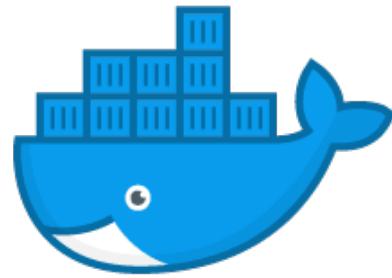


Mastering Docker

Complete and Professional Guide

Containers, Networking, Storage, Registry & Security



docker

Written by: Maryem CHERIF

Cybersecurity Engineer | DevSecOps

November 9, 2025

Advanced Documentation

Contents

1	Introduction	3
1.1	Objectives of this Documentation	3
2	Container Fundamentals	3
2.1	Container Concept	3
2.1.1	Containers vs Virtual Machines	4
2.2	Docker Engine and Daemon	4
2.3	Namespaces and Cgroups	4
2.3.1	Namespaces	4
2.3.2	Control Groups (cgroups)	5
3	Advanced Images and Dockerfile	5
3.1	Image Architecture	5
3.2	Optimized Dockerfile - Complete Example	6
3.3	Dockerfile Best Practices - Complete Guide	7
3.4	Advanced Image Management	8
4	Advanced Docker Networking	8
4.1	Docker Network Types	8
4.2	Advanced Network Configuration	9
4.3	DNS and Service Discovery	10
4.4	Network Inspection and Debugging	10
5	Docker Storage - Advanced Management	10
5.1	Storage Types - Comparison	11
5.2	Volumes - Advanced Operations	11
5.3	Backup and Restore - Professional Strategies	11
5.4	Volume Drivers	12
6	Docker Registry - Private Infrastructure	12
6.1	Registry Types	12
6.2	Secure Private Registry Deployment	13
6.3	Nginx Configuration for Registry	14
6.4	Authentication and Access Control	14
7	Docker Compose - Multi-Container Orchestration	15
7.1	Complete Example - 3-Tier Application	15
7.2	Advanced Docker Compose Commands	17
8	Docker Security - Complete Guide	17
8.1	Security Principles	17
8.1.1	1. Image Security	17
8.1.2	2. Isolation and Capabilities	18
8.1.3	3. User Namespaces	18
8.2	Docker Security Checklist	18
8.3	Docker Secrets (Swarm)	19

9 Monitoring and Logging	19
9.1 Monitoring with Prometheus and Grafana	19
9.2 Centralized Logging with ELK Stack	20
9.3 Docker Logging Drivers Configuration	21
10 Performance and Optimization	22
10.1 Image Optimization	22
10.2 Container Benchmarking	22
11 Docker Daemon Tuning	22
12 CI/CD with Docker	23
12.1 GitLab CI Pipeline	23
12.2 GitHub Actions Pipeline	25
13 Docker Swarm - Native Orchestration	26
13.1 Swarm Cluster Initialization	26
13.2 Service Deployment	26
13.3 Complete Swarm Stack	27
14 Troubleshooting and Debugging	29
14.1 Diagnostic Commands	29
14.2 Common Issues and Solutions	30
15 Migration to Docker	30
15.1 Containerization Strategy	30
15.2 Pre-Migration Checklist	30
16 Resources and Tools	31
16.1 Essential Tools	31
16.2 Maintenance Commands	31
17 Essential Definitions	31
18 Why Each Step?	32
19 Glossary	33
20 Conclusion and Outlook	34
20.1 Key Takeaways	34
20.2 Future Developments	34
20.3 Next Steps	34

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

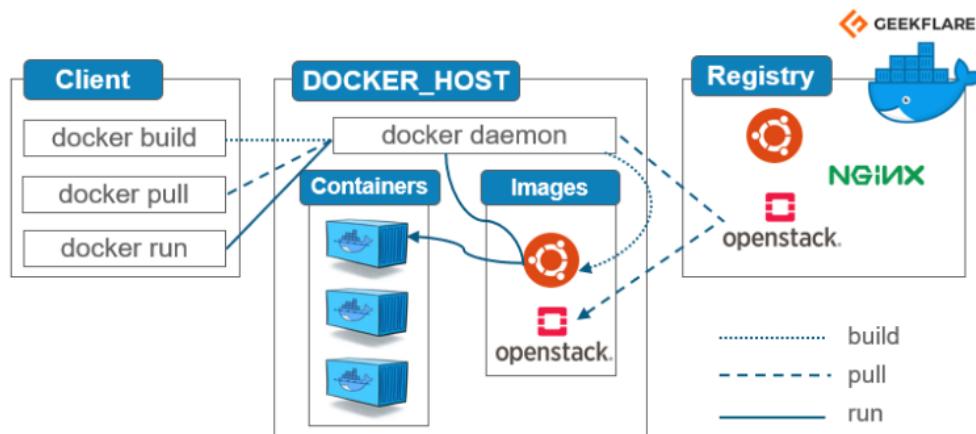


Figure 1: Complete Docker Ecosystem Architecture

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

```

1 # Check daemon status
2 sudo systemctl status docker
3 # Start/Stop daemon
4 sudo systemctl start docker
5 sudo systemctl stop docker
6 # Enable at boot
7 sudo systemctl enable docker
8 # View daemon logs
9 sudo journalctl -u docker -f
10 # System information
11 docker info
12 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:

```
1 # Limit memory
2 docker run -m 512m nginx
3 # Limit CPU
4 docker run --cpus="1.5" nginx
5 # Limit disk I/O
6 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.

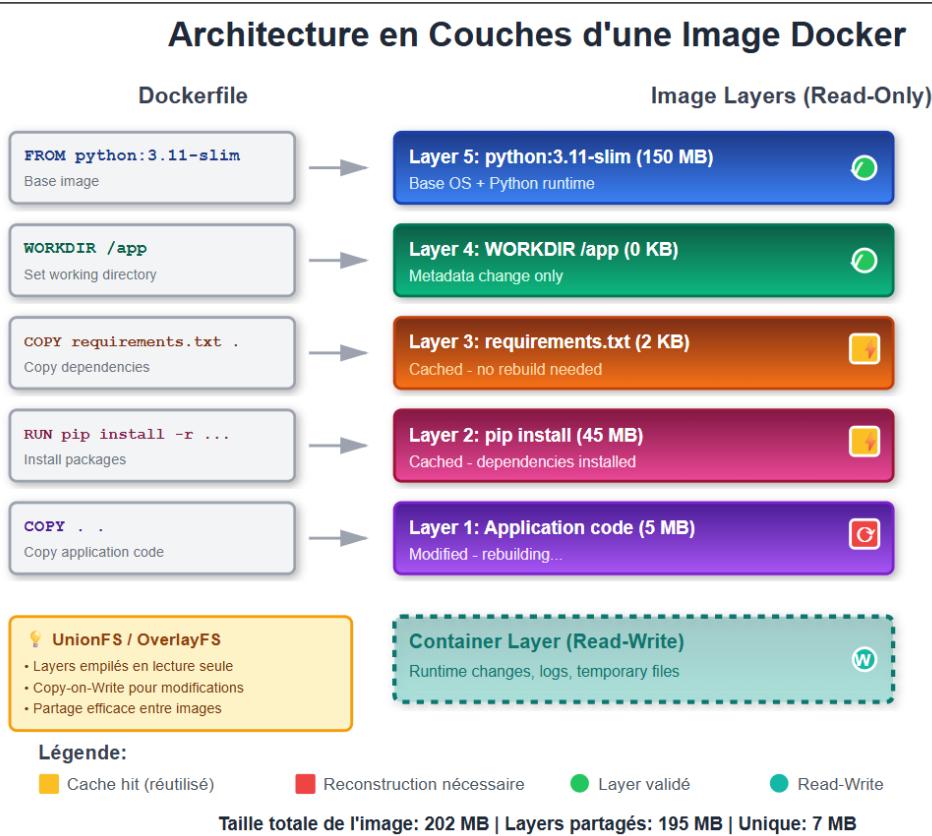


Figure 2: Layered Docker Image Architecture with Caching

Optimized Dockerfile - Complete Example

Listing 1: Multi-stage Optimized Python Dockerfile

```

1 # Stage 1: Builder
2 FROM python:3.11-slim AS builder
3 # Environment variables
4 ENV PYTHONDONTWRITEBYTECODE=1 \
5     PYTHONUNBUFFERED=1 \
6     PIP_NO_CACHE_DIR=1 \
7     PIP_DISABLE_PIP_VERSION_CHECK=1
8 WORKDIR /build
9 # Install system dependencies
10 RUN apt-get update && apt-get install -y --no-install-recommends \
11     gcc \
12     libpq-dev \
13     && rm -rf /var/lib/apt/lists/*
14 # Install Python dependencies
15 COPY requirements.txt .
16 RUN pip wheel --no-cache-dir --wheel-dir /build/wheels \
17     -r requirements.txt
18 # Stage 2: Runtime
19 FROM python:3.11-slim
20 # Create non-root user
21 RUN groupadd -r appuser && useradd -r -g appuser appuser

```

```
22 WORKDIR /app
23 # Copy wheels from builder stage
24 COPY --from=builder /build/wheels /wheels
25 RUN pip install --no-cache /wheels/*
26 # Copy application code
27 COPY --chown=appuser:appuser . .
28 # Switch to non-root user
29 USER appuser
30 # Health check
31 HEALTHCHECK --interval=30s --timeout=3s --start-period=5s \
32   CMD python -c "import requests; requests.get('http://localhost
33   ↪ :8000/health')"
34 # Expose port
35 EXPOSE 8000
36 # Entry point
37 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

```

1 # Git files
2 .git
3 .gitignore
4 # Virtual environments
5 venv/
6 env/
7 *.pyc
8 __pycache__/
9 # Documentation
10 *.md
11 docs/
12 # Tests
13 tests/
14 *.test
15 # Local config files
16 .env
17 .env.local
18 docker-compose*.yml

```

Advanced Image Management

```

1 # Build with custom cache
2 docker build --cache-from myapp:latest -t myapp:v2.0 .
3 # Multi-platform build (ARM + AMD64)
4 docker buildx build --platform linux/amd64,linux/arm64 \
5   -t myapp:multiarch --push .
6 # Inspect an image
7 docker image inspect nginx:latest
8 docker history nginx:latest
9 # Clean unused images
10 docker image prune -a --filter "until=168h"
11 # Export/Import images
12 docker save -o myapp.tar myapp:latest
13 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

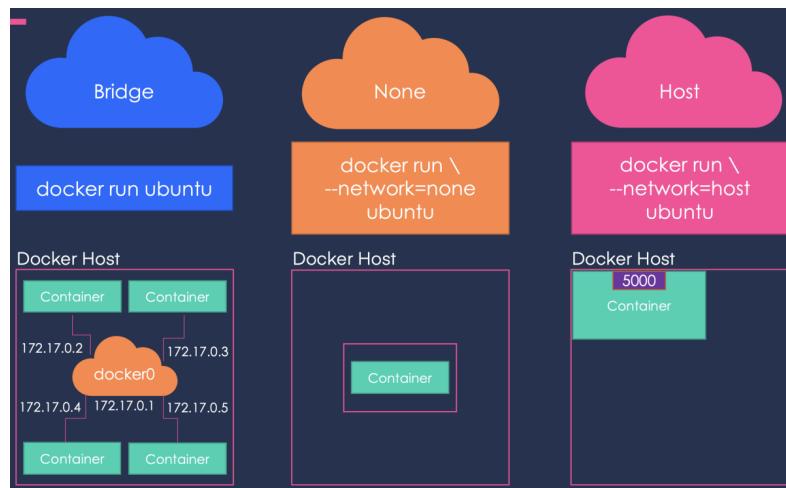


Figure 3: Topology of Docker Network Types

Advanced Network Configuration

```

1 # Create custom bridge network
2 docker network create --driver bridge \

```

```

3   --subnet=172.20.0.0/16 \
4   --ip-range=172.20.240.0/20 \
5   --gateway=172.20.0.1 \
6   --opt "com.docker.network.bridge.name"="br-custom" \
7   my_custom_network
8 # Overlay network for Swarm
9 docker network create --driver overlay \
10  --attachable \
11  --subnet=10.0.9.0/24 \
12  my_overlay_network
13 # Macvlan network (direct physical access)
14 docker network create -d macvlan \
15  --subnet=192.168.1.0/24 \
16  --gateway=192.168.1.1 \
17  -o parent=eth0 \
18  macvlan_net
19 # Connect container to multiple networks
20 docker network connect frontend web_server
21 docker network connect backend web_server

```

DNS and Service Discovery

Docker provides embedded DNS for name resolution between containers:

```

1 # Containers resolve by name
2 docker run -d --name db --network app_net postgres
3 docker run -d --name api --network app_net myapi
4 # API can access: postgresql://db:5432
5 # Custom DNS aliases
6 docker run -d --name web --network app_net \
7   --network-alias webserver \
8   --network-alias www nginx

```

Network Inspection and Debugging

```

1 # Inspect network
2 docker network inspect my_network
3 # List connected containers
4 docker network inspect my_network \
5   --format='{{range .Containers}}{{.Name}} {{end}}'
6 # Test connectivity
7 docker exec web ping -c 3 api
8 # Capture network traffic
9 docker run --rm --net container:web \
10   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

```

1 # Create volume with specific driver
2 docker volume create --driver local \
3   --opt type=nfs \
4   --opt o=addr=192.168.1.100,rw \
5   --opt device=/path/to/share \
6   nfs_volume
7 # Create labeled volume
8 docker volume create --label env=production \
9   --label backup=daily \
10  prod_data
11 # List with filters
12 docker volume ls --filter label=env=production
13 # Use volume with permissions
14 docker run -v myvolume:/data:ro nginx # Read-only
15 docker run -v myvolume:/data:rw nginx # Read-write
16 # Share volume between containers
17 docker run -d --name app1 -v shared_data:/data app1
18 docker run -d --name app2 -v shared_data:/data app2

```

Backup and Restore - Professional Strategies

```

1 # Full volume backup with compression
2 docker run --rm \
3   -v my_volume:/source:ro \
4   -v $(pwd):/backup \
5   alpine \
6   tar czf /backup/backup-$(date +%Y%m%d-%H%M%S).tar.gz -C /source
7   ↘ .
8 # Restore volume
9 docker run --rm \
10  -v my_volume:/target \
11  -v $(pwd):/backup \
12  alpine \
13  tar xzf /backup/backup-20250108-120000.tar.gz -C /target
14 # Clone volume
15 docker volume create new_volume

```

```
15 docker run --rm \
16   -v old_volume:/source:ro \
17   -v new_volume:/target \
18   alpine \
19   sh -c "cp -av /source/. /target/"
20 # Backup to S3 (AWS)
21 docker run --rm \
22   -v my_volume:/data:ro \
23   -e AWS_ACCESS_KEY_ID \
24   -e AWS_SECRET_ACCESS_KEY \
25   amazon/aws-cli \
26   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

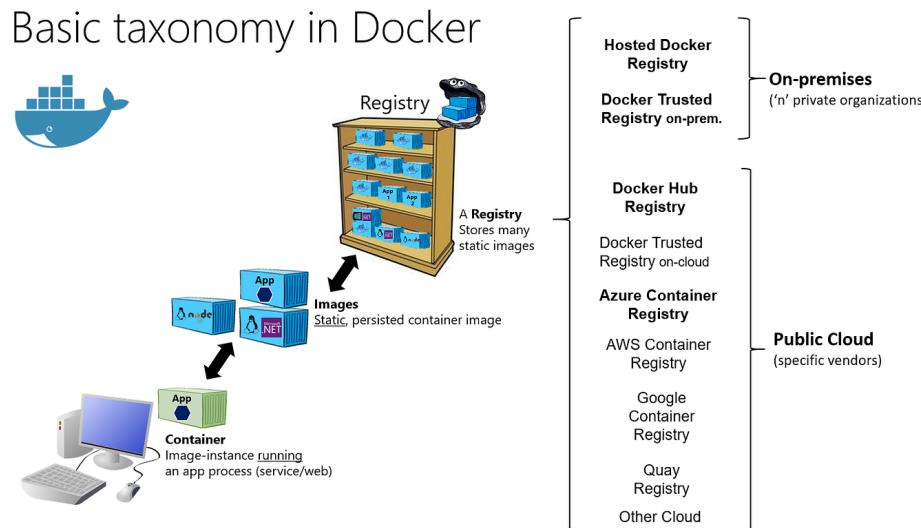


Figure 4: Private Docker Registry Architecture with Reverse Proxy

Secure Private Registry Deployment

Listing 2: Docker Compose for Registry with Nginx

```

1 # docker-compose.yml
2 version: '3.8'
3 services:
4   registry:
5     image: registry:2
6     restart: always
7     environment:
8       REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
9       REGISTRY_AUTH: htpasswd
10      REGISTRY_AUTH_HTPASSWD_REALM: Registry
11      REGISTRY_AUTH_HTPASSWD_PATH: /auth/htpasswd
12      REGISTRY_STORAGE_DELETE_ENABLED: 'true'
13     volumes:
14       - registry_data:/data
15       - ./auth:/auth
16     networks:
17       - registry_net
18   nginx:
19     image: nginx:alpine
20     restart: always
21     ports:
22       - "443:443"
23     volumes:
24       - ./nginx.conf:/etc/nginx/nginx.conf:ro
25       - ./ssl:/etc/nginx/ssl:ro
26     depends_on:
27       - registry
28     networks:
29       - registry_net

```

```

30 volumes:
31   registry_data:
32 networks:
33   registry_net:

```

Nginx Configuration for Registry

Listing 3: Nginx Configuration with SSL

```

1 # nginx.conf
2 events {
3     worker_connections 1024;
4 }
5 http {
6     upstream docker-registry {
7         server registry:5000;
8     }
9     server {
10        listen 443 ssl http2;
11        server_name registry.example.com;
12        ssl_certificate /etc/nginx/ssl/cert.pem;
13        ssl_certificate_key /etc/nginx/ssl/key.pem;
14        ssl_protocols TLSv1.2 TLSv1.3;
15        ssl_ciphers HIGH:!aNULL:!MD5;
16        client_max_body_size 0;
17        chunked_transfer_encoding on;
18        location / {
19            proxy_pass http://docker-registry;
20            proxy_set_header Host $host;
21            proxy_set_header X-Real-IP $remote_addr;
22            proxy_set_header X-Forwarded-For
23                ↳ $proxy_add_x_forwarded_for;
24            proxy_set_header X-Forwarded-Proto $scheme;
25            proxy_read_timeout 900;
26        }
27    }
}

```

Authentication and Access Control

```

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

```

```

10 docker pull registry.example.com/myapp:v1.0
11 # List images in registry
12 curl -X GET https://registry.example.com/v2/_catalog \
13   -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

```

```
39     dockerfile: Dockerfile
40     target: production
41     restart: unless-stopped
42     environment:
43       DATABASE_URL: postgresql://${DB_USER}:${DB_PASSWORD}
44         ↢ @database:5432/${DB_NAME}
45       REDIS_URL: redis://:${REDIS_PASSWORD}@cache:6379/0
46       JWT_SECRET: ${JWT_SECRET}
47     depends_on:
48       database:
49         condition: service_healthy
50       cache:
51         condition: service_healthy
52     networks:
53       - backend
54       - frontend
55   deploy:
56     replicas: 2
57     resources:
58       limits:
59         cpus: '1'
60         memory: 512M
61 # Web Frontend
62 web:
63   build:
64     context: ./frontend
65     dockerfile: Dockerfile
66   restart: unless-stopped
67   environment:
68     API_URL: http://api:8000
69   depends_on:
70     - api
71   networks:
72     - frontend
73 # Nginx Reverse Proxy
74 nginx:
75   image: nginx:alpine
76   restart: unless-stopped
77   ports:
78     - "80:80"
79     - "443:443"
80   volumes:
81     - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
82     - ./nginx/ssl:/etc/nginx/ssl:ro
83     - ./nginx/logs:/var/log/nginx
84   depends_on:
85     - web
86     - api
87   networks:
88     - frontend
89   volumes:
```

```

89  postgres_data:
90    driver: local
91  redis_data:
92    driver: local
93  networks:
94    frontend:
95      driver: bridge
96    backend:
97      driver: bridge
98      internal: true

```

Advanced Docker Compose Commands

```

1 # Start with logs
2 docker-compose up -d && docker-compose logs -f
3 # Rebuild and restart specific service
4 docker-compose up -d --build --force-recreate api
5 # Scale service
6 docker-compose up -d --scale api=5
7 # Execute command in service
8 docker-compose exec api python manage.py migrate
9 docker-compose exec database psql -U postgres
10 # Use multiple env files
11 docker-compose --env-file .env.prod \
12   -f docker-compose.yml \
13   -f docker-compose.prod.yml up -d
14 # View merged config
15 docker-compose config
16 # View resource usage
17 docker-compose top

```

Docker Security - Complete Guide

Security Principles

8.1.1 1. Image Security

```

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

```

8.1.2 2. Isolation and Capabilities

```

1 # Drop capabilities
2 docker run --cap-drop=ALL --cap-add=NET_BIND_SERVICE nginx
3 # Read-only mode
4 docker run --read-only --tmpfs /tmp nginx
5 # AppArmor / SELinux
6 docker run --security-opt apparmor=docker-default nginx
7 docker run --security-opt label=level:s0:c100,c200 nginx
8 # Disable new privileges
9 docker run --security-opt=no-new-privileges nginx

```

8.1.3 3. User Namespaces

```

1 # Configure daemon for user namespaces
2 # /etc/docker/daemon.json
3 {
4     "userns-remap": "default"
5 }
6 # Restart Docker
7 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)

```

1 # Create secret
2 echo "mysecretpassword" | docker secret create db_password -
3 # Use in service
4 docker service create --name myapp \
5   --secret db_password \
6   myimage
7 # Access secret in container
8 # 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

```

1 version: '3.8'
2 services:
3   prometheus:
4     image: prom/prometheus:latest
5     volumes:
6       - ./prometheus.yml:/etc/prometheus/prometheus.yml
7       - prometheus_data:/prometheus
8     command:
9       - '--config.file=/etc/prometheus/prometheus.yml'
10      - '--storage.tsdb.retention.time=30d'
11     ports:
12       - "9090:9090"
13     networks:
14       - monitoring
15   grafana:
16     image: grafana/grafana:latest
17     environment:
18       GF_SECURITY_ADMIN_PASSWORD: ${GRAFANA_PASSWORD}
19       GF_INSTALL_PLUGINS: grafana-piechart-panel
20     volumes:
21       - grafana_data:/var/lib/grafana
22       - ./grafana/dashboards:/etc/grafana/provisioning/dashboards
23     ports:
24       - "3000:3000"

```

```

25   depends_on:
26     - prometheus
27   networks:
28     - monitoring
29   cadvisor:
30     image: gcr.io/cadvisor/cadvisor:latest
31     privileged: true
32   volumes:
33     - /:/rootfs:ro
34     - /var/run:/var/run:ro
35     - /sys:/sys:ro
36     - /var/lib/docker/:/var/lib/docker:ro
37     - /dev/disk/:/dev/disk:ro
38   ports:
39     - "8080:8080"
40   networks:
41     - monitoring
42   node_exporter:
43     image: prom/node-exporter:latest
44     command:
45       - '--path.rootfs=/host'
46     volumes:
47       - /:/host:ro,rslave
48     ports:
49       - "9100:9100"
50     networks:
51       - monitoring
52   volumes:
53     prometheus_data:
54     grafana_data:
55   networks:
56     monitoring:
57       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

```

1 version: '3.8'
2 services:
3   elasticsearch:
4     image: docker.elastic.co/elasticsearch/elasticsearch:8.11.0
5     environment:

```

```

6      - discovery.type=single-node
7      - "ES_JAVA_OPTS=-Xms512m\u2014-Xmx512m"
8      - xpack.security.enabled=false
9
10     volumes:
11       - elasticsearch_data:/usr/share/elasticsearch/data
12     ports:
13       - "9200:9200"
14     networks:
15       - elk
16   logstash:
17     image: docker.elastic.co/logstash/logstash:8.11.0
18     volumes:
19       - ./logstash/pipeline:/usr/share/logstash/pipeline
20     ports:
21       - "5000:5000/tcp"
22       - "5000:5000/udp"
23       - "9600:9600"
24     depends_on:
25       - elasticsearch
26     networks:
27       - elk
28   kibana:
29     image: docker.elastic.co/kibana/kibana:8.11.0
30     environment:
31       ELASTICSEARCH_HOSTS: http://elasticsearch:9200
32     ports:
33       - "5601:5601"
34     depends_on:
35       - elasticsearch
36     networks:
37       - elk
38     volumes:
39       elasticsearch_data:
40     networks:
41       elk:
42         driver: bridge

```

Docker Logging Drivers Configuration

```

1 # Syslog logging
2 docker run --log-driver=syslog \
3   --log-opt syslog-address=udp://logserver:514 \
4   --log-opt tag="{{.Name}}" \
5   nginx
6 # Fluentd logging
7 docker run --log-driver=fluentd \
8   --log-opt fluentd-address=localhost:24224 \
9   --log-opt tag="docker.{{.Name}}" \
10  nginx
11 # JSON logging with rotation

```

```

12 docker run --log-driver=json-file \
13   --log-opt max-size=10m \
14   --log-opt max-file=3 \
15   nginx
16 # Global config in daemon.json
17 # /etc/docker/daemon.json
18 {
19   "log-driver": "json-file",
20   "log-opt": {
21     "max-size": "10m",
22     "max-file": "5"
23   }
24 }
```

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

```

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

Docker Daemon Tuning

Listing 7: Optimized daemon.json

```

1  {
2      "log-driver": "json-file",
3      "log-opt": {
4          "max-size": "10m",
5          "max-file": "3"
6      },
7      "storage-driver": "overlay2",
8      "storage-opt": [
9          "overlay2.override_kernel_check=true"
10     ],
11     "max-concurrent-downloads": 10,
12     "max-concurrent-uploads": 10,
13     "default-ulimits": {
14         "nofile": {
15             "Name": "nofile",
16             "Hard": 64000,
17             "Soft": 64000
18         }
19     },
20     "live-restore": true,
21     "userland-proxy": false,
22     "icc": false,
23     "default-address-pools": [
24         {
25             "base": "172.80.0.0/16",
26             "size": 24
27         }
28     ]
29 }
```

CI/CD with Docker

GitLab CI Pipeline

Listing 8: Complete GitLab CI Pipeline

```

1 stages:
2   - build
3   - test
4   - scan
5   - deploy
6 variables:
7   DOCKER_DRIVER: overlay2
8   DOCKER_TLS_CERTDIR: "/certs"
9   IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA
10 before_script:
11   - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
12     ↘ $CI_REGISTRY
```

```
12 build:
13   stage: build
14   image: docker:latest
15   services:
16     - docker:dind
17   script:
18     - docker build -t $IMAGE_TAG .
19     - docker push $IMAGE_TAG
20   only:
21     - main
22     - develop
23 test:
24   stage: test
25   image: $IMAGE_TAG
26   script:
27     - pytest tests/
28     - coverage report
29   coverage: '/TOTAL.*\s+(\d+%)$/'
30 security_scan:
31   stage: scan
32   image: aquasec/trivy:latest
33   script:
34     - trivy image --exit-code 1 --severity CRITICAL $IMAGE_TAG
35   allow_failure: false
36 deploy_staging:
37   stage: deploy
38   image: alpine:latest
39   before_script:
40     - apk add --no-cache openssh-client
41     - eval $(ssh-agent -s)
42     - echo "$SSH_PRIVATE_KEY" | ssh-add -
43   script:
44     - ssh user@staging-server "docker pull $IMAGE_TAG"
45     - ssh user@staging-server "docker-compose up -d"
46 environment:
47   name: staging
48   url: https://staging.example.com
49 only:
50   - develop
51 deploy_production:
52   stage: deploy
53   image: alpine:latest
54   script:
55     - ssh user@prod-server "docker pull $IMAGE_TAG"
56     - ssh user@prod-server "docker stack deploy -c docker-compose.
57       ↪ yml app"
58   environment:
59     name: production
60     url: https://example.com
61 only:
62   - main
```

62 when: manual

GitHub Actions Pipeline

Listing 9: GitHub Actions Docker CI/CD

```

1 name: Docker CI/CD
2 on:
3   push:
4     branches: [ main, develop ]
5   pull_request:
6     branches: [ main ]
7 env:
8   REGISTRY: ghcr.io
9   IMAGE_NAME: ${{ github.repository }}
10 jobs:
11   build-and-push:
12     runs-on: ubuntu-latest
13     permissions:
14       contents: read
15       packages: write
16     steps:
17       - name: Checkout repository
18         uses: actions/checkout@v4
19       - name: Set up Docker Buildx
20         uses: docker/setup-buildx-action@v3
21       - name: Log in to Container Registry
22         uses: docker/login-action@v3
23         with:
24           registry: ${{ env.REGISTRY }}
25           username: ${{ github.actor }}
26           password: ${{ secrets.GITHUB_TOKEN }}
27       - name: Extract metadata
28         id: meta
29         uses: docker/metadata-action@v5
30         with:
31           images: ${{ env.REGISTRY }}/{{ env.IMAGE_NAME }}
32           tags: |
33             type=ref,event=branch
34             type=sha,prefix={{branch}}-
35             type=semver,pattern={{version}}
36       - name: Build and push Docker image
37         uses: docker/build-push-action@v5
38         with:
39           context: .
40           push: true
41           tags: ${{ steps.meta.outputs.tags }}
42           labels: ${{ steps.meta.outputs.labels }}
43           cache-from: type=gha
44           cache-to: type=gha,mode=max

```

```

45      - name: Run Trivy vulnerability scanner
46        uses: aquasecurity/trivy-action@master
47        with:
48          image-ref: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}:${
49            ↪   github.sha }
50          format: 'sarif'
51          output: 'trivy-results.sarif'
52      - name: Upload Trivy results to GitHub Security
53        uses: github/codeql-action/upload-sarif@v2
54        with:
55          sarif_file: 'trivy-results.sarif'
```

Docker Swarm - Native Orchestration

Swarm Cluster Initialization

```

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

Service Deployment

```

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

```

18 # Scale service
19 docker service scale web=10
20 # Update service
21 docker service update --image nginx:latest web
22 # Rollback
23 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:

```

```
40      parallelism: 2
41      delay: 10s
42      failure_action: rollback
43      rollback_config:
44          parallelism: 2
45          delay: 5s
46      restart_policy:
47          condition: any
48          delay: 5s
49          max_attempts: 3
50
51  networks:
52      - frontend
53      - backend
54
55  secrets:
56      - db_password
57      - jwt_secret
58
59  healthcheck:
60      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
61      interval: 30s
62      timeout: 10s
63      retries: 3
64      start_period: 40s
65
66  database:
67      image: postgres:15-alpine
68
69  deploy:
70      replicas: 1
71      placement:
72          constraints:
73              - node.labels.type == database
74
75  restart_policy:
76      condition: on-failure
77
78  volumes:
79      - db_data:/var/lib/postgresql/data
80
81  networks:
82      - backend
83
84  secrets:
85      - db_password
86
87  environment:
88      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
89
90  configs:
91      nginx_config:
92          file: ./nginx.conf
93
94  secrets:
95      db_password:
96          external: true
97      jwt_secret:
98          external: true
99      ssl_cert:
100         external: true
101     ssl_key:
102         external: true
```

```

91 volumes:
92   db_data:
93     driver: local
94 networks:
95   frontend:
96     driver: overlay
97   backend:
98     driver: overlay
99     internal: true

```

```

1 # Deploy stack
2 docker stack deploy -c docker-stack.yml myapp
3 # List stacks
4 docker stack ls
5 # List services
6 docker stack services myapp
7 # View service logs
8 docker service logs -f myapp_api
9 # Remove stack
10 docker stack rm myapp

```

Troubleshooting and Debugging

Diagnostic Commands

```

1 # Inspect container
2 docker inspect container_name
3 docker inspect --format='{{.State.Health.Status}}' container_name
4 # View processes
5 docker top container_name
6 # Real-time stats
7 docker stats --no-stream
8 # Docker events
9 docker events --since '30m' --filter 'type=container'
10 # Logs with timestamps
11 docker logs --timestamps --since 30m container_name
12 # Follow logs
13 docker logs -f --tail 100 container_name
14 # Execute shell
15 docker exec -it container_name /bin/sh
16 # Copy files
17 docker cp container_name:/app/logs/app.log ./
18 docker cp ./config.yml container_name:/app/config/
19 # Compare filesystem
20 docker diff container_name
21 # Export filesystem
22 docker export container_name > container_fs.tar

```

Common Issues and Solutions

Troubleshooting Guide

1. Container restarting in loop

```

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

```

2. Network issues

```

1 docker exec container1 ping container2
2 docker network inspect network_name
3 docker network prune

```

3. Disk space shortage

```

1 docker system df
2 docker system prune -a --volumes
3 docker image prune -a --filter "until=168h"

```

4. Slow performance

```

1 docker stats container_name
2 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

```

1 # Security audit
2 docker run --rm -it \
3   --net host --pid host --userns host \
4   --cap-add audit_control \
5   -v /var/lib:/var/lib \
6   -v /var/run/docker.sock:/var/run/docker.sock \
7   -v /etc:/etc \
8   docker/docker-bench-security
9 # Image analysis with Dive
10 dive nginx:alpine
11 # Scheduled cleanup (cron)
12 # 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