Docker Workshops

Simon Da Silva

October 19, 2025

Contents

INTRODUCTION TO DOCKER AND INSTALLATION

This document contains Docker training materials organized into three parts:

- **Introduction:** Introduction to Docker concepts and installation instructions
- **Workshop 1:** Docker fundamentals - hands-on exercises with basic commands, networking, and Docker Compose
- **Workshop 2:** Advanced Docker - building a complete video streaming platform with microservices architecture

Prerequisites for all workshops:

- Basic Linux command-line knowledge
- Text editor (vim, nano, VS Code, etc.)
- Terminal access
- Internet connection for downloading images

1 Introduction to Docker

1.1 What is Docker?

Docker is the world's leading software container platform. It allows developers to package applications with all their dependencies into standardized units called containers.
Developers use Docker to eliminate "works on my machine" problems when collaborating on code with co-workers. Operators use Docker to run and manage apps side-by-side in isolated containers to get better compute density. Enterprises use Docker to build agile software delivery pipelines to ship new features faster, more securely and with confidence for both Linux and Windows applications.

1.2 Why Use Docker?

Docker automates the repetitive tasks of setting up and configuring development environments so that developers can focus on what matters: building great software.
Key Benefits:

- **Consistency:** "Works on my machine" problems disappear - if it works in a Docker container on your laptop, it will work the same way in production
- **Isolation:** Applications run in isolated environments, preventing conflicts between dependencies
- **Portability:** Run anywhere - laptop, server, cloud (AWS, Azure, Google Cloud)
- **Efficiency:** Lightweight compared to virtual machines, use less resources, start in seconds
- **Scalability:** Easy to scale applications horizontally by running multiple container instances
- **Version Control:** Images are versioned and can be rolled back easily
- **Microservices:** Perfect for microservices architectures

When an app is dockerized, complexity is pushed into containers that are easily built, shared and run. Onboarding a co-worker to a new codebase no longer means hours spent installing software and explaining setup procedures.

1.3 What is a Container?

Containers are a way to package software in a format that can run isolated on a shared operating system.
Virtual Machines run guest operating systems. This is resource intensive, and the resulting disk image and application state is an entanglement of OS settings, system-installed dependencies, OS security patches, and other easy-to-lose, hard-to-replicate ephemera.
Containers vs Virtual Machines:
Virtual Machines:

- Run full guest operating systems
- Resource intensive (GB of RAM, GB of disk space)
- Minutes to start
- Complete isolation with hypervisor
- Each VM includes a complete OS

Containers:

- Share host OS kernel
- Lightweight (MB of disk space)
- Seconds to start
- Process-level isolation
- Only include application and dependencies

Unlike VMs, containers do not bundle a full operating system. Containers can share a single kernel, and the only information that needs to be in a container image is the executable and its package dependencies. These processes run like native processes. Because containers contain all their dependencies, a containerized app runs anywhere.

1.4 Docker Terminology

Understanding these key terms is essential:

- **Image:** A read-only template with application code, libraries, and dependencies
- **Container:** A running instance of an image - what the image becomes when executed
- **Dockerfile:** A text file containing instructions to build an image
- **Docker Hub:** A public registry for Docker images (like GitHub for containers)
- **Volume:** Persistent data storage for containers
- **Network:** Virtual networks connecting containers
- **Docker Compose:** Tool for defining and running multi-container applications
- **Registry:** Storage and distribution system for Docker images

2 Installation Instructions

2.1 Installing Docker Engine

Docker Engine (Docker CE - Community Edition) is the core Docker runtime that manages containers on your system.
Installation: Follow the official installation guide for your operating system:
`https://docs.docker.com/engine/install/`

2.2 Installing Docker Compose Plugin

Docker Compose is a tool for defining and running multi-container applications. The modern approach is to install it as a Docker plugin (Compose V2).
Important Note: This workshop uses Docker Compose plugin version, which uses the command `docker compose` (with a space). The older standalone version used `docker-compose` (with a hyphen).
Installation: Follow the official installation guide:
`https://docs.docker.com/compose/install/linux/`

3 Verifying Your Installation

```
# Check Docker version
$ docker --version
# Check Docker Compose version
$ docker compose version
# Run a test container
$ docker run hello-world
# List running containers (should be empty)
$ docker ps
# List all containers (including stopped)
$ docker ps -a
# List downloaded images
$ docker images
```

4 Common Installation Issues

4.1 Permission Denied

Error: `Got permission denied while trying to connect to Docker daemon`

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

4.2 Docker Daemon Not Running

Error: `Cannot connect to the Docker daemon`

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

5 Docker Architecture Overview

5.1 Docker Components

Docker uses a client-server architecture:

- **Docker Client:** Commands like `docker run` communicate with the daemon via API
- **Docker Daemon (dockerd):** Manages Docker objects (images, containers, networks, volumes)
- **Docker Registry:** Stores Docker images (Docker Hub is the default public registry)
- **Docker Objects:** Images, containers, networks, and volumes

5.2 How Docker Works

When you run `docker run`:

1. Docker client sends command to daemon
2. Daemon checks if image exists locally
3. If not, pulls image from registry
4. Daemon creates container from image
5. Allocates filesystem and network interface
6. Starts container and executes command
7. Output streams back to client

6 Getting Help and Resources

6.1 Built-in Help

```
# General Docker help
$ docker --help

# Help for specific commands
$ docker run --help
$ docker build --help

# Docker Compose help
$ docker compose --help
```

6.2 Useful Resources

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

WORKSHOP 1: DOCKER FUNDAMENTALS

Workshop Overview

This workshop provides hands-on practice with Docker fundamentals.
Duration: 3-4 hours
Prerequisites: Completed Introduction (Docker installed)
Learning Objectives:

- Master essential Docker commands
- Configure Docker networking
- Create custom images with Dockerfiles
- Deploy multi-container applications with Docker Compose
- Manage data persistence with volumes

7 Basic Docker Commands

7.1 Working with Images

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

# List all local images
$ docker images

# Remove an image
$ docker rmi nginx
```

7.2 Running Containers

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

# Run a container in detached mode
$ docker run -d nginx

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

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

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

7.3 Managing Containers

```
# List running containers
$ docker ps

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

# Stop a container
$ docker stop my-nginx

# Start a stopped container
$ docker start my-nginx

# Restart a container
$ docker restart my-nginx

# Remove a container
$ docker rm my-nginx

# Remove a running container (force)
$ docker rm -f my-nginx

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

# View container logs
$ docker logs my-nginx

# Follow container logs in real-time
$ docker logs -f my-nginx
```

1. Basic Container Operations
   (a) Pull the official `nginx` image from Docker Hub.
   (b) Run an nginx container in detached mode, name it `webserver`, and map port 8080 on your host to port 80 in the container.
   (c) Verify the container is running using `docker ps`.
   (d) Access the nginx welcome page by opening `http://localhost:8080` in your browser.
   (e) View the logs of your container.
   (f) Stop and remove the container.

8 Docker Networking Fundamentals

8.1 Default Bridge Network

By default, Docker creates a bridge network called `bridge` where containers can communicate.

```
# Inspect the default bridge network
$ docker network inspect bridge
```

```
# Check host network interfaces
$ ip addr show docker0
```

1. Default Network Exploration
   (a) What is the IP address of the `docker0` bridge on your host?
   (b) Run two containers: `container1` and `container2` using the `alpine` image with the command `sleep 3600`.
   (c) Find the IP addresses of both containers using `docker inspect`.
   (d) Execute a shell in `container1` and try to ping `container2` by IP address. Does it work?
   (e) Try to ping `container2` by name. Does it work? Why or why not?

8.2 Container Linking (Legacy)

```
# Run container1
$ docker run -d --name container1 alpine sleep 3600

# Run container2 with link to container1
$ docker run -it --name container2 --link container1 alpine sh

# Inside container2, ping container1 by name
$ ping container1
```

1. Container Linking
   (a) After linking, can `container2` ping `container1` by name?
   (b) Inspect `/etc/hosts` inside `container2`. What do you see?
   (c) What are the limitations of the `--link` approach?

8.3 Port Exposure

```
# Run nginx without port mapping
$ docker run -d --name web1 nginx

# Run nginx with random host port
$ docker run -d -p 80 --name web2 nginx

# Run nginx with specific host port
$ docker run -d -p 8080:80 --name web3 nginx

# Check port mappings
$ docker ps
$ docker port web3
```

1. Port Mapping
   (a) Run an nginx container without port mapping. Can you access it from your host browser?

(b) Run an nginx container with `-p 80` (random port mapping). What port is it mapped to on the host?

(c) Run an nginx container with `-p 8080:80`. Verify you can access it.

(d) What Linux networking tool does Docker use to implement port forwarding? (Hint: check `iptables`)

8.4 Custom Bridge Networks

Custom networks provide better isolation and automatic DNS resolution.

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

# Inspect the network
$ docker network inspect my-network

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

# Test connectivity
$ docker exec -it web1 sh
# ping web2 (should work by name!)
```

1. Custom Networks

   (a) Create two custom networks: `network1` and `network2`.

   (b) Run `container1` on `network1` and `container2` on `network2`.

   (c) Can `container1` ping `container2`? Why or why not?

   (d) Connect `container1` to `network2` using `docker network connect`. Can they communicate now?

   (e) What is the advantage of custom networks over the default bridge network?

9 Building Docker Images

9.1 Introduction to Dockerfiles

A Dockerfile is a text file containing instructions to build a Docker image.
Common Dockerfile Instructions:

- `FROM`: Base image to build upon
- `RUN`: Execute commands during image build
- `COPY`: Copy files from host to image
- `WORKDIR`: Set working directory
- `ENV`: Set environment variables
- `EXPOSE`: Document which ports the container listens on
- `CMD`: Default command when container starts (can be overridden)

- `ENTRYPOINT`: Configure container as executable (harder to override, often used with CMD)

CMD vs ENTRYPOINT:

- Use `CMD` for default commands that users might want to override
- Use `ENTRYPOINT` when you always want a specific executable to run
- Can combine both: `ENTRYPOINT` defines the executable, `CMD` provides default arguments
- Example: `ENTRYPOINT ["python"]` with `CMD ["app.py"]` allows running different scripts

9.2 Building Your First Image

Create a directory for your project:

```
$ mkdir my-python-app && cd my-python-app
```

Create a simple Python application (`app.py`):

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return "Hello from Docker!\n"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Create a requirements file (`requirements.txt`):

```
Flask==3.0.0
Werkzeug==3.0.1
```

Create a Dockerfile:

```
FROM python:3.9-slim

WORKDIR /app

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

COPY app.py .

EXPOSE 5000

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

Note about EXPOSE: The `EXPOSE` instruction is documentation only - it tells users which port the application uses but does not actually publish the port. You must use `-p` or `-P` flags with `docker run` to publish ports. Think of `EXPOSE` as a label, not a network configuration.
Build and run the image:

```
$ docker build -t my-python-app .
$ docker run -d -p 5000:5000 --name python-web my-python-app
$ curl http://localhost:5000
```

1. Building Custom Images
   (a) Explain the purpose of each instruction in the Dockerfile above.
   (b) Why do we `COPY requirements.txt` before `COPY app.py`?
   (c) Build the image and verify it works correctly.
   (d) Modify `app.py` to return a different message, rebuild, and verify the change.
   (e) Use `docker history my-python-app` to see the image layers. What do you observe?

9.3 Multi-Stage Builds

Multi-stage builds help create smaller production images by separating the build environment from the runtime environment. This is especially useful for compiled languages.
Benefits:

- Smaller final image size (only runtime dependencies included)
- More secure (no build tools in production image)
- Faster deployment and less attack surface

Create a new directory for this exercise:

```
$ mkdir go-app && cd go-app
```

Create a simple Go application (`main.go`):

```
package main

import (
  "fmt"
  "net/http"
)

func main() {
  http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello from Go!")
  })

  fmt.Println("Server starting on :8080")
  http.ListenAndServe(":8080", nil)
}
```

Create a multi-stage Dockerfile:

```
# Build stage
FROM golang:1.20 AS builder
WORKDIR /app
COPY . .
RUN go build -o myapp
```

```
# Runtime stage
FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /app
COPY --from=builder /app/myapp .
EXPOSE 8080
CMD ["./myapp"]
```

Build and run the application:

```
# Build the image
$ docker build -t go-multistage .

# Compare image sizes
$ docker images | grep -E "golang|go-multistage"

# Run the container
$ docker run -d -p 8080:8080 --name go-app go-multistage

# Test the application
$ curl http://localhost:8080

# Clean up
$ docker stop go-app && docker rm go-app
```

1. Multi-Stage Build Exercise
   (a) Build the Go application using the multi-stage Dockerfile.
   (b) Compare the size of `golang:1.20` base image with your final `go-multistage` image. What is the size difference?
   (c) Run the application and verify it works correctly.
   (d) Try building without the multi-stage approach (use only the golang image). How much larger is it?
   (e) What are the security benefits of multi-stage builds?

10 Docker Compose for Multi-Container Applications

10.1 Introduction to Docker Compose

Docker Compose is a tool for defining and running multi-container applications using a YAML file.

Benefits:

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

10.2 Basic Docker Compose Syntax

Example `docker-compose.yml`:

```
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    volumes:
      - ./html:/usr/share/nginx/html

  db:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: secretpass
      POSTGRES_DB: mydb
    volumes:
      - db-data:/var/lib/postgresql/data

volumes:
  db-data:

networks:
  app-network:
```

Common Docker Compose Commands:

```
# Start all services in foreground
$ docker compose up

# Stop and remove all services
$ docker compose down

# View logs
$ docker compose logs

# View logs for specific service
$ docker compose logs web

# List running services
$ docker compose ps

# Execute command in service
$ docker compose exec web bash

# Rebuild images
$ docker compose build

# Restart specific service
$ docker compose restart web
```

11 Hands-on Project: Blog Application

Now let's build a complete blog application with a web frontend and database backend.

11.1 Project Architecture

We'll create a simple blog with:

- **Frontend:** Node.js web application
- **Database:** PostgreSQL for data storage
- **Cache:** Redis for session management

11.2 Project Setup

Create project structure:

```
$ mkdir blog-app && cd blog-app
$ mkdir frontend
```

Create `frontend/package.json`:

```
{
  "name": "blog-frontend",
  "version": "1.0.0",
  "main": "server.js",
  "dependencies": {
    "express": "^4.18.0",
    "pg": "^8.11.0",
    "redis": "^4.6.0"
  }
}
```

Note on Dependencies: In production, you should pin exact versions (without `^`) or use a lockfile (`package-lock.json`) to ensure reproducible builds.
Create `frontend/server.js`:

```
const express = require('express');
const { Client } = require('pg');
const redis = require('redis');

const app = express();
const port = 3000;

// Database connection
const dbClient = new Client({
  host: process.env.DB_HOST || 'database',
  port: 5432,
  user: 'bloguser',
  password: 'blogpass',
  database: 'blogdb'
});

// Redis connection
```

```
const redisClient = redis.createClient({
  url: `redis://${process.env.REDIS_HOST || 'cache'}:6379`
});

app.get('/', (req, res) => {
  res.send(`
    <h1>Welcome to the Blog App!</h1>
    <p>Database: ${dbClient.database}</p>
    <p>Running in Docker!</p>
  `);
});

app.get('/health', async (req, res) => {
  try {
    await dbClient.query('SELECT 1');
    await redisClient.ping();
    res.json({ status: 'healthy', db: 'connected', cache: 'connected' });
  } catch (err) {
    res.status(500).json({ status: 'unhealthy', error: err.message });
  }
});

async function start() {
  try {
    await dbClient.connect();
    await redisClient.connect();

    // Create posts table if not exists
    await dbClient.query(`
      CREATE TABLE IF NOT EXISTS posts (
        id SERIAL PRIMARY KEY,
        title VARCHAR(255),
        content TEXT,
        created_at TIMESTAMP DEFAULT NOW()
      )
    `);

    app.listen(port, () => {
      console.log(`Blog app listening on port ${port}`);
    });
  } catch (error) {
    console.error('Failed to start application:', error);
    process.exit(1);
  }
}

start();
```

Create `frontend/Dockerfile`:

```
FROM node:18-alpine
```

```
WORKDIR /app

COPY package*.json ./
RUN npm install

COPY server.js ./

EXPOSE 3000

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

Create `docker-compose.yml` in the root directory:
Security Note: The following example includes hardcoded passwords for simplicity. In production, always use environment files (`.env`) or Docker secrets to manage sensitive credentials.

```
services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
      - DB_HOST=database
      - REDIS_HOST=cache
    depends_on:
      - database
      - cache
    restart: unless-stopped

  database:
    image: postgres:14-alpine
    environment:
      POSTGRES_USER: bloguser
      POSTGRES_PASSWORD: blogpass
      POSTGRES_DB: blogdb
    volumes:
      - postgres-data:/var/lib/postgresql/data
    restart: unless-stopped

  cache:
    image: redis:7-alpine
    restart: unless-stopped

volumes:
  postgres-data:

networks:
  blog-network:
```

Note on Service Scaling: The frontend service cannot be scaled to multiple replicas as configured above because all replicas would try to bind to the same host port (3000), causing a

conflict. To enable scaling, you would need to either remove the port mapping (and add a load balancer like Nginx), or use random port assignment with `- "3000"` instead of `- "3000:3000"`.

11.3 Running the Application

```
# Build and start all services
$ docker compose up -d

# Check running services
$ docker compose ps

# View logs
$ docker compose logs -f frontend

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

# Stop the application
$ docker compose down
```

1. Blog Application Exercise
   (a) Deploy the blog application using Docker Compose.
   (b) Verify all three services are running with `docker compose ps`.
   (c) Access the application at `http://localhost:3000` and the health endpoint.
   (d) Check the logs of the database service. What do you see?
   (e) Stop the `cache` service using `docker compose stop cache`. What happens to the application?
   (f) Restart the cache service and verify the application works again.
   (g) Try to scale the frontend to 3 replicas using `docker compose up -d --scale frontend=3`. What error do you get? Why does this fail? (Hint: think about port conflicts)
   (h) Clean up everything with `docker compose down -v` (including volumes).

12 Data Persistence with Volumes

12.1 Understanding Volumes

Containers are ephemeral - data is lost when they're removed. Volumes provide persistent storage.

Types of mounts:

- **Named volumes:** Managed by Docker (recommended for production)
- **Bind mounts:** Mount host directory into container (useful for development)
- **tmpfs mounts:** Temporary in-memory storage (data lost on container stop)

Volume Permissions:
A common issue with volumes is file permission mismatches. By default, processes in containers may run as root or with a specific UID that doesn't match your host user.

- **Named volumes:** Docker manages permissions automatically
- **Bind mounts:** Files created in container may be owned by root on host
- **Solution:** Run container processes as non-root user with matching UID

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

# List volumes
$ docker volume ls

# Inspect volume
$ docker volume inspect my-data

# Use volume with container
$ docker run -d -v my-data:/data alpine sleep 3600

# Use bind mount
$ docker run -d -v $(pwd):/app alpine sleep 3600

# Remove unused volumes
$ docker volume prune

# Remove specific volume
$ docker volume rm my-data
```

1. Volume Persistence
   (a) Run a PostgreSQL container with a named volume for data persistence.
   (b) Connect to the database and create a test table with some data.
   (c) Stop and remove the container.
   (d) Start a new PostgreSQL container using the same volume.
   (e) Verify that your data still exists.
   (f) What happens if you use a bind mount instead? What are the pros and cons?

13 Best Practices and Tips

13.1 Dockerfile Best Practices

- Use official base images
- Use specific image tags, not `latest`
- Minimize number of layers
- Use `.dockerignore` to exclude unnecessary files
- Don't run containers as root (create and use a non-root user)
- Use multi-stage builds for smaller images

- Group `RUN` commands to reduce layers
- Pin dependency versions for reproducibility
- Use `EXPOSE` to document ports (note: this doesn't publish ports, only `-p` does)

13.2 Using .dockerignore

The `.dockerignore` file works like `.gitignore` but for Docker builds. It excludes files from the build context, making builds faster and images smaller.
Benefits:

- Faster builds (less data to transfer)
- Smaller images (fewer unnecessary files)
- Better security (exclude sensitive files)
- Avoid cache invalidation from irrelevant file changes

Example .dockerignore file:

```
# Version control
.git
.gitignore

# Python
__pycache__
*.pyc
*.pyo
*.pyd
.Python
*.so
*.egg
*.egg-info
dist
build
.pytest_cache
.coverage

# Environment
.env
.venv
venv/
ENV/

# IDE
.vscode
.idea
*.swp
*.swo

# OS
.DS_Store
```

```
Thumbs.db

# Documentation
*.md
!README.md

# Logs
*.log

# Dependencies (for Node.js)
node_modules
npm-debug.log
```

13.3 Docker Compose Best Practices

- Use `depends_on` with health checks to define service dependencies properly
- Define named volumes for data persistence
- Use environment variables for configuration
- Set restart policies for production deployments (`restart: unless-stopped`)
- Use networks to isolate services (Docker creates a default network, but you can define custom ones)
- Keep secrets in environment files, not in docker-compose.yml
- Use specific image tags instead of `latest`
- Add health checks for critical services
- Never commit `.env` files with secrets to version control

Restart Policy Options:

- `no`: Never restart (default)
- `always`: Always restart if container stops
- `on-failure`: Restart only if container exits with error
- `unless-stopped`: Always restart unless explicitly stopped by user

Network Driver Options:

- `bridge`: Default driver for standalone containers on same host
- `host`: Container uses host's network directly (no isolation)
- `none`: No networking
- `overlay`: For multi-host networking in Docker Swarm

Using Environment Files:
Instead of hardcoding credentials in `docker-compose.yml`, use a `.env` file:

```
# .env file (never commit this!)
POSTGRES_USER=bloguser
POSTGRES_PASSWORD=secure_random_password_here
POSTGRES_DB=blogdb
REDIS_PASSWORD=another_secure_password
```

```
# docker-compose.yml
services:
  database:
    image: postgres:14-alpine
    env_file:
      - .env
    # Or reference specific variables:
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
```

Create a `.env.example` file with dummy values to commit to version control, showing which variables are needed.

13.4 Common Troubleshooting

Quick Troubleshooting Checklist:

- Is Docker daemon running? (`systemctl status docker`)
- Are ports already in use? (`netstat -tulpn | grep PORT`)
- Do you have permission to access Docker socket? (`groups | grep docker`)
- Are health checks passing? (`docker compose ps`)
- Are there any error messages in logs? (`docker compose logs`)
- Is there enough disk space? (`df -h`)

Common Commands:

```
# View container resource usage
$ docker stats

# Inspect container details
$ docker inspect <container>

# Check container processes
$ docker top <container>

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

# Clean up everything (WARNING: removes all unused resources)
$ docker system prune -a --volumes
```

```
# View Docker disk usage
$ docker system df -v
```

14 Workshop 1 Cleanup

Before moving on to Workshop 2, clean up the resources created in this workshop:

```
# Stop and remove all containers from the app
$ docker compose down -v

# Remove custom images
$ docker rmi my-python-app go-multistage

# Remove custom networks (if created)
$ docker network rm my-network network1 network2 2>/dev/null || true

# Clean up all unused resources (optional - removes everything not in use)
$ docker system prune -a --volumes

# For a gentler cleanup, remove only dangling images and stopped containers:
$ docker system prune
```

15 Workshop 1 Conclusion

Congratulations! You've completed the beginner Docker workshop. You should now be able to:

- Understand Docker fundamentals and terminology
- Work with Docker images and containers
- Configure Docker networking
- Build custom Docker images with Dockerfiles
- Deploy multi-container applications with Docker Compose
- Manage data persistence with volumes
- Troubleshoot common Docker issues
- Apply Docker best practices

15.1 Quick Reference: Essential Commands

Images:

```
docker images # List images
docker pull <image> # Download image
docker build -t <name> . # Build image
docker rmi <image> # Remove image
```

Containers:

```
docker ps # List running containers
docker ps -a # List all containers
docker run <image> # Run container
docker stop <container> # Stop container
docker rm <container> # Remove container
docker logs <container> # View logs
docker exec -it <c> bash # Execute command
```

Docker Compose:

```
docker compose up # Start services
docker compose up -d # Start in background
docker compose down # Stop and remove
docker compose down -v # Also remove volumes
docker compose logs -f # Follow logs
docker compose ps # List services
docker compose build # Rebuild images
```

Cleanup:

```
docker system prune # Remove unused resources
docker system prune -a # Remove all unused
docker volume prune # Remove unused volumes
```

Next Steps:

- Explore Docker Hub for useful images
- Practice building your own applications
- Learn about Docker security best practices
- Move on to Workshop 2 for advanced topics!

WORKSHOP 2: VIDEO STREAMING APPLICATION

Workshop Overview

In this advanced workshop, you'll build and containerize a complete video streaming application that demonstrates real-world microservices architecture with Docker.
Duration: 4 hours
Prerequisites: Completed Workshop 1 or equivalent Docker knowledge
Learning Objectives:

- Build a multi-service application with microservices architecture
- Create production-ready Dockerfiles for different service types
- Implement background job processing with worker containers
- Configure service dependencies and health checks
- Manage shared volumes between multiple services
- Scale services horizontally
- Apply Docker best practices in a real-world scenario

16 Application Overview

16.1 What You'll Build

You will containerize a video streaming platform that:

- Accepts MP4 video uploads from users
- Transcodes videos to DASH format using FFmpeg
- Processes videos in the background using worker containers
- Stores video metadata in a PostgreSQL database
- Uses Redis as a task queue for job distribution
- Serves transcoded video segments via a web interface
- Supports horizontal scaling of workers

16.2 Application Architecture

The application follows a microservices architecture:
Component Descriptions:

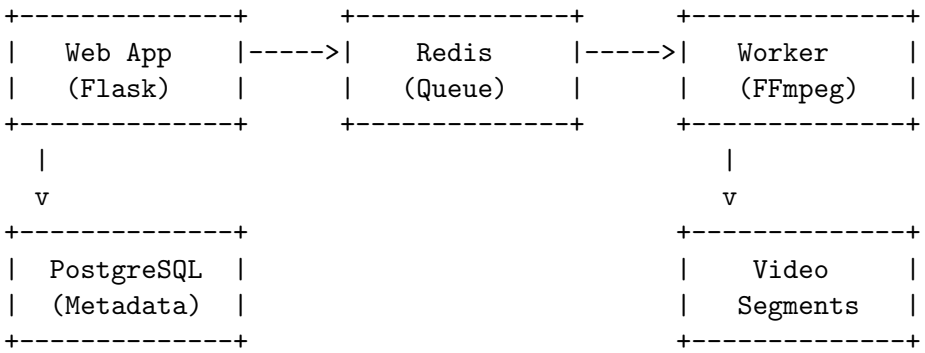
- **Web Application (Flask):** Frontend that handles video uploads, displays job status, and streams videos
- **PostgreSQL Database:** Stores video metadata, upload information, and processing status
- **Redis (Message Queue):** Connects web app and workers for asynchronous job processing

- **Worker Service:** Background workers that transcode videos using FFmpeg
- **Shared Volume:** Storage for uploaded videos and transcoded segments, accessible by both web and worker services

Critical Redis Limitation:
WARNING: Default Redis configuration is **not persistent** - all queued jobs will be lost if Redis restarts! For production deployments, you must either:

- Enable Redis persistence (RDB snapshots or AOF logging)
- Use a dedicated persistent queue like RabbitMQ or AWS SQS
- Store job state in PostgreSQL alongside the queue

This workshop uses default Redis for simplicity, but be aware of this limitation.



16.3 Project Structure

The workspace is organized as follows:

```
starter/
+-- docker-compose.yml          # To be created
+-- web/
|   +-- app.py                  # Flask web application
|   +-- Dockerfile              # To be created
|   +-- requirements.txt        # Python dependencies
|   +-- templates/
|       +-- index.html          # Upload interface
|       +-- player.html         # Video player
+-- worker/
  +-- worker.py               # Background worker
  +-- Dockerfile              # To be created
  +-- requirements.txt        # Worker dependencies
```

17 Understanding the Application Code

Before containerizing, let's understand what each component does.

17.1 Web Application (web/app.py)

The Flask web application provides:

- `/` - Upload page for MP4 videos
- `/upload` - Endpoint that accepts video uploads and queues transcoding jobs
- `/status/<video_id>` - Check processing status of a video
- `/watch/<video_id>` - Video player page
- `/videos/<path>` - Serves video segments and manifests

Key Features:

- Connects to PostgreSQL to store video metadata
- Pushes transcoding jobs to Redis queue
- Serves DASH video segments
- Initializes database schema on startup

17.2 Worker Service (worker/worker.py)

The worker process:

- Listens to Redis queue for transcoding jobs
- Uses FFmpeg to convert MP4 to DASH format
- Updates job status in PostgreSQL database
- Can run multiple instances for parallel processing

FFmpeg Command: The worker uses FFmpeg to create DASH streams with multiple quality levels for adaptive streaming.

17.3 Dependencies

Web Application (web/requirements.txt):

- Flask - Web framework
- psycopg2-binary - PostgreSQL driver
- redis - Redis client

Note: These dependencies should be pinned to specific versions in the actual `requirements.txt` file for reproducible builds. Example: `Flask==3.0.0`.

Worker (worker/requirements.txt):

- psycopg2-binary - PostgreSQL driver
- redis - Redis client
- FFmpeg - Must be installed in container (not a Python package)

18 Task 1: Create Dockerfile for Web Application

18.1 Requirements

Create `starter/web/Dockerfile` that:

- Uses `python:3.11-slim` as base image
- Sets working directory to `/app`
- Copies and installs dependencies from `requirements.txt`
- Copies all application code
- Creates directories for video storage: `/app/videos/uploads` and `/app/videos/output`
- Exposes port 5000
- Runs the Flask application

18.2 Hints and Best Practices

- Copy `requirements.txt` first and run `pip install` before copying application code (layer caching)
- Use `pip install --no-cache-dir` to reduce image size
- Use `mkdir -p` to create nested directories
- Flask should run with host `0.0.0.0` to accept external connections
- Consider using `CMD` with JSON array syntax for the run command
- **Note:** For a production deployment, use a WSGI server like gunicorn instead of Flask's development server. This workshop uses Flask's built-in server for simplicity, but you can add gunicorn as a bonus improvement.

1. Web Application Dockerfile
   (a) Create the `starter/web/Dockerfile` following the requirements above.
   (b) Build the image with: `docker build -t video-web ./starter/web`
   (c) Inspect the image layers with: `docker history video-web`
   (d) How many layers does your image have? Which layer is the largest?

19 Task 2: Create Dockerfile for Worker

19.1 Requirements

Create `starter/worker/Dockerfile` that:

- Uses `python:3.11-slim` as base image
- Installs FFmpeg (required for video transcoding)
- Sets working directory to `/app`
- Copies and installs Python dependencies
- Copies worker script
- Runs the worker process

19.2 Installing FFmpeg

FFmpeg is not available in the slim Python image. You need to:

1. Update apt package lists: `apt-get update`
2. Install FFmpeg: `apt-get install -y ffmpeg`

Important: Chain these commands in a single `RUN` instruction to minimize layers.

1. Worker Dockerfile
   (a) Create the `starter/worker/Dockerfile` following the requirements above.
   (b) Build the image with: `docker build -t video-worker ./starter/worker`
   (c) Compare the image sizes: `docker images | grep video`
   (d) Why is the worker image larger than the web image?

20 Task 3: Create Docker Compose Configuration

20.1 Requirements

Create `starter/docker-compose.yml` that defines four services:

1. Database Service (db):

- Image: `postgres:15-alpine`
- Environment variables: `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD`
- Named volume for data persistence: `postgres-data:/var/lib/postgresql/data`
- Health check to ensure database is ready before dependent services start

2. Redis Service:

- Image: `redis:7-alpine`
- Health check to ensure Redis is ready

3. Web Service:

- Build from `./web` directory
- Port mapping: `5000:5000`
- Environment variables: `DATABASE_URL`, `REDIS_URL`
- Named volume for video storage: `video-data:/app/videos`
- Depends on: `db`, `redis`

4. Worker Service:

- Build from `./worker` directory
- Same environment variables as web service
- Same video volume as web service (shared storage)
- Depends on: `db`, `redis`
- Can be scaled using: `docker compose up -d --scale worker=2`

Note on Scaling: Docker Compose v2 uses the `--scale` flag to run multiple replicas. The `deploy.replicas` syntax only works in Docker Swarm mode, which is Docker's orchestration platform for production clusters.

20.2 Environment Variables Format

```
DATABASE_URL=postgresql://streamuser:streampass@db:5432/videostream
REDIS_URL=redis://redis:6379/0
```

Important Notes:

- Use service names (`db`, `redis`) as hostnames
- These names are resolved via Docker's internal DNS
- Services on the same network can communicate using service names

20.3 Health Checks

Health checks ensure services are ready before dependent services start.
PostgreSQL Health Check:

```
healthcheck:
  test: ["CMD-SHELL", "pg_isready -U streamuser -d videostream"]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 30s
```

Redis Health Check:

```
healthcheck:
  test: ["CMD", "redis-cli", "ping"]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 5s
```

Health Check Parameters:

- `test`: Command to run to check health
- `interval`: Time between health checks (runs every X seconds)
- `timeout`: Maximum time for a single check (fails if exceeds this)
- `retries`: Consecutive failures before marking unhealthy
- `start_period`: Grace period during container initialization. Health check failures during this time don't count toward the retry limit, allowing the service time to start up without being marked unhealthy immediately. Essential for services like databases that take time to initialize.

Why Health Checks Matter:
Without health checks, `depends_on` only waits for the container to start, not for the service inside to be ready. For example, PostgreSQL container might start, but the database might not be ready to accept connections for several seconds. Health checks ensure dependent services only start when their dependencies are truly ready.

1. Docker Compose Configuration

(a) Create the `starter/docker-compose.yml` file following the structure above.

(b) Why do we use `depends_on` with `condition: service_healthy`?

(c) What happens if you remove the health check conditions? (Hint: try it and observe the errors)

(d) Why do the web and worker services share the same `video-data` volume?

(e) Create `.dockerignore` files in both `web/` and `worker/` directories using the example provided earlier.

21 Testing Your Solution

21.1 Building and Starting the Application

Navigate to the starter directory and start all services:

```
# Navigate to starter directory
$ cd starter

# Build and start all services
$ docker compose up --build
```

What to Expect:

- Docker builds images for web and worker services
- PostgreSQL and Redis start first (with health checks)
- Web and worker services wait for healthy dependencies
- Web service initializes database schema
- Workers connect to Redis and wait for jobs

Important Notes:

- The `--build` flag forces Docker to rebuild images even if they exist. Use this when you've made changes to Dockerfiles or application code.
- If you don't use `--build`, Docker will use cached images, which may not include your latest changes.
- Database initialization happens when the web service starts. If you scale the web service later, ensure only one instance initializes the schema to avoid race conditions.
- First startup may take 30-60 seconds for health checks to pass and services to initialize.

Expected Startup Sequence:

1. PostgreSQL container starts, health check waits for database to be ready (10-30 seconds)
2. Redis container starts, health check confirms it's ready (5-10 seconds)
3. Web service starts after db and redis are healthy, initializes database schema
4. Worker service starts and connects to Redis queue
5. All services show as "healthy" in `docker compose ps`

21.2 Verifying Services are Running

```
# Check running services
$ docker compose ps

# Expected output: 4 services running (db, redis, web, worker)

# Follow logs in real-time
$ docker compose logs -f
```

21.3 Testing Video Upload and Transcoding

Step 1: Access the Web Interface
Open your browser to `http://localhost:5000`
You should see the video upload interface.
Step 2: Prepare a Test Video
For testing, use a small MP4 video file (under 50MB recommended). You can:

- Use an existing video file
- Download a sample video from the internet
- Create a test video using FFmpeg:

```
# Note: This requires FFmpeg on your HOST machine, not in containers
$ ffmpeg -f lavfi -i testsrc=duration=10:size=1280x720:rate=30 \
        -pix_fmt yuv420p test-video.mp4
```

Alternative: If you don't have FFmpeg installed locally, just use any small MP4 video file you have available.
Step 3: Upload the Video

1. Click "Choose File" and select your MP4 video
2. Click "Upload"
3. The application will save the file and queue a transcoding job
4. You'll be redirected to the status page

Step 4: Monitor Processing Status
Watch the status change through these stages:

- `queued` - Job is in Redis queue
- `processing` - Worker is transcoding the video
- `completed` - Transcoding finished successfully
- `failed` - If an error occurred

Step 5: Watch the Transcoded Video
Once status shows `completed`:

1. Click the "Watch" button
2. The video player will load
3. Video plays in DASH format with adaptive streaming

21.4 Verifying Worker Processing

```
# Watch worker logs in real-time
$ docker compose logs -f worker

# You should see:
# - Worker connecting to Redis
# - Picking up the transcoding job
# - FFmpeg processing output
# - Job completion message
```

Processing Time: Video transcoding can take anywhere from a few seconds to several minutes depending on:

- Video length and resolution
- CPU resources available to the worker container
- Number of quality levels being generated (currently 2: 360p and 720p)
- FFmpeg preset (currently using "fast")

Don't be alarmed if transcoding takes several minutes for larger videos. This is normal!

21.5 Testing with cURL (Alternative)

```
# Upload a video
$ curl -F "file=@test-video.mp4" http://localhost:5000/upload

# Check status (replace VIDEO_ID with actual ID from upload response)
$ curl http://localhost:5000/status/VIDEO_ID

# Expected response:
# {"status": "queued"} -> {"status": "processing"} -> {"status": "completed"}
```

1. Application Testing
   (a) Upload a test video and verify it processes successfully.
   (b) Monitor the worker logs. Which worker instance processed your video?
   (c) Upload two videos simultaneously. How are they distributed among workers?
   (d) Check the `video-data` volume contents. What files were created?

22 Understanding Service Communication

22.1 Network Architecture

Docker Compose automatically creates a default network where all services can communicate.

```
# List networks
$ docker network ls

# Inspect the network (replace PROJECT with your directory name)
```

```
$ docker network inspect starter_default

# Examine network configuration
# Note the IP addresses assigned to each service
```

Note: Docker Compose creates a network named `<directory>_default` by default. In the solution, we explicitly define a network named `videostream-network` for better organization.

22.2 Service Discovery via DNS

Services communicate using service names as hostnames:

```
# Execute command in web container
$ docker compose exec web bash

# Inside the container, test connectivity
$ ping db
$ ping redis
$ nslookup redis

# Test Redis connection
$ apt-get update && apt-get install -y redis-tools
$ redis-cli -h redis ping

# Test PostgreSQL connection
$ apt-get install -y postgresql-client
$ psql $DATABASE_URL -c "SELECT * FROM videos;"
```

1. Service Communication
   (a) From the web container, ping the database service. What IP address does it resolve to?
   (b) Connect to PostgreSQL from the web container and query the videos table.
   (c) What would happen if you changed the service name from `db` to `database`?
   (d) How does Docker implement this internal DNS resolution?

23 Scaling Workers

One of the key benefits of containerization is easy horizontal scaling.

23.1 Scaling Worker Replicas

```
# Scale to 3 worker instances
$ docker compose up -d --scale worker=3

# Check running services
$ docker compose ps

# You should see 3 worker containers

# View all worker logs
```

```
$ docker compose logs worker
```

23.2 Testing Load Distribution

```
# Upload multiple videos quickly
$ for i in {1..5}; do
    curl -F "file=@test-video.mp4" http://localhost:5000/upload
done

# Watch worker logs to see job distribution
$ docker compose logs -f worker

# You should see different workers processing different jobs
```

1. Worker Scaling
   (a) Scale workers to 4 instances. Upload multiple videos and observe the distribution.
   (b) What happens if a worker crashes while processing? (Use `docker compose stop worker-1`)
   (c) Scale down to 1 worker. How does performance change?
   (d) What are the limitations of scaling workers? What resources might become bottlenecks?

24 Data Persistence and Volumes

24.1 Understanding Named Volumes

The application uses two named volumes:

```
# List volumes
$ docker volume ls

# Inspect volume details
$ docker volume inspect starter_postgres-data
$ docker volume inspect starter_video-data

# Check volume contents
$ docker compose exec web ls -lah /app/videos/uploads
$ docker compose exec web ls -lah /app/videos/output
```

24.2 Testing Data Persistence

```
# Upload a video and let it complete processing

# Stop and remove all containers
$ docker compose down

# Restart the application
$ docker compose up -d
```

```
# Check if videos are still accessible
$ curl http://localhost:5000/

# Database records and video files should persist
```

24.3 Volume Cleanup

```
# Stop and remove containers and volumes
$ docker compose down -v

# This removes:
# - All containers
# - The network
# - Named volumes (postgres-data, video-data)

# Warning: This deletes all uploaded videos and database data!
```

1. Volume Management
   (a) Upload a video, then stop and restart the application. Is the video still available?
   (b) What is stored in the `postgres-data` volume? Why is it important?
   (c) Compare the size of the two volumes using `docker system df -v`.
   (d) What would happen if you deleted only the `video-data` volume?

25 Troubleshooting Common Issues

25.1 Connection Errors

Symptom: Web or worker services can't connect to database or Redis.
Possible Causes:

- Services not on the same network
- Database not ready (health check failing)
- Incorrect environment variable format
- Service name mismatch in DATABASE_URL or REDIS_URL

Solutions:

```
# Check service health
$ docker compose ps

# View database logs
$ docker compose logs db

# Test connection manually
$ docker compose exec web python -c "import psycopg2; \
  conn = psycopg2.connect('$DATABASE_URL'); print('OK')"
```

25.2 Worker Not Processing Jobs

Symptom: Videos stay in "queued" status forever.
Possible Causes:

- Worker crashed or not running
- Redis connection failed
- FFmpeg not installed in worker container
- Shared volume not mounted correctly

Solutions:

```
# Check worker logs
$ docker compose logs worker

# Verify worker is running
$ docker compose ps worker

# Test Redis connection from worker
$ docker compose exec worker python -c "import redis; \
  r = redis.from_url('$REDIS_URL'); print(r.ping())"

# Check FFmpeg installation
$ docker compose exec worker ffmpeg -version
```

25.3 FFmpeg Transcoding Errors

Symptom: Job status changes to "failed", worker logs show FFmpeg errors.
Possible Causes:

- Invalid or corrupted input video
- Insufficient disk space
- FFmpeg not installed
- Permission issues with video directories

Solutions:

```
# Check available disk space
$ df -h

# Verify video file exists and is readable
$ docker compose exec worker ls -lh /app/videos/uploads/

# Test FFmpeg manually
$ docker compose exec worker ffmpeg -i /app/videos/uploads/VIDEO.mp4 \
  -f null -
```

25.4 Database Connection Issues

Symptom: "Could not connect to database" errors.
Possible Causes:

- PostgreSQL container not healthy
- Incorrect DATABASE_URL format
- Database credentials mismatch
- Database initialization failed

Solutions:

```
# Check database health
$ docker compose exec db pg_isready -U streamuser -d videostream

# Connect to database manually
$ docker compose exec db psql -U streamuser -d videostream

# Inside psql:
videostream=# \dt
videostream=# SELECT * FROM videos;
```

1. Troubleshooting Exercise
   (a) Intentionally break the DATABASE_URL and observe the errors. Fix it.
   (b) Stop the Redis service. What happens when you try to upload a video?
   (c) Remove FFmpeg from the worker Dockerfile and rebuild. What error do you see?
   (d) Set worker replicas to 0. Upload a video. What happens?

26 Bonus Challenges

26.1 Challenge 1: Add Health Checks to Application Services

Add custom health checks to web and worker services in docker-compose.yml.
Hints:

- Web: Check if Flask is responding on port 5000
- Worker: Check if worker process is running and Redis is accessible
- Use `curl` or Python commands in health check

26.2 Challenge 2: Optimize Docker Images

Reduce image sizes using best practices:

- Use multi-stage builds
- Use even smaller base images (e.g., Alpine-based Python)
- Minimize installed packages
- Combine RUN commands

- Remove unnecessary files
- Add non-root user for security
- Use `.dockerignore` to exclude build context bloat

Goal: Reduce total image size by at least 30%.
Security Bonus: Implement non-root user in both web and worker containers:

```
# Add before CMD in Dockerfile
RUN useradd -m -u 1000 appuser && \
    chown -R appuser:appuser /app
USER appuser
```

26.3 Challenge 3: Add Nginx Reverse Proxy

Add an Nginx service that:

- Acts as a reverse proxy for the web service
- Serves static video files directly (bypassing Flask)
- Adds caching headers for video segments
- Handles SSL termination (optional)

26.4 Challenge 4: Add Resource Limits

Configure resource limits for each service:
Note: The `deploy.resources` syntax only works in Docker Swarm mode. For Docker Compose, use the following syntax:

```
services:
  worker:
    # ... other configuration ...
    mem_limit: 512m
    mem_reservation: 256m
    cpus: 0.5
```

Test how the application behaves under resource constraints. Monitor with `docker stats` to see actual resource usage.

26.5 Challenge 5: Implement Graceful Shutdown

Modify worker code to:

- Handle SIGTERM signal
- Finish current job before exiting
- Re-queue unfinished jobs
- Update job status appropriately

26.6 Challenge 6: Add Monitoring

Integrate Prometheus and Grafana:

- Export metrics from Flask app (processing time, queue length, etc.)
- Collect container metrics with cAdvisor
- Create Grafana dashboard for visualization
- Set up alerts for queue backlog

26.7 Challenge 7: Production Deployment

Prepare the application for production:

- Use environment variable files (.env)
- Implement secret management (Docker secrets or external vault)
- Replace Flask development server with Gunicorn WSGI server
- Add logging to external system (ELK stack or Loki)
- Configure backup strategy for volumes
- Set up automatic container restart policies
- Implement blue-green deployment strategy
- Enable Redis persistence (RDB snapshots or AOF)
- Add database migration tool (like Alembic) to handle schema changes
- Implement proper error handling and job retry logic in workers

26.8 Challenge 8: Handle Edge Cases

Improve application robustness:

- What happens if a worker crashes during transcoding? Implement job recovery.
- What if the uploaded file is corrupted? Add validation before queuing.
- What if disk space runs out? Add checks and graceful degradation.
- What if multiple web instances scale up simultaneously? Prevent concurrent schema initialization.
- What if Redis loses all data? Add job status persistence in PostgreSQL.

1. Bonus Challenges
   (a) Complete at least two bonus challenges from the list above.
   (b) Document your approach and any difficulties encountered.
   (c) Measure the improvements (if applicable) - image size, performance, etc.
   (d) What other improvements would you suggest for a production deployment?

27 Best Practices Demonstrated

27.1 Dockerfile Best Practices

This project demonstrates:

- **Layer caching:** Copy requirements.txt before application code
- **Image size optimization:** Use slim base images, clean up package managers
- **Single responsibility:** Separate web and worker images
- **Explicit instructions:** Use WORKDIR, EXPOSE, CMD appropriately
- **Dependency pinning:** Pin versions for reproducible builds (should be improved)
- **Security considerations:** Non-root users should be added (see bonus challenges)

Areas for Improvement:

- Add non-root user for both services
- Use production WSGI server (Gunicorn) instead of Flask development server
- Add `.dockerignore` files
- Consider multi-stage builds for even smaller images

27.2 Docker Compose Best Practices

- **Service dependencies:** Use depends_on with health checks
- **Environment variables:** Centralized configuration
- **Named volumes:** Data persistence and sharing
- **Health checks:** Ensure services are ready
- **Scalability:** Worker replicas for horizontal scaling
- **Separation of concerns:** Each service has a single responsibility

27.3 Microservices Architecture

- **Loose coupling:** Services communicate via Redis and database
- **Independent scaling:** Scale workers without affecting web service
- **Fault isolation:** Worker failure doesn't crash web service
- **Technology flexibility:** Could replace worker with different language/tool
- **Asynchronous processing:** Long-running tasks don't block web requests

Limitations to Be Aware Of:

- **Redis volatility:** Default configuration loses queued jobs on restart
- **Database race conditions:** Multiple web instances can cause initialization conflicts

- **No job recovery:** Worker crashes lose in-progress jobs
- **Development server:** Flask's built-in server is not production-ready
- **Security:** Containers run as root and lack proper secret management

These limitations are intentional to keep the workshop focused on Docker fundamentals. The bonus challenges address many of these concerns.

28 Workshop 2 Conclusion

Congratulations! You've completed the advanced Docker workshop. You've successfully:

- Built a production-like microservices application
- Created optimized Dockerfiles for multiple service types
- Configured service orchestration with Docker Compose
- Implemented background job processing with workers
- Managed data persistence and shared volumes
- Handled service dependencies and health checks
- Scaled services horizontally
- Troubleshot common containerization issues
- Applied Docker and microservices best practices

28.1 Key Takeaways

- **Containers enable microservices:** Each service runs independently with its own dependencies
- **Docker Compose simplifies orchestration:** Define complex multi-container apps in one file
- **Volumes provide persistence:** Critical for databases and file storage
- **Health checks ensure reliability:** Services wait for dependencies to be ready
- **Scaling is straightforward:** Add replicas with a simple command
- **Networking is automatic:** Services discover each other via DNS

28.2 Real-World Applications

This architecture pattern is used in production for:

- Video streaming platforms (Netflix, YouTube)
- E-commerce sites (background order processing)
- Data processing pipelines (ETL jobs)
- Image processing services (thumbnails, optimization)
- Email sending services (queue-based)
- Report generation systems

28.3 Next Steps

To continue your Docker journey:

- **Container orchestration:** Learn Kubernetes for production-scale deployments
- **CI/CD integration:** Automate building and deploying containers
- **Security:** Study container security, image scanning, and secrets management
- **Monitoring:** Implement comprehensive monitoring and logging solutions
- **Performance:** Learn about container resource management and optimization
- **Cloud platforms:** Deploy to AWS ECS, Google Cloud Run, or Azure Container Instances

29 Workshop 2 Cleanup

Clean up all resources created during this workshop:

```
# Stop and remove all services and volumes
$ docker compose down -v

# Remove the custom images
$ docker rmi starter-web starter-worker

# Clean up build cache and unused resources
$ docker system prune -a

# Verify cleanup
$ docker ps -a
$ docker images
$ docker volume ls
$ docker network ls
```

Final Note: Thank you for completing these Docker workshops! The skills you've learned form the foundation for modern application deployment and microservices architecture. Keep practicing and building!