

Docker For Beginners

Visual handwritten study guide

Understand

Clear, compact explanations

Visualize

Diagrams that show the moving parts

Practice

Readable commands and code

Remember

Highlighted revision points

53 topics - each topic begins on a fresh page

A digital asset by Sooraj | soorajdev.cloud

Course Map

Scan the complete learning path before you begin.

Part 1 - The Big Picture

1	Course Introduction	2	The Problem Containers Solve
3	Virtual Machines vs Containers	4	Under the Hood: Namespaces, cgroups and Layers
5	Docker Architecture	6	Installing Docker
7	Your First Container		

Part 2 - Images

8	What Is an Image?	9	Working with Images
10	Docker Hub and Registries	11	Layers, Caching and Image Internals

Part 3 - Containers

12	The Container Lifecycle	13	docker run: Flags That Matter
14	Inspecting and Interacting with Containers	15	Running a Web Server
16	Environment Variables, Config and Secrets	17	Resource Limits and Health
18	Cleaning Up		

Part 4 - Building Images

19	Dockerfile Basics	20	Dockerfile Instruction Reference
21	CMD vs ENTRYPOINT	22	docker build in Depth
23	Build Lab 1: A Python Flask App	24	Build Lab 2: A Node.js Express App
25	Build Lab 3: A Static Site with nginx	26	Smaller, Faster Images
27	Multi-Stage Builds	28	Image Best Practices and Hardening

Part 5 - Data and Storage

29	Why Container Data Disappears	30	Volumes
31	Bind Mounts and tmpfs	32	Backup, Restore and the Data Lab

Part 6 - Networking

33	Networking Basics	34	User-Defined Networks and Container DNS
35	Publishing Ports and Traffic Flow	36	Networking Lab: App Plus Database

Part 7 - Docker Compose

37	Introduction to Docker Compose	38	Compose Services in Depth
39	Compose Networks and Volumes	40	The Compose Workflow
41	Compose Lab: A Full Stack		

Part 8 - Real-World Docker

42	Debugging and Troubleshooting	43	Logging and Monitoring Basics
44	Docker Security Basics	45	Docker in Your Workflow and CI/CD
46	What to Learn Next		

Appendices - Reference and Practice

APPENDIX A	Docker CLI Cheat Sheet	APPENDIX B	Dockerfile and Compose Cheat Sheet
APPENDIX C	Glossary	APPENDIX D	Final Review: 40 Questions
PROJECT 1	Guided Project 1: WordPress with MariaDB	PROJECT 2	Guided Project 2: Reverse Proxy for Two Apps
APPENDIX E	Error Message Dictionary		

Course Introduction

A hands-on beginner course on Docker: what containers are, how to run and build them, store data, connect them, and ship a multi-container app with Compose.

Ever shipped an app that ran perfectly on your laptop and then fell over on the server? Different Python version, a missing library, a config file nobody remembered to copy. **Docker exists to make that problem go away.** It packages your application together with everything it needs to run, so it behaves the same everywhere.

These notes take you from zero to confident. You do not need any Docker experience. You do need to be comfortable opening a terminal and typing a command. Every topic follows the same rhythm: a short idea, a diagram where one helps, commands you can run, and a recap you can revise from.

Learn by doing. Reading about `docker run` teaches you far less than running it, breaking it, and reading the error. Keep a terminal open next to these notes and type every example.

What this course covers

Part	What you will learn	Outcome
1. Big picture	Why containers exist, VMs vs containers, Docker architecture, installing, first container	You can explain Docker and run it
2. Images	Layers, tags, digests, registries, Docker Hub	You can find, pull and inspect images
3. Containers	Lifecycle, run flags, exec, logs, limits, cleanup	You can operate containers confidently
4. Building	Dockerfile, build cache, multi-stage builds, best practices	You can containerise your own apps
5. Data	Volumes, bind mounts, tmpfs, backups	Your data survives container deletion
6. Networking	Bridge networks, DNS, port publishing	Containers talk to each other and the world
7. Compose	compose.yaml, services, networks, volumes, workflow	You run a full stack with one command
8. Real world	Debugging, logging, security, CI/CD, next steps	You are ready for real projects

How to read these notes

- ✓ Topic pages start with a pink summary line, then the explanation, commands and a recap.
- ✓ Code blocks are meant to be typed or pasted. Lines starting with `$` are what you type; other lines are output.
- ✓ Yellow boxes hold ideas worth remembering. Red boxes warn about common mistakes. Green boxes are shortcuts.
- ✓ Labs are guided exercises. Do them in order; later topics reuse earlier results.
- ✓ Review questions close each part. Try to answer before reading the answer.

What you need before you start

- ✓ A computer running Windows 10/11, macOS or a modern Linux distribution, with administrator rights.

- ✓ About 10 GB of free disk space. Images add up quickly.
- ✓ A terminal: Terminal on macOS/Linux, PowerShell or Windows Terminal on Windows.
- ✓ A text editor. VS Code is a good default and has a Docker extension.
- ✓ Basic command-line comfort: `cd`, `ls`, `cat`, editing a file. Nothing more is assumed.

DIAGRAM The path through this course



Each stage uses what the previous one taught. Skipping ahead is possible, but the labs assume the earlier ones.

Conventions used in the commands

```
BASH SNIPPET 7 lines
$ docker version          # a command you type
Client:                   # output you read
Version:                  29.8.2
API version:              1.56

$ docker run --rm alpine echo hello # --rm cleans up afterwards
hello
```

Version numbers on your machine will differ from the ones shown. That is fine: the commands in these notes are the stable core of Docker and have not changed for years.

Watch out: On Linux you may need to prefix commands with `sudo` until you complete the post-install step in Topic 6. On Windows, run Docker commands in PowerShell or WSL 2; the examples use Linux-style shell syntax.

Key takeaways

- Docker packages an app plus its dependencies so it runs the same everywhere.
- You will learn by running commands, not by reading about them.
- The course moves from big picture to images, containers, building, data, networking and Compose.

The Problem Containers Solve

Why deploying software used to be painful, and how containers turned an application plus its environment into a single portable unit.

The bad old days

Before virtualisation, one application meant one physical server. Buying a server took weeks. To be safe, teams over-provisioned: a server sized for peak load that spent most of its life at 5–10% utilisation. Every new application meant new hardware, new rack space, new power and cooling, and new people to look after it.

Worse, the application and the server were welded together. The app depended on a particular operating system version, particular library versions, particular configuration. Moving it to another machine meant recreating all of that by hand.

Dependency hell

Imagine two applications on the same server. App A needs Python 3.8 and a library at version 1.2. App B needs Python 3.12 and the same library at version 2.0. They cannot both be installed system-wide without breaking one of them. This is **dependency hell**, and it is the everyday reason for the sentence every developer has said: *it works on my machine*.

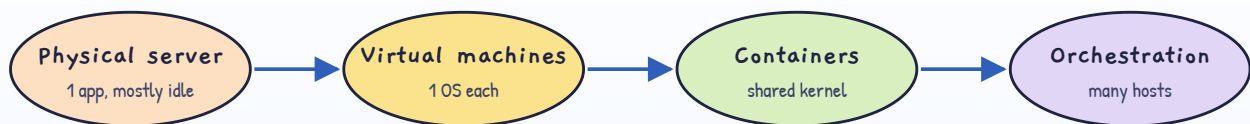
Symptom	Typical cause	What containers change
Works on my laptop, fails on server	Different OS, library or runtime versions	The same image runs in both places
Two apps need conflicting versions	Shared system-wide dependencies	Each app has its own isolated filesystem
New teammate takes days to set up	Long, error-prone install instructions	<code>docker run</code> or <code>docker compose up</code>
Deployments are scary and manual	Hand-edited servers drift over time	Immutable images, rebuilt not patched
Scaling means building another server	Environment is not reproducible	Start more copies of the same image

Enter virtual machines

Virtualisation was the first big fix. A hypervisor lets one physical server host many **virtual machines (VMs)**, each with its own operating system. Hardware use went up and isolation improved. But every VM carries a full operating system: gigabytes of disk, its own RAM reservation, and a boot time measured in minutes. Running ten small apps meant running ten operating systems.

Enter containers

A **container** keeps the isolation idea but drops the duplicated operating system. All containers on a host share the host's kernel; each gets its own isolated view of the filesystem, processes and network. They start in a fraction of a second and an image may be tens of megabytes instead of gigabytes.

DIAGRAM**From one app per server to many containers per host**

Each step packs more applications onto the same hardware with less overhead.

The shipping container analogy

Before standard steel shipping containers, cargo was loaded loose: sacks, barrels and crates, each handled differently by each port. The standard container did not change the cargo. It changed the **interface**: every crane, ship, train and truck now handled the same box. Docker does the same for software. Whatever is inside (Python, Java, Node, a database), the outside looks identical, so any host that can run containers can run it.

The key insight is standardisation. A container is a standard package for software. Tooling, cloud platforms and orchestrators all agree on that package, which is why the whole ecosystem grew around it.

What a container actually gives you

- ✓ **Portability** - build once, run on a laptop, a CI server or any cloud.
- ✓ **Isolation** - an app cannot see other containers' files or processes unless you allow it.
- ✓ **Reproducibility** - the same image gives the same result today and next year.
- ✓ **Speed** - starts in under a second, so scaling and testing are cheap.
- ✓ **Density** - far more workloads per server than with VMs.
- ✓ **Clear contracts** - a Dockerfile documents exactly how the app is built and run.

What containers do not solve

Containers are not magic. They do not fix a badly designed application, they are weaker isolation than a VM (a kernel bug affects every container on the host), and stateful software such as databases needs careful handling of data. You still need monitoring, backups and security. We cover each of these as the course progresses.

Key takeaways

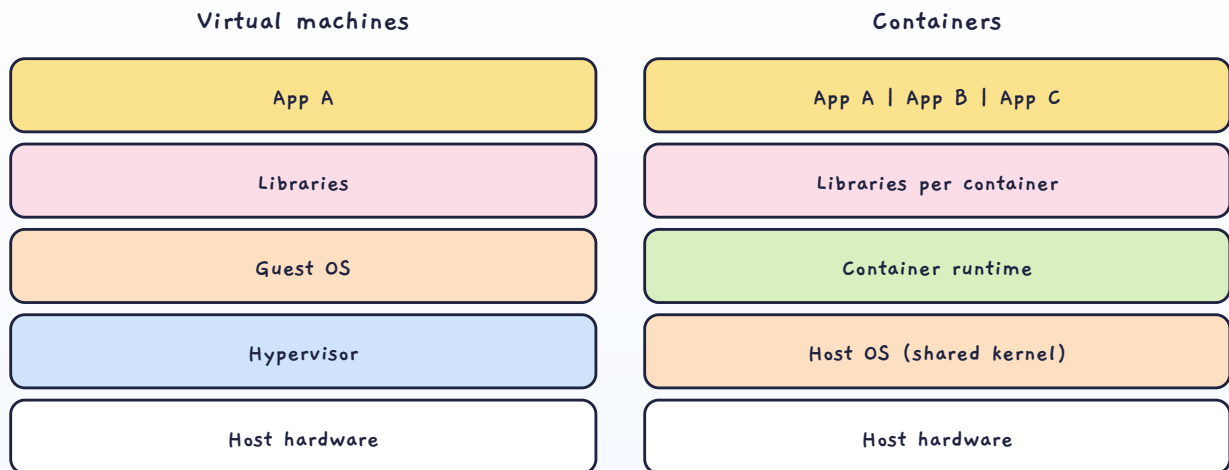
- Apps depend on their environment; moving them used to be slow and fragile.
- VMs solved isolation but duplicate a whole operating system per app.
- Containers share the host kernel and isolate processes, so they are small and fast.
- Docker's big contribution was a standard, easy-to-use package format.

Virtual Machines vs Containers

A side-by-side comparison of how VMs and containers are built, how they isolate workloads, and when to choose each.

Both technologies run isolated workloads on shared hardware, but they virtualise at **different layers**. A VM virtualises the hardware: each VM believes it owns a computer. A container virtualises the operating system: each container believes it owns the machine's userspace.

DIAGRAM Two ways to isolate an application



VMs stack a full guest OS per application. Containers share one kernel and add only the libraries each app needs.

Side-by-side comparison

Aspect	Virtual machine	Container
Isolation boundary	Hardware (hypervisor)	OS kernel features (namespaces, cgroups)
Includes an OS?	Yes, a full guest OS	No, shares the host kernel
Typical size	Gigabytes	Megabytes to a few hundred MB
Start-up time	Tens of seconds to minutes	Milliseconds to seconds
Density per host	Tens	Hundreds to thousands
Security isolation	Stronger (separate kernels)	Weaker (one shared kernel) but hardenable
Best for	Different OSES, strong tenant isolation, legacy apps	Microservices, CI, fast scaling, dev environments
Portability	Image tied to hypervisor format	OCI standard image runs anywhere

Why containers are lighter

When a VM boots it loads a kernel, starts system services, and reserves memory for its own operating system before your app even starts. A container is just a **normal Linux process** with restrictions placed around it. There is no boot. Starting a container means starting that process inside its sandbox, which is why it takes milliseconds.