Docker for Beginners

A practical introduction to Docker — from containers and images to multi-container stacks with Compose, including real-world scenarios for web serving, Python development, and Claude API integration.

10 CHAPTERS

osztromok.com

Table of Contents

Chapter 1 — What is Docker and Why Use It?

Chapter 1 — What is Docker and Why Use It?

If you've ever set up a web server, a Python project, or a database and thought *"this took ages — I never want to do this again"*, Docker is for you. Docker lets you package an application and everything it needs to run — the runtime, the libraries, the config — into a single portable unit called a **container**. You can start it in seconds, stop it cleanly, copy it to another machine, and it will behave exactly the same way every time.

In this chapter you'll understand what Docker actually is, how it differs from virtual machines, and get it installed and running your very first container.

1. The Problem Docker Solves

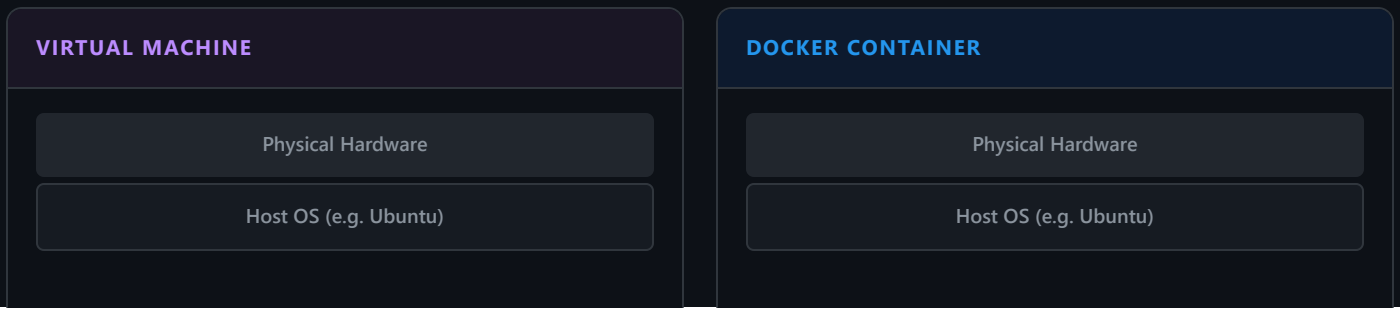
Picture this: you set up Apache on your Raspberry Pi. It works perfectly. A few months later your hosting provider's server has a slightly different OS version, different library versions, different PHP config — and suddenly things break in ways that are hard to debug. This is so common it has a name: **"works on my machine"**.

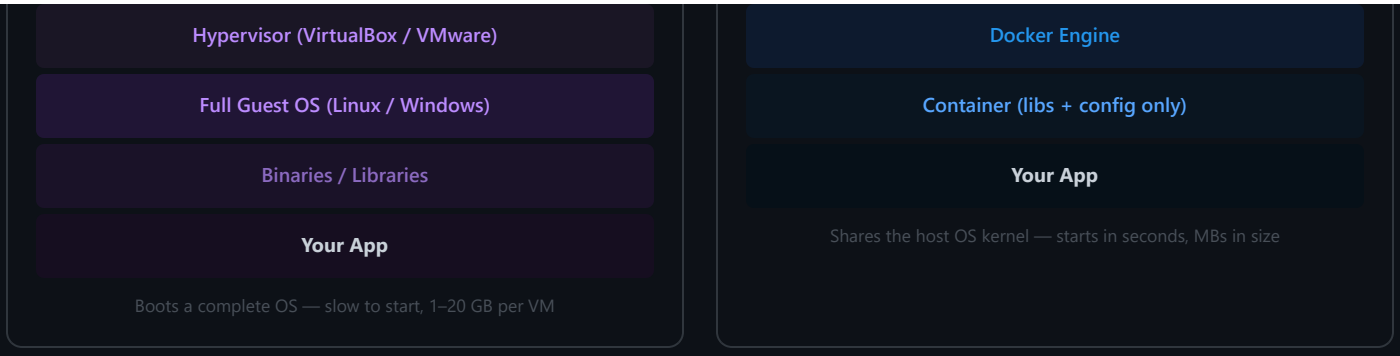
Docker solves this by making the environment part of the application. Instead of installing Apache onto a server and hoping the server stays consistent, you run an Apache *container* that includes everything Apache needs. The server just needs Docker — nothing else.

> **The key idea:** A container is not an app installed on your system. It is a self-contained environment that runs *on top of* your system, isolated from everything else, and portable to any machine running Docker.

2. Containers vs Virtual Machines

Both containers and virtual machines (VMs) solve the "consistent environment" problem, but they do it very differently — and the difference matters for everyday use.





A VM runs a complete operating system inside a hypervisor. It's powerful and fully isolated, but it takes minutes to boot and gigabytes of disk space. A Docker container shares the host's OS kernel and only packages what's different — the libraries and config your app needs. The result is a container that starts in one or two seconds and might be 50 MB instead of 10 GB.

- **Start time:** VM = minutes. Container = seconds.
- **Disk space:** VM = gigabytes. Container = megabytes.
- **Isolation:** Both are isolated. Containers share the kernel but have their own filesystem, processes, and network.
- **Portability:** Both are portable, but a Docker image moves far more easily than a VM disk file.

> **When would you still use a VM?** When you need a completely different OS (e.g. running Windows on a Linux host), or when you need hardware-level isolation for security. For everything else — web servers, databases, development environments — Docker is faster and lighter.

3. Key Concepts

Docker has three core concepts you need to understand before anything else makes sense. Everything in Docker flows from these three ideas.

Image

A read-only blueprint. An image contains the filesystem, libraries, and instructions needed to run an application. You don't run an image directly — you create a container from it.

Think of it like a cake recipe. The recipe itself isn't a cake.

Container

A running instance of an image. You can run many containers from the same image simultaneously, and each is independent.

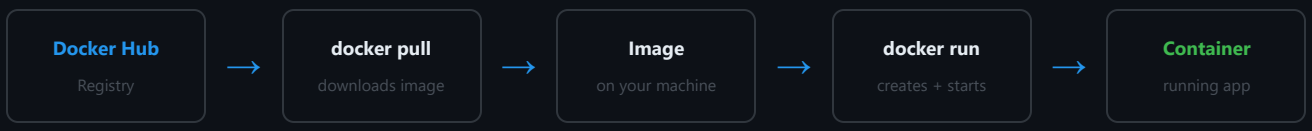
Using the recipe analogy — a container is the actual cake you baked from it.

Registry

A place to store and share images. Docker Hub is the default public registry — it hosts official images for Apache, MySQL, Python, Node, and thousands more, free to pull and use.

Docker Engine

The background service (daemon) that does all the work — pulling images, creating containers, managing networking and storage. You talk to it via the `docker` command.



4. Installing Docker

On Windows (Docker Desktop)

1. **Check your Windows version**

   Docker Desktop requires Windows 10 64-bit (Build 19041+) or Windows 11. Open Settings → System → About to check. WSL 2 (Windows Subsystem for Linux) is the recommended backend — Docker Desktop installs it for you if needed.

2. **Enable virtualisation in BIOS**

   Virtualisation must be enabled. Most modern PCs have it on by default. Check Task Manager → Performance → CPU — if "Virtualization: Enabled" is shown, you're good. If not, enable it in your BIOS under CPU settings (Intel VT-x / AMD-V).

3. **Download Docker Desktop**

   Go to `docker.com/products/docker-desktop` and download the Windows installer. Run it and follow the prompts — it will install WSL 2 if needed and ask you to restart.

4. **Start Docker Desktop**

   After restarting, launch Docker Desktop from the Start menu. Wait for the whale icon in the taskbar to stop animating — that means the Docker engine is running. You won't need the Docker Desktop GUI much; most of the work is done in the terminal.

On Ubuntu / Debian Linux

```
# Remove any old Docker packages first
sudo apt remove docker docker-engine docker.io containerd runc

# Install prerequisites
sudo apt update
sudo apt install ca-certificates curl gnupg

# Add Docker's official GPG key and repository
sudo install -m 0755 -d /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
  sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
  https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list

# Install Docker Engine
sudo apt update
sudo apt install docker-ce docker-ce-cli containerd.io docker-compose-plugin

# Add your user to the docker group (avoids needing sudo every time)
sudo usermod -aG docker $USER

# Log out and back in, then verify:
docker --version
```

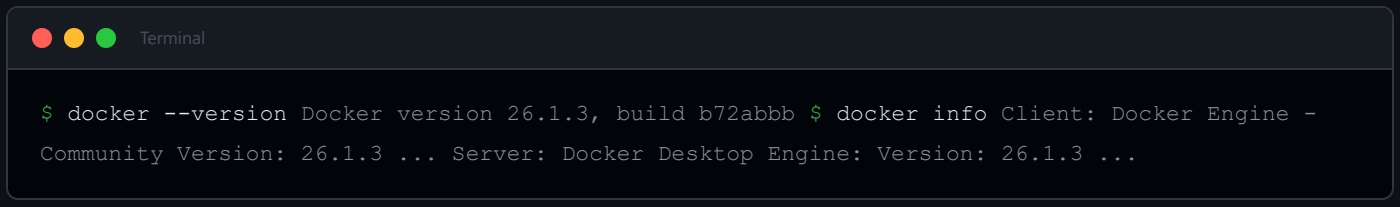
> **Raspberry Pi users:** Use the convenience script instead — `curl -fsSL https://get.docker.com | sh` — it handles the ARM architecture automatically. Then run `sudo usermod -aG docker $USER` and log back in.

On macOS

Download Docker Desktop from `docker.com/products/docker-desktop`, choose the correct version for your chip (Apple Silicon or Intel), and drag it to Applications. Start it and wait for the whale icon to settle in the menu bar.

5. Verifying the Install

Open a terminal (PowerShell on Windows, or any terminal on Linux/Mac) and run these two commands to confirm Docker is installed and the engine is running:



If `docker info` returns an error like "Cannot connect to the Docker daemon", the Docker engine isn't running. On Windows/Mac, make sure Docker Desktop is open. On Linux, start it with `sudo systemctl start docker`.

6. Your First Container

The traditional first Docker command pulls a tiny test image from Docker Hub and runs it. It's deliberately simple — but watch what actually happens:

Terminal

```
$ docker run hello-world Unable to find image 'hello-world:latest' locally latest: Pulling from library/hello-world 719385e32844: Pull complete Digest: sha256:4e83453afed1b... Status: Downloaded newer image for hello-world:latest Hello from Docker! This message shows that your installation appears to be working correctly. To generate this message, Docker took the following steps: 1. The Docker client contacted the Docker daemon. 2. The Docker daemon pulled the "hello-world" image from Docker Hub. 3. The Docker daemon created a new container from that image, which runs the executable that produces this output. 4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.
```

Notice what happened automatically:

- **Docker looked for the image locally** — it wasn't there
- **Docker pulled it from Docker Hub** — `library/hello-world` is an official image
- **Docker created and ran a container from it** — the container printed its message and exited
- **The container stopped** — it finished its job, so it stopped. Containers only run as long as their process runs

Now try something slightly more interesting — run an interactive Ubuntu shell inside a container:

Terminal

```
$ docker run -it ubuntu bash Unable to find image 'ubuntu:latest' locally latest: Pulling from library/ubuntu ... root@a3f9c2d1e8b7:/# cat /etc/os-release NAME="Ubuntu" VERSION="24.04 LTS (Noble Numbat)" root@a3f9c2d1e8b7:/# exit $
```

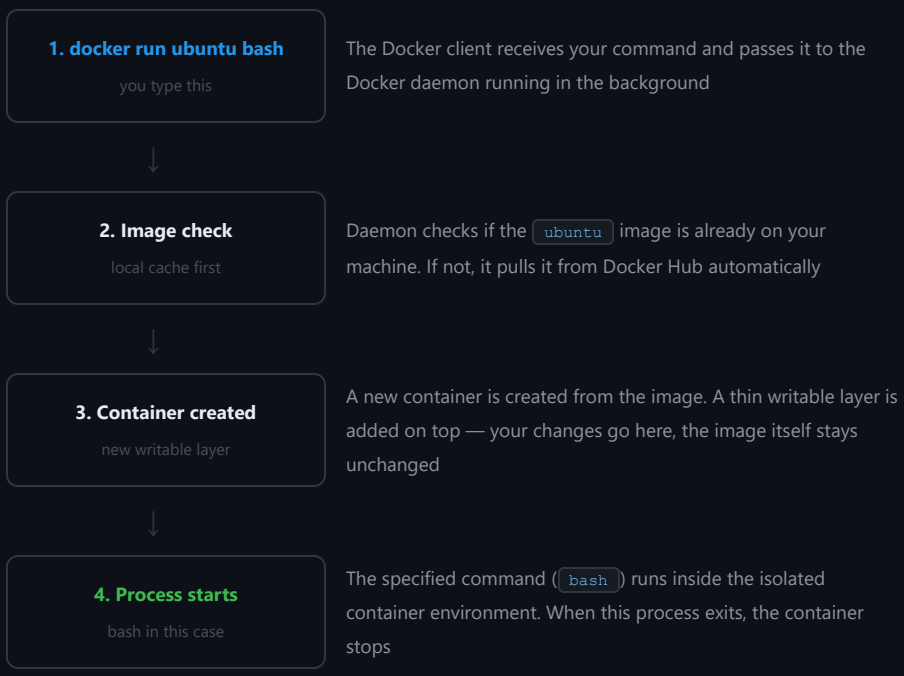
You just ran a full Ubuntu environment, got a shell inside it, ran a command, and exited — all without installing Ubuntu on your machine. The two flags used here are important:

- `-i` — interactive mode (keep STDIN open)
- `-t` — allocate a terminal (so you get a proper shell prompt)

These are almost always used together as `-it` whenever you want to type commands inside a container.

7. What Happened Under the Hood

Every `docker run` command goes through the same sequence:



> **Images are layered and shared:** If you run both `ubuntu` and a custom image built on top of `ubuntu`, they share the same underlying Ubuntu layers on disk. Docker only stores each layer once, which is why pulling a second Ubuntu-based image is much faster than the first.

8. Three Things to Remember About Containers

These three properties catch beginners out. Keep them in mind and a lot of Docker will make more sense:

- **Containers are ephemeral by default.** When a container stops, any files written inside it are gone the next time you start a fresh container from the same image. This is intentional — containers are meant to be disposable. Chapter 4 covers how to persist data using volumes.
- **Each container has its own network.** A web server running inside a container is not automatically accessible from your browser — you have to explicitly map a port from the container to your host. You'll see this in Chapter 5 and in the Apache scenario in Chapter 7.
- **The image is never modified by running it.** No matter what you do inside a running container, the original image stays exactly as it was. Next time you create a container from that image, it starts fresh. To save changes into a new image, you write a Dockerfile (Chapter 6).

EXERCISES

1. **Run hello-world** and read the output carefully. It tells you exactly the four steps Docker took to produce the message. Make sure you can explain each step in your own words.
2. **Run an interactive Ubuntu container** with `docker run -it ubuntu bash`. Once inside, run `apt update && apt install -y curl` to install curl. Then exit the container and run a brand new `docker run -it ubuntu bash` — confirm that curl is no longer installed. This demonstrates that containers are ephemeral.
3. **Run an Alpine container** — `docker run -it alpine sh` (Alpine uses `sh` not `bash`). Alpine is a tiny Linux distro, very popular in Docker images. Check its size: exit the container and run `docker images` to compare the size of `alpine` vs `ubuntu`.
4. **List your images and containers**: run `docker images` to see what you've pulled, and `docker ps -a` to see all containers including stopped ones. Notice that each `docker run` created a separate container even when using the same image.
5. **Clean up**: remove all stopped containers with `docker container prune` and confirm with `docker ps -a`. This is a habit worth building — stopped containers don't use CPU or memory, but they do use a small amount of disk space.

Next: Chapter 2 — Images, Containers and the Docker Hub

Now that Docker is running, Chapter 2 goes deeper into images: how they're structured in layers, how to search and browse Docker Hub, how to pull specific versions with tags (`nginx:1.27` vs `nginx:latest`), and how to manage the images and containers on your machine. You'll also run your first web server container and open it in a browser.

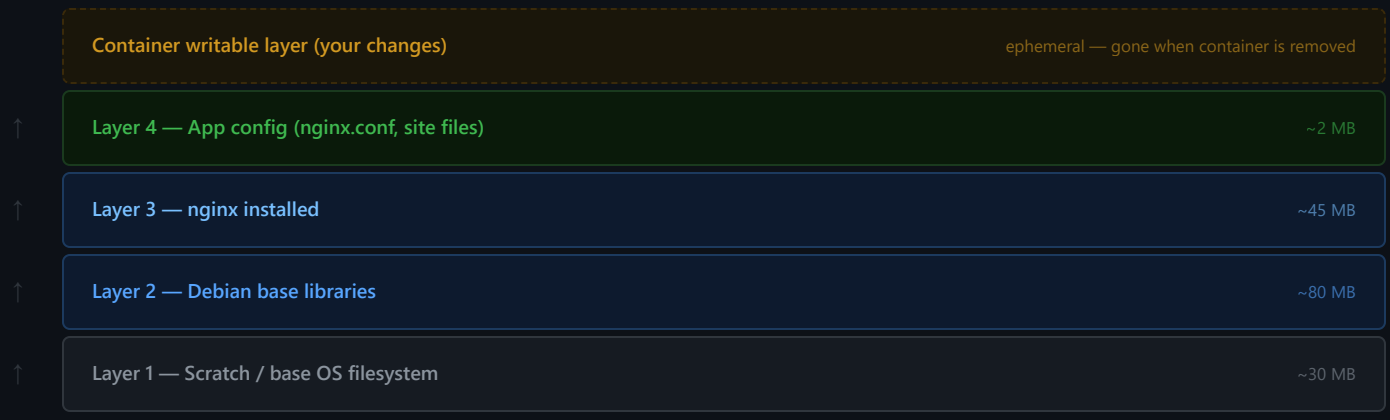
Chapter 2 — Images, Containers and the Docker Hub

Chapter 2 — Images, Containers and the Docker Hub

In Chapter 1 you ran your first container and saw Docker pull an image automatically. In this chapter we go deeper: how images are actually structured, how to find and choose the right image on Docker Hub, how tags work, and how to manage the images and containers accumulating on your machine. By the end you'll have a real web server running inside a container and visible in your browser.

1. How Images Are Built — Layers

A Docker image is not a single flat file. It is a stack of read-only **layers**, each one representing a set of changes made on top of the previous layer. When you run a container, Docker adds one more thin writable layer on top for your changes.



This layered design has two important practical benefits:

- **Sharing saves disk space.** If you pull an `nginx` image and a `php` image that both build on the same Debian base, Docker only stores the Debian layers once. The shared layers are reused.
- **Caching speeds up builds.** When you update your app's config (Layer 4 above), Docker only rebuilds that layer and above — it reuses the cached nginx and OS layers. This is why the second build is always faster than the first.

> **Copy-on-write:** When a running container modifies a file that exists in a read-only layer, Docker copies that file up into the writable layer and modifies it there. The original image layer is never touched. This is called copy-on-write (CoW) and it's why you can run ten containers from the same image without ten copies of the image.

2. Docker Hub — Finding the Right Image

Docker Hub (`hub.docker.com`) is the default public registry. It hosts tens of thousands of images. When you type `docker pull nginx`, Docker looks there first. You can also search from the command line:

Terminal

```
$ docker search nginx NAME DESCRIPTION STARS OFFICIAL nginx Official build of Nginx... 20138 [OK] unit Official build of NGINX Unit... 76 [OK] nginx/nginx-ingress NGINX and NGINX Plus Ingress... 96 bitnami/nginx Bitnami container image for... 198 ...
```

The most important column is **OFFICIAL**. Official images are maintained by the software's own team (or Docker itself) and are the safest choice for production use. Here's how to read the results:

nginx

Official build of Nginx. High performance web server and reverse proxy.

✓ Official Image | ⬇ 1B+ pulls | ★ 20,138

bitnami/nginx

Bitnami container image for Nginx. Includes non-root user, health checks, extra tooling.

Community Image | ⬇ 100M+ pulls | ★ 198

> **Stick to official images when starting out.** They're well documented, regularly updated with security patches, and use minimal base images to keep sizes small. For the scenarios in this course (Apache, Python, MySQL) we'll always use the official images.

3. Understanding Tags

Every image on Docker Hub has one or more **tags** — labels that identify different versions or variants of the same image. The format is `image:tag`. If you omit the tag, Docker assumes `:latest`.

Tag	What you get	Use when
`nginx:latest`	Most recent stable release (same as `nginx`)	Experimenting / learning
`nginx:1.27`	Specific version pinned	Production — you control when you upgrade ✓ recommended

Tag	What you get	Use when
`nginx:alpine`	Built on Alpine Linux — much smaller image (~45 MB vs ~190 MB)	When image size matters, e.g. Raspberry Pi
`nginx:1.27-alpine`	Specific version + Alpine base	Best of both — pinned version, small size
`python:3.12-slim`	Slim variant — fewer pre-installed packages	When you know exactly what you need
`mysql:8.0`	MySQL 8.0.x (patch version floats)	Gets security patches automatically within 8.0

> **Avoid `:latest` in anything long-running.** If you use `nginx:latest` today and pull it again in three months, you might get a different version. For a home web server this is fine; for anything that needs to stay consistent, always pin a version tag.

To see all available tags for an image, visit its Docker Hub page (e.g. `hub.docker.com/_/nginx`) and click the Tags tab — official images typically have dozens.

4. Pulling and Inspecting Images

You can pull an image without running it using `docker pull`. This is useful when you want to download an image in advance or fetch a specific tag:

Terminal

```
$ docker pull nginx:alpine alpine: Pulling from library/nginx 2d429b9e73a6: Pull complete # Layer 1 — Alpine base 20c8b3871098: Pull complete # Layer 2 — nginx binaries 06da587a7970: Pull complete # Layer 3 — nginx config Digest: sha256:a45ee5d042aaa9e81... Status: Downloaded newer image for nginx:alpine $ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	alpine	a6a8fe5e09c3	2 weeks ago	47.1MB
ubuntu	latest	35a88802559d	3 weeks ago	78.1MB
hello-world	latest	74cc54e27dc4	5 months ago	13.3kB

To see the layers that make up an image, use `docker image history`:

Terminal

```
$ docker image history nginx:alpine IMAGE CREATED CREATED BY SIZE a6a8fe5e09c3 2 weeks ago CMD ["nginx" "-g" "daemon off;"] 0B <missing> 2 weeks ago EXPOSE map[80/tcp:{}] 0B <missing> 2 weeks
```

```
ago COPY nginx.conf /etc/nginx/... 1.01kB <missing> 2 weeks ago RUN apk add --no-cache nginx
15.2MB <missing> 2 weeks ago FROM alpine:3.19 7.38MB
```

Reading bottom to top, you can see exactly how the image was built: start from Alpine, install nginx, copy in a config file, expose port 80, set the start command.

5. Running Your First Web Server

Let's run nginx and actually see it in a browser. The key new concept here is **port mapping** — telling Docker to connect a port inside the container to a port on your machine.

Terminal

```
$ docker run -d -p 8080:80 --name my-nginx nginx:alpine a91b3c2d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b
```

Now open `http://localhost:8080` in your browser — you should see the nginx welcome page. Let's break down the flags used:

- `-d` — detached mode. The container runs in the background so your terminal is free. Without this, the container's output streams to your terminal and you can't type anything else.
- `-p 8080:80` — port mapping. Format is `host-port:container-port`. nginx listens on port 80 inside the container; we map that to port 8080 on our machine. You can use any free port on the left side.
- `--name my-nginx` — give the container a memorable name. Without this, Docker assigns a random name like `funny_einstein`.

> **Port mapping syntax:** `-p HOST:CONTAINER`
> `-p 8080:80` → your browser visits `localhost:8080`, Docker forwards traffic to port `80` inside the container.
> `-p 3000:3000` → same port on both sides (common for dev servers).
> `-p 127.0.0.1:8080:80` → only accessible from localhost, not the network.

6. Managing Running Containers

With a container running in the background, here are the key commands for checking on it, peeking inside it, and stopping it:

Terminal

```
$ docker ps CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES a91b3c2d4e5f nginx:alpine "/docker-entrypoint.…" 2 minutes ago Up 2 minutes 0.0.0.0:8080->80/tcp my-nginx $ docker logs my-nginx 2024/06/16 10:32:14 [notice] 1#1: start worker process 6 172.17.0.1 - - [16/Jun/2024:10:32:22] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0..." $ docker exec -it my-nginx sh / # ls /etc/nginx/ conf.d fastcgi.conf mime.types nginx.conf ... / # exit $ docker stop my-nginx my-nginx $ docker start my-nginx my-nginx
```

- **`docker ps`** — list running containers. Add `-a` to also see stopped ones.
- **`docker logs my-nginx`** — see the container's output (access logs, errors). Add `-f` to follow in real time.
- **`docker exec -it my-nginx sh`** — open a shell *inside a running container*. Different from `docker run` — this attaches to an existing container rather than starting a new one. Alpine-based images use `sh` not `bash`.
- **`docker stop my-nginx`** — gracefully stop the container (sends SIGTERM, waits 10 seconds, then SIGKILL).
- **`docker start my-nginx`** — restart a stopped container. It keeps the same name, port mappings, and settings.

7. The Container Lifecycle

A stopped container still exists — it occupies a small amount of disk space and retains its writable layer. Only `docker rm` truly removes it. Removing a container does *not* remove the image it came from — the image stays on disk for next time.

```
# Remove a stopped container
docker rm my-nginx

# Stop and remove in one command
docker rm -f my-nginx

# Remove all stopped containers at once
docker container prune

# Remove an image
docker rmi nginx:alpine

# Remove all unused images (not referenced by any container)
docker image prune -a
```

> **You can't remove an image while a container using it exists** — even if the container is stopped. Remove the container first, then the image. Or use `docker rm -f` to force-remove a running container.

8. Restart Policies

If your server reboots, Docker containers don't restart automatically by default. For anything you want to stay running — a web server, a database — add a **restart policy** when you create the container:

```
# always restart (even after Docker Desktop restarts)
docker run -d -p 8080:80 --name my-nginx --restart always nginx:alpine

# restart unless you manually stopped it
docker run -d -p 8080:80 --name my-nginx --restart unless-stopped nginx:alpine

# only restart on failure (not if you stopped it manually)
docker run -d -p 8080:80 --name my-nginx --restart on-failure nginx:alpine
```

For a fallback web server (like the one we'll build in Chapter 7), `--restart unless-stopped` is the right choice — it comes back automatically after a reboot, but stays stopped if you deliberately stop it.

EXERCISES

1. **Run nginx and open it in a browser.** Use `docker run -d -p 8080:80 --name my-nginx nginx:alpine`. Visit `http://localhost:8080` and confirm you see the nginx welcome page. Check the access log with `docker logs my-nginx` and find the line recording your browser request.
2. **Explore tags.** Pull `nginx:latest` and `nginx:alpine` and compare their sizes with `docker images`. The alpine version should be roughly 4× smaller. Then pull `python:3.12-slim` and compare it to `python:3.12` — again check the size difference.
3. **Peek inside a running container.** While my-nginx is running, use `docker exec -it my-nginx sh` to get a shell inside it. Navigate to `/usr/share/nginx/html/` and use `cat index.html` to read the default welcome page. Exit the container — the web server keeps running.
4. **Practise the lifecycle.** Stop my-nginx with `docker stop my-nginx`, verify it appears in `docker ps -a` as Exited, then restart it with `docker start my-nginx` and confirm it's accessible in the browser again. Finally remove it with `docker rm -f my-nginx`.
5. **Clean up your image cache.** Run `docker images` to see everything you've pulled so far. Remove the `hello-world` image with `docker rmi hello-world` (remove any stopped hello-world containers first if needed). Then run

`docker system df` to see a summary of how much disk space Docker is using across images, containers, and volumes.

Next: Chapter 3 — Essential Docker Commands

You now know how to pull images, run containers, check their status, and clean up. Chapter 3 is your practical command reference — covering `run`, `ps`, `stop`, `rm`, `exec`, `logs`, `inspect`, and `cp` with real examples for each, plus the most useful flags you'll reach for every day.

Chapter 3 — Essential Docker Commands

Chapter 3 — Essential Docker Commands

This is your practical reference chapter. You've already seen several Docker commands in action — now we cover each one properly, with all the useful flags, real annotated output, and the patterns you'll reach for every single day. Keep this chapter handy while you work through the rest of the course.

We'll cover commands in logical groups: running containers, monitoring them, getting inside them, moving files, and cleaning up. At the end there's a condensed cheat sheet you can refer back to at a glance.

1. docker run — Create and Start a Container

`docker run` is the command you'll use most. It creates a new container from an image and starts it. Every flag you attach shapes how the container behaves for its entire lifetime.

`docker run [flags] IMAGE [COMMAND]` Create and start a new container

FLAG	WHAT IT DOES	EXAMPLE
`-d`	Detached — run in background, return container ID	`docker run -d nginx`
`-it`	Interactive terminal — keep STDIN open and allocate a TTY. Almost always used together	`docker run -it ubuntu bash`
`-p HOST:CONT`	Map a host port to a container port	`-p 8080:80`
`--name NAME`	Give the container a memorable name instead of a random one	`--name my-site`
`-e KEY=VALUE`	Set an environment variable inside the container	`-e MYSQL_ROOT_PASSWORD=secret`
`-v HOST:CONT`	Mount a volume or host directory into the container	`-v /my/data:/var/lib/mysql`
`--rm`	Automatically remove the container when it exits — great for one-off tasks	`docker run --rm alpine echo hi`
`--restart POLICY`	Restart policy: `no` (default), `always`, `unless-stopped`, `on-failure`	`--restart unless-stopped`
`--network NAME`	Connect to a specific Docker network	`--network my-net`
`-w DIR`	Set the working directory inside the container	`-w /app`
`--memory 512m`	Limit RAM the container can use	`--memory 256m`

Terminal — common docker run patterns

```
# Background web server, port 8080, auto-restart on reboot $ docker run -d -p 8080:80 --name site --restart unless-stopped nginx:alpine # One-off task — container removed automatically when done $ docker run --rm alpine echo "Hello from Alpine" Hello from Alpine # Interactive Python session in a container $ docker run -it --rm python:3.12-slim python >>> print("Running inside Docker!") Running inside Docker! >>> exit() # Pass environment variable (e.g. database password) $ docker run -d --name mydb \ -e MYSQL_ROOT_PASSWORD=mysecret \ -e MYSQL_DATABASE=myapp \ mysql:8.0
```

--rm is your best friend for experimentation. Any time you want to try something — run a quick Python script, test a CLI tool, check a config — add `--rm` and the container vanishes the moment it finishes. No cleanup needed.

2. docker ps — List Containers

`docker ps [flags]` List containers (running only by default)

FLAG	WHAT IT DOES
`-a`	Show all containers, including stopped ones
`-q`	Quiet — only output container IDs (useful in scripts)
`--filter status=exited`	Filter by status: created, running, paused, exited, dead
`--format`	Custom output format using Go templates

Terminal

```
$ docker ps -a CONTAINER ID IMAGE COMMAND CREATED STATUS NAMES a91b3c2d4e5f nginx:alpine "nginx..." 5 minutes ago Up 5 minutes site b82c4d3e5f6a ubuntu "bash" 2 hours ago Exited (0) 2 hours ago loving_einstein c73d5e4f6a7b hello-world "/hello" 1 day ago Exited (0) 1 day ago hopeful_morse # Stop ALL running containers in one go $ docker stop $(docker ps -q) # Clean up ALL stopped containers $ docker container prune WARNING! This will remove all stopped containers. Are you sure you want to continue? [y/N] y Deleted Containers: b82c4d3e5f6a c73d5e4f6a7b Total reclaimed space: 24B
```

3. docker logs — View Container Output

`docker logs [flags] CONTAINER` Fetch the logs (stdout/stderr) of a container

FLAG	WHAT IT DOES	EXAMPLE
`-f`	Follow — stream new log lines in real time (like `tail -f`). Press Ctrl+C to stop following without stopping the container	`docker logs -f site`
`--tail N`	Only show the last N lines	`docker logs --tail 50 site`
`--since TIME`	Show logs since a timestamp or relative time	`docker logs --since 5m site`
`-t`	Add timestamps to each log line	`docker logs -t site`

Terminal

```
# Watch live access log as requests come in $ docker logs -f site 172.17.0.1 - -
[16/Jun/2024:10:45:22 +0000] "GET / HTTP/1.1" 200 615 172.17.0.1 - - [16/Jun/2024:10:45:25
+0000] "GET /favicon.ico HTTP/1.1" 404 555 ^C # Last 10 lines with timestamps — useful for
diagnosing a crash $ docker logs --tail 10 -t site 2024-06-16T10:44:01.234567890Z 2024/06/16
10:44:01 [notice] nginx started 2024-06-16T10:45:22.123456789Z 172.17.0.1 - GET / 200
```

Where do logs come from? Docker captures whatever the container's main process writes to stdout and stderr. Well-behaved applications write their logs there by default (nginx, Apache, MySQL all do). If you're building your own image and logs aren't appearing, make sure your app logs to stdout rather than to a file.

4. docker exec — Run Commands Inside a Running Container

`docker exec [flags] CONTAINER COMMAND` Execute a command in a running container

FLAG	WHAT IT DOES
`-it`	Interactive terminal — needed when opening a shell
`-d`	Detached — run the command in background
`-e KEY=VALUE`	Set environment variable for this command only
`-u USER`	Run as a specific user (e.g. `-u root`)
`-w DIR`	Set working directory for this command

Terminal

```
# Open an interactive shell inside a running container $ docker exec -it site sh / # nginx -v nginx version: nginx/1.27.0 / # exit # Run a single command without opening a shell $ docker exec site nginx -t nginx: the configuration file /etc/nginx/nginx.conf syntax is ok nginx: configuration file /etc/nginx/nginx.conf test is successful # Reload nginx config without restarting the container $ docker exec site nginx -s reload # Connect to MySQL inside a running database container $ docker exec -it mydb mysql -u root -p Enter password: Welcome to the MySQL monitor...
```

exec vs run: `docker exec` runs inside an *already running* container. `docker run` creates a brand new container. If you want to poke around inside your nginx container without stopping it, always use `exec`.

5. docker cp — Copy Files To and From Containers

`docker cp SRC DEST` Copy files between host and container. Works on running or stopped containers.

Use `CONTAINER:PATH` syntax to refer to a path inside a container.

DIRECTION	COMMAND
Host → Container	`docker cp ./index.html site:/usr/share/nginx/html/`
Container → Host	`docker cp site:/etc/nginx/nginx.conf ./nginx.conf`
Copy a directory	`docker cp ./mysite/ site:/usr/share/nginx/html/`

Terminal

```
# Copy a custom HTML page into a running nginx container $ echo "<h1>Hello from Docker!</h1>" > index.html $ docker cp index.html site:/usr/share/nginx/html/index.html Successfully copied 2.05kB to site:/usr/share/nginx/html/index.html # Now refresh http://localhost:8080 — you'll see "Hello from Docker!" # Extract the default nginx config as a starting point $ docker cp site:/etc/nginx/nginx.conf ./nginx.conf Successfully copied 2.77kB to /home/user/nginx.conf
```

docker cp vs volumes: `docker cp` is great for one-off file transfers and extracting config files. For ongoing work — where you want file changes on your host to be immediately visible inside the container — use a volume mount (`-v`). Chapter 4 covers volumes in detail.

6. docker inspect — Full Container Details

`docker inspect` returns everything Docker knows about a container or image as a JSON object. It's invaluable for debugging — finding the IP address of a container, checking what volumes are mounted, seeing environment variables, and more.

Terminal

```
$ docker inspect site [ { "Id": "a91b3c2d4e5f...", "Name": "/site", "State": { "Status":
"running", "Running": true, ... }, "NetworkSettings": { "IPAddress": "172.17.0.2", "Ports": {
"80/tcp": [{ "HostPort": "8080" }] } }, "Mounts": [], "Config": { "Env":
["PATH=/usr/local/sbin:/usr/local/bin:..."], "Image": "nginx:alpine" } } ]
```

The output is large. Use `--format` with Go template syntax to extract just what you need:

```
# Get just the container's IP address
docker inspect --format '{{.NetworkSettings.IPAddress}}' site
172.17.0.2

# Get the restart policy
docker inspect --format '{{.HostConfig.RestartPolicy.Name}}' site
unless-stopped

# Get all environment variables
docker inspect --format '{{range .Config.Env}}{{println .}}{{end}}' site

# Works on images too — see the exposed ports
docker inspect --format '{{.Config.ExposedPorts}}' nginx:alpine
map[80/tcp:{}]
```

7. Stopping, Starting, and Removing Containers

```
# Graceful stop (sends SIGTERM, waits 10s, then SIGKILL)
docker stop site

# Immediate kill — no grace period
docker kill site

# Change the grace period to 30 seconds
docker stop --time 30 site
```

```
# Restart a running or stopped container
docker restart site

# Remove a stopped container
docker rm site

# Force remove a running container (stop + rm in one)
docker rm -f site

# Remove all stopped containers
docker container prune

# Rename an existing container
docker rename old-name new-name
```

8. Image Commands

```
# List all local images
docker images
docker image ls             # same thing, newer syntax

# Pull without running
docker pull nginx:alpine

# Remove an image (container must be removed first)
docker rmi nginx:alpine
docker image rm nginx:alpine  # same thing

# Remove ALL unused images (not used by any container)
docker image prune -a

# Show image layers and how they were built
docker image history nginx:alpine

# Tag an image with a new name
docker tag nginx:alpine my-nginx:v1

# Search Docker Hub from the CLI
docker search --filter is-official=true python
```

9. docker system — Disk Usage and Global Cleanup

Terminal

```
# See how much disk Docker is using $ docker system df TYPE TOTAL ACTIVE SIZE RECLAIMABLE Images 6 2 1.23GB 890MB (72%) Containers 3 1 12.5kB 12.5kB Local Volumes 2 1 145MB 0B Build Cache 18 0 234MB 234MB # Remove everything unused: stopped containers, unused images, # unused networks, build cache. The nuclear option — use with care. $ docker system prune -a WARNING! This will remove: - all stopped containers - all networks not used by at least one container - all images without at least one container associated to them - all build cache Are you sure you want to continue? [y/N]
```

> **docker system prune -a** removes all images not currently in use by a running container — including images you might want again soon. It's useful for freeing up disk space, but you'll have to re-pull those images next time. Leave out `-a` to only remove dangling (untagged) images and be a bit more conservative.

10. Quick-Reference Cheat Sheet

RUNNING CONTAINERS	
`docker run -d IMAGE`	Start in background
`docker run -it IMAGE sh`	Interactive shell
`docker run --rm IMAGE`	Auto-remove on exit
`docker run -p 8080:80`	Map host:container port
`docker run -e KEY=VAL`	Set env variable
`docker run -v /h:/c`	Mount directory
`--name NAME`	Name the container
`--restart unless-stopped`	Auto-restart on reboot

MONITORING	
`docker ps`	List running containers
`docker ps -a`	All containers (inc. stopped)
`docker logs NAME`	View output
`docker logs -f NAME`	Follow live output
`docker stats`	Live CPU/RAM usage
`docker inspect NAME`	Full container details (JSON)
`docker top NAME`	Processes inside container
`docker system df`	Disk usage summary

CONTAINER CONTROL	
`docker stop NAME`	Graceful stop
`docker start NAME`	Start stopped container
`docker restart NAME`	Stop then start
`docker rm NAME`	Remove stopped container
`docker rm -f NAME`	Force remove (running)
`docker exec -it NAME sh`	Shell into container
`docker cp src NAME:dest`	Copy file in
`docker cp NAME:src dest`	Copy file out

IMAGES AND CLEANUP	
`docker images`	List local images
`docker pull IMAGE:TAG`	Download image
`docker rmi IMAGE`	Remove image
`docker image prune -a`	Remove unused images
`docker container prune`	Remove stopped containers
`docker system prune -a`	Remove everything unused
`docker search TERM`	Search Docker Hub
`docker image history IMG`	Show image layers

EXERCISES

1. **Run a one-off command with --rm.** Use `docker run --rm python:3.12-slim python -c "import sys; print(sys.version)"` to print the Python version from inside a container, then verify with `docker ps -a` that no container was left behind.
2. **Use docker cp to serve a custom page.** Start an nginx container with `docker run -d -p 8080:80 --name site nginx:alpine`. Create a simple `index.html` on your host with any content you like. Copy it into the container using `docker cp` and verify you can see your page at `http://localhost:8080`.
3. **Follow the logs.** With your nginx container running, open a second terminal and run `docker logs -f site`. In your browser, visit `http://localhost:8080` several times. Watch the access log entries appear in real time in the second terminal.
4. **Use docker inspect to find the IP.** Run `docker inspect --format '{{.NetworkSettings.IPAddress}}' site` to get the container's internal IP address. Then use `docker exec -it site sh` to open a shell and run `ip addr` to confirm it matches.
5. **Check disk usage and clean up.** Run `docker system df` to see your current Docker disk usage. Stop and remove the nginx container, then run `docker image prune -a` to remove any images not in use. Run `docker system df` again to see how much space you reclaimed.

Next: Chapter 4 — Volumes and Persistent Data

You've seen that containers lose their data when removed. Chapter 4 solves this with volumes — Docker's mechanism for keeping data alive independent of containers. You'll learn the difference between bind mounts (pointing at a folder on your host) and named volumes (managed by Docker), and see how to use each one to persist a database and serve website files that you can edit directly from your machine.

Chapter 4 — Volumes and Persistent Data

Chapter 4 — Volumes and Persistent Data

Back in Chapter 1 we noted that containers are ephemeral — anything written inside a container is lost the moment the container is removed. For a web server serving static files that's fine. For a database storing your data, or a development environment where you want your edits to survive, it's a problem. **Volumes** are Docker's answer.

This chapter covers how data loss actually happens, the three types of storage Docker provides, and the two you'll use every day: named volumes and bind mounts.

1. The Data Loss Problem

Every container has a thin writable layer on top of its read-only image layers. Any file created or modified inside the container goes into this writable layer. When you remove the container with `docker rm`, that writable layer is deleted with it — permanently.

This happens even if you only run `docker rm` — not `docker stop`. Stopping a container leaves its writable layer intact; removing it deletes the layer. The solution is to store data *outside* the container's writable layer, somewhere that persists independently of any individual container.

2. Three Types of Docker Storage

NAMED VOLUMES

- Managed by Docker — stored in Docker's own area on disk
- Survive container removal
- Easy to back up and migrate

BIND MOUNTS

- Points at a specific folder on your host machine
- Changes on host are instantly visible in container

TMPFS MOUNTS

- Stored in host memory only — never written to disk
- Gone when container stops
- Very fast

- Best for: databases, app data you don't need to edit directly
- Use this for databases ✓

- Changes in container are instantly visible on host
- Best for: website files, source code, config files
- Use this for dev work ✓

- Best for: sensitive data (secrets, tokens) that should never touch disk
- Advanced / specialist use

For the rest of this chapter we'll focus on named volumes and bind mounts — the two you'll use in almost every real scenario.

3. Named Volumes

A named volume is a storage area that Docker creates and manages. You give it a name, mount it into one or more containers at a specific path, and Docker handles where the data actually lives on disk. You don't need to know or care about the exact host path — Docker manages it for you.

Terminal

```
# Create a named volume explicitly (optional — docker run creates it automatically) $ docker volume create mysql-data mysql-data # Start MySQL, mounting the named volume at /var/lib/mysql (where MySQL stores data) $ docker run -d \ --name mydb \ -e MYSQL_ROOT_PASSWORD=secret \ -e MYSQL_DATABASE=myapp \ -v mysql-data:/var/lib/mysql \ --restart unless-stopped \ mysql:8.0 a91b3c2d4e5f... # Write some data into the database $ docker exec -it mydb mysql -u root -psecret myapp mysql> CREATE TABLE users (id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100)); mysql> INSERT INTO users (name) VALUES ('Philip'); mysql> exit # Now destroy the container completely $ docker rm -f mydb mydb # Create a brand new container from the same volume $ docker run -d \ --name mydb \ -e MYSQL_ROOT_PASSWORD=secret \ -v mysql-data:/var/lib/mysql \ mysql:8.0 # The data is still there $ docker exec -it mydb mysql -u root -psecret myapp -e "SELECT * FROM users;" +----+---------+ | id | name | +----+---------+ | 1 | Philip | +----+---------+
```

> **The volume outlives the container.** You could remove and recreate the MySQL container ten times and the data would still be there, because it lives in the `mysql-data` volume — not inside the container itself.

Managing Named Volumes

Command	What it does
`docker volume create NAME`	Create a named volume manually
`docker volume ls`	List all volumes
`docker volume inspect NAME`	Show volume details including where it lives on disk

Command	What it does
`docker volume rm NAME`	Delete a volume (must not be in use by any container)
`docker volume prune`	Remove all volumes not used by any container — careful!

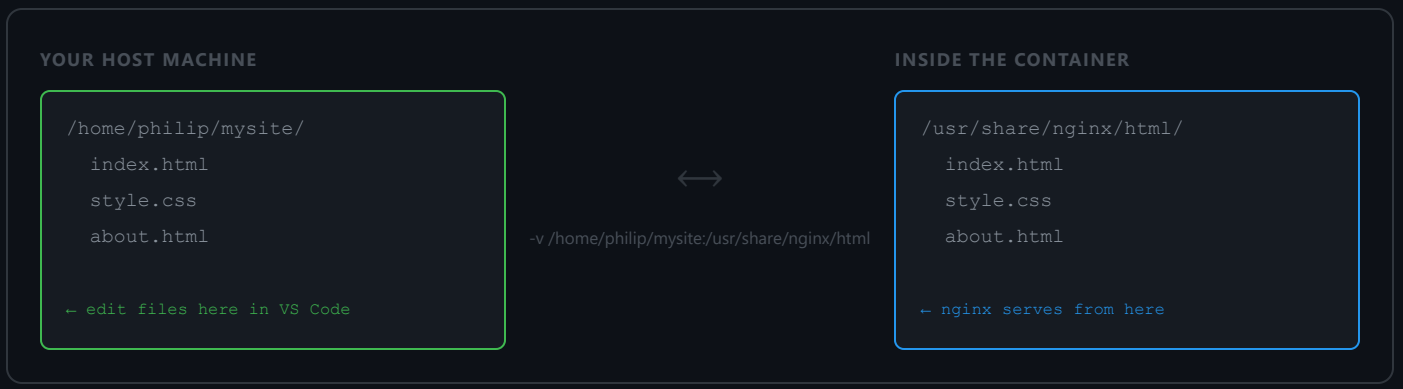
```
# See where Docker actually stores a named volume on disk
docker volume inspect mysql-data
[
  {
    "Name": "mysql-data",
    "Driver": "local",
    "Mountpoint": "/var/lib/docker/volumes/mysql-data/_data",
    "Scope": "local"
  }
]

# On Windows with Docker Desktop, volumes live inside the WSL2 VM
# You can still access them via \\wsl$\docker-desktop-data in Explorer
```

docker volume prune removes data permanently. Unlike `docker container prune` which only removes stopped containers, `docker volume prune` deletes the actual data. Always double-check which volumes are safe to remove before running it.

4. Bind Mounts

A bind mount maps a specific folder on your host machine directly into the container. Unlike named volumes, you control exactly where the data lives — it's just a regular folder on your computer. This makes bind mounts ideal for development: edit files in your editor on the host, and the container sees the changes instantly.



Terminal — serving your website files with a bind mount

```
# On Linux/Mac — use the full path to your site folder $ docker run -d \ --name mysite \ -p
8080:80 \ -v /home/philip/mysite:/usr/share/nginx/html:ro \ --restart unless-stopped \
nginx:alpine # On Windows — use the Windows path (Docker Desktop converts it) $ docker run -d \
--name mysite \ -p 8080:80 \ -v C:\Users\Philip\mysite:/usr/share/nginx/html:ro \ --restart
unless-stopped \ nginx:alpine # Or use $PWD (current directory) on Linux/Mac $ cd
/home/philip/mysite && docker run -d \ --name mysite \ -p 8080:80 \ -v
$(pwd):/usr/share/nginx/html:ro \ nginx:alpine
```

The `:ro` at the end means read-only — the container can read the files but cannot modify them. This is appropriate for serving a website. For a development environment where the container also writes files (logs, compiled output), leave off `:ro`.

> **Live editing workflow:** With a bind mount in place, open `index.html` in your editor on the host, save a change, and refresh your browser — you'll see the change immediately without restarting the container. The container is always reading the actual files on your disk.

5. Named Volume vs Bind Mount — Which to Use

Situation	Use	Why
Database data (MySQL, PostgreSQL, Redis)	Named volume	You don't need to browse the data files directly; Docker manages them safely
Serving website files you're editing	Bind mount	You edit files on the host in your editor and see changes live
Source code for a dev environment	Bind mount	Your editor lives on the host; the container runs/builds the code
Config files (nginx.conf, php.ini)	Bind mount	Easy to edit on host, version-control alongside your project
Shared data between multiple containers	Named volume	Mount the same volume into multiple containers — they share the data
Production app data	Named volume	More portable — not tied to a specific host path

6. Practical Example — nginx with Custom Config and Site Files

This is a preview of Chapter 7's Apache scenario, but with nginx to show the pattern now. We'll mount both the site files and a custom config file from the host:

```
# Folder structure on your host:
# mysite/
#   html/         ← your website files
#     index.html
#   nginx.conf    ← custom nginx configuration

# First, extract the default nginx config to use as a starting point
docker run --rm nginx:alpine cat /etc/nginx/conf.d/default.conf > mysite/nginx.conf

# Run nginx with both mounts
docker run -d \
    --name mysite \
    -p 8080:80 \
    -v $(pwd)/mysite/html:/usr/share/nginx/html:ro \
    -v $(pwd)/mysite/nginx.conf:/etc/nginx/conf.d/default.conf:ro \
    --restart unless-stopped \
    nginx:alpine

# Edit nginx.conf on your host, then reload nginx without restarting the container
docker exec mysite nginx -t         # test the config first
docker exec mysite nginx -s reload  # apply it
```

> **Mounting a single file:** You can bind-mount a single file rather than a whole directory. This is handy for config files — you keep the file in your project folder, bind-mount it into exactly the right place in the container, and edit it like any other file on your machine.

7. Backing Up and Restoring Volumes

Named volumes are managed by Docker, so they're not immediately accessible as a folder. The standard backup technique is to spin up a temporary container, mount the volume, and archive its contents:

```
# ── BACKUP ──────────────────────────────────────────────────────── #
# Mount the volume + a backup destination directory, tar the contents
docker run --rm \
    -v mysql-data:/source:ro \
    -v $(pwd)/backups:/backup \
    alpine \
    tar czf /backup/mysql-data-$(date +%Y%m%d).tar.gz -C /source .

# Result: backups/mysql-data-20240616.tar.gz on your host
```

```
# ── RESTORE ──────────────────────────────────────────────────────────── #
# Create a fresh volume and extract the backup into it
docker volume create mysql-data-restored

docker run --rm \
    -v mysql-data-restored:/target \
    -v $(pwd)/backups:/backup:ro \
    alpine \
    tar xzf /backup/mysql-data-20240616.tar.gz -C /target
```

> **For MySQL specifically** it's better to use `mysqldump` (a logical backup) rather than copying the raw data files. Raw file copying can produce a corrupt backup if MySQL is writing at the same time. Use:
>
> `docker exec mydb mysqldump -u root -psecret myapp > backup.sql`

8. Common Volume Patterns

```
# ── Named volume — database ──────────────────────────────────────────── #
docker run -d \
    -v db-data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=secret \
    mysql:8.0

# ── Bind mount — serve website files (read-only) ─────────────────────── #
docker run -d \
    -p 80:80 \
    -v /home/philip/mysite:/usr/share/nginx/html:ro \
    nginx:alpine

# ── Bind mount — development (read-write, edits visible both ways) ── #
docker run -it \
    -v $(pwd):/app \
    -w /app \
    python:3.12-slim \
    bash

# ── Mount a single config file ──────────────────────────────────────── #
docker run -d \
    -v $(pwd)/nginx.conf:/etc/nginx/conf.d/default.conf:ro \
    nginx:alpine
```

```
# ── Two containers sharing the same named volume ──────────────── #
docker run -d --name writer -v shared-data:/data myapp-writer
docker run -d --name reader -v shared-data:/data:ro myapp-reader
```

EXERCISES

1. **Prove containers are ephemeral.** Run `docker run -it --name test ubuntu bash`, create a file inside with `echo "hello" > /myfile.txt`, then exit and run `docker rm test`. Start a new ubuntu container and confirm `/myfile.txt` doesn't exist. This is the problem volumes solve.
2. **Persist data with a named volume.** Create a named volume called `test-vol` and run `docker run -it --rm -v test-vol:/data ubuntu bash`. Inside, write `echo "persisted!" > /data/hello.txt` and exit. Run a second container with the same volume mount and confirm the file is still there with `cat /data/hello.txt`.
3. **Serve your website with a bind mount.** Create a folder called `mysite` on your host with a simple `index.html`. Run `docker run -d -p 8080:80 -v /full/path/to/mysite:/usr/share/nginx/html:ro nginx:alpine`. Visit `http://localhost:8080` to confirm it serves your file. Now edit `index.html` on your host and refresh — you should see the change immediately without restarting the container.
4. **Inspect a volume.** Run `docker volume ls` to see your volumes, then `docker volume inspect test-vol` to find its Mountpoint on disk. On Linux you can navigate directly to that path and see your files. On Windows with Docker Desktop, note the WSL2 path shown.
5. **Back up a named volume.** Using the backup command from Section 7, create a `.tar.gz` backup of your `test-vol` into a `backups/` folder on your host. Then create a new volume called `test-vol-restore`, restore the backup into it, and verify the file is there by running a container with the restored volume mounted.

Next: Chapter 5 — Networking Basics

Volumes solved persistent data. Chapter 5 solves connectivity. You'll learn how Docker containers communicate — with your browser, with your host, and with each other. Covered: port mapping in depth, Docker's default bridge network, creating custom networks so containers can find each other by name, and the key difference between `localhost` on your host vs inside a container.

Chapter 5 — Networking Basics

Chapter 5 — Networking Basics

Every container you've run so far has been somewhat isolated from the world. You used `-p` to poke a hole in that isolation for your browser, but there's a lot more to Docker networking. How do containers talk to each other? Why does `localhost` behave differently inside a container? How do you connect a web app container to a database container?

This chapter answers those questions and gives you the networking knowledge you need for the practical scenarios in Chapters 7–9.

1. Port Mapping in Depth

A container has its own network namespace — it's like a tiny machine with its own network interface and its own set of ports. A service listening on port 80 inside a container is not the same as port 80 on your host. To make the container's port reachable from outside, you map it with `-p`.


```
# Basic mapping: host 8080 → container 80
docker run -d -p 8080:80 nginx:alpine

# Same port on both sides (common for dev servers)
docker run -d -p 3000:3000 node:20-alpine

# Multiple ports at once
docker run -d -p 80:80 -p 443:443 nginx:alpine

# Bind to a specific host IP — only localhost can reach it, not the network
docker run -d -p 127.0.0.1:8080:80 nginx:alpine

# Let Docker choose a random available host port
```

```
docker run -d -p 80 nginx:alpine
docker port my-container    # find out which port was chosen

# See port mappings for a running container
docker port my-nginx
```

Exposing to the whole network vs localhost only. By default `-p 8080:80` binds to `0.0.0.0` — every network interface on your host, meaning anyone on your local network can reach the container. Use `-p 127.0.0.1:8080:80` if you only want it accessible from your own machine. This matters especially for databases — never expose MySQL's 3306 to the network unless you specifically need remote access.

2. Docker's Built-in Networks

Every Docker installation comes with three networks pre-created. You can see them with `docker network ls`:

Terminal

```
$ docker network ls NETWORK ID NAME DRIVER SCOPE a1b2c3d4e5f6 bridge bridge local b2c3d4e5f6a7 host host local c3d4e5f6a7b8 none null local
```

BRIDGE (DEFAULT)

- All containers join this unless told otherwise
- Containers can reach each other by IP address
- Cannot reach each other by name (DNS doesn't work on the default bridge)
- Containers are isolated from the host network
- Use port mapping (-p) to expose ports

HOST

- Container shares the host's network interface directly
- No port mapping needed — container ports ARE host ports
- Less isolation — container can see all host network traffic
- Linux only (no effect on Docker Desktop for Windows/Mac)
- Useful for high-performance or monitoring containers

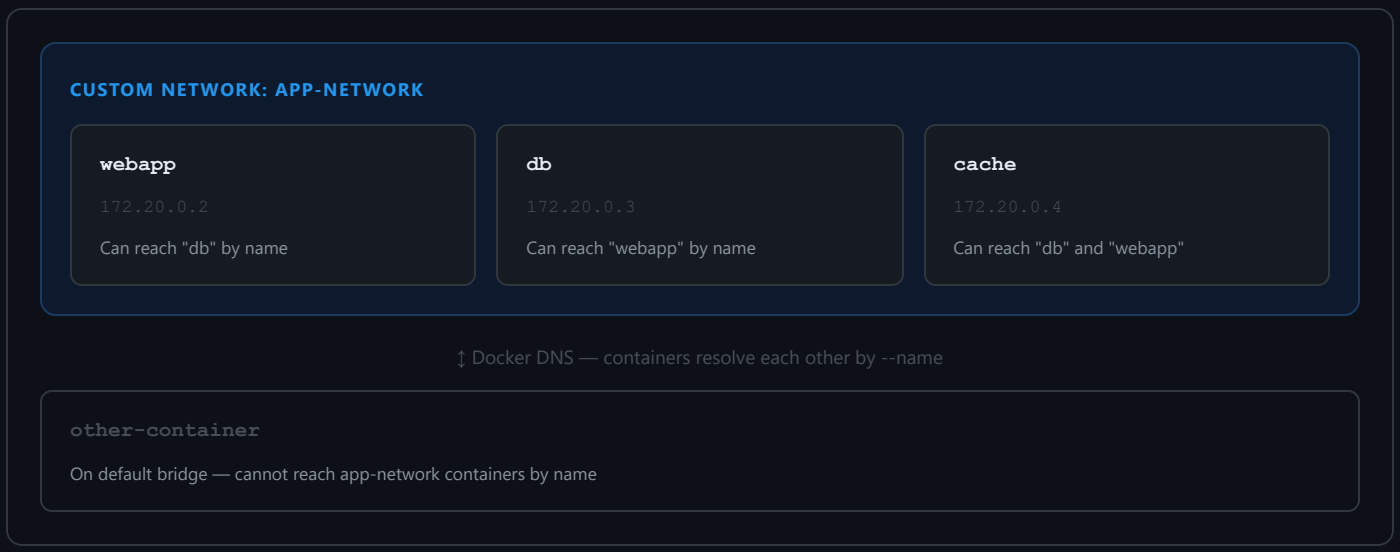
NONE

- No networking at all
- Container has only a loopback interface
- Cannot reach the internet or other containers
- Used for batch jobs that process files and don't need network access

The important limitation of the default bridge: Two containers on the default bridge network can only talk to each other using IP addresses (e.g. `172.17.0.3`), not names. IP addresses change every time a container restarts, making them unreliable. The solution is a custom network — covered in the next section.

3. Custom Networks — Containers Finding Each Other by Name

When you create a custom network, Docker enables its built-in DNS for that network. Any container on the network can reach any other container simply by using its **name** as a hostname. No IP addresses, no guessing, no fragile configuration.



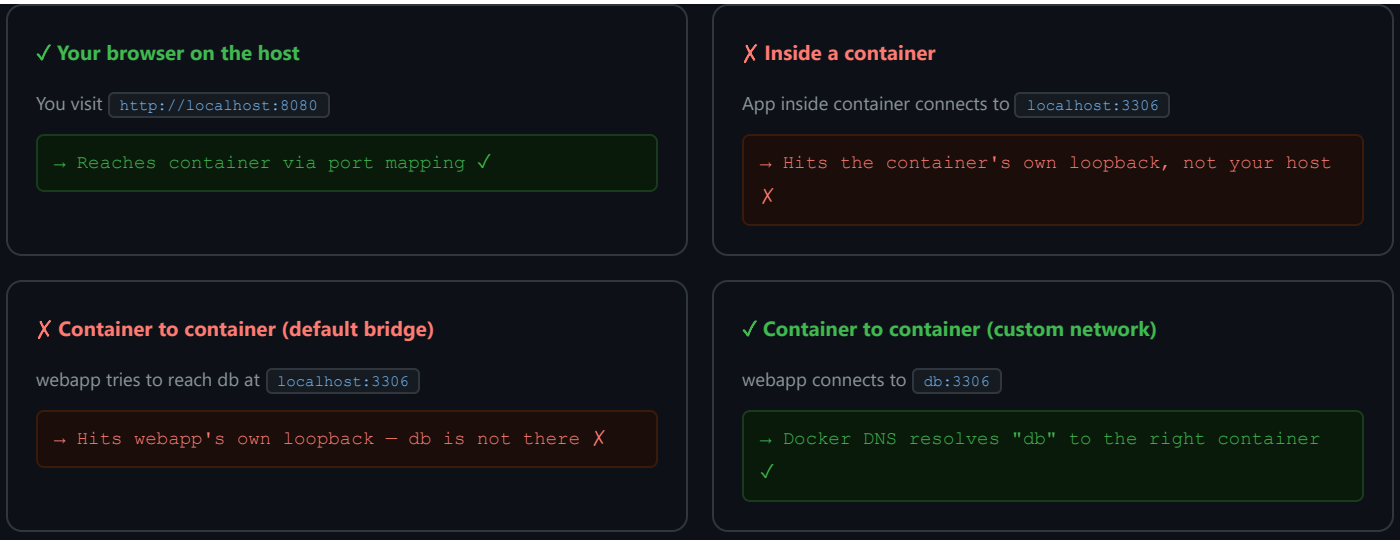
Terminal — web app + database on a custom network

```
# Step 1: create the network $ docker network create app-network
d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7a8 # Step 2: start the database on the network $ docker run -d \ --name db \ --network app-network \ -e MYSQL_ROOT_PASSWORD=secret \ -e MYSQL_DATABASE=myapp \ -v db-data:/var/lib/mysql \ mysql:8.0 # Step 3: start the web app on the same network $ docker run -d \ --name webapp \ --network app-network \ -p 8080:80 \ -e DB_HOST=db \ -e DB_NAME=myapp \ -e DB_PASSWORD=secret \ myapp:latest # The webapp can now connect to MySQL using hostname "db" # Connection string inside webapp: mysql://root:secret@db:3306/myapp # Prove it — ping db from webapp by name $ docker exec webapp ping -c 2 db PING db (172.20.0.3): 56 data bytes 64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.142 ms 64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.098 ms
```

> **Always use custom networks for multi-container setups.** The default bridge network's lack of DNS is a gotcha that catches many beginners. Get into the habit of creating a network first and attaching all related containers to it. Docker Compose (Chapter 10) does this automatically.

4. The localhost Confusion

This trips up almost everyone coming to Docker for the first time. `localhost` means different things depending on where you use it:



Reaching the host from inside a container

Sometimes you need a container to reach a service running directly on your host machine (not in another container). `localhost` won't work — use these special hostnames instead:

```
# On Windows and Mac (Docker Desktop):
host.docker.internal    # resolves to the host machine's IP

# On Linux — add this flag when running the container:
docker run --add-host=host.docker.internal:host-gateway myapp

# Example: container connecting to a database running directly on the host
docker run -e DB_HOST=host.docker.internal myapp
```

5. Connecting Containers to Networks

A container can be connected to multiple networks simultaneously, and you can connect or disconnect existing containers without restarting them:

```
# Connect a running container to an additional network
docker network connect app-network my-container

# Disconnect from a network
docker network disconnect app-network my-container

# A container can be on multiple networks at once
# (useful for a proxy that bridges a public and private network)
```

```
docker network connect public-net  proxy-container
docker network connect private-net proxy-container
```

6. Inspecting Networks

Terminal

```
# List all networks $ docker network ls NETWORK ID NAME DRIVER SCOPE a1b2c3d4e5f6 bridge bridge local d3e4f5a6b7c8 app-network bridge local b2c3d4e5f6a7 host host local # Inspect a network — see which containers are attached $ docker network inspect app-network [{ "Name": "app-network", "Driver": "bridge", "Subnet": "172.20.0.0/16", "Containers": { "a91b...": { "Name": "db", "IPv4Address": "172.20.0.3/16" }, "b82c...": { "Name": "webapp", "IPv4Address": "172.20.0.2/16" } } }] # Remove a network (all containers must be disconnected first) $ docker network rm app-network # Remove all unused networks $ docker network prune
```

7. Network Commands Reference

Command	What it does
`docker network ls`	List all networks
`docker network create NAME`	Create a new network (bridge driver by default)
`docker network inspect NAME`	Show full details including connected containers and their IPs
`docker network connect NET CONTAINER`	Attach a running container to a network
`docker network disconnect NET CONTAINER`	Detach a container from a network
`docker network rm NAME`	Delete a network (must be empty first)
`docker network prune`	Remove all networks not used by any container
`docker port CONTAINER`	Show port mappings for a container

8. Putting It Together — WordPress + MySQL

WordPress connecting to MySQL is a classic two-container setup that uses everything from this chapter: a custom network, named volumes, port mapping, and environment variables passed as connection details. This is also a preview of what Docker Compose (Chapter 10) automates.

```
# 1. Create a dedicated network
docker network create wordpress-net

# 2. Start MySQL on the network — note: NOT exposing port 3306 to host
#    (only WordPress needs to reach it, not the outside world)
docker run -d \
    --name mysql \
    --network wordpress-net \
    -v mysql-data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=rootsecret \
    -e MYSQL_DATABASE=wordpress \
    -e MYSQL_USER=wpuser \
    -e MYSQL_PASSWORD=wpsecret \
    --restart unless-stopped \
    mysql:8.0

# 3. Start WordPress on the same network
#    DB_HOST = "mysql" — the container name, resolved by Docker DNS
docker run -d \
    --name wordpress \
    --network wordpress-net \
    -p 8080:80 \
    -v wp-data:/var/www/html \
    -e WORDPRESS_DB_HOST=mysql:3306 \
    -e WORDPRESS_DB_USER=wpuser \
    -e WORDPRESS_DB_PASSWORD=wpsecret \
    -e WORDPRESS_DB_NAME=wordpress \
    --restart unless-stopped \
    wordpress:latest

# Visit http://localhost:8080 — WordPress setup wizard
# MySQL is not exposed publicly — only reachable by WordPress inside the network
```

> **Security note:** MySQL's port 3306 is not mapped to the host in this example — it's only reachable from inside `wordpress-net`. This is good practice: only expose ports that external users or your browser actually need to reach. Inter-container communication happens on the private network.

EXERCISES

1. **Explore the default bridge.** Start two containers: `docker run -d --name c1 nginx:alpine` and `docker run -d --name c2 nginx:alpine`. Run `docker network inspect bridge` and note the IP addresses assigned to each. Try `docker exec c1 ping -c 2 c2` — it will fail by name but work if you use the IP address directly. This demonstrates the default bridge's DNS limitation.

2. **Create a custom network and prove DNS works.** Run `docker network create test-net`, then start c1 and c2 with `--network test-net`. Now repeat `docker exec c1 ping -c 2 c2` — this time it should work using the name. Run `docker network inspect test-net` to see both containers listed.

3. **Understand the port binding options.** Start nginx with `-p 127.0.0.1:8080:80`. Confirm you can reach it at `http://localhost:8080`. Then stop it and restart with `-p 0.0.0.0:8080:80`. Use `docker port my-nginx` to see the difference in the binding shown. On a machine with multiple network interfaces, the first is only accessible locally while the second is accessible from the network.

4. **Build the WordPress stack.** Follow the example in Section 8 to run MySQL and WordPress together. Once WordPress is running, complete the setup wizard in your browser to create a site. Then use `docker rm -f wordpress`, recreate the WordPress container with the same command, and verify that your site and setup are still intact — demonstrating that the data is in the volume, not the container.

5. **Use docker network inspect for debugging.** With the WordPress stack running, run `docker network inspect wordpress-net`. Identify the IP addresses of both containers. Then from inside the WordPress container (`docker exec -it wordpress bash`), try `ping mysql` and `curl -s http://mysql:80` (which will fail — MySQL doesn't speak HTTP — but proves the name resolves). This is the approach you'd use to debug connectivity problems between containers.

Next: Chapter 6 — Writing Your First Dockerfile

So far you've used images built by others. Chapter 6 shows you how to build your own: writing a Dockerfile that defines exactly what goes into an image, using instructions like `FROM`, `RUN`, `COPY`, `WORKDIR`, `EXPOSE`, and `CMD`. You'll build a custom image, tag it, and run it — the foundation for the scenario chapters ahead.

Chapter 6 — Writing Your First Dockerfile

Chapter 6 — Writing Your First Dockerfile

Until now you've used images that someone else built. A Dockerfile lets you define exactly what goes into your own image — which base OS, which packages to install, which files to include, and how to start the application. Once you can write a Dockerfile, you can package any application into a portable, repeatable image that runs the same way on your laptop, your Raspberry Pi, and any server.

This chapter covers every common Dockerfile instruction with real examples, the build caching system that makes rebuilds fast, and best practices that keep your images small and secure.

1. What a Dockerfile Is

A Dockerfile is a plain text file named exactly `Dockerfile` (no extension) that lives in your project folder. Each line is an instruction that adds a layer to the image. When you run `docker build`, Docker reads the file top to bottom and executes each instruction in order.

```
# The simplest possible Dockerfile
FROM    ubuntu:24.04
RUN     apt-get update && apt-get install -y curl
CMD     ["curl", "--version"]
```

Terminal — build and run

```
# Build an image from the Dockerfile in the current directory # -t gives it a name:tag. The trailing . is the build context (current folder) $ docker build -t my-curl:1.0 . [+] Building 8.3s (6/6) FINISHED => [internal] load build context 0.0s => [1/3] FROM ubuntu:24.04 2.1s => [2/3] RUN apt-get update && apt-get install -y curl 5.8s => [3/3] CMD ["curl", "--version"] 0.0s => exporting to image 0.4s $ docker run --rm my-curl:1.0 curl 8.5.0 (x86_64-pc-linux-gnu) libcurl/8.5.0 OpenSSL/3.0.13...
```

> **The build context** is the folder you pass to `docker build` (the `.` at the end). Docker sends all files in that folder to the Docker daemon as the "context" — files you reference in `COPY` instructions must be inside it. Large build contexts slow builds down, which is why `.dockerignore` matters (Section 6).

2. Dockerfile Instructions

FROM

`FROM image:tag`

Sets the base image. Every Dockerfile must start with FROM. Choose the smallest base that includes what you need.

RUN

`RUN command`

Executes a shell command during the build and saves the result as a new layer. Used for installing packages, creating directories, etc.

COPY

`COPY src dest`

Copies files from the build context (your project folder) into the image. Prefer COPY over ADD for simple file copying.

ADD

`ADD src dest`

Like COPY but also extracts .tar files and can fetch URLs. Use COPY unless you specifically need these extra features.

WORKDIR

`WORKDIR /path`

Sets the working directory for all subsequent RUN, COPY, CMD, and ENTRYPOINT instructions. Creates the directory if it doesn't exist.

EXPOSE

`EXPOSE port`

Documents which port the container listens on. Does NOT actually publish the port — you still need -p when running. It's informational.

ENV

`ENV KEY=value`

Sets an environment variable that's available during the build and inside running containers. Can be overridden with -e at runtime.

ARG

`ARG NAME=default`

A build-time variable passed with --build-arg. Unlike ENV, ARG values are not available in running containers. Good for version numbers.

CMD

`CMD ["exec", "arg"]`

The default command to run when the container starts. Can be overridden by passing a command to docker run. Only the last CMD takes effect.

ENTRYPOINT

`ENTRYPOINT ["exec"]`

Like CMD but harder to override — docker run arguments are appended rather than replacing it. Use for containers that behave like a command.

VOLUME

`VOLUME /path`

Declares a mount point. Docker will create an anonymous volume here if no -v is specified. Signals to users that this path contains persistent data.

USER

`USER username`

Switch to a non-root user for subsequent instructions and for running the container. Best practice: don't run as root unless necessary.

3. A Real Dockerfile — Python Web App

Let's build a Dockerfile for a simple FastAPI application. This is a realistic starting point for the Python dev scenario in Chapter 8.

```
# Project structure:
# myapp/
```

```
#   Dockerfile
#   requirements.txt
#   main.py
#   .dockerignore

########################################
# Dockerfile
########################################

# 1. Start from the official slim Python image
FROM python:3.12-slim

# 2. Set the working directory inside the image
WORKDIR /app

# 3. Copy requirements first — enables layer caching (explained in Section 4)
COPY requirements.txt .

# 4. Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# 5. Copy the rest of the application code
COPY . .

# 6. Document the port (informational)
EXPOSE 8000

# 7. Switch to a non-root user for security
RUN useradd -m appuser && chown -R appuser /app
USER appuser

# 8. Start the application
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
# requirements.txt
fastapi==0.111.0
uvicorn==0.30.1
```

```
# main.py
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
```

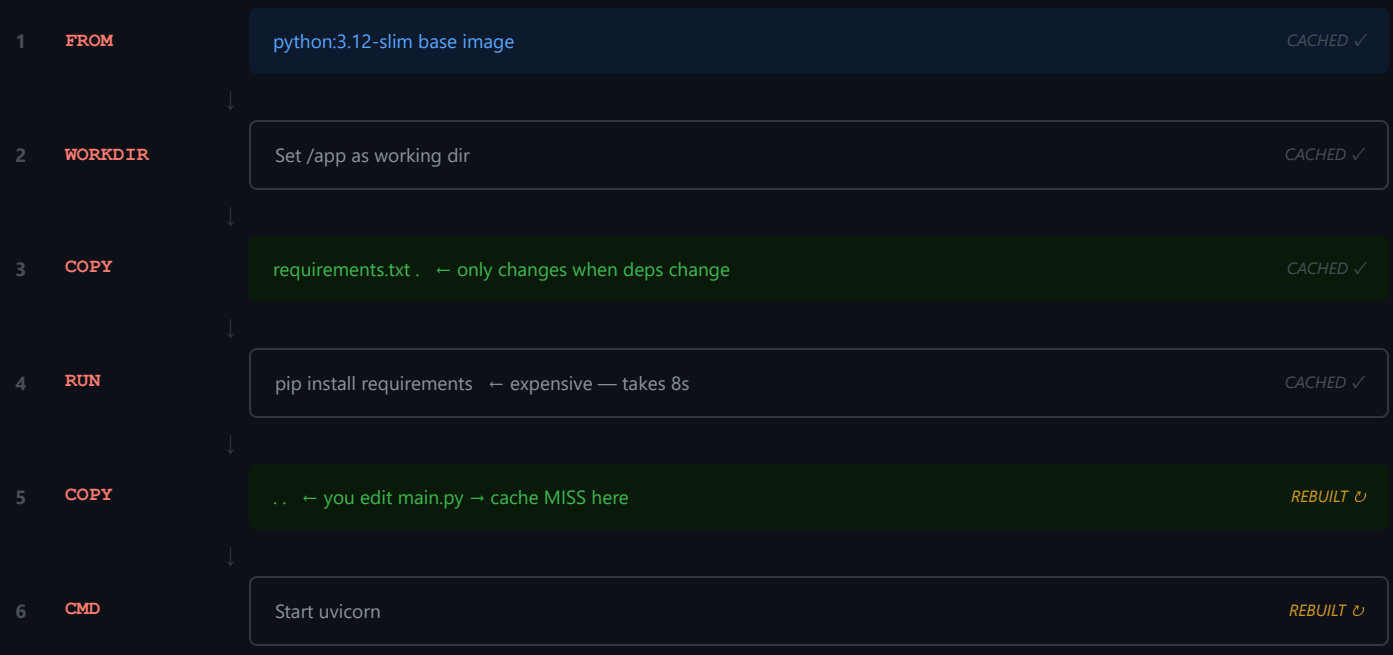
```
def root():
    return {"message": "Hello from Docker!"}
```

Terminal — build and run the app

```
$ docker build -t myapp:1.0 . [+] Building 12.4s (9/9) FINISHED => [1/5] FROM python:3.12-slim
3.2s => [2/5] WORKDIR /app 0.0s => [3/5] COPY requirements.txt . 0.0s => [4/5] RUN pip install -
-no-cache-dir -r requirements.txt 8.1s => [5/5] COPY . . 0.1s $ docker run -d -p 8000:8000 --
name myapp myapp:1.0 $ curl http://localhost:8000 {"message":"Hello from Docker!"}
```

4. Layer Caching — Making Builds Fast

Every instruction in a Dockerfile creates a layer. Docker caches each layer and reuses it if nothing above it has changed. This is why the order of your instructions matters enormously for build speed.



When you edit `main.py`, only steps 5 and 6 are rebuilt — the slow `pip install` step is reused from cache. If you had copied all files first and then run pip install, *every* code change would invalidate the pip cache and rebuild dependencies from scratch.

> **The golden rule of cache ordering:** Things that change rarely go near the top. Things that change often go near the bottom. Dependencies (`requirements.txt`, `package.json`) before source code. Config before application logic.

Terminal — second build after editing main.py

```
$ docker build -t myapp:1.1 . [+] Building 0.8s (9/9) FINISHED => [1/5] FROM python:3.12-slim
CACHED => [2/5] WORKDIR /app CACHED => [3/5] COPY requirements.txt . CACHED => [4/5] RUN pip
install --no-cache-dir -r requirements.txt CACHED => [5/5] COPY . . 0.1s => exporting to image
0.2s # 12 seconds → 0.8 seconds just by ordering instructions correctly
```

5. CMD vs ENTRYPOINT

Both define what runs when a container starts, but they behave differently when you pass extra arguments to `docker run`:

CMD

Provides default arguments. Completely replaced if you pass a command to `docker run`. Use when you want a sensible default that users can override.

```
Dockerfile: CMD ["python", "app.py"]

docker run myimage
→ runs: python app.py ✓

docker run myimage bash
→ runs: bash (CMD ignored)
```

ENTRYPOINT

Always runs. Arguments to `docker run` are appended after the ENTRYPOINT, not replacing it. Use when the container should always behave like a specific command.

```
Dockerfile: ENTRYPOINT ["python"]

docker run myimage app.py
→ runs: python app.py ✓

docker run myimage --version
→ runs: python --version ✓
```

The most flexible pattern combines both: `ENTRYPOINT` sets the executable and `CMD` provides the default argument — which can be overridden at runtime:

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

# Normal run: uvicorn main:app --host 0.0.0.0 --port 8000
docker run myapp

# Override port at runtime without touching the image
docker run myapp main:app --host 0.0.0.0 --port 9000
```

> **Always use the exec form** — `["executable", "arg1", "arg2"]` — for both CMD and ENTRYPOINT. The shell form (`CMD command arg1`) runs via `/bin/sh -c`, which means signals like SIGTERM don't reach your process. This breaks graceful shutdown, causing Docker to forcibly kill the container after a 10-second wait.

6. The .dockerignore File

Just like `.gitignore`, a `.dockerignore` file in your project root tells Docker which files to exclude from the build context. This matters for two reasons: speed (don't send large folders Docker doesn't need) and security (don't accidentally bake secrets into your image).

```
# .dockerignore
.git
.gitignore
__pycache__
*.pyc
*.pyo
.env                    # CRITICAL — never bake secrets into an image
.env.*
*.log
node_modules            # if you have any JS tooling
.venv                   # Python virtual environment
venv
tests/
docs/
*.md
Dockerfile
docker-compose.yml
```

Pattern	Matches	Why exclude
`.env`	Environment files with secrets	API keys, passwords end up in the image layer history
`.git`	Git repository data	Can be hundreds of MB, never needed in the image
`__pycache__ / *.pyc`	Python bytecode	Platform-specific, rebuilt inside the container anyway
`.venv / venv`	Virtual environment	Host packages aren't compatible inside the container — pip installs fresh
`tests/ docs/`	Test and doc folders	Not needed at runtime — keeps the image smaller
`node_modules`	npm packages	Thousands of files, often gigabytes — reinstall inside is cleaner

> **Never put secrets in a Dockerfile or image.** Even if you delete a secret in a later layer (`RUN rm .env`), it still exists in the earlier layer and can be extracted with `docker image history` or `docker save`. Pass secrets at runtime with `-e` flags or Docker secrets, never bake them in at build time.

7. Building, Tagging and Running

```
# Build with a name and tag
docker build -t myapp:1.0 .

# Build from a Dockerfile in a different location
docker build -t myapp:1.0 -f path/to/Dockerfile .

# Build with a build-time argument
docker build --build-arg PYTHON_VERSION=3.12 -t myapp:1.0 .

# Tag an existing image with a second name
docker tag myapp:1.0 myapp:latest

# List images — confirm yours is there
docker images myapp

# Run your image
docker run -d -p 8000:8000 --name myapp myapp:1.0

# See the build output in verbose mode
docker build --progress=plain -t myapp:1.0 .

# Force a fresh build ignoring all cache
docker build --no-cache -t myapp:1.0 .
```

8. Keeping Images Small

```
# ✗ BAD — three RUN commands = three layers
RUN apt-get update
RUN apt-get install -y curl
RUN rm -rf /var/lib/apt/lists/*

# ✓ GOOD — one layer, cache cleaned in the same step
RUN apt-get update \
    && apt-get install -y --no-install-recommends curl \
    && rm -rf /var/lib/apt/lists/*

# ✓ Use slim or alpine base images when possible
FROM python:3.12-slim      # ~45 MB vs ~1 GB for python:3.12
FROM python:3.12-alpine    # ~22 MB — even smaller, fewer preinstalled tools
```

```
# ✓ Use --no-install-recommends with apt to skip optional packages
RUN apt-get install -y --no-install-recommends curl

# ✓ Use --no-cache-dir with pip to skip the pip cache
RUN pip install --no-cache-dir -r requirements.txt

# ✓ Use --no-cache with apk (Alpine's package manager)
RUN apk add --no-cache curl
```

Check your image size after building. Run `docker images myapp` and look at the SIZE column. If it's unexpectedly large, use `docker image history myapp:1.0` to see which layer is responsible. The culprit is almost always a package install step that didn't clean up the package manager cache.

EXERCISES

1. **Build your first image.** Create a folder called `hello-docker` with a `Dockerfile` that starts from `alpine`, runs `echo "Image built successfully"` during the build, and has a `CMD` of `["echo", "Container started!"]`. Build it with `docker build -t hello-docker:1.0 .` and run it with `docker run --rm hello-docker:1.0`. Then override CMD by running `docker run --rm hello-docker:1.0 echo "Override works!"`.
2. **Build the FastAPI app.** Create the three files from Section 3 (`Dockerfile`, `requirements.txt`, `main.py`) in a new folder. Build the image and run the container. Visit `http://localhost:8000` in your browser (or `http://localhost:8000/docs` for the automatic FastAPI docs). Then edit `main.py` to change the return message, rebuild, and observe how only the last two steps are rebuilt thanks to caching.
3. **Prove the caching order matters.** Create a second Dockerfile that copies ALL files before running pip install (reverse the order of COPY and RUN from the example). Build it, then make a small change to `main.py` and build again. Time how long it takes compared to the correctly ordered version — pip install runs again from scratch every time.
4. **Create a .dockerignore file.** Add a `.env` file with `SECRET_KEY=topsecret` to your FastAPI project folder. Build the image without a `.dockerignore` and run `docker run --rm myapp cat /app/.env` — you'll see the secret is inside the image. Now add a `.dockerignore` excluding `.env`, rebuild, and run the same command — it should say "No such file or directory". This is why `.dockerignore` matters for security.
5. **Reduce image size.** Build the FastAPI app using `python:3.12` as the base and note the size. Then change to `python:3.12-slim` and rebuild — compare the sizes with `docker images`. Finally try `python:3.12-alpine` and resolve any dependency issues that arise (Alpine uses musl libc, which occasionally causes pip install failures for packages with C extensions). Note which base image gives the best balance of size and compatibility.

Next: Chapter 7 — Scenario: Apache Web Server
Fundamentals complete — time for real scenarios. Chapter 7 builds a practical Apache web server container you can use as a

fallback for osztromok.com: pulling the official httpd image, mounting your site files, configuring virtual hosts, and making it publicly accessible using Cloudflare Tunnel — no router port forwarding needed.

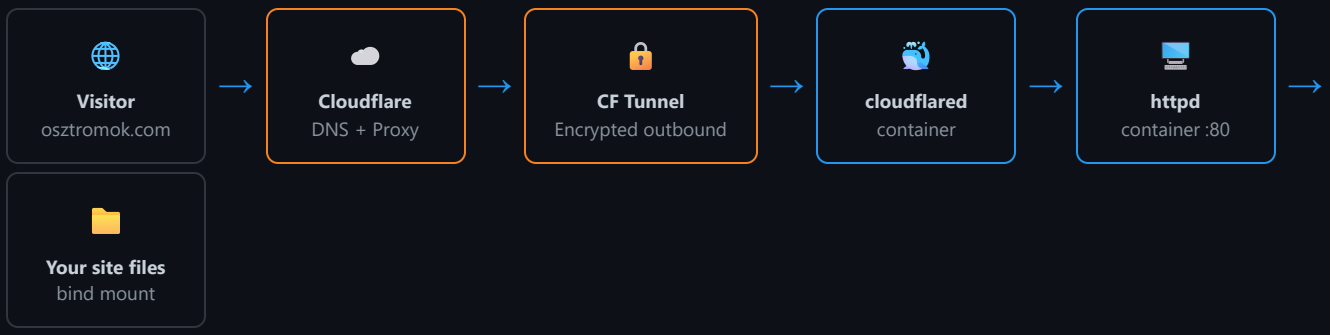
Chapter 7 — Scenario: Apache Web Server

Chapter 7 — Scenario: Apache Web Server

Time to put Docker's fundamentals to practical use. In this chapter you'll build a working Apache web server inside a Docker container that serves your actual website files — and then make it publicly accessible on the internet using Cloudflare Tunnel, with no router configuration, no port forwarding, and no static IP required.

This gives you a useful fallback for osztromok.com: if your primary server ever goes down for maintenance or has a problem, you can spin up this container on any machine — your Raspberry Pi, a laptop, a VPS — and the site stays up while you fix the real issue. The whole thing takes about ten minutes once it's configured.

What we're building



We'll run two containers on a shared Docker network: one running Apache (`httpd`) serving your site files, and one running `cloudflared` creating a secure outbound tunnel to Cloudflare's network. Visitors reach your site through Cloudflare — your machine never needs an open inbound port.

1 The Official httpd Image

The `httpd` image is the official Apache HTTP Server image on Docker Hub, maintained by the Apache project. It's production-ready, security- patched regularly, and remarkably easy to use — just mount your files at the right path and it serves them.

Terminal — pull and inspect

```
# Pull the latest stable Apache image $ docker pull httpd:2.4 2.4: Pulling from library/httpd
Digest: sha256:a5f58c... Status: Downloaded newer image for httpd:2.4 # Where does httpd serve
files from inside the container? $ docker inspect httpd:2.4 --format '{{json .Config.Env}}'
["PATH=/usr/local/apache2/bin:...", "HTTPD_VERSION=2.4.62"] # The document root is
/usr/local/apache2/htdocs # Quick test — serve the default Apache page $ docker run -d -p
```

```
8080:80 --name apache-test httpd:2.4 $ curl http://localhost:8080 <html><body><h1>It works!</h1>
</body></html> $ docker rm -f apache-test
```

> **Why tag 2.4 not latest?** The `latest` tag follows whatever the most recent release is. For a web server used as a fallback, you want predictability — pin to `2.4` and you'll always get the current 2.4.x patch release, never accidentally jump to a major version that might change behaviour.

2 Mounting Your Site Files

Apache inside the container looks for files in `/usr/local/apache2/htdocs`. We'll use a **bind mount** to point that path at your actual website folder on the host machine. Any changes you make to the files on disk are immediately visible to Apache — no restart, no rebuild.

```
# Replace the path with your actual website folder
# Windows PowerShell — use ${PWD} or an absolute path
docker run -d \
  --name osztromok-fallback \
  -p 8080:80 \
  -v C:\Users\emuba\OneDrive\My Learning\WebSites\website:/usr/local/apache2/htdocs:ro \
  httpd:2.4

# :ro makes it read-only — Apache should never write to your source files
```

Terminal — verify the site loads

```
$ docker run -d --name osztromok-fallback -p 8080:80 -v "C:\Users\emuba\OneDrive\My
Learning\WebSites\website":/usr/local/apache2/htdocs:ro httpd:2.4 9f4c2a1b8e3d... # Check it
started cleanly $ docker logs osztromok-fallback AH00558: httpd: Could not reliably determine
the server's fully qualified domain name [Mon Jun 16 10:23:01.000000 2025] [mpm_event:notice]
[pid 1:tid 1] AH00489: Apache/2.4.62 configured [Mon Jun 16 10:23:01.000001 2025]
[mpm_event:notice] [pid 1:tid 1] AH00094: Command line: 'httpd -D FOREGROUND' # The "fully
qualified domain name" warning is harmless — ignore it # Open http://localhost:8080 in your
browser to see your site
```

> **Windows path with spaces:** When your path contains spaces (as yours does — "My Learning"), always wrap the entire `-v` argument in double quotes in PowerShell: `-v "C:\Users\emuba\OneDrive\My Learning\WebSites\website":/usr/local/apache2/htdocs:ro`

3 Custom Apache Configuration (Optional)

The default `httpd.conf` works for basic file serving, but you might want to tweak it — for example to enable `.htaccess` files, set up a virtual host for your domain, or enable `mod_rewrite`. There are two ways to do this.

Option A — Mount a custom httpd.conf

Extract the default config, edit it, then mount it back in:

Terminal — extract default config

```
# Copy the default config out of the image so you have something to edit $ docker run --rm
httpd:2.4 cat /usr/local/apache2/conf/httpd.conf > my-httpd.conf # Edit my-httpd.conf, then
mount it back in $ docker run -d \ --name osztromok-fallback \ -p 8080:80 \ -v
"C:\Users\emuba\OneDrive\My Learning\WebSites\website":/usr/local/apache2/htdocs:ro \ -v
"$(pwd)/my-httpd.conf":/usr/local/apache2/conf/httpd.conf:ro \ httpd:2.4
```

Option B — Build a custom image (recommended for repeatability)

If your config changes are permanent, bake them into a custom image. This is the Dockerfile approach from Chapter 6 applied to Apache:

```
# Dockerfile for a pre-configured Apache fallback server
FROM httpd:2.4

# Copy a customised httpd.conf (you extracted and edited this)
COPY my-httpd.conf /usr/local/apache2/conf/httpd.conf

# Copy your site files directly into the image
# (alternative to bind mount — good for a truly portable image)
COPY website/ /usr/local/apache2/htdocs/
```

Key httpd.conf tweaks for osztromok.com

```
# Enable mod_rewrite (needed for clean URLs / .htaccess rewrites)
# Uncomment this line in httpd.conf:
LoadModule rewrite_module modules/mod_rewrite.so

# Allow .htaccess to override settings in your document root
# Find the <Directory "/usr/local/apache2/htdocs"> block and change:
AllowOverride None    # default — change to:
AllowOverride All

# Set your server name to suppress the FQDN warning
ServerName osztromok.com
```

```
# Optional: add a virtual host block at the bottom of httpd.conf
<VirtualHost *:80>
    ServerName    osztromok.com
    ServerAlias   www.osztromok.com
    DocumentRoot  /usr/local/apache2/htdocs
    DirectoryIndex index.html index.php
</VirtualHost>
```

> **Test your config before restarting:** Run `docker exec osztromok-fallback httpd -t` to validate the configuration syntax. Apache prints "Syntax OK" if everything is fine, or a helpful error message if something is wrong.

4 Making It Public with Cloudflare Tunnel

Your container currently answers on `localhost:8080` — only accessible from the machine it's running on. To make it reachable at `osztromok.com` from the internet, you'd normally need to open inbound ports on your router and have a static IP. Cloudflare Tunnel eliminates both requirements entirely.

The `cloudflared` process runs as its own container, opens an outbound encrypted connection to Cloudflare's edge, and tells Cloudflare to route traffic for your domain down that tunnel to your Apache container — all without a single inbound port.

Prerequisites

- Your domain (osztromok.com) must be on Cloudflare's nameservers (it already is if you're using Cloudflare Tunnel for your main server)
- You need a Cloudflare account with access to the Zero Trust dashboard
- The tunnel token you'll generate in the next step

Creating the tunnel token

> **One-time setup in the Cloudflare dashboard:**
>
> 1. Go to **Cloudflare Zero Trust → Networks → Tunnels**
> 2. Click **Add a tunnel** → choose **Cloudflared**
> 3. Give it a name (e.g. *osztromok-docker-fallback*)
> 4. Copy the token — a long string starting with `eyJ...`
> 5. Under **Public Hostnames**, add: Subdomain `@`, Domain `osztromok.com`, Service `http://osztromok-apache:80`
>
> The service URL uses the Docker container name — this is Docker's internal DNS resolving the httpd container by name on the shared network.

5 Running Both Containers Together

From Chapter 5 you know that containers on the same custom network can reach each other by name. We'll create a network, start Apache on it (named `osztromok-apache` so Cloudflare can route to it), then start cloudflared on the same network.

Terminal — full setup

```
# 1. Create a dedicated network for these two containers $ docker network create osztromok-net # 2. Start Apache — note the name matches what Cloudflare tunnel points to $ docker run -d \ --name osztromok-apache \ --network osztromok-net \ -v "C:\Users\emuba\OneDrive\My Learning\WebSites\website":/usr/local/apache2/htdocs:ro \ httpd:2.4 a3f9c2... # 3. Start cloudflared — paste your tunnel token in place of TOKEN_HERE $ docker run -d \ --name osztromok-tunnel \ --network osztromok-net \ cloudflare/cloudflared:latest tunnel \ --no-autoupdate run \ --token TOKEN_HERE b7e1d4... # 4. Check both are running $ docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Image}}" NAMES STATUS IMAGE osztromok-tunnel Up 8 seconds cloudflare/cloudflared:latest osztromok-apache Up 12 seconds httpd:2.4 # 5. Watch the tunnel connect $ docker logs osztromok-tunnel INF Starting metrics server on 127.0.0.1:2000/metrics INF Registered tunnel connection connIndex=0 ip=198.41.200.33 location=LHR INF Registered tunnel connection connIndex=1 ip=198.41.200.34 location=LHR # Once you see "Registered tunnel connection" it's live — visit osztromok.com
```

> **Don't expose port 8080 when using the tunnel.** The tunnel gives Cloudflare access to your container by name on the internal network. You don't need (and shouldn't have) `-p 8080:80` on the Apache container in production — that would open it to

everyone on your local network. Only add `-p` when you want localhost access for testing.

6 Saving It as a Startup Script

When your primary server goes down, you want to activate the fallback in seconds, not spend time reconstructing commands. Save this as a PowerShell script and keep it somewhere you can find it quickly.

```
# start-fallback.ps1
# ─────────────────────────────────────────────────────────
# osztromok.com Docker fallback web server
# Run this if the primary server is down.
# ─────────────────────────────────────────────────────────

$SITE_PATH  = "C:\Users\emuba\OneDrive\My Learning\WebSites\website"
$CF_TOKEN   = "eyJ..."   # paste your Cloudflare tunnel token here
$NETWORK    = "osztromok-net"
$APACHE_IMG = "httpd:2.4"
$CF_IMG     = "cloudflare/cloudflared:latest"

# Ensure the images are up to date
docker pull $APACHE_IMG
docker pull $CF_IMG

# Remove any leftover containers from a previous run
docker rm -f osztromok-apache osztromok-tunnel 2>&$null

# Create network (safe to run even if it already exists)
docker network create $NETWORK 2>&$null

# Start Apache
docker run -d `
  --name osztromok-apache `
  --network $NETWORK `
  --restart unless-stopped `
  -v "${SITE_PATH}":/usr/local/apache2/htdocs:ro `
  $APACHE_IMG

# Start Cloudflare Tunnel
docker run -d `
  --name osztromok-tunnel `
  --network $NETWORK `
  --restart unless-stopped `
  $CF_IMG tunnel --no-autoupdate run --token $CF_TOKEN

Write-Host "Fallback started. Waiting for tunnel to connect..."
```

```
Start-Sleep 5
docker logs --tail 5 osztromok-tunnel
```

> **--restart unless-stopped** means both containers will come back up automatically if your machine reboots while the fallback is active. To stop the fallback intentionally, run `docker stop osztromok-apache osztromok-tunnel` — after an explicit stop, the containers won't auto-restart.

Stopping the fallback when your main server is back

```
# stop-fallback.ps1
docker stop osztromok-apache osztromok-tunnel
docker rm   osztromok-apache osztromok-tunnel
docker network rm osztromok-net
Write-Host "Fallback stopped. Remember to update DNS back to your primary server."
```

7 Troubleshooting

SYMPTOM

Site shows "403 Forbidden"

FIX

Apache can't read the files. Check that `index.html` exists in your website root. On Windows you may need to share the drive in Docker Desktop Settings → Resources → File Sharing.

SYMPTOM

Tunnel logs show "failed to connect" or exits immediately

FIX

Wrong or expired token. Re-generate the token in the Cloudflare Zero Trust dashboard, delete the container, and re-run with the new token.

SYMPTOM

tunnel logs show "connection refused" reaching osztromok-apache

FIX

Container names must match exactly. The tunnel config says `http://osztromok-apache:80` — verify the Apache container has exactly that name: `docker ps --format '{{.Names}}'`

SYMPTOM

CSS/images load on homepage but 404 on other pages

FIX

mod_rewrite not enabled. Edit httpd.conf to uncomment the rewrite_module line and set AllowOverride All, then restart: `docker restart osztromok-apache`

SYMPTOM

Docker Desktop says drive sharing is blocked

FIX

In Docker Desktop → Settings → Resources → File Sharing, add `C:\Users\emuba\OneDrive`. OneDrive paths sometimes need explicit permission.

Terminal — useful diagnostic commands

```
# Check Apache is serving files (from inside the container) $ docker exec osztromok-apache ls /usr/local/apache2/htdocs # Validate Apache config syntax $ docker exec osztromok-apache httpd -t Syntax OK # Test Apache is reachable from the cloudflared container $ docker exec osztromok-tunnel wget -qO- http://osztromok-apache:80 # Follow live access logs $ docker logs -f osztromok-apache 192.168.1.1 - - [16/Jun/2025:10:45:02 +0000] "GET / HTTP/1.1" 200 4521 192.168.1.1 - - [16/Jun/2025:10:45:02 +0000] "GET /css/style.css HTTP/1.1" 200 8103
```

FALLBACK ACTIVATION CHECKLIST

- ☐ Docker Desktop is running
- ☐ Run `start-fallback.ps1`
- ☐ Wait for "Registered tunnel connection" in cloudflared logs
- ☐ Visit `https://osztromok.com` and confirm the site loads
- ☐ Check Cloudflare Zero Trust dashboard → Tunnels → your tunnel is "Healthy"
- ☐ When primary server is back: run `stop-fallback.ps1`
- ☐ Verify primary server is responding before stopping fallback

EXERCISES

1. **Local test run.** Start just the Apache container (no tunnel yet) with `-p 8080:80` and bind-mount your website folder. Visit `http://localhost:8080` and navigate around the site. Make a visible edit to `index.html` (e.g. change a heading), save the file, and refresh the browser — the change should appear instantly without restarting the container, because it reads directly from the bind mount.
2. **Inspect the access log.** While the container is running and you browse the site, run `docker logs -f osztromok-apache`. Notice how each page visit, stylesheet, image and script generates a separate log line. Try visiting a page that doesn't exist and observe the 404 log entry.
3. **Validate your httpd.conf.** Extract the default config with `docker run --rm httpd:2.4 cat /usr/local/apache2/conf/httpd.conf > my-httpd.conf`. Open the file and find the lines that need uncommenting to enable `mod_rewrite` and set `AllowOverride All`. Make those changes, mount the config, restart, and run `docker exec osztromok-apache httpd -t` to confirm "Syntax OK".

4. **Set up a tunnel (if you have Cloudflare access).** Follow Step 4 to create a tunnel in Cloudflare Zero Trust. Run both containers using the commands in Step 5, wait for the tunnel to show "Registered", and visit `https://osztromok.com` — you should see your site served from the Docker container. Check the Cloudflare dashboard to confirm the tunnel is "Healthy".
5. **Create the startup script.** Copy the `start-fallback.ps1` from Step 6 into a real file on your machine, fill in your Cloudflare token and site path, and do a full test run: stop any running fallback containers, run the script, confirm the site loads, then run `stop-fallback.ps1`. Store both scripts somewhere you'll find them under pressure — desktop, OneDrive, or a dedicated ops folder.

Next: Chapter 8 — Scenario: Python Development Environment

Now that you've seen Docker used as an operations tool, Chapter 8 switches to the developer angle: building a Python development container for FastAPI work. You'll keep dependencies isolated from your Windows system, use VS Code's Dev Containers extension to edit code inside the container, and never touch your host Python installation again.

Chapter 8 — Scenario: Python Development Environment

Chapter 8 — Scenario: Python Development Environment

Installing Python packages directly on Windows can quickly become a mess — conflicting versions between projects, packages that behave differently on Windows vs Linux, and the constant temptation to install things globally and forget about them. A Docker-based development environment solves all of this: every project gets its own clean container with exactly the right Python version and dependencies, isolated from everything else, and guaranteed to behave the same way on any machine.

In this chapter you'll build a FastAPI development environment in Docker, connect VS Code to it so you can edit, run, and debug code as if Python were installed locally — because as far as VS Code is concerned, it is.

The problem with installing Python directly on Windows

WITHOUT DOCKER

- Multiple Python versions conflict (3.10, 3.11, 3.12)
- pip installs are global unless you remember venv
- Package built for Linux doesn't work on Windows
- "It works on my machine" — breaks on the server
- Setting up a second machine means reinstalling everything
- Cleaning up a failed project is painful

WITH DOCKER

- Each project has its own Python version, pinned exactly
- Dependencies are inside the container — nothing leaks
- Linux container → same behaviour as your Linux server
- New machine: pull the image, open VS Code, done
- Delete the container → environment is completely gone
- requirements.txt is the only record you need

1 Project Structure

We'll build this in two stages: first a basic development container you can use from the terminal, then the VS Code Dev Containers integration that makes it feel like a native Python install.

```
fastapi-project/
├── .devcontainer/          # VS Code Dev Containers config
│   └── devcontainer.json
├── app/
│   ├── main.py             # FastAPI application
│   └── __init__.py
├── tests/
│   └── test_main.py
├── Dockerfile              # Development image
├── Dockerfile.prod         # Production image (leaner)
├── requirements.txt        # Runtime dependencies
├── requirements-dev.txt    # Dev-only (pytest, black, etc.)
└── .dockerignore
```

2 The Development Dockerfile

A development Dockerfile is different from a production one — it prioritises iteration speed and developer tools over a small image size.

```
# Dockerfile  (development)
FROM python:3.12-slim

# Install system tools useful during development
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
       curl git vim \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /workspace

# Install runtime dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Install dev-only dependencies (testing, linting, formatting)
COPY requirements-dev.txt .
RUN pip install --no-cache-dir -r requirements-dev.txt

# Don't copy source here — we'll bind-mount it for live editing
EXPOSE 8000

# Start uvicorn with --reload so it restarts on file changes
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]
```

```
# requirements.txt — runtime
fastapi==0.111.0
uvicorn[standard]==0.30.1
pydantic==2.7.1
```

```
# requirements-dev.txt — development only
pytest==8.2.2
pytest-asyncio==0.23.7
httpx==0.27.0        # for testing FastAPI routes
black==24.4.2        # code formatter
ruff==0.4.8          # fast linter
mypy==1.10.0         # type checker
```

```
# .dockerignore
.git
__pycache__
*.pyc
.env
.env.*
.venv
venv
.pytest_cache
.mypy_cache
.ruff_cache
*.egg-info
dist/
.devcontainer/
```

3 Build and Run (Terminal Workflow)

Before setting up VS Code integration, confirm everything works from the terminal. The key technique here is bind-mounting your source code so that uvicorn's `--reload` picks up every file change instantly.

Terminal — build and run with live reload

```
# Build the dev image $ docker build -t fastapi-dev:latest . [+] Building 14.2s (8/8) FINISHED # Run with source code bind-mounted # Windows PowerShell — use ${PWD} for the current directory $ docker run -d ` --name fastapi-dev ` -p 8000:8000 ` -v "${PWD}:/workspace" ` fastapi-dev:latest a7c3f9... # Confirm it started and auto-reload is active $ docker logs fastapi-dev INFO: Will watch for changes in these directories: ['/workspace'] INFO: Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit) INFO: Started reloader process [1] using WatchFiles INFO: Started server process [8] INFO: Application startup complete. # Edit app/main.py on your Windows machine — uvicorn sees the change WARNING: WatchFiles detected changes in 'app/main.py'. Reloading... INFO: Application startup complete.
```

> **No rebuild needed when you edit code.** Because the source is bind-mounted, changes on your Windows filesystem are immediately visible inside the container. You only rebuild the image when you change `requirements.txt` (to install new packages).

Running commands inside the dev container

Terminal — one-off commands in the container

```
# Run the test suite $ docker exec fastapi-dev pytest tests/ -v ========================= test session starts ========================= tests/test_main.py::test_root_returns_200 PASSED [100%] ========================= 1 passed in 0.42s ========================= # Format all
```

```
code with black $ docker exec fastapi-dev black app/ reformatted app/main.py All done! ✨ 🍰 ✨
# Run the linter $ docker exec fastapi-dev ruff check app/ All checks passed! # Open an
interactive shell inside the container $ docker exec -it fastapi-dev bash
root@a7c3f9:/workspace# python -c "import fastapi; print(fastapi.__version__)" 0.111.0
root@a7c3f9:/workspace# exit # Install a new package (then add it to requirements.txt) $ docker
exec fastapi-dev pip install sqlalchemy # Don't forget: also add sqlalchemy to requirements.txt
and rebuild
```

4 VS Code Dev Containers

The terminal workflow is powerful but you still edit files in Windows while running them in Linux. VS Code Dev Containers goes further: VS Code itself connects *inside* the container. Your editor, IntelliSense, debugger, linter, and terminal all run in the same Linux environment as the application. There's no mismatch — what VS Code sees is exactly what runs.

Prerequisites

- VS Code with the **Dev Containers** extension installed (`ms-vscode-remote.remote-containers`)
- Docker Desktop running
- That's it — no Python on Windows required

The devcontainer.json file

Create a `.devcontainer/devcontainer.json` file in your project. This tells VS Code how to build and configure the dev container:

```
{
  "name": "FastAPI Dev",

  // Use our Dockerfile rather than a pre-built image
  "build": {
    "dockerfile": "../Dockerfile",
    "context": ".."
  },

  // Forward port 8000 from the container to localhost:8000
  "forwardPorts": [8000],
```

```
  // VS Code extensions to install inside the container
  "customizations": {
    "vscode": {
      "extensions": [
        "ms-python.python",
        "ms-python.black-formatter",
        "charliermarsh.ruff",
        "ms-python.mypy-type-checker",
        "humao.rest-client"
      ],
      "settings": {
        "python.defaultInterpreterPath": "/usr/local/bin/python",
        "editor.formatOnSave": true,
        "editor.defaultFormatter": "ms-python.black-formatter",
        "terminal.integrated.defaultProfile.linux": "bash"
      }
    }
  },

  // Run uvicorn automatically when the container starts
  "postStartCommand": "uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload &",

  // Mount the project at /workspace (default for Dev Containers)
  "workspaceFolder": "/workspace",
  "workspaceMount": "source=${localWorkspaceFolder},target=/workspace,type=bind,consistency=ca
}
```

Opening the project in a Dev Container

> **Three ways to open:**
>
> 1. Open VS Code in your project folder → a notification appears bottom-right: *"Reopen in Container"* — click it.
> 2. Press `F1` → type *Dev Containers: Reopen in Container* → Enter.
> 3. Click the blue `><` icon in the bottom-left corner of VS Code → *Reopen in Container*.

VS Code builds the image (first time only, ~30s), starts the container, installs the extensions inside it, and reconnects. The status bar shows **Dev Container: FastAPI Dev** in blue when you're inside the container. From here, the integrated terminal is a Linux bash shell inside the container — `python`, `pip`, `pytest` all run there.

> **The first open is slow; every subsequent open is instant.** The image is cached — Docker only rebuilds it if you change the Dockerfile or requirements files. When you close VS Code and reopen the folder later, it reconnects to the existing container (or starts a fresh one in seconds if you stopped it).

5 What Works Inside the Dev Container

🔍

IntelliSense & autocomplete

The Python extension runs inside the container and sees all installed packages. Import completions, type hints, and docstrings work for FastAPI, Pydantic, and everything in requirements.txt.

🐛

Full debugger

Set breakpoints in VS Code, press F5, and the debugger attaches to the running FastAPI process inside the container. Step through code, inspect variables, evaluate expressions — as if Python were local.

🧪

Test explorer

The Testing panel discovers and runs pytest tests inside the container. Click the play button next to any test to run it individually, with pass/fail shown inline in the editor.

✨

Format on save

The devcontainer.json enables black formatting on every save. Code is automatically reformatted when you press Ctrl+S — no manual black invocation needed.

🔴

Inline linting

ruff and mypy highlight problems directly in the editor as you type. Unused imports, type mismatches, and style violations show up with squiggly underlines before you even save.

🔗

Port forwarding

Port 8000 is automatically forwarded. Visit http://localhost:8000 or http://localhost:8000/docs in your Windows browser and reach the FastAPI server running inside the container.

Configuring the debugger

Add a `.vscode/launch.json` to configure the debugger for FastAPI (this file lives on the Windows side, VS Code reads it when connecting to the container):

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "FastAPI (uvicorn)",
      "type": "debugpy",
      "request": "launch",
      "module": "uvicorn",
      "args": ["app.main:app", "--host", "0.0.0.0", "--port", "8000"],
      "jinja": true,
      "justMyCode": true
    }
  ]
}
```

> **Don't run uvicorn --reload when debugging.** The reloader forks a child process that the debugger can't attach to. Stop the auto-reload uvicorn first (`Ctrl+C` in the terminal), then press F5 to start under the debugger.

6 The Production Dockerfile

The development image has git, vim, pytest, black, and mypy — none of which belong in production. Keep a separate `Dockerfile.prod` for deploying the application:

```
# Dockerfile.prod  (production — lean and non-root)
FROM python:3.12-slim

WORKDIR /app

# Runtime deps only — no dev tools
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application source (not the whole project root)
COPY app/ ./app/

# Run as non-root
RUN useradd -m appuser && chown -R appuser /app
USER appuser

EXPOSE 8000

# No --reload in production
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000", "--workers", "2"]
```

Terminal — build and compare dev vs prod image sizes

```
$ docker build -t fastapi-dev:latest -f Dockerfile . $ docker build -t fastapi-prod:latest -f Dockerfile.prod . $ docker images fastapi-dev fastapi-prod --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}" REPOSITORY TAG SIZE fastapi-dev latest 312MB ← dev tools add ~150MB fastapi-prod latest 161MB ← only what's needed to run
```

7 Environment Variables and Secrets

FastAPI applications typically read configuration from environment variables (database URLs, secret keys, API keys). Never put these in the Dockerfile — use a `.env` file for local development and pass them at runtime.

```
# .env  (local development — add to .dockerignore and .gitignore!)
DATABASE_URL=sqlite:///./dev.db
SECRET_KEY=dev-secret-key-not-for-production
DEBUG=true
```

```
# Pass the .env file when starting the container
docker run -d \
  --name fastapi-dev \
  -p 8000:8000 \
  -v "${PWD}:/workspace" \
  --env-file .env \
  fastapi-dev:latest

# Or pass individual variables with -e
docker run -d \
  --name fastapi-dev \
  -p 8000:8000 \
  -v "${PWD}:/workspace" \
  -e DATABASE_URL=sqlite:///./dev.db \
  -e SECRET_KEY=dev-secret \
  fastapi-dev:latest
```

Read them in FastAPI using Pydantic's `BaseSettings`:

```
# app/config.py
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    database_url: str = "sqlite:///./dev.db"
    secret_key: str
    debug: bool = False

    class Config:
        env_file = ".env"

settings = Settings()
```

> **Dev Containers and .env files:** Add an `envFile` entry to `devcontainer.json` to automatically load the .env file when VS Code opens the container:
> `"runArgs": ["--env-file", "${localWorkspaceFolder}/.env"]`

EXERCISES

1. **Build the dev environment.** Create the project structure from Step 1. Write a minimal `app/main.py` with a single `GET /` route that returns `{"message": "hello"}`. Build the dev image and run it with the bind mount. Edit the

return message, save the file, and watch uvicorn's reload message appear in `docker logs -f fastapi-dev` — confirm the change shows at `http://localhost:8000` without restarting the container.

2. **Add a route and run the tests.** Add a `GET /items/{item_id}` route that returns `{"item_id": item_id}`. Write a test in `tests/test_main.py` using `httpx.AsyncClient` that calls that route and asserts the response. Run `docker exec fastapi-dev pytest tests/ -v` and confirm the test passes. Change the route to return a wrong value and watch the test fail.

3. **Set up VS Code Dev Containers.** Install the Dev Containers extension, create the `.devcontainer/devcontainer.json` from Step 4, and use *Reopen in Container*. Open `app/main.py` and confirm that typing `from fastapi import` gives IntelliSense completions for FastAPI classes. Open the integrated terminal and run `python --version` — confirm it reports Python 3.12 (Linux, not your Windows Python).

4. **Use the debugger.** Add the `.vscode/launch.json` from Step 5. Stop the auto-running uvicorn in the Dev Container terminal (Ctrl+C), then press F5 to start under the debugger. Set a breakpoint on the return line of your `/` route, visit `http://localhost:8000` in your browser, and confirm VS Code pauses at the breakpoint and shows the local variables. Press F5 to continue.

5. **Compare dev and prod image sizes.** Build both `Dockerfile` and `Dockerfile.prod`. Run `docker image history fastapi-prod:latest` and confirm there are no pytest, black, ruff, or mypy layers — only the runtime packages. Note the size difference. Consider: which other developer tools might you be tempted to leave in a production image by accident?

Next: Chapter 9 — Scenario: Working with the Claude API

Chapter 9 builds a dedicated Claude API development container: a Python environment pre-configured for working with the Anthropic SDK, with safe `.env` key handling so your API key never ends up baked into an image or committed to git, plus a simple script structure for running Claude-powered experiments from the terminal or VS Code.

Chapter 9 — Scenario: Working with the Claude API

Chapter 9 — Scenario: Working with the Claude API

You already have experience working with Claude through Claude Code. The Claude API lets you go further — calling Claude programmatically from your own Python scripts, building it into applications, automating tasks, and experimenting with prompts in a controlled environment.

In this chapter you'll build a dedicated Docker container for Claude API development: a clean Python environment with the Anthropic SDK pre-installed, your API key handled safely through environment variables (never baked into an image), and a script structure you can extend for any Claude-powered project.

> **What you need:** An Anthropic API key from *console.anthropic.com*. If you don't have one yet, you can follow this chapter and substitute a placeholder key — the container setup and code structure are the same. API keys start with `sk-ant-api03-...`

1 Keeping Your API Key Safe

An API key is a password. Leak it and anyone who finds it can use your Anthropic account, run up a bill, and access anything you've built. The Docker workflow makes it easy to handle keys correctly — but also easy to get wrong if you're not paying attention.

✅	**.env file on your local machine** — key lives here, never leaves this machine	SAFE
✅	**docker run --env-file .env** — injects key as environment variable at runtime	SAFE
✅	**os.environ["ANTHROPIC_API_KEY"]** — read in Python at runtime, never stored	SAFE
❌	**ENV ANTHROPIC_API_KEY=sk-ant-... in Dockerfile** — baked into every layer, visible in image history	DANGER
❌	**api_key = "sk-ant-..." hardcoded in .py file** — one accidental git push and it's public	DANGER
❌	**.env file committed to git** — git history preserves it even after deletion	DANGER

> **If you ever accidentally expose a key:** Immediately go to *console.anthropic.com → API Keys* and revoke it. Create a new key. Revoking takes effect immediately — the leaked key stops working the moment you revoke it.

2 Project Structure

```
claude-api-dev/ ├── Dockerfile ├── requirements.txt ├── .env # ANTHROPIC_API_KEY=sk-ant-... — NEVER commit this ├── .env.example # ANTHROPIC_API_KEY=your-key-here — safe to commit ├──
```

```
.dockerignore ├── .gitignore └── scripts/ ├── hello_claude.py # Basic API call — start here ├──
conversation.py # Multi-turn conversation loop ├── streaming.py # Streaming responses └──
batch_process.py # Process a list of inputs
```

```
# .gitignore  (and .dockerignore — add .env to both!)
.env
.env.*
!.env.example      # allow the example file through
__pycache__
*.pyc
.venv
venv
```

```
# .env  (your actual key — stays on your machine)
ANTHROPIC_API_KEY=sk-ant-api03-your-real-key-here

# .env.example  (placeholder — safe to commit to git)
ANTHROPIC_API_KEY=your-anthropic-api-key-here
```

3 Dockerfile and Requirements

```
# Dockerfile
FROM python:3.12-slim

WORKDIR /workspace

# Install the Anthropic SDK and supporting packages
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Source code will be bind-mounted — no COPY here
# API key will be injected at runtime — no ENV here

CMD ["python", "-c", "print('Claude API dev container ready.')"]
```

```
# requirements.txt
anthropic==0.28.0          # Anthropic SDK (pin the version)
python-dotenv==1.0.1       # load .env files in scripts
rich==13.7.1               # pretty terminal output
httpx==0.27.0              # async HTTP (already a dep of anthropic)
```

Terminal — build the image

```
$ docker build -t claude-dev:latest . [+] Building 9.4s (6/6) FINISHED => [1/3] FROM
python:3.12-slim 2.8s => [2/3] COPY requirements.txt . 0.0s => [3/3] RUN pip install --no-cache-
dir -r requirements.txt 6.1s # Confirm the SDK is installed $ docker run --rm claude-dev python
-c "import anthropic; print(anthropic.__version__)" 0.28.0
```

4 Your First API Call

```python
# scripts/hello_claude.py
import os
import anthropic
from dotenv import load_dotenv

# load_dotenv() reads .env from the current working directory
# If the key is already in the environment (injected by Docker), this is a no-op
load_dotenv()

client = anthropic.Anthropic(
    api_key=os.environ["ANTHROPIC_API_KEY"]
)

message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    messages=[
        {
            "role": "user",
            "content": "Explain what Docker is in exactly two sentences."
        }
    ]
)

print(message.content[0].text)
```

Terminal — run with API key injected safely

```
# --env-file injects the key at runtime — it is NEVER stored in the image $ docker run --rm ` --
env-file .env ` -v "${PWD}/scripts:/workspace/scripts" ` claude-dev ` python
scripts/hello_claude.py Docker is a platform that packages applications and their dependencies
into lightweight, portable containers that run consistently across any environment. Unlike
virtual machines, containers share the host operating system kernel, making them faster to start
and more efficient with resources.
```

Understanding the API response object

```
{ "id": "msg_01XFDUDYJgAACzvnptvVoYEL", "type": "message", "role": "assistant", "model": "claude-sonnet-4-6", "content": [ { "type": "text", "text": "Docker is a platform that..." ← message.content[0].text } ], "stop_reason": "end_turn", "usage": { "input_tokens": 18, ← what you sent "output_tokens": 67 ← what Claude generated (you pay for both) } }
```

Always log token usage during development. Add `print(f"Tokens: {message.usage.input_tokens} in / {message.usage.output_tokens} out")` to your scripts while experimenting. It's easy to accidentally write a loop that calls the API hundreds of times and run up a large bill — watching the token count keeps you aware of what you're spending.

5 Choosing a Model

Model ID	Best for	Speed	Cost
`claude-haiku-4-5-20251001`	High-volume tasks, classification, summarisation, simple Q&A	**Fastest**	**Lowest**
`claude-sonnet-4-6`	General development, code generation, analysis, most everyday tasks	Fast	Mid
`claude-opus-4-8`	Complex reasoning, nuanced writing, hard problems requiring deep thought	Slower	Higher

Development tip: Use `claude-haiku-4-5-20251001` while building and testing your scripts — it's fast and cheap, so iteration is painless. Switch to `claude-sonnet-4-6` or `claude-opus-4-8` only when you need more capability. A single model string at the top of your script makes switching trivial.

6 Useful Patterns

System prompt

Sets Claude's persona, instructions, or constraints for the entire conversation. Passed separately from the user message.

Use for: giving Claude a role, restricting output format, injecting context

Multi-turn conversation

Pass the full message history on every call — the API is stateless. Your code accumulates the messages list.

Use for: chatbots, interactive tools, anything needing context from earlier turns

Streaming

Receive tokens as they're generated instead of waiting for the full response. Dramatically better UX for long outputs.

Use for: any user-facing interface, long responses, progress feedback

Batch processing

Loop through a list of inputs, call the API for each, collect results. Add a delay or rate-limit check between calls.

Use for: processing files, summarising a list of articles, classifying records

System prompt

```python
# Add a system parameter to messages.create()
message = client.messages.create(
    model="claude-sonnet-4-6",
    max_tokens=1024,
    system="""You are a concise technical writer.
Always respond in plain text, no markdown.
Keep answers under 100 words unless asked for more.""",
    messages=[
        {"role": "user", "content": "What is a Docker volume?"}
    ]
)
```

Multi-turn conversation

```python
# scripts/conversation.py
import os, anthropic
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"])

history = []   # accumulate the full conversation

print("Chat with Claude (type 'quit' to exit)\n")

while True:
    user_input = input("You: ").strip()
    if user_input.lower() in ("quit", "exit", ""):
        break

    history.append({"role": "user", "content": user_input})

    response = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=1024,
        messages=history   # send the full history every time
    )

    reply = response.content[0].text
    history.append({"role": "assistant", "content": reply})

    print(f"\nClaude: {reply}\n")
```

```
# -it is essential for interactive input (stdin attached) $ docker run -it --rm ` --env-file
.env ` -v "${PWD}/scripts:/workspace/scripts" ` claude-dev ` python scripts/conversation.py Chat
with Claude (type 'quit' to exit) You: What is the capital of Hungary? Claude: Budapest is the
capital of Hungary. You: And what language do they speak there? Claude: They speak Hungarian
(Magyar) — Claude remembers context from the previous turn. You: quit
```

Streaming responses

```python
# scripts/streaming.py
import os, anthropic
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"])

print("Claude: ", end="", flush=True)

# stream=True returns a context manager — tokens arrive as they're generated
with client.messages.stream(
    model="claude-sonnet-4-6",
    max_tokens=512,
    messages=[{"role": "user", "content": "Write a short poem about containers."}]
) as stream:
    for text in stream.text_stream():
        print(text, end="", flush=True)

print()   # newline after streaming finishes
```

Batch processing a list of inputs

```python
# scripts/batch_process.py
import os, time, anthropic
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"])

# Example: classify a list of sentences as positive/negative
sentences = [
    "I love learning new things every day.",
    "This is incredibly frustrating.",
    "Docker makes deployment so much simpler.",
    "I can't get this to work no matter what I try.",
]
```

```python
for i, sentence in enumerate(sentences, 1):
    response = client.messages.create(
        model="claude-haiku-4-5-20251001",    # use Haiku for high-volume batch work
        max_tokens=10,
        system="Classify the sentiment as exactly one word: positive, negative, or neutral.",
        messages=[{"role": "user", "content": sentence}]
    )
    sentiment = response.content[0].text.strip()
    print(f"[{i}/{len(sentences)}] {sentiment:10s} — {sentence}")
    time.sleep(0.5)    # gentle rate limiting between calls
```

Terminal — run the batch script

```
$ docker run --rm ` --env-file .env ` -v "${PWD}/scripts:/workspace/scripts" ` claude-dev `
python scripts/batch_process.py [1/4] positive — I love learning new things every day. [2/4]
negative — This is incredibly frustrating. [3/4] positive — Docker makes deployment so much
simpler. [4/4] negative — I can't get this to work no matter what I try.
```

7 Persistent Shell for Experimentation

When you're actively experimenting — trying different prompts, inspecting response objects, testing ideas — it's more efficient to keep a container running and exec into it rather than starting a fresh container for every script run.

Terminal — long-lived dev container

```
# Start a long-lived container (no --rm, no specific command) $ docker run -d ` --name claude-
dev ` --env-file .env ` -v "${PWD}/scripts:/workspace/scripts" ` claude-dev ` sleep infinity
3a9b1c... # Run any script without starting a new container each time $ docker exec claude-dev
python scripts/hello_claude.py $ docker exec claude-dev python scripts/streaming.py # Open an
interactive Python REPL inside the container $ docker exec -it claude-dev python >>> import
anthropic, os >>> client = anthropic.Anthropic(api_key=os.environ["ANTHROPIC_API_KEY"]) >>> r =
client.messages.create(model="claude-haiku-4-5-20251001", max_tokens=50, messages=
[{"role":"user","content":"hi"}]) >>> r.content[0].text 'Hello! How can I help you today?' >>>
r.usage Usage(input_tokens=8, output_tokens=12) # When done experimenting, stop the container $
docker stop claude-dev && docker rm claude-dev
```

> **save-output.ps1 — capture Claude's response to a file:**
>
> `docker exec claude-dev python scripts/hello_claude.py | Out-File -Encoding utf8 output.txt`
>
> Useful when generating long content you want to review or use elsewhere.

8 Error Handling

```python
import os, anthropic
from dotenv import load_dotenv

load_dotenv()

# Fail early with a clear message if the key isn't set
api_key = os.environ.get("ANTHROPIC_API_KEY")
if not api_key:
    raise EnvironmentError("ANTHROPIC_API_KEY not set. Add it to .env or pass with --env-file.

client = anthropic.Anthropic(api_key=api_key)

try:
    message = client.messages.create(
        model="claude-sonnet-4-6",
        max_tokens=1024,
        messages=[{"role": "user", "content": "Hello!"}]
    )
    print(message.content[0].text)

except anthropic.AuthenticationError:
    print("Invalid API key — check your .env file")

except anthropic.RateLimitError:
    print("Rate limit hit — slow down or upgrade your plan")

except anthropic.APIConnectionError:
    print("No connection to Anthropic API — check your network")

except anthropic.APIStatusError as e:
    print(f"API error {e.status_code}: {e.message}")
```

EXERCISES

1. **Run hello_claude.py.** Create all the files from Steps 2–4 (Dockerfile, requirements.txt, .env with your real key, .dockerignore, .gitignore, and scripts/hello_claude.py). Build the image and run the script with `--env-file .env`. Confirm Claude's response appears in the terminal. Then run `docker run --rm claude-dev env | grep ANTHROPIC` to see the key is injected — and run `docker image history claude-dev:latest` to confirm it is NOT baked into any image layer.

2. **Experiment with models.** Edit hello_claude.py to use `claude-haiku-4-5-20251001` instead of Sonnet. Add a line to print the token usage after the response. Run it with the same prompt and note the token counts. Try the same with Sonnet. Is the response quality noticeably different for a simple question? Try a harder question where Haiku struggles.

3. **Build the conversation loop.** Create scripts/conversation.py from Step 6 and run it with `docker run -it`. Have a multi-turn conversation about something you're learning (Japanese, Docker, Python — anything). After you quit, notice that the next time you run it, Claude has no memory of the previous session — because history was stored in a Python list that was discarded when the container exited. Think about how you'd save that history to a file to persist it across runs.
4. **Streaming vs non-streaming.** Create scripts/streaming.py and run it. Notice how the response appears word-by-word instead of all at once after a delay. Now change the prompt to ask for something long (e.g. "Write a detailed explanation of how Docker networking works"). The difference in perceived responsiveness between streaming and non-streaming is most obvious with longer responses.
5. **Build something useful for your own work.** Write a new script that uses Claude to help with something you actually do. Examples: summarise a webpage (paste in text), generate a lesson plan outline, translate a short paragraph to Hungarian or Japanese, review a piece of Python code you're working on, or generate test cases for a function. The goal is to go from "API example" to "tool I'd actually use" — even if it's simple.

Next: Chapter 10 — Docker Compose: Running Multi-Container Apps

Throughout this course you've been starting containers one by one with `docker run`. Chapter 10 introduces Docker Compose — a single YAML file that defines your entire multi-container stack (web server, database, cache, tunnel) and brings it all up with one command. It's the natural endpoint of the course, and the gateway to everything beyond: production deployments, CI pipelines, and the intermediate Docker course.

Chapter 10 — Docker Compose: Running Multi-Container Apps

Chapter 10 — Docker Compose: Running Multi-Container Apps

Every multi-container scenario in this course — Apache plus Cloudflare Tunnel, FastAPI plus a database, Claude API scripts with supporting services — required you to type several `docker run` commands, remember the right flags, create networks manually, and tear everything down in the right order. Docker Compose replaces all of that with a single YAML file and two commands: `docker compose up` and `docker compose down`.

Compose isn't just a convenience wrapper. It gives your infrastructure a permanent written record, makes it reproducible on any machine, and becomes the foundation for everything more advanced — Swarm, Kubernetes, CI pipelines. It's the natural final step of this course.

1. Before and After Compose

WITHOUT COMPOSE — CHAPTER 7 STARTUP

```
docker network create osztromok-net docker run -d \ --name osztromok-apache \ --network osztromok-net \ -v "C:\...\website":\ /usr/local/apache2/htdocs:ro \ httpd:2.4 docker run -d \ --name osztromok-tunnel \ --network osztromok-net \ cloudflare/cloudflared:latest \ tunnel --no-autoupdate run \ --token TOKEN_HERE # Remember to stop both, in order, to clean up
```

WITH COMPOSE — SAME THING

```
docker compose up -d # That's it. Compose reads docker-compose.yml, # creates the network, starts both containers # in the right order, names everything # consistently, and remembers every flag. # To stop and clean up everything: docker compose down
```

2 Anatomy of a Compose File

A Compose file is a YAML file named `docker-compose.yml` (or `compose.yml`) that lives in your project root. It has three top-level sections: `services`, `volumes`, and `networks`.

```yaml
# docker-compose.yml — annotated skeleton

services:                          # one entry per container

  web:                             # service name (used for DNS inside the network)
    image: nginx:1.27              # use an existing image...
    build: ./app                   # ...or build from a Dockerfile in ./app
    ports:
      - "8080:80"                  # host:container port mapping
    volumes:
      - ./site:/usr/share/nginx/html:ro    # bind mount
```

```
      - app-data:/var/lib/data            # named volume
    environment:                # env vars (avoid secrets here)
      - APP_ENV=production
    env_file:                    # load from a .env file (use for secrets)
      - .env
    networks:
      - app-net
    restart: unless-stopped     # restart policy
    depends_on:                # start db before web
      - db

  db:
    image: mysql:8.0
    volumes:
      - db-data:/var/lib/mysql  # persist database across restarts
    environment:
      - MYSQL_ROOT_PASSWORD=secret
      - MYSQL_DATABASE=myapp
    networks:
      - app-net

volumes:                        # declare named volumes
  app-data:
  db-data:

networks:                       # declare custom networks
  app-net:
```

> **Services communicate by name.** Inside a Compose network, every service is reachable at its service name as a hostname. The `web` service can connect to MySQL at `db:3306` — no IP addresses, no manual network creation, Compose handles all of it.

3 Example: osztromok.com Fallback Stack

Let's rewrite the Chapter 7 fallback (Apache + Cloudflare Tunnel) as a Compose file. This replaces `start-fallback.ps1` entirely.

```
# docker-compose.yml  (osztromok fallback)

services:

  apache:
    image: httpd:2.4
    container_name: osztromok-apache
```

```
    volumes:
      - C:\Users\emuba\OneDrive\My Learning\WebSites\website:/usr/local/apache2/htdocs:ro
    networks:
      - osztromok-net
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost/"]
      interval: 30s
      timeout:  5s
      retries:  3

  tunnel:
    image: cloudflare/cloudflared:latest
    container_name: osztromok-tunnel
    command: tunnel --no-autoupdate run --token ${CF_TUNNEL_TOKEN}
    env_file:
      - .env
    networks:
      - osztromok-net
    restart: unless-stopped
    depends_on:
      apache:
        condition: service_healthy  # wait until Apache passes healthcheck

networks:
  osztromok-net:
```

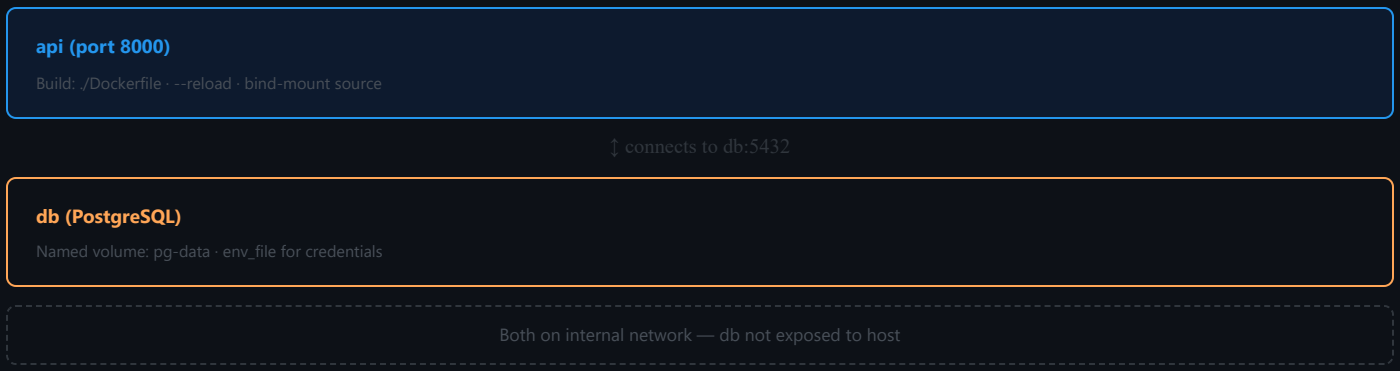
```
# .env  (same file as before — Compose reads it automatically)
CF_TUNNEL_TOKEN=eyJ...
```

Terminal — up and down

```
# Start the whole stack in the background $ docker compose up -d [+] Running 3/3 ✓ Network
osztromok-net Created ✓ Container osztromok-apache Started ✓ Container osztromok-tunnel Started
# Check both containers are running $ docker compose ps NAME IMAGE STATUS osztromok-apache
httpd:2.4 Up (healthy) osztromok-tunnel cloudflare/cloudflared:latest Up # Follow logs from both
containers at once $ docker compose logs -f apache | AH00094: Command line: 'httpd -D
FOREGROUND' tunnel | INF Registered tunnel connection connIndex=0 # Stop and remove containers +
network (volumes preserved by default) $ docker compose down [+] Running 3/3 ✓ Container
osztromok-tunnel Removed ✓ Container osztromok-apache Removed ✓ Network osztromok-net Removed
```

4 Example: FastAPI + PostgreSQL Dev Stack

This is the natural evolution of the Chapter 8 Python dev environment — adding a real database so you can develop with something closer to production.




```yaml
# docker-compose.yml  (FastAPI + PostgreSQL)

services:

  api:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/workspace          # bind-mount source for --reload
    env_file:
      - .env
    environment:
      - DATABASE_URL=postgresql://appuser:${DB_PASSWORD}@db:5432/appdb
    networks:
      - app-net
    depends_on:
      db:
        condition: service_healthy
    restart: on-failure

  db:
    image: postgres:16-alpine
    volumes:
      - pg-data:/var/lib/postgresql/data
    env_file:
      - .env
    environment:
      - POSTGRES_USER=appuser
      - POSTGRES_PASSWORD=${DB_PASSWORD}
      - POSTGRES_DB=appdb
    networks:
      - app-net
    healthcheck:
```

```
      test: ["CMD-SHELL", "pg_isready -U appuser -d appdb"]
      interval: 5s
      timeout:  3s
      retries:  5
    # db is NOT exposed with ports: — only reachable inside app-net

volumes:
  pg-data:

networks:
  app-net:
```

```
# .env
DB_PASSWORD=localdevpassword123
```

Terminal — dev stack workflow

```
# First time: build the api image and pull postgres $ docker compose up -d --build [+] Building 14.2s api image... [+] Running 4/4 ✓ Network app-net Created ✓ Volume pg-data Created ✓ Container db Started (healthy after 8s) ✓ Container api Started # Run database migrations inside the api container $ docker compose exec api alembic upgrade head # Connect to postgres directly for inspection $ docker compose exec db psql -U appuser -d appdb appdb=# \dt # Rebuild only the api image (after changing Dockerfile or requirements) $ docker compose up -d --build api # Stop stack but KEEP the pg-data volume (data survives) $ docker compose down # Stop stack AND delete volumes (fresh database next time) $ docker compose down -v
```

> **depends_on with condition: service_healthy** waits for the healthcheck to pass before starting the dependent service. Without this, the API container starts immediately, tries to connect to PostgreSQL before it's ready, crashes, and Docker restarts it — which usually works eventually but produces confusing error logs. The healthcheck approach is cleaner.

5 Compose Command Reference

Command	What it does
`docker compose up -d`	Create and start all services in the background (`-d` = detach)
`docker compose up -d --build`	Same but rebuild images first — use after changing Dockerfile or requirements
`docker compose down`	Stop and remove containers and networks. Volumes are preserved.
`docker compose down -v`	As above, also delete named volumes. Use when you want a completely fresh state.
`docker compose ps`	List containers in this Compose project with their status and ports

Command	What it does
`docker compose logs -f`	Follow logs from all services. Add a service name to filter: `logs -f api`
`docker compose exec api bash`	Open a shell inside the running `api` service container
`docker compose exec db psql ...`	Run a command inside a running service (here: psql in the db container)
`docker compose restart api`	Restart just one service without touching the others
`docker compose stop`	Stop containers without removing them — faster to restart than `down/up`
`docker compose pull`	Pull latest versions of all images defined with `image:`
`docker compose config`	Validate and print the resolved Compose file — useful for debugging variable substitution

6 Override Files: dev vs prod

Compose supports layering files with `-f`. A common pattern is a base `docker-compose.yml` with shared config, and a `docker-compose.override.yml` that Compose applies automatically in development:

```yaml
# docker-compose.yml  (shared base — commit this)
services:
  api:
    image: myapp:latest
    networks:
      - app-net
  db:
    image: postgres:16-alpine
    volumes:
      - pg-data:/var/lib/postgresql/data
    networks:
      - app-net
volumes:
  pg-data:
networks:
  app-net:
```

```yaml
# docker-compose.override.yml  (dev additions — applied automatically)
services:
  api:
    build: .              # build from source in dev
    ports:
      - "8000:8000"
    volumes:
      - .:/workspace      # bind-mount for live reload
```

```
    command: uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload
  db:
    ports:
      - "5432:5432"      # expose postgres to host only in dev
```

```
# Production — explicit file selection (no override applied)
docker compose -f docker-compose.yml up -d

# Development — automatic (override.yml applied on top)
docker compose up -d
```

> **docker compose config** shows you exactly what the merged result looks like before you run it — essential for debugging when you have multiple override files or complex variable substitution.

7 Profiles: Optional Services

Profiles let you define services that are off by default and only started when explicitly requested — useful for debugging tools, admin UIs, or services only needed occasionally:

```
services:

  api:
    image: myapp:latest     # always started

  db:
    image: postgres:16-alpine  # always started

  adminer:                    # web-based DB admin UI
    image: adminer
    ports:
      - "8080:8080"
    profiles:
      - tools              # only start with --profile tools
```

Terminal — profiles

```
# Normal start — adminer NOT started $ docker compose up -d ✓ Container api Started ✓ Container db Started # Start with tools profile — adminer now included $ docker compose --profile tools up -d ✓ Container api Started ✓ Container db Started ✓ Container adminer Started
```

EXERCISES

1. **Convert the Chapter 7 fallback to Compose.** Create a folder called `osztromok-fallback` and write the `docker-compose.yml` from Step 3. Add a `.env` file with your Cloudflare tunnel token as `CF_TUNNEL_TOKEN`. Run `docker compose up -d` and confirm both containers appear in `docker compose ps`. Visit osztromok.com to confirm the site loads. Then run `docker compose down` and confirm both containers and the network are gone. Compare how much simpler this is than the PowerShell startup script.

2. **Build the FastAPI + PostgreSQL stack.** Set up the full stack from Step 4. Start it with `docker compose up -d --build` and watch it wait for PostgreSQL's healthcheck before starting the API. Run `docker compose logs -f` and observe the interleaved log output from both services. Connect to the database with `docker compose exec db psql -U appuser -d appdb` and run `\conninfo` to confirm you're inside the container's postgres. Stop and restart — confirm the volume persists (data survives). Then run `docker compose down -v` and confirm the volume is gone.

3. **Use docker compose exec for development tasks.** With the FastAPI stack running, use `docker compose exec api` to run each of the following without opening a separate terminal: `pip list` to see installed packages, `python -c "import fastapi; print(fastapi.__version__)"`, and `ruff check app/`. Notice how Compose knows which container to target just from the service name.

4. **Try override files.** Split the FastAPI compose file into a base `docker-compose.yml` (no ports, no volumes, image-based) and a `docker-compose.override.yml` (adds ports, bind-mount, build, --reload). Run `docker compose config` and read the merged output to confirm the override is being applied. Then run `docker compose -f docker-compose.yml config` to see what production would look like without the override.

5. **Add an optional service with profiles.** Add `adminer` to your FastAPI stack under a `tools` profile. Run `docker compose up -d` (without the profile) and confirm adminer isn't started. Then run `docker compose --profile tools up -d` and visit `http://localhost:8080` — log in to your PostgreSQL database using the credentials from your .env file. This is a useful pattern for admin tools you don't want running all the time.

Docker for Beginners — Complete

You've covered everything from containers vs VMs all the way to multi-service Compose stacks. You can now pull and run images, manage volumes and networks, write Dockerfiles, build custom images, run a real web server with a Cloudflare Tunnel fallback, develop Python apps in a containerised environment, call the Claude API safely, and orchestrate multi-container stacks with a single YAML file.

What's next: A Docker Intermediate course would cover multi-stage builds, Docker Swarm, container registries, CI/CD pipelines with Docker, and production security hardening. When you're ready, ask for a course outline.