

# Building, Managing, and Running Containers

Comprehensive Guide to Docker and  
Containerization

*Prepared By: Rashi Rana  
Corporate Trainer*

# What are Containers?

**Containers** are lightweight, portable, and self-sufficient units that package an application and all its dependencies, libraries, and configuration files needed to run the application consistently across different computing environments.

## Container Characteristics

- **Lightweight:** Share the host OS kernel, consuming fewer resources than virtual machines
- **Portable:** Run consistently across development, testing, and production environments
- **Isolated:** Applications run in separate namespaces with their own file systems
- **Scalable:** Can be quickly started, stopped, and replicated
- **Immutable:** Container images are read-only templates

## Containers vs Virtual Machines

Aspect	Containers	Virtual Machines
Resource Usage	Lightweight, share host OS	Heavy, full OS per VM
Startup Time	Seconds	Minutes
Isolation	Process-level isolation	Hardware-level isolation

Aspect	Containers	Virtual Machines
Portability	Highly portable	Less portable
Performance	Near-native performance	Overhead from hypervisor

## Container Use Cases

### Microservices

Package individual services with their dependencies for independent deployment and scaling.

### CI/CD Pipelines

Consistent build and deployment environments across development lifecycle.

### Application Modernization

Containerize legacy applications for cloud migration and modernization.

### Development Environments

Standardized development environments that work across different machines.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Why Docker in DevOps?

---

## The DevOps Challenge

Traditional software development faces challenges with environment inconsistencies, complex deployments, and scaling difficulties. Docker addresses these challenges by providing a standardized way to package, distribute, and run applications.

## Docker's Role in DevOps

### Consistent Environments

Eliminates "works on my machine" problems by ensuring identical environments across development, testing, and production.

### Faster Deployments

Containers start in seconds, enabling rapid deployment and rollback capabilities.

### Resource Efficiency

Higher density deployments with better resource

### Simplified CI/CD

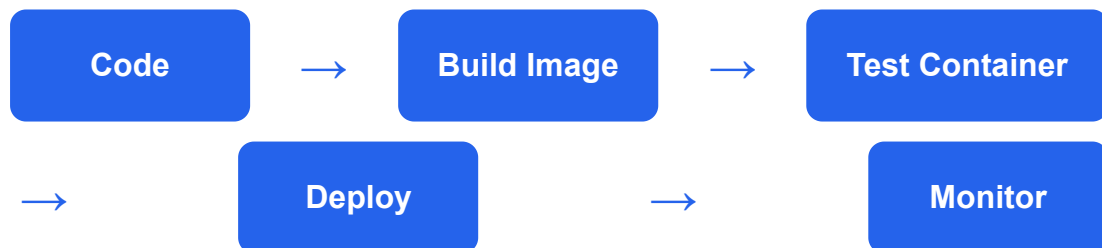
Streamlined build, test, and deployment pipelines with containerized applications.

utilization compared to traditional VMs.

## DevOps Benefits with Docker

- **Infrastructure as Code:** Dockerfile defines infrastructure requirements
- **Version Control:** Container images are versioned and stored in registries
- **Automated Testing:** Consistent test environments across all stages
- **Scalability:** Easy horizontal scaling with container orchestration
- **Rollback Capability:** Quick rollback to previous container versions
- **Multi-Cloud Deployment:** Portable across different cloud providers

## Docker in the DevOps Toolchain



## Real-World Impact

Organizations using Docker report 50% faster deployment times, 60% reduction in infrastructure costs, and 70% improvement in developer productivity through standardized containerized environments.

**Key Insight:** Docker transforms DevOps by making applications portable, scalable, and consistent across all environments,

enabling true continuous delivery.

*Prepared By: Rashi Rana  
Corporate Trainer*

# Installing Docker

---

## System Requirements

Docker requires a 64-bit operating system and supports Linux, Windows, and macOS. For production environments, Linux is the preferred platform.

## Installation on Ubuntu (Recommended for Labs)

```
# Update package index
sudo apt update

# Install prerequisite packages
sudo apt install apt-transport-https ca-certificates
curl software-properties-common

# Add Docker's official GPG key
curl -fsSL https://download.docker.com/linux/ubuntu/gpg
| sudo gpg --dearmor -o /usr/share/keyrings/docker-
archive-keyring.gpg

# Add Docker repository
echo "deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -
cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null

# Update package index again
sudo apt update

# Install Docker CE
```

```
sudo apt install docker-ce docker-ce-cli containerd.io
docker-buildx-plugin docker-compose-plugin
```

## Post-Installation Steps

```
# Start and enable Docker service
sudo systemctl start docker
sudo systemctl enable docker

# Add user to docker group (avoid using sudo)
sudo usermod -aG docker $USER

# Log out and log back in, or run:
newgrp docker

# Verify installation
docker --version
docker run hello-world
```

## Installation on Other Platforms

Platform	Installation Method	Notes
Windows	Docker Desktop for Windows	Requires WSL2 or Hyper-V
macOS	Docker Desktop for Mac	Intel and Apple Silicon support
CentOS/RHEL	yum/dnf package manager	Similar to Ubuntu process
Amazon Linux	yum install docker	Pre-installed on many AMIs



# Verification Commands

```
# Check Docker version
docker --version

# Display system information
docker info

# Test Docker installation
docker run hello-world

# Check running containers
docker ps

# Check all containers
docker ps -a
```

**Security Note:** Adding users to the docker group grants root-level privileges. In production, consider using rootless Docker or proper access controls.

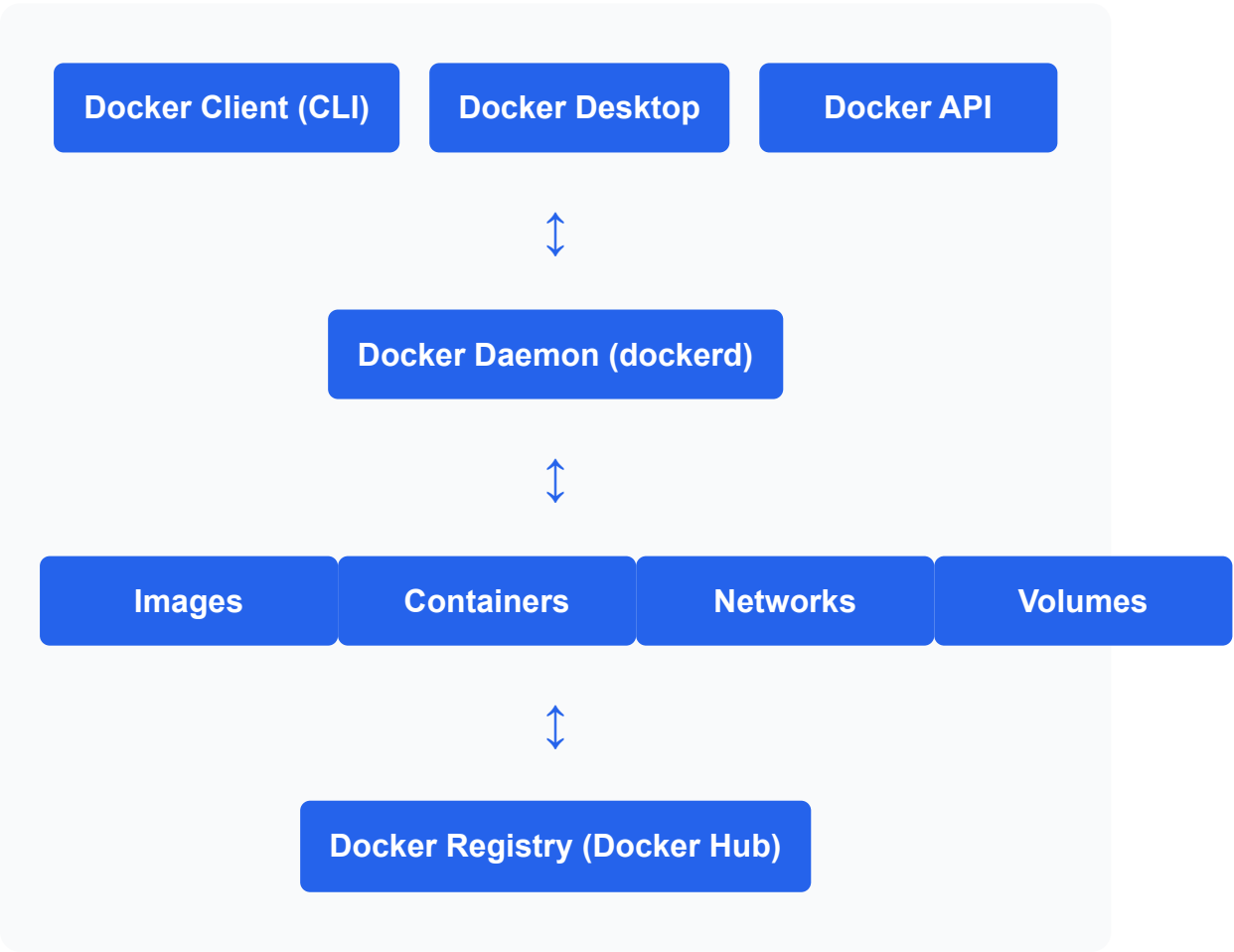
*Prepared By: Rashmi Rana  
Corporate Trainer*

# Docker Architecture: Overview

## Docker Architecture Components

Docker uses a client-server architecture with three main components: **Docker Client**, **Docker Daemon**, and **Docker Registry**. These components work together to build, run, and distribute containers.

## Architecture Diagram



## Component Descriptions

## Docker Client

Command-line interface that users interact with. Sends commands to Docker daemon via REST API.

## Docker Daemon

Background service that manages Docker objects like images, containers, networks, and volumes.

## Docker Images

Read-only templates used to create containers. Built from Dockerfile instructions.

## Docker Registry

Storage and distribution system for Docker images. Docker Hub is the default public registry.

## How Components Interact

1. **Client Command:** User runs `docker run nginx`
2. **API Call:** Client sends REST API request to daemon
3. **Image Check:** Daemon checks if nginx image exists locally
4. **Image Pull:** If not found, daemon pulls from registry
5. **Container Creation:** Daemon creates and starts container from image
6. **Response:** Client receives confirmation and container ID

**Key Concept:** Docker's architecture separates the user interface (client) from the execution engine (daemon), enabling remote Docker management and API-based automation.

*Prepared By: Rashi Rana*  
*Corporate Trainer*

# Docker Engine

---

## What is Docker Engine?

**Docker Engine** is the core runtime that creates and manages containers. It consists of the Docker daemon (dockerd), REST API, and command-line interface (CLI).

## Docker Engine Components

### Docker Daemon (dockerd)

Background process that manages Docker objects and handles API requests from Docker clients.

### REST API

Interface that programs can use to talk to and instruct the Docker daemon.

### Docker CLI

Command-line interface client that uses the REST API to control the Docker daemon.

### containerd

High-level container runtime that manages container lifecycle and image management.

## Engine Architecture Layers

Layer	Component	Responsibility
User Interface	Docker CLI, Docker Desktop	User interaction and commands
API Layer	Docker REST API	Communication protocol
Engine Layer	Docker Daemon (dockerd)	Container management and orchestration
Runtime Layer	containerd, runc	Container execution and lifecycle
OS Layer	Linux Kernel	Namespaces, cgroups, filesystem

## Docker Engine Operations

```
# Check Docker Engine status
sudo systemctl status docker

# Start Docker Engine
sudo systemctl start docker

# Stop Docker Engine
sudo systemctl stop docker

# Restart Docker Engine
sudo systemctl restart docker

# View Docker Engine logs
sudo journalctl -u docker.service

# Configure Docker Engine
sudo nano /etc/docker/daemon.json
```

# Engine Configuration

```
# /etc/docker/daemon.json
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  },
  "storage-driver": "overlay2",
  "insecure-registries": ["myregistry.com:5000"],
  "registry-mirrors": ["https://mirror.gcr.io"]
}
```

**Important:** Always restart the Docker daemon after modifying the configuration file for changes to take effect.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Docker Images

---

## Understanding Docker Images

**Docker Images** are read-only templates that contain a set of instructions for creating containers. They include the application code, runtime, system tools, libraries, and settings needed to run an application.

## Image Characteristics

- **Layered Architecture:** Images are built in layers, each representing a set of file changes
- **Immutable:** Once created, image layers cannot be modified
- **Shareable:** Layers can be shared between different images
- **Versioned:** Images can be tagged with different versions
- **Portable:** Can run on any system with Docker installed

## Image Layers

Application Layer

Dependencies Layer

Runtime Layer



OS Libraries Layer

Base OS Layer

## Image Naming Convention

```
# Full image name format:  
[registry-host[:port]]/[username/]repository[:tag]  
  
# Examples:  
nginx # Official nginx image, latest tag  
nginx:1.21 # nginx version 1.21  
ubuntu:20.04 # Ubuntu 20.04 LTS  
docker.io/library/nginx:latest # Full name for nginx  
myregistry.com:5000/myapp:v1.0 # Private registry image  
username/myapp:dev # User's repository
```

## Image Storage

### Local Storage

Images stored locally in  
/var/lib/docker/ directory  
with efficient layer sharing.

### Registry Storage

Images stored in registries  
like Docker Hub, AWS ECR,  
or private registries.

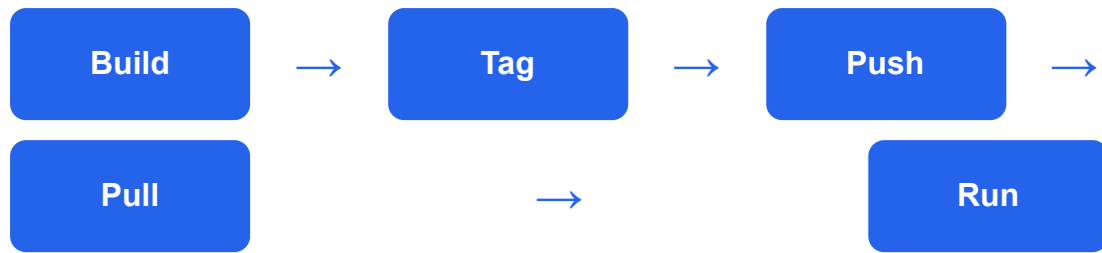
### Layer Caching

Docker caches layers to  
speed up builds and reduce  
storage usage.

### Content Addressing

Images identified by  
cryptographic hashes  
ensuring integrity.

# Image Lifecycle



## Base Images

Most Docker images are built on top of base images that provide the foundation operating system and basic tools. Popular base images include:

- **Alpine Linux:** Minimal, security-focused (5MB)
- **Ubuntu:** Full-featured, widely used (72MB)
- **Debian:** Stable, well-maintained (124MB)
- **CentOS:** Enterprise-focused (200MB)
- **Scratch:** Empty image for static binaries

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Docker Containers

---

## Understanding Containers

**Docker Containers** are running instances of Docker images. They are isolated, lightweight, and include everything needed to run an application: code, runtime, system tools, libraries, and settings.

## Container vs Image Relationship

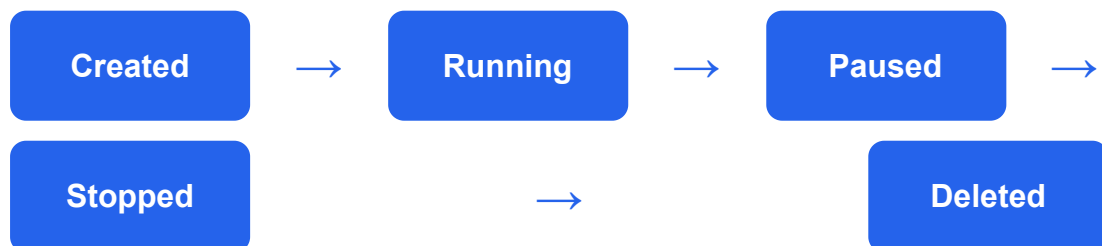
### Docker Image

- Read-only template
- Static blueprint
- Stored in registry
- Immutable layers

### Docker Container

- Running instance
- Dynamic process
- Has writable layer
- Can be started/stopped

## Container Lifecycle States



## Container Isolation

Isolation Type	Technology	Purpose
Process Isolation	PID Namespaces	Separate process trees
Network Isolation	Network Namespaces	Separate network stacks
Filesystem Isolation	Mount Namespaces	Separate filesystem views
User Isolation	User Namespaces	Separate user/group IDs
Resource Limits	Control Groups (cgroups)	CPU, memory, I/O limits

## Container Runtime

Containers run as processes on the host system but with isolation provided by Linux kernel features:

- **Namespaces:** Provide isolation for system resources
- **Control Groups (cgroups):** Limit and monitor resource usage
- **Union Filesystems:** Layer multiple filesystems together
- **Security:** AppArmor, SELinux, seccomp profiles

## Container Advantages

Fast Startup	Resource Efficient
--------------	--------------------

Containers start in seconds,  
not minutes like VMs

Share host OS kernel,  
minimal overhead

### **Consistent Environment**

Same runtime environment  
everywhere

### **Easy Scaling**

Quickly create multiple  
instances

## **Container Networking**

Containers can communicate through various networking modes:

- **Bridge Network:** Default isolated network
- **Host Network:** Share host's network stack
- **None Network:** No network access
- **Custom Networks:** User-defined networks

**Key Insight:** Containers provide process-level isolation while sharing the host OS kernel, making them much more efficient than traditional virtual machines.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Basic Docker Commands

---

## Essential Docker Commands

Docker provides a comprehensive set of commands to manage containers, images, networks, and volumes. Understanding these basic commands is essential for working with Docker effectively.

## Command Categories

### Container Management

`docker run`, `docker ps`,  
`docker stop`, `docker start`, `docker rm`

### Image Management

`docker images`, `docker pull`, `docker push`,  
`docker build`, `docker rmi`

### Container Interaction

`docker exec`, `docker logs`, `docker cp`,  
`docker attach`

### System Management

`docker info`, `docker version`, `docker system`, `docker stats`

## Command Structure

```
# Basic Docker command structure
docker [OPTIONS] COMMAND [ARG...]

# Examples:
docker --version # Show Docker version
docker info # Display system information
docker help # Show help information
docker COMMAND --help # Show help for specific command
```

## Common Options

Option	Description	Example
-d, --detach	Run container in background	<code>docker run -d nginx</code>
-it	Interactive terminal	<code>docker run -it ubuntu bash</code>
-p, --publish	Publish container port	<code>docker run -p 80:80 nginx</code>
-v, --volume	Mount volume	<code>docker run -v /data:/app/data nginx</code>
--name	Assign container name	<code>docker run --name web nginx</code>

## Getting Help

```
# General help
docker --help

# Command-specific help
docker run --help
docker build --help
docker ps --help
```

```
# List all commands
docker command ls

# Docker documentation
# Visit: https://docs.docker.com/
```

**Pro Tip:** Use `docker COMMAND --help` to quickly access documentation for any Docker command without leaving the terminal.

*Prepared By: Rashmi Rana  
Corporate Trainer*



# docker run Command

## Understanding docker run

**docker run** creates and starts a new container from a Docker image. It's the most commonly used Docker command and combines image pulling, container creation, and container starting into one command.

## Basic Syntax

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

# Simple examples:

```
docker run hello-world # Run hello-world container
docker run ubuntu # Run Ubuntu container
docker run nginx # Run Nginx web server
docker run -d nginx # Run Nginx in background
```

## Common docker run Options

Option	Description	Example
-d, --detach	Run in background (daemon mode)	docker run -d nginx
-it	Interactive mode with TTY	docker run -it ubuntu bash

Option	Description	Example
-p, --publish	Port mapping host:container	docker run -p 8080:80 nginx
--name	Assign container name	docker run --name web nginx
-v, --volume	Mount volume or bind mount	docker run -v /data:/var/data nginx
-e, --env	Set environment variables	docker run -e NODE_ENV=production node
--rm	Remove container when it exits	docker run --rm ubuntu echo "hello"

## Practical Examples

```
# Run Nginx web server on port 8080
docker run -d -p 8080:80 --name my-nginx nginx

# Run interactive Ubuntu container
docker run -it --name my-ubuntu ubuntu:20.04 /bin/bash

# Run container with environment variables
docker run -d -e MYSQL_ROOT_PASSWORD=secret mysql:8.0

# Run container with volume mount
docker run -d -p 80:80 -v
/host/data:/usr/share/nginx/html nginx

# Run temporary container (auto-remove)
docker run --rm -it python:3.9 python

# Run container with custom command
docker run ubuntu echo "Hello from Ubuntu container"
```

# Port Mapping Examples

```
# Map single port
docker run -p 8080:80 nginx # Host:8080 -> Container:80

# Map multiple ports
docker run -p 8080:80 -p 8443:443 nginx # Multiple port
mappings

# Map to specific interface
docker run -p 127.0.0.1:8080:80 nginx # Bind to localhost
only

# Map random host port
docker run -P nginx # Docker assigns random ports

# UDP port mapping
docker run -p 53:53/udp dns-server # Map UDP port
```

## What Happens When You Run docker run

- 1 Image Check:** Docker checks if the image exists locally
- 2 Image Pull:** If not found locally, pulls from registry
- 3 Container Creation:** Creates a new container from the image
- 4 Network Setup:** Configures networking for the container
- 5 Container Start:** Starts the container and runs the specified command

*Prepared By: Rashmi Rana  
Corporate Trainer*

# docker ps Command

---

## Listing Containers

**docker ps** lists containers on your system. By default, it shows only running containers, but with options, you can view all containers, filter results, and customize the output format.

## Basic Usage

```
# List running containers
docker ps

# List all containers (running and stopped)
docker ps -a

# List only container IDs
docker ps -q

# List all container IDs
docker ps -aq
```

## docker ps Options

Option	Description	Example
-a, --all	Show all containers (running and stopped)	docker ps -a

Option	Description	Example
<code>-q, --quiet</code>	Only show container IDs	<code>docker ps -q</code>
<code>-l, --latest</code>	Show the latest created container	<code>docker ps -l</code>
<code>-n</code>	Show last n created containers	<code>docker ps -n 5</code>
<code>-s, --size</code>	Display total file sizes	<code>docker ps -s</code>
<code>--format</code>	Pretty-print using Go template	<code>docker ps --format "table {{.Names}}\t{{.Status}}"</code>

## Understanding docker ps Output

```
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES
alb2c3d4e5f6  nginx  "/docker-entrypoint..."  2 minutes ago  Up
2 minutes  0.0.0.0:8080->80/tcp  my-nginx
b2c3d4e5f6g7  ubuntu  "/bin/bash"  5 minutes ago  Exited (0) -
my-ubuntu
```

## Column Explanations

- **CONTAINER ID:** Unique identifier for the container (short form)
- **IMAGE:** Docker image used to create the container
- **COMMAND:** Command that was run when container started
- **CREATED:** When the container was created
- **STATUS:** Current status (Up, Exited, Restarting, etc.)
- **PORTS:** Port mappings between host and container

- **NAMES:** Container name (auto-generated or user-assigned)

## Filtering Containers

```
# Filter by status
docker ps --filter "status=running"
docker ps --filter "status=exited"

# Filter by name
docker ps --filter "name=nginx"

# Filter by image
docker ps --filter "ancestor=ubuntu"

# Filter by label
docker ps --filter "label=environment=production"

# Multiple filters
docker ps --filter "status=running" --filter "name=web"
```

## Custom Output Formatting

```
# Custom table format
docker ps --format "table
{{.Names}}\t{{.Image}}\t{{.Status}}"

# JSON format
docker ps --format "{{json .}}"

# Simple format
docker ps --format "{{.Names}}: {{.Status}}"

# Show only names and ports
docker ps --format "table {{.Names}}\t{{.Ports}}"
```

## Practical Examples

```
# Count running containers
docker ps -q | wc -l

# Get container IDs for cleanup
docker ps -aq --filter "status=exited"

# Monitor container status
watch docker ps

# Show containers with their sizes
docker ps -as
```

*Prepared By: Rashi Rana*  
*Corporate Trainer*

# docker stop and docker rm

## Commands

---

### Stopping Containers

**docker stop** gracefully stops running containers by sending a SIGTERM signal, followed by SIGKILL if the container doesn't stop within the timeout period (default 10 seconds).

### docker stop Command

```
# Stop a container by name
docker stop my-nginx

# Stop a container by ID
docker stop a1b2c3d4e5f6

# Stop multiple containers
docker stop container1 container2 container3

# Stop all running containers
docker stop $(docker ps -q)

# Stop with custom timeout (in seconds)
docker stop -t 30 my-nginx
```

### docker kill vs docker stop



Command	Signal	Behavior	Use Case
<code>docker stop</code>	SIGTERM then SIGKILL	Graceful shutdown	Normal container shutdown
<code>docker kill</code>	SIGKILL (default)	Immediate termination	Force stop unresponsive containers

## Removing Containers

**docker rm** removes one or more containers. Containers must be stopped before they can be removed, unless you use the force option.

### docker rm Command

```
# Remove a stopped container
docker rm my-nginx

# Remove multiple containers
docker rm container1 container2

# Force remove a running container
docker rm -f my-nginx

# Remove all stopped containers
docker rm $(docker ps -aq --filter "status=exited")

# Remove container and its volumes
docker rm -v my-container
```

### docker rm Options

Option	Description	Example
<code>-f, --force</code>	Force remove running container	<code>docker rm -f my-container</code>
<code>-v, --volumes</code>	Remove associated volumes	<code>docker rm -v my-container</code>
<code>-l, --link</code>	Remove specified link	<code>docker rm -l my-link</code>

## Container Lifecycle Management



## Cleanup Commands

```
# Stop and remove all containers
docker stop $(docker ps -aq) && docker rm $(docker ps -aq)

# Remove all stopped containers
docker container prune

# Remove all containers (force)
docker rm -f $(docker ps -aq)

# Clean up everything (containers, networks, images)
docker system prune

# Clean up with volumes
docker system prune --volumes
```

## Best Practices

- **Graceful Shutdown:** Use `docker stop` instead of `docker kill` when possible
- **Auto-removal:** Use `--rm` flag with `docker run` for temporary containers
- **Regular Cleanup:** Periodically remove stopped containers to free up space
- **Container Names:** Use meaningful names for easier management

**Warning:** `docker rm -f` forcefully terminates containers without graceful shutdown. Use with caution in production environments.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# docker exec Command

---

## Executing Commands in Running Containers

**docker exec** runs commands inside running containers. This is essential for debugging, maintenance, and interactive work with containerized applications.

## Basic Syntax

```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

```
# Basic examples:
```

```
docker exec my-container ls /app
```

```
docker exec my-container cat /etc/hostname
```

```
docker exec my-container ps aux
```

## Interactive Mode

```
# Start interactive bash session  
docker exec -it my-container bash
```

```
# Start interactive shell (sh for Alpine)  
docker exec -it my-container sh
```

```
# Run interactive Python  
docker exec -it my-python-app python
```

```
# Access MySQL command line  
docker exec -it my-mysql mysql -u root -p
```

# docker exec Options

Option	Description	Example
<code>-i, --interactive</code>	Keep STDIN open	<code>docker exec -i container cat &gt; file.txt</code>
<code>-t, --tty</code>	Allocate pseudo-TTY	<code>docker exec -t container ls</code>
<code>-d, --detach</code>	Run in background	<code>docker exec -d container service nginx start</code>
<code>-u, --user</code>	Run as specific user	<code>docker exec -u root container bash</code>
<code>-w, --workdir</code>	Set working directory	<code>docker exec -w /app container ls</code>
<code>-e, --env</code>	Set environment variables	<code>docker exec -e VAR=value container env</code>

## Common Use Cases

### Debugging

Access container filesystem, check logs, inspect processes

### Maintenance

Update configurations, restart services, clean up files

### Database Access

### File Operations

Connect to database CLI,  
run queries, backup data

Edit files, check  
permissions, transfer data

## Practical Examples

```
# Check container processes
docker exec my-container ps aux

# View container logs
docker exec my-container tail -f /var/log/app.log

# Check disk usage
docker exec my-container df -h

# Install package (temporary)
docker exec -it my-container apt update && apt install -y vim

# Check network configuration
docker exec my-container ip addr show

# Run as different user
docker exec -u www-data my-web-container whoami
```

## Database Examples

```
# MySQL database access
docker exec -it mysql-container mysql -u root -p

# PostgreSQL database access
docker exec -it postgres-container psql -U postgres

# MongoDB access
docker exec -it mongo-container mongo
```

```
# Redis CLI
docker exec -it redis-container redis-cli
```

## File Operations

```
# Edit configuration file
docker exec -it my-container vi /etc/nginx/nginx.conf

# Create directory
docker exec my-container mkdir -p /app/logs

# Change file permissions
docker exec my-container chmod 755 /app/script.sh

# Check file contents
docker exec my-container cat /app/config.json
```

## docker exec vs docker run

Aspect	docker exec	docker run
Target	Existing running container	Creates new container
Use Case	Debug, maintain, interact	Start new application
Process	Additional process in container	Main container process

**Pro Tip:** Use `docker exec -it container bash` to get a shell inside a running container for interactive debugging and exploration.

# Docker Image Commands

---

## Image Management Commands

Docker provides comprehensive commands for managing images: building custom images, listing available images, removing unused images, and transferring images to/from registries.

## Core Image Commands

### **docker build**

Build images from  
Dockerfile

### **docker images**

List local images

### **docker pull**

Download images from  
registry

### **docker push**

Upload images to registry

### **docker rmi**

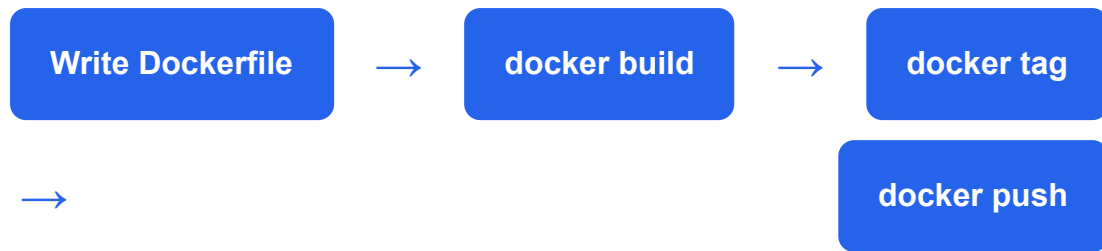
Remove local images

### **docker tag**

Tag images with names



## Image Workflow



## Image Lifecycle

- 1 **Create Dockerfile:** Define image build instructions
- 2 **Build Image:** Use `docker build` to create image
- 3 **Tag Image:** Assign meaningful names and versions
- 4 **Test Image:** Run containers to verify functionality
- 5 **Push to Registry:** Share image via Docker Hub or private registry
- 6 **Pull and Deploy:** Download and run on target systems

## Image Storage Locations

- **Local Storage:** `/var/lib/docker/` (Linux) or Docker Desktop storage
- **Docker Hub:** Public registry with millions of images
- **Private Registries:** AWS ECR, Azure ACR, Google GCR
- **Self-hosted:** Docker Registry, Harbor, Nexus

**Best Practice:** Always tag your images with meaningful names and version numbers for better organization and deployment tracking.

*Prepared By: Rashi Rana*  
*Corporate Trainer*

# docker build Command

---

## Building Custom Images

**docker build** creates Docker images from a Dockerfile and build context. It reads instructions from the Dockerfile and executes them step by step to create a new image.

## Basic Syntax

```
docker build [OPTIONS] PATH | URL | -  
  
# Basic examples:  
docker build . # Build from current directory  
docker build -t myapp:latest . # Build with tag  
docker build -f Dockerfile.prod . # Use specific Dockerfile  
docker build https://github.com/user/repo.git # Build from  
Git repo
```

## Sample Dockerfile

```
# Dockerfile  
FROM node:16-alpine  
  
# Set working directory  
WORKDIR /app  
  
# Copy package files  
COPY package*.json ./  
  
# Install dependencies  
RUN npm install
```

```
# Copy application code
COPY . .

# Expose port
EXPOSE 3000

# Define startup command
CMD ["npm", "start"]
```

## docker build Options

Option	Description	Example
-t, --tag	Name and tag for the image	docker build -t myapp:v1.0 .
-f, --file	Specify Dockerfile path	docker build -f Dockerfile.prod .
--no-cache	Don't use cache when building	docker build --no-cache .
--build-arg	Set build-time variables	docker build --build-arg VERSION=1.0 .
--target	Build specific stage in multi-stage	docker build --target production .
--platform	Set platform for build	docker build --platform linux/amd64 .

## Build Process

- 1 Context Creation:** Docker creates build context from specified path

- 2 **Dockerfile Parsing:** Docker reads and validates Dockerfile instructions
- 3 **Layer Building:** Each instruction creates a new layer
- 4 **Cache Utilization:** Docker reuses cached layers when possible
- 5 **Image Creation:** Final image is created with all layers

## Build Examples

```
# Simple build with tag
docker build -t webapp:latest .

# Build with build arguments
docker build --build-arg NODE_ENV=production -t
webapp:prod .

# Build without cache
docker build --no-cache -t webapp:fresh .

# Build from specific Dockerfile
docker build -f docker/Dockerfile.dev -t webapp:dev .

# Multi-stage build targeting specific stage
docker build --target builder -t webapp:builder .

# Build from Git repository
docker build -t webapp
https://github.com/username/webapp.git
```

## Multi-stage Build Example

```
# Multi-stage Dockerfile
FROM node:16 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
```

```
RUN npm run build
```

```
FROM nginx:alpine AS production
```

```
COPY --from=builder /app/dist /usr/share/nginx/html
```

```
EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```

## Build Optimization Tips

- **Layer Caching:** Order instructions from least to most frequently changing
- **.dockerignore:** Exclude unnecessary files from build context
- **Multi-stage Builds:** Reduce final image size by excluding build tools
- **Minimal Base Images:** Use Alpine or distroless images when possible

**Security Tip:** Never include secrets like passwords or API keys in Dockerfile. Use build arguments or runtime environment variables instead.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Image Management Commands

---

## docker images - Listing Images

```
# List all local images
docker images

# List images with specific format
docker images --format "table
{{.Repository}}\t{{.Tag}}\t{{.Size}}"

# List only image IDs
docker images -q

# Show all images including intermediate
docker images -a

# Filter images
docker images --filter "dangling=true"
docker images --filter "reference=nginx:"
```

## docker pull - Downloading Images

**docker pull** downloads images from a registry to your local machine. Docker automatically pulls images when running containers if they don't exist locally.

```
# Pull latest version
docker pull nginx

# Pull specific version
docker pull nginx:1.21
```

```
# Pull from specific registry
docker pull myregistry.com:5000/myapp:latest

# Pull all tags of an image
docker pull -a nginx

# Pull with platform specification
docker pull --platform linux/amd64 nginx
```

## docker push - Uploading Images

**docker push** uploads images from your local machine to a registry. You must be authenticated and have push permissions to the target repository.

```
# Push to Docker Hub
docker push username/myapp:latest

# Push specific tag
docker push username/myapp:v1.0

# Push to private registry
docker push myregistry.com:5000/myapp:latest

# Push all tags
docker push -a username/myapp
```

## docker rmi - Removing Images

```
# Remove single image
docker rmi nginx:latest

# Remove multiple images
docker rmi nginx:1.20 nginx:1.21
```



```
# Force remove image
docker rmi -f myapp:latest

# Remove all unused images
docker image prune

# Remove all images
docker rmi $(docker images -q)

# Remove dangling images
docker image prune -f
```

## docker tag - Tagging Images

```
# Tag existing image
docker tag myapp:latest myapp:v1.0

# Tag for Docker Hub
docker tag myapp:latest username/myapp:latest

# Tag for private registry
docker tag myapp:latest myregistry.com:5000/myapp:latest

# Multiple tags
docker tag myapp:latest myapp:stable
docker tag myapp:latest myapp:production
```

## Image Information Commands

Command	Purpose	Example
docker inspect	Detailed image information	docker inspect nginx:latest
docker history	Show image layer history	docker history nginx:latest

Command	Purpose	Example
<code>docker image ls</code>	Alternative to docker images	<code>docker image ls</code>
<code>docker image prune</code>	Remove unused images	<code>docker image prune -a</code>

## Registry Authentication

```
# Login to Docker Hub
docker login

# Login to private registry
docker login myregistry.com:5000

# Login with credentials
docker login -u username -p password

# Logout
docker logout

# View stored credentials
cat ~/.docker/config.json
```

**Workflow Tip:** Always test your images locally before pushing to a registry. Use meaningful tags and maintain a consistent tagging strategy.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Why We Need Containers

---

## The Traditional Deployment Problem

Before containers, applications faced numerous challenges in deployment and scaling. The "it works on my machine" problem was common, along with dependency conflicts, environment inconsistencies, and resource inefficiencies.

## Traditional Challenges

### Environment Inconsistency

Different OS versions, libraries, and configurations between development, testing, and production environments.

### Dependency Hell

Conflicting library versions, missing dependencies, and complex installation procedures.

### Resource Waste

Virtual machines consuming excessive resources with full OS

### Slow Deployment

Time-consuming setup processes, manual configuration, and lengthy deployment cycles.

overhead for each application.

## How Containers Solve These Problems

Problem	Traditional Solution	Container Solution
<b>Environment Differences</b>	Manual configuration management	Identical container images everywhere
<b>Dependency Conflicts</b>	Complex dependency management	Isolated container environments
<b>Resource Inefficiency</b>	One VM per application	Multiple containers per host
<b>Slow Scaling</b>	Provision new VMs	Start containers in seconds
<b>Complex Deployment</b>	Manual installation steps	Single container deployment

## Container Benefits in Detail

- **Portability:** Run anywhere Docker is installed - laptop, server, cloud
- **Consistency:** Same environment from development to production
- **Efficiency:** Share host OS kernel, minimal resource overhead
- **Scalability:** Rapid horizontal scaling with orchestration tools
- **Isolation:** Applications don't interfere with each other
- **Version Control:** Image versioning enables easy rollbacks
- **DevOps Integration:** Perfect fit for CI/CD pipelines

## Real-World Impact

### Development Speed

70% faster development cycles with consistent environments and rapid testing.

### Infrastructure Costs

60% reduction in infrastructure costs through better resource utilization.

### Deployment Frequency

10x more frequent deployments with automated container pipelines.

### System Reliability

50% fewer production issues due to environment consistency.

## Container Use Cases

1. **Microservices Architecture:** Independent service deployment and scaling
2. **Cloud Migration:** Lift and shift applications to cloud platforms
3. **CI/CD Pipelines:** Consistent build and test environments
4. **Development Environments:** Standardized developer workspaces
5. **Legacy Application Modernization:** Containerize without rewriting
6. **Multi-Cloud Deployment:** Portable across different cloud providers

**Key Insight:** Containers solve the fundamental problem of "it works on my machine" by packaging applications with all their

dependencies into portable, consistent units.

*Prepared By: Rashi Rana*  
*Corporate Trainer*

# Setting Up Docker Hub Account

---

## What is Docker Hub?

**Docker Hub** is Docker's official cloud-based registry service for sharing container images. It provides public and private repositories, automated builds, and integration with development workflows.

## Docker Hub Features

### Public Repositories

Free unlimited public repositories for open-source projects

### Private Repositories

Secure private repositories for proprietary applications

### Automated Builds

Automatic image builds from GitHub/Bitbucket repositories

### Official Images

Curated, secure images for popular software

## Creating Docker Hub Account

1

**Visit Docker Hub:** Go to <https://hub.docker.com>

- 2 Sign Up:** Click "Sign Up" and fill in your details
- 3 Verify Email:** Check your email and verify your account
- 4 Choose Plan:** Select free or paid plan based on needs
- 5 Complete Profile:** Add profile information and avatar

## Docker Hub Plans

Plan	Price	Public Repos	Private Repos	Features
Free	\$0/month	Unlimited	1	Basic features
Pro	\$5/month	Unlimited	Unlimited	Advanced features, support
Team	\$25/month	Unlimited	Unlimited	Team management, SSO

## Connecting Docker CLI to Docker Hub

```
# Login to Docker Hub
docker login

# Enter your Docker Hub credentials when prompted
Username: your-username
Password: your-password

# Verify login
docker info | grep Username

# Alternative login with credentials
docker login -u your-username -p your-password
```



# Repository Naming Convention

```
# Docker Hub repository format:  
username/repository-name:tag  
  
# Examples:  
rashirana/my-web-app:latest  
rashirana/my-web-app:v1.0  
rashirana/my-web-app:development  
  
# Official images (no username):  
nginx:latest  
ubuntu:20.04  
node:16-alpine
```

## Creating Your First Repository

- 1 **Login to Docker Hub:** Access your Docker Hub dashboard
- 2 **Create Repository:** Click "Create Repository" button
- 3 **Repository Details:** Enter name, description, and visibility
- 4 **Configure Settings:** Set up automated builds if needed
- 5 **Save Repository:** Complete the repository creation

**Security Note:** Never store sensitive information like passwords or API keys in public Docker images. Use environment variables or secrets management instead.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Pulling and Pushing to Docker Hub

---

## Pulling Images from Docker Hub

Docker Hub hosts millions of container images. You can pull official images, community images, or images from specific users to use as base images or run directly.

## Common Pull Operations

```
# Pull official images
docker pull nginx # Latest nginx
docker pull ubuntu:20.04 # Specific Ubuntu version
docker pull node:16-alpine # Node.js on Alpine Linux
docker pull mysql:8.0 # MySQL database

# Pull user images
docker pull username/myapp:latest # User's application
docker pull username/myapp:v1.0 # Specific version

# Pull from organizations
docker pull bitnami/nginx # Bitnami's nginx image
docker pull microsoft/dotnet # Microsoft's .NET image
```

## Exploring Docker Hub

### Official Images

Curated by Docker,  
regularly updated, security-

### Verified Publishers

Images from verified  
software vendors and

scanned images for popular software

organizations

### Community Images

User-contributed images for various applications and use cases

### Automated Builds

Images automatically built from source code repositories

## Pushing Images to Docker Hub

To push images to Docker Hub, you must tag them with your Docker Hub username and ensure you're authenticated. The image will be uploaded to your repository.

## Push Workflow

```
# 1. Build your image
docker build -t my-web-app .

# 2. Tag for Docker Hub
docker tag my-web-app username/my-web-app:latest
docker tag my-web-app username/my-web-app:v1.0

# 3. Login to Docker Hub
docker login

# 4. Push to Docker Hub
docker push username/my-web-app:latest
docker push username/my-web-app:v1.0
```

```
# 5. Verify on Docker Hub website
# Visit: https://hub.docker.com/r/username/my-web-app
```

## Tagging Strategies

Tag Type	Example	Use Case
Latest	myapp:latest	Most recent stable version
Semantic Version	myapp:v1.2.3	Specific release versions
Environment	myapp:dev, myapp:prod	Environment-specific builds
Git Commit	myapp:abc123	Specific code commits
Date-based	myapp:2024-01-15	Time-based releases

## Managing Repository Settings

```
# View repository information
docker search username/myapp

# Pull specific tags
docker pull username/myapp:v1.0
docker pull username/myapp:latest

# List all tags for an image
# Visit Docker Hub web interface or use API
curl -s
https://registry.hub.docker.com/v2/repositories/username/my
```

# Automated Builds

- 1 **Connect Repository:** Link GitHub/Bitbucket repository to Docker Hub
- 2 **Configure Build Rules:** Set up build triggers and tag rules
- 3 **Dockerfile Location:** Specify Dockerfile path in repository
- 4 **Build Settings:** Configure build environment and variables
- 5 **Trigger Builds:** Automatic builds on code commits

## Best Practices

- **Meaningful Tags:** Use descriptive tags that indicate version or purpose
- **Security Scanning:** Enable vulnerability scanning for your images
- **Documentation:** Provide clear README and usage instructions
- **Size Optimization:** Keep images small using multi-stage builds
- **Regular Updates:** Keep base images and dependencies updated

**Pro Tip:** Always test your images locally before pushing to Docker Hub. Use semantic versioning for better version management.

*Prepared By: Rashmi Rana  
Corporate Trainer*

# Lab: Build Docker Image and Run Container

---

## Lab Environment Setup

We'll create a complete hands-on lab using an EC2 Ubuntu instance to build a custom Docker image and run containers. This lab covers the entire Docker workflow from installation to deployment.

## Prerequisites

### AWS Account

Access to AWS console to launch EC2 instances

### SSH Key Pair

EC2 key pair for secure instance access

### Docker Hub Account

Free Docker Hub account for image registry

### Basic Linux Knowledge

Familiarity with Linux commands and text editors

## EC2 Instance Setup

**Launch EC2 Instance**

- 1 AWS Console:** Navigate to EC2 service in AWS console
- 2 Launch Instance:** Click "Launch Instance" button
- 3 Choose AMI:** Select Ubuntu Server 22.04 LTS (Free tier eligible)
- 4 Instance Type:** Choose t2.micro (Free tier eligible)
- 5 Key Pair:** Select existing key pair or create new one
- 6 Security Group:** Allow SSH (port 22) and HTTP (port 80)
- 7 Launch:** Review and launch the instance

**Security Group Configuration**

Type	Protocol	Port Range	Source	Purpose
SSH	TCP	22	Your IP/0.0.0.0/0	Remote access
HTTP	TCP	80	0.0.0.0/0	Web application
Custom TCP	TCP	8080	0.0.0.0/0	Alternative web port
Custom TCP	TCP	3000	0.0.0.0/0	Node.js application

## Connect to EC2 Instance

```
# Connect via SSH (replace with your details)
ssh -i "your-key.pem" ubuntu@ec2-xx-xx-xx-xx.compute-
1.amazonaws.com

# Update system packages
sudo apt update && sudo apt upgrade -y

# Install required packages
sudo apt install -y curl wget git vim
```

## Lab Project Overview

We'll build a simple Node.js web application, containerize it with Docker, and deploy it to our EC2 instance. The application will:

- Display a welcome message with system information
- Show current date and time
- Demonstrate Docker environment variables
- Include health check endpoint

**Lab Goal:** By the end of this lab, you'll have a complete understanding of the Docker workflow: build, tag, push, pull, and run containers.

*Prepared By: Rashmi Rana  
Corporate Trainer*



# Lab: Docker Implementation

---

## Step 1: Install Docker on Ubuntu

```
# Install Docker
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

# Add user to docker group
sudo usermod -aG docker ubuntu

# Logout and login again, or run:
newgrp docker

# Verify installation
docker --version
docker run hello-world
```

## Step 2: Create Application Files

```
# Create project directory
mkdir docker-lab && cd docker-lab

# Create package.json
cat > package.json << 'EOF'
{
  "name": "docker-lab-app",
  "version": "1.0.0",
  "description": "Docker Lab Web Application",
  "main": "app.js",
  "scripts": {
```

```
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.18.0"
  }
}
EOF
```

## Step 3: Create Node.js Application

```
# Create app.js
cat > app.js << 'EOF'
const express = require('express');
const os = require('os');
const app = express();
const PORT = process.env.PORT || 3000;

app.get('/', (req, res) => {
  const response = {
    message: 'Hello from Docker Lab!',
    hostname: os.hostname(),
    platform: os.platform(),
    uptime: os.uptime(),
    timestamp: new Date().toISOString(),
    environment: process.env.NODE_ENV ||
'development'
  };
  res.json(response);
});

app.get('/health', (req, res) => {
  res.status(200).json({ status: 'healthy' });
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
EOF
```

## Step 4: Create Dockerfile

```
# Create Dockerfile
cat > Dockerfile << 'EOF'
# Use official Node.js runtime as base image
FROM node:16-alpine

# Set working directory in container
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install --only=production

# Copy application code
COPY . .

# Create non-root user
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nextjs -u 1001
USER nextjs

# Expose port
EXPOSE 3000

# Health check
HEALTHCHECK --interval=30s --timeout=3s --start-
period=5s --retries=3 \
    CMD curl -f http://localhost:3000/health ||
exit 1

# Start application
CMD ["npm", "start"]
EOF
```



# Lab: Build, Run, and Deploy

---

## Step 5: Build Docker Image

```
# Build the Docker image
docker build -t docker-lab-app:latest .

# Verify image was created
docker images

# Inspect the image
docker inspect docker-lab-app:latest

# Check image history
docker history docker-lab-app:latest
```

## Step 6: Run Container Locally

```
# Run container in background
docker run -d -p 3000:3000 --name my-app docker-lab-app:latest

# Check running containers
docker ps

# Test the application
curl http://localhost:3000
curl http://localhost:3000/health

# View container logs
docker logs my-app
```

```
# Execute commands in container
docker exec -it my-app sh
```

## Step 7: Tag and Push to Docker Hub

```
# Login to Docker Hub
docker login

# Tag image for Docker Hub (replace 'username'
with your Docker Hub username)
docker tag docker-lab-app:latest username/docker-
lab-app:latest
docker tag docker-lab-app:latest username/docker-
lab-app:v1.0

# Push to Docker Hub
docker push username/docker-lab-app:latest
docker push username/docker-lab-app:v1.0

# Verify on Docker Hub website
echo "Visit:
https://hub.docker.com/r/username/docker-lab-app"
```

## Step 8: Test Pull and Run from Registry

```
# Stop and remove local container
docker stop my-app
docker rm my-app

# Remove local image
docker rmi docker-lab-app:latest

# Pull from Docker Hub and run
docker run -d -p 8080:3000 --name production-app
username/docker-lab-app:latest

# Test the deployed application
curl http://localhost:8080
```

```
# Check public access (replace with your EC2
public IP)
curl http://your-ec2-public-ip:8080
```

## Step 9: Container Management








```
# Monitor container resources
docker stats production-app

# View detailed container information
docker inspect production-app

# Update container with environment variables
docker stop production-app
docker rm production-app
docker run -d -p 8080:3000 -e NODE_ENV=production
--name production-app username/docker-lab-
app:latest

# Test environment variable
curl http://localhost:8080
```

## Lab Verification Checklist

-  Docker installed and running on EC2 Ubuntu instance
-  Node.js application created with health check endpoint
-  Dockerfile created with best practices (non-root user, health check)
-  Docker image built successfully
-  Container runs locally and responds to HTTP requests
-  Image tagged and pushed to Docker Hub
-  Image pulled from registry and deployed

-  Application accessible from public internet

**Congratulations!** You've successfully completed the Docker lab, demonstrating the complete container lifecycle from development to deployment.

*Prepared By: Rashi Rana  
Corporate Trainer*



# Summary and Best Practices

---

## Key Concepts Covered

- Container fundamentals and benefits over traditional deployment
- Docker architecture: Engine, Images, Containers, and Registry
- Essential Docker commands for container and image management
- Docker Hub setup and image registry operations
- Hands-on experience building and deploying containerized applications
- Complete Docker workflow from development to production

## Docker Best Practices

### Image Optimization

Use multi-stage builds, minimal base images, and .dockerignore files

### Security

Run as non-root user, scan for vulnerabilities, avoid secrets in images

### Tagging Strategy

### Resource Management

Use semantic versioning, meaningful tags, and consistent naming

Set resource limits, use health checks, and monitor container metrics

## Production Considerations

- **Orchestration:** Use Kubernetes, Docker Swarm, or cloud container services
- **Monitoring:** Implement logging, metrics, and alerting for containers
- **Networking:** Design proper network architecture for container communication
- **Storage:** Use persistent volumes for stateful applications
- **CI/CD Integration:** Automate build, test, and deployment pipelines
- **Security Scanning:** Regular vulnerability assessments and compliance checks

## Next Steps

1. **Container Orchestration:** Learn Kubernetes for production container management
2. **Docker Compose:** Multi-container application orchestration
3. **Advanced Networking:** Custom networks, service discovery, load balancing
4. **Persistent Storage:** Volumes, bind mounts, and storage drivers
5. **Security Hardening:** Container security best practices and tools
6. **Cloud Integration:** AWS ECS, Azure Container Instances, Google Cloud Run

## Troubleshooting Tips

### Container Issues

Check logs with `docker logs`, inspect with `docker inspect`

### Image Problems

Verify Dockerfile syntax, check build context, review layer history

### Network Connectivity

Verify port mappings, check firewall rules, test with `docker exec`

### Performance Issues

Monitor with `docker stats`, optimize images, set resource limits

## Additional Resources

- Docker Official Documentation: <https://docs.docker.com/>
- Docker Hub: <https://hub.docker.com/>
- Docker Best Practices: <https://docs.docker.com/develop/best-practices/>
- Dockerfile Reference: <https://docs.docker.com/engine/reference/builder/>

**Remember:** Containerization is not just about technology—it's about creating consistent, portable, and scalable applications that enable modern DevOps practices.