Docker

Introduction

Modern software applications are expected to run consistently across development, testing, and production environments. Traditionally, developers faced a common challenge known as the **"It Works on My Machine"** problem — where an application would function perfectly on one machine but fail on another due to differences in operating systems, libraries, configurations, or software versions.

Docker was created to solve this challenge through a technology known as **containerisation**. Today, Docker has become one of the most widely used platforms for application development, deployment, and cloud-native computing.

What is Docker?

Docker is a containerisation platform that packages an application along with all its dependencies, libraries, runtime environments, and configuration files into a standardised unit called a **container**. This ensures that applications run consistently regardless of the environment in which they are deployed.

In simple terms:

Docker = Application + Dependencies + Runtime + Configuration

What is Containerisation?

Containerisation is the process of packaging an application and everything it needs to run into an isolated environment known as a container. A container typically contains:

- Application Code
- Runtime Environment
- Libraries & Dependencies
- Configuration Files

Containers are lightweight because they share the host operating system kernel instead of running their own operating system — making them faster and far more resource-efficient than traditional virtual machines.

Docker vs Virtual Machines

Understanding the difference between Docker containers and Virtual Machines (VMs) is fundamental to understanding why Docker has become so widely adopted.

Feature	Docker Containers	Virtual Machines
OS Required	Shared Host OS	Separate OS per VM
Startup Time	Seconds	Minutes
Resource Usage	Low	High
Storage	Lightweight	Heavy
Performance	High	Moderate
Portability	Excellent	Limited

Docker Architecture

Docker consists of several key components that work together to build, ship, and run containers.

Docker Client

The Docker Client is the interface through which users interact with Docker. It accepts commands and sends them to the Docker Daemon to execute.

```
docker run
docker build
docker ps
docker logs
```

Docker Daemon

The Docker Daemon is the background service responsible for building images, running containers, managing networks, managing volumes, and pulling images from registries. It performs all the actual work behind Docker operations.

Docker Hub

Docker Hub is Docker's default public image registry. It stores thousands of pre-built images — including Ubuntu, Python, Nginx, MySQL, PostgreSQL, and Redis — so developers can pull them directly without building from scratch.

Docker Image

A Docker Image is a **read-only blueprint** used to create containers. It contains the application code, dependencies, runtime, and configuration. Images are built from a Dockerfile and can be shared via registries like Docker Hub.

Docker Container

A Docker Container is a **running instance** of an image. One image can create multiple independent containers. The relationship flows as:

```
Dockerfile →      Image →      Container(s)
```

Docker Desktop

Docker Desktop is the application installed on a local machine for development. It bundles together the Docker Engine, Docker CLI, Docker Compose, and a graphical user interface — simplifying Docker management for developers on macOS, Windows, and Linux.

Docker Volumes

Containers are ephemeral by nature — if a container is deleted, any data stored inside it is lost. Docker Volumes provide **persistent storage** that lives outside the container's lifecycle, ensuring data remains available even after a container is removed or recreated.

Docker Logs

Logs record application activity inside running containers, including startup messages, database connections, user events, and errors. They are essential for monitoring and troubleshooting.

```
docker logs <container-id>
```

docker Networking

Docker Networking allows containers to communicate with each other securely. Containers reference each other by **service name** rather than IP address, and Docker resolves these names via its internal DNS system.

```
backend:8000
mysql:3306
```

Docker Compose

Docker Compose manages multi-container applications using a single YAML configuration file. Instead of running each service manually, you define all services, their images, ports, and dependencies in one place.

Then start everything with a single command:

```
docker compose up
```

Docker automatically builds images, creates containers, sets up networks, and starts all services in the correct order.

What is a Dockerfile?

A Dockerfile is a plain-text file containing step-by-step instructions for building a Docker image. Each instruction adds a layer to the image.

```
FROM python:3.12
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["uvicorn", "main:app", "--host", "0.0.0.0"]
```

This tells Docker which base image to use, which directory to work in, which files to copy in, which dependencies to install, and which command to run when the container starts.

Docker Installation

Getting Docker up and running on your machine takes just a few minutes.

Step 1 — Install Docker Desktop

Download Docker Desktop from Docker's official website at **docker.com**. The installer includes the Docker Engine, Docker CLI, Docker Compose, and the Docker Desktop GUI.

Step 2 — Verify Installation

Open a terminal and run:

```
docker --version
```

You should see output similar to: *Docker version 26.x.x*

Step 3 — Test Docker

Run the following command to confirm Docker is working correctly:

```
docker run hello-world
```

If Docker is installed properly, you will see a success message confirming the setup is complete.

Deploying an Application with Docker

The following steps walk through the full workflow of containerising and running a Python application with Docker.

Step 1 — Create a Dockerfile

```
FROM python:3.12
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["uvicorn", "main:app"]
```

Step 2 — Build the Image

```
docker build -t myapp .
```

Step 3 — Verify the Image

```
docker images
```

Step 4 — Run the Container

```
docker run -p 8000:8000 myapp
```

Step 5 — Check Running Containers

```
docker ps
```

Step 6 — Access the Application

Open your browser and navigate to:

```
http://localhost:8000
```

Your application is now live inside a Docker container.

Common Docker Commands

Command	Description
docker pull nginx	Download an image from Docker Hub
docker run nginx	Create and start a container
docker ps	List all running containers
docker images	List all local images
docker stop	Stop a running container
docker rm	Remove a stopped container
docker logs	View container logs
docker build -t myapp .	Build an image from a Dockerfile

docker compose up	Start a multi-container application
docker compose down	Stop and remove all Compose services

Conclusion

Docker has transformed the way modern applications are built, packaged, and deployed. By using containers, developers can ensure consistent behaviour across all environments while reducing infrastructure complexity and overhead.

Components such as Docker Images, Containers, Volumes, Networking, and Docker Compose together make Docker an indispensable technology in modern software development, DevOps, cloud computing, and microservices architectures.

Whether you are deploying a simple web application or orchestrating hundreds of microservices with Kubernetes, Docker provides the foundation that modern engineering teams rely on every day.

Revision #2
Created 2026-07-02 13:36:56 UTC by Anagha N
Updated 2026-07-02 13:47:42 UTC by Anagha N