



19Z603 - DISTRIBUTED COMPUTING

TEAM - 3

DOCKER

DHEEKSHITHA R	22Z216
O KEERTHI	22Z243
PRAMODINI P	22Z244
AKASH S	22Z255
SANJITHA R	22Z259
SREERAGHAVAN R	22Z261

Introduction to Docker

O Keerthi | 22z43



What is Docker?

1. Docker is a platform that allows developers to build, ship, and run applications inside lightweight, portable containers.
2. Containers package applications with their dependencies, ensuring consistency across environments.
3. It eliminates the “it works on my machine” problem.



Why Use Docker?

1. **Portability:** Run anywhere (development, testing, production).
2. **Consistency:** Same environment across all stages.
3. **Efficiency:** Uses fewer resources than Virtual Machines.
4. **Scalability:** Easily scale applications with container orchestration tools.



Docker vs. Virtual Machines

Feature	Docker Containers	Virtual Machines
Size	Lightweight (MBs)	Heavy (GBs)
Boot Time	Seconds	Minutes
Resource Usage	Shares OS kernel, efficient	Requires full OS, resource-heavy
Isolation	Process-level	Full OS-level isolation
Portability	High	Limited



Real-World Use Cases

1. **Microservices:** Companies like Netflix and Uber use Docker to deploy scalable microservices.
2. **CI/CD Pipelines:** Automate application testing and deployment using Docker in DevOps.
3. **Cloud Deployments:** Docker integrates well with cloud platforms like AWS, Azure, and Google Cloud.
4. **Software Development:** Developers use Docker to create consistent environments for testing and development.

Docker Installation & Basic Commands



Docker Desktop

Docker Desktop is a **one-click-install application** for your **Mac, Linux, or Windows** environment that lets you build, share, and run containerized applications and microservices.

It provides a **straightforward GUI** (Graphical User Interface) that lets you **manage your containers, applications, and images directly from your machine**.

Docker Desktop **reduces the time spent** on complex setups so you can focus on writing code. It takes care of port mappings, file system concerns, and other default settings, and is **regularly updated** with bug fixes and security updates.



Products ▾

Developers ▾

Pricing

Support

Blog

Company ▾



Sign In

Get started

Get Started with Docker

Build applications faster and more securely with Docker for developers

[Learn how to install Docker](#)

[Download Docker Desktop](#)



Download for Mac - Intel Chip



Download for Mac - Apple Silicon



Download for Windows - AMD64

An experier

Customize your development experier

tech stack




What's included in Docker Desktop?

What are the key features of Docker Desktop?

- [Docker Engine](#)
- Docker CLI client
- [Docker Scout](#) (additional subscription may apply)
- [Docker Build](#)
- [Docker Extensions](#)
- [Docker Compose](#)
- [Docker Content Trust](#)
- [Kubernetes](#)
- [Credential Helper](#)

Docker Desktop works with your **choice of development tools and languages** and gives you **access to a vast library** of certified images and templates in [Docker Hub](#). This allows development teams to **extend their environment** to rapidly auto-build, continuously integrate, and collaborate using a secure repository.




Install Docker Desktop

Install Docker Desktop on [Mac](#), [Windows](#), or [Linux](#).




Explore Docker Desktop

Navigate Docker Desktop and learn about its key features.




View the release notes

Find out about new features, improvements, and bug fixes.



Browse common FAQs

Explore general FAQs or FAQs for specific platforms.



Find additional resources

Find information on networking features, deploying on Kubernetes, and more.



Give feedback

Provide feedback on Docker Desktop or Docker Desktop features.

Installing Docker (Windows, Mac, Linux)

Windows:

1. Download **Docker Desktop**
2. Install and enable **WSL 2 Backend** (for Windows 10/11).
3. Start Docker Desktop.

Mac:

4. Download **Docker Desktop** for Mac.
5. Install and run Docker.

Linux (Ubuntu/Debian):

```
sudo apt update
```

```
sudo apt install docker.io -y
```

```
sudo systemctl start docker
```

```
sudo systemctl enable docker
```

Verify Installation: `docker --version`

Walkthrough : <https://docs.docker.com/desktop/>



Run Docker Desktop for Windows in a VM or VDI environment

To run Docker Desktop in a virtual desktop environment, it is essential **nested virtualization** is enabled on the virtual machine that provides the virtual desktop. This is because, under the hood, Docker Desktop is using a **Linux VM** in which it runs Docker Engine and the containers.

The only hypervisors we have successfully tested are VMware ESXi and Azure, and there is no support for other VMs



BASIC DOCKER INSTRUCTIONS

<https://docs.docker.com/reference/dockerfile/>

<https://docker-curriculum.com/>

Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Specify default commands.
COPY	Copy files and directories.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.
LABEL	Add metadata to an image.

MAINTAINER

Specify the author of an image.

ONBUILD

Specify instructions for when the image is used in a build.

RUN

Execute build commands.

SHELL

Set the default shell of an image.

STOPSIGNAL

Specify the system call signal for exiting a container.

USER

Set user and group ID.

VOLUME

Create volume mounts.

WORKDIR

Change working directory.

Running a Simple Container

1. Run a Hello World Container

```
docker run hello-world
```

2. Run a Ubuntu Container & Access It

```
docker run -it ubuntu bash
```

3. Check Running Containers

```
docker ps
```

4. Stop a Running Container

```
docker stop <container_id>
```

5. Remove a Stopped Container

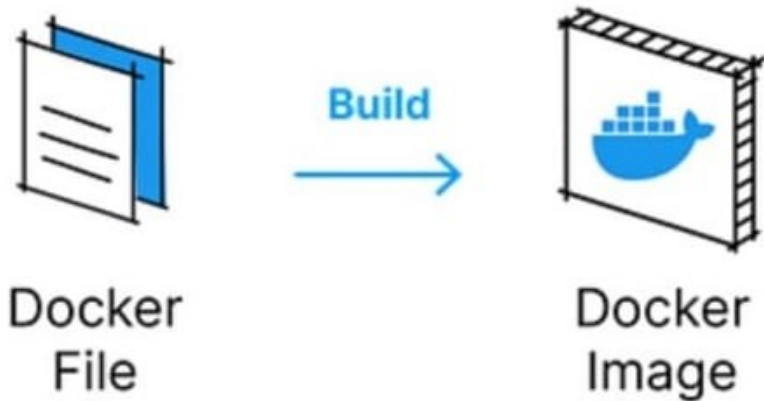
```
docker rm <container_id>
```

Dockerfile , Images and Containers

Pramodini P| 22z44

Dockerfile

A Dockerfile is a **text document** that contains all the commands a user could call on the command line to assemble an image. This page describes the commands you can use in a Dockerfile.



Format

Here is the format of the Dockerfile:

```
# Comment
```

```
INSTRUCTION arguments
```

Instruction	Description
ADD	Add local or remote files and directories.
ARG	Use build-time variables.
CMD	Specify default commands.
COPY	Copy files and directories.
ENTRYPOINT	Specify default executable.
ENV	Set environment variables.
EXPOSE	Describe which ports your application is listening on.
FROM	Create a new build stage from a base image.
HEALTHCHECK	Check a container's health on startup.
LABEL	Add metadata to an image.
MAINTAINER	Specify the author of an image.
ONBUILD	Specify instructions for when the image is used in a build.
RUN	Execute build commands.
SHELL	Set the default shell of an image.
STOPSIGNAL	Specify the system call signal for exiting a container.
USER	Set user and group ID.
VOLUME	Create volume mounts.
WORKDIR	Change working directory.

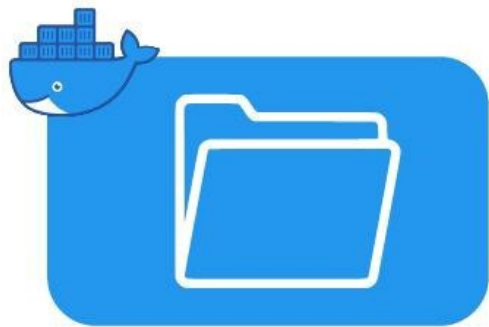
Dockerfile Example

```
Dockerfile X
Dockerfile > ...
1 FROM python:3.8
2 COPY app.py /app/app.py
3 WORKDIR /app
4 RUN pip install flask
5 CMD ["python", "app.py"]
6
7
8
```

Docker Image and Docker and Container

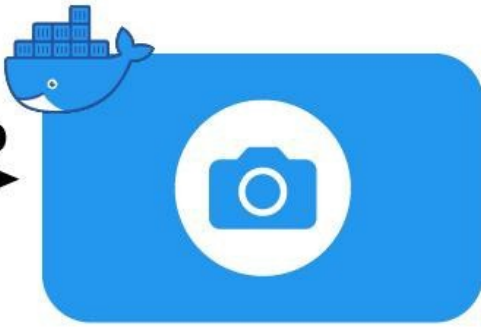
Docker Image is an **executable package** of software that includes everything needed to run an application. This image informs how a container should instantiate, determining which software components will run and how.

Docker Container is a **virtual environment** that bundles application code with all the dependencies required to run the application. The application runs quickly and reliably from one computing environment to another.



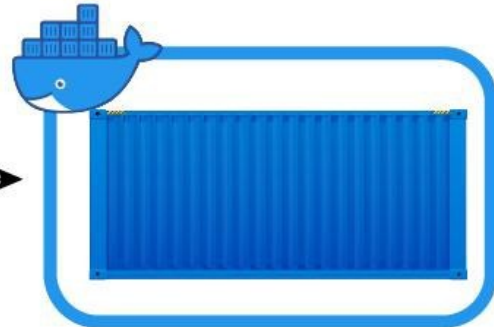
Docker File

BUILD
→



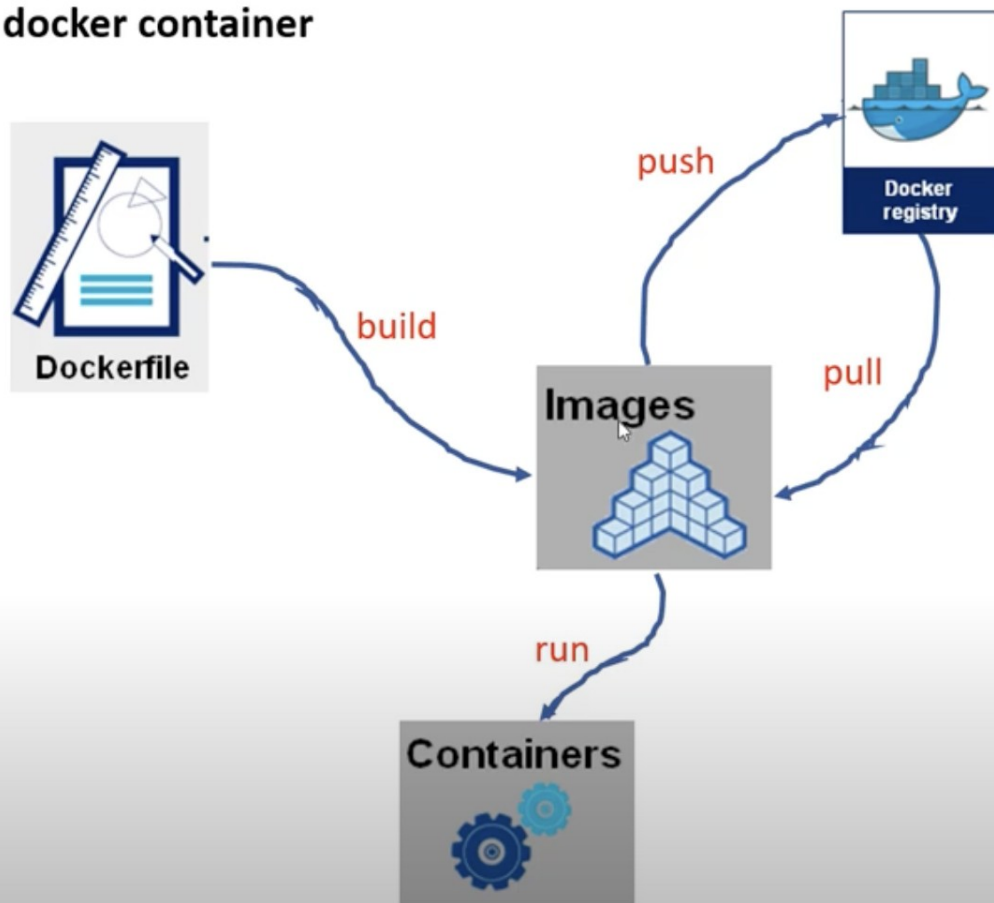
Docker Image

RUN
→



Docker Container

Dockerfile, Docker registry, Docker image and docker container



Docker Architecture

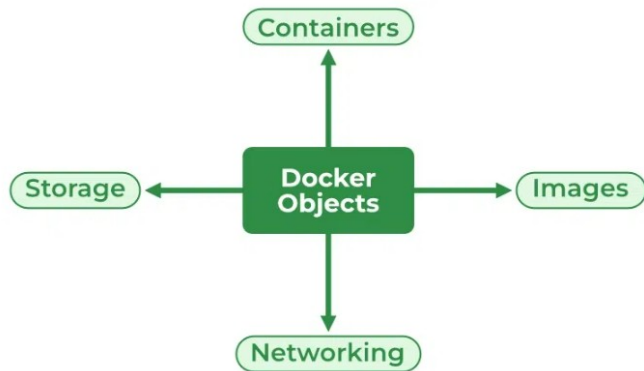
Sanjitha R | 22z259



Docker Architecture

- Docker uses a **Client-Server architecture**
- It has three main components:
 - a. **Docker Client** – Sends commands to the Docker Engine.
 - b. **Docker Engine (Daemon)** – Builds and runs containers.
 - c. **Docker Registry** – Stores and distributes images.

Docker Objects



- **Docker Images** - A Docker Image is a pre-configured template that contains an application and all its dependencies.
- **Docker Containers** - A Docker Container is a running instance of an image.
- **Docker Networks** - Networks are used for communication between containers
- **Docker Volumes** - Volumes provide persistent storage outside the container.

Docker Client

- It provides a way for users to interact with docker
- It sends requests to the Docker Daemon

Example: `docker ps`

```
$ docker ps -a
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
7557d0720549   tomcat     "catalina.sh run"       3 minutes ago   Exited (143) 9 seconds ago
jovial_rhodes   1041a2d84eff  nginx                  3 minutes ago   Up 3 minutes   80/tcp
adoring_bell
```



Docker Engine

The **Docker Engine** responsible for:

- Managing containers
- Creating and running images
- Handling networking and storage

It consists of:

- Docker Daemon – Manages Docker objects.
- REST API – Allows communication between the client and daemon.



Docker Registry

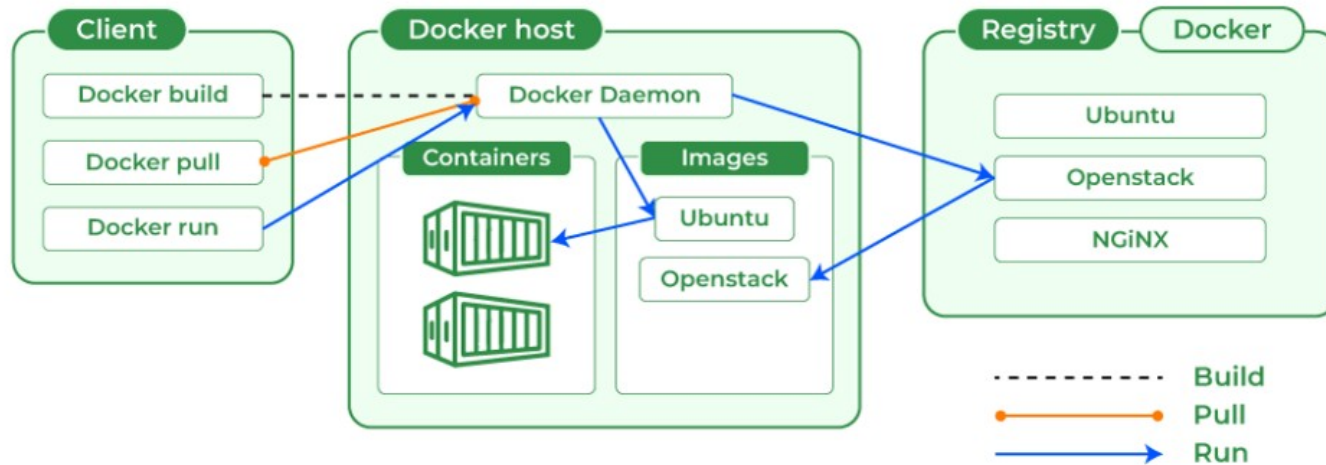
- A Docker Registry is a repository that stores Docker Images.
- Docker Hub is the default public registry.

Example:

```
docker push myrepo/myapp
```

```
docker pull myrepo/myapp
```

How Docker Works?



Docker Compose and Container Orchestration



Docker Compose

- **Management Simplification:** Simplifies managing multiple services in a microservices architecture with a single command.
- **Environment Management:** Facilitates easy recreation and collaboration with environment definition in a docker-compose.yml file.
- **Declarative Configuration:** Uses YAML to define the *desired state* of the application, ensuring consistency across different environments.
- **Network and Volume Management:** Automatically sets up networking and volume management between containers, allowing for communication and data persistence.



Docker Compose

- Was initially introduced by Docker on December 2014, under the name **Fig**
- Fig was officially replaced by `docker-compose` on July 2015
- Both `docker compose` and `docker-compose` can be used interchangeably



Docker Compose - How to define it

- Service is the main section where all the application and its configurations go
- Every single application inside service is another section
- Can configure a service's cross dependency, environment, port, restart condition, sensitive information, deployment configuration, health check settings
- Allows to define other docker infrastructure like networks and volumes as a sections of their own in separate sections

Docker Compose - Sample File

○ ○ ○

```
services:
  loki:
    image: grafana/loki:2.9.10
    ports:
      - "3100:3100"
    volumes:
      - ./loki.yaml:/etc/loki/local-config.yaml
      - ./loki-data:/loki
    command: -config.file=/etc/loki/local-config.yaml
    user: "${UID}:${UID}"

volumes:
  loki-data:
```

Docker Compose - Sample File

```
1  version: '3.8'
2
3  services:
4  >   init: ...
13
14  >   tempo: ...
29
30  >   loki: ...
39
40  >   prometheus: ...
47
48  >   grafana: ...
61
62  >   app: ...
73
74  volumes:
75  |   grafana-data:
76  |   loki-data:|
```



docker-compose.yml

&&

docker compose up
-d

```
docker run -d --name init \  
  --user root \  
  --entrypoint "chown" \  
  -v $(pwd)/tempo-data:/var/tempo \  
  grafana/tempo:latest \  
  "10001:10001" "/var/tempo"  
docker run -d --name tempo \  
  --user $(id -u):$(id -g) \  
  --entrypoint "" \  
  -v $(pwd)/tempo.yaml:/etc/tempo.yaml \  
  -v $(pwd)/tempo-data:/var/tempo \  
  -p 14268:14268 \  
  -p 3200:80 \  
  -p 9095:9095 \  
  -p 4417:4317 \  
  -p 4418:4318 \  
  -p 9411:9411 \  
  --link init:init \  
  --link init:tempo
```

It's just better in every way



```
ports:
```

```
- "3200:3100"
```

```
docker stop loki
```

```
docker rm loki
```

```
docker run -d --name loki \
  --user $(id -u):$(id -g) \
  -v
$(pwd)/loki.yaml:/etc/loki/local-
config.yaml \
-v $(pwd)/loki-data:/loki \
-p 3200:3100 \
grafana/loki:2.9.10 \
-config.file=/etc/loki/local-
config.yaml
```

It's just better in every way

```
volumes:
```

```
| kriya-loki-data:
```

```
docker stop loki
```

```
docker rm loki
```

```
docker run -d --name loki \
  --user $(id -u):$(id -g) \
  -v
$(pwd)/loki.yaml:/etc/loki/local-
config.yaml \
-v $(pwd)/kriya-loki-data:/loki \
-p 3200:3100 \
grafana/loki:2.9.10 \
-config.file=/etc/loki/local-
config.yaml
```




Kubernetes (k8s)

- **Automated Deployment & Scaling:** Deploy applications easily and scale them up or down based on demand.
- **Self-Healing:** Automatically restarts failed containers and replaces unhealthy ones.
- **Load Balancing & Service Discovery:** Distributes traffic efficiently among containers.
- **Storage Orchestration:** Manages storage dynamically for stateful applications.
- **Declarative Configuration & Automation:** Uses YAML configuration files to define application states and automates their management.



deployment.yaml

A **Deployment** file in Kubernetes is used to manage and control the lifecycle of **Pods**. It defines how many replicas of a Pod should run, how updates are handled, and how to roll back in case of failure.

Use Cases:

- Ensures the desired number of Pods are always running.
- Handles rolling updates and rollbacks.
- Manages scaling of applications.

service.yaml

A **Service** file is used to expose a group of **Pods** as a network service. Since Pods have dynamic IPs, a Service provides a stable endpoint to access them.

Use Cases:


- Enables communication between microservices within a cluster.
- Exposes applications internally or externally.
- Load balances traffic across multiple Pods.

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app-image:latest
          ports:
            - containerPort: 80
```


service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP # Other options:
NodePort, LoadBalancer
```



Resource Requests and Limits for Containers

```
spec:
  containers:
    - name: my-app
      image: my-app-image:latest
      ports:
        - containerPort: 80
      resources:
        requests:
          cpu: "100m"
          memory: "128Mi"
        limits:
          cpu: "500m"
          memory: "512Mi"
```



Horizontal Pod Autoscaler (HPA) Configuration

```
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50 #
Scale when CPU exceeds 50%
```

Real World Use Cases

Sreeraghavan R | 22z261



Enabling Microservices

- Containerization is an enabler of the microservices architecture
- It lowers the cost of dependencies
- It allows for independent scaling
- Netflix uses Docker to deploy its microservices such that each server runs in its own container



Infinite Scaling

- Kubernetes is a container orchestration platform that performs load balancing and distribution on dockers to ensure that containerized applications scale well in production environments
- This is only possible due to containerization



Legacy Software

- Software often loses support over time due to various factors
- Docker allows these softwares to still be used in modern enterprise solutions
- Docker improves overall interoperability and heterogeneity of solutions by forming an extra abstraction



Others

- Isolated Testing
- Cloud Deployment
- Edge Computing
- Gaming



Deployed Docker Solutions

A lot of tech companies use Docker for features such as CI/CD, testing and maintenance. Some of them include Spotify, Netflix, Uber, Visa, Kriya

Apart from just Docker, containerization as a concept is also implemented by other services such as containerd, Podman etc.