Docker

Container platforms and container runtimes serve different purposes in the container ecosystem, and while they are related, they have distinct roles. Here's a breakdown of the key differences between container platforms and container runtimes:

1. \*\*Container Runtimes:\*\*

- \*\*Definition:\*\* A container runtime is a software component responsible for running and managing containers.

- \*\*Functionality:\*\* It deals with the low-level operations of containers, such as container creation, starting, stopping, and resource isolation.

- \*\*Examples:\*\* Docker (with containerd or other runtimes), containerd, rkt, and others.

- \*\*Interacts with:\*\* The container runtime interacts directly with the operating system's kernel to leverage features like namespaces and control groups for process and resource isolation.

2. \*\*Container Platforms:\*\*

- \*\*Definition:\*\* A container platform is a broader solution that encompasses various tools and services to simplify the entire container lifecycle, from development to deployment and management.

- \*\*Components:\*\* In addition to a container runtime, a container platform often includes orchestration tools, image registries, networking solutions, and monitoring tools.

- \*\*Functionality:\*\* It provides a comprehensive environment for developing, deploying, and managing containerized applications at scale.

- \*\*Examples:\*\* Kubernetes, Docker Enterprise, OpenShift, and others.

- \*\*Interacts with:\*\* A container platform interacts with the container runtime but also includes higher-level services for tasks like orchestration, scaling, and load balancing.

In summary, a container runtime is a fundamental component responsible for the execution of containers, while a container platform is a more extensive solution that provides tools and services for managing the complete containerized application lifecycle. The container platform often includes a container runtime as one of its components but goes beyond it to offer additional features for orchestration, scalability, and other aspects of container management.

In Docker, namespaces are a key feature of containerization that provide isolation and control over system resources. Namespaces allow processes within a container to have their own view of system resources, making it seem like they have their own isolated instance of the system.

Here are some of the namespaces used in Docker:

1. **PID Namespace**: Provides isolation for process IDs. Each container has its own set of processes, and a process in one container can't see or interfere with processes in another container.
2. **Network Namespace**: Isolates network resources such as network interfaces, routing tables, and firewall rules. Each container can have its own network stack, making it possible for containers to have separate IP addresses, ports, etc.
3. **Mount Namespace**: Controls the filesystem mount points seen by a process. This allows containers to have their own filesystem view, with its own root filesystem, isolating them from changes in the host system.
4. **UTS Namespace**: Isolates hostname and domain name values. Containers can have their own hostname and domain name separate from the host and other containers.
5. **IPC Namespace**: Provides isolation for inter-process communication resources such as System V IPC and POSIX message queues. Processes in different IPC namespaces can't communicate with each other using these mechanisms.

Control Groups (cgroups) are a Linux kernel feature used to manage and limit the system resources utilized by processes or groups of processes. They provide resource isolation, prioritization, and accounting for system resources such as CPU, memory, disk I/O, and more.

Here's a breakdown of what cgroups do:

1. \*\*Resource Allocation\*\*: Cgroups allow administrators to allocate resources like CPU time, system memory, network bandwidth, or combinations thereof to specific processes or groups of processes.

2. \*\*Resource Limitation\*\*: They enable setting limits on how much of a particular resource a process or a group of processes can consume. For instance, you can restrict the amount of CPU usage or memory that a container can utilize.

3. \*\*Prioritization\*\*: Cgroups allow setting priorities for different processes or groups, ensuring that critical processes get more resources compared to non-critical ones.

4. \*\*Accounting\*\*: They enable tracking and reporting resource usage by processes, helping administrators understand resource utilization patterns and manage resources effectively.

In the context of containers (like Docker), cgroups are used alongside namespaces to provide resource isolation and control. For example, when you specify CPU or memory limits for a Docker container, Docker uses cgroups to enforce these limits.

Cgroups can be managed via tools like `systemd`, `cgconfig`, or directly through the `/sys/fs/cgroup` file system, allowing administrators to allocate and manage resources efficiently across the system. They play a crucial role in ensuring fair resource distribution and preventing resource contention among different processes or containers running on the same host.

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

cd1a24ad8207 my-node:01 "npm run start" 25 minutes ago Up 25 minutes 0.0.0.0:3030->3030/tcp fervent\_booth

4fa1ee1d1d47 frontend:01 "/docker-entrypoint.…" 47 minutes ago Up 47 minutes 80/tcp, 0.0.0.0:8300->8300/tcp youthful\_galileo

c809db7b756d postgres:latest "docker-entrypoint.s…" 20 hours ago Up 20 hours 0.0.0.0:5432->5432/tcp postgres-container

cb05f85b27a7 dpage/pgadmin4 "/entrypoint.sh" 20 hours ago Up 20 hours 0.0.0.0:80->80/tcp, 443/tcp angry\_kirch

Commands:

# docker run --interactive --tty --rm ubuntu:22.04

--interactive –tty = -it

In Docker, the **--interactive** (**-i**) and **--tty** (**-t**) flags are used when running containers to interact with the container's command-line interface, allowing a more interactive experience similar to working directly on the terminal.

* **--interactive (-i):** This flag keeps STDIN open even if not attached, allowing you to interact with the container's command-line interface. It's typically used when you want to manually input commands or provide input to the container's running processes. Without this flag, Docker might not keep STDIN open by default, making it challenging to provide input or execute commands.
* **--tty (-t):** This flag allocates a pseudo-TTY (Teletype) or terminal for the container. It simulates a terminal-like environment, enabling features like colored output and command-line editing within the container. This flag is often used in conjunction with the **-i** flag to create an interactive terminal session.

When using both **-i** and **-t** together (**-it**), you create an interactive session with a terminal. For instance, running a container with **docker run -it <image>** allows you to interact with the command prompt inside the container, just like you would in a typical terminal environment.

In Docker, the --rm flag is used with the docker run command to automatically remove the container once it exits. It's particularly useful when you want to run a container for a short-lived process and don't want to keep the container around after it finishes running.

The --name attribute in Docker commands is used to assign a specific name to a container when it's created. By default, Docker assigns random names to containers using combinations of two words, but using --name allows you to give a more meaningful or descriptive name to a container.

docker run -it --name my-ubuntu-container ubuntu:22.04

*# List all containers*

docker container ps -a | grep my-ubuntu-container

docker container inspect my-ubuntu-container

*# Restart the container and attach to running shell*

docker start my-ubuntu-container

docker attach my-ubuntu-container

The **--tag** or **-t** flag in Docker commands is used to assign a specific tag to an image during its build or when pulling an image from a repository. Tags are like version labels for Docker images. They help identify different versions or variations of the same image.

When building an image using the docker build command,

docker build --tag my-ubuntu-image -<<EOF

FROM ubuntu:22.04

RUN apt update && apt install iputils-ping --yes

EOF

# Run a container based on that image

docker run -it --rm my-ubuntu-image

……….

Network

# docker network ls

# docker run -it –rm –network net-2 my-ubuntu-image

………..

volumes

<https://medium.com/@williehung/persisting-data-using-docker-volume-and-bind-mount-52a8cb42f4f0>

# create a named volume

docker volume create my-volume

# Create a container and mount the volume into the container filesystem

docker run -it --rm --mount source=my-volume,destination=/my-data/ ubuntu:22.04

# There is a similar (but shorter) syntax using -v which accomplishes the same

docker run -it --rm -v my-volume:/my-data ubuntu:22.04

# Now we can create and store the file into the location we mounted the volume

echo "Hello from the container!" > /my-data/hello.txt

cat my-data/hello.txt

exit

# Create a container that mounts a directory from the host filesystem into the container

docker run -it --rm --mount type=bind,source="${PWD}"/my-data,destination=/my-data ubuntu:22.04

# Again, there is a similar (but shorter) syntax using -v which accomplishes the same

docker run -it --rm -v ${PWD}/my-data:/my-data ubuntu:22.04

# list image

Docker ls image

# attach to running container

Docker attach <container name>

# detach

Docker -d <container name>

…………..

# delete container

Docker rm -f <container-name>

# attach to running container

Docker attach <container name>

# detach

Docker -d <container name>

* Use docker attach when you want direct access to the primary process of a running container to observe or interact with it in real-time.
* Use docker exec -it with bash when you need to start an additional process within the container for running commands or performing operations without directly interacting with the main process.

# docker exec -it myfrontend /bin/bash

…………………..

Logs:

# docker logs -f <container-name>

….

Stop/start container

# docker container start/stop <name>

…..

Remove container

# docker container ls

# docker container rm <mame>

…………

Run postgres DB :

docker run -d \

--name mypostgres \

-e POSTGRES\_PASSWORD=mysecretpassword \

-p 5432:5432 \

postgres:latest

………….

Run pgadmin

docker run -p 80:80 \

--name mypgadmin \

-e PGADMIN\_DEFAULT\_EMAIL=user@example.com \

-e PGADMIN\_DEFAULT\_PASSWORD=mysecretpassword \

-d dpage/pgadmin4

…………….

Docker login

docker tag postgres:latest ahmedeldesoki/udapeople:postgres

docker push ahmedeldesoki/udapeople:postgres

…………..

kubectl run postgres --image=ahmedeldesoki/udapeople:postgres --env="POSTGRES\_PASSWORD=mysecretpassword" --env="POSTGRES\_USER=postgres" --env="POSTGRES\_DB=glee"

kubectl port-forward postgres 5432:5432

………………..

Docker-compode tutorial:

[Docker Compose Tutorial (youtube.com)](https://www.youtube.com/watch?v=HG6yIjZapSA)

Docker by mumshat

[Docker Tutorial for Beginners - A Full DevOps Course on How to Run Applications in Containers (youtube.com)](https://www.youtube.com/watch?v=fqMOX6JJhGo)

…………..

To work with docker in different host:

# docker -H=remote-docker-engine:23755

# docker -H=remote-docker-engine:23755 run nginx

To limit the resource for container

# docker run –cpus=.5 ubuntu …> ensure the container will not take more than 50% of host cpu.

# docker run -memory=100m ubuntu

………………….