Docker
Overview and Use Case

--Jannat

BRAC UNIVERSITY

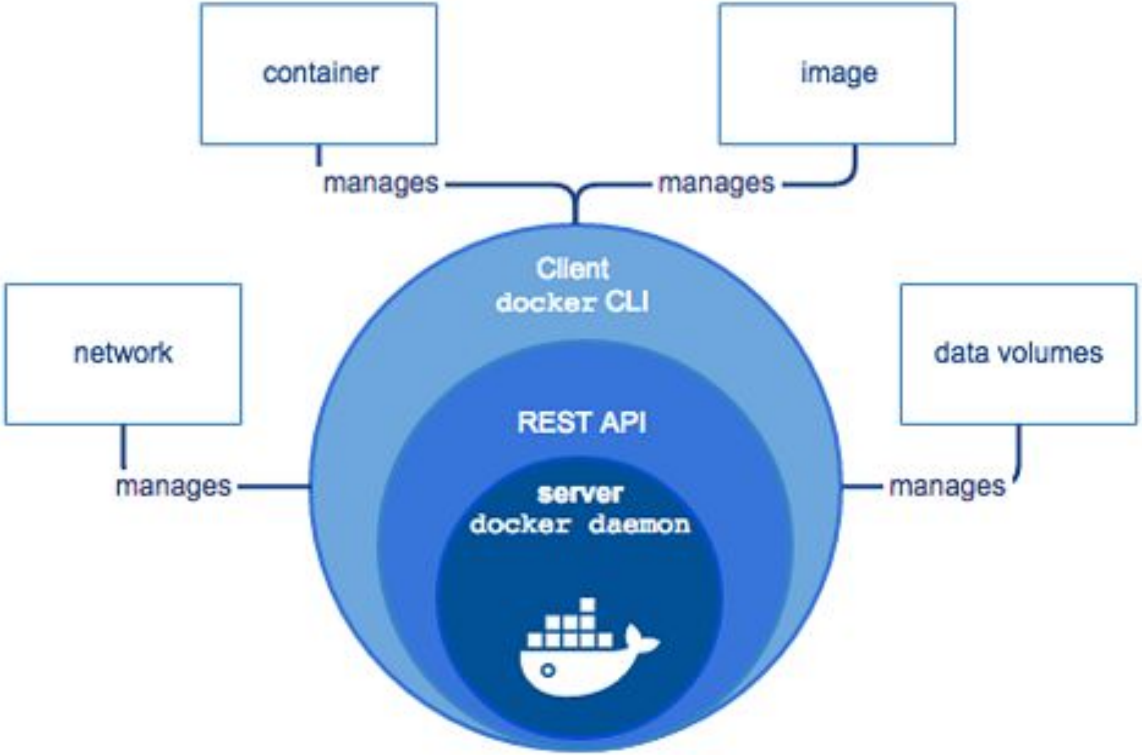
Inspiring Excellence

Topics

- What is Virtualization?
- What is Containerization
- Advantages of Containerization over Virtualization
- Introduction to Docker
- Benefits of Docker
- Virtualization vs Containerization
- Docker Installation
- Dockerfile, Docker Image & Docker Container
- What is Docker Hub?
- Docker Architecture
- Docker Compose
- Basic command outputs

Docker Architecture

- **Docker's Client**
- **Docker Host**
- **Docker Objects**
  - 1. Images
  - 2. Containers
  - 3. Networks
    - Bridge
    - Host
    - Overlay
    - None
    - Macvlan
  - 4. Storage
    - Data Volumes
    - Volume Container
    - Directory Mounts
    - Storage Plugins
- **Docker's Registry/Hub**



Docker Architecture

The architecture of Docker uses a client-server model and consists of the Docker’s Client, Docker Host, Network and Storage components, and the Docker Registry/Hub. Let’s look at each of these in some detail.

Docker’s Client

- Docker users can interact with Docker through a client. When any docker commands runs, the client sends them to dockerd daemon, which carries them out. Docker API is used by Docker commands. It is possible for Docker client to communicate with more than one daemon.

Docker Host

- The Docker host provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage. As previously mentioned, the daemon is responsible for all container-related actions and receives commands via the CLI or the REST API. It can also communicate with other daemons to manage its services.

Docker’s Registry/Hub

- Docker registries are services that provide locations from where you can store and download images. In other words, a Docker registry contains Docker repositories that host one or more Docker Images. Public Registries include two components namely the **Docker Hub and Docker Cloud.** You can also use Private Registries. The most common commands when working with registries include: **docker push, docker pull, docker run**

Docker Architecture (Cont.)

Docker Objects

1. Images

- Images are nothing but a read-only binary template that can build containers.
- They also contain metadata that describe the container’s capabilities and needs.
- Images are used to store and ship applications.
- An image can be used on its own to build a container or customized to add additional elements to extend the current configuration.
- You can share the container images across teams within an enterprise with the help of a private container registry, or share it with the world using a public registry like Docker Hub.
- Images are the core element of the Docker experience as they enable collaboration between developers in a way that was not possible before.

Docker Architecture (Cont.)

Docker Objects

2. Containers

- Containers are sort of encapsulated environments in which you run applications.
- Container is defined by the image and any additional configuration options provided on starting the container, including and not limited to the network connections and storage options.
- Containers only have access to resources that are defined in the image, unless additional access is defined when building the image into a container.
- You can also create a new image based on the current state of a container.
- Since containers are much smaller than VMs, they can be spun in a matter of seconds, and result in much better server density

Docker Architecture (Cont.)

Docker Objects

3. Networks

Docker networking is a passage through which all the isolated container communicate. There are mainly five network drivers in docker:

1. **Bridge:** It is the default network driver for a container. You use this network when your application is running on standalone containers, i.e. multiple containers communicating with the same docker host.
2. **Host:** This driver removes the network isolation between docker containers and docker host. You can use it when you don’t need any network isolation between host and container.
3. **Overlay:** This network enables swarm services to communicate with each other. You use it when you want the containers to run on different Docker hosts or when you want to form swarm services by multiple applications.
4. **None:** This driver disables all the networking.
5. **macvlan:** This driver assigns mac address to containers to make them look like physical devices. It routes the traffic between containers through their mac addresses. You use this network when you want the containers to look like a physical device, for example, while migrating a VM setup.

Docker Architecture (Cont.)

Docker Objects

4. Storage

You can store data within the writable layer of a container but it requires a storage driver. Being non-persistent, it perishes whenever the container is not running. Moreover, it is not easy to transfer this data. With respect to persistent storage, Docker offers four options:

1. **Data Volumes:** They provide the ability to create persistent storage, with the ability to rename volumes, list volumes, and also list the container that is associated with the volume. Data Volumes are placed on the host file system, outside the containers copy on write mechanism and are fairly efficient.
2. **Volume Container:** It is an alternative approach wherein a dedicated container hosts a volume and to mount that volume to other containers. In this case, the volume container is independent of the application container and therefore you can share it across more than one container.
3. **Directory Mounts:** Another option is to mount a host's local directory into a container. In the previously mentioned cases, the volumes would have to be within the Docker volumes folder, whereas when it comes to Directory Mounts any directory on the Host machine can be used as a source for the volume.
4. **Storage Plugins:** Storage Plugins provide the ability to connect to external storage platforms. These plugins map storage from the host to an external source like a storage array or an appliance. You can see a list of storage plugins on Docker's Plugin page.

Difference between Docker Swarm, Compose, and Networks

Docker Compose is basically used to run multiple Docker Containers as a single server. Let me give you an example:

Suppose if I have an application which requires WordPress, Maria DB and PHP MyAdmin. I can create one file which would start both the containers as a service without the need to start each one separately. It is really useful especially if you have a microservice architecture.

- **Docker Swarm**: Docker Swarm is the term which is used when you are trying to manage containers across multiple physical or virtual machines.
- **Docker Compose**: This term is used when you are trying to define a multi-container application.
- **Docker Networks**: It helps you achieve where you would want your containers to be able to talk to each other and also to the external world.

Difference between Docker Swarm, Compose, and Networks (Cont.)

- **Docker Compose is a tool to define and run multi-container applications.** Multi-container applications are applications where multiple containers work together to provide the required functionality.
- You define your application in the Compose file.
- You can run your application with the \`docker-compose\` command. The \`docker-compose\` command takes your Compose file as input and makes sure that your application's state is what you described in the Compose file (we call this the desired state).
- **Docker Compose can run your multi-container application on a SINGLE HOST ONLY,** it cannot run your application on a computer cluster.
- *If you want to run your application on a COMPUTER CLUSTER, then you need Docker Swarm.*

Difference between Docker Swarm, Compose, and Networks (Cont.)

***Docker Swarm* runs multi-container applications, just like Compose does. The key difference is that Swarm schedules and manages your containers across multiple machines, while *Compose* schedules and manages containers on a single host only.**

You can use a standard Compose file to deploy your application to the Swarm. It’s the same Compose file that you use with Compose, but some options are limited to either Swarm or Compose in the Compose file.

Difference between Docker Swarm, Compose, and Networks (Cont.)

- **Docker Networks is a toolset to define how your containers connect to each other.**
- It is an architecture design decision to specify which containers talk to each other on the same network, how many networks your application has to separate concerns and what technologies you use to create gateways between these networks.
- You define your network configuration in the Compose file. (You can create and manage Docker networks with the `docker network` command, but it’s better to define you configuration in the form of code in the Compose file.)
- You usually define your own networks, these are called user defined networks.

Some commands output

```
jannat@jannat-lp:~$ systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2019-10-24 13:33:41 +06; 13min ago
     Docs: https://docs.docker.com
 Main PID: 20276 (dockerd)
    Tasks: 14
   CGroup: /system.slice/docker.service
           └─20276 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
22e816666fd6: Pull complete
079b6d2a1e53: Pull complete
11048ebae908: Pull complete
c58094023a2e: Pull complete
Digest: sha256:a7b8b7b33e44b123d7f997bd4d3d0a59fafc63e203d17efedf09ff3f6f516152
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker info
Client:
 Debug Mode: false

Server:
 Containers: 1
  Running: 0
  Paused: 0
  Stopped: 1
 Images: 2
 Server Version: 19.03.4
 Storage Driver: overlay2
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 ……..
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker images
REPOSITORY          TAG                   IMAGE ID            CREATED              SIZE
ubuntu              latest              cf0f3ca922e0       5 days ago          64.2MB
hello-world         latest              fce289e99eb9        9 months ago        1.84kB

**************

root@jannat-lp:/home/jannat# cd /var/lib/docker/
root@jannat-lp:/var/lib/docker# ls
builder  buildkit  containers  image  network  overlay2  plugins  runtimes  swarm  tmp
trust  volumes

root@jannat-lp:/var/lib/docker/containers# ls
c132c5b907824242f4acb94adf2bdfada1c1dc0611ea385b0cf879cb4289ca1b

********************************************
```

Some commands output

```
#### running a docker container in iterative mode
jannat@jannat-lp:~$ sudo docker run -it ubuntu
root@9cd9097fb108:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@9cd9097fb108:/# cd home/
root@9cd9097fb108:/home# ls
root@9cd9097fb108:/home# ifconfig
bash: ifconfig: command not found
root@9cd9097fb108:/home# ip addr
bash: ip: command not found
root@9cd9097fb108:/home# ls
root@9cd9097fb108:/home# cd ~
root@9cd9097fb108:~# ls
root@9cd9097fb108:~# python
bash: python: command not found
root@9cd9097fb108:~# pwd
/root
root@9cd9097fb108:~# touch abc
root@9cd9097fb108:~# ls
abc
```

Some commands output

```
/// new container using same image
jannat@jannat-lp:~$ sudo docker run -it ubuntu /bin/bash
root@12bbfd6c3441:/# cp /root/.
./        ../       .bashrc   .profile
root@12bbfd6c3441:/# cat /etc/os-release
root@12bbfd6c3441:/# apt-get update
...
Reading package lists... Done
root@12bbfd6c3441:/# apt-get install net-tools
...
root@12bbfd6c3441:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 11729  bytes 18265904 (18.2 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 7337  bytes 501203 (501.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker ps -a
CONTAINER ID      IMAGE             COMMAND           CREATED           STATUS
PORTS             NAMES
12bbfd6c3441      ubuntu            "/bin/bash"       12 minutes ago    Exited (0) 12
seconds ago                         mystifying_curran
9cd9097fb108      ubuntu            "/bin/bash"       18 minutes ago    Exited (0) 13
minutes ago                         happy_ritchie
c132c5b90782      hello-world       "/hello"          About an hour ago   Exited (0) About
an hour ago                         epic_hodgkin
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker start 9cd9097fb108
9cd9097fb108
jannat@jannat-lp:~$ sudo docker ps -a
CONTAINER ID     IMAGE          COMMAND          CREATED          STATUS                    PORTS
NAMES
0c33075d3df7     ubuntu/curl     "/bin/bash"     2 minutes ago    Exited (0) 59 seconds ago
compassionate_mcclintock
56e72bdc1605     ubuntu/curl     "/bin/bash"     3 minutes ago    Exited (0) 3 minutes ago
pensive_yalow
7001192b0198     ubuntu/curl     "/bin/bash"     5 minutes ago    Exited (0) 5 minutes ago
quizzical_roentgen
7e2166c3b3bf     ubuntu/curl     "/bin/bash"     5 minutes ago    Exited (0) 5 minutes ago
recursing_bell
12bbfd6c3441     ubuntu          "/bin/bash"     19 minutes ago   Exited (0) 7 minutes ago
mystifying_curran
9cd9097fb108     ubuntu          "/bin/bash"     25 minutes ago   Up 44 seconds                       happy_ritchie
c132c5b90782     hello-world     "/hello"        2 hours ago      Exited (0) 2 hours ago              epic_hodgkin
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker attach 9cd9097fb108
root@9cd9097fb108:/# python
bash: python: command not found
root@9cd9097fb108:/# pip install python
bash: pip: command not found
root@9cd9097fb108:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr
var
root@9cd9097fb108:/# cd /root/
root@9cd9097fb108:~# ls
abc
root@9cd9097fb108:~# cat abc
efjeljfklhjfg
```

Some commands output

```
jannat@jannat-lp:~$ sudo docker commit 3a819128ce43 jubuntu:25-10-2019-001
sha256:3f75d4ab1d82518169d4683dcab58a090f497a1041ec1598a39969797cc94a15
jannat@jannat-lp:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
jubuntu             25-10-2019-001      3f75d4ab1d82        8 seconds ago       184MB
ubuntu/curl         latest              3a4bf8ff60f5        2 hours ago         106MB
ubuntu              latest              cf0f3ca922e0        5 days ago          64.2MB
hello-world         latest              fce289e99eb9        9 months ago        1.84kB

jannat@jannat-lp:~$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
3a819128ce43        ubuntu/curl         "/bin/bash"         12 minutes ago      Exited (1) 3
minutes ago                             quizzical_ellis
```

Ways to use Containers

1. **To run a single task:** This could be a shell script or a custom app.
2. **Interactively:** This connects you to the container similar to the way you SSH into a remote server.
3. **In the background:** For long-running services like websites and databases.

Run a single task in an Alpine Linux container

Run the following command in your Linux console.

```
docker container run alpine hostname
```

The output below shows that the `alpine:latest` image could not be found locally. When this happens, Docker automatically *pulls* it from Docker Hub. After the image is pulled, the container’s hostname is displayed (`888e89a3b36b` in the example below).

```
jannat@jannat-lp:$ docker container run alpine hostname
 Unable to find image 'alpine:latest' locally
 latest: Pulling from library/alpine
.........
 Status: Downloaded newer image for alpine:latest
 888e89a3b36b
```

Run an interactive Ubuntu container

Run a Docker container and access its shell.

```
docker container run --interactive --tty --rm ubuntu bash
```

1. In this example, we’re giving Docker three parameters:
   - `--interactive` says you want an interactive session.
   - `--tty` allocates a pseudo-tty.
   - `--rm` tells Docker to go ahead and remove the container when it’s done executing.
2. The first two parameters allow you to interact with the Docker container. We’re also telling the container to run `bash` as its main process (PID 1). When the container starts you’ll drop into the bash shell with the default prompt `root@<container id>:/#`. Docker has attached to the shell in the container, relaying input and output between your local session and the shell session in the container.

Run an interactive Ubuntu container (Cont.)

Run the following commands in the container.
`ls /` will list the contents of the root director in the container, `ps aux` will show running processes in the container, `cat /etc/issue` will show which Linux distro the container is running, in this case Ubuntu 18.04.3 LTS.

```
ls /
ps aux
cat /etc/issue
```

Type `exit` to leave the shell session. This will terminate the `bash` process, causing the container to exit.

```
 Exit
```

Note: As we used the `--rm` flag when we started the container, Docker removed the container when it stopped. This means if you run another `docker container ls --all` you won’t see the Ubuntu container.

Run a background MySQL container

Background containers are how you'll run most applications. Here's a simple example using MySQL.

Run a new MySQL container with the following command.

```
docker container run \
--detach \
--name mydb \
-e MYSQL_ROOT_PASSWORD=my-secret-pw \
mysql:latest
```

1.
    - `--detach` will run the container in the background.
    - `--name` will name it **mydb**.
    - `-e` will use an environment variable to specify the root password (NOTE: This should never be done in production).

As the MySQL image was not available locally, Docker automatically pulled it from Docker Hub.

```
Unable to find image 'mysql:latest' locallylatest: Pulling from library/mysql
aa18ad1a0d33: Pull complete
.....
....
```

Run a background MySQL container (Cont.)

As long as the MySQL process is running, Docker will keep the container running in the background.

List the running containers.

```
 docker container ls
```

Notice your container is running.

```
 CONTAINER ID        IMAGE                  COMMAND
CREATED             STATUS                PORTS              NAMES
 3f4e8da0caf7        mysql:latest
"docker-entrypoint..."   52 seconds ago      Up 51 seconds
3306/tcp            mydb
```

Run a background MySQL container (Cont.)

You can check what’s happening in your containers by using a couple of built-in Docker commands: `docker container logs` and `docker container top`.

```
 docker container logs mydb
```

This shows the logs from the MySQL Docker container.

```
   <output truncated>
   2017-09-29T16:02:58.605004Z 0 [Note] Executing 'SELECT * FROM
INFORMATION_SCHEMA.TABLES;' to get a list of tables using the deprecated partition
engine. You may use the startup option '--disable-partition-engine-check' to skip
this check.
   2017-09-29T16:02:58.605026Z 0 [Note] Beginning of list of non-natively partitioned
tables
   2017-09-29T16:02:58.616575Z 0 [Note] End of list of non-natively partitioned
tables
```

Let’s look at the processes running inside the container.

```
   docker container top mydb
```

You should see the MySQL daemon (`mysqld`) is running in the container.

```
 PID                 USER                TIME                COMMAND
 2876                999                 0:00                mysqld
```

Run a background MySQL container (Cont.)

Although MySQL is running, it is isolated within the container because no network ports have been published to the host. Network traffic cannot reach containers from the host unless ports are explicitly published.

List the MySQL version using `docker container exec`.
`docker container exec` allows you to run a command inside a container. In this example, we'll use `docker container exec` to run the command-line equivalent of `mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version` inside our MySQL container.

```
 docker exec -it mydb \
 mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
```

You will see the MySQL version number, as well as a handy warning.

```
 mysql: [Warning] Using a password on the command line interface can be
insecure.
 mysql  Ver 14.14 Distrib 5.7.19, for Linux (x86_64) using  EditLine wrapper
```

Run a background MySQL container (Cont.)

You can also use `docker container exec` to connect to a new shell process inside an already-running container. Executing the command below will give you an interactive shell (`sh`) inside your MySQL container.

```
 docker exec -it mydb sh
```

Notice that your shell prompt has changed. This is because your shell is now connected to the `sh` process running inside of your container.

Let’s check the version number by running the same command again, only this time from within the new shell session in the container.

```
 mysql --user=root --password=$MYSQL_ROOT_PASSWORD --version
```

Notice the output is the same as before.

Type `exit` to leave the interactive shell session.

```
 exit
```

Reference

https://www.edureka.co/blog/docker-tutorial
https://docs.docker.com/
https://www.edureka.co/blog/docker-commands/
https://www.edureka.co/blog/docker-architecture/
https://www.quora.com/Whats-the-difference-between-Docker-Swarm-Docker-Compose-and-Docker-Networks
https://training.play-with-docker.com/beginner-linux/#Task_0