DOCKER HANDBOOK

A Step-by-Step Handbook for Dockerfiles, Compose, Security, and Swarm/Kubernetes

Prepared By

Shreyas Ladhe

shreyasl9819@gmail.com

shreyas-shack.vercel.app

Chapter 1: Install and Configure Docker on Your Local Machine

Docker has transformed how developers build, ship, and run applications. Before diving into containerized workflows, the first step is getting Docker installed and properly configured on your local machine. This chapter will walk you through installing Docker across major operating systems, verifying the installation, configuring key settings, and preparing your environment for smooth usage.

1.1 Understanding Docker Installation

Docker consists of two main components:

- **Docker Engine**: The runtime that builds and runs containers.
- **Docker CLI (Command Line Interface)**: The tool you use to interact with Docker Engine.

On desktops (Windows, macOS), Docker is installed via **Docker Desktop**, which bundles the engine, CLI, and a lightweight VM. On Linux, you typically install Docker Engine directly.

1.2 Installation by Operating System

Windows

1. Download Docker Desktop from Docker's official website.
2. Run the installer and follow the setup wizard.
3. Enable **WSL 2** (Windows Subsystem for Linux) if not already installed, as Docker Desktop relies on it for better performance.
4. After installation, launch Docker Desktop from the Start Menu and ensure it starts successfully.

Verification:

Open PowerShell or Command Prompt and run:

```
docker --version
docker run hello-world
```

MacOS

1. Download the `.dmg` file for Docker Desktop from Docker's site.
2. Drag and drop Docker into the Applications folder.
3. Start Docker Desktop and grant necessary permissions.

Verification:
Run the same commands as above in the Terminal.

Linux (Ubuntu example)

```
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg lsb-release
sudo mkdir -p /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 \
  $(lsb_release -cs) stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Post-install steps:

Add your user to the `docker` group:

```
sudo usermod -aG docker $USER
```

- (Log out and log back in for this to take effect.)

Verification:

```
docker run hello-world
```

1.3 Configuring Docker

After installation, configuration ensures Docker integrates seamlessly into your workflow.

- **Startup Settings:**
  On Docker Desktop, configure whether Docker launches at startup. This can save time if

you use it frequently.

- **Resource Allocation (CPU, RAM, Disk):**
  Containers run inside a lightweight VM on macOS and Windows. Use Docker Desktop's preferences to adjust CPU, memory, and disk space to match your workloads.
- **Proxies and Network Settings:**
  If working behind a corporate proxy, configure proxy settings in Docker Desktop or `/etc/docker/daemon.json` on Linux.
- **Logging:**
  By default, Docker uses the JSON logging driver. You can configure drivers globally in the daemon config file.

1.4 Verifying Installation and Basic Commands

Test your installation with some basic commands:

Check Docker version:

```
docker --version
```

List running containers (should be empty initially):

```
docker ps
```

Run a test container:

```
docker run -it ubuntu bash
```

- This pulls the Ubuntu image if not present, starts a container, and drops you into a shell inside it.

Remove unused containers/images:

```
docker system prune
```

1.5 Troubleshooting Installation Issues

- **Docker not starting:**
  Restart Docker Desktop or run `sudo systemctl start docker` on Linux.
- **Permission issues on Linux:**
  Ensure your user is in the `docker` group.

- **Networking issues:**
  Reset Docker's network settings, or restart the Docker daemon.

1.6 Preparing for Next Steps

At this stage, Docker is installed and ready to use. Your local machine is configured with the appropriate permissions, and you can run test containers successfully. This foundation is critical before moving to more advanced tasks like pulling containerized applications, building custom images, or orchestrating containers.

In the next chapter, we will dive into **Pulling and Running Containerized Applications using Docker CLI**, where you'll learn how to find images, run them interactively or in detached mode, and understand container lifecycle management.

Key Takeaways

- ☐ Install Docker Desktop on Windows/macOS, Docker Engine on Linux.
- ☐ Verify installation with `docker run hello-world`.
- ☐ Configure resources, proxies, and logging for smooth operation.
- ☐ Test basic commands to ensure Docker works properly.

Chapter 2: Pull and Run Containerized Applications using Docker CLI

Once Docker is installed and configured, the next logical step is to use it to run real applications. This chapter will cover how to pull container images from a registry, run them interactively or in detached mode, manage their lifecycle, and understand what happens under the hood when you start a container.

2.1 What Does “Pulling an Image” Mean?

Docker containers are created from **images**. An image is a lightweight, immutable package containing everything needed to run a piece of software, including code, runtime, libraries, and configuration.

When you run a container, Docker looks for the image locally. If it doesn’t exist, Docker automatically **pulls** it from a remote registry (default: **Docker Hub**).

2.2 Pulling Images with Docker CLI

To pull an image explicitly, use the `docker pull` command:

```
docker pull nginx:latest
```

Here’s the breakdown:

- **nginx** → the image name.
- **:latest** → the tag (defaults to `latest` if omitted).

Commonly used public images:

- `nginx` → Web server.
- `mysql` → Relational database.
- `redis` → In-memory key-value store.
- `alpine` → Minimal Linux distribution (great for testing).

You can view downloaded images with:

```
docker images
```

2.3 Running Containers

To run an image as a container, use `docker run`:

```
docker run -d -p 8080:80 nginx
```

- **-d** → Detached mode (runs in the background).
- **-p 8080:80** → Maps port 8080 on your machine to port 80 inside the container.
- **nginx** → The image name.

Now if you open **`http://localhost:8080`** in your browser, you'll see the Nginx welcome page served from the container.

2.4 Interactive Mode vs Detached Mode

Interactive Mode

Use when you want to interact with the container directly (e.g., debugging).

```
docker run -it ubuntu bash
```

- This starts a container, attaches your terminal, and opens a Bash shell inside it.

Detached Mode

Use when running long-lived applications (e.g., web servers).

```
docker run -d redis
```

2.5 Managing Container Lifecycle

Containers can be started, stopped, paused, restarted, or removed.

List running containers:

```
docker ps
```

List all containers (including stopped):

```
docker ps -a
```

Stop a container:

```
docker stop <container_id>
```

Start a stopped container:

```
docker start <container_id>
```

Remove a container:

```
docker rm <container_id>
```

You can also use container names instead of IDs.

2.6 Inspecting and Accessing Containers

View logs of a container:

```
docker logs <container_name>
```

Attach to a running container:

```
docker exec -it <container_name> bash
```

- This opens an interactive shell session inside the container.

Inspect container details (network, volumes, etc.):

```
docker inspect <container_name>
```

2.7 Practical Example: Running a MySQL Database

```
docker run -d \
  --name mydb \
  -e MYSQL_ROOT_PASSWORD=secret \
  -p 3306:3306 \
```

```
  mysql:8.0
```

- **--name mydb** → Assigns a custom name.
- **-e MYSQL_ROOT_PASSWORD=secret** → Sets an environment variable inside the container.
- **-p 3306:3306** → Exposes MySQL's default port.

You can now connect to this MySQL container using any client.

2.8 Cleaning Up

Over time, unused containers and images pile up. Use these commands to clean:

Remove all stopped containers:

```
docker container prune
```

Remove unused images:

```
docker image prune
```

Remove everything (be careful):

```
docker system prune -a
```

2.9 Best Practices

- Always pull specific versions of images (`nginx:1.25.2`) instead of `latest` to avoid surprises.
- Use `docker logs` to monitor running containers.
- Map only necessary ports to reduce security exposure.
- Name your containers meaningfully (e.g., `frontend-app`, `backend-db`) for easier management.

Key Takeaways

- ☐ Use `docker pull` to fetch images.
- ☐ `docker run` starts containers; add `-it` for interactive, `-d` for detached.
- ☐ Manage containers with `docker ps`, `docker stop`, `docker rm`.
- ☐ Use volumes, environment variables, and ports for application configuration.

Chapter 3: Create and Manage Docker Volumes, Networks, and Port Bindings

Containers are ephemeral by design, when a container is removed, anything written inside its writable layer disappears. Volumes give you persistence. Networks let containers discover and talk to each other. Port bindings expose services to the host or external world. This chapter covers the concepts, CLI commands, examples, and best practices for volumes, networks, and port mappings.

3.1 Volumes: persistent, shareable storage

Types

- **Named volumes** (managed by Docker): `docker volume create mydata`. Good for production persistence.
- **Bind mounts** (host path to container): `-v /host/path:/container/path`. Useful for dev but riskier in production.
- **Anonymous volumes**: created when you `-v /container/path` without a name.
- **tmpfs**: in-memory mounts for ephemeral sensitive data: `--tmpfs /tmp`.

Two syntaxes

- Short form (older): `-v host_path:container_path:ro`
- Recommended modern form: `--mount type=volume,source=myvol,target=/data,readonly` (clearer, less ambiguous)

Common commands

```
docker volume create myvol
docker run -d --name db --mount source=myvol,target=/var/lib/mysql mysql:8
docker volume ls
docker volume inspect myvol
docker volume rm myvol
docker volume prune   # remove unused volumes
```

Backups & migration
Create backups with temporary containers:

```
docker run --rm -v myvol:/data -v $(pwd):/backup alpine \
  sh -c "tar czf /backup/myvol.tar.gz -C /data ."
```

Best practices

- Prefer named volumes for databases and important data.
- Use bind mounts for local development only.
- Use `readonly` when container should not modify host data.
- On SELinux hosts, add `:z` or `:Z` to bind mounts to fix permissions.

3.2 Networks: container connectivity and isolation

Drivers

- **bridge** (default): isolates containers on the host. The default `bridge` network does not provide automatic DNS name resolution; **user-defined bridge networks do.**
- **host**: container shares host network stack (no port mapping needed).
- **none**: container has no network.
- **overlay**: used for multi-host networking (Docker Swarm).
- **macvlan**: give containers direct L2 presence on LAN.

Create and use

```
docker network create --driver bridge app-net
docker run -d --name redis --network app-net redis
docker run -d --name web --network app-net --network-alias api nginx
# containers on app-net can resolve each other by name
docker network ls
docker network inspect app-net
docker network connect app-net existing_container
docker network rm app-net
```

Service discovery

- On **user-defined bridge networks**, containers can reach each other using container names or `--network-alias`.
- Avoid legacy `--link`; use networks.

Best practices

- Use separate networks for tiers (frontend, backend, db) to limit blast radius.
- Use overlay networks for swarm clusters; consider CNI plugins in Kubernetes.
- Restrict container network exposure and use firewall rules on the host.

3.3 Port Bindings: exposing container ports safely

Publish vs EXPOSE

- `EXPOSE 80` in a Dockerfile is documentation/metadata only.
- `-p` or `--publish` actually binds container ports to host ports.

Examples

```
docker run -d -p 8080:80 nginx         # host 8080 -> container 80
docker run -d -p 127.0.0.1:8080:80 nginx  # bind only to localhost
docker run -d -P redis                 # -P publishes exposed ports to
random host ports
docker run -d -p 53:53/udp bind-dns    # specify protocol
```

Notes

- Binding the same host port for multiple containers causes conflict.
- Use `127.0.0.1:hostport:containerport` to restrict access to the host.
- For production with many services, use a reverse proxy (nginx, Traefik) or load balancer instead of many host ports.

Compose example (snippet)

```
version: "3.8"
services:
  web:
    image: nginx
    ports:
      - "8080:80"
    networks:
      - frontend
  db:
    image: mysql:8
    volumes:
      - dbdata:/var/lib/mysql
    networks:
      - backend

volumes:
  dbdata:

networks:
```

```
  frontend:
  backend:
```

3.4 Summary / Best practices checklist

- Use `--mount` for clarity; prefer named volumes for persistence.
- Separate networks by tier and use user-defined bridge networks for DNS-based discovery.
- Avoid publishing unnecessary ports; prefer localhost bindings or reverse proxies for controlled exposure.
- Backup volumes regularly and prune unused items safely.

This foundation will let your containers persist data reliably, communicate securely, and expose services intentionally. Next chapter will dive into **Securing containers and managing secrets with Jenkins and HashiCorp Vault**.

Chapter 4: Secure Containers and Manage Secrets with Jenkins and HashiCorp Vault

Security is one of the most critical aspects of containerized environments. Containers often run applications that require access to sensitive data like API keys, database passwords, TLS certificates, or cloud credentials. Hardcoding these secrets in Dockerfiles or passing them directly via environment variables exposes significant risks. This chapter explores container security basics, introduces Jenkins for CI/CD pipeline integration, and demonstrates how to manage secrets securely with **HashiCorp Vault**.

4.1 Why Container Security Matters

Containers provide isolation, but misconfigurations or poor practices can still expose vulnerabilities. Common risks include:

- **Leaking secrets** in Dockerfiles, Git repositories, or images.
- **Overprivileged containers** running as root unnecessarily.
- **Exposed ports** accessible publicly without authentication.
- **Unscanned images** with outdated libraries containing CVEs.

A solid security posture combines **runtime hardening** with **secret management**.

4.2 Docker Security Best Practices

Before we integrate Jenkins and Vault, let's establish container security fundamentals:

Least privilege principle:
Run containers as non-root users (`USER` in Dockerfile).

Immutable images:
Use minimal base images (e.g., `alpine`) to reduce attack surface.

Patch regularly:
Rebuild and redeploy images often to pick up upstream fixes.

Scan images:
Use tools like `trivy`, `grype`, or Docker Hub's built-in scanning.

Limit capabilities:
Drop Linux capabilities not required by the container:

```
docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx
```

- **Restrict networks:**
  Place sensitive services on isolated networks.

4.3 Secret Management: The Wrong Way

A common mistake is embedding secrets in:

- Dockerfiles (`ENV DB_PASSWORD=secret`)
- Source code (`config.js`)
- CI/CD pipeline definitions (`pipeline.yml`)

All of these risk exposure if the repo is leaked, logs are shared, or images are pushed to public registries.

4.4 Introducing HashiCorp Vault

HashiCorp Vault is a popular open-source tool for secure secret storage, dynamic credential generation, and access control.

Key Features

- Centralized storage for secrets with strong encryption.
- Dynamic secrets: e.g., temporary database credentials that auto-expire.
- Fine-grained access policies (who can access what).
- Audit logs for compliance.

Basic Workflow

1. **Vault Server** stores secrets securely.
2. **Applications or Jenkins pipelines** request secrets via Vault API.
3. **Vault authenticates** the request (using tokens, AWS IAM, Kubernetes auth, etc.).
4. **Vault returns secrets** dynamically, often short-lived.

4.5 Jenkins and Vault Integration

Jenkins is widely used for automating builds, tests, and deployments. When integrated with Vault, Jenkins pipelines can fetch secrets at runtime instead of storing them statically.

Step 1: Install Plugins

- Install the **Vault Plugin** in Jenkins.

Step 2: Configure Vault in Jenkins

1. Navigate to **Manage Jenkins > Configure System**.
2. Add a Vault server with its address (e.g., **`https://vault.example.com:8200`**).
3. Authenticate Jenkins with Vault using either:
    - Token authentication (simple but less secure).
    - AppRole authentication (preferred for production).

Step 3: Fetch Secrets in Pipelines

Example Jenkins pipeline snippet:

```
pipeline {
  agent any
  stages {
    stage('Deploy') {
      steps {
        withVault([vaultSecrets: [
          [path: 'secret/data/db', secretValues: [
            [envVar: 'DB_USER', vaultKey: 'username'],
            [envVar: 'DB_PASS', vaultKey: 'password']
          ]]
        ]]) {
          sh 'echo "Deploying with user $DB_USER"'
        }
      }
    }
  }
}
```

Secrets are pulled securely from Vault and injected as environment variables only during pipeline execution.

4.6 Running Containers with Secrets

Docker CLI also supports secret injection via files or environment variables. Example with Vault + Docker:

```
docker run -d \
  --name app \
  --env DB_USER=$(vault kv get -field=username secret/data/db) \
  --env DB_PASS=$(vault kv get -field=password secret/data/db) \
```

```
myapp:latest
```

In production, prefer orchestrators (Kubernetes/Docker Swarm) that integrate with Vault natively for secret injection without exposing them in shell history.

4.7 Best Practices for Secrets and Security

- **Never store secrets in images**, use runtime injection.
- **Use Vault dynamic secrets** instead of static ones.
- **Rotate credentials** frequently; Vault can automate this.
- **Apply access policies** to restrict who/what can fetch secrets.
- **Audit logs** to detect unauthorized attempts.
- **Combine with container hardening**: even if secrets leak inside a compromised container, minimize what an attacker can do.

4.8 Wrapping Up

By now, you understand how to secure Docker containers and handle secrets responsibly. Jenkins pipelines combined with HashiCorp Vault provide a powerful way to automate deployments without compromising sensitive data.

This chapter emphasized **why security is crucial**, **what not to do**, and **how to implement proper secret management**.

In the next chapter, we'll shift gears and focus on **writing Dockerfiles to build custom container images**, which will teach you how to create lean, efficient, and secure container images tailored to your applications.

Key Takeaways

- ☐ Security requires both container hardening and secret management.
- ☐ Vault centralizes secret storage, rotation, and auditing.
- ☐ Jenkins integrates seamlessly with Vault for secure CI/CD pipelines.
- ☐ Avoid static secrets; prefer dynamic, short-lived credentials.

Chapter 5: Write Dockerfiles to Build Custom Container Images

A **Dockerfile** is the blueprint for building Docker images. It defines the base image, dependencies, environment configuration, and the commands needed to run your application. Well-written Dockerfiles are essential for creating lean, secure, and efficient container images. In this chapter, you'll learn the anatomy of a Dockerfile, best practices for building images, and examples for different use cases.

5.1 What is a Dockerfile?

A **Dockerfile** is a text file containing instructions that Docker reads line by line to build an image. Each instruction creates a new **layer** in the image. Layers are cached, which speeds up rebuilds.

Basic workflow:

1. Write a Dockerfile.
2. Build the image using `docker build`.
3. Run containers from the image.

5.2 Common Dockerfile Instructions

FROM: Sets the base image.

```
FROM ubuntu:22.04
```

RUN: Executes commands inside the image during build.

```
RUN apt-get update && apt-get install -y curl
```

COPY: Copies files from host to image

```
COPY . /app
```

ADD: Similar to `COPY` but can also extract archives or pull from URLs. Use sparingly.

WORKDIR: Sets the working directory inside the container.

```
WORKDIR /app
```

ENV: Sets environment variables.

```
ENV NODE_ENV=production
```

EXPOSE: Documents the port the container listens on (doesn't publish it).

```
EXPOSE 8080
```

CMD: Provides the default command to run when the container starts.

```
CMD ["node", "server.js"]
```

ENTRYPOINT: Similar to `CMD` but better for scripts where you want arguments passed consistently.

5.3 Building an Image from Dockerfile

Example: Node.js app

```
# Base image
FROM node:20-alpine

# Set working directory
WORKDIR /usr/src/app

# Copy package.json and install deps
COPY package*.json ./
RUN npm install --only=production

# Copy app code
COPY . .

# Expose port
EXPOSE 3000

# Start app
CMD ["node", "index.js"]
```

Build and run:

```
docker build -t my-node-app .
docker run -d -p 3000:3000 my-node-app
```

5.4 Multi-Stage Builds

Multi-stage builds reduce image size by separating build dependencies from runtime.

Example: Go app

```
# Build stage
FROM golang:1.22 AS builder
WORKDIR /src
COPY . .
RUN go build -o app

# Runtime stage
FROM alpine:3.19
WORKDIR /app
COPY --from=builder /src/app .
CMD ["./app"]
```

This way, the final image only contains the compiled binary, not the Go toolchain.

5.5 Best Practices for Writing Dockerfiles

Use minimal base images:
Prefer `alpine`, `distroless`, or slim variants.

```
FROM python:3.12-slim
```

Leverage caching:
Place commands that change less frequently (e.g., dependencies) before app code.

Avoid root user:

```
RUN adduser -D appuser
USER appuser
```

Combine RUN instructions to reduce layers:

```
RUN apt-get update && apt-get install -y curl && rm -rf
/var/lib/apt/lists/*
```

Use `.dockerignore`:
Exclude unnecessary files (logs, node_modules, .git).

Keep images small:
Remove build tools in final stage, install only what you need.

Don't store secrets in Dockerfiles. Use runtime injection instead.

Document with labels:

```
LABEL maintainer="you@example.com"
LABEL version="1.0"
```

5.6 Debugging and Inspecting Images

Build with debug info:

```
docker build -t debug-app --progress=plain .
```

Inspect image layers:

```
docker history my-node-app
```

Run shell inside an image for debugging:

```
docker run -it my-node-app sh
```

5.7 Special Use Cases

Python Flask App

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

Static Website with Nginx

```
FROM nginx:stable-alpine
COPY ./html /usr/share/nginx/html
EXPOSE 80
```

5.8 Wrapping Up

By now, you know how to write Dockerfiles, build images, and follow best practices to ensure efficiency and security. Dockerfiles are the foundation of reproducible environments, every production-ready container begins here.

In the next chapter, we'll expand into **Docker Compose**, where you'll learn how to define and run **multi-container applications** with a single configuration file.

Key Takeaways

- A Dockerfile defines the environment and instructions for your image.
- Use multi-stage builds to keep images lean.
- Apply best practices: minimal base images, caching, non-root users, and `.dockerignore`.
- Always test and inspect your images before deployment.

Chapter 6: Using Docker Compose to Build and Run Multi-Container Applications

When building modern applications, it's rare that you run just a single service. A typical web app might need a backend API, a database, a cache, and maybe a message broker, all working together. Running each container individually with the `docker run` command quickly becomes cumbersome. This is where **Docker Compose** shines. Docker Compose is a tool that allows you to **define, configure, and run multi-container applications** using a single YAML configuration file.

6.1 Introduction to Docker Compose

Docker Compose uses a **declarative YAML file** (`docker-compose.yml`) to describe your application's services, networks, and volumes. With this file, you can launch the entire application stack with a single command:

```
docker compose up
```

Key benefits:

- Simplifies multi-container orchestration.
- Provides reproducibility…your entire stack is described in one file.
- Enables service isolation with networks.
- Allows persistent data with volumes.
- Provides environment portability across dev, test, and production.

6.2 Installing Docker Compose

In modern Docker Desktop, Docker Compose comes pre-installed. On Linux, you may need to install it separately:

```
sudo apt-get update
sudo apt-get install docker-compose-plugin
```

Verify installation:

```
docker compose version
```

6.3 Anatomy of a `docker-compose.yml` File

Here's the basic structure:

```
version: "3.9"
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: appdb
    volumes:
      - db_data:/var/lib/postgresql/data
volumes:
  db_data:
```

- **services:** Define individual containers (e.g., `web`, `db`).
- **image:** Specify the container image.
- **ports:** Expose container ports to the host.
- **environment:** Inject environment variables.
- **volumes:** Define persistent storage.

6.4 Common Commands in Docker Compose

Start containers:

```
docker compose up
```

Start in detached mode:

```
docker compose up -d
```

Stop containers:

```
docker compose down
```

Rebuild images:

```
docker compose build
```

View logs across services:

```
docker compose logs -f
```

Scale services:

```
docker compose up -d --scale web=3
```

6.5 Networking in Compose

By default, Docker Compose creates a **dedicated network** so services can talk to each other using their service names as hostnames.

Example:

```
services:
  api:
    build: ./api
  frontend:
    build: ./frontend
    depends_on:
      - api
```

Here, the `frontend` container can reach the `api` service simply by connecting to `http://api:port`.

6.6 Using Volumes with Compose

Compose allows you to easily mount volumes for persistent data:

```
volumes:
  db_data:
services:
  db:
    image: mysql:8
    environment:
```

```
      MYSQL_ROOT_PASSWORD: rootpass
    volumes:
      - db_data:/var/lib/mysql
```

This ensures the database data persists even if the container restarts.

6.7 Multi-Container Application Example

Let's build a **Node.js + MongoDB application** with Compose:

```
version: "3.9"
services:
  app:
    build: ./app
    ports:
      - "3000:3000"
    environment:
      - MONGO_URL=mongodb://db:27017/mydb
    depends_on:
      - db

  db:
    image: mongo:6
    volumes:
      - mongo_data:/data/db

volumes:
  mongo_data:
```

Steps to run:

1. Place your Node.js app Dockerfile inside `./app`.
2. Run `docker compose up -d`.
3. Access the app at `http://localhost:3000`.

6.8 Compose Best Practices

- Always use `.env` files for secrets, not hardcoding values.
- Use version control to track `docker-compose.yml`.
- Separate dev and production Compose files (e.g., `docker-compose.override.yml`).

- Use `depends_on` only for startup order, not for health checks, consider healthcheck configurations for production.

6.9. Advantages Over Manual Docker CLI

- One command replaces multiple `docker run` invocations.
- Automatic networking setup.
- Better scalability and reproducibility.
- Simplified development-to-production migration.

Conclusion

Docker Compose is the bridge between simple single-container applications and full-scale orchestrators like Kubernetes. It allows developers to **prototype, test, and manage multi-service applications effortlessly**. Whether you're spinning up a quick development environment or deploying a lightweight stack, Docker Compose provides all the tools you need to manage multi-container applications efficiently.

Chapter 7: Publishing and Pulling Images Using a Docker Registry

Container images are the backbone of Docker. They package your application code, runtime, libraries, and dependencies into a consistent, portable unit. But images aren't meant to live only on your local machine, you often need to **share them with teammates, deploy them to servers, or integrate them into CI/CD pipelines**. This is where **Docker registries** come into play.

A Docker registry is a centralized service for storing and distributing container images. The most common registry is **Docker Hub**, but enterprises often use private registries like **Amazon Elastic Container Registry (ECR)**, **Google Container Registry (GCR)**, or **Harbor**.

In this chapter, you'll learn how to **publish, pull, and manage images** with registries.

7.1 Understanding Docker Registries

A **Docker registry** hosts repositories, and each repository contains multiple tagged images.

Example repository:

```
shreyasladhe/myapp:1.0
```

- `shreyasladhe` → namespace (username or organization).
- `myapp` → repository name.
- `1.0` → image tag.

If no tag is specified, Docker assumes `:latest`.

7.2 Logging In to a Registry

Before publishing images, authenticate with the registry.

Docker Hub:

```
docker login
```

- You'll be prompted for your username and password.

AWS ECR:

```
aws ecr get-login-password --region ap-south-1 | docker login --username
AWS --password-stdin <account_id>.dkr.ecr.ap-south-1.amazonaws.com
```

GCP Artifact Registry:

```
gcloud auth configure-docker
```

7.3 Tagging Images for the Registry

Docker uses tags to identify images. To push an image, you need to tag it with the registry URL.

Example:

```
docker build -t myapp:1.0 .
docker tag myapp:1.0 shreyasladhe/myapp:1.0
```

For AWS ECR:

```
docker tag myapp:1.0
<account_id>.dkr.ecr.ap-south-1.amazonaws.com/myapp:1.0
```

7.4 Publishing (Pushing) Images

Once tagged, push the image:

```
docker push shreyasladhe/myapp:1.0
```

Docker will upload image layers to the registry. Layers already present in the registry are skipped, which saves time.

7.5. Pulling Images

Anyone with access can now pull your image:

```
docker pull shreyasladhe/myapp:1.0
```

Run the container directly:

```
docker run -d -p 8080:80 shreyasladhe/myapp:1.0
```

7.6 Working with Private Registries

For sensitive projects, you may use private repositories.

- On Docker Hub, mark a repository as **private**.
- On ECR, permissions are managed with **IAM roles and policies**.
- On self-hosted Harbor, set up users and access controls.

To pull from a private registry, you must first log in with `docker login`.

7.7 Automating Push/Pull in CI/CD Pipelines

Registries integrate seamlessly with CI/CD tools like Jenkins, GitHub Actions, or GitLab CI.

Example Jenkins pipeline snippet:

```
stage('Build and Push') {
  steps {
    sh 'docker build -t myapp:${BUILD_NUMBER} .'
    sh 'docker tag myapp:${BUILD_NUMBER}
<account_id>.dkr.ecr.ap-south-1.amazonaws.com/myapp:${BUILD_NUMBER}'
    sh 'docker push
<account_id>.dkr.ecr.ap-south-1.amazonaws.com/myapp:${BUILD_NUMBER}'
  }
}
```

This ensures every build produces a uniquely tagged image stored in the registry.

7.8 Managing Image Versions and Tags

- Use semantic versioning (`1.0.0`, `1.1.0`) for clarity.
- Maintain `latest` as the stable tag.
- Automate cleanup of old or unused tags using registry lifecycle policies (e.g., in ECR).

7.9 Security and Best Practices

- **Scan images**: Docker Hub and ECR can automatically scan images for vulnerabilities.
- **Use least privilege**: Don't push/pull with root accounts, use IAM roles or scoped tokens.
- **Minimize image size**: Use lightweight base images like `alpine`.
- **Sign images**: With Docker Content Trust (`DOCKER_CONTENT_TRUST=1`) or Notary, ensure integrity.

7.10 Example: Publishing to Docker Hub

Let's walk through a full cycle:

Build the image:

```
docker build -t myapp:1.0 .
```

Tag for Docker Hub:

```
docker tag myapp:1.0 shreyasladhe/myapp:1.0
```

Push to Docker Hub:

```
docker push shreyasladhe/myapp:1.0
```

Pull on another machine:

```
docker pull shreyasladhe/myapp:1.0
docker run -p 8080:80 shreyasladhe/myapp:1.0
```

Conclusion

Publishing and pulling images from registries is a cornerstone of containerized workflows. Whether you're using Docker Hub for quick sharing or AWS ECR for enterprise-grade security, registries enable seamless collaboration, version control, and deployment. By tagging consistently, automating in CI/CD, and following security best practices, you ensure your container images remain reliable, portable, and secure across environments.

Chapter 8: Orchestrating Containers Using Docker Swarm and Kubernetes

Running a few containers manually is manageable, but modern applications often scale to **hundreds or thousands of containers** across multiple machines. Managing deployments, scaling, networking, storage, and fault tolerance by hand would be impossible. This is where **container orchestration** tools like **Docker Swarm** and **Kubernetes** come in.

These tools provide a framework for deploying and managing containerized applications across a **cluster of servers**. In this chapter, we'll cover both Docker Swarm (lightweight, built into Docker) and Kubernetes (industry standard, highly extensible).

8.1 Why Orchestration?

Orchestration tools solve problems that go beyond single-host Docker:

- **Automated scaling**: Increase or decrease containers based on demand.
- **High availability**: Restart failed containers and distribute load.
- **Service discovery**: Let containers find each other by name, without manual networking.
- **Rolling updates**: Deploy new versions with zero downtime.
- **Resource management**: Efficiently allocate CPU, memory, and storage.

8.2 Docker Swarm Overview

Docker Swarm is Docker's native orchestration tool. It transforms multiple Docker hosts into a **single virtual cluster**.

Key Concepts:

- **Swarm**: A group of Docker engines (nodes).
- **Node**: A host machine in the cluster (manager or worker).
- **Service**: A definition of containers (tasks) to run.
- **Overlay networks**: Enable secure multi-host networking.

8.3 Setting Up a Swarm Cluster

Initialize a Swarm:

```
docker swarm init
```

This creates a manager node. To add workers, Docker provides a join token:

```
docker swarm join --token <token> <manager_ip>:2377
```

Check nodes:

```
docker node ls
```

8.4 Deploying Services in Swarm

Deploy a service with 3 replicas:

```
docker service create --name web --replicas 3 -p 8080:80 nginx
```

Check running services:

```
docker service ls
docker service ps web
```

Update service:

```
docker service update --image nginx:1.23 web
```

Remove service:

```
docker service rm web
```

8.5 Scaling in Swarm

Scale services up or down easily:

```
docker service scale web=5
```

Swarm automatically distributes replicas across available worker nodes.

8.6 Kubernetes Overview

While Swarm is simpler, **Kubernetes (K8s)** has become the **de facto standard** for container orchestration. Originally developed by Google, it's now maintained by the CNCF.

Core Concepts:

- **Cluster**: A group of machines running Kubernetes.
- **Node**: A worker machine.
- **Pod**: The smallest deployable unit (one or more containers).
- **Deployment**: Defines how pods are created and managed.
- **Service**: Exposes pods to the network (internal or external).
- **ConfigMap & Secret**: Manage configuration and sensitive data.

8.7 Kubernetes Architecture

- **Master components (control plane)**: API Server, Scheduler, Controller Manager, etcd.
- **Worker components**: Kubelet (manages pods), Kube-proxy (networking).

8.8. Running Apps on Kubernetes

1. **Define a Deployment (YAML):**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25
        ports:
        - containerPort: 80
```

Apply it:

```
kubectl apply -f nginx-deployment.yaml
```

2. **Expose the Service:**

```
kubectl expose deployment nginx-deployment --type=LoadBalancer --port=80
```

3. **Check pods and services:**

```
kubectl get pods
kubectl get svc
```

8.9 Rolling Updates and Scaling in Kubernetes

Update Deployment:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.26
```

Scale Pods:

```
kubectl scale deployment nginx-deployment --replicas=5
```

Roll back if update fails:

```
kubectl rollout undo deployment/nginx-deployment
```

8.10 Swarm vs Kubernetes

Feature	Docker Swarm	Kubernetes
Ease of setup	Simple (few commands)	Complex (steeper learning)
Features	Basic orchestration	Full enterprise features
Community adoption	Limited	Very large & growing
Use case	Small/medium apps	Production-scale systems

8.11 Best Practices for Orchestration

- Use **health checks** for services.
- Store secrets securely (Vault, Kubernetes Secrets).

- Monitor cluster health with Prometheus, Grafana, or built-in tools.
- Automate deployments with CI/CD.
- For production, always use Kubernetes unless simplicity is critical.

Conclusion

Container orchestration is the key to scaling applications reliably. Docker Swarm provides a simple introduction to clustering, while Kubernetes offers a **powerful, flexible, and production-grade orchestration platform**. Together, they allow you to move from running a few containers locally to managing **large-scale, highly available containerized applications in the cloud or on-premises**.

By mastering both, you're equipped to handle workloads of any size, from small dev clusters to enterprise-grade deployments.