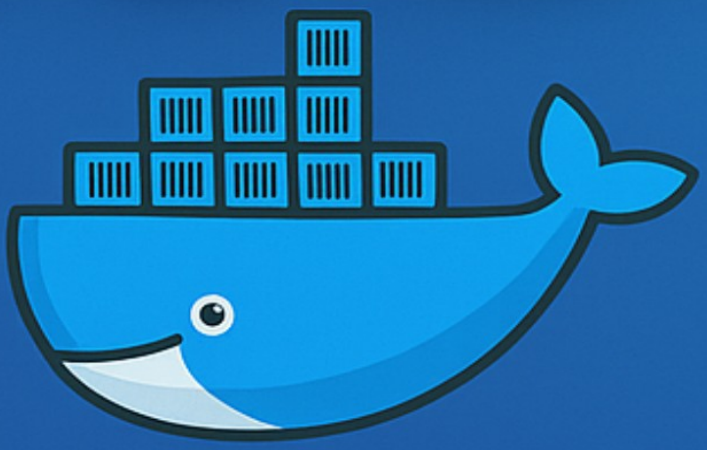
Devops Shack

Docker Comprehensive GuiDE



By DevOps Shack

Click here for DevSecOps and Cloud DevOps Course

DevOps Shack

Docker Comprehensive

Table of Contents

12. Docker Build – Building Custom Docker Images
13. Docker Hub – Using and Pushing Images
14. Docker Run – Running and Managing Containers
15. Docker Exec – Executing Commands Inside a Container
16. Docker Logs – Viewing Container Logs
17. Docker Volumes – Persistent Data Storage
18. Bind Mounts vs Volumes – Deep Comparison
19. Docker Networks – Types and Use Cases
20. Linking Containers using Custom Networks
21. Docker Compose – Purpose and Use Cases
22. Writing docker-compose.yml – Services, Networks, and Volumes 23.Environment Variables and `.env` Files in Compose

24.Docker Compose Commands – Up, Down, Build, Logs

25.Multi-container Applications using Docker Compose 26.Dockerfile Best Practices and Optimization

27.Docker Image Layers and Caching

28.Multi-stage Builds in Docker

29. Docker Security Basics – Image Scanning and Secrets

30. Managing Secrets in Docker Compose and Swarm

31. Dockerizing a Node.js Application – Complete Example

32. Dockerizing a Python Application – Complete Example

33. Dockerizing a Java Spring Boot App – Complete Example

34. Using Docker with CI/CD – GitHub Actions and GitLab CI

35. Docker in Kubernetes vs Docker Swarm

36. Understanding Docker Contexts and Remote Deployments

37. Docker System Prune, Clean-up and Maintenance

38. Docker Registry – Private and Public

Repositories 39.Troubleshooting Common Docker

Issues

40. Advanced Docker Use Cases – Sidecars, Init Containers

1. Introduction to Containerization and its Need

Containerization is the process of packaging an application and its dependencies into a standardized unit called a container. Before containers, developers faced the "It works on my machine" problem because of mismatched environments across development, testing, and production. Containers solve this by offering environment consistency, fast scalability, and lightweight deployment across platforms.

2. What is Docker and Why Use It

Docker is a platform and toolset for building, distributing, and running containers. It simplifies the containerization process and enables developers to quickly spin up isolated, reproducible environments. Docker's ecosystem includes tools for image creation, orchestration, registry management, and integration into CI/CD pipelines.

3. Difference Between Virtual Machines and Docker

VMs include a full OS with its own kernel, making them heavyweight and slow to boot. Docker containers share the host OS kernel, making them lightweight, fast, and resource-efficient. VMs emulate hardware; containers use OS-level virtualization. Docker uses namespaces and cgroups to isolate containers on a shared kernel.

4. Docker Architecture and Components

Docker has three core components:

- **Docker Engine (which includes the daemon)**

- **Docker CLI (command-line interface)**

- **Docker Objects (images, containers, volumes, networks).**
  It also includes Docker Compose, Docker Hub, and the Docker Registry.
  The client-server architecture allows Docker clients to communicate with the Docker daemon to build, run, and manage containers.

5. Understanding Docker Daemon, CLI, and Docker Client

- **Docker Daemon (`dockerd`): Listens for Docker API requests and manages objects.**

- **Docker Client (`docker`): The interface users interact with; it sends commands to the daemon.**

- **Docker API: REST API used by clients and tools.**
  Together, these components let developers efficiently build and run applications inside containers.

6. Docker Engine and How It Works

Docker Engine is the runtime that powers Docker. It handles:

- **Image building and layer management**

- **Container lifecycle (create, start, stop, destroy)**

- **Volume and network management**
  Docker Engine runs as a background service and interacts via the Docker API.

7. Installing Docker on Linux, Windows, and macOS

- Linux: Install via package manager (`apt`, `yum`). Configure post-install access for non-root users.
- Windows/macOS: Use Docker Desktop. It includes Docker Engine, CLI, Docker Compose, and Kubernetes.
  Docker Desktop also provides a GUI dashboard for managing containers and images.

8. Basic Docker Commands Every Developer Should Know

Key commands include:

- `docker version`, `docker info` – show system details
- `docker ps`, `docker images` – list containers/images
- `docker run`, `docker exec` – start a container or run a command
- `docker build`, `docker pull`, `docker push` – image operations
- `docker stop`, `docker rm`, `docker rmi` – container/image cleanup

9. Working with Docker Images

Docker images are read-only templates used to create containers. They are built using layers, each representing an instruction in a Dockerfile. Images can be pulled from Docker Hub or a private registry. The image contains everything: the app, dependencies, system libraries, and environment configurations.

10. Dockerfile – Purpose and Structure

A Dockerfile is a plain-text file with a set of instructions to build an image. Key instructions include:

- `FROM` **(base image)**
- `RUN` **(commands to execute during build)**
- `COPY` **/** `ADD` **(include files)**
- `CMD` **/** `ENTRYPOINT` **(default command)**
- `EXPOSE` **(port information)**
- `ENV` **(environment variables)**
  It serves as the blueprint for Docker images.

11. Writing a Simple Dockerfile – Step-by-Step

Example for Node.js:

dockerfile

```
FROM node:18

WORKDIR /app

COPY package*.json ./

RUN npm install
```

```
COPY . .

EXPOSE 3000

CMD ["node", "index.js"]
```

Each step builds a layer. Use `.dockerignore` to avoid copying unnecessary files like `node_modules`.

12. Docker Build – Building Custom Docker Images

Use the `docker build` command to create images from a Dockerfile:

```
docker build -t myapp:latest .
```

Flags like `-f` specify alternate Dockerfile location. Tag images clearly to manage versions. Avoid `latest` in production workflows for predictability.

13. Docker Hub – Using and Pushing Images

Docker Hub is the default registry for images. You can pull official or custom images:

```
docker pull nginx
```

To push your own image:

```
docker tag myapp myusername/myapp:v1

docker push myusername/myapp:v1
```

Set up private repositories for internal use cases.

14. Docker Run – Running and Managing Containers

Use `docker run` to start containers:

```
docker run -d -p 8080:80 nginx
```

Options like `-d` (detached), `-p` (port), `--name`, `-v` (volume), and `--network` make containers configurable.
Use `docker stop`, `start`, `restart`, and `rm` to manage them.

15. Docker Exec – Executing Commands Inside a Container

`docker exec` allows you to run a command in a running container. Example:

```
docker exec -it myapp
```

This is useful for debugging or running scripts in a live container. Combine it with `-it` for an interactive shell.

16. Docker Logs – Viewing Container Logs

To check logs from a container, use:

```
docker logs myapp
```

Add `-f` to follow logs in real-time. This helps monitor application behavior. Use logging drivers (`json-file`, `syslog`, `awslogs`, etc.) to send logs elsewhere.

17. Docker Volumes – Persistent Data Storage

By default, data inside containers is ephemeral. Use volumes to persist data:

```
docker volume create mydata
```

```
docker run -v mydata:/app/data myapp
```

Volumes are managed by Docker and ideal for databases or persistent application state.

18. Bind Mounts vs Volumes – Deep Comparison

- **Bind mounts use host file paths (e.g., `/home/user/data:/data`) and are good for development.**
- **Volumes are managed by Docker and are better for production. Volumes are portable, secure, and work across container restarts.**

19. Docker Networks – Types and Use Cases

Docker provides:

- **bridge (default for standalone containers)**
- **host (container uses host's network)**
- **overlay (for multi-host networking in Swarm) Networks enable secure communication between containers using names instead of IPs.**

20. Linking Containers using Custom Networks

Use user-defined bridge networks:

```
docker network create mynet
```

```
docker run --network=mynet --name=backend backend-app
```

```
docker run --network=mynet frontend-app
```

Containers can now communicate using `backend` as the hostname.

21. Docker Compose – Purpose and Use Cases

Docker Compose is a tool that allows you to define and manage multi-container applications using a simple YAML file. Instead of manually running multiple `docker run` commands, you can define services, networks, and volumes in one file and bring everything up with `docker-compose up`. It's ideal for local development, microservices, and quick prototyping.

22. Writing docker-compose.yml – Services, Networks, and Volumes

A basic `docker-compose.yml` structure includes: yaml

```
version: "3.9"
```

```
services:
```

```
  web:
```

```
  image: nginx
```

```
  Ports:
```

```
      - "8080:80"
  app:
    build: .
    ports:
      - "3000:3000"
    volumes:
      - ./app:/usr/src/app
volumes:
  dbdata:
networks:
  default:
    driver: bridge
```

You define services (containers), assign them to networks, and use volumes for persistent storage.

23. Environment Variables and `.env` Files in Compose

Compose supports environment variables for flexibility. You can inject values directly or use a `.env` file:

Yaml

```
environment:
  - NODE_ENV=production
```

Or inside `.env`:

ini

```
MYSQL_ROOT_PASSWORD=secret
```

This makes configuration easy and more secure, especially in different environments.

24. Docker Compose Commands – Up, Down, Build, Logs

Common commands:

- **`docker-compose up -d`: Start containers in detached mode**
- **`docker-compose down`: Stop and remove all containers, networks, and volumes**
- **`docker-compose build`: Rebuild images from Dockerfile**
- **`docker-compose logs -f`: Stream logs of services**
  These commands simplify lifecycle management for multi-container apps.

25. Multi-container Applications using Docker Compose

For example, a Node.js + MongoDB setup:

yaml

```
services:
  backend:
  build: ./backend
  ports:
  - "5000:5000"
  depends_on:
  - mongo
  mongo:
  image: mongo
  volumes:
  - mongo-data:/data/db
  volumes:
  mongo-data:
```

This launches both the app and the database together. Containers communicate by service names (e.g., `mongo`).

26. Dockerfile Best Practices and Optimization

Key best practices:

- **Use minimal base images like `alpine`**
- **Minimize layers and merge `RUN` commands**
- **Use `.dockerignore` to reduce build context**
- **Avoid hardcoded secrets**
- **Use multi-stage builds for smaller final images**
  Optimized Dockerfiles lead to faster builds and smaller images.

27. Docker Image Layers and Caching

Each instruction in a Dockerfile creates a layer. Docker caches these layers to speed up builds. If nothing changes in a layer, Docker reuses the cache. Therefore, put less frequently changing layers (e.g., `npm install`) before frequently changing ones (e.g., `COPY . .`).

28. Multi-stage Builds in Docker

Multi-stage builds allow you to separate build and runtime environments:

dockerfile

```
FROM node:18 AS builder
```

```
WORKDIR /app

COPY . .

RUN npm install && npm run build


FROM nginx:alpine

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

This keeps final images lean and secure by excluding build tools and intermediate files.

29. Docker Security Basics – Image Scanning and Secrets

Use tools like Trivy, Grype, or Docker Scan to check for known vulnerabilities in your images.
Avoid embedding secrets (API keys, passwords) in Dockerfiles or images. Leverage `.env` files, Docker Secrets, or secret managers like AWS Secrets Manager or HashiCorp Vault.

30. Managing Secrets in Docker Compose and Swarm

In Docker Swarm, secrets can be managed securely:

```
echo "my_secret" | docker secret create my_password -
```

In Compose, simulate secrets using external volumes or environment variables. But for production, prefer encrypted solutions with access control.

31. Dockerizing a Node.js Application – Complete Example

Dockerfile for Node.js:

dockerfile

```
FROM node:18

WORKDIR /app

COPY package*.json

./ RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]
```

Build and run:

```
docker build -t node-app .

docker run -p 3000:3000 node-app
```

32. Dockerizing a Python Application – Complete

Example Dockerfile for Flask:

dockerfile

```
FROM python:3.10

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

EXPOSE 5000

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

Run:

```
docker build -t flask-app .

docker run -p 5000:5000 flask-app
```

33. Dockerizing a Java Spring Boot App – Complete

Example Dockerfile for Spring Boot:

dockerfile

```
FROM openjdk:17-jdk-slim

ARG JAR_FILE=target/*.jar

COPY ${JAR_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]
```

Build:

```
mvn clean package

docker build -t springboot-app .

docker run -p 8080:8080 springboot-app
```

34. Using Docker with CI/CD – GitHub Actions and GitLab CI

Use Docker in CI/CD to:

- **Build and test in isolated environments**
- **Push Docker images to registries**
- **Deploy to staging or production Example GitHub Actions:**

yaml

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: docker build -t myapp .
```

35. Docker in Kubernetes vs Docker Swarm

- **Kubernetes is feature-rich, highly configurable, but complex**
- **Swarm is simpler to set up and uses Docker-native tools**
  Both handle container orchestration but Kubernetes has become the industry standard due to its flexibility and ecosystem.

36. Understanding Docker Contexts and Remote Deployments

Docker Contexts let you switch between multiple Docker environments (local, remote, cloud):

```
docker context create remote --docker
"host=ssh://user@server"
```

```
docker context use remote
```

```
docker ps
```

This is ideal for deploying containers to remote servers without logging in.

37. Docker System Prune, Clean-up and Maintenance

To clean up unused containers, images, volumes, and networks:

```
docker system prune
```

```
docker volume prune
```

```
docker image prune
```

Use with care in production environments. Regular cleanup saves disk space and keeps your environment healthy.

38. Docker Registry – Private and Public Repositories

You can host your own Docker Registry or use third-party options:

- **Docker Hub**
- **GitHub Container Registry**
- **Harbor (open-source)**
- **AWS ECR, GCP Artifact Registry**
  Private registries let teams securely manage internal images.

39. Troubleshooting Common Docker Issues

Common issues include:

- **Container not starting (check `docker logs`)**
- **Port conflicts (use `docker ps` to find existing mappings)**
- **Build errors (check Dockerfile and context)**
- **Volume issues (check mount paths and permissions)**
  Always start debugging with `docker inspect` or `docker logs`.

40. Advanced Docker Use Cases – Sidecars, Init Containers, and More

- **Sidecar containers: Containers that augment the primary app (e.g., logging agents, proxies)**
- **Init containers: Run before the main app for setup (common in Kubernetes)**
- **Service Mesh integrations: Like Istio or Linkerd**
- **Custom entrypoints for complex bootstrapping**
  Docker's flexibility allows for creative and powerful container-based architectures.