Date: 17.06.2025

Docker

Contents

1. Introduction to Docker 3
   a. What is Docker? 3
   b. Advantages of Docker 3
2. Container vs Virtual Machine 4
3. Docker Architecture 5
   a. The Docker daemon 5
   b. The Docker client 5
   c. Docker Desktop 5
   d. Docker registries 6
   e. Docker objects 6
      i. Images 6
      ii. Containers 6
4. Docker File 7
5. Docker Image 8
   a. Building an Image 8
   b. Tagging an Image 8
   c. Publishing an image 8
6. Docker Container 9
   a. Publishing ports 9
   b. Publishing to ephemeral ports 9
   c. Publishing all ports 9

7. Docker Compose 10
a. What is Docker Compose? 10
b. Compose File 10
c. Key Commands 10
i. To start all the services defined in your compose.yaml file 10
ii. To stop and remove the running services 10
iii. Monitor output of running containers and debug issues (view logs) 10
iv. List all the services with their current status 10
8. Docker Networking 11
a. What is Docker Networking? 11
b. Types of Network Drivers 11
c. Key Commands 11
i. Creating a custom Docker network and deploy container 11
ii. Connecting a running container to an existing network 11
iii. List networks 11
iv. Find details of a Docker Network 11
v. Remove a container from the network 11
vi. Remove a network 12
vii. Remove all unused networks 12
9. Docker Swarm 12
a. What is Docker Swarm? 12
b. How it works? 12
c. Docker Container vs Docker Swarm 13
10. Docker Alternatives 14
a. Podman 14
b. LXC/LXD 14
c. Red Hat OpenShift 15

1. Introduction to Docker

a. What is Docker?

- Docker is a virtualization tool that allows you to develop, ship and run applications within lightweight containers.
- It allows the separation of infrastructure (packages, dependencies, configurations, system tools and runtime) from software, thus enabling quick software delivery.
- It reduces the delay between writing code and running it in production, thus making developing and deploying applications much easier.

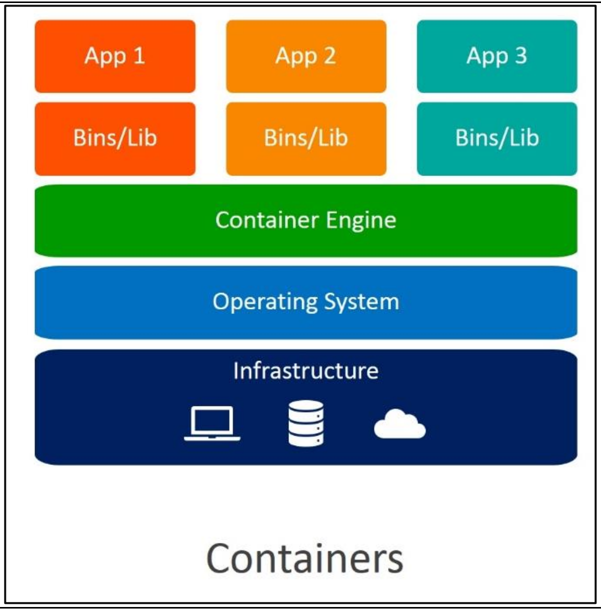
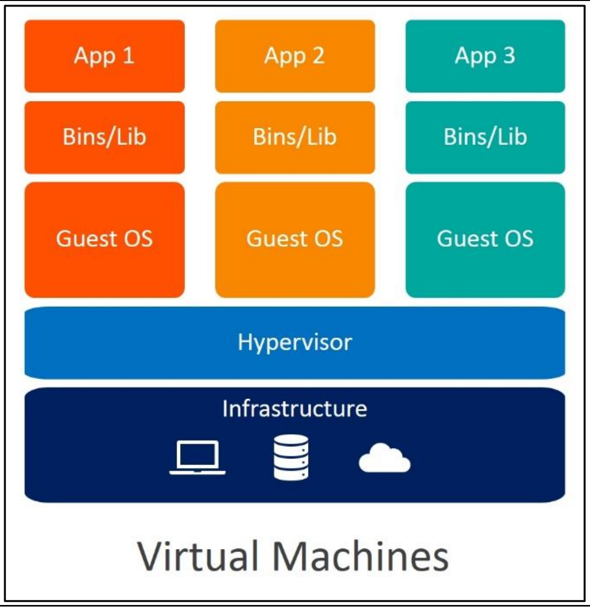
b. Advantages of Docker

i. **Fast, consistent delivery of your applications**
– Simplifies the development process by enabling developers to operate in uniform environments through local containers that host your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

ii. **Responsive deployment and scaling**
– Containers’ portability and lightweight nature makes dynamically managing workloads easy, scaling up or tearing down applications and services whenever required.

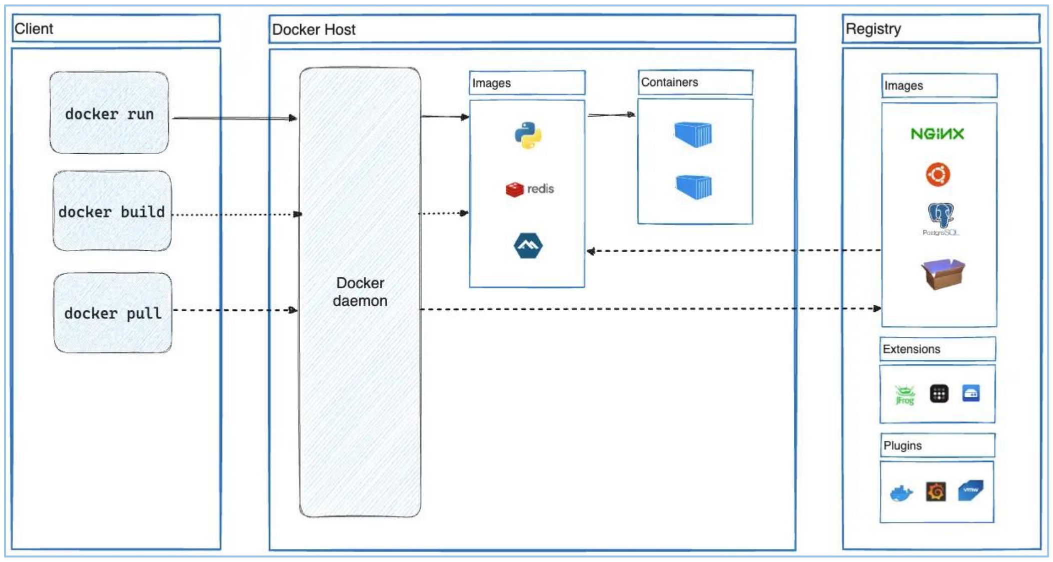
iii. **Running more workloads on the same hardware**
– It is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

2. Container vs Virtual Machine

Container	VM
Software that allows different functionalities of an application independently.	Software that allows installation of other software inside of it that can be controlled virtually as opposed to installing the software directly on the computer.
Applications running in a container environment share a single OS.	Applications running on a VM system, or hypervisor, can run different OS.
Containers virtualize the OS/software only.	VM virtualizes the computer system hardware.
Container size is very light, generally a few 100 MBs.	VM size is very large, generally in GBs.
Each container shares the host OS' kernel.	Each VM has its own OS.
Faster startup time.	Slower startup time.
Requires very less memory.	Uses lot of system memory.
Less secure as the virtualization is software-based and memory is shared.	More secure as the underlying hardware isn't shared between processes.
Useful when maximizing of the running applications is required using minimal servers.	Useful when all of the OS' resources are required to run various applications.
 App 1 App 2 App 3 Bins/Lib Bins/Lib Bins/Lib Container Engine Operating System Infrastructure Containers	 App 1 App 2 App 3 Bins/Lib Bins/Lib Bins/Lib Guest OS Guest OS Guest OS Hypervisor Infrastructure Virtual Machines

3. Docker Architecture

- Docker uses a client-server architecture.
- The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.
- Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



a. The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

b. The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

c. Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper.

d. Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker looks for images on Docker Hub by default. You can even run your own private registry.

When you use the docker pull or docker run commands, Docker pulls the required images from your configured registry. When you use the docker push command, Docker pushes your image to your configured registry.

e. Docker objects

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.

i. Images

- An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization.
- For example, you may build an image which is based on the Ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.
- You might create your own images or you might only use those created by others and published in a registry.
- To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt.
- This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

ii. Containers

- A container is a runnable instance of an image.
- You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it.
- When a container is removed, any changes to its state that aren't stored in persistent storage disappear.

4. Docker File

A Dockerfile is a text-based document that's used to create a container image. It provides instructions to the image builder on the commands to run, files to copy, startup command, and more.

Some of the most common instructions in a Dockerfile include:

Syntax	Description
`FROM <image>`	Specifies the base image that the build will extend.
`WORKDIR <path>`	Sets the working directory for subsequent instructions.
`COPY <host-path> <image-path>`	Copies files from the host to the container image.
`RUN <command>`	Executes a command during the image build process.
`ENV <name> <value>`	Sets an environment variable in the container.
`EXPOSE <port-number>`	Informs Docker that the container listens on the specified port.
`USER <user-or-uid>`	Sets the default user for executing subsequent instructions.
`CMD ["<command>", "<arg1>"]`	Specifies the default command to run when a container starts from the image.

Example Dockerfile:

```
FROM python:3.12
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
COPY src ./src
EXPOSE 5000

# Setup an app user so the container doesn't run as the root user
RUN useradd app
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Here is a reference to all dockerfile instructions: Dockerfile reference | Docker Docs

5. Docker Image

a. Building an Image

- Images are built using a Dockerfile.
- The most basic docker build command:

```
docker build <dir>
```

- `<dir>` provides the path or URL to the build context. At this location, the builder will find the Dockerfile and other referenced files.
- With the previous command, the image will have no name but the ID of the image.
- The produced id might be: `sha256:9924dfd9350407b3df01d1a0e1033b1e543523ce7d5d5e2c83a724480ebe8f00`
- We may start a container by using the referenced image id:

```
docker run
sha256:9924dfd9350407b3df01d1a0e1033b1e543523ce7d5d5e2c83a724480ebe8f00
```

- That name isn't memorable, which is where tagging becomes useful.

b. Tagging an Image

- Tagging images is the method to provide an image with a memorable name.
- However, there is a structure to the name of an image: `[HOST[:PORT_NUMBER]/]PATH[:TAG]`

Component	**Description**
`HOST`	Optional registry hostname; defaults to Docker's public registry (`docker.io`) if not specified.
`PORT_NUMBER`	Registry port number, used when a custom hostname is provided.
`PATH`	Image path as `[NAMESPACE/]REPOSITORY`; default namespace is `library` (used for Docker Official Images).
`TAG`	Human-readable identifier for image version/variant; defaults to latest if not specified.

- To tag an image during a build, add the `-t` or `--tag` flag:

```
docker build -t my-username/my-image <dir>
```

- Add another tag to the image using `image tag` command:

```
docker image tag my-username/my-image another-username/another-image:v1
```

c. Publishing an image

- Once an image is built and tagged, push it to a registry using the docker push command:

```
docker push my-username/my-image
```

- All layers for your image will be pushed to the registry.

6. Docker Container

a. Publishing ports

- Containers provide isolated processes for each component of your application.
- Containers are isolated for security and dependency management, which means you can't access them directly – like opening a web app in your browser.
- Publishing a port breaks this isolation by forwarding traffic from a host port (e.g., `8080`) to a container port (e.g., `80`).
- Ports are published during container creation using the -p (or --publish) flag with docker run command:

```
docker run -d -p HOST_PORT:CONTAINER_PORT nginx
```

Term	Description
`HOST_PORT`	The port number on your host machine where you want to receive traffic.
`CONTAINER_PORT`	The port number within the container that's listening for connections.

- For example, to publish the container's port `80` to host port `8080`:

```
docker run -d -p 8080:80 nginx
```

- Now, any traffic sent to port 8080 on your host machine will be forwarded to port 80 within the container.

b. Publishing to ephemeral ports

- To publish a port without a specific host port, Docker can choose a port by itself.
- To do so, simply omit the HOST_PORT configuration.
- For example, the following command will publish the container's port `80` onto an ephemeral port on the host:

```
docker run -p 80 nginx
```

- Once the container is running, to view the chosen port, use:

```
docker ps
```

c. Publishing all ports

- When creating a container image, the `EXPOSE` instruction is used to indicate the port that the packaged application will use. These ports aren't published by default.
- `-P` or `--publish-all` flag, automatically publishes all exposed ports to ephemeral ports.
- Quite useful when trying to avoid port conflicts in development or testing environments.
- For example, the following command will publish all of the exposed ports configured by the image:

```
docker run -P nginx
```

7. Docker Compose

a. What is Docker Compose?

- Docker Compose is a tool for defining and running multi-container applications.
- Compose simplifies the control of your entire application stack, making it easy to manage services, networks, and volumes in a single YAML configuration file.
- With a single command, create and start all the services from the config file.
- Compose works in all environments – production, staging, development, testing, and CI workflows.
- It also has commands for managing the whole lifecycle of your application:
  - Start, stop, and rebuild services
  - View the status of running services
  - Stream the log output of running services
  - Run a one-off command on a service

b. Compose File

- The compose file must be placed within the working directory.
- The name for a Compose file is –
  - `compose.yaml` (preferred)
  - `compose.yml`
  - `docker-compose.yaml` (for backwards compatibility)
  - `docker-compose.yml` (for backwards compatibility)
- If both files exist, Compose prefers the canonical `compose.yaml`.
- Use fragments and extensions to keep Compose file efficient and easy to maintain.
- Multiple Compose files can be merged together to define the application model. The combination of YAML files is implemented by appending or overriding YAML elements based on the Compose file order you set.

c. Key Commands

i. To start all the services defined in your compose.yaml file

```
docker compose up
```

ii. To stop and remove the running services

```
docker compose down
```

iii. Monitor output of running containers and debug issues (view logs)

```
docker compose logs
```

iv. List all the services with their current status

```
docker compose ps
```

8. Docker Networking

a. What is Docker Networking?

- Docker Networking allows you to create a network of Docker containers managed by a master node called as Manager.
- Containers inside the Docker network can communicate by sharing packets of information.
- The Docker Network is a virtual network created by Docker to enable communication between Docker containers.
- If two containers are running on the same host, they can communicate with each other without the need for ports to be exposed to the host machine.

b. Types of Network Drivers

Driver	Description
`bridge`	The default network driver.
`host`	Remove network isolation between the container and the Docker host.
`none`	Completely isolate a container from the host and other containers.
`overlay`	Overlay networks connect multiple Docker daemons together.
`ipvlan`	IPvlan networks provide full control over both IPv4 and IPv6 addressing.
`macvlan`	Assign a MAC address to a container.

c. Key Commands

i. Creating a custom Docker network and deploy container

```
docker network create -d bridge NETWORK
docker run --network=NETWORK -itd --name=CONTAINER_NAME busybox
```

ii. Connecting a running container to an existing network

```
docker network connect NETWORK CONTAINER_NAME
```

iii. List networks

```
docker network ls
```

iv. Find details of a Docker Network

```
docker network inspect NETWORK
```

v. Remove a container from the network

```
docker network disconnect NETWORK CONTAINER_NAME
```

vi. **Remove a network**

```
docker network rm NETWORK
```

vii. **Remove all unused networks**

```
docker network prune
```

9. Docker Swarm

a. What is Docker Swarm?

- Docker Swarm is a native clustering and container orchestration tool built into Docker.
- It allows you to manage a group (or "swarm") of Docker hosts as a single virtual system.
- It enables you to deploy, scale, and manage containerized applications across multiple machines.
- Thus, making it easier to handle production workloads and maintain high availability.

b. How it works?

Term	Definition
`Nodes`	Individual Docker hosts participating in the swarm, acting as managers or workers.
`Managers`	Nodes that handle orchestration, scheduling, and maintain the cluster state.
`Workers`	Nodes that execute the tasks assigned by manager nodes.
`Services`	Definitions of tasks (containers) to run on the swarm, including scaling and networking configurations.
`Tasks`	Individual containers managed by Docker Swarm as part of a service.
`Load Balancing`	A load balancer that process requests across all containers in the service.

- A task is a running container managed by a swarm manager as part of a swarm service, unlike standalone containers.
- Standalone containers can run on any daemon, while only swarm managers control swarm services.
- Docker Compose is used to define and run containers; Swarm is used to define and run service stacks.

c. Docker Container vs Docker Swarm

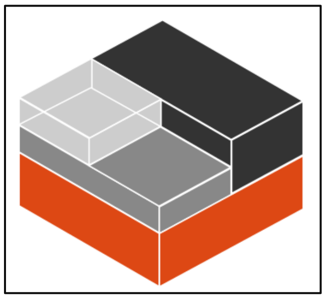
Aspect	Docker Container	Docker Swarm
Definition	Package with code and dependencies to run an app.	Tool for managing a cluster of Docker containers.
Purpose	Runs individual containers.	Orchestrates containers across multiple nodes.
Portability	Runs on any OS via Docker runtime.	Manages containers on multiple systems.
Architecture	Isolated unit for applications.	Cluster-based container management.
Use Case	Suited for microservices.	Used for scaling, deploying, and load-balancing containers.
Management Scope	Local (single host).	Distributed (multi-host cluster).

10. Docker Alternatives

a. Podman

- Podman is a tool for managing containers and images, volumes mounted into those containers, and pods made from groups of containers.
- It is OCI (Open Container Initiative) compliant
- It runs containers on Linux, but can also be used on Mac and Windows systems using a Podman-managed virtual machine.
- It is based on libpod, a library for container lifecycle management that is also contained in this repository.
- The libpod library provides APIs for managing containers, pods, container images, and volumes.

b. LXC/LXD



- LXD provides a unified experience for running and managing full Linux systems inside containers or virtual machines.
- It supports images for a large number of Linux distributions (official Ubuntu images and images provided by the community)
- It is built around a powerful, yet simple, REST API.
- It scales from one instance on a single machine to a cluster in a full data center rack, making it suitable for running workloads both for development and in production.
- It allows you to easily set up a system that feels like a small private cloud. You can run any type of workload in an efficient way while keeping your resources optimized.
- To control LXD, you typically use two different commands: lxd and lxc.
- lxd command controls the LXD daemon.
  lxc command is a command-line client for LXD, which you can use to interact with the LXD daemon.

c. **Red Hat OpenShift**

- Red Hat OpenShift is available as a managed application platform on the cloud provider of your choice.

Cloud Service	Infrastructure	Billed by	Managed by	Supported by
RH OpenShift Service on AWS	AWS	AWS	Red Hat and AWS	Red Hat and AWS
Microsoft Azure RH OpenShift	Azure	Microsoft	Red Hat and Microsoft	Red Hat and Microsoft
RH OpenShift Dedicated	Google Cloud	1. Red Hat (OpenShift) 2. Google Cloud (infrastructure)	Red Hat	Red Hat
RH OpenShift on IBM Cloud	IBM Cloud	IBM	IBM	Red Hat and IBM

- Red Hat OpenShift also provides self-managed editions build upon each other.

Edition	Overview
OpenShift Platform Plus	Provides a complete platform for accelerating application development and application modernization.
OpenShift Container Platform	Provides a full set of operations and developer services and tools.
OpenShift Kubernetes Engine	Provides the foundational, security-focused capabilities of enterprise Kubernetes on Red Hat Enterprise Linux CoreOS to run containers in hybrid cloud environments.
OpenShift Virtualization	Provides the proven virtualization capabilities of Red Hat OpenShift in a streamlined, cost-effective solution to deploy, manage, and scale VMs exclusively.