Docker - Introduction and Basics

Table of Contents

What is Docker?

Docker is an open-source platform that automates the deployment, scaling, and management of applications using containerization technology. It packages applications and their dependencies into standardized units called containers.

Key Features of Docker:

- **Portability**: Run anywhere - development, testing, production
- **Lightweight**: Shares host OS kernel, uses minimal resources
- **Fast**: Containers start in seconds
- **Isolated**: Each container runs independently
- **Scalable**: Easy to scale up or down

Real-World Analogy:

Think of Docker like shipping containers in logistics: - Standard size and shape - Can be loaded on any ship, truck, or train - Contents are isolated and protected - Easy to track and manage

Example:

```
# Docker allows you to do this:
docker run -d -p 80:80 nginx

# And boom! You have a web server running in seconds
```

Why We Need Docker?

Problem 1: "It Works on My Machine" Syndrome

Traditional Scenario:

```
Developer: "The app works perfectly on my laptop!"
QA Team: "It's broken on our test server"
```

```
Operations: "It crashes in production"
```

Root Causes: - Different OS versions - Missing dependencies - Different library versions - Configuration differences - Environment variables mismatch

Problem 2: Dependency Hell

Example Without Docker:

```
Application A requires:
- Python 3.8
- Node.js 14
- PostgreSQL 12
- Redis 5.0

Application B requires:
- Python 3.11
- Node.js 18
- PostgreSQL 15
- Redis 7.0

How do you run both on the same server?
```

Problem 3: Resource Wastage with Virtual Machines

Traditional VM Approach:

```
Server (64GB RAM, 16 CPU cores)
   VM1 (16GB RAM, 4 cores) - Runs App A
   VM2 (16GB RAM, 4 cores) - Runs App B
   VM3 (16GB RAM, 4 cores) - Runs App C
   VM4 (16GB RAM, 4 cores) - Runs App D

Each VM runs a full OS = Lots of wasted resources!
```

Docker Solutions:

Solution 1: Consistent Environment

```dockerfile
# Define your environment once
FROM node:18
WORKDIR /app
COPY package.json .
RUN npm install
COPY . .
CMD ["npm", "start"]

# Now it works EVERYWHERE!
```

Solution 2: Isolated Dependencies

```
# Run multiple versions without conflict
docker run -d --name app-a python:3.8
docker run -d --name app-b python:3.11

# Each container has its own environment
```

Solution 3: Efficient Resource Usage

```
Server (64GB RAM, 16 CPU cores)
  Container 1 (2GB RAM) - App A
  Container 2 (1GB RAM) - App B
  Container 3 (3GB RAM) - App C
  Container 4 (2GB RAM) - App D
  ...many more containers

All share the same OS kernel = More efficient!
```

Benefits Summary:

Benefit	Description	Example
Consistency	Same environment everywhere	Dev = Test = Production
Isolation	Apps don't interfere	Multiple Node.js versions
Portability	Run on any Docker host	AWS, Azure, Local
Speed	Fast startup and deployment	Start in milliseconds
Scalability	Easy to scale horizontally	Run 100 containers easily
Version Control	Track image versions	Roll back easily

What is a Container?

Definition

A **container** is a lightweight, standalone, executable package that includes everything needed to run a piece of software: - Code - Runtime - System tools - System libraries - Settings

Container Characteristics:

1. Lightweight

```
Virtual Machine: 1-2 GB+
Container: 50-200 MB

Why? Containers share the host OS kernel!
```

2. Isolated

```
# Each container has its own:
- Filesystem
- Process space
- Network interface
- Users and groups

# Example: Two containers, same port, no conflict
docker run -d -p 8080:80 nginx  # Container 1
docker run -d -p 8081:80 nginx  # Container 2
```

3. Portable

```
# Build on Mac
docker build -t myapp .

# Run on Linux server
docker run myapp

# Run on Windows
docker run myapp

# Same result everywhere!
```

Container vs Process

Regular Process:

```
Your App → Shares everything with host
           (filesystem, network, users)
```

Container:

```
Your App → Isolated environment
             Own filesystem
             Own network
             Own processes
             Controlled resources
```

Real-World Example:

Scenario: Running a Node.js application

Without Container:

```bash
# Install Node.js globally
sudo apt install nodejs

# Install dependencies
npm install

# Run app
node app.js

# Problem: What if another app needs different Node.js version?
```

With Container:

```dockerfile
# Dockerfile
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["node", "app.js"]
```

```bash
# Build
docker build -t my-node-app .

# Run
docker run -p 3000:3000 my-node-app

# Benefits:
#   Specific Node.js version
#   All dependencies included
#   Doesn't affect host system
#   Can run multiple versions simultaneously
```

Container Lifecycle:

```
1. CREATE     → Container is created from image
                docker create nginx

2. START      → Container starts running
                docker start container_id

3. RUNNING    → Container is actively running
                docker ps

4. STOP       → Container stops gracefully
```

```
               docker stop container_id

5. REMOVE    → Container is deleted
               docker rm container_id
```

Example:

```
# Complete lifecycle example
docker create --name my-nginx nginx
# Output: container_id_12345

docker start my-nginx
# Output: my-nginx

docker ps
# Shows running container

docker stop my-nginx
# Gracefully stops container

docker rm my-nginx
# Removes container completely
```

Before Docker - The Traditional Problem

The Dark Ages of Application Deployment (Pre-2013)

Problem 1: Manual Server Configuration **Traditional Setup**:

```
# On Server 1
ssh admin@server1.company.com

# Install dependencies manually
sudo apt update
sudo apt install python3.8
sudo apt install postgresql
sudo apt install redis-server
sudo apt install nginx

# Configure each service
sudo nano /etc/postgresql/postgresql.conf
sudo nano /etc/nginx/nginx.conf
sudo nano /etc/redis/redis.conf

# Deploy application
git clone https://github.com/company/app.git
```

```bash
cd app
pip install -r requirements.txt
python manage.py migrate
python manage.py runserver

# Problem: Need to repeat this on 50 servers?
```

Problem 2: Inconsistent Environments **The Conversation**:

```
Developer (Monday 9 AM):
"I built this awesome feature over the weekend!"
Runs: python manage.py runserver
 Works perfectly!

QA Engineer (Monday 2 PM):
"The feature is broken, can't even start the app!"
Runs: python manage.py runserver
 Error: Module 'numpy' not found

Operations (Tuesday 10 AM):
"Production is down! Your feature crashed everything!"
Runs: python manage.py runserver
 Error: Python version mismatch

Developer:
"But... it works on my machine!"
```

Why It Happened:

```
Developer's Laptop:
- Ubuntu 22.04
- Python 3.10
- PostgreSQL 14
- numpy 1.24.0

QA Server:
- Ubuntu 20.04
- Python 3.8
- PostgreSQL 12
- numpy not installed

Production Server:
- Ubuntu 18.04
- Python 3.6
- PostgreSQL 10
- numpy 1.19.0
```

Problem 3: Dependency Conflicts **Scenario**:

```
# Company runs multiple projects on one server

# Project A requirements:
Django==3.2
PostgreSQL==12

# Project B requirements:
Django==4.2
PostgreSQL==15

# Try to install both:
pip install Django==3.2  # For Project A
pip install Django==4.2  # For Project B

# Result: Only one version can exist!
# Project A breaks when Project B is deployed
```

Problem 4: Scaling Nightmare **Traditional Scaling**:

```
Step 1: Buy new server ($$$)
Step 2: Wait for delivery (days/weeks)
Step 3: Install OS (2 hours)
Step 4: Install dependencies (3 hours)
Step 5: Configure services (2 hours)
Step 6: Deploy application (1 hour)
Step 7: Test everything (2 hours)
Step 8: Debug issues (??? hours)

Total Time: Days or weeks
Total Cost: Thousands of dollars
Success Rate: 60%
```

Problem 5: Server Snowflakes Each server becomes unique over time:

```
Server 1:
- Deployed in 2018
- Manually patched 47 times
- Configuration files modified by 5 different admins
- Nobody knows what's different anymore

Server 2:
- Deployed in 2019
- Slightly different setup
- Different versions of libraries
- Works differently than Server 1
```

```
Result: "Snowflake servers" - each one is unique and fragile
```

Enter Virtualization (First Solution Attempt)

Virtual Machines to the Rescue... Sort of:

```
Physical Server
   Hypervisor (VMware, VirtualBox)
        VM1 (full OS + App A) - 8GB RAM
        VM2 (full OS + App B) - 8GB RAM
        VM3 (full OS + App C) - 8GB RAM

Problems:
  Each VM needs full OS (GBs of disk space)
  Slow to start (minutes)
  Resource intensive
  Still requires manual configuration per VM
  But at least isolated!
```

Why Docker Changed Everything

With Docker (2013 onwards):

```
Physical Server
   Docker Engine
        Container A (App only) - 200MB
        Container B (App only) - 150MB
        Container C (App only) - 180MB

Benefits:
  Lightweight (MBs not GBs)
  Fast to start (seconds)
  Efficient resource usage
  Consistent across all environments
  Easy to scale
  Infrastructure as Code
```

Docker Workflow:

```bash
# Day 1: Developer creates Dockerfile
cat > Dockerfile << EOF
FROM python:3.10
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "manage.py", "runserver"]
```

```bash
EOF

# Day 2: Build image
docker build -t myapp:1.0 .

# Day 3: Run anywhere
docker run -p 8000:8000 myapp:1.0

# Day 4: Scale to 100 servers
for i in {1..100}; do
  docker run -d -p 800$i:8000 myapp:1.0
done

# It just works! Everywhere! Every time!
```

History of Docker

Timeline of Containerization

1979 - Unix V7

```
chroot command introduced
- Change root directory for a process
- First step toward isolation
```

2000 - FreeBSD Jails

```
More advanced isolation
- Separate filesystem
- Network stack
- Process space
```

2001 - Linux VServer

```
Operating system-level virtualization
- Multiple Linux instances on one server
```

2004 - Solaris Containers

```
Sun Microsystems introduces zones
- Similar to modern containers
```

2005 - OpenVZ

```
Operating system-level virtualization for Linux
- Patched Linux kernel
```

2006 - Process Containers (cgroups)

```
Google introduces cgroups (Control Groups)
- Limit and isolate resource usage
- Foundation for modern containers
```

2008 - LXC (Linux Containers)

```
First complete Linux container manager
- Uses cgroups and namespaces
- Template-based
```

2013 - Docker is Born!

```
Solomon Hykes demos Docker at PyCon
- Makes containers easy to use
- Introduces Docker Hub
- Standardizes container format

March 2013: Docker 0.1 released
October 2013: Docker gets $15M funding
```

2014 - The Docker Revolution

```
- Docker 1.0 released (production-ready)
- Microsoft announces Windows Server containers
- Google announces Kubernetes
- AWS announces EC2 Container Service
```

2015 - Open Container Initiative (OCI)

```
- Docker donates container format to OCI
- Standardizes container specification
- Industry-wide adoption
```

2016-2020 - Container Ecosystem Explodes

```
- Docker Swarm for orchestration
- Kubernetes becomes dominant orchestrator
- Docker Desktop for Mac/Windows
- Enterprise adoption soars
```

2021-Present - Cloud Native Era

```
- Containers everywhere
- Serverless containers
- Edge computing with containers
- Docker becomes industry standard
```

Docker's Impact in Numbers

2013 (Launch Year): - Few thousand users - Small community

2015: - 1 billion+ container pulls - 100,000+ Dockerized applications

2020: - 13+ million developers - 5.6 million repositories on Docker Hub

2025: - Industry standard for containerization - Used by 90%+ of Fortune 500 companies

---

What is DevOps?

Definition

DevOps is a culture, methodology, and set of practices that combines software **Dev**elopment and IT **Op**eration**s** to shorten the development lifecycle and deliver high-quality software continuously.

Before DevOps (The Waterfall Era)

Traditional Approach:

```
Development Team         vs.        Operations Team
     ↓                                    ↓
Write code                          Maintain servers
     ↓                                    ↓
Throw over the wall      →         Deploy (and pray!)
     ↓                                    ↓
"It works on my machine!"          "Your code broke production!"
     ↓                                    ↓


Timeline: Months or years per release
Quality: Lots of bugs in production
Communication: Hostile
Blame game: Epic
```

The Wall of Confusion:

```
Developers say:           |    Operations say:
"Deploy faster!"          |    "Deploy safely!"
"I need more features!"   |    "I need stability!"
"It worked in dev!"       |    "You didn't test properly!"
"Not my problem!"         |    "Fix your code!"
```

The DevOps Philosophy

Core Principles:

1. Culture - Break Down Silos

```
Before:
Dev Team → Operations Team (enemies)

After:
DevOps Team → Shared responsibility
- Developers on-call for their code
- Operations involved in development
- Shared goals and metrics
```

2. Automation - Automate Everything

```bash
# Manual deployment (old way)
ssh server1.com
cd /var/www/app
git pull
npm install
systemctl restart app
# Repeat for 50 servers

# Automated deployment (DevOps way)
git push origin main
# CI/CD pipeline automatically:
# - Tests code
# - Builds containers
# - Deploys to all servers
# - Monitors health
# - Rolls back if issues
# Done in minutes!
```

3. Measurement - Monitor Everything

```
Track:
- Deployment frequency
- Lead time for changes
- Mean time to recovery
- Change failure rate
- Application performance
- User experience
```

4. Sharing - Knowledge and Tools

```
Share:
- Code repositories (Git)
- Documentation (Wiki)
- Runbooks
```

```
- Monitoring dashboards
- On-call schedules
```

DevOps Lifecycle

```
         CONTINUOUS CYCLE


1. PLAN
     Define features
     Create user stories
     Sprint planning

2. CODE
     Write code
     Version control (Git)
     Code review

3. BUILD
     Compile code
     Run unit tests
     Create artifacts

4. TEST
     Integration tests
     Security tests
     Performance tests

5. RELEASE
     Staging deployment
     User acceptance testing
     Approval gates

6. DEPLOY
     Production deployment
     Blue-green deployment
     Canary releases

7. OPERATE
     Monitor applications
     Manage infrastructure
     Handle incidents

8. MONITOR
     Collect metrics
```

```
    Analyze logs
    User feedback

  → Back to PLAN (Continuous Improvement)
```

DevOps Tools Ecosystem

Version Control

```bash
# Git for source code
git init
git add .
git commit -m "New feature"
git push origin main
```

Continuous Integration/Continuous Deployment (CI/CD)

```yaml
# Jenkins, GitLab CI, GitHub Actions
name: CI/CD Pipeline
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Build
        run: docker build -t myapp .
      - name: Test
        run: docker run myapp npm test
      - name: Deploy
        run: kubectl apply -f deployment.yaml
```

Containerization

```bash
# Docker
docker build -t myapp .
docker run -d -p 80:80 myapp
```

Orchestration

```bash
# Kubernetes
kubectl create deployment myapp --image=myapp:latest
kubectl scale deployment myapp --replicas=10
```

Infrastructure as Code

```
# Terraform
resource "aws_instance" "web" {
```

```
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

Monitoring & Logging

```
# Prometheus, Grafana, ELK Stack
# Monitor CPU, memory, requests, errors
# Alert on anomalies
```

Docker's Role in DevOps

Docker Solves Multiple DevOps Challenges:

1. Environment Consistency

```
# Define environment once
FROM node:18
# Works everywhere: dev, test, prod
```

2. Fast Deployment

```
# Old way: Hours or days
# Docker way: Seconds
docker run -d myapp:latest
```

3. Easy Scaling

```
# Scale from 1 to 100 instances
docker-compose up --scale web=100
```

4. Microservices Architecture

```
# docker-compose.yml
services:
  frontend:
    image: frontend:latest
  backend:
    image: backend:latest
  database:
    image: postgres:14
  cache:
    image: redis:7
```

5. Infrastructure as Code

```
# Dockerfile IS your infrastructure
FROM python:3.10
```

```
RUN apt-get update
RUN pip install django
# Version controlled, reproducible
```

DevOps Metrics (DORA Metrics)

Elite Performers:

```
Deployment Frequency:     Multiple times per day
Lead Time for Changes:    Less than 1 hour
Time to Restore Service:  Less than 1 hour
Change Failure Rate:      0-15%
```

Low Performers:

```
Deployment Frequency:     Once per month or less
Lead Time for Changes:    1-6 months
Time to Restore Service:  1 week to 1 month
Change Failure Rate:      46-60%
```

Docker Helps You Become Elite:

```
# Deploy in seconds
docker push myapp:v2.1.5
docker pull myapp:v2.1.5
docker run myapp:v2.1.5

# Rollback in seconds if needed
docker run myapp:v2.1.4
```

Real-World DevOps Success Story

Netflix Example:

```
Before DevOps:
- Quarterly releases
- Weeks of testing
- Frequent outages
- Slow recovery

After DevOps + Docker + Microservices:
- Thousands of deployments per day
- Minutes of testing per service
- Minimal downtime
- Auto-recovery
- Serves 200+ million subscribers
```

Key Takeaways

1. **Docker** is a containerization platform that packages apps with their dependencies
2. **Containers** are lightweight, isolated, and portable environments
3. **DevOps** is a culture that brings development and operations together
4. **Docker + DevOps** = Fast, reliable, scalable deployments
5. **History** shows containers evolved over decades, Docker made them accessible

---

What's Next?

In the next sections, we'll cover: - Docker Architecture - Installing Docker - Working with Containers - Building Docker Images - And much more!

Remember: Docker is not just a tool, it's a game-changer for modern software development!