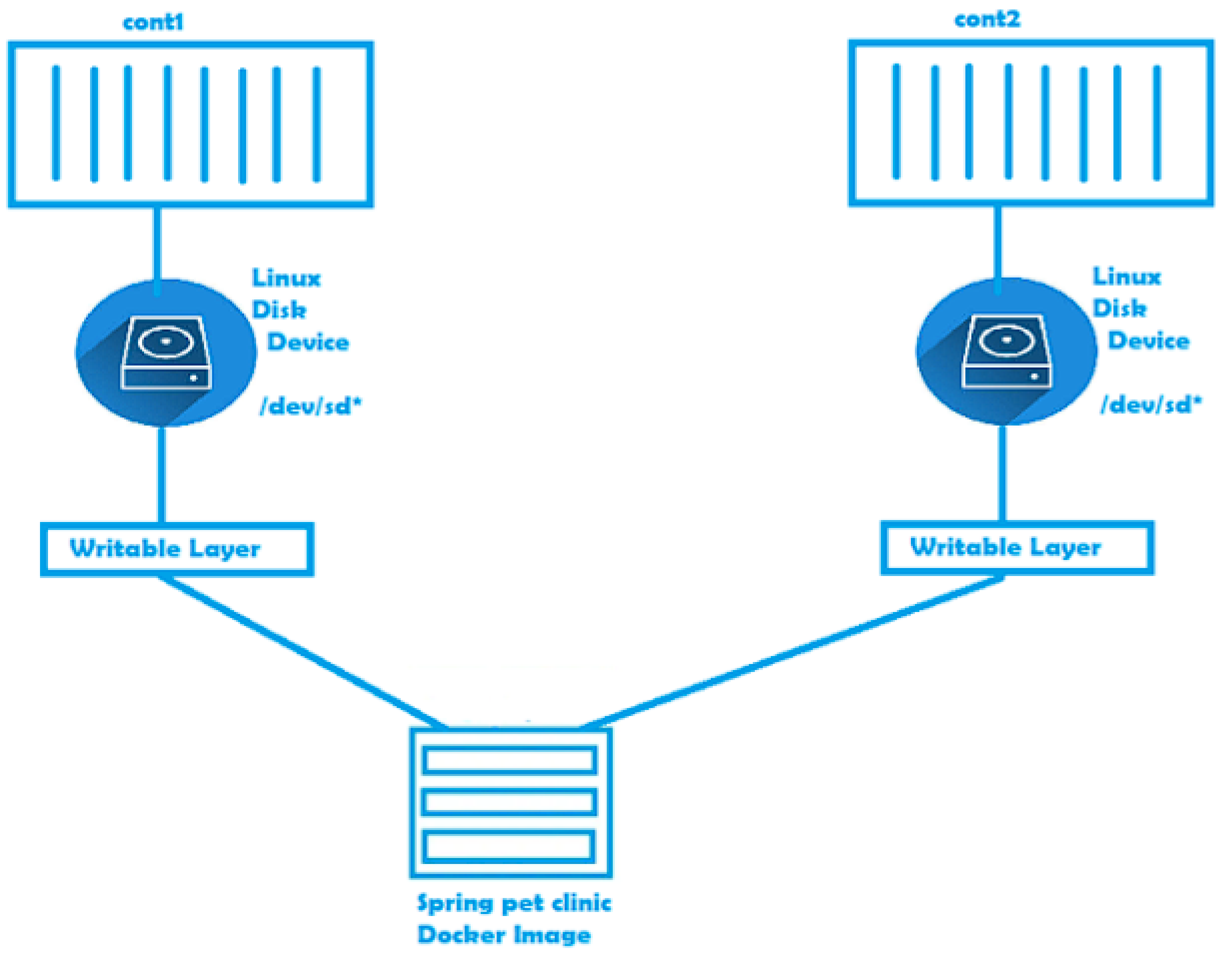
- Every instruction (RUN, ADD, COPY etc) leads to the creation of new image layers
  - All of these layers are combined in the image & as well as container.
  - These layers are readonly layers
- By default, all the files created/modified by the containers reside on the writable layer of the container.
- The problem with the writable layer of the container is "Writable layer will be deleted once the container is deleted"


Understanding Docker Layers

1. **Image Layers**: Docker images consist of multiple layers, each representing a set of filesystem changes (additions, deletions, or modifications) made during the image build process. These layers are immutable and shared among containers.

2. **Writable Layer**: When a container is created from an image, a **writable layer** is added on top of these immutable layers. This writable layer is where all changes made during the container's runtime are stored (e.g., new files created, existing files modified) and is unique to that specific container.

Container-1
Container-2
Writable Layer
Layer 3
write layer created on container creation
Writable Layer
Layer 3
write layer created on container creation
Iamge Layers
Layer 2
Layer 1
Readlayers created on Image Building (RUN, COPY,ADD,CMD,ENTRYPOINT)



- When any container tries to change the contents of the existing files in image layers (As we are aware image layers are readonly), then the file which is supposed to be changed is copied into the write layer of the container and the modifications will be present only for the container changing it.
- One of the major advantage with this approach is storage optimization in terms of space occupied on disk in the case of multiple container running.
- There is also one challenge. The write layer is avaiable only as long as container is alive. The moment container is removed from the machine, the writable layer is also deleted.
- To overcome this challenge lets understand
    - Storage Drivers
    - Persist Data in Containers
        - Storage Drivers
            - Storage Drivers allow you to create the data in the writable layer of the container
            - Docker Default storage driver implementation is overlay2
                - overlay2 :
                    - Preferred storage driver for all linux distributions
                    - Requires no extra configuration
- **To persist the data** into the Docker Host (Machine on which Docker is installed), Docker provides the following options:
    - Volumes :
        - Stored in the hostfile system managed by Docker (/var/lib/docker/volumes/ on Linux).

        - Non Docker Processes should not modify the file system*
    - Bind Mounts :
        - Can be stored any where on the host system.
        - Non-Docker Processes, data path is User managed
        - `docker run -d -P -v ~/test /app --name web-app alpine:1.27`
            - `-v ~/test:/app`: This option mounts a volume from your host machine into the container. In this case, it maps the ~/test directory on your host to the /app directory inside the container. This allows you to share files between your host and the container.
    - tmpfs Mounts :
        - Store on the host systems memory, never written to file system

Docker Volumes

- Docker Volumes are preferred mechanisms for persisting data generated by and used by Docker Containers.
- we can use same volume across multiple containers (or) one volume for one container that is user choice

```
docker volume create my-vol
# the volume mountpoint can be found throught `docker inspect my-vol` command
docker run -d -P -v my-vol:/data --name db-app mysql:latest
# `-v my-vol:/data`: This mounts a Docker volume named my-vol to the /data
directory inside the container. This volume is used for persistent storage,
meaning that any data written to /data will be saved even if the container is
stopped or removed.
# `docker volume ls` to list available volumes
```

VOLUME

- The VOLUME instruction in a Dockerfile specifies a directory in the container that will be used as a mount point for a volume.
- This creates a persistent storage area outside of the container's writable layer, ensuring that data remains intact even if the container is stopped or deleted.
- Note: if the user gives the volume it will use that, otherwise it will create one

ARG and ENV

- ARG can be used as a parameter while building images
- ENV can be used as a parameter while running containers
  ENV instruction **sets the environment variable**

passing env variables in docker build command:

```
docker run --name lib-db -d \
-e 'POSTGRES_PASSWORD=password' \
-e 'POSTGRES_USER=lt' \
-e 'POSTGRES_DB=library' \
-v lib-vol:/var/lib/postgresql/data \
--network my-net \
postgres:17
```

passing env variables in Dockerfile

```
# Use the official PostgreSQL image as the base
FROM postgres:17

# Set environment variables for PostgreSQL
ENV POSTGRES_USER=lt
ENV POSTGRES_PASSWORD=password
ENV POSTGRES_DB=library

# Optionally, copy any initialization scripts if needed
# COPY init-scripts/ /docker-entrypoint-initdb.d/

# Expose the default PostgreSQL port
EXPOSE 5432

# Define the volume for persistent data storage
VOLUME ["/var/lib/postgresql/data"]

# The default command to run PostgreSQL server
CMD ["postgres"]
```

passing ARG

```
# Use a base image
FROM ubuntu:20.04

# Define build arguments
ARG APP_VERSION=1.0
ARG APP_NAME=myapp

# Set environment variables using the build arguments
ENV VERSION=$APP_VERSION
ENV NAME=$APP_NAME

# Install dependencies (example)
RUN apt-get update && \
    apt-get install -y curl

# Print out the values of the environment variables
RUN echo "Building $NAME version $VERSION"
```

```
# Set the command to run when the container starts
CMD ["echo", "Container is running"]
```

```
docker build --build-arg APP_VERSION=2.0 --build-arg APP_NAME=my_custom_app -t
my_custom_image .
```

Connectivity Between Application and Database

Connection strings

- Every Database generally has a
  - hostname (name/ip address)
  - username
  - password
  - database name (optional)
- Generally all languages use a connection string to establish connection
- When we run our application in containers in most of the cases the db details are passed as environmental variables and in somecases it will be file

DEFNITION

- A database URI (Uniform Resource Identifier) is a string that **specifies the connection details required to access a database**. It typically includes the database engine, user credentials, host address, port number, database name, and any additional parameters needed for the connection.
  - `postgres://user:password@localhost:5432/library`
  - `mysql://user:password@localhost:3306/library`

```
FROM python:3.9-slim
# Set environment variables for the database connection
ENV DATABASE_URI=postgres://lt:password@lib-db:5432/library
# Set the working directory in the container
WORKDIR /app
# Copy the requirements file and install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# Copy the rest of the application code into the container
COPY app.py .
# Expose the port that Flask runs on
EXPOSE 5000
# Command to run the application
CMD ["python", "app.py"]
```

connection strings

```
mysql -h <hostname> -P <port> -u <username> -p
psql -h <hostname> -p <port> -U <username> -d <database>
```

.dockerignore

- You can use a .dockerignore file to exclude files or directories from the build context. Refer here for .dockerignore docs

---

vulnerability Scanning

```
# Install Trivy
cd /usr/local/bin
curl -sfL
https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sh
-s -- -b /usr/local/bin
# trivy --version
# Basic Scan Command
trivy image <image-name>
```

Networking

- Docker supports creating networks of different types
  - bridge network (single host)
  - host network (single host)
  - macvoverlan (single host)
  - overlay (multi host)
- Default bridge network cannot resolve by names but uses only ip, custom bridge networks can resolve with names as well as ips
  - `docker network create my-net`

switch between networks

```
# Creating Custom network
docker network create my-net
# List All Networks
docker network ls
# inspect network
docker network inspect <network_name>
# Connect the Container to the New Network
docker network disconnect <current_network> <container_name>
# Disconnect the Container from the Current Networ
docker network connect my-new-network <container_name>
```

Docker Compose

Docker Compose is a tool that lets you define and manage multi-container applications using a single YAML configuration file, making it easy to start, stop, and configure multiple Docker containers at once.

Aspect	Docker Compose	Kubernetes (K8s)
Purpose	Local development and testing for multi-container apps.	Production-grade container orchestration.
Scalability	Limited to single-host setups.	Scales across multiple nodes and clusters.
Fault Tolerance	Basic restart policies; no self-healing.	Self-healing with failover and replication.
Networking	Basic single-host networking.	Advanced multi-node networking and load balancing.
Resource Management	Minimal resource control.	Fine-grained resource allocation (CPU, memory).
Production Use	Not suited for production due to lack of scalability and reliability.	Designed for scalable and reliable deployments.
Complexity	Simple setup and usage.	More complex, requires expertise to manage.

Look into **Techworld_nana Tutorial** for writing docker compose file, It is Important for Developers to Known how to write Docker Compose file for Local development and testing multi-container apps.

Editional topics

docker cp command

The `docker cp` command is a powerful utility for copying files and directories between your local filesystem and Docker containers. Here is a detailed guide with examples, key features, and tips for effective usage.

Syntax

The basic syntax of the `docker cp` command is:

```
# Copy from container to host
docker cp <containerId/containerName>:<path_in_container> <path_on_host>

# Copy from host to container
docker cp <path_on_host> <containerId/containerName>:<path_in_container>
```

Key Features

1. **Bidirectional Copying**: Files and directories can be copied both from a container to the host and vice versa.
2. **Recursive Copy**: When copying directories, `docker cp` behaves like the Unix `cp -a` command, copying recursively and preserving file permissions where possible.
3. **Container Reference**: You can specify the container using either its name or ID.
4. **Flexibility with Paths**: Both absolute and relative paths can be used for source and destination.