What is Docker?

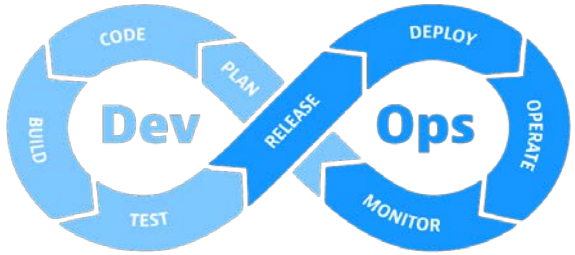
And how does it even work?

A brief overview of Docker and its underlying core Linux concepts that make such a powerful system possible.

Presentation by Debojyoti Roy (21ME10028)
For KOSS Selection Task

What is Docker?

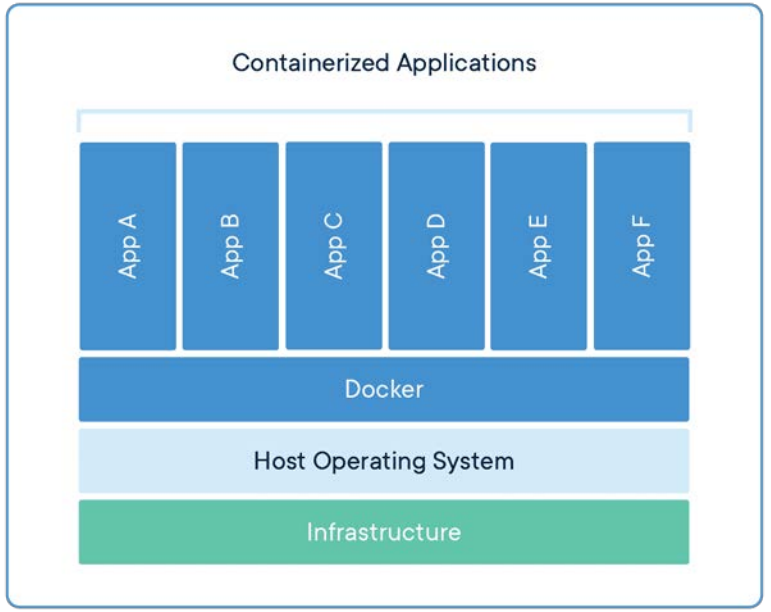
- Docker is a popular open-source project written in Go and developed by Dotcloud.
- It is basically a container engine using the Linux Kernel features like “namespaces” and “control groups” to create “containers” on top of an operating system.
- Containers enable the developers to combine the application source code along with the operating system libraries and dependencies in order to create a single standardized executable component.
- It simplifies the delivery of distributed applications and has become increasingly popular as organizations start to adopt the Dev-Ops Model of software development.



A schematic diagram describing the Dev-Ops Model.

What are containers?

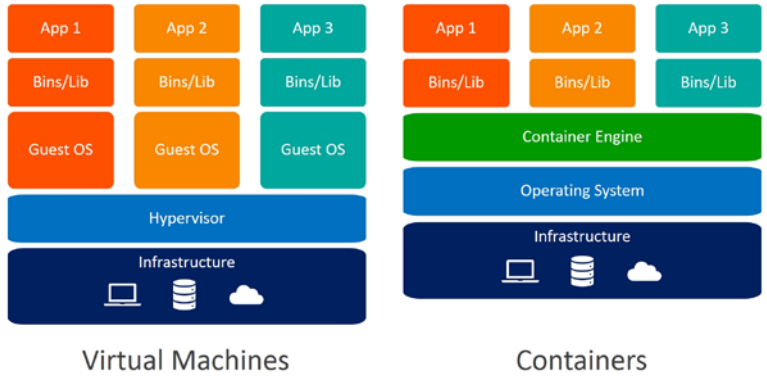
- If you have ever seen a container in a ship, freight train or truck, you have seen the product of the largest trade revolution in the 20th century. Packing items of various sizes and shapes into a single standardized container introduced uniformity in shipping drastically reducing the loading and unloading times in the ports and stations.
- The containers we will be talking about now work in a very similar fashion. They are a reliable answer to the problem of how to get the software to run correctly when moved from one computational environment to another.
- In layman's terms, a container consists of an entire runtime environment: an application, plus all it's dependencies, libraries and other binaries, and configuration files needed to run it, bundled into one package.
- By containerizing the application platform and its dependencies, differences in OS distributions can be easily abstracted away.



Containerized Applications running on top of Docker

Difference between Docker Containers and Virtualization

- The key differentiator between containers and virtual machines is that virtual machines virtualize an entire machine down to the hardware layers and containers only virtualize software layers above the operating system level.
- A Virtual Machine runs on top of an emulating software called the hypervisor which sits between the hardware and the virtual machine. Each virtual machine runs its own guest operating system. They are less agile and have low portability than containers.
- A container is an isolated, lightweight silo for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system) and contain only apps and some lightweight operating system APIs and services that run in user mode.



Why are containers so popular?

- Container technology offers all the functionality and benefits of VMs - including application isolation, cost-effective scalability, and disposability - plus important additional advantages:

1. Lighter weight: Unlike VMs, containers don't carry the payload of an entire OS instance and hypervisor; they include only the OS processes and dependencies necessary to execute the code. Container sizes are measured in megabytes (vs. gigabytes for some VMs), make better use of hardware capacity, and have faster startup times.
2. Greater resource efficiency: With containers, you can run several times as many copies of an application on the same hardware as you can using VMs. This can reduce your resource spending.
3. Improved developer productivity: Compared to VMs, containers are faster and easier to deploy, provision and restart. This makes them a better fit for development teams adopting DevOps practices.

Why use Docker?

- Docker enhanced the native Linux containerization capabilities with technologies that enable:

1. **Improved—and seamless—portability:** Docker containers run without modification across any desktop, data centre or cloud environment.
2. **Automated container creation:** Docker can automatically build a container based on application source code.
3. **Container versioning:** Docker can track versions of a container image, roll back to previous versions, and trace who built a version and how. It can even upload only the deltas between an existing version and a new one.
4. **Container reuse:** Existing containers can be used as *base images*—essentially like templates for building new containers.
5. **Shared container libraries:** Developers can access an open-source registry containing thousands of user-contributed containers.

Google Trends Data on Docker

Behind the scenes of containers: Namespaces and Cgroups

- Containers wisely use two features of the Linux Kernel to achieve near-perfect lightweight isolation and segregation of the resources of the system to run its processes. This is important for both, resource utilization and security and so several containers can share CPU and memory while staying within the predefined constraints.

1. Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.
2. A control group (cgroup) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, and so on) of a collection of processes.

More on namespaces

- In other words, the key feature of namespaces is that they isolate processes from each other. On a server where you are running many different services, isolating each service and its associated processes from other services means that there is a smaller blast radius for changes, as well as a smaller footprint for security-related concerns.
- There are the following types of namespaces with their own unique properties:

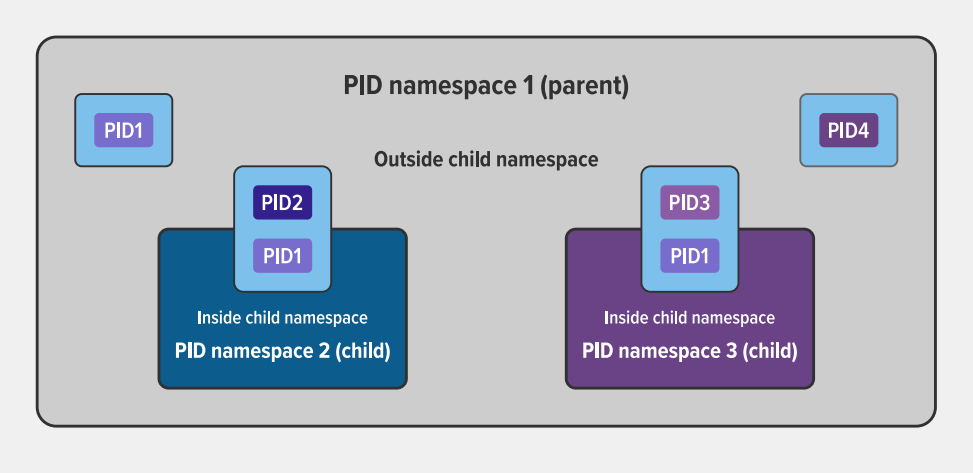
1. A **user namespace** has its own set of user IDs and group IDs for assignment to processes. In particular, this means that a process can have root privilege within its user namespace without having it in other user namespaces.
2. A **process ID** (PID) namespace assigns a set of PIDs to processes that are independent of the set of PIDs in other namespaces. The first process created in a new namespace has PID 1 and child processes are assigned subsequent PIDs. If a child process is created with its own PID namespace, it has PID 1 in that namespace as well as its PID in the parent process' namespace.

More on namespaces (contd.)

3. A **network namespace** has an independent network stack: its own private routing table, set of IP addresses, socket listing, connection tracking table, firewall, and other network-related resources.
4. A **mount namespace** has an independent list of mount points seen by the processes in the namespace. This means that you can mount and unmount filesystems in a mount namespace without affecting the host filesystem.
5. An **interprocess communication** (IPC) namespace has its own IPC resources, for example, POSIX message queues.
6. A **UNIX Time-Sharing** (UTS) namespace allows a single system to appear to have different host and domain names for different processes.

Example of parent and child PID namespaces

- In the diagram, there are three PID namespaces – a parent namespace and two child namespaces. Within the parent namespace, there are four processes, named **PID1** through **PID4**. These are normal processes which can all see each other and share resources.
- The child processes with **PID2** and **PID3** in the parent namespace also belong to their own PID namespaces in which their PID is 1. From within a child namespace, the **PID1** process cannot see anything outside. For example, **PID1** in both child namespaces cannot see **PID4** in the parent namespace.
- This provides isolation between (in this case) processes within different namespaces.



Namespaces demo

- We list all the available namespaces in our system using the lsns command.
- Then, we unshare the Z-Shell process from its parent namespace.
- Then, if we look at the PID of the Z-Shell from its own interactive shell within its PID namespace, we find that the PID of the forked process is 1.
- However, if we look at the PID of Z-Shell from a second terminal on the system, we find that it has a different PID.
- The “host system” sees the big picture and sees the process as PID 7723 but on the other hand, the process sees itself as PID 1. That is the beauty of namespaces.

```
$ lsns
        NS TYPE   NPROCS   PID USER    COMMAND
4026531835 cgroup     85  1571 seth /usr/lib/systemd/systemd --user
4026531836 pid        85  1571 seth /usr/lib/systemd/systemd --user
4026531837 user       80  1571 seth /usr/lib/systemd/systemd --user
4026532601 user        1  6266 seth /usr/lib64/firefox/firefox [...]
4026532928 net         1  7164 seth /usr/lib64/firefox/firefox [...]
[...]
```

```
$ sudo unshare --fork --pid --mount-proc zsh
%
```

```
% pidof zsh
pid 1
```

```
$ pidof zsh
7723
```

More on cgroups

- A control group (cgroup) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, and so on) of a collection of processes.
- Cgroups provide the following features:

1. **Resource limits** – You can configure a cgroup to limit how much of a particular resource (memory or CPU, for example) a process can use.
2. **Prioritization** – You can control how much of a resource (CPU, disk, or network) a process can use compared to processes in another cgroup when there is resource contention.
3. **Accounting** – Resource limits are monitored and reported at the cgroup level.
4. **Control** – You can change the status (frozen, stopped, or restarted) of all processes in a cgroup with a single command.

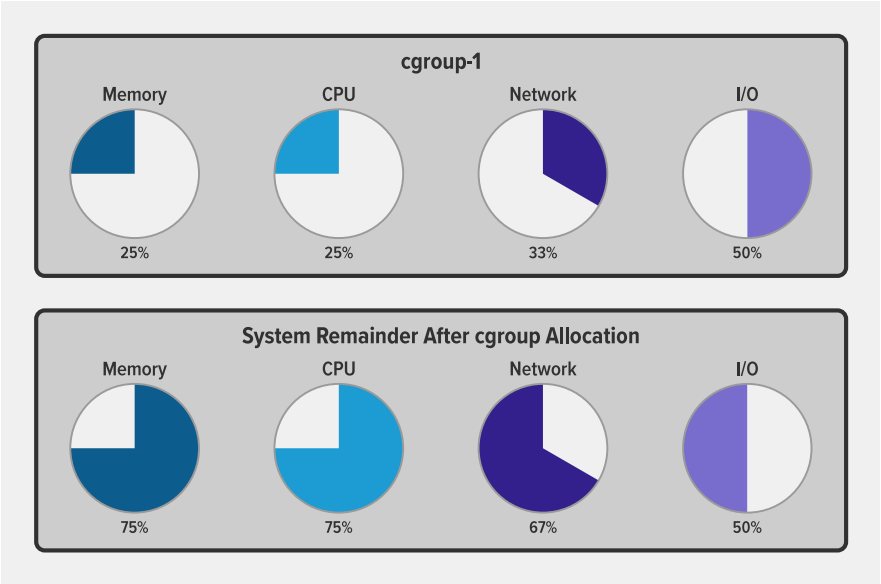


Illustration of how cgroups work

cgroups demo

- In the first piece of code we are seeing the set of cgroup subsystems that are enabled on the system.
- We see the devices cgroup to view the hierarchy for that particular subsystem of the cgroup. The tasks file shows all the processes that are associated with this cgroup.
- Then we create a new cgroup called ‘lfnw’ and then move the shell into the particular cgroup.

```
$ ls /sys/fs/cgroup
blkio     cpu,cpuacct  freezer  net_cls          perf_event
cpu       cpuset       hugetlb  net_cls,net_prio  pids
cpuacct   devices      memory   net_prio          systemd
$ ls /sys/fs/cgroup/devices
cgroup.clone_children  devices.allow  docker            system.slice
cgroup.procs           devices.deny   notify_on_release  tasks
cgroup.sane_behavior   devices.list   release_agent      user.slice
$
```

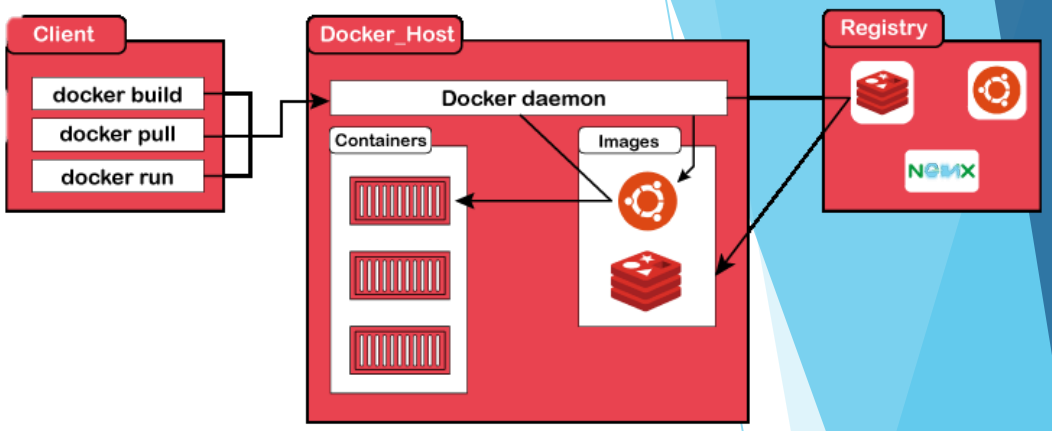
```
$ sudo mkdir /sys/fs/cgroup/pids/lfnw
$ ls /sys/fs/cgroup/pids/lfnw
cgroup.clone_children  notify_on_release  pids.events  tasks
cgroup.procs           pids.current       pids.max
$ cat /sys/fs/cgroup/pids/lfnw/tasks
$ echo 5222 | sudo tee /sys/fs/cgroup/pids/lfnw/tasks
5222
$ cat /proc/5222/cgroup
11:freezer:/
10:memory:/user.slice
9:devices:/user.slice
8:perf_event:/
7:cpuset:/
6:pids:/lfnw
5:hugetlb:/
4:blkio:/user.slice
3:cpu,cpuacct:/user.slice
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-1000.slice/session-45.scope
```

cgroups demo (contd.)

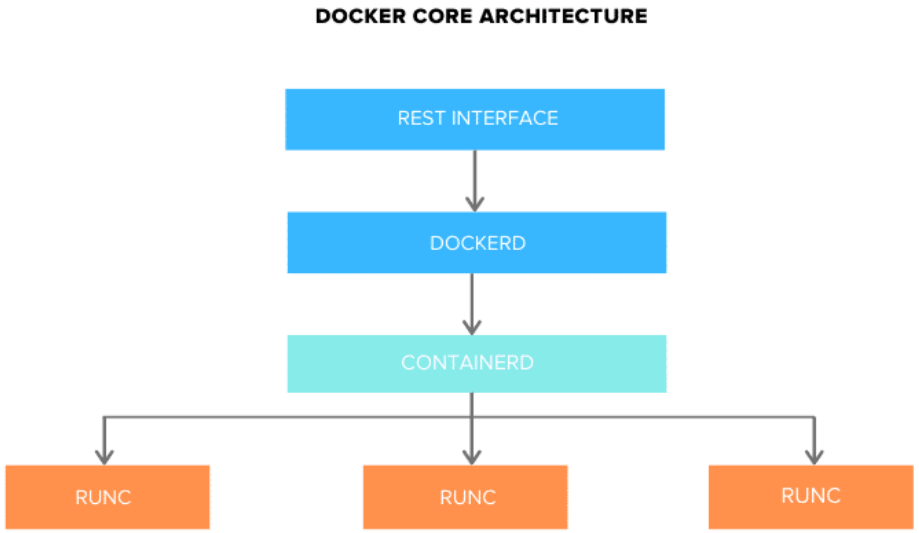
- We view the CPU cgroup from the perspective of a docker container.
- The **CPUShares** value provides tasks in a cgroup with a relative amount of CPU time.

```
$ docker run --name demo --cpu-shares 256 -d --rm amazonlinux sleep 600
397b366a7ae2276b25367935f3f24cd8b28dd179238c44bfa7cf8ecf48fcbbce
$ docker exec demo ls /sys/fs/cgroup/cpu
cgroup.clone_children
cgroup.procs
cpu.cfs_period_us
cpu.cfs_quota_us
cpu.rt_period_us
cpu.rt_runtime_us
cpu.shares
cpu.stat
cpuacct.stat
cpuacct.usage
cpuacct.usage_all
cpuacct.usage_percpu
cpuacct.usage_percpu_sys
cpuacct.usage_percpu_user
cpuacct.usage_sys
cpuacct.usage_user
notify_on_release
tasks
$ docker exec demo cat /sys/fs/cgroup/cpu/cpu.shares
256
$
```

The Docker Architecture

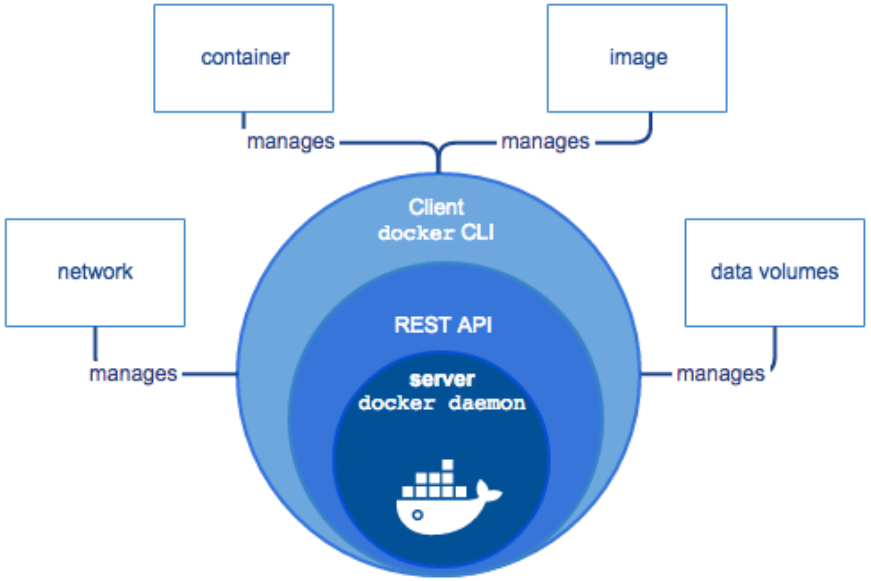


- Docker works on a client-server architecture. It includes the docker client, docker host, and docker registry. The docker client is used for triggering docker commands, the docker host is used to running the docker daemon, and the docker registry to store docker images.
- The docker client communicates to the docker daemon using a REST API, which internally supports to build, run, and distribution of docker containers. Both the client and daemon can run on the same system or can be connected remotely.



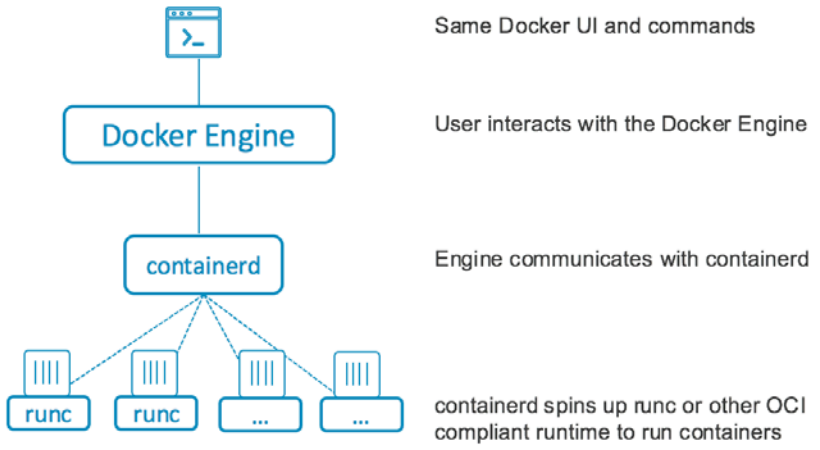
The Docker Engine

- Docker engine comprises the docker daemon, an API interface, and Docker CLI. Docker daemon (dockerd) runs continuously as dockerd systemd service. It is responsible for building the docker images.
- To manage images and run containers, dockerd calls the docker-containerd APIs.



docker-containerd (containerd)

- containerd is another system daemon service than is responsible for downloading the docker images and running them as a container. It exposes its API to receive instructions from the dockerd service
- It manages the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond.



docker-runc

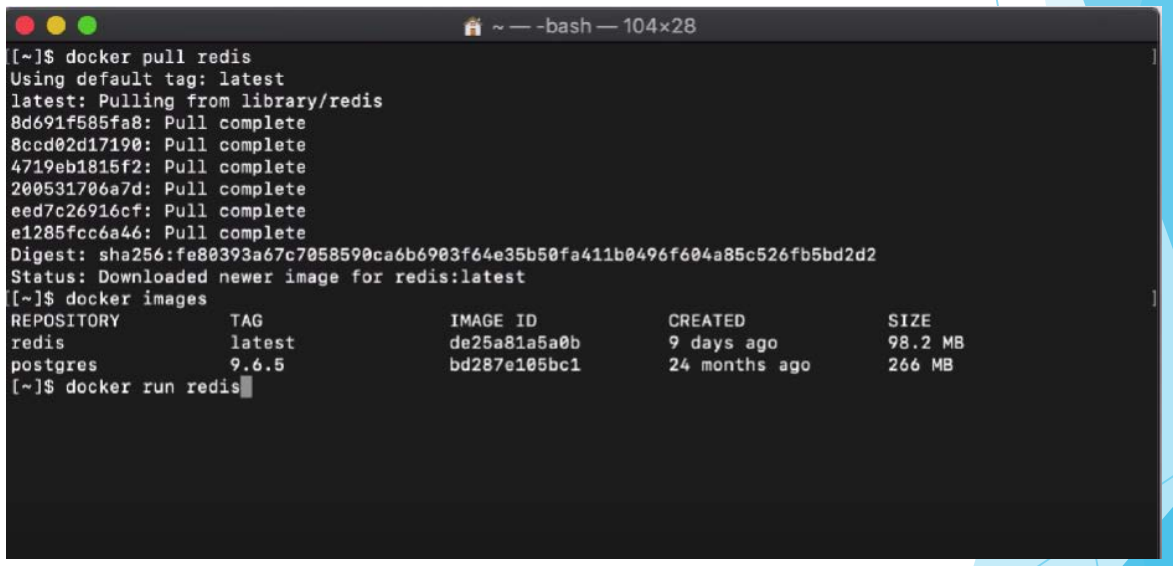
- runc is the container runtime responsible for creating the namespaces and cgroups required for a container. It then runs the container commands inside those namespaces. runc runtime is implemented as per the OCI specification.
- runC is a lightweight, portable container runtime. It includes all of the plumbing code used by Docker to interact with system features related to containers. It is designed with the following principles in mind:
- • Designed for security.
  • Usable at large scale, in production, today.
  • No dependency on the rest of the Docker platform: just the container runtime and nothing else.

Docker commands

Action Performed	Docker command
Pull image from DockerHub (e.g. Apache HTTP Server)	docker pull httpd
List all the images	docker images
Run an image as a container (e.g. Apache HTTP Server)	docker run -it -d httpd
List all the containers running	docker ps
Rin commands inside the container. (e.g. container id 09ca6feb6efc)	docker exec -it 09ca6feb6efc bash
Remove a container. (e.g. container id 9b6343d3b5a0)	docker rm 9b6343d3b5a0
Remove an image. (e.g. image id fce289e99eb9)	docker rmi fce289e99eb9
Restart a container. (e.g. container id 09ca6feb6efc)	docker restart 09ca6feb6efc
Stop a container. (e.g. container id 09ca6feb6efc)	docker stop 09ca6feb6efc
Start a container. (e.g. container id 09ca6feb6efc)	docker start 09ca6feb6efc

Docker demo

- Here, we pull the Redis image from DockerHub and then try to run it in our system as a container.



Review and Conclusion

- Hence, we learned what is Docker, what are containers, and how they use cgroups and namespaces in various ways to isolate their applications from the rest of the systems.
- We looked at the various forms of advantages Docker and its containers offer and how it differs from a Virtual Machine.
- We also got familiar with the docker architecture and its underlying components of runc and containerd and saw how they work as a team to deliver our needs.
- We got a brief understanding of how namespaces and cgroups can be used from the Linux shell and tweak with them.
- We also derived a basic understanding of Docker commands and how to use them.
- I hope that this has been very informative for you. Thank you very much. Happy learning!