Complete Docker DevOps Training Guide

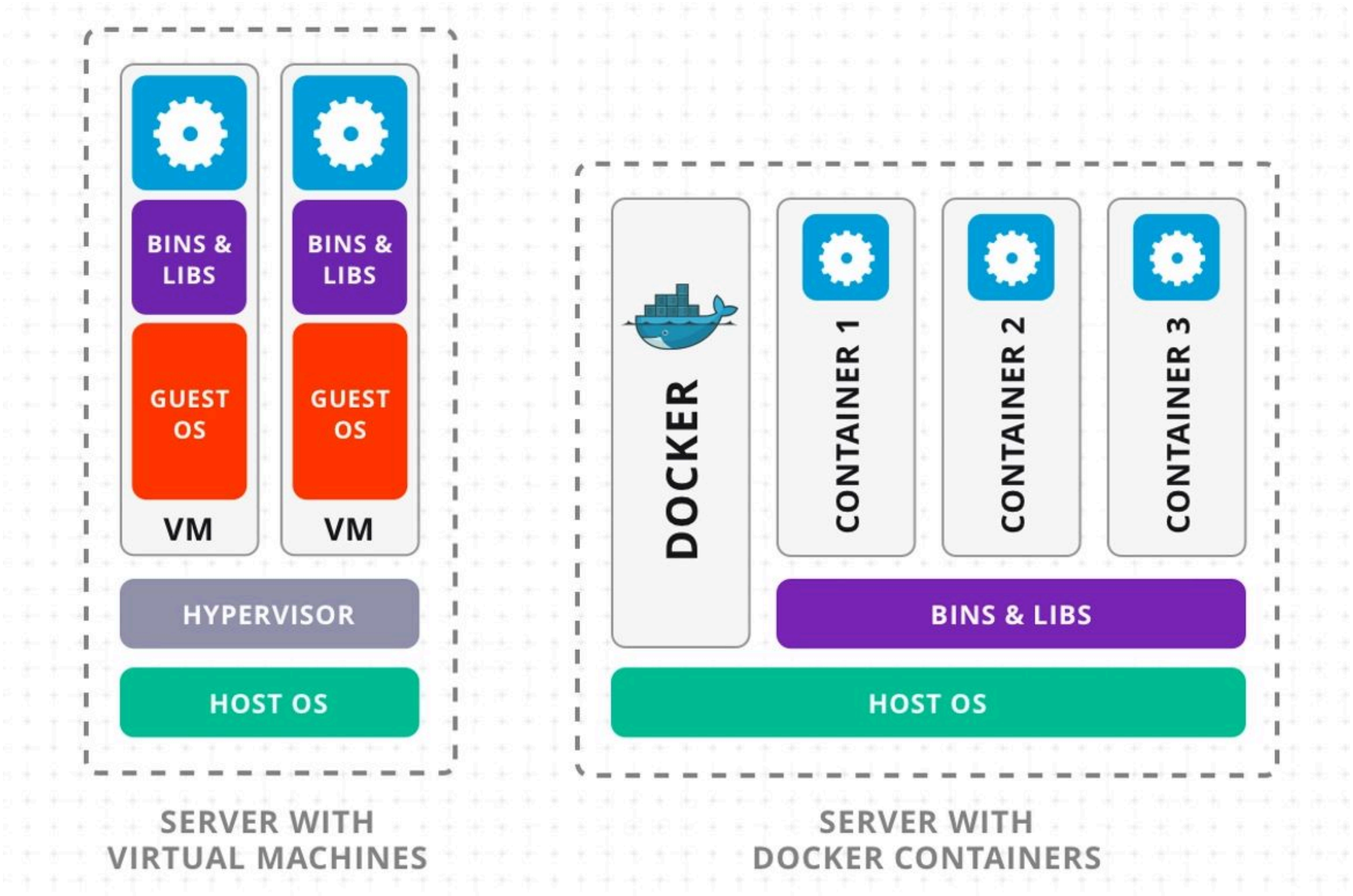
Open: Pasted image 20250621173436.png

Install Docker Engine on Ubuntu

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
sudo usermod -aG docker username && newgrp docker
 ## This command activates your new `docker` group membership in the current
```

0. Docker VS Vm

Open: Pasted image 20250621163544.png



Docker vs Virtual Machines: Key Differences

Feature	Docker Containers	Virtual Machines
Architecture	Shares the host OS kernel, uses containerization technology	Runs on hypervisor with complete OS virtualization
Operating System	Shares the host OS kernel, no guest OS needed	Requires full guest OS for each VM
Isolation Level	Process-level isolation through namespaces and cgroups	Complete hardware-level isolation
Resource Utilization	Lightweight, uses resources on demand	Heavier, pre-allocates resources
Boot Time	Seconds or milliseconds	Minutes
Performance	Near-native performance with minimal overhead	Some overhead due to hypervisor layer
Storage Requirements	Typically MBs in size (10-100x smaller than VMs)	Typically GBs in size
Portability	Highly portable across any system running Docker	Less portable, requires compatible hypervisor

Feature	Docker Containers	Virtual Machines
Security	Less isolated than VMs, potential kernel vulnerabilities	Strong isolation, each VM has its own kernel
Scalability	Can run hundreds of containers on a single host	Limited by physical resources, typically dozens per host

**Security Comparison

Virtual machines provide stronger isolation since each VM has its own kernel and operating system, making them more suitable for high-security environments. Docker containers share the host kernel, which can potentially lead to security vulnerabilities if one container is compromised.

**Use Cases

**When to Use Docker Containers

- Microservices architecture implementation
- Continuous integration/continuous deployment (CI/CD) pipelines
- Development and testing environments
- Applications requiring rapid scaling and deployment

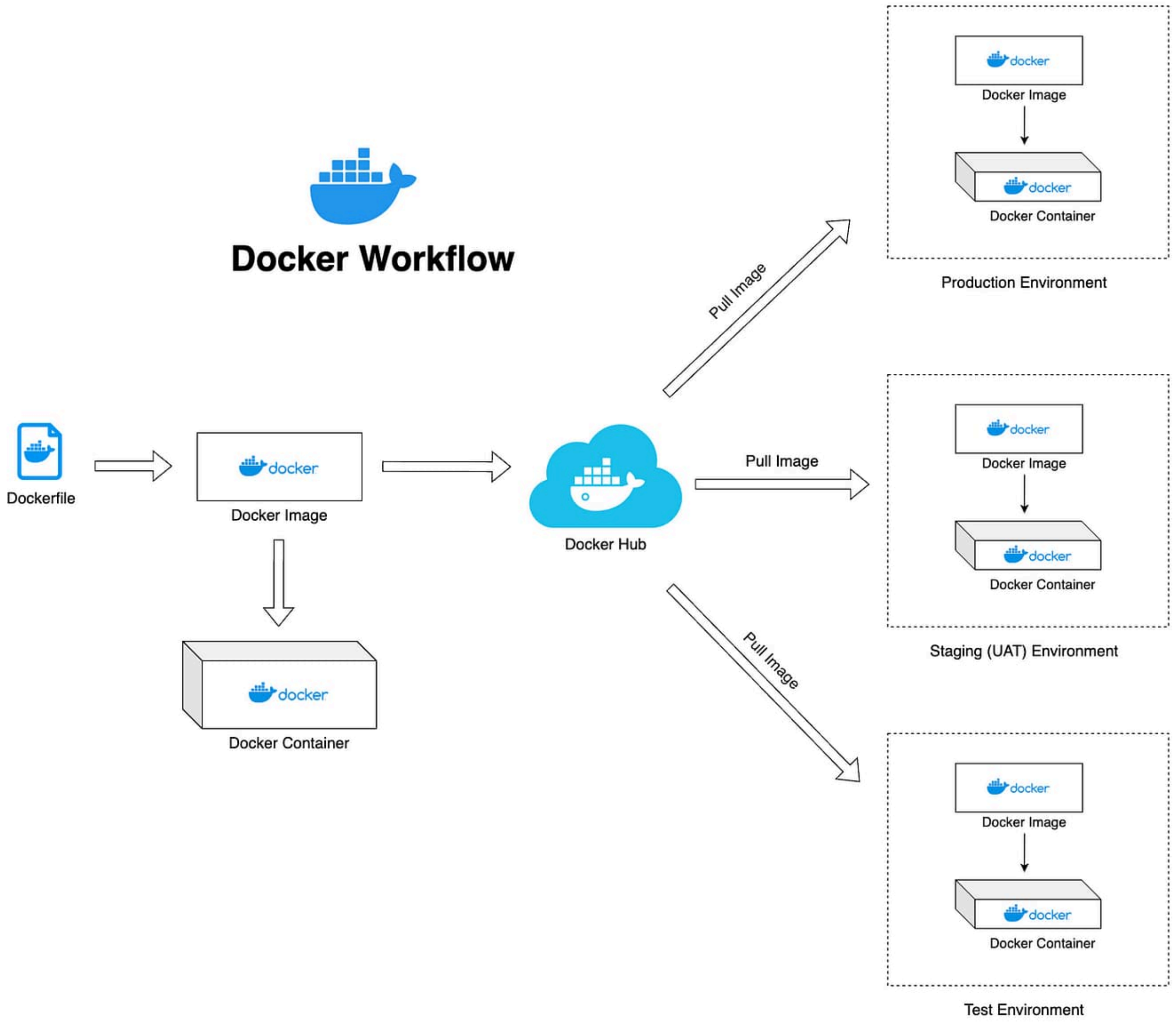
**When to Use Virtual Machines

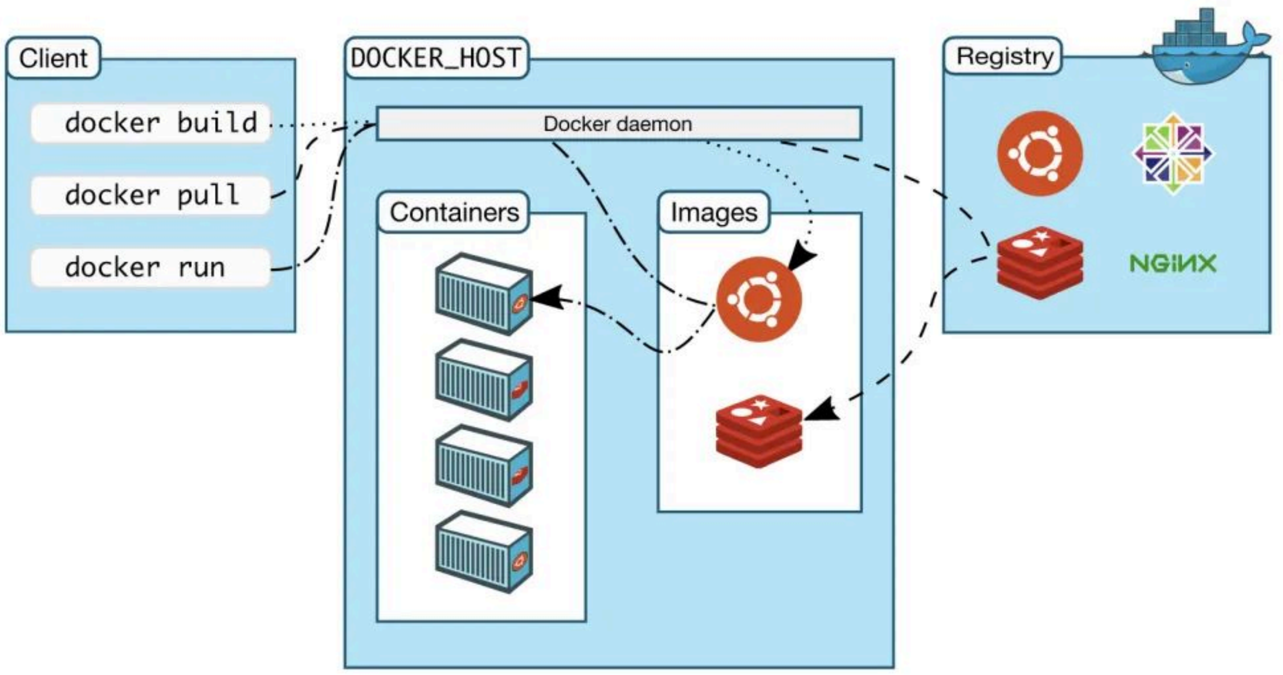
- Running applications with different operating system requirements
- Legacy applications that require specific OS versions
- Workloads requiring strong security isolation
- Applications with substantial hardware resource requirements

1. Docker Architecture: Core Components and Workflow

Docker implements a client-server architecture where multiple components work together to create, distribute, and run containerized applications. This architecture consists of several key components that handle different aspects of the containerization process.

00. Docker Workflow


Docker Workflow
Dockerfile
docker
Docker Image
docker
Docker Container
Docker Hub
Pull Image
Pull Image
Pull Image
docker
Docker Image
docker
Docker Container
Production Environment
docker
Docker Image
docker
Docker Container
Staging (UAT) Environment
docker
Docker Image
docker
Docker Container
Test Environment


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

1.1 Docker Daemon (dockerd)

The Docker daemon (dockerd) is the persistent process that manages Docker objects including images, containers, networks, and volumes. It listens for Docker API requests and processes them accordingly. The daemon is responsible for:

- Building and storing images
- Creating and managing containers
- Establishing networks between containers
- Managing persistent storage volumes
- Handling container lifecycle events

When you execute Docker commands, the daemon performs a series of internal operations to fulfill your request, often interacting with lower-level components like containerd.

1.2 Docker Client

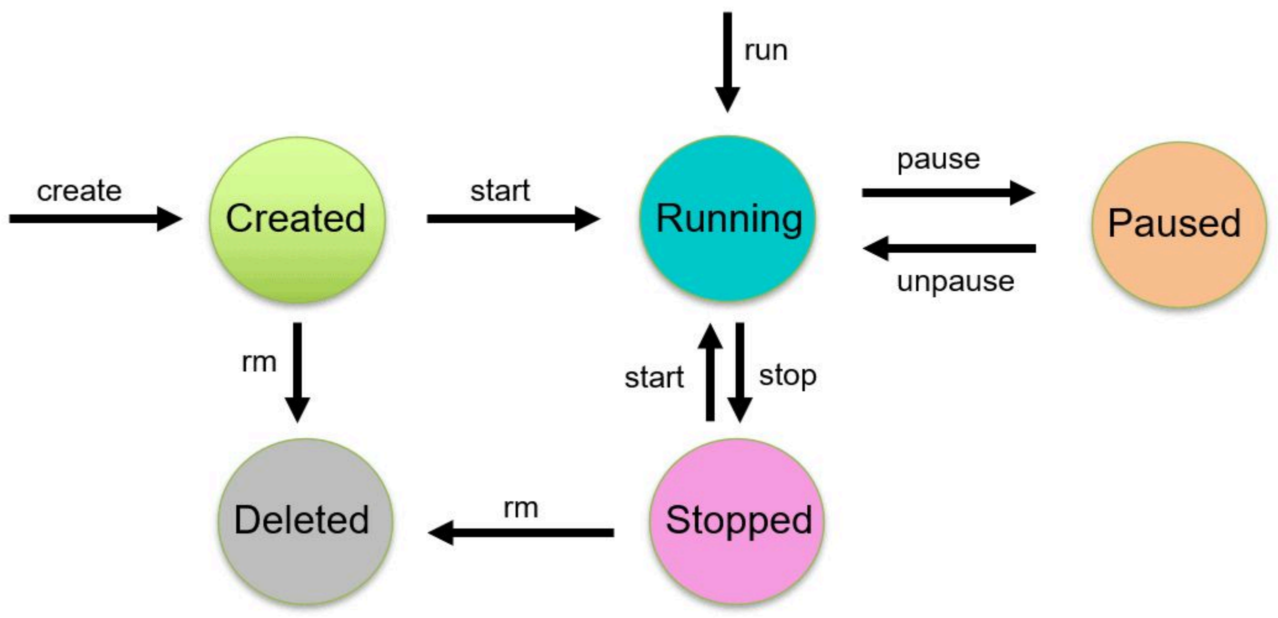
The Docker client is the primary way users interact with Docker through the command-line interface (CLI). When you run commands like `docker run` or `docker build`, the client:

1. Processes your command
2. Converts it to appropriate API requests
3. Sends these requests to the Docker daemon
4. Waits for and displays the response

The client can communicate with the daemon on the same system or connect to a remote daemon, providing flexibility in how you manage your Docker infrastructure.

2. Container Lifecycle Management

Open: Pasted image 20250621140340.png



Docker containers follow a well-defined lifecycle with distinct states and transitions between them. Understanding this lifecycle is crucial for effective container management.

2.1 Container States

Containers can exist in five primary states throughout their lifecycle:

State	Description	Command
Created	Container is initialized from an image but not started	`docker create`
Running	Container is active and executing processes	`docker start`, `docker run`
Paused	Container processes are temporarily suspended	`docker pause`
Stopped	Container is shut down but still exists	`docker stop`
Deleted	Container is permanently removed	`docker rm`

Each state represents a specific operational condition of the container, and Docker provides commands to transition between these states.

2.2 Lifecycle Commands

2.2.1 Creating Containers

```
# Create a container without starting it
docker create --name web-server nginx:alpine

# Create with port mapping and environment variables
docker create -p 8080:80 -e ENV_VAR=value --name api-server api-image:1.0
```

The create command prepares a container from an image but doesn't start it, allowing you to configure it before running.

2.2.2 Starting and Running Containers

```
# Start a previously created container
docker start web-server

# Create and start a container in one command
docker run -d --name database -v data:/var/lib/postgresql/data postgres:15
```

The `docker run` command combines `create` and `start` operations, while `docker start` initiates a previously created container.

2.2.3 Pausing and Unpausing Containers

```
# Pause a running container
docker pause web-server

# Resume a paused container
docker unpause web-server
```

Pausing a container sends a SIGSTOP signal to all processes in the container, suspending them without termination, while unpausing sends SIGCONT to resume execution.

2.2.4 Stopping Containers

```
# Stop a container gracefully (SIGTERM, then SIGKILL)
docker stop web-server

# Force immediate stop (SIGKILL)
docker kill web-server
```

The `stop` command attempts a graceful shutdown by sending SIGTERM first, followed by SIGKILL if the container doesn't stop within the timeout period (default 10 seconds).

docker paus

- All processes are frozen but remain in memory.
- No CPU is used, but memory and state are preserved.
- Network sockets and file handles stay open.
- Can be resumed instantly with `docker unpause`.
- Useful for temporarily halting activity without losing state or for testing/debugging.

docker stop

- Sends a termination signal (SIGTERM), then a kill signal (SIGKILL) to processes.
- All processes are stopped and memory is released.
- Network connections are closed.
- To restart, the container must be started again, which reinitializes processes and state.
- Used for permanent or longer-term shutdowns, freeing up system resources

2.2.5 Removing Containers

```
# Remove a stopped container
docker rm web-server

# Force remove a running container
docker rm -f web-server

# Remove all stopped containers
docker container prune
```

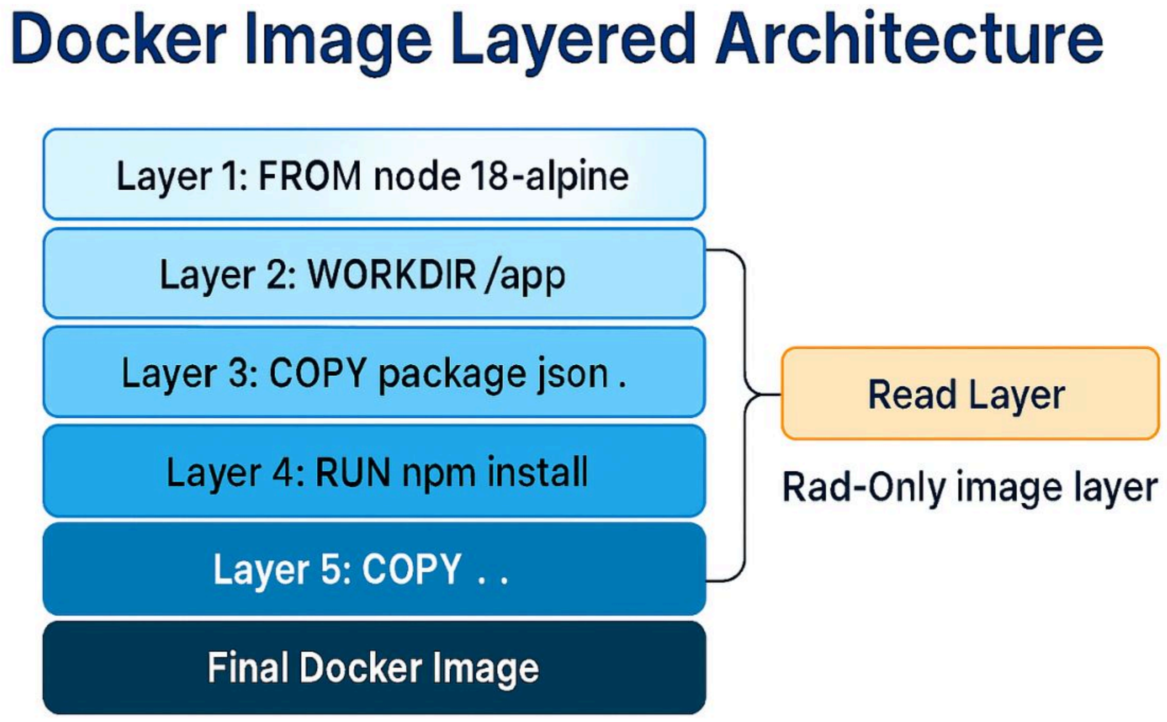
Removing a container permanently deletes it and frees up resources, though any data not stored in volumes will be lost.

3. Docker Images: Building and Optimization

Docker images are the foundation of containers, containing the application code, runtime, libraries, and dependencies needed to run an application. Understanding how to build and optimize images is essential for efficient Docker usage.

3.1 Docker Image And Layering Concept

![[f16d4803d06b63efb179cb3ab82aaa8b_MD5.jpeg]]
Open: Pasted image 20250621141318.png



Docker images are composed of multiple read-only layers, each representing a set of filesystem changes. These layers are stacked on top of each other to form the final image:

- **Efficient storage**: Layers are cached and reused across images
- **Faster builds**: Only modified layers need to be rebuilt
- **Bandwidth optimization**: Only new or changed layers are transferred during pulls/pushes

3.2 Dockerfile Best Practices

***First Example

```
FROM python:3.11-slim AS base

# Add curl for healthcheck
RUN apt-get update && \
    apt-get install -y --no-install-recommends curl && \
    rm -rf /var/lib/apt/lists/*

# Set the application directory
WORKDIR /usr/local/app

# Install our requirements.txt
COPY requirements.txt ./requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# dev defines a stage for development, where it'll watch for filesystem chang

RUN pip install watchdog
ENV FLASK_ENV=development


# Copy our code from the current folder to the working directory inside the c
COPY . .

# Make port 80 available for links and/or publish
EXPOSE 80

# Define our command to be run when launching the container
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--log-file", "-", "--access-
```

Creating efficient Dockerfiles is crucial for building optimized images:

```
FROM python:3.11-slim AS base

# Add curl for healthcheck
RUN apt-get update && \
    apt-get install -y --no-install-recommends curl && \
    rm -rf /var/lib/apt/lists/*

# Set the application directory
WORKDIR /usr/local/app

# Install our requirements.txt
COPY requirements.txt ./requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
```

```
# dev defines a stage for development, where it'll watch for filesystem chang
FROM base AS dev
RUN pip install watchdog
ENV FLASK_ENV=development
CMD ["python", "app.py"]

# final defines the stage that will bundle the application for production
FROM base AS final

# Copy our code from the current folder to the working directory inside the c
COPY . .

# Make port 80 available for links and/or publish
EXPOSE 80

# Define our command to be run when launching the container
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:80", "--log-file", "-", "--access-
```

Key best practices include:

- Using specific base image tags for reproducibility
- Ordering instructions from least to most frequently changing
- Combining related commands to reduce layer count
- Removing unnecessary files in the same layer they're created
- Using .dockerignore to exclude unnecessary files

3.3 Multi-Stage Builds

Multi-stage builds allow you to use multiple FROM statements in a single Dockerfile, with each stage building on the previous one. This approach significantly reduces final image size by including only necessary artifacts:

```
FROM node:18-slim

COPY ./
# add curl for healthcheck
RUN apt-get update && \
    apt-get install -y --no-install-recommends curl tini && \
    rm -rf /var/lib/apt/lists/*
```

```
WORKDIR /usr/local/app

# have nodemon available for local dev use (file watching)
RUN npm install -g nodemon

COPY package*.json ./

RUN npm ci && \
 npm cache clean --force && \
 mv /usr/local/app/node_modules /node_modules

COPY . .

ENV PORT=80
EXPOSE 80

# ENTRYPOINT ["/usr/bin/tini", "--"]
CMD ["node", "server.js"]
```

Open: Pasted image 20250622113303.png

```
NAME                                                    READY   STATUS        RESTARTS      AGE
alertmanager-main-0                                     0/6     Terminating   6             212d
```

```
# === Stage 1: Build dependencies and app ===
FROM node:18  AS build

# Add curl and tini for healthcheck and init
RUN apt-get update && \
    apt-get install -y --no-install-recommends curl && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /usr/local/app

# Install nodemon globally for development (not copied to final image)
RUN npm install -g nodemon

# Copy package files and install dependencies
COPY package*.json ./
RUN npm ci && \
    npm cache clean --force

# Copy app source code
COPY . .

# === Stage 2: Production image ===
FROM node:18-slim

# Install tini and curl for healthcheck/init
RUN apt-get update && \
    apt-get install -y --no-install-recommends tini  && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /usr/local/app

# Copy only the node_modules and built app from build stage
COPY --from=build /usr/local/app /usr/local/app
COPY --from=build /node_modules /node_modules

ENV PORT=80
EXPOSE 80

ENTRYPOINT ["/usr/bin/tini", "--"]
CMD ["node", "server.js"]
```

Benefits of multi-stage builds include:

- **Smaller final images**: Only production artifacts are included
- Build tools stay in the build stage
- **Improved security**: Fewer packages in the final image means a smaller attack surface
- **Parallel execution**: Independent stages can run concurrently

Multi-stage builds are particularly valuable for compiled languages or frontend applications where build tools aren't needed at runtime.

lab: https://github.com/docker/awesome-compose

4. Docker Volumes: Advanced Data Management

Docker volumes provide persistent storage for containers, allowing data to survive container lifecycle events. Understanding the different volume types and their use cases is essential for proper data management.

4.1 Volume Types and Use Cases

Docker offers several storage options, each with specific characteristics and use cases:

4.1.1 Named Volumes

Named volumes are managed by Docker and stored in a designated location on the host (`/var/lib/docker/volumes/` on Linux).

```
# Create a named volume
docker volume create postgres_data

# Use the volume with a container
docker run -d --name db -v postgres_data:/var/lib/postgresql/data postgres:15
```

Use cases:

- Database storage
- Application data that needs to persist
- Sharing data between containers
- When you need Docker to manage volume lifecycle

4.1.2 Anonymous Volumes

Anonymous volumes are similar to named volumes but with automatically generated names.

```
# Create an anonymous volume
docker run -d --name cache -v /data/cache redis:alpine
```

Use cases:

- Temporary data that should outlive the container
- When volume naming isn't important
- For quick testing scenarios

4.1.3 Bind Mounts

Bind mounts map a specific host directory to a container path.

```
# Mount a host directory to a container
docker run -d --name web -v /home/user/website:/usr/share/nginx/html nginx:al
```

docker run -d --name web1 nginx:alpine
Use cases:

- Development environments for live code updates
- Sharing configuration files from the host
- Accessing host filesystem from containers
- When you need direct control over storage location

4.1.4 tmpfs Mounts

tmpfs mounts store data in the host's memory only.

```
# Create a tmpfs mount
docker run -d --name temp-data --tmpfs /app/temp alpine
```

Use cases:

- Sensitive data that shouldn't be persisted
- High-performance temporary storage
- When you need to ensure data is cleared on container stop

4.2 Volume Management Commands

Docker provides a comprehensive set of commands for managing volumes throughout their lifecycle:

```
# Create a volume
docker volume create logs_data

# List all volumes
docker volume ls

# Inspect volume details
docker volume inspect logs_data
```

```
# Remove a specific volume
docker volume rm logs_data

# Remove all unused volumes
docker volume prune
```

These commands allow you to manage volumes independently of containers, providing flexibility in how you handle persistent data.

4.3 Data Sharing Between Containers

Volumes can be shared between containers, enabling data exchange and communication:

```
# Create a volume for sharing
docker volume create shared_data

# Use the volume in multiple containers
docker run -d --name writer -v shared_data:/data alpine sh -c "while true; do
docker run -d --name reader -v shared_data:/data alpine sh -c "while true; do
```

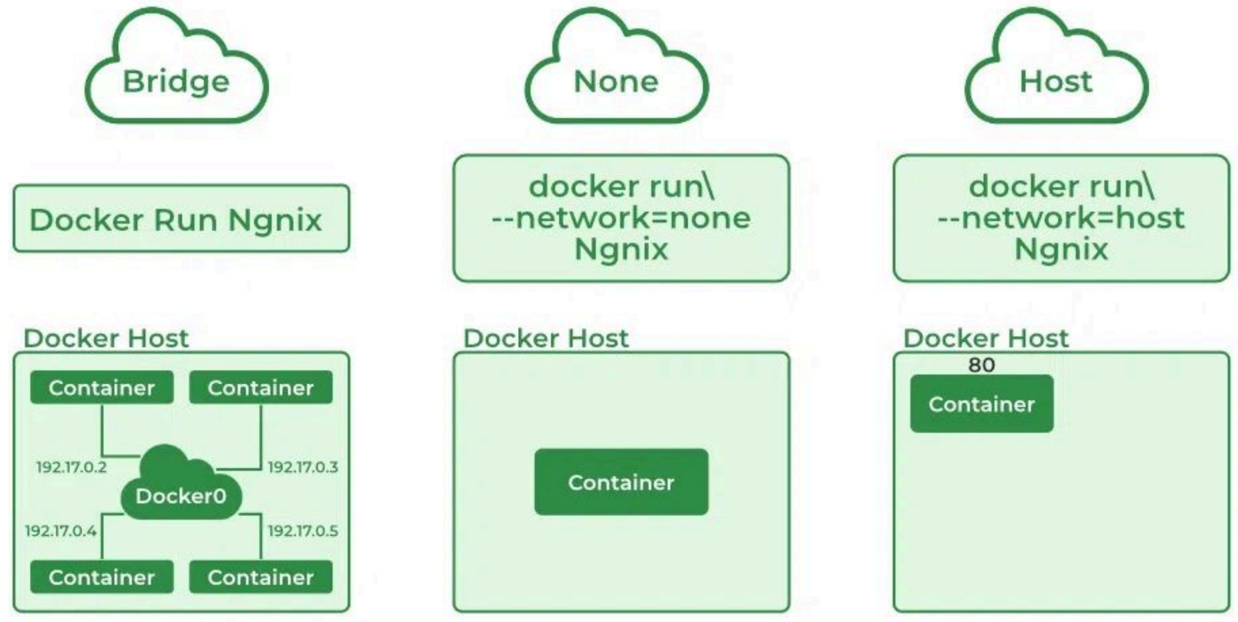
This approach allows containers to work with the same data without direct network communication, which is useful for:

- Microservice architectures where services need to share data
- Worker patterns where multiple containers process shared files
- Backup and restore operations
- Database replication scenarios

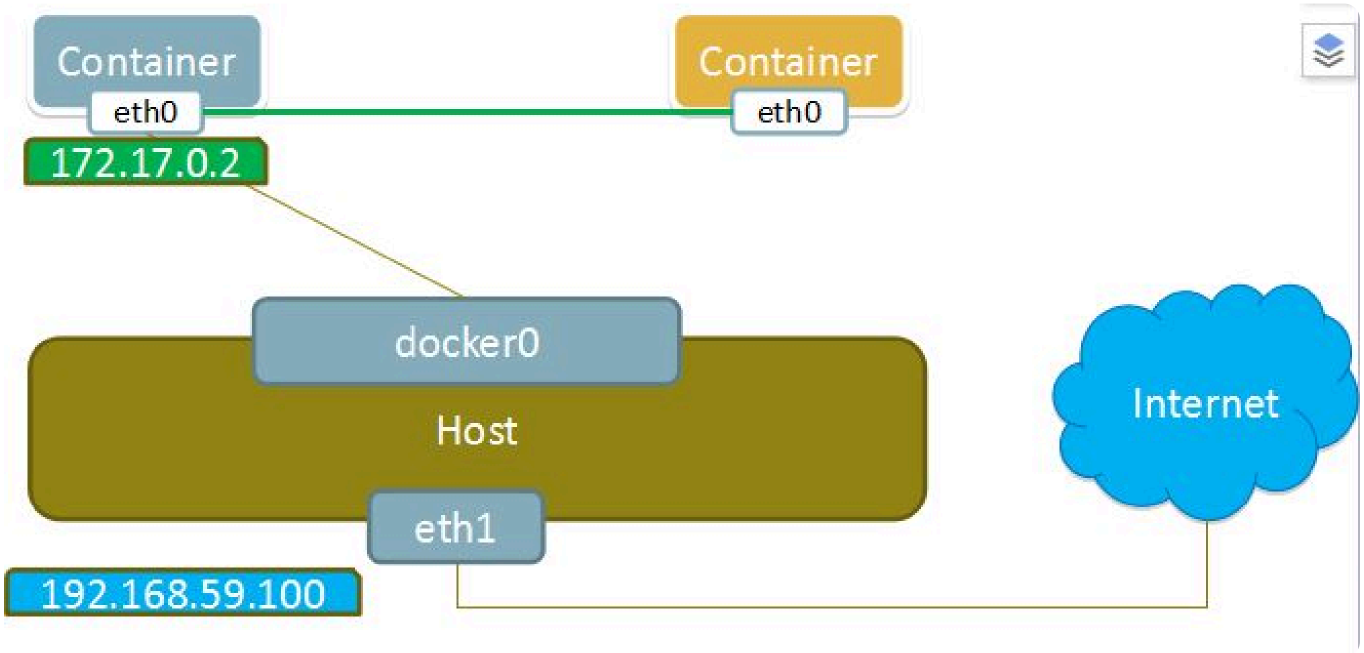
5. Docker Networking: Deep Dive

Docker networking enables communication between containers and with external networks. Understanding the different network types and their characteristics is crucial for designing effective container architectures.

Open: Pasted image 20250622002607.png



Open: Pasted image 20250622002705.png



5.1 Network Drivers and Types

Docker supports several network drivers, each designed for specific use cases:

5.1.1 Bridge Networks

The default network type in Docker, providing isolated networks for containers on the same host.

```
# Create a custom bridge network
docker network create --driver bridge app_network

# Connect containers to the network
docker run -d --name api --network app_network api-image
docker run -d --name db --network app_network db-image
```

Characteristics:

- Containers on the same bridge can communicate via container names (DNS resolution)
- Isolated from other bridge networks
- Provides NAT for outbound connectivity
- Configurable with custom subnets and IP ranges

5.1.2 Host Network

Containers using the host network share the host's network namespace, eliminating network isolation.

```
# Run a container with host networking
docker run -d --network host nginx
```

Characteristics:

- No network isolation between container and host
- Direct access to host network interfaces
- Better performance (no NAT overhead)
- Port conflicts possible with host services
- Cannot run multiple containers on the same port

5.1.3 Overlay Networks

Enables communication between containers across multiple Docker hosts, essential for swarm mode.

```
# Create an overlay network (in swarm mode)
docker network create --driver overlay --attachable swarm_network
```

Characteristics:

- Spans multiple Docker hosts
- Uses VXLAN encapsulation for container-to-container traffic
- Provides built-in service discovery and load balancing
- Supports encrypted communication between nodes

5.1.4 Macvlan Networks

Assigns MAC addresses to containers, making them appear as physical devices on the network.

```
# Create a macvlan network
docker network create --driver macvlan \
  --subnet=192.168.1.0/24 \
  --gateway=192.168.1.1 \
  -o parent=eth0 macvlan_network
```

Characteristics:

- Containers receive their own MAC and IP addresses
- Appears as physical devices on the network
- Direct communication with physical network
- Requires promiscuous mode on host interface

5.2 Network Configuration and Management

Docker provides commands to create, inspect, and manage networks:

```
# List all networks
docker network ls

# Inspect network details
docker network inspect app_network
```

```
# Connect a running container to a network
docker network connect app_network existing_container

# Disconnect a container from a network
docker network disconnect app_network container_name

# Remove a network
docker network rm app_network
```

These commands allow you to manage the complete lifecycle of Docker networks and control container connectivity.

5.3 Container Communication Patterns

Docker networking supports various communication patterns between containers:

5.3.1 Service Discovery

Containers on the same network can resolve each other by name using Docker's embedded DNS server:

```
# From inside 'api' container
curl http://db:5432/
```

This automatic DNS resolution simplifies container communication without hardcoding IP addresses.

5.3.2 Port Publishing

To allow external access to container services, you can publish container ports to the host:

```
# Publish container port 80 to host port 8080
docker run -d -p 8080:80 nginx

# Publish to a random host port
docker run -d -P nginx
```

Port publishing creates NAT rules that forward traffic from the host to the container.

5.3.3 Network Isolation

Docker networks provide isolation between container groups, allowing you to segment your applications:

```
# Create isolated networks for different application tiers
docker network create frontend
docker network create backend

# Connect containers to appropriate networks
docker run -d --name web --network frontend web-image
docker run -d --name api --network frontend --network backend api-image
docker run -d --name db --network backend db-image
```

This pattern allows the API container to communicate with both frontend and backend services while keeping the database isolated from direct frontend access.

6. Docker Health Checks and Monitoring

Health checks allow Docker to monitor the health of containerized applications and take appropriate actions when issues are detected.

6.1 Implementing Health Checks

Health checks can be defined in Dockerfiles or at runtime:

6.1.1 In Dockerfile

```
FROM nginx:alpine

# Add health check
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost/ || exit 1
```

```
# Rest of Dockerfile...
```

6.1.2 At Runtime

```
docker run -d --name web \
  --health-cmd="curl -f http://localhost/ || exit 1" \
  --health-interval=30s \
  --health-timeout=10s \
  --health-start-period=5s \
  --health-retries=3 \
  nginx:alpine
```

Health check parameters include:

- `--interval` : Time between health checks (default: 30s)
- `--timeout` : Maximum time for a check to complete (default: 30s)
- `--start-period` : Initialization time before checks count (default: 0s)
- `--retries` : Number of consecutive failures before unhealthy (default: 3)
- `CMD` : Command to run for health check (exit 0 = healthy, non-zero = unhealthy)

6.2 Health Check States

Containers with health checks can be in one of three health states:

- `starting` : During the start period, checks run but failures don't count
- `healthy` : The health check is passing
- `unhealthy` : The health check has failed the specified number of retries

You can view the health status of containers using:

```
# View health status in container list
docker ps

# Get detailed health information
docker inspect --format='{{.State.Health.Status}}' container_name
```

```
# View health check logs
docker inspect --format='{{json .State.Health}}' container_name | jq
```

6.3 Health Checks in Docker Compose

Health checks can also be defined in Docker Compose files:

```yaml
version: '3.8'
services:
  web:
    image: nginx:alpine
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost/"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 5s
    depends_on:
      db:
        condition: service_healthy

  db:
    image: postgres:15
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5
    environment:
      POSTGRES_PASSWORD: example
```

The `depends_on` with `condition: service_healthy` ensures that services start only after their dependencies are healthy, creating robust startup sequences.

7. Docker Compose: Advanced Features and Patterns

Docker Compose simplifies multi-container application management with a declarative YAML configuration. Beyond basic usage, Compose offers advanced features for complex deployments.

7.1 Service Dependencies and Startup Order

Docker Compose provides mechanisms to control service startup order:

```yaml
version: '3.8'
services:
  web:
    image: nginx:alpine
    depends_on:
      db:
        condition: service_healthy
      cache:
        condition: service_started

  db:
    image: postgres:15
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5

  cache:
    image: redis:alpine
```

The `depends_on` directive supports several conditions:

- `service_started` : Wait until the dependency container has started
- `service_healthy` : Wait until the dependency container's health check passes
- `service_completed_successfully` : Wait until the dependency container has completed successfully (for one-time tasks)

This ensures proper initialization sequence for interdependent services.

7.2 Environment Variable Management

Docker Compose offers multiple ways to manage environment variables:

7.2.1 Using .env Files

```yaml
# docker-compose.yml
version: '3.8'
services:
  web:
    image: nginx:alpine
    environment:
      - DEBUG=${DEBUG:-false}
      - API_URL=${API_URL}
```

With a `.env` file in the same directory:

```
DEBUG=true
API_URL=https://api.example.com
```

7.2.2 Using environment Section

```yaml
version: '3.8'
services:
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: appuser
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: appdb
```

7.2.3 Using env_file

```yaml
version: '3.8'
services:
  api:
    image: api-image
    env_file:
      - ./common.env
      - ./api.env
```

This allows grouping environment variables in separate files for better organization.

7.3 Secrets Management

For sensitive data, Docker Compose supports secrets management:

```yaml
version: '3.8'
services:
  api:
    image: api-image
    environment:
      - API_KEY_FILE=/run/secrets/api_key
    secrets:
      - api_key

  db:
    image: postgres:15
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/db_password
    secrets:
      - db_password

secrets:
  api_key:
    file: ./secrets/api_key.txt
  db_password:
    file: ./secrets/db_password.txt
```

This approach:

- Mounts secret files at `/run/secrets/` inside containers
- Keeps secrets out of environment variables
- Provides better isolation between services
- Supports the `_FILE` suffix convention used by many official images

7.4 Service Scaling

Docker Compose allows scaling services horizontally for increased capacity:

```bash
# Scale the web service to 3 instances
docker compose up --scale web=3
```

For this to work effectively with published ports, you need to either:

1. Use different host ports for each instance
2. Use a reverse proxy to distribute traffic

```yaml
version: '3.8'
services:
  web:
    image: nginx:alpine
    deploy:
      replicas: 3
    ports:
      - "8080-8082:80"  # Range of host ports
```

This configuration maps each container's port 80 to a different host port in the range 8080-8082.

**Custom Networks for Service Isolation

Creating custom networks allows logical separation of services and controlled communication paths:

```yaml
version: '3.8'
services:
  frontend:
    image: frontend-image
    networks:
      - frontend-net

  api:
    image: api-image
    networks:
      - frontend-net
      - backend-net

  db:
    image: postgres:15
    networks:
      - backend-net

networks:
  frontend-net:
  backend-net:
```

This pattern:

- Isolates the database from direct frontend access
- Creates clear communication boundaries
- Improves security through network segmentation
- Simplifies service discovery within each network

8. Docker Security Best Practices

Security is a critical aspect of Docker deployments, requiring attention at multiple levels.

8.1 Image Security

Securing Docker images is the first step in building a secure container environment:

```
# Use specific version tags
FROM node:18-alpine

# Update packages and remove package cache
RUN apk update && \
    apk upgrade && \
    apk add --no-cache dumb-init && \
    rm -rf /var/cache/apk/*

# Create non-root user
RUN addgroup -g 1001 appgroup && \
    adduser -u 1001 -G appgroup -s /bin/sh -D appuser

# Set proper permissions
WORKDIR /app
COPY --chown=appuser:appgroup . .

# Use non-root user
USER appuser

# Use init system for proper signal handling
ENTRYPOINT ["/usr/bin/dumb-init", "--"]
CMD ["node", "server.js"]
```

Key image security practices include:

- Using specific image tags instead of `latest`

- Keeping base images updated with security patches
- Scanning images for vulnerabilities with tools like Trivy or Docker Scout
- Using minimal base images (Alpine, distroless) to reduce attack surface
- Implementing multi-stage builds to exclude build tools from final images
- Never embedding secrets in images

8.2 Runtime Security

Securing container runtime involves proper configuration and resource constraints:

```
# Run with security options
docker run -d \
  --name secure-app \
  --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE \
  --security-opt=no-new-privileges \
  --read-only \
  --tmpfs /tmp \
  secure-image
```

Important runtime security practices include:

- Dropping unnecessary capabilities
- Using read-only filesystems where possible
- Implementing resource limits to prevent DoS attacks
- Setting the `no-new-privileges` flag
- Using seccomp profiles to restrict system calls
- Implementing network segmentation with custom networks

**Default Docker Capabilities (Can Be Dropped with --cap-drop)

Docker containers run with a restricted set of Linux capabilities by default. The following table shows all the default capabilities that Docker allows and can be dropped using the `--cap-drop` flag for improved security:

Capability	Description	Security Risk Level	Recommended Action
AUDIT_WRITE	Write records to kernel auditing log	Low	Can drop unless logging required

Capability	Description	Security Risk Level	Recommended Action
CHOWN	Make arbitrary changes to file UIDs and GIDs	Medium	Drop unless file ownership changes needed
DAC_OVERRIDE	Bypass file read, write, and execute permission checks	High	Drop unless privileged file access required
FOWNER	Bypass permission checks on operations requiring file system UID match	Medium	Drop unless file ownership operations needed
FSETID	Don't clear set-user-ID and set-group-ID permission bits	Medium	Drop unless setuid/setgid programs required
KILL	Bypass permission checks for sending signals	Medium	Drop unless signal management needed
MKNOD	Create special files using mknod(2)	Medium	Drop unless device file creation required
NET_BIND_SERVICE	Bind a socket to privileged ports (< 1024)	Low	Keep only if binding to ports < 1024
NET_RAW	Use RAW and PACKET sockets	High	Drop unless raw network access required
SETFCAP	Set file capabilities	High	Drop unless capability management needed
SETGID	Make arbitrary manipulations of process GIDs	High	Drop unless group ID changes required
SETPCAP	Modify process capabilities	High	Drop unless capability modification needed
SETUID	Make arbitrary manipulations of process UIDs	High	Drop unless user ID changes required
SYS_CHROOT	Use chroot(2), change root directory	Medium	Drop unless chroot operations required

**Additional Capabilities (Not Granted by Default)

These capabilities are not granted by default but can be added with `--cap-add` if specifically needed1:

Capability	Description	Use Case
NET_ADMIN	Perform various network-related operations	Network interface management
SYS_ADMIN	Perform a range of system administration operations	System administration tasks
SYS_PTRACE	Trace arbitrary processes using ptrace(2)	Debugging and monitoring
SYS_TIME	Set system clock	Time synchronization
IPC_LOCK	Lock memory (mlock, mlockall, mmap, shmctl)	High-performance applications

Capability Management Strategy

1. **Start with ALL dropped**: Use `--cap-drop=ALL` to remove all capabilities
2. **Add only what's needed**: Use `--cap-add` to grant specific required capabilities
3. **Regular audit**: Review and minimize capabilities periodically

High-Risk Capabilities to Always Drop

The following capabilities pose the highest security risks and should be dropped unless absolutely necessary:

- DAC_OVERRIDE: Bypasses all file permission checks
- NET_RAW: Allows raw network packet manipulation
- SETFCAP: Can modify file capabilities
- SETGID/SETUID: Can change user/group identities
- SETPCAP: Can modify process capabilities

**Web Application (Minimal Privileges)

```
docker run -d \
  --cap-drop=ALL \
  --cap-add=NET_BIND_SERVICE \
  web-app:latest
```

9 Multi-Stage Builds for Optimization

Multi-stage builds allow you to use multiple FROM statements in a Dockerfile, with each stage building on the previous one:

```
# Build stage
FROM golang:1.20 AS builder
WORKDIR /app
COPY go.* ./
RUN go mod download
COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o server .

# Security scan stage
FROM aquasec/trivy:latest AS security
COPY --from=builder /app/server /app/server
RUN trivy fs --exit-code 1 --severity HIGH,CRITICAL /app

# Final stage
FROM alpine:3.18
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=builder /app/server .
CMD ["./server"]
```

Benefits of multi-stage builds include:

- **Dramatically smaller images**: Only necessary artifacts are included in the final image
- **Improved security**: Build tools and dependencies are excluded from the production image
- **Better organization**: Each stage has a clear, single responsibility
- **Parallel execution**: Independent stages can be built concurrently

9.2 Health Checks for Robust Applications

Implementing health checks ensures that Docker can detect and respond to application issues:

```
FROM nginx:alpine

# Copy custom configuration
COPY nginx.conf /etc/nginx/nginx.conf
COPY app /usr/share/nginx/html
```

```
# Add health check
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
  CMD wget --quiet --tries=1 --spider http://localhost/ || exit 1

EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Health checks can be used to:

- Automatically restart unhealthy containers
- Prevent traffic to unhealthy containers in swarm mode
- Create dependency chains where services wait for dependencies to be healthy
- Provide visibility into application health status