Docker & Docker Compose – Beginner-Friendly Guide

Introduction

Docker is a powerful tool that simplifies packaging, shipping, and running applications using containerization. The course aims to make Docker easy through explanations, visuals, demos, and hands-on labs.

Why Docker?

Traditional application setup often involves compatibility issues between operating systems, libraries, and services—commonly referred to as the **matrix from hell**. Docker helps avoid this by packaging each component in its own isolated container.

What Are Containers?

Containers are lightweight, isolated environments that share the host OS kernel but contain their own dependencies and libraries. They:

- Start quickly (in seconds)
- Take less space (usually MBs)
- Are easily portable

Docker vs Virtual Machines

Virtual Machines:

- Each has its own full OS
- Heavy and slow to start
- Provide strong isolation

Containers:

- Share the host OS kernel
- Lightweight and fast
- Designed to run specific processes

Most real-world systems use **both** containers and VMs together.

Images vs Containers

- **Images:** Templates or blueprints used to create containers.
- **Containers:** Running instances of images.

You can pull images from Docker Hub or build your own using a Dockerfile.

Installing Docker (Summary)

Docker has two editions:

- **Community Edition (CE)** – free
- **Enterprise Edition (EE)** – includes advanced features

You can install Docker on Linux, macOS, Windows, or cloud platforms. Linux installation using a convenience script is the simplest.

Basic Docker Commands

- `docker run <image>` – Run a container from an image
- `docker ps` – List running containers
- `docker ps -a` – List all containers
- `docker stop <id/name>` – Stop a container
- `docker rm <id/name>` – Remove a container
- `docker images` – List images
- `docker rmi <image>` – Remove an image
- `docker pull <image>` – Download an image
- `docker exec <container> <command>` – Run a command inside a container
- `docker run -d <image>` – Run in detached mode

Docker Compose: A Complete, Beginner-Friendly Guide

Docker Compose is one of the most important tools you will use when working with containerized applications. It allows you to define, configure, and run multi-container applications using a single YAML file—making development, deployment, and maintenance dramatically easier.

This guide distills a long lecture on Docker Compose into a clean, structured article suitable for Medium. It covers Compose fundamentals, the evolution of Compose file versions, networking, and how Compose fits into the wider Docker ecosystem.

🚀 Why Docker Compose?

When you start learning Docker, you first launch containers using the `docker run` command. But running multiple services this way quickly becomes messy:

- You need to manually run each container one by one
- You must correctly set ports, environment variables, volumes, and links
- You must remember the exact configuration each time

Docker Compose solves all of this.

With Compose, you describe your entire application stack in a single YAML file—typically named `docker-compose.yaml`—and then start everything with:

docker compose up

All configuration lives in one place, making the setup more maintainable, readable, and reproducible.

🧩 Example: Docker's Sample Voting Application

To understand how Compose works, let's look at Docker's famous sample voting application. It includes:

- **Voting App (Python):** Frontend where users vote between *cats* and *dogs*
- **Redis:** In-memory store holding incoming votes
- **Worker (DotNet):** Processes votes and updates the database
- **PostgreSQL:** Stores vote counts

- **Result App (Node.js):** Displays the results

Data Flow

1. User votes → Voting app writes to **Redis**
2. **Worker** reads from Redis → updates **PostgreSQL**
3. **Result app** fetches vote totals from PostgreSQL → displays results

This architecture combines multiple frameworks and technologies, which makes it a perfect example for Docker Compose.

🛑 The Problem With Only Using `docker run`

You *could* start every component manually:

- `docker run redis`
- `docker run postgres`
- `docker run voting-app`
- `docker run result-app`
- `docker run worker`

But the containers won't know how to connect to each other. You'd have to:

- set container names
- create links (deprecated)
- expose ports
- manage order of startup

It’s too much effort and prone to errors.

🟦 Enter Docker Compose

A Compose file lets you define all containers as **services**. For example:

```
services:
  redis:
    image: redis

  db:
    image: postgres

  vote:
    build: ./vote
    ports:
      - "5000:80"
```

Instead of running five commands and remembering which service needs what configuration, you run **one command**.

Compose also supports:

- dependency ordering via `depends_on`
- automatic networking
- multiple networks
- building images directly

🌐 Docker Compose and Networking
Compose makes networking extremely simple. By default:

Each stack gets its own bridge network

Each service can reach others using the service name as the hostname

Example: the voting app connects to Redis using:

redis:6379

No special links or host edits needed.

Custom Networks
You can create multiple networks to isolate traffic:

```
networks:

  front_end:

  back_end:


services:

  vote:

    networks:

      - front_end

      - back_end
```

Useful for:

separating internal and external traffic

applying different network policies🗒️ Docker Compose File Versions Explained
You'll encounter different Compose file versions in the wild. Here's how they evolved.

Version 1 (legacy)
No version: line at the top

No services: block

No networking features

Uses deprecated links:

Version 2
Introduced version: '2'

Introduced services: section

Auto-created a dedicated network for the app

No need for links: anymore

Added depends_on for startup ordering

Version 3
Still uses version: and services:

Designed for Docker Swarm (stacks)

Some options removed/changed to better support orchestration

Compose v3 remains widely used and is compatible with modern Compose implementations.

🌐 Docker Compose and Networking

Compose makes networking extremely simple. By default:

Each stack gets its own bridge network

Each service can reach others using the service name as the hostname

Example: the voting app connects to Redis using:

redis:6379

No special links or host edits needed.

Custom Networks
You can create multiple networks to isolate traffic:

```
networks:
  front_end:
  back_end:

services:
  vote:
    networks:
      - front_end
      - back_end
```

Useful for:

separating internal and external traffic

applying different network policies

🏛️ Docker Registry Basics

When you run:

docker run nginx

Docker pulls the image from **Docker Hub**, the default registry.

Image name format:

<registry>/<username-or-org>/<image>:<tag>

Examples:

- `nginx` → resolves to `docker.io/library/nginx`
- `myserver:5000/myapp` → custom private registry

⚙️ Docker Engine Architecture

Installing Docker on Linux gives you:

1. **Docker Daemon** – core service managing containers & images
2. **Docker REST API** – interface daemon exposes
3. **Docker CLI** – command-line tool that talks to the daemon

You can even run the CLI from a different machine:

docker -H 10.123.2.1:2375 ps

🧱 How Docker Provides Isolation

Containers are *not* virtual machines. Docker relies on Linux kernel features:

1. Namespaces (isolation)

They isolate:

- process IDs

- networking
- filesystem mounts
- IPC
- UTS / hostname

This allows processes inside containers to think they have process ID 1, even though the actual PID on the host is different.

2. Control Groups (cgroups) (resource limits)

Docker uses cgroups to limit:

- CPU
- memory
- IO

Example:

docker run --cpus="0.5" myimage

Limits the container to 50% of a CPU.

Kubernetes (Modern Industry Standard)

Kubernetes is a highly scalable container-orchestration platform used by companies like Google, Netflix, and Airbnb.

Key Concepts

- **Cluster** – a group of machines (nodes)
- **Nodes** – machines that run your containers
- **Control Plane** – brains of the cluster

- **Pods** – the smallest deployable unit (containers grouped together)
- **Deployments** – manage replica sets and updates

Control Plane Components

- **API Server** – the front door; all commands go through it
- **etcd** – stores cluster state
- **Scheduler** – decides where containers (pods) run
- **Controller Manager** – ensures desired state
- **Kubelet** – runs on every node to manage containers

What Kubernetes Provides

- Horizontal scaling (automatic)
- Rolling updates & rollbacks
- Load balancing via Services
- Resilience & self-healing
- Multi-environment consistency
- Support for public, private, and hybrid clouds

Basic Commands

- kubectl get pods – List all pods in the current namespace
- kubectl get pods -n <namespace> – List pods in a specific namespace
- kubectl describe pod <pod_name> – Show detailed info about a pod
- kubectl logs <pod_name> – View logs of a pod
- kubectl logs -f <pod_name> – Stream logs in real-time
- kubectl exec -it <pod_name> -n <namespace> -- /bin/bash – Execute a command inside a running pod
- kubectl delete pod <pod_name> – Delete a pod
- kubectl apply -f <file.yaml> – Create or update resources from a YAML file
- kubectl get nodes – List all nodes in the cluster

- kubectl get services – List all services in the current namespace

This is the Github repository link which has a basic docker project - https://github.com/shreeja0/DockerPractice