Introduction to Docker

Containerization for Modern Development

Simon Da Silva

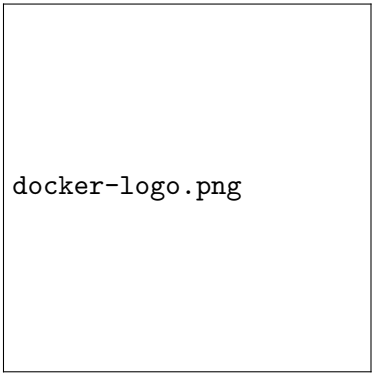
CESI

January 27, 2026

Agenda

What is Docker?

- World's leading **container platform**
- Packages applications with all dependencies
- Creates standardized units called **containers**
- Eliminates "works on my machine" problems



Why Use Docker?

Key Benefits

- **Consistency** – Same behavior everywhere
- **Isolation** – No dependency conflicts
- **Portability** – Run anywhere (laptop, cloud, server)
- **Efficiency** – Lightweight, starts in seconds
- **Scalability** – Easy horizontal scaling
- **Version Control** – Images are versioned

Containers vs Virtual Machines

Virtual Machines	Containers
Full guest OS	Share host kernel
GB of RAM/disk	MB of disk space
Minutes to start	Seconds to start
Heavy isolation	Process-level isolation
Complete OS included	Only app + dependencies

Containers are lightweight and fast!

Architecture Comparison

Virtual Machines

App A	App B	App C
Bins/Libs	Bins/Libs	Bins/Libs
Guest OS	Guest OS	Guest OS
Hypervisor		
Host OS		
Infrastructure		

Containers

App A	App B	App C
Bins/Libs	Bins/Libs	Bins/Libs
Docker Engine		
Host OS		
Infrastructure		

Essential Docker Terms

Image	Read-only template with app code, libraries, dependencies
Container	Running instance of an image
Dockerfile	Text file with instructions to build an image
Docker Hub	Public registry for Docker images
Volume	Persistent data storage for containers
Network	Virtual networks connecting containers
Docker Compose	Tool for multi-container applications

Docker Architecture

Client-Server Model

- **Docker Client** – CLI commands (`docker run`, etc.)
- **Docker Daemon** – Manages images, containers, networks
- **Docker Registry** – Stores Docker images (Docker Hub)

How `docker run` Works

1. Client sends command to daemon
2. Daemon checks if image exists locally
3. If not, pulls from registry
4. Creates container from image
5. Starts container and executes command

Working with Images

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

# List all local images
$ docker images

# Remove an image
$ docker rmi nginx

# Build an image from Dockerfile
$ docker build -t myapp .
```

Running Containers

```
# Run a container (basic)
$ docker run nginx

# Run in detached mode (background)
$ docker run -d nginx

# Run with a name
$ docker run -d --name my-nginx nginx

# Run with port mapping
$ docker run -d -p 8080:80 --name web nginx

# Run interactively
$ docker run -it ubuntu bash
```

Managing Containers

```
# List running containers
$ docker ps

# List all containers (including stopped)
$ docker ps -a

# Stop / Start / Restart
$ docker stop my-nginx
$ docker start my-nginx
$ docker restart my-nginx

# Remove a container
$ docker rm my-nginx

# Execute command in running container
$ docker exec -it my-nginx bash

# View logs
```

Dockerfile Instructions

FROM	Base image to build upon
RUN	Execute commands during build
COPY	Copy files from host to image
WORKDIR	Set working directory
ENV	Set environment variables
EXPOSE	Document container ports (documentation only!)
CMD	Default command (can be overridden)
ENTRYPOINT	Configure container as executable

Example Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 5000

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

Note: Copy requirements.txt first for better layer caching!

Building and Running

```
# Build the image
$ docker build -t my-python-app .

# Run the container
$ docker run -d -p 5000:5000 my-python-app

# Test it
$ curl http://localhost:5000
```

Layer Caching

Docker caches each layer. If a layer hasn't changed, Docker reuses it. This is why we copy `requirements.txt` before the application code.

Network Types

bridge	Default for standalone containers (same host)
host	Container uses host's network directly
none	No networking
overlay	Multi-host networking (Docker Swarm)

Custom Networks

- Better isolation
- Automatic DNS resolution (containers communicate by name!)
- Recommended for multi-container applications

Networking Commands

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

# Run containers on the network
$ docker run -d --name web1 --network my-network nginx
$ docker run -d --name web2 --network my-network nginx

# Containers can now ping each other by name!
$ docker exec web1 ping web2

# Inspect the network
$ docker network inspect my-network

# List networks
$ docker network ls
```

Port Mapping

```
# No port mapping (not accessible from host)
$ docker run -d --name web1 nginx

# Random host port
$ docker run -d -p 80 --name web2 nginx

# Specific host port
$ docker run -d -p 8080:80 --name web3 nginx

# Check port mappings
$ docker port web3
```

Important

`EXPOSE` in Dockerfile is documentation only!
You must use `-p` to actually publish ports.

Volume Types

Problem

Containers are ephemeral – data is lost when they're removed!

Named Volumes Managed by Docker (recommended for production)

Bind Mounts Mount host directory into container (dev use)

tmpfs Mounts Temporary in-memory storage

Volume Commands

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

# List volumes
$ docker volume ls

# Use volume with container
$ docker run -d -v my-data:/data alpine

# Use bind mount (current directory)
$ docker run -d -v $(pwd):/app alpine

# Remove unused volumes
$ docker volume prune

# Inspect volume
$ docker volume inspect my-data
```

What is Docker Compose?

Definition

Tool for defining and running **multi-container** applications using a YAML file.

Benefits

- Define entire application stack in one file
- Single command to start/stop all services
- Automatic network creation and service discovery
- Easy environment configuration
- Version controlled infrastructure

docker-compose.yml Example

```yaml
services:
  web:
    build: ./frontend
    ports:
      - "3000:3000"
    depends_on:
      - db
    environment:
      - DATABASE_URL=postgresql://user:pass@db/mydb

  db:
    image: postgres:14-alpine
    volumes:
      - postgres-data:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=pass

volumes:
```

Docker Compose Commands

```
# Start all services
$ docker compose up

# Start in background
$ docker compose up -d

# Rebuild and start
$ docker compose up --build

# Stop and remove everything
$ docker compose down

# Stop and remove (including volumes)
$ docker compose down -v

# View logs
$ docker compose logs -f
```

Health Checks

```yaml
services:
  db:
    image: postgres:14-alpine
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U user"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 30s

  web:
    depends_on:
      db:
        condition: service_healthy
```

Why Health Checks?

depends_on alone only waits for container to start, not for the service to be ready!

Dockerfile Best Practices

- ✔ Use official base images
- ✔ Use specific image tags, not `latest`
- ✔ Minimize number of layers
- ✔ Use `.dockerignore` to exclude files
- ✔ Don't run containers as root
- ✔ Use multi-stage builds for smaller images
- ✔ Pin dependency versions
- ✔ Copy dependencies file before application code

Docker Compose Best Practices

- ✅ Use `depends_on` with health checks
- ✅ Define named volumes for persistence
- ✅ Use environment variables for configuration
- ✅ Set restart policies (`unless-stopped`)
- ✅ Use networks to isolate services
- ✅ Keep secrets in `.env` files (not in compose file!)
- ✅ Never commit `.env` files to version control

Example .dockerignore

```
# Version control
.git
.gitignore

# Dependencies
node_modules
__pycache__
*.pyc

# Environment
.env
.venv

# IDE
.vscode
.idea

# Logs
```

Common Issues & Solutions

Permission Denied

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

Docker Daemon Not Running

```
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

Useful Debug Commands

```
# View container resource usage
$ docker stats

# Inspect container details
$ docker inspect <container>

# Check container processes
$ docker top <container>

# Copy files from container
$ docker cp <container>:/path/file .

# View Docker disk usage
$ docker system df -v

# Clean up everything
$ docker system prune -a --volumes
```

Command Cheat Sheet

Images

```
docker images
docker pull <image>
docker build -t <name> .
docker rmi <image>
```

Containers

```
docker ps / docker ps -a
docker run <image>
docker stop/start <container>
docker rm <container>
docker logs <container>
docker exec -it <c> bash
```

Docker Compose

```
docker compose up [-d]
docker compose down [-v]
docker compose logs -f
docker compose ps
docker compose build
```

Cleanup

```
docker system prune -a
docker volume prune
docker network prune
```

Summary

What You've Learned

- What Docker is and why it matters
- Difference between containers and VMs
- Essential Docker commands
- Building images with Dockerfiles
- Docker networking and volumes
- Multi-container apps with Docker Compose
- Best practices and troubleshooting

Resources

- Docker Documentation: `https://docs.docker.com`
- Docker Hub: `https://hub.docker.com`
- Docker GitHub: `https://github.com/docker`
- Compose Reference: `https://docs.docker.com/compose/compose-file/`

Questions?

Thank You!

Time for hands-on practice!