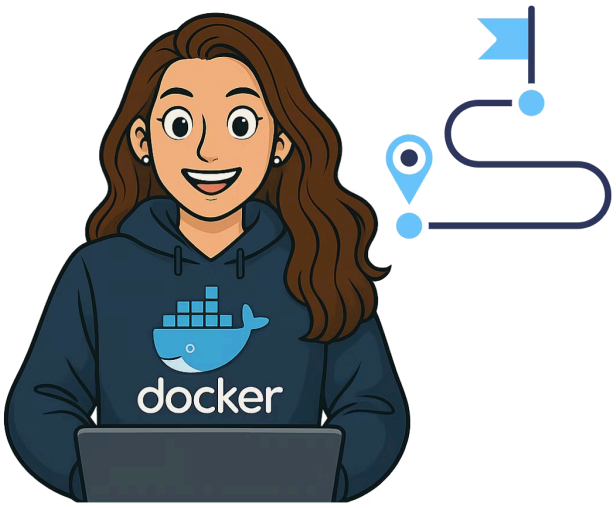


Docker Learning Roadmap

Your Complete Step-by-Step Guide from Docker Beginner to Pro

by TechWorld with Nana

TECHWORLD WITH NANA

Welcome, Future Docker Star! 🐳

This roadmap transforms the overwhelming world of containerization into a clear, step-by-step journey.

No fluff, no confusion—just the essential knowledge you need to master Docker in weeks instead of years.

✅ **Problem-first approach** - Understand why before how

✅ **Beginner Friendly** - Start from zero, end up sipping containers like a pro

Who the heck created this?

Glad you asked :)

I'm Nana, Co-Founder of TechWorld with Nana.

As an engineer and trainer, I'm dedicated to helping engineers build the most valuable and highly-demanded DevOps and Cloud skills.

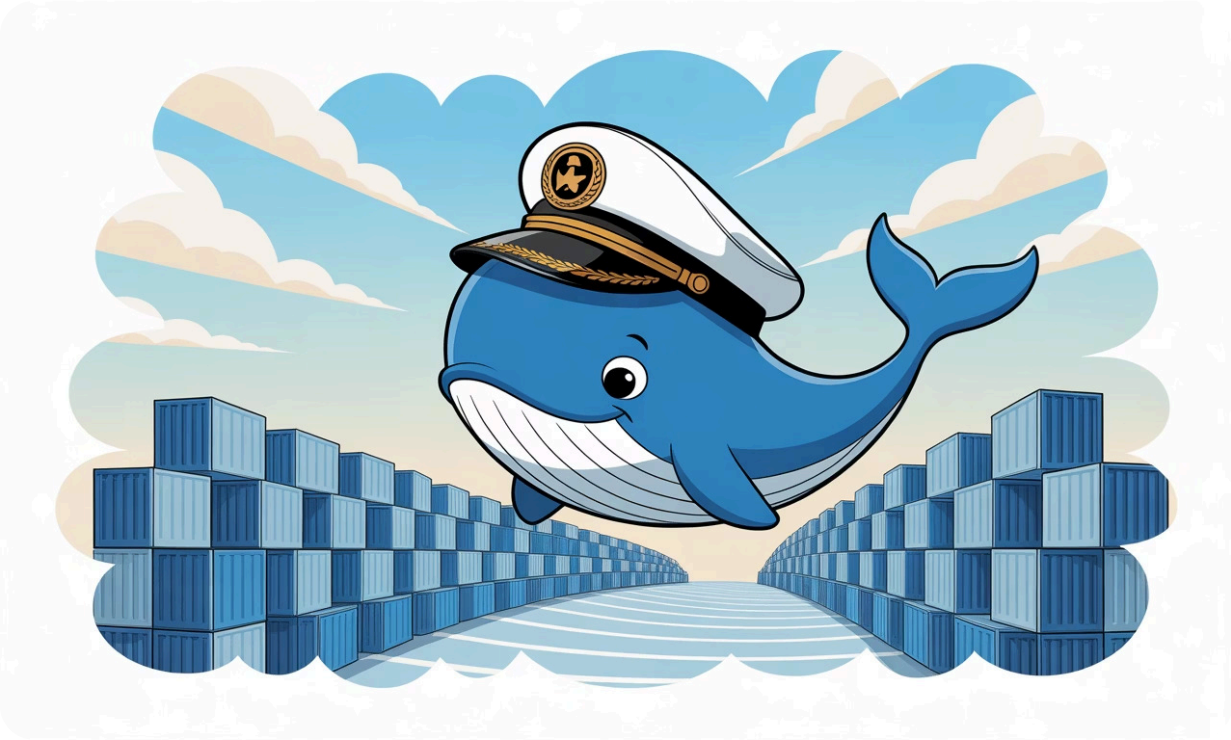
Through my **YouTube channel** and my comprehensive **DevOps bootcamps**, **I've helped millions of engineers** master the tools and concepts that drive modern software development.

Happy learning!

Nana

🧭 Learning Path Overview

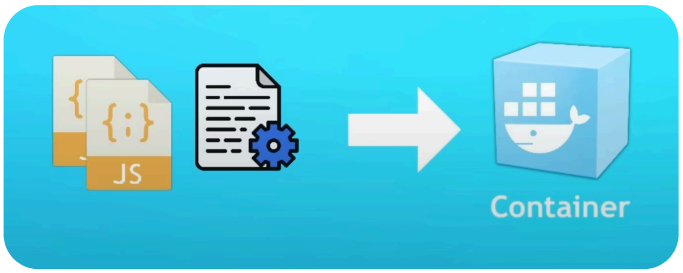
1. Phase 1: Docker Fundamentals (Week 1)
2. Phase 2: Building your Own Images (Week 2-3)
3. Phase 3: Docker Networking & Data Persistence (Week 4)
4. Phase 4: Multi-Container Apps (Week 5-6)
5. Phase 5: Advanced (Week 7-8)
6. BONUS: Moving Beyond Docker



Docker has become the **standard technology for containerization** that revolutionized software development, testing and deployment.

But, why?

Docker solves the infamous **"it works on my machine"** problem by packaging your application WITH its environment into containers that run exactly the same everywhere:



The Docker lightbulb moment:

When you stop worrying about environment setup and start focusing on what your application actually does.

📚 Prerequisites

But before starting your Docker journey, you should have:

Command Line Basics

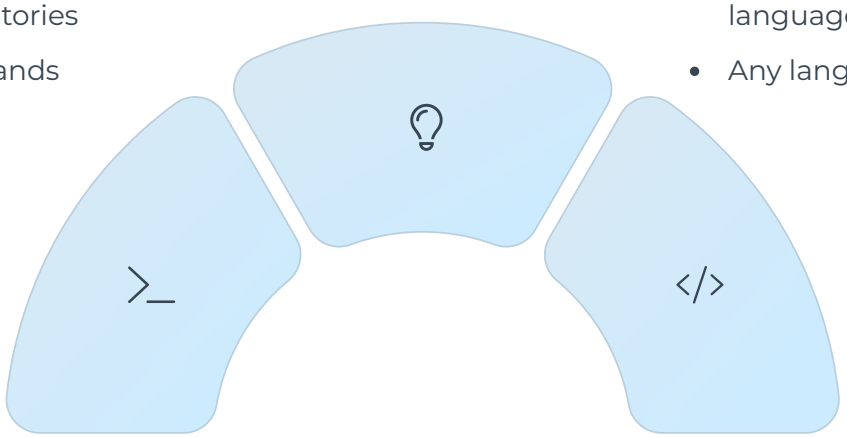
- Basic command line/terminal usage
- Navigating directories
- Running commands

Conceptual Knowledge

- Understanding of applications
- Understanding of servers
- How software environments work

Programming Fundamentals

- Basic knowledge of at least one programming language
- Any language is fine



Ready? Let's do this! 👏

Phase 1: Docker Fundamentals

Docker Fundamental Concepts

The first step is to understand what Docker is solving. Before diving into Docker commands, you need to understand the problem of "it works on my machine" and how containers solve this.

1) Understand the WHAT and the WHY?

What are containers?

Lightweight, portable packages that include everything needed to run an application

Why do we need containers?

Solve environment consistency, dependency conflicts, and deployment issues

VMs vs Containers

Understand the key differences in resource usage, startup time, and isolation

2) Understand fundamental Docker concepts

Docker architecture

Images, containers, Docker Engine, and how they work together

Container lifecycle

How containers are created, started, stopped, and removed

Docker Image

Think of it like a recipe or blueprint. It's a file that has everything needed to run an app - the code, settings, and tools. It's just sitting there, not doing anything yet.

Docker Container

This is when you actually use the recipe. It's the running version of the image - your app is alive and working. You can have many containers from one image, like making multiple cakes from the same recipe.

Simple way to remember:

- Image = Recipe (static)
- Container = The actual cake you made (running)

🛠️ Docker Installation & Setup

Follow these steps to get Docker Desktop running on your system:

Docker Desktop Installation

Download

Download Docker Desktop from **docker.com**

Install

Run the installer and follow the setup wizard

Launch

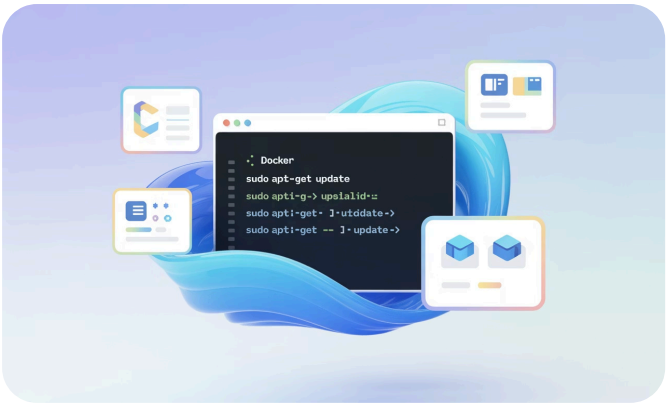
Start Docker Desktop after installation

Verify

Confirm installation by typing: docker --version

For Linux:

1. Install Docker Engine using your distribution's package manager
2. Add your user to the docker group: sudo usermod -aG docker $USER
3. Log out and back in for group changes to take effect
4. Verify installation: docker --version



First Test

```
# Run the hello-world container to test your setup

docker run hello-world
```

If you see a "Hello from Docker!" message, you're ready to go!

Public Images

One of the most powerful aspects of Docker is that you don't need to create images for common services yourself.

💡 **Key insight:** Almost every major software service has official Docker images ready to use!

Instead of installing PostgreSQL locally, simply:

```
docker pull postgres

docker run -e POSTGRES_PASSWORD=mysecretpassword -p 5432:5432 postgres
```

And just like that, you have a fully functional PostgreSQL database running on your machine without installing anything except Docker.

This access to ready-made containers drastically reduces the time to get started with new technologies and ensures you're using configurations that follow best practices.

Public Docker Repository (Docker Hub)

All these images are stored in Docker repositories, with Docker Hub being the most popular public repository. Docker Hub serves as a **centralized repository** where you can find and pull official images for almost any software service.

ⓘ Think of Docker Hub like GitHub but for container images instead of code.

Find Image

Search for official images on Docker Hub

Pull Image

Download the image to your local machine

Run Container

Create and start a container from the image

Use Service

Access the running service via mapped ports

Get Hands-On with Docker Commands

Start with these **core commands to pull and run existing images:**

```
# Pull an image from Docker Hub
docker pull nginx

# Run a container from an image
docker run -d -p 8080:80 nginx

# List running containers
docker ps

# Stop a running container
docker stop container_id

# Remove a container
docker rm container_id

# List all images
docker images
```

Example to run an "nginx" container:

Pull Image

docker pull nginx

Run Container

docker run -d -p 8080:80 nginx

Stop Container

docker stop *container_id*

Remove Container

docker rm *container_id*

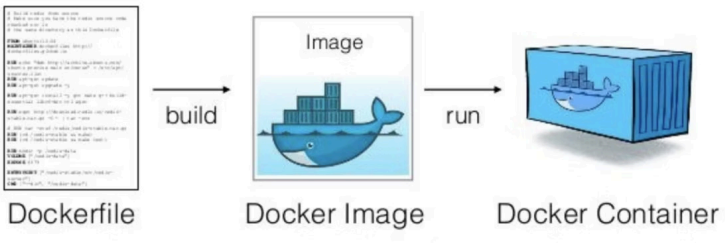
I recommend starting with a simple public image like Nginx or Redis.

Try pulling it, running it, and interacting with it. This gives you hands-on practice with these commands in a real context without needing to build anything yourself yet.

Phase 2: Building Your Own Images

Phase 2: Building Your Own Images

Building custom Docker images is a fundamental skill. And then you just run them as containers, just like any pre-existing image from the repository.



It's much easier than you can imagine 👏

Here's what you'll learn:

Docker images vs containers

Images are templates, containers are running instances

Dockerfile syntax

All essential instructions (FROM, RUN, COPY, WORKDIR, EXPOSE, CMD)

Image layers

How Docker builds images in layers for efficiency

Build context

What files Docker can access during build

Image tagging

How to name and version your images

Best practices

Writing efficient, maintainable Dockerfiles

Understanding Dockerfiles

A Dockerfile is your recipe for creating custom images. Here's a Node.js example:


```
FROM node:14-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]
```

What each line does:

- **FROM** Base image to start with
- **WORKDIR** Sets working directory inside container
- **COPY** Moves files from your machine to the image
- **RUN** Executes commands when building the image
- **EXPOSE** Documents which ports the container uses
- **CMD** Defines what runs when container starts

💡 **Key insight:** Each instruction creates a new layer in the image, building upon the previous layers in sequence. This layered approach enables efficient builds and image sharing.

ⓘ The way I always approach learning new tools is by defining a simple but realistic use case. Instead of just memorizing commands, take a simple application you already have and try to dockerize it. Through this process, you'll naturally learn what each command does and why it's necessary.

Example project you can use: [Project URL]

Building and Running Your Image

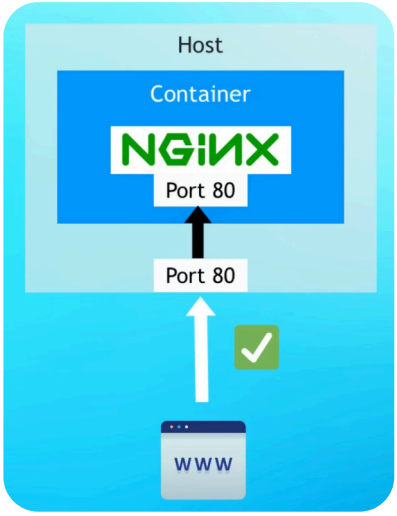
Once you have Dockerfile, with very simple docker commands, you can build and run your own application image like this:

```
# Build your image

docker build -t my-node-app .
```

```
# Run custom image

docker run -p 3000:3000 my-node-app
```

What happens during build?

Docker creates a layered filesystem, with each instruction in your Dockerfile becoming a new layer in the image.



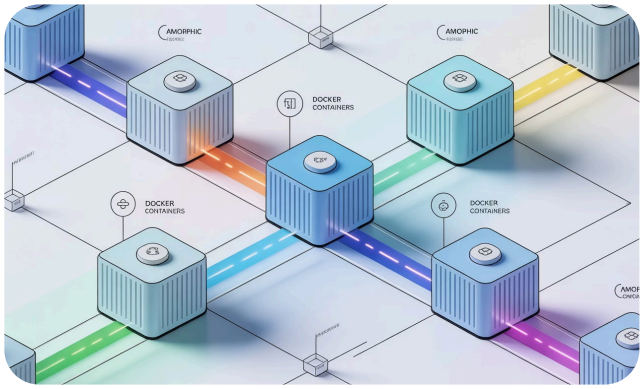
What happens during run?

Docker launches a container from your image, maps ports from the container to your host machine, and executes the CMD instruction.

Phase 3: Docker Networking and Data Persistence

Understanding these two concepts is essential for building production-ready containerized applications that can communicate effectively and maintain data between container lifecycles.

Docker Networking



Now while Dockerfiles are excellent for defining single containers, real-world applications typically require multiple containers working together.

So typically you have multiple containers, like frontend container, a backend API container and a database container that need to **communicate** with each other.

This brings us to the next crucial concept in your Docker journey: Docker networking!

What you need to learn:

- **Container networking basics:** How containers communicate with each other and the outside world
- **Docker network types:** Bridge, host, overlay networks and when to use each
- **Port mapping:** Exposing container services to the host machine
- **Container-to-container communication:** Using container names as hostnames

Example: Frontend needs to talk to a backend API 👇

```
# Create a network
docker network create my-network

# Run backend container in the network
docker run -d --name api --network my-network my-backend-image

# Run frontend container in the same network
docker run -d --name frontend --network my-network -p 8080:80 my-frontend-image
```

💡 Now the frontend container can communicate with the backend using the container name "api" as the hostname, because they are in the same container network.

Data Persistence

As your applications grow more complex with multiple interconnected containers, you'll face another challenge:

- How to ensure data persists even when containers are destroyed and recreated?

Our apps will need databases and if we run databases as containers to save data:

- How do we persist the data?

❌ **The problem**: Containers are by nature ephemeral, so they are designed to be temporary and easily replaceable. That's why data disappears when you remove the container.

✅ **The solution**: Volumes store data outside containers. Think of Volumes like a persistent bridge between your temporary container and permanent storage on your host machine.

ⓘ Docker says: "We manage the environment runtime, you manage the underlying data as you wish." But with Volumes gives us an interface to connect this data storage to containers, to have persistence.

What you need to learn:

- ◯ How data persistence works with Docker
- ◯ **Volume types:** Named volumes, bind mounts, and tmpfs mounts
- ◯ **Volume management:** Creating, using, and cleaning up volumes

Example: Creating a Volume

```
# Create a volume
docker volume create my-data

# Use volume with a database
docker run -d --name mysql -v my-data:/var/lib/mysql -e MYSQL_ROOT_PASSWORD=password my
```

💡 Even if you delete the container, your data survives in the volume.

Phase 4: Docker Compose - Multi-Container Magic

Docker Compose - Multi-Container Magic

❌ Now that you've mastered the basic commands, you'll notice that typing these commands over and over can become repetitive and error-prone

✅ This is where the power of Docker compose comes in - allowing you to define multiple containers that should run in the same network in 1 place in code.

It saves time and avoids manual setup every time you run your app.

ⓘ This shift from manual commands to **configuration as code** is a fundamental DevOps practice that enhances reproducibility and consistency.

What you need to learn:

- ◯ **Infrastructure as Code** - Defining your entire application stack in YAML
- ◯ **Docker Compose file structure** - Services, networks, volumes, and environment variables
- ◯ **Service dependencies** - Using depends_on to control startup order
- ◯ **Environment configuration** - Managing different environments (dev, staging, prod)
- ◯ **Scaling services** - Running multiple instances of the same service
- ◯ **Docker Compose commands** up, down, logs, exec, and scaling operations

✓ I've created a **deep dive tutorial on Docker Compose** here:

ULTIMATE DOCKER COMPOSE TUTORIAL

Example docker-compose.yml:

Docker Compose defines your multi-container application with a single YAML file:

```
version: '3'
services:
  database:
    image: postgres:13
    environment:
      - POSTGRES_PASSWORD=password
    volumes:
      - db-data:/var/lib/postgresql/data

  api:
    build: ./api
    depends_on:
      - database
    environment:
      - DATABASE_URL=postgres://postgres:password@db:5432/postgres

  frontend:
    build: ./frontend
    ports:
      - "8080:80"
    depends_on:
      - api
volumes:
  db-data:
```

One Command to Rule Them All

Run your entire application stack with one command.

Start Everything

```
docker-compose up -d
```

Stop & Clean Up

```
docker-compose down
```

Docker Compose automatically creates networks and manages dependencies!

Phase 5: Advanced

Production Best Practices

Now as you become comfortable with Docker local development, you'll need to start thinking about how to properly prepare your Docker applications for production environments.

Building secure, efficient, and maintainable Docker images requires following best practices that go beyond just getting containers to run.

What You Need to Learn:

- ◯ **Security fundamentals** - Running as non-root user, scanning for vulnerabilities
- ◯ **Image optimization** - Multi-stage builds, minimizing layers, choosing base images
- ◯ **Resource management** - Setting memory and CPU limits
- ◯ **Health checks** - Ensuring container health and automatic restarts
- ◯ **Logging strategies** - Centralized logging and log management
- ◯ **Configuration management** - Using environment variables and secrets
- ◯ **Image versioning** - Semantic versioning and tagging strategies

Some Important Best Practices:

1. Use Specific Image Tags

❌ Don't do this:

```
FROM node:latest
```

✅ Do this:

```
FROM node:18.17-alpine
```

Why? "latest" can change overnight and break your app!

Production Best Practices

2. Minimize Image Layers

❌ Multiple RUN commands:

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

✅ Combine with &&:

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

3. Multi-Stage Builds

Build your app in one stage, copy only what you need to production:

```
# Build stage
FROM node:14 AS build WORKDIR /app COPY . . RUN npm ci && npm run build
```

```
# Production stage
FROM nginx:alpine COPY --from=build /app/build /usr/share/nginx/html
```

4. Don't Run as Root

```
RUN addgroup -S appgroup && adduser -S appuser -G appgroup USER appuser
```

5. Scan for Vulnerabilities

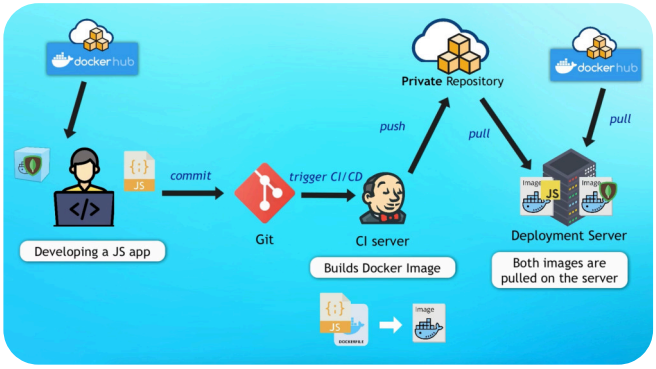
```
docker scout cves node:18.17-alpine
```

Learn from my **free video on Top 8 Production Best Practices** here:

[Top 8 Docker Best Practices for using Docker in Production]

Docker in whole software development and deployment workflow

After mastering Docker production best practices, you'll want to integrate Docker into your overall development and deployment workflow.



Common Workflow 👆

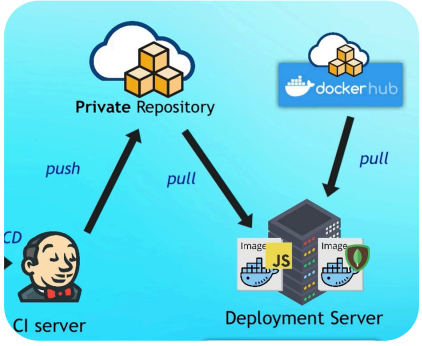
This means learning how to store your Docker images in private registries and **automate the process of building and deployment your Docker images through CI/CD pipelines**.

What you need to learn:

- **Container registries** - Public vs private registries (Docker Hub, AWS ECR, GitHub Container Registry)
- **CI/CD integration** - Execute docker commands on CI server through pipeline as code
- **Environment promotion** - Moving images through dev → staging → production
- **Registry security** - Access control, image signing, and vulnerability scanning

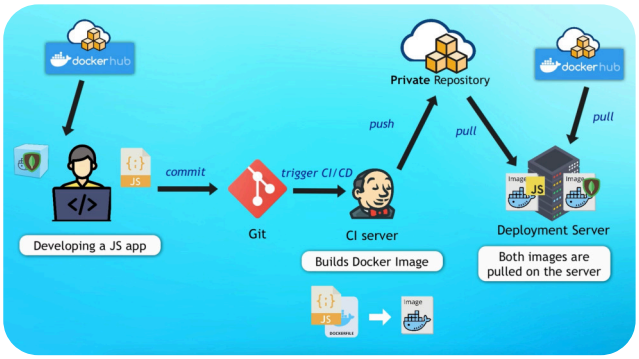
Storing Images in Registries

Use Docker Hub, AWS ECR, or GitHub Container Registry to store and share your images.



Container registries are a core component of integrating Docker into CI/CD pipelines.

They **act as the bridge between your build and deployment processes.**



👨‍💻 1. Development Phase

- A developer writes a JavaScript application on their local machine.
- During development, the developer might pull base images (e.g., MongoDB) from Docker Hub to use in local containers for testing the application.

🔀 2. Version Control

- The developer commits the code changes to a Git repository (e.g., GitHub, GitLab).
- This commit triggers a CI/CD pipeline (continuous integration/continuous deployment).

⚙️ 3. CI/CD

- The CI server (e.g., Jenkins, GitHub Actions) is automatically triggered by the Git commit.
- Jenkins runs the CI pipeline which:
  - Runs tests and checks
  - Builds a Docker image from the source code using a Dockerfile

📦 4. Docker Image Build

- The JS app and dependencies are packaged into a Docker image.
- This image is a standalone executable package with everything needed to run the application.

☁️ 5. Push to Private Repository

- The newly built Docker image is pushed to a private Docker image repository (e.g., hosted on AWS ECR, GitHub Container Registry, or a private Docker Registry).

🚀 6. Deployment Phase

- The deployment server (e.g., a VM or Kubernetes node) pulls:
  - The new JS app image from the private repository
  - Any external dependency images (e.g., MongoDB) directly from Docker Hub

🖥️ 7. Final Setup

- On the deployment server, both Docker images are now available:
  - The application image
  - The database or dependency image
- These containers are run together, allowing the app to function in a production-like environment.

Summary of Docker's Role in Each Step

Phase	Docker's Role
Development	Run services locally using Docker containers (e.g., MongoDB)
CI/CD	Package app into Docker image for consistent builds
Artifact Storage	Store Docker image in a private registry
Deployment	Pull and run Docker images on the server

GitHub Actions Example

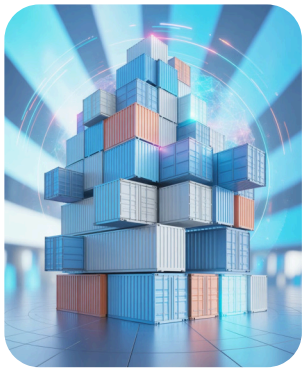
```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Build Docker image
        run: docker build -t myapp:${{ github.sha }} .
      - name: Push to registry
        run: |
          docker tag myapp:${{ github.sha }} myregistry.com/myapp:${{ github.sha }}
          docker push myregistry.com/myapp:${{ github.sha }}
```

This builds your Docker image and pushes it to your registry on every commit, automating your delivery pipeline.

ⓘ This automation is a core principle of DevOps that improves efficiency and reduces human error.

Bonus: Moving Beyond Docker

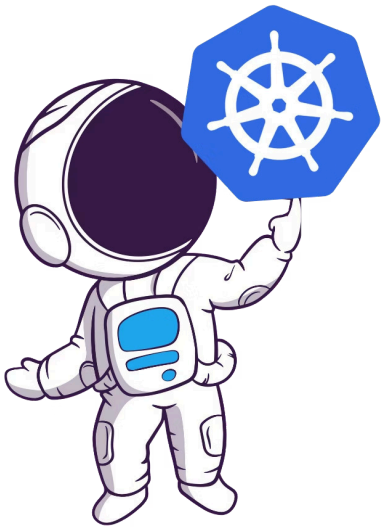
Container Orchestration



Why it's needed - Microservices

In practice, in modern projects, you will most certainly be working with micro services application which are perfect use case for Docker containers.

So if you have a large micro services application, you will have a container per microservice, **ending up with tons of containers to manage**.



Kubernetes - Managing containers at scale

As your containerized applications grow, you'll need to think about:

- Automatically **restarting** failed containers
- **Scaling** containers up or down based on load
- **Rolling updates** with zero downtime
- **Load balancing** traffic across container instances

This is where container orchestration platforms like Kubernetes come into play - the natural next step after mastering Docker.

Kubernetes builds on the foundation of Docker containers but **adds powerful features for managing hundreds or sometimes 1000s of containers across 100s of servers**, and to automatically deploy, scale, and manage these containers.

ⓘ 👉 **Docker** - How to package and run containers.

👉 **Kubernetes** - How to run containers at scale, reliably, and in production.

Container Orchestration

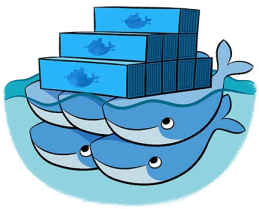
Wait, wasn't Docker Compose for running multi-containers?

Docker Compose is fantastic for what it was designed for: development environments and simple applications. But it's not an orchestration platform.

When Docker Compose works great:	When you need something more:
🟢 Local development (this is where it shines!)	🟠 Multiple environments with different scaling needs
🟢 Simple staging environments	🟠 Applications with 5+ services
🟢 Small applications with 2-3 services	🟠 Need for rolling deployments
🟢 Proof of concepts and demos	🟠 Auto-scaling requirements
	🟠 Complex networking or load balancing

Other Container Orchestration Tools

Kubernetes is the most popular container orchestration tool, but there are other alternatives.



Docker Swarm is Docker's native container orchestration tool that allows users to deploy, manage, and scale containers across a cluster of Docker hosts.

Docker Swarm is simpler and easier to learn than Kubernetes, but it has **critical limitations.**

Cloud Services: AWS ECS, Google Cloud Run, Azure Container Instances

Managed Kubernetes Services by Cloud Provider: AWS EKS, Azure AKS, Google GKE

These services take care of the technical stuff behind the scenes, so teams can focus on building their applications instead of worrying about how to run them.

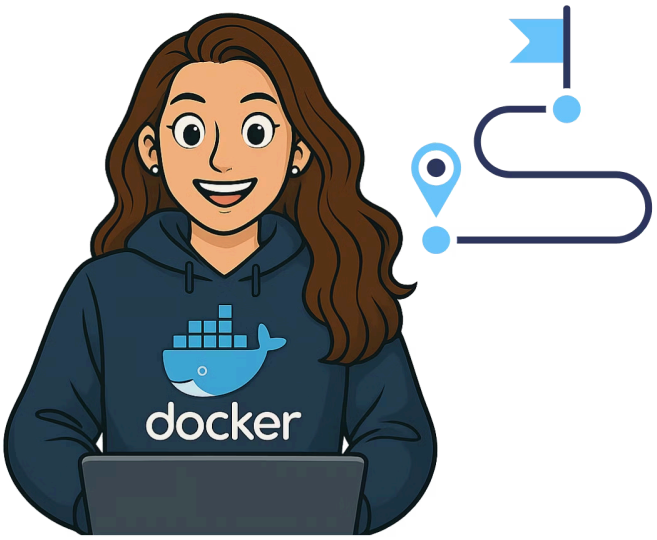
Need a Structured Learning Path?

Learning Docker on your own is great, but sometimes you need guidance and a clear path to follow.

If you prefer step-by-step instruction with hands-on projects and mentorship, check out our **DevOps Bootcamp** where we cover Docker, Kubernetes, CI/CD, and basically everything you need in DevOps and cloud space - in a **structured program designed to get you job-ready**.

*No pressure - this roadmap has everything you need to learn independently. The bootcamp is just there if you want the extra support!*All the best on your Docker journey! :)

All the best on your Docker journey! :)



Docker isn't just another tool to learn—it's the bridge between *"works on my laptop*" and *"works everywhere."*

Start with Phase 1, take your time with each concept, and remember: ***every Docker expert started exactly where you are now.***

The journey from "docker run hello-world" to confidently deploying production applications is incredibly rewarding. You've got this! ✊

Remember: **Slow, patient learning is slow at the beginning, but FASTER IN THE LONG RUN.** Build solid foundations and avoid knowledge gaps—it always pays off in the end!