

THE COMPLETE PRACTICAL GUIDE

Docker from Zero to Deployment

Containers, Compose, and Production

Dockerfiles · Volumes · Networking · Compose · Security · CI/CD · Reverse Proxies · Real-World Deployment

28 CHAPTERS · 4 APPENDICES · HANDS-ON

THENEXBOOKS

Docker from Zero to Deployment

The Complete Practical Guide to Containers, Compose, and Production

From Your First Container to a Live, Secure Deployment

First Edition

THENEXBOOKS

Docker from Zero to Deployment
The Complete Practical Guide to Containers, Compose, and Production

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Docker, cloud platforms, and related tools change frequently; always consult the official documentation for the versions you use.

Docker is a trademark of Docker, Inc. Kubernetes is a trademark of The Linux Foundation. Nginx, PostgreSQL, Redis, GitHub, AWS, Google Cloud, Microsoft Azure, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with or endorsed by Docker, Inc.

Riverbend Recipes, the running example in this book, is fictional. Any resemblance to real companies or products is coincidental.

Preface

"It works on my machine" used to be the end of a conversation. Your laptop had one version of Python, the test server had another, production had a third, and a missing system library could take an afternoon to track down. Containers changed that. With Docker, an application and everything it needs to run travel together as a single image that behaves the same on a laptop, in CI, and on a production server.

Docker is now a basic skill for developers, not just operations teams. Many job descriptions expect it, many open-source projects ship a Dockerfile, and many modern deployment platforms run containers. Yet many people learn Docker in fragments — a command copied from a README here, a Compose file borrowed from a colleague there — and never build the mental model that makes containers predictable.

This book starts from zero and goes all the way to deployment. You will learn what containers really are, how images are built and cached, how data, networking, and configuration work, how to run multi-container applications with Docker Compose, how to secure and ship images through a CI/CD pipeline, and how to deploy to a real server and to managed container platforms. Every chapter builds on one realistic application, so by the end you will have taken a complete system from your laptop to production.

Who This Book Is For

- **Developers** who want to containerize their applications and stop fighting environment differences.
- **Students and career changers** who need a solid, job-ready foundation in containers.
- **Junior DevOps engineers and system administrators** who will build and operate container-based systems.
- **Anyone who has used Docker by copying commands** and now wants to truly understand it.

You only need basic command-line skills and some programming experience. No prior Docker knowledge is assumed.

What You Will Learn

- How containers work — images, layers, namespaces, and cgroups — explained without jargon.
- How to write clean, fast, small, and secure Dockerfiles for Python, Node.js, Java, and Go.
- Volumes, networking, configuration, secrets, logging, health checks, and restart policies.
- Docker Compose for development and multi-environment setups, including hot reload and testing.
- Image security, registries and tagging, BuildKit and multi-platform builds, and CI/CD with GitHub Actions.

- Running containers in production: resource limits, monitoring, troubleshooting, reverse proxies, TLS, and zero-downtime deploys.
- Deploying to a single server and to managed container platforms, and when to move on to Kubernetes.

How This Book Is Organized

Part I — Container Foundations (Chapters 1–5) builds the mental model. **Part II — Building Images** (Chapters 6–10) teaches Dockerfiles from first principles to production quality. **Part III — Data, Networking, and Configuration** (Chapters 11–14) covers what every real application needs. **Part IV — Multi-Container Applications with Docker Compose** (Chapters 15–19) assembles the full Riverbend Recipes stack. **Part V — Production Readiness and Security** (Chapters 20–25) makes it safe and shippable. **Part VI — Deployment** (Chapters 26–28) puts it live. The appendices provide a CLI cheat sheet, templates, a deployment checklist, and a glossary. Each chapter ends with a **Key Takeaways** box.

The Running Example: Riverbend Recipes

Throughout the book we build and ship **Riverbend Recipes**, a fictional recipe-sharing web application. It starts as a single container and grows into a complete system: a web front end, a REST API, a background worker that creates image thumbnails, a PostgreSQL database, a Redis cache and queue, and a reverse proxy with TLS in production.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause data loss, security problems, or outages.

Commands you type are shown with a `$` prompt; lines without it are output. Your output — especially IDs, sizes, and timings — will differ from what is printed here.

Contents

Preface

PART I — CONTAINER FOUNDATIONS

1 Why Containers? The Problems Docker Solves

2 Installing Docker and Running Your First Containers

3 Images, Containers, and Registries: The Core Mental Model

4 Working with Containers Day to Day

5 How Containers Work Under the Hood: Namespaces, cgroups, and Layered Filesystems

PART II — BUILDING IMAGES

6 Writing Your First Dockerfile

7 Dockerfile Instructions in Depth

8 The Build Cache, .dockerignore, and Fast Builds

9 Multi-Stage Builds and Small, Efficient Images

10 Containerizing Real Applications: Python, Node.js, Java, and Go

PART III — DATA, NETWORKING, AND CONFIGURATION

11 Volumes, Bind Mounts, and Persistent Data

12 Container Networking: Bridge Networks, Ports, DNS, and Service Discovery

13 Configuration, Environment Variables, and Secrets

14 Logging, Health Checks, and Restart Policies

PART IV — MULTI-CONTAINER APPLICATIONS WITH DOCKER COMPOSE

15 Docker Compose Fundamentals

16 Building the Riverbend Recipes Stack with Compose

17 Development Workflows: Hot Reload, Debugging, and Compose Watch

18 Testing with Containers: Integration Tests, Ephemeral Databases, and Testcontainers

19 Compose Profiles, Overrides, and Multiple Environments

PART V — PRODUCTION READINESS AND SECURITY

20 Image Security: Non-Root Users, Minimal Bases, Scanning, and SBOMs

21 Registries, Tagging, and Versioning Strategies

22 BuildKit, Buildx, and Multi-Platform Images

23 CI/CD for Containers with GitHub Actions

24 Running Containers in Production: Resource Limits, Monitoring, and Troubleshooting

25 Reverse Proxies, TLS, and Zero-Downtime Deployments

PART VI — DEPLOYMENT

26 Deploying to a Single Server (VPS) with Docker Compose

27 Deploying to Managed Container Platforms

28 From Docker to Kubernetes: Next Steps and the Complete Journey

APPENDICES

A Docker CLI Cheat Sheet

B Dockerfile and Compose Templates

C Production Deployment Checklist

D Glossary

PART I

Container Foundations

What containers are, why they exist, and how to work with them confidently — from installing Docker to understanding the Linux features that make it all possible.

1 · Why Containers?
2 · Installing Docker and Your First Containers
3 · Images, Containers, and Registries
4 · Working with Containers Day to Day
5 · How Containers Work Under the Hood

CHAPTER 1 · PART I — CONTAINER FOUNDATIONS

Why Containers? The Problems Docker Solves

Before we type a single Docker command, we need to understand the pain that containers were invented to relieve. In this chapter you will learn what a container is, how it differs from a virtual machine, and why a small recipe-sharing application called Riverbend Recipes is about to become much easier to build, share, and deploy.

A Familiar Story: "It Works on My Machine"

Almost every developer has lived through some version of the following story. You write an application on your laptop. It runs perfectly. You hand it to a colleague, push it to a test server, or deploy it to production, and it immediately breaks. The error messages are confusing: a library cannot be found, a function behaves differently, a configuration file is in the wrong place, or the program crashes on startup with a message about an incompatible version of something you have never heard of.

After an afternoon of investigation, you discover the cause. Your laptop has Python 3.12, but the server has an older version. Or your laptop has a system library that was installed years ago for another project and quietly satisfied a dependency that nobody wrote down. Or the server uses a different operating system release with a different default locale, different file paths, or different permissions. The code was never the whole story. The code only worked because of everything *around* it: the runtime, the libraries, the operating system packages, the environment variables, and the file layout.

This is the heart of the problem Docker solves. Software does not run in a vacuum. It runs inside an **environment**, and traditionally that environment has been built by hand, differently on every machine, and documented poorly if at all. The famous phrase "it works on my machine" is really a confession: "my machine has an environment that yours does not."

The hidden cost of environment drift

Environment differences cause more than the occasional failed deployment. They create a steady, grinding cost across a whole team:

- **Slow onboarding.** A new team member spends their first days following a long setup document, installing databases, language runtimes, and command-line tools, and discovering which steps are out of date.
- **Fragile servers.** Production servers are configured by hand over months or years. Nobody is quite sure what is installed on them, and nobody wants to touch them in case something breaks.

- **Conflicting projects.** One project needs PostgreSQL 15 and another needs PostgreSQL 16. One needs Node.js 18 and another needs Node.js 22. Installing both side by side on the same machine is awkward and error-prone.
- **Unreliable testing.** Tests pass on a developer's laptop and fail in the continuous integration system, or the other way round, because the two environments are subtly different.
- **Risky upgrades.** Upgrading a shared library on a server might fix one application and break three others that also depended on it.

None of these problems are caused by bad programmers. They are caused by the fact that the environment is treated as something separate from the application, something that lives on the machine rather than traveling with the code. Containers change that assumption.

Meet Riverbend Recipes

Throughout this book we will build, containerize, and deploy a single realistic application so that every concept has a concrete home. The application is called **Riverbend Recipes**, a fictional recipe-sharing website where people can publish recipes, upload photos of their dishes, and search for something to cook tonight. It is small enough to understand completely but large enough to have the same moving parts as many real production systems.

By the end of the book, Riverbend Recipes will consist of the following services:

Service	Technology	Purpose
`web`	Nginx serving static files	The front end people see in the browser
`api`	Python with FastAPI	The REST API that reads and writes recipes
`worker`	Node.js	Creates thumbnails of recipe photos from a queue
`db`	PostgreSQL	Stores recipes, users, and comments
`cache`	Redis	Caching and the job queue for the worker
`proxy`	Nginx, Caddy, or Traefik	Routes traffic and terminates TLS in production

Look at that list from the point of view of a new developer joining the Riverbend team. Without containers, they would need to install Python and the right set of Python packages, Node.js and its packages, a PostgreSQL server, a Redis server, and Nginx. They would need the correct versions of each, configured to talk to one another, with the right ports, users, passwords, and startup order. They would need to repeat much of this on the test server and again on the production server. Each of those machines might run a different operating system.

Here is what a typical setup document for such a project might look like before containers. You do not need to run any of this; it is here to show the problem.

SETUP.txt (the old way)

```
1. Install Python 3.12 (NOT 3.13, the image library breaks)
2. python -m venv .venv && source .venv/bin/activate
3. pip install -r api/requirements.txt
   - On macOS you may need: brew install libjpeg zlib
   - On Ubuntu you may need: sudo apt install libpq-dev build-essential
4. Install Node.js 20 and run: cd worker && npm install
5. Install PostgreSQL 16, create user "riverbend", create database "recipes"
6. Install Redis and make sure it listens on port 6379
7. Install Nginx and copy web/nginx.conf to the right folder for your OS
8. Export DATABASE_URL, REDIS_URL, and SECRET_KEY in your shell profile
9. Start everything in this order: db, cache, api, worker, web
10. If step 3 fails, ask someone on the team
```

Every line in that document is a place where something can go wrong. Step 10 says it all. Later in the book, the same setup will shrink to a single command, `docker compose up`, that works the same way on every machine that has Docker installed. Getting from here to there is the journey of this book.

What Is a Container?

A **container** is a running process, or a small group of processes, that is isolated from the rest of the system and packaged together with everything it needs to run: its code, its runtime, its libraries, its system tools, and its default configuration. From the inside, a container looks a lot like its own small computer. It has its own file system, its own network interfaces, its own list of processes, and its own hostname. From the outside, on the host machine, it is just an ordinary process with some extra restrictions applied by the operating system kernel.

That last point is important and we will return to it often. A container is *not* a tiny virtual machine. It does not boot its own operating system kernel. All containers on a Linux host share that host's kernel. What makes them feel separate is a set of Linux kernel features that control what a process can *see* and how much it can *use*. We will explore those features in detail in Chapter 5; for now, here is the short version:

- **Namespaces** control what a process can see. A process in its own PID namespace sees only its own processes. A process in its own network namespace has its own network interfaces and ports. There are namespaces for mounts, hostnames, inter-process communication, and user IDs as well.
- **Control groups (cgroups)** control how much a process can use: how much memory, how much CPU time, how many processes, and so on.
- **Layered file systems** such as OverlayFS let many containers share the same read-only base files while each container gets its own thin writable layer on top.

Put those three ideas together and you get something that behaves like an isolated machine, starts about as quickly as a normal program, and uses far fewer resources than a full virtual machine.

Images: the recipe for a container

If a container is a running thing, what is it created from? The answer is an **image**. An image is a read-only package that contains a file system and some metadata: which program to run by default, which environment variables to set, which ports the program expects to use, and so on. You can think of an image as a template or a snapshot, and a container as a live instance created from that template.

Since our running example is a recipe website, the kitchen analogy fits nicely. An image is like a written recipe card: it is fixed, it can be copied and shared, and it describes exactly how to produce a dish. A container is the dish you actually cook from that card. You can cook the same recipe many times, and each dish is independent; eating one does not affect the others. You can also throw a dish away and cook a fresh one at any moment, because the card never changes.

Images are built from a text file called a **Dockerfile**, which lists the steps needed to assemble the image. Here is a tiny preview of what the Riverbend Recipes API's Dockerfile will look like once we reach Part II of the book. Do not worry about understanding every line yet.

api/Dockerfile (preview)

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Read it top to bottom and it almost explains itself. Start from an official image that already contains Python 3.12. Work in a folder called `/app`. Copy the list of dependencies and install them. Copy the rest of the application code. Document that the program listens on port 8000. Finally, say which command to run when a container starts. Every dependency that the old setup document described in prose is now captured precisely, in a file that lives in the repository alongside the code.

Registries: sharing images

Once an image is built, you need a way to move it between machines. A **registry** is a server that stores images and lets you push them up and pull them down, much as a Git hosting service stores repositories. Docker Hub is the default public registry, and it hosts official images for popular software such as `nginx`, `postgres`, `redis`, `python`, and `node`. Companies often run private registries too. In this book, our fictional private registry is `registry.example.com`, and our images will have names such as `riverbend/api:1.4.0`.

Containers Versus Virtual Machines

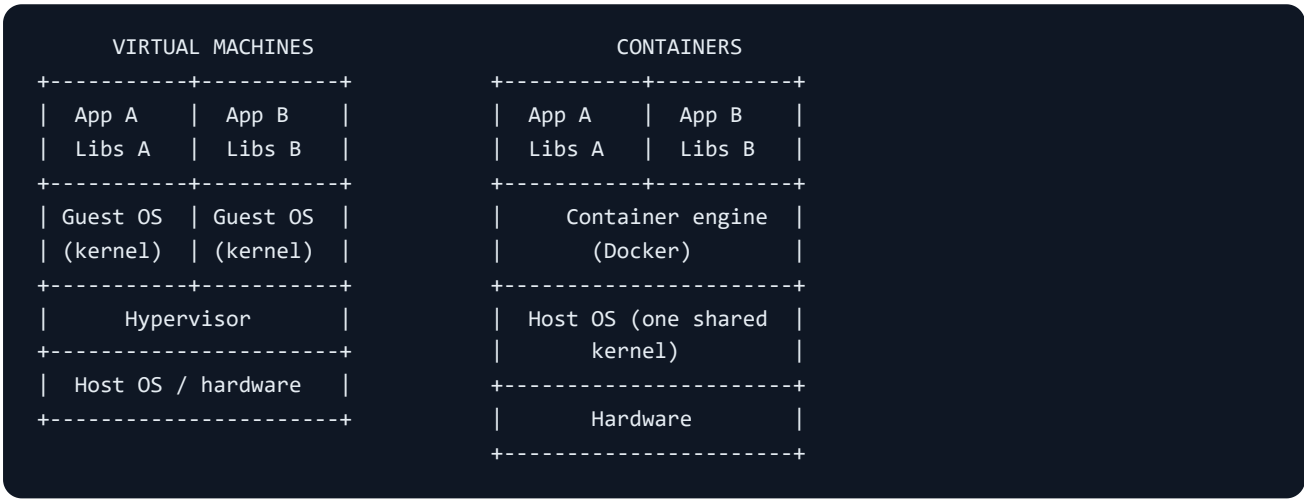
If you have used virtual machines (VMs) before, you might be thinking: "I can already get an isolated environment. I just create a VM." That is true, and VMs remain extremely useful. But containers and VMs isolate things at different levels, and that difference has big practical consequences.

A **virtual machine** emulates an entire computer. A piece of software called a *hypervisor* presents virtual hardware to each VM, and each VM boots its own complete operating system, including its own kernel. On top of that guest operating system you install your application and its dependencies.

A **container**, by contrast, shares the host's kernel. There is no virtual hardware and no second operating system to boot. The container holds only the user-space files your application needs: libraries, runtimes, and tools. The kernel isolates it from other processes using namespaces and cgroups.

The following diagram shows the two stacks side by side.

Diagram: virtual machines vs containers



Because containers skip the guest operating system, they tend to have several practical advantages:

- **They start quickly.** Starting a container is essentially starting a process. There is no operating system boot sequence.
- **They are lightweight.** A container does not need memory reserved for a whole guest operating system, so you can usually run many more containers than VMs on the same machine.
- **Images are smaller than VM disk images.** An image carries only the files the application needs, not an entire installed operating system with its kernel and drivers.
- **They are easy to throw away.** Because they are cheap to create, containers encourage a style where you replace rather than repair.

VMs, however, have genuine strengths of their own. Each VM has its own kernel, which gives a stronger isolation boundary. A VM can run a completely different operating system from the host, such as Windows on a Linux server. And for some workloads, the full-machine model is simply a better fit.

> **WARNING**
>
> Containers are not a strong security boundary by default. Because every container on a host shares the same kernel, a serious kernel vulnerability or a badly configured container can affect the host and other containers. Throughout this book we will follow safe habits: running as a non-root user, never baking secrets into images, and avoiding dangerous options such as `--privileged` or mounting the Docker socket into a container. Chapter 20 covers image security in depth.

In practice, containers and VMs are often used together: cloud servers are usually VMs running Docker, and Docker Desktop on macOS and Windows runs a small Linux VM behind the scenes. Containers package applications; VMs virtualize machines.

Aspect	Virtual machine	Container
What is isolated	A whole computer	A process and its view of the system
Kernel	Each VM has its own	Shared with the host
Startup	Boots an operating system	Starts a process
Typical package	Full disk image	Layered image of app files
Isolation strength	Stronger by default	Weaker; needs careful configuration
Different OS than host	Yes	Needs a compatible kernel

A Short History, Without the Hype

The ideas behind containers are older than Docker. Unix `chroot`, FreeBSD jails, Solaris zones, and Linux tools such as LXC all isolated processes long before Docker appeared in 2013. What Docker added was a **developer-friendly workflow**: the Dockerfile, a layered image format, a public registry of ready-made images, and a simple command-line tool. Later, the Open Container Initiative (OCI) standardized image and runtime formats, which is why images built with Docker also run on Kubernetes and on managed cloud platforms. For practical purposes in this book, a Docker image is an OCI image.

The Problems Docker Solves

Let us now be concrete. Here are the specific problems from the start of this chapter and how Docker addresses each one. Keep Riverbend Recipes in mind as you read.

1. Consistent environments everywhere

The image for the Riverbend API contains a specific Python version and a specific set of installed packages. When you run that image on your laptop, on a colleague's laptop, in your CI system, and on the production server, the code runs against the same files. The host still matters in some ways (it

provides the kernel, the CPU architecture, and resources), but the enormous source of drift that used to come from hand-installed dependencies disappears.

2. Fast, repeatable onboarding

A new developer on the Riverbend team installs Docker, clones the repository, and runs one command. Here is a preview of what that will look like once we build the full stack in Chapter 16:

Terminal

```
$ git clone https://github.com/example/riverbend-recipes.git
$ cd riverbend-recipes
$ docker compose up -d
[+] Running 6/6
 ✓ Network riverbend-recipes_default     Created
 ✓ Container riverbend-recipes-db-1      Started
 ✓ Container riverbend-recipes-cache-1   Started
 ✓ Container riverbend-recipes-api-1     Started
 ✓ Container riverbend-recipes-worker-1  Started
 ✓ Container riverbend-recipes-web-1     Started
```

The first command downloads the source code. The second moves into the project folder. The third, `docker compose up -d`, reads a file describing all the services, downloads any images that are missing, builds the ones that come from the project's own code, creates a private network for them, and starts every container in the background (that is what `-d`, for "detached", means). The exact output on your machine will differ, but the idea is the same: the setup document has become executable.

3. Isolation between projects

Suppose you work on Riverbend Recipes, which uses PostgreSQL 16, and on an older internal tool that still needs PostgreSQL 14. Without containers, running both database versions on one laptop is a chore. With containers, each project runs its own database container from its own image, with its own data, on its own port or private network. They never see each other. When you are done with a project, you remove its containers and nothing is left installed on your system.

Terminal

```
$ docker run -d --name old-tool-db -e POSTGRES_PASSWORD=devonly \
    -p 5414:5432 postgres:14
$ docker run -d --name riverbend-db -e POSTGRES_PASSWORD=devonly \
    -p 5416:5432 postgres:16
```

These two commands start two separate PostgreSQL servers of different major versions side by side. The `-p` option maps a port on your machine to the port inside the container, so one database is reachable on port 5414 and the other on port 5416. We will explain every option of `docker run` carefully in Chapters 2 and 4; for now, simply notice how little effort it takes.

4. Reliable testing and CI

Because the same image runs in development and in continuous integration, tests become far more trustworthy. If the tests pass against the `riverbend/api:1.4.0` image in CI, you know they passed against the exact bits that will be deployed. Containers also make it easy to spin up a throwaway database for each test run and delete it afterwards, which we will do in Chapter 18.

5. Simpler, safer deployments

Deploying without containers often means logging into a server, pulling new code, installing new dependencies, and restarting services, all while hoping nothing conflicts. Deploying with containers means pulling a new, already-built image and replacing the old container with a new one. If something goes wrong, rolling back is as simple as running the previous image tag again. The image you tested is the image you run.

Terminal (preview of a deployment)

```
$ docker pull registry.example.com/riverbend/api:1.4.1
$ docker compose up -d api
# Something is wrong? Point the service back at 1.4.0 and run it again.
```

6. Portability across infrastructure

Because container images follow open standards, the same image can run on a laptop, a single virtual server, a managed container platform such as AWS ECS with Fargate, Google Cloud Run, or Azure Container Apps, or a Kubernetes cluster. You are not locked into one hosting approach. Riverbend Recipes will travel through several of these in Part VI of the book, and the images themselves will not need to change.

> **NOTE**
>
> Portability has limits. An image built for one CPU architecture, such as `amd64` (typical Intel and AMD machines), will not run natively on another, such as `arm64` (Apple silicon Macs and many cloud servers), unless it was built for both. Similarly, Linux container images need a Linux kernel. Chapter 22 shows how to build multi-platform images that cover more than one architecture.

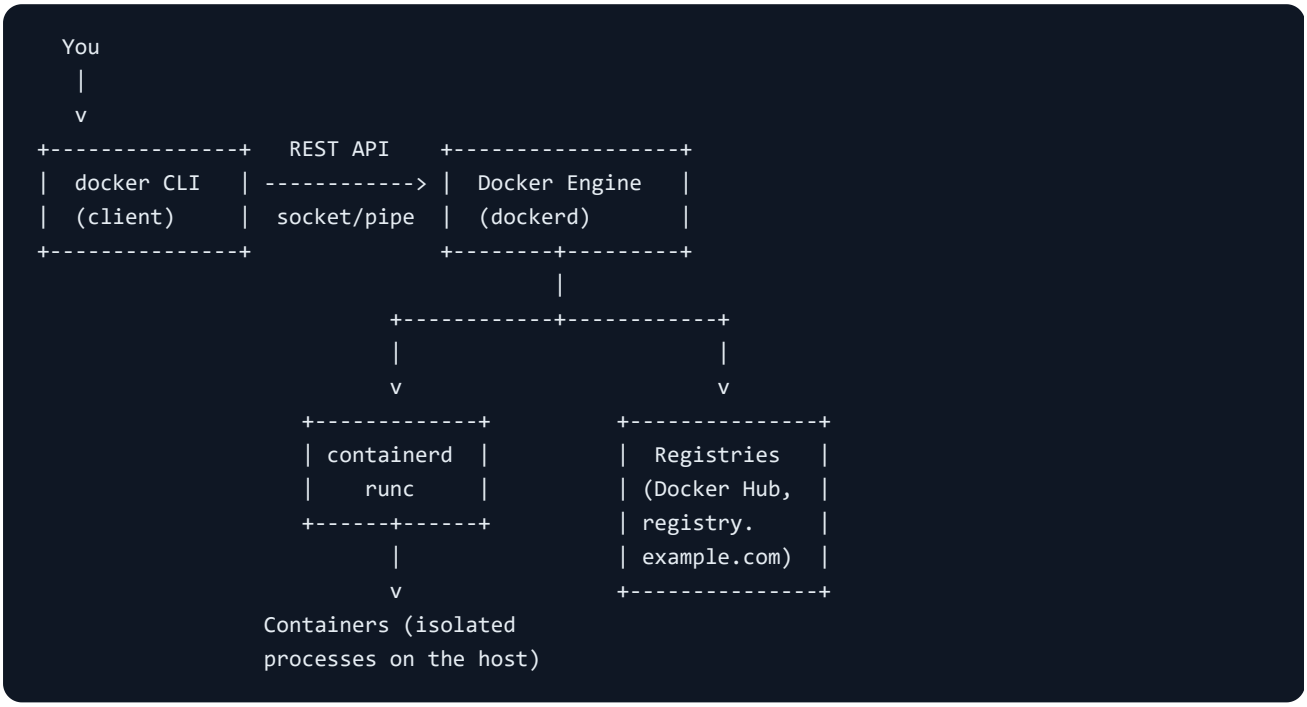
> **TIP**
>
> A helpful habit from day one is to treat containers as **disposable**. Never make an important change by logging into a running container and editing files by hand. Instead, change the Dockerfile or configuration, rebuild the image, and start a new container. This keeps your environment reproducible, which is the whole point of using containers.

How the Docker Pieces Fit Together

When people say "Docker", they usually mean a collection of cooperating components. Understanding how they relate will make error messages much less mysterious later.

The **Docker CLI** is the `docker` program you type commands into. It does not run containers itself. Instead, it sends requests to the **Docker Engine**, also called the Docker daemon (`dockerd`), which is a background service that manages images, containers, networks, and volumes. The CLI and the daemon talk over an API, usually through a local socket file on Linux or a named pipe on Windows. The daemon in turn relies on lower-level components, such as `containerd` and a runtime like `runc`, to actually create the isolated processes using kernel features.

Diagram: the Docker architecture



This client-server design has a useful consequence: the CLI can talk to a Docker Engine on a different machine. You will see this idea again when we discuss Docker contexts and remote deployment. It also explains a common beginner error: if the daemon is not running, the CLI prints a message saying it cannot connect to the Docker daemon. The CLI is fine; there is simply nothing listening on the other end. Chapter 2 shows how to check for this.

Docker Compose sits on top of the CLI and Engine. It reads a YAML file, usually named `compose.yaml`, that describes several services, and creates all the necessary containers, networks, and volumes for you. In recent versions it is built into the Docker CLI as a plugin and is invoked as `docker compose` (with a space). You may still see older tutorials that use a separate `docker-compose` command with a hyphen; that standalone version 1 tool is deprecated, and this book uses `docker compose` throughout.

Here is a short glimpse of the Compose file that Riverbend Recipes will grow into. Again, you are not expected to understand it yet. Notice how it reads almost like the old setup document, but in a precise, machine-readable form.

compose.yaml (early preview)

```
services:
  web:
    image: nginx:1.27
    ports:
      - "8080:80"
    volumes:
      - ./web/public:/usr/share/nginx/html:ro

  api:
    build: ./api
    environment:
      DATABASE_URL: postgresql://riverbend:devonly@db:5432/recipes
      REDIS_URL: redis://cache:6379/0
    depends_on:
      - db
      - cache

  db:
    image: postgres:16
    environment:
      POSTGRES_USER: riverbend
      POSTGRES_PASSWORD: devonly
      POSTGRES_DB: recipes
    volumes:
      - db-data:/var/lib/postgresql/data

  cache:
    image: redis:7

volumes:
  db-data:
```

Each entry under `services` describes one container. The `web` service uses the official Nginx image and serves files from the project folder. The `api` service is built from our own code. The `db` and `cache` services use official PostgreSQL and Redis images. Notice that the API reaches the database simply by the hostname `db`; Compose sets up networking so services can find each other by name, which we explore in Chapter 12. Also notice there is no top-level `version:` key. Older Compose files often started with one, but recent versions of Compose treat it as obsolete and ignore it.

> **WARNING**
>
> The password `devonly` in these examples is acceptable only for a local development database on your own machine. Never commit real passwords to a repository and never place secrets inside an image. Chapter 13 shows proper ways to handle configuration and secrets.

A Word on Docker Desktop and Licensing

On Windows and macOS, most people use **Docker Desktop**, an application that bundles the Docker Engine, the CLI, Compose, and a graphical dashboard, running inside a lightweight Linux virtual machine. On Linux servers, people usually install **Docker Engine** directly from Docker's package repositories. Docker Desktop requires a paid subscription for some commercial uses, while other uses, such as personal projects and education, may be free. The terms have changed over time, so check Docker's current terms before using Docker Desktop at work. Docker Engine on Linux is open source. Chapter 2 walks through both installation paths.

How This Book Is Organized

The book follows the life of Riverbend Recipes from a pile of source code to a production deployment. Each part builds on the one before it.

1. **Part I — Container Foundations** (Chapters 1 to 5): you install Docker, run existing images such as `nginx`, `redis`, and `postgres`, learn the image-container-registry model, manage the container lifecycle, and peek under the hood.
2. **Part II — Building Images** (Chapters 6 to 10): you write Dockerfiles, learn every major instruction, make builds fast with the cache, shrink images with multi-stage builds, and containerize the Riverbend API and worker, with variants in several languages.
3. **Part III — Data, Networking, and Configuration** (Chapters 11 to 14): you keep data safe with volumes, connect containers on networks, manage configuration and secrets, and add logging, health checks, and restart policies.
4. **Part IV — Multi-Container Applications with Docker Compose** (Chapters 15 to 19): you describe the whole Riverbend stack in a Compose file, set up productive development workflows, test with containers, and manage multiple environments.
5. **Part V — Production Readiness and Security** (Chapters 20 to 25): you harden images, version and publish them, build for multiple platforms, automate with CI/CD, run and monitor containers in production, and put a reverse proxy with TLS in front of everything.
6. **Part VI — Deployment** (Chapters 26 to 28): you deploy Riverbend Recipes to a single server, then to managed container platforms, and finally take a first look at Kubernetes.

The appendices provide a CLI cheat sheet, ready-to-use Dockerfile and Compose templates, a production deployment checklist, and a glossary.

How to get the most from this book

Throughout the book, terminal sessions are shown with a `$` prompt before each command. Do not type the `$` itself. Lines without a prompt are output. Output on your machine will often differ in details such as IDs, timestamps, and version numbers; that is expected. Long commands are split

across lines with a backslash (`\`) at the end of each line, which tells the shell that the command continues. On Windows PowerShell, the line-continuation character is a backtick instead, or you can simply type the whole command on one line.

Terminal (how to read examples)

```
$ docker run --rm hello-world     # you type this (without the $)
Hello from Docker!                # this is output
...                               # "..." means output was shortened
```

> **TIP**
>
> Create a dedicated folder for your practice work, for example `~/docker-practice`, and a separate folder called `riverbend-recipes` for the running example. Keeping experiments apart from the main project makes it easy to delete the experiments later without worrying about losing anything important.

Thinking in Containers

Before we close this chapter, it is worth describing the change in mindset that containers encourage, because it affects how you design applications as much as how you run them.

Build once, run anywhere. Instead of building your application separately on each server, you build one image, test it, and then run that exact image in every environment. Differences between environments are expressed through configuration, such as environment variables, rather than through different builds.

One concern per container. A common and healthy convention is that each container does one main job. Riverbend Recipes does not put the API, the database, and the cache into one giant container; each gets its own. This makes each piece easier to update, scale, restart, and reason about. Chapter 16 shows how Compose ties these pieces together.

Immutable infrastructure. Once an image is built, you do not change it. To make a change, you build a new image with a new tag. Running containers are replaced, not patched. This sounds strict, but it is what makes rollbacks trivial and environments reproducible.

State lives outside the container. Because containers are disposable, anything that must survive, such as database files or uploaded recipe photos, belongs in a volume or an external service, not in the container's own writable layer.