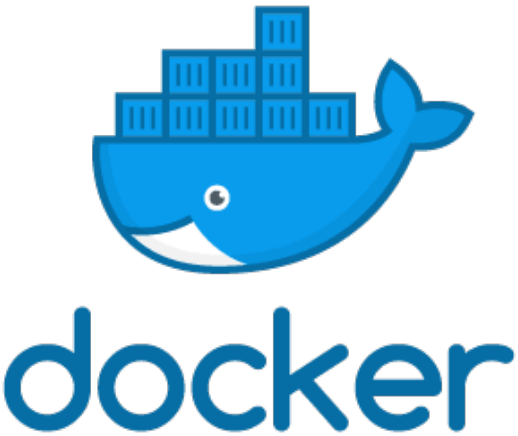
Mastering Docker

Complete and Professional Guide

Containers, Networking, Storage, Registry & Security



Written by: Maryem CHERIF

Cybersecurity Engineer | DevSecOps

November 9, 2025

Advanced Documentation

Contents

Introduction

Docker has become the standard tool for application containerization, revolutionizing DevOps, CI/CD, and cloud deployment practices. This in-depth documentation covers the entire Docker ecosystem, from foundational concepts to advanced production techniques.

Objectives of this Documentation

- Master Docker's architecture and internal mechanisms
- Optimize images and container performance
- Implement production-grade networking and storage strategies
- Secure Docker infrastructure according to industry standards
- Automate deployment with Docker Compose and orchestration

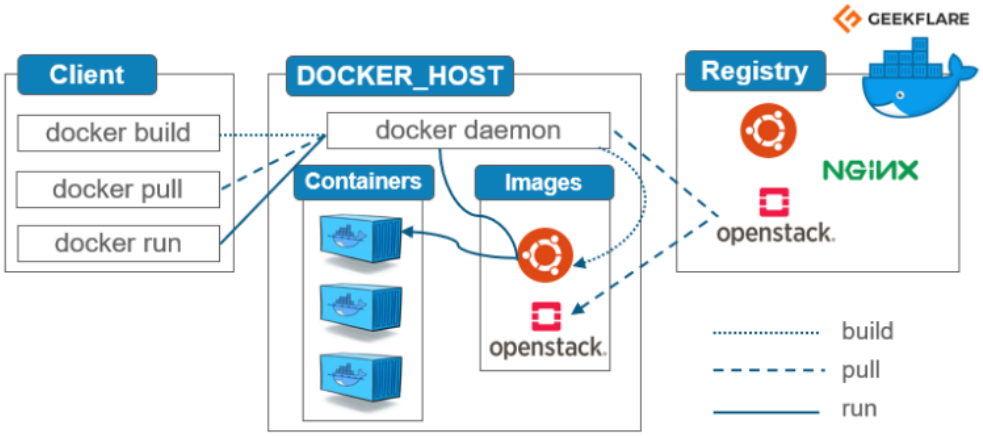


Figure 1: Complete Docker Ecosystem Architecture

Container Fundamentals

Container Concept

A container is a lightweight execution unit that encapsulates an application and its dependencies in an isolated environment. Unlike virtual machines that virtualize hardware, containers share the host operating system kernel, offering better performance and reduced memory footprint.

2.1.1 Containers vs Virtual Machines

Containers	Virtual Machines
Share the host kernel	Have their own kernel
Start in seconds	Start in minutes
Lightweight (MB)	Heavy (GB)
Process-level isolation	Full isolation
Lower resource usage	Higher resource usage

Table 1: Containers vs VMs Comparison

Docker Engine and Daemon

The **Docker Engine** is the core of the containerization system and includes several critical components:

- **dockerd (daemon)**: main service managing container lifecycle, images, volumes, and networks
- **containerd**: high-level runtime managing container execution
- **runc**: low-level OCI-compliant runtime
- **Docker CLI**: command-line interface to interact with the daemon
- **Docker REST API**: enables automation and programmatic integration

Essential Docker Daemon Commands

```
# Check daemon status
sudo systemctl status docker
# Start/Stop daemon
sudo systemctl start docker
sudo systemctl stop docker
# Enable at boot
sudo systemctl enable docker
# View daemon logs
sudo journalctl -u docker -f
# System information
docker info
docker version
```

Namespaces and Cgroups

Docker leverages Linux kernel features for isolation:

2.3.1 Namespaces

Provide resource isolation:

- **PID**: process isolation

- **NET**: isolated network stack
- **MNT**: isolated filesystem
- **UTS**: isolated hostname and domain
- **IPC**: isolated inter-process communication
- **USER**: user mapping

2.3.2 Control Groups (cgroups)

Limit and monitor resource usage:

```
# Limit memory
docker run -m 512m nginx
# Limit CPU
docker run --cpus="1.5" nginx
# Limit disk I/O
docker run --device-write-bps /dev/sda:1mb nginx
```

Advanced Images and Dockerfile

Image Architecture

Docker images follow a layered architecture based on UnionFS. Each Dockerfile instruction creates a new immutable layer.

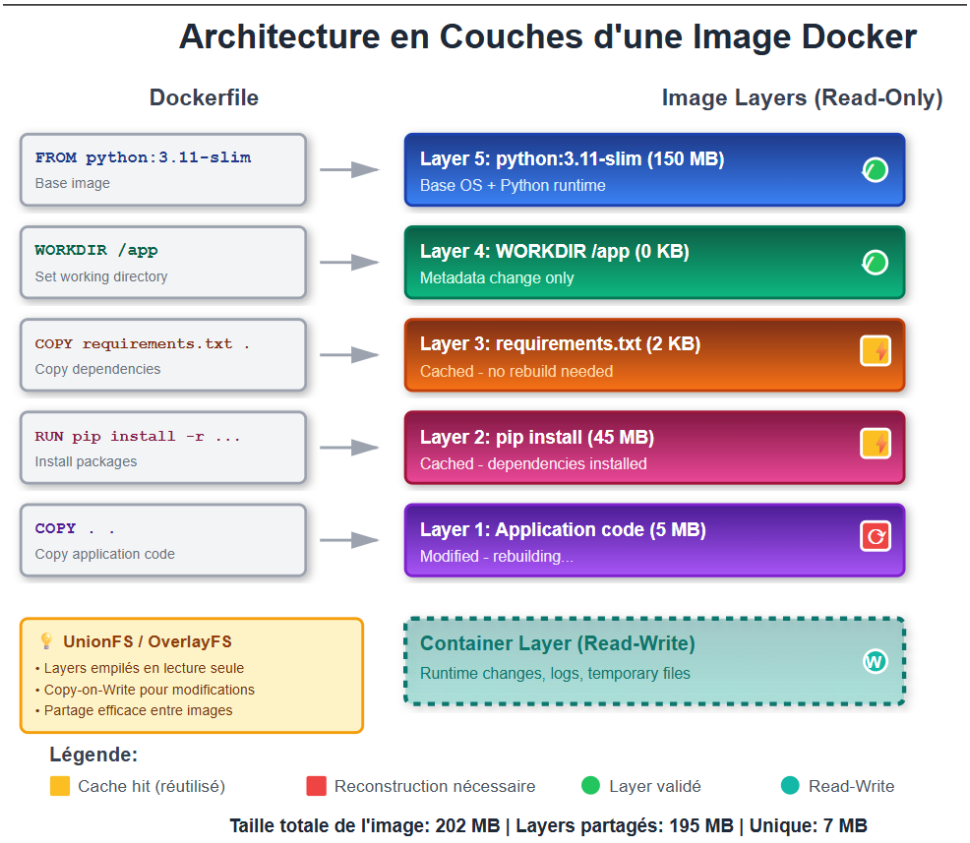


Figure 2: Layered Docker Image Architecture with Caching

Optimized Dockerfile - Complete Example

Listing 1: Multi-stage Optimized Python Dockerfile

```
# Stage 1: Builder
FROM python:3.11-slim AS builder
# Environment variables
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PIP_NO_CACHE_DIR=1 \
    PIP_DISABLE_PIP_VERSION_CHECK=1
WORKDIR /build
# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    gcc \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*
# Install Python dependencies
COPY requirements.txt .
RUN pip wheel --no-cache-dir --wheel-dir /build/wheels \
    -r requirements.txt
# Stage 2: Runtime
FROM python:3.11-slim
# Create non-root user
RUN groupadd -r appuser && useradd -r -g appuser appuser
```

```
WORKDIR /app
# Copy wheels from builder stage
COPY --from=builder /build/wheels /wheels
RUN pip install --no-cache /wheels/*
# Copy application code
COPY --chown=appuser:appuser . .
# Switch to non-root user
USER appuser
# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s \
  CMD python -c "import␣requests;␣requests.get('http://localhost
    ↪ :8000/health')"
# Expose port
EXPOSE 8000
# Entry point
ENTRYPOINT ["python"]
CMD ["app.py"]
```

Dockerfile Best Practices - Complete Guide

1. **Multi-stage builds**: Separate compilation from execution to reduce size
2. **Minimal base images**: Prefer alpine, distroless, or scratch
3. **Layer order**: Place least-changing instructions first
4. **Combine RUN commands**: Reduce layer count
5. **.dockerignore**: Exclude unnecessary files from build context
6. **Non-root user**: Never run as root
7. **Labels and metadata**: Document images with labels
8. **Security scanning**: Integrate into CI/CD pipeline

Example .dockerignore

```
# Git files
.git
.gitignore
# Virtual environments
venv/
env/
*.pyc
__pycache__/
# Documentation
*.md
docs/
# Tests
tests/
*.test
# Local config files
.env
.env.local
docker-compose*.yml
```

Advanced Image Management

```
# Build with custom cache
docker build --cache-from myapp:latest -t myapp:v2.0 .
# Multi-platform build (ARM + AMD64)
docker buildx build --platform linux/amd64,linux/arm64 \
  -t myapp:multiarch --push .
# Inspect an image
docker image inspect nginx:latest
docker history nginx:latest
# Clean unused images
docker image prune -a --filter "until=168h"
# Export/Import images
docker save -o myapp.tar myapp:latest
docker load -i myapp.tar
```

Advanced Docker Networking

Docker Network Types

Docker provides several network types to isolate and manage container communication. Each has specific use cases and characteristics.

Bridge

Bridge: Default private network for containers on the same host. Containers communicate via an isolated network.

Host

Host: Container shares the host's network stack. No network isolation, improves performance.

Overlay

Overlay: Enables communication between containers across different hosts in a Docker Swarm cluster. Ideal for distributed services.

Macvlan

Macvlan: Assigns a unique MAC address to each container, making it appear as a physical device on the local network.

None

None: No network. Container has no external network access and runs in isolation.

Driver	Usage	Scope
bridge	Default private network	local
host	Shares host network stack	local
overlay	Multi-host communication (Swarm)	swarm
macvlan	MAC address assignment	local
none	No network	local

Table 2: Summary of Docker Network Drivers

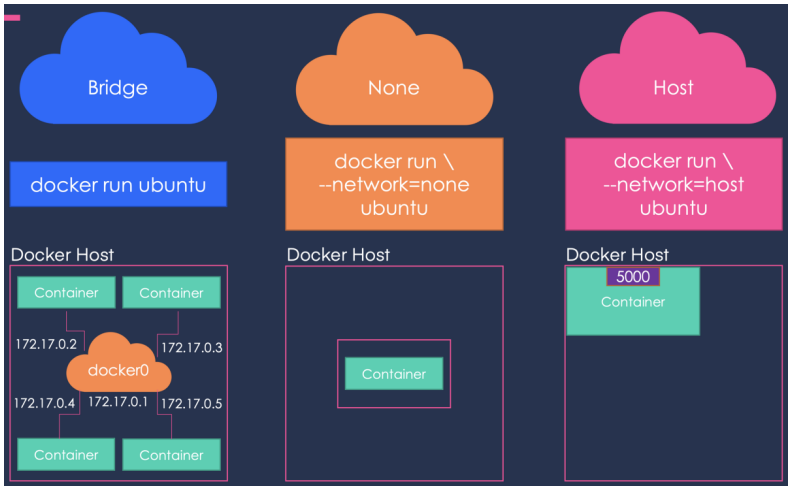


Figure 3: Topology of Docker Network Types

Advanced Network Configuration

```
# Create custom bridge network
docker network create --driver bridge \
```

```
  --subnet=172.20.0.0/16 \
  --ip-range=172.20.240.0/20 \
  --gateway=172.20.0.1 \
  --opt "com.docker.network.bridge.name"="br-custom" \
  my_custom_network
# Overlay network for Swarm
docker network create --driver overlay \
  --attachable \
  --subnet=10.0.9.0/24 \
  my_overlay_network
# Macvlan network (direct physical access)
docker network create -d macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 \
  macvlan_net
# Connect container to multiple networks
docker network connect frontend web_server
docker network connect backend web_server
```

DNS and Service Discovery

Docker provides embedded DNS for name resolution between containers:

```
# Containers resolve by name
docker run -d --name db --network app_net postgres
docker run -d --name api --network app_net myapi
# API can access: postgresql://db:5432
# Custom DNS aliases
docker run -d --name web --network app_net \
  --network-alias webserver \
  --network-alias www nginx
```

Network Inspection and Debugging

```
# Inspect network
docker network inspect my_network
# List connected containers
docker network inspect my_network \
  --format='{{range␣.Containers}}{{.Name}}␣{{end}}'
# Test connectivity
docker exec web ping -c 3 api
# Capture network traffic
docker run --rm --net container:web \
  nicolaka/netshoot tcpdump -i eth0
```

Docker Storage - Advanced Management

Storage Types - Comparison

Type	Advantages	Disadvantages
Volumes	Managed by Docker, performant, backupable	Less direct control
Bind Mounts	Direct host access	Host path dependency
tmpfs	Very fast (RAM)	Volatile data

Table 3: Docker Storage Types Comparison

Volumes - Advanced Operations

```
# Create volume with specific driver
docker volume create --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.100,rw \
  --opt device=:/path/to/share \
  nfs_volume
# Create labeled volume
docker volume create --label env=production \
  --label backup=daily \
  prod_data
# List with filters
docker volume ls --filter label=env=production
# Use volume with permissions
docker run -v myvolume:/data:ro nginx # Read-only
docker run -v myvolume:/data:rw nginx # Read-write
# Share volume between containers
docker run -d --name app1 -v shared_data:/data app1
docker run -d --name app2 -v shared_data:/data app2
```

Backup and Restore - Professional Strategies

```
# Full volume backup with compression
docker run --rm \
  -v my_volume:/source:ro \
  -v $(pwd):/backup \
  alpine \
  tar czf /backup/backup-$(date +%Y%m%d-%H%M%S).tar.gz -C /source
    ↪ .
# Restore volume
docker run --rm \
  -v my_volume:/target \
  -v $(pwd):/backup \
  alpine \
  tar xzf /backup/backup-20250108-120000.tar.gz -C /target
# Clone volume
docker volume create new_volume
```

```
docker run --rm \
  -v old_volume:/source:ro \
  -v new_volume:/target \
  alpine \
  sh -c "cp -av /source/. /target/"
# Backup to S3 (AWS)
docker run --rm \
  -v my_volume:/data:ro \
  -e AWS_ACCESS_KEY_ID \
  -e AWS_SECRET_ACCESS_KEY \
  amazon/aws-cli \
  s3 sync /data s3://my-bucket/backups/$(date +%Y%m%d)/
```

Volume Drivers

Docker supports multiple volume drivers for different backends:

- **local**: local storage (default)
- **nfs**: Network File System
- **cifs/smb**: Windows file shares
- **rexray**: cloud storage (AWS EBS, Azure Disk)
- **convoy**: snapshots and backups
- **flocker**: data migration between hosts

Docker Registry - Private Infrastructure

Registry Types

- **Docker Hub**: official public registry
- **Harbor**: enterprise registry with security scanning
- **Quay.io**: Red Hat registry with advanced features
- **GitHub Container Registry**: integrated with GitHub
- **AWS ECR / Azure ACR / GCP GCR**: native cloud registries
- **Docker Registry (OSS)**: simple open-source registry

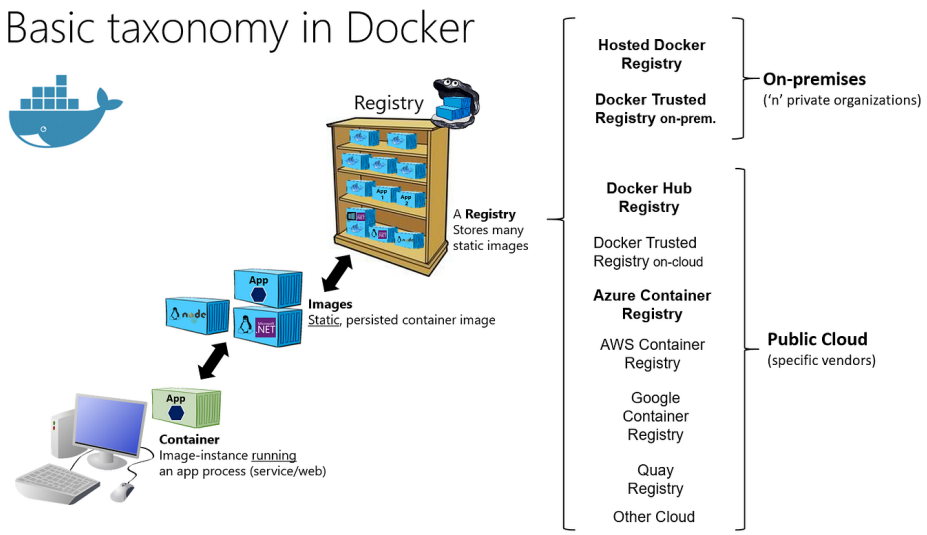


Figure 4: Private Docker Registry Architecture with Reverse Proxy

Secure Private Registry Deployment

Listing 2: Docker Compose for Registry with Nginx

```
# docker-compose.yml
version: '3.8'
services:
  registry:
    image: registry:2
    restart: always
    environment:
      REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
      REGISTRY_AUTH: htpasswd
      REGISTRY_AUTH_HTPASSWD_REALM: Registry
      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
      REGISTRY_STORAGE_DELETE_ENABLED: 'true'
    volumes:
      - registry_data:/data
      - ./auth:/auth
    networks:
      - registry_net
  nginx:
    image: nginx:alpine
    restart: always
    ports:
      - "443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./ssl:/etc/nginx/ssl:ro
    depends_on:
      - registry
    networks:
      - registry_net
```

```
volumes:
  registry_data:
networks:
  registry_net:
```

Nginx Configuration for Registry

Listing 3: Nginx Configuration with SSL

```
# nginx.conf
events {
    worker_connections 1024;
}
http {
    upstream docker-registry {
        server registry:5000;
    }
    server {
        listen 443 ssl http2;
        server_name registry.example.com;
        ssl_certificate /etc/nginx/ssl/cert.pem;
        ssl_certificate_key /etc/nginx/ssl/key.pem;
        ssl_protocols TLSv1.2 TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
        client_max_body_size 0;
        chunked_transfer_encoding on;
        location / {
            proxy_pass http://docker-registry;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For
               ↪ $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_read_timeout 900;
        }
    }
}
```

Authentication and Access Control

```
# Create htpasswd file
docker run --rm --entrypoint htpasswd \
  httpd:2 -Bbn admin password > auth/htpasswd
# Login to registry
docker login registry.example.com
# Tag and push image
docker tag myapp:latest registry.example.com/myapp:v1.0
docker push registry.example.com/myapp:v1.0
# Pull from registry
```

```
docker pull registry.example.com/myapp:v1.0
# List images in registry
curl -X GET https://registry.example.com/v2/_catalog \
  -u admin:password
```

Docker Compose - Multi-Container Orchestration

Complete Example - 3-Tier Application

Listing 4: docker-compose.yml for Full Stack

```
version: '3.8'
services:
  # PostgreSQL Database
  database:
    image: postgres:15-alpine
    restart: unless-stopped
    environment:
      POSTGRES_DB: ${DB_NAME:-myapp}
      POSTGRES_USER: ${DB_USER:-postgres}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql:ro
    networks:
      - backend
    healthcheck:
      test: ["CMD-SHELL", "pg_isready␣-U␣postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
  # Redis Cache
  cache:
    image: redis:7-alpine
    restart: unless-stopped
    command: redis-server --appendonly yes --requirepass ${
      ↪ REDIS_PASSWORD}
    volumes:
      - redis_data:/data
    networks:
      - backend
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 5
  # Backend API
  api:
    build:
      context: ./backend
```

```
      dockerfile: Dockerfile
      target: production
    restart: unless-stopped
    environment:
      DATABASE_URL: postgresql://${DB_USER}:${DB_PASSWORD}
        ↪ @database:5432/${DB_NAME}
      REDIS_URL: redis://:${REDIS_PASSWORD}@cache:6379/0
      JWT_SECRET: ${JWT_SECRET}
    depends_on:
      database:
        condition: service_healthy
      cache:
        condition: service_healthy
    networks:
      - backend
      - frontend
    deploy:
      replicas: 2
      resources:
        limits:
          cpus: '1'
          memory: 512M
  # Web Frontend
  web:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    restart: unless-stopped
    environment:
      API_URL: http://api:8000
    depends_on:
      - api
    networks:
      - frontend
  # Nginx Reverse Proxy
  nginx:
    image: nginx:alpine
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
      - ./nginx/ssl:/etc/nginx/ssl:ro
      - ./nginx/logs:/var/log/nginx
    depends_on:
      - web
      - api
    networks:
      - frontend
volumes:
```

```
  postgres_data:
    driver: local
  redis_data:
    driver: local
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
    internal: true
```

Advanced Docker Compose Commands

```
# Start with logs
docker-compose up -d && docker-compose logs -f
# Rebuild and restart specific service
docker-compose up -d --build --force-recreate api
# Scale service
docker-compose up -d --scale api=5
# Execute command in service
docker-compose exec api python manage.py migrate
docker-compose exec database psql -U postgres
# Use multiple env files
docker-compose --env-file .env.prod \
  -f docker-compose.yml \
  -f docker-compose.prod.yml up -d
# View merged config
docker-compose config
# View resource usage
docker-compose top
```

Docker Security - Complete Guide

Security Principles

8.1.1 1. Image Security

```
# Scan with Trivy
trivy image --severity HIGH,CRITICAL nginx:latest
# Scan with Docker Scout
docker scout cves nginx:latest
# Scan with Snyk
snyk container test nginx:latest
# Sign images with Docker Content Trust
export DOCKER_CONTENT_TRUST=1
docker push registry.example.com/myapp:signed
```

8.1.2 2. Isolation and Capabilities

```
# Drop capabilities
docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx
# Read-only mode
docker run --read-only --tmpfs /tmp nginx
# AppArmor / SELinux
docker run --security-opt apparmor=docker-default nginx
docker run --security-opt label=level:s0:c100,c200 nginx
# Disable new privileges
docker run --security-opt=no-new-privileges nginx
```

8.1.3 3. User Namespaces

```
# Configure daemon for user namespaces
# /etc/docker/daemon.json
{
  "userns-remap": "default"
}
# Restart Docker
sudo systemctl restart docker
```

Docker Security Checklist

Security Checklist

- Always use official or verified images
- Scan images before deployment
- Use specific tags, never `latest`
- Run containers as non-root user
- Limit resources (CPU, memory, I/O)
- Use private networks
- Enable Docker Content Trust
- Regularly update Docker Engine
- Audit with Docker Bench Security
- Encrypt sensitive data
- Implement secret rotation
- Monitor logs and metrics

Docker Secrets (Swarm)

```
# Create secret
echo "mysecretpassword" | docker secret create db_password -
# Use in service
docker service create --name myapp \
  --secret db_password \
  myimage
# Access secret in container
# Available at /run/secrets/db_password
```

Monitoring and Logging

Monitoring with Prometheus and Grafana

This stack enables performance and availability monitoring:

- **Prometheus:** Collects and stores metrics (CPU, memory, etc.)
- **Grafana:** Visualizes metrics via interactive dashboards
- **cAdvisor:** Docker metrics exporter
- **Node Exporter:** System metrics exporter

Listing 5: Complete Monitoring Stack with Docker Compose

```
version: '3.8'
services:
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.retention.time=30d'
    ports:
      - "9090:9090"
    networks:
      - monitoring
  grafana:
    image: grafana/grafana:latest
    environment:
      GF_SECURITY_ADMIN_PASSWORD: ${GRAFANA_PASSWORD}
      GF_INSTALL_PLUGINS: grafana-piechart-panel
    volumes:
      - grafana_data:/var/lib/grafana
      - ./grafana/dashboards:/etc/grafana/provisioning/dashboards
    ports:
      - "3000:3000"
```

```
    depends_on:
      - prometheus
    networks:
      - monitoring
  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    privileged: true
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:ro
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
      - /dev/disk/:/dev/disk:ro
    ports:
      - "8080:8080"
    networks:
      - monitoring
  node_exporter:
    image: prom/node-exporter:latest
    command:
      - '--path.rootfs=/host'
    volumes:
      - /:/host:ro,rslave
    ports:
      - "9100:9100"
    networks:
      - monitoring
volumes:
  prometheus_data:
  grafana_data:
networks:
  monitoring:
    driver: bridge
```

Centralized Logging with ELK Stack

The ELK stack centralizes and visualizes container logs:

- **Elasticsearch:** Stores and indexes logs
- **Logstash:** Collects, transforms, and sends logs
- **Kibana:** Web interface for log analysis

Listing 6: ELK Stack Docker Compose

```
version: '3.8'
services:
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.11.0
    environment:
```

```
      - discovery.type=single-node
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
      - xpack.security.enabled=false
    volumes:
      - elasticsearch_data:/usr/share/elasticsearch/data
    ports:
      - "9200:9200"
    networks:
      - elk
  logstash:
    image: docker.elastic.co/logstash/logstash:8.11.0
    volumes:
      - ./logstash/pipeline:/usr/share/logstash/pipeline
    ports:
      - "5000:5000/tcp"
      - "5000:5000/udp"
      - "9600:9600"
    depends_on:
      - elasticsearch
    networks:
      - elk
  kibana:
    image: docker.elastic.co/kibana/kibana:8.11.0
    environment:
      ELASTICSEARCH_HOSTS: http://elasticsearch:9200
    ports:
      - "5601:5601"
    depends_on:
      - elasticsearch
    networks:
      - elk
volumes:
  elasticsearch_data:
networks:
  elk:
    driver: bridge
```

Docker Logging Drivers Configuration

```
# Syslog logging
docker run --log-driver=syslog \
  --log-opt syslog-address=udp://logserver:514 \
  --log-opt tag="{{.Name}}" \
  nginx
# Fluentd logging
docker run --log-driver=fluentd \
  --log-opt fluentd-address=localhost:24224 \
  --log-opt tag="docker.{{.Name}}" \
  nginx
# JSON logging with rotation
```

```
docker run --log-driver=json-file \
  --log-opt max-size=10m \
  --log-opt max-file=3 \
  nginx
# Global config in daemon.json
# /etc/docker/daemon.json
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "5"
  }
}
```

Performance and Optimization

Image Optimization

Technique	Description	Gain
Multi-stage build	Separate build and runtime	50-80%
Alpine images	Minimal base	60-90%
Distroless	No full OS	70-85%
Layer caching	Reuse layers	Build time
.dockerignore	Reduce context	Build time

Table 4: Image Optimization Techniques

Container Benchmarking

```
# CPU stress test
docker run --rm -it progrium/stress --cpu 2 --timeout 60s
# Network performance
docker run --rm -it networkstatic/iperf3 -c server_ip
# Disk I/O test
docker run --rm -v /data:/data \
  ubuntu:latest dd if=/dev/zero of=/data/test bs=1M count=1000
# cAdvisor profiling
curl http://localhost:8080/api/v2.0/stats?type=docker&count=1
# Real-time stats
docker stats --format "table␣{{.Name}}\t{{.CPUPerc}}\t{{.MemUsage
  ↪ }}"
```

Docker Daemon Tuning

Listing 7: Optimized daemon.json

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  },
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ],
  "max-concurrent-downloads": 10,
  "max-concurrent-uploads": 10,
  "default-ulimits": {
    "nofile": {
      "Name": "nofile",
      "Hard": 64000,
      "Soft": 64000
    }
  },
  "live-restore": true,
  "userland-proxy": false,
  "icc": false,
  "default-address-pools": [
    {
      "base": "172.80.0.0/16",
      "size": 24
    }
  ]
}
```

CI/CD with Docker

GitLab CI Pipeline

Listing 8: Complete GitLab CI Pipeline

```
stages:
  - build
  - test
  - scan
  - deploy
variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: "/certs"
  IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
before_script:
  - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
    ↪ $CI_REGISTRY
```

```
build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
  only:
    - main
    - develop
test:
  stage: test
  image: $IMAGE_TAG
  script:
    - pytest tests/
    - coverage report
  coverage: '/TOTAL.*\s+(\d+%)$/'
security_scan:
  stage: scan
  image: aquasec/trivy:latest
  script:
    - trivy image --exit-code 1 --severity CRITICAL $IMAGE_TAG
  allow_failure: false
deploy_staging:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | ssh-add -
  script:
    - ssh user@staging-server "docker␣pull␣$IMAGE_TAG"
    - ssh user@staging-server "docker-compose␣up␣-d"
  environment:
    name: staging
    url: https://staging.example.com
  only:
    - develop
deploy_production:
  stage: deploy
  image: alpine:latest
  script:
    - ssh user@prod-server "docker␣pull␣$IMAGE_TAG"
    - ssh user@prod-server "docker␣stack␣deploy␣-c␣docker-compose.
      ↪ yml␣app"
  environment:
    name: production
    url: https://example.com
  only:
    - main
```

```
    when: manual
```

GitHub Actions Pipeline

Listing 9: GitHub Actions Docker CI/CD

```yaml
name: Docker CI/CD
on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]
env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3
      - name: Log in to Container Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=sha,prefix={{branch}}-
            type=semver,pattern={{version}}
      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

```
      - name: Run Trivy vulnerability scanner
        uses: aquasecurity/trivy-action@master
        with:
          image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${{
            ↪ github.sha }}
          format: 'sarif'
          output: 'trivy-results.sarif'
      - name: Upload Trivy results to GitHub Security
        uses: github/codeql-action/upload-sarif@v2
        with:
          sarif_file: 'trivy-results.sarif'
```

Docker Swarm - Native Orchestration

Swarm Cluster Initialization

```
# Initialize manager
docker swarm init --advertise-addr 192.168.1.100
# Add workers
docker swarm join --token SWMTKN-... 192.168.1.100:2377
# List nodes
docker node ls
# Promote worker to manager
docker node promote worker-node-1
# Label nodes
docker node update --label-add environment=production node-1
docker node update --label-add type=database node-2
```

Service Deployment

```
# Create simple service
docker service create --name web \
  --replicas 3 \
  --publish 80:80 \
  nginx:alpine
# Service with constraints
docker service create --name db \
  --constraint 'node.labels.type==database' \
  --mount type=volume,source=db_data,target=/var/lib/postgresql/
    ↪ data \
  postgres:15
# Rolling update
docker service create --name api \
  --replicas 5 \
  --update-parallelism 2 \
  --update-delay 10s \
  --update-failure-action rollback \
  myapi:latest
```

```
# Scale service
docker service scale web=10
# Update service
docker service update --image nginx:latest web
# Rollback
docker service rollback web
```

Complete Swarm Stack

Listing 10: docker-stack.yml for Production

```
version: '3.8'
services:
  web:
    image: nginx:alpine
    ports:
      - "80:80"
      - "443:443"
    deploy:
      replicas: 3
      update_config:
        parallelism: 1
        delay: 10s
      restart_policy:
        condition: on-failure
        max_attempts: 3
      placement:
        constraints:
          - node.role == worker
          - node.labels.environment == production
    networks:
      - frontend
    configs:
      - source: nginx_config
        target: /etc/nginx/nginx.conf
    secrets:
      - ssl_cert
      - ssl_key
  api:
    image: myregistry.com/api:v2.0
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.25'
          memory: 256M
      update_config:
```

```
        parallelism: 2
        delay: 10s
        failure_action: rollback
      rollback_config:
        parallelism: 2
        delay: 5s
      restart_policy:
        condition: any
        delay: 5s
        max_attempts: 3
    networks:
      - frontend
      - backend
    secrets:
      - db_password
      - jwt_secret
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s
  database:
    image: postgres:15-alpine
    deploy:
      replicas: 1
      placement:
        constraints:
          - node.labels.type == database
      restart_policy:
        condition: on-failure
    volumes:
      - db_data:/var/lib/postgresql/data
    networks:
      - backend
    secrets:
      - db_password
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
configs:
  nginx_config:
    file: ./nginx.conf
secrets:
  db_password:
    external: true
  jwt_secret:
    external: true
  ssl_cert:
    external: true
  ssl_key:
    external: true
```

```
volumes:
  db_data:
    driver: local
networks:
  frontend:
    driver: overlay
  backend:
    driver: overlay
    internal: true
```

```
# Deploy stack
docker stack deploy -c docker-stack.yml myapp
# List stacks
docker stack ls
# List services
docker stack services myapp
# View service logs
docker service logs -f myapp_api
# Remove stack
docker stack rm myapp
```

Troubleshooting and Debugging

Diagnostic Commands

```
# Inspect container
docker inspect container_name
docker inspect --format='{{.State.Health.Status}}' container_name
# View processes
docker top container_name
# Real-time stats
docker stats --no-stream
# Docker events
docker events --since '30m' --filter 'type=container'
# Logs with timestamps
docker logs --timestamps --since 30m container_name
# Follow logs
docker logs -f --tail 100 container_name
# Execute shell
docker exec -it container_name /bin/sh
# Copy files
docker cp container_name:/app/logs/app.log ./
docker cp ./config.yml container_name:/app/config/
# Compare filesystem
docker diff container_name
# Export filesystem
docker export container_name > container_fs.tar
```

Common Issues and Solutions

Troubleshooting Guide

1. Container restarting in loop

```
docker logs --tail 50 container_name
docker update --restart=no container_name
docker inspect --format='{{json .State.Health}}'
    ↪ container_name
```

2. Network issues

```
docker exec container1 ping container2
docker network inspect network_name
docker network prune
```

3. Disk space shortage

```
docker system df
docker system prune -a --volumes
docker image prune -a --filter "until=168h"
```

4. Slow performance

```
docker stats container_name
docker update --cpus="1.5" --memory="1g" container_name
```

Migration to Docker

Containerization Strategy

1. **Assessment**: Identify candidate applications
2. **Stateless first**: Start with stateless apps
3. **Dependencies**: Map all external dependencies
4. **Data**: Plan persistence strategy
5. **Configuration**: Externalize configs (12-factor)
6. **Testing**: Validate in staging
7. **Monitoring**: Implement observability
8. **Progressive rollout**: Deploy in phases

Pre-Migration Checklist

Application architecture documented

Dependencies identified and versioned

Data management strategy defined

Rollback plan prepared

Performance baselines established

Load tests planned

Operations documentation created

Teams trained

Resources and Tools

Essential Tools

Tool	Usage	URL
Docker Desktop	Local dev (Mac/Windows)	docker.com
Portainer	GUI management	portainer.io
Dive	Image analysis	github.com/wagoodman/dive
Hadolint	Dockerfile linting	hadolint.github.io
Docker Bench	Security audit	github.com/docker/docker-bench-security
Trivy	Vulnerability scanning	aquasecurity.github.io/trivy
Lazydocker	TUI for Docker	github.com/jesseduffield/lazydocker
Watchtower	Auto-update containers	containrrr.dev/watchtower

Table 5: Recommended Docker Ecosystem Tools

Maintenance Commands

```
# Security audit
docker run --rm -it \
  --net host --pid host --userns host \
  --cap-add audit_control \
  -v /var/lib:/var/lib \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v /etc:/etc \
  docker/docker-bench-security
# Image analysis with Dive
dive nginx:alpine
# Scheduled cleanup (cron)
# 0 2 * * * docker system prune -af --filter "until=168h"
```

Essential Definitions

Container Lightweight, portable software unit encapsulating an application and its dependencies.

Docker Image
: Immutable file containing code, libraries, and configurations needed to run an application.

Layer
: Each Dockerfile instruction creates an immutable layer (UnionFS principle).

Volume
: Docker-managed data persistence mechanism, independent of container lifecycle.

Bind Mount
: Direct mount of a host directory into a container.

Registry
: Centralized repository for storing and distributing Docker images.

Daemon (dockerd)
: System service managing containers, images, volumes, and networks.

Namespace
: Linux kernel feature enabling resource isolation (PID, NET, MNT, etc.).

Cgroup
: Linux mechanism to limit and monitor resource usage (CPU, RAM, I/O).

Overlay Network
: Multi-host virtual network used in Docker Swarm.

Healthcheck
: Periodic instruction verifying container health.

Multi-stage Build
: Technique separating compilation and execution to reduce image size.

Docker Content Trust
: Cryptographic signature system ensuring image integrity.

User Namespace
: UID/GID isolation between host and container for enhanced security.

Swarm Mode
: Docker's native orchestration mode for managing container clusters.

Why Each Step?

Multi-stage build
: Reduces final image size by removing compilation tools.

Non-root user
: Prevents privilege escalation in case of compromise.

Security scanning
: Detects vulnerabilities early (Shift-Left Security).

Named volumes
: Ensures data persistence and simplifies backup/rotation.

Private networks (internal: true)
Isolates backend services from external access.

Healthcheck Enables orchestrator to auto-restart failing containers.

Specific tags (v1.2.3)
Ensures reproducibility and avoids unpredictable behavior.

Docker Content Trust
Verifies image integrity and authenticity.

Resource limits (cgroups)
Prevents container from monopolizing host resources.

Monitoring (Prometheus + Grafana)
Provides real-time anomaly detection.

Automated CI/CD
Ensures every change is tested, scanned, and deployed reproducibly.

Rolling updates (Swarm)
Enables zero-downtime deployments.

Glossary

Container Isolated execution unit containing an application and dependencies

Image Immutable template used to create containers

Dockerfile Script describing image construction

Registry Image storage and distribution system

Volume Data persistence mechanism

Network Virtual infrastructure connecting containers

Compose Tool for defining multi-container applications

Swarm Docker's native orchestration solution

Layer Image layer representing a Dockerfile instruction

Daemon System service managing containers (dockerd)

OCI Open Container Initiative - container standard

Overlay Network
Multi-host virtual network

Health Check
Test verifying container health status

Conclusion and Outlook

Docker has transformed how we develop, deploy, and operate modern applications. This comprehensive documentation covers all aspects needed to master Docker in a professional environment.

Key Takeaways

- Containerization improves portability and deployment consistency
- Security must be integrated from image build phase
- Image optimization reduces costs and improves performance
- Monitoring and logging are essential in production
- Orchestration (Swarm/Kubernetes) is required at scale
- DevOps best practices apply naturally with Docker

Future Developments

The Docker ecosystem continues to evolve with:

- **WebAssembly**: Native Wasm support in containers
- **Rootless mode**: Default privilege-less execution
- **BuildKit**: Next-generation, faster builder
- **Docker Extensions**: Plugin ecosystem for Docker Desktop
- **Supply Chain Security**: Full artifact traceability

Next Steps

To deepen your knowledge:

1. Practice with real projects
2. Explore Kubernetes for advanced orchestration
3. Implement complete CI/CD pipelines
4. Contribute to open-source ecosystem
5. Earn certifications (Docker Certified Associate)

Documentation written by Maryem CHERIF
Cybersecurity Engineer | DevSecOps
- November 9, 2025

For questions or suggestions: cherif.maryem24@gmail.com