Docker Usage

Reviewing Susie's Docker Adventure

Prerequisites

Before starting, ensure you have:

1. **Docker Desktop** installed on your computer
    - Download from: https://www.docker.com/products/docker-desktop
    - Follow the installation instructions for your operating system
2. **A text editor** (VS Code, Sublime Text, or any editor you prefer)
3. **Basic command line knowledge**

To verify Docker is installed correctly:

```
> docker --version
# Should output something like: Docker version 24.0.0
> which docker # Linux/Mac
/usr/local/bin/docker
```

Understanding Docker Architecture

Remember Susie's problem? She needed to deploy her Node.js application (`hello.js`) to different servers. Docker solves this by creating a **containerized environment** that packages:

1. **The Application Code** - Susie's `hello.js` file
2. **Runtime Environment** - Node.js v18
3. **Operating System Layer** - Alpine Linux
4. **Configuration** - Environment variables, exposed ports
5. **Execution Instructions** - How to start the app

Key Components

Docker Engine

- The core runtime that creates and manages containers
- Consists of:
  - **Docker Daemon** (`dockerd`) - Background service
  - **Docker CLI** (`docker`) - Command-line interface
  - **REST API** - Interface between CLI and daemon

Images vs Containers

- **Image**: Read-only template (like Susie's `susie-app` after `docker build`)
- **Container**: Running instance of an image (what happens after `docker run`)
- Think of it as: Image = Class, Container = Object

Docker Registry

A Docker Registry is like GitHub for Docker images — it lets you store, share, and distribute container images securely and efficiently.

- Central repository for images (Docker Hub)
- Where `node:18-alpine` comes from in Susie's Dockerfile

Common Example:

- Docker Hub (https://hub.docker.com) is the default public registry.
- You can also host your own private registry (e.g., registry.example.com).

Usage Flow:

```
docker pull nginx # Downloads the image from a registry.
docker tag myapp:1.0 yourusername/myapp:1.0

docker login # login to push
docker push yourusername/myapp:1.0 # Uploads your image to a registry.
```

Dockerfile Instructions Explained

Let's understand each line of Susie's Dockerfile:

```
FROM node:18-alpine
WORKDIR /app
COPY hello.js .
EXPOSE 3000
CMD ["node", "hello.js"]
```

FROM node:18-alpine

What it does:

- Sets the base image for your container
- `node:18-alpine` = Node.js version 18 on Alpine Linux

Why Alpine?

- **Size**: Alpine Linux is ~5MB (vs Ubuntu ~70MB)
- **Security**: Minimal attack surface
- **Speed**: Faster to download and deploy

Behind the scenes:

```
Docker Hub → Downloads node:18-alpine → Uses as foundation
```

Remember:

1. Docker does not download Linux kernel, it only downloads the core file system.
2. Docker also downloads all the necessary configuration files.

WORKDIR /app

What it does:

- Creates the directory `/app` if it doesn't exist
- Sets it as the current working directory
- All subsequent commands run from this directory

Why important:

- Organizes files within the container
- Avoids cluttering the root directory
- Makes COPY operations cleaner

COPY hello.js .

What it does:

- Copies `hello.js` from your host machine
- Places it in the current WORKDIR (`/app`)
- The `.` means "current directory in container"

Important notes:

- COPY happens at build time, not runtime
- Changes to `hello.js` require rebuilding the image
- Can use patterns: `COPY *.js .` or `COPY src/ ./src/`

EXPOSE 3000

What it does:

- Documents that the container listens on port 3000
- Does NOT actually open the port
- Serves as documentation for other developers

Common misconception:

- EXPOSE doesn't publish the port
- You still need `-p` flag when running

CMD ["node", "hello.js"]

What it does:

- Defines the default command when container starts
- Runs `node hello.js` inside the container
- Can be overridden at runtime

CMD vs RUN:

- `RUN` : Executes during build (e.g., `RUN npm install`)
- `CMD` : Executes when container starts

Building Images: The docker build Command

When Susie runs:

```
docker build -t susie-app .
```

What Happens Step by Step:

1. **Context Preparation**
    - Docker client sends the build context (current directory) to daemon
    - `.dockerignore` file can exclude files
2. **Layer Creation**
    - Each Dockerfile instruction creates a new layer
    - Layers are cached for efficiency

3. **Build Process**

```
Step 1/5 : FROM node:18-alpine
→ Downloads/uses cached base image

Step 2/5 : WORKDIR /app
→ Creates directory, adds layer

Step 3/5 : COPY hello.js .
→ Copies file, adds layer

Step 4/5 : EXPOSE 3000
→ Adds metadata, adds layer

Step 5/5 : CMD ["node", "hello.js"]
→ Sets default command, adds layer
```

4. **Image Tagging**

   - `-t susie-app` creates a tag

Build Options Explained:

```
# Build with tag
docker build -t myapp:v1.0 .

# Build with multiple tags
docker build -t myapp:latest -t myapp:v1.0 .

# Build from specific Dockerfile
docker build -f Dockerfile.prod -t myapp:prod .

# Build with build arguments
docker build --build-arg NODE_VERSION=18 -t myapp .

# Build without cache
docker build --no-cache -t myapp .
```

Docker Cache

Docker's cache saves time by reusing unchanged layers: `––no–cache` rebuilds everything fresh when you want to ensure consistency.

When you rebuild the image, Docker:

- Reuses unchanged layers from previous builds
- Skips re-running commands that haven't changed

This build cache makes builds much faster, since it avoids repeating steps such as installing packages or copying files that haven't changed.

Running Containers: The docker run Command

When Susie runs:

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

Complete Anatomy of docker run:

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

Essential Options:

Port Mapping (-p)

```
-p 3000:3000
# Format: -p [host-port]:[container-port]
# Maps port 3000 on host to port 3000 in container
```

Detached Mode (-d)

```
docker run -d -p 3000:3000 susie-app
# Runs in background, returns container ID
```

Interactive Mode (-it)

```
docker run -it node:18-alpine /bin/sh
# -i: Keep STDIN open
# -t: Allocate pseudo-TTY
# Used for shell access
```

Name Assignment (--name)

```
docker run -d --name my-app -p 3000:3000 susie-app
# Assigns friendly name instead of random ID
```

Environment Variables (-e)

```
docker run -e NODE_ENV=production -e PORT=3000 susie-app
# Sets environment variables inside container
```

Volume Mounting (-v)

```
docker run -v $(pwd)/data:/app/data susie-app
# Mounts host directory to container directory
```

What Happens When You Run:

1. Container Creation

- A new writable layer is added on top of the image
- The container gets a unique ID

2. Network Setup

- The container receives an IP address
- Port mappings are configured

3. Process Execution

- The image’s CMD instruction is executed
- The main process runs as PID 1 inside the container

4. Output Handling

- STDOUT and STDERR are captured
- Viewable using docker logs

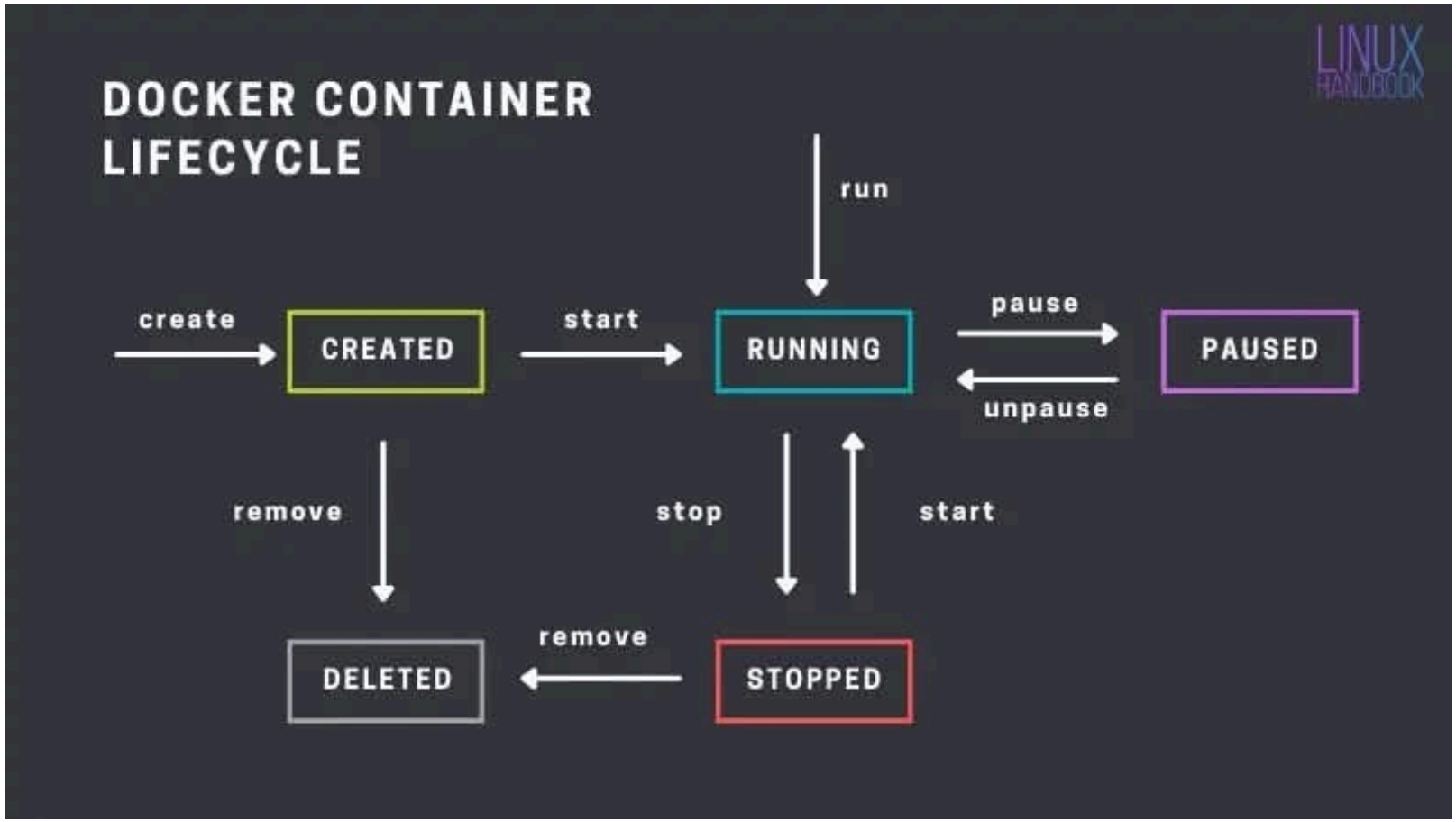
Each container is a Linux process, but it’s fully isolated because:

1. It can’t see processes outside the container
2. Its filesystem root (/) is isolated
3. It uses a union (layered) filesystem

Container Lifecycle Management

Container States:

1. **Created**: Container exists but not started
2. **Running**: Container is executing
3. **Paused**: Container is suspended
4. **Stopped**: Container finished executing
5. **Removed**: Container deleted


DOCKER CONTAINER
LIFECYCLE
LINUX HANDBOOK
run
create
CREATED
start
RUNNING
pause
PAUSED
unpause
remove
stop
start
remove
DELETED
STOPPED

Lifecycle Commands:

Create without starting:

```
docker create --name my-app susie-app
```

Start a created/stopped container:

```
docker start my-app
```

Stop a running container:

```
docker stop my-app          # Graceful shutdown (SIGTERM)
docker stop -t 30 my-app    # 30 second timeout
docker kill my-app          # Force kill (SIGKILL)
```

Pause/Unpause:

```
docker pause my-app    # Suspend all processes
docker unpause my-app  # Resume processes
```

Restart:

```
docker restart my-app  # Stop + Start
```

Remove:

```
docker rm my-app        # Remove stopped container
docker rm -f my-app     # Force remove running container
```

Viewing Container Information:

List containers:

```
docker ps                # Running containers
docker ps -a             # All containers
docker ps -q             # Only IDs
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
```

Inspect container:

```
docker inspect my-app  # Full JSON details
docker inspect -f '{{.NetworkSettings.IPAddress}}' my-app
```

View logs:

```
docker logs my-app              # All logs
docker logs -f my-app           # Follow logs (like tail -f)
docker logs --tail 50 my-app    # Last 50 lines
docker logs --since 2h my-app # Logs from last 2 hours
```

Execute commands:

```
docker exec my-app ls -la               # Run command
docker exec -it my-app /bin/sh          # Interactive shell
docker exec -u root my-app whoami       # Run as different user
```

Docker Compose Deep Dive

Even with a **Dockerfile**, running complex apps can be messy.

Problem with Just Dockerfile

- A **Dockerfile** defines **how to build one image**
- Real-world apps need **multiple containers** (e.g., web + DB + cache)
- You must run and link them **manually** using long `docker run` commands

What Docker Compose Does

- Defines **multi-container applications** in **one YAML file**
- Manages **build, network, and volume** configurations together
- Runs everything with **one command**

Purpose of `docker-compose.yml`

- A **single YAML file** that defines how multiple containers work together
- Describes each **service**, its **image/build**, **network**, and **volumes**
- Used by the command:

```
docker-compose up      # Build & start all services
docker-compose down    # Stop & remove containers
docker-compose ps      # List running services
```

Looking at Susie's docker-compose.yml:

```yaml
version: '3.8'

services:
  app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - app
```

Understanding Each Section:

version: '3.8'

- Specifies Compose file format version
- Determines available features
- Version 3.x is recommended for most use cases

services:

- Defines the containers to run
- Each service gets its own configuration

app service breakdown:

- `build: .` - Build from Dockerfile in current directory
- `ports:` - Same as `-p` in docker run
- `environment:` - Same as `-e` in docker run

nginx service breakdown:

- `image:` - Use pre-built image (no build needed)
- `volumes:` - Mount configuration file
- `depends_on:` - Start app before nginx

Docker Compose Commands:

Start services:

```
docker-compose up           # Start and attach
docker-compose up -d        # Start detached
docker-compose up --build   # Rebuild images
```

Stop services:

```
docker-compose down         # Stop and remove containers
docker-compose down -v      # Also remove volumes
docker-compose stop         # Just stop, don't remove
```

View status:

```
docker-compose ps           # List containers
docker-compose logs         # View all logs
docker-compose logs -f app # Follow specific service
```

Scale services:

```
docker-compose up -d --scale app=3  # Run 3 app instances
```

Docker Networking Concepts

How Containers Communicate:

When Susie uses Docker Compose, containers can communicate using service names:

Default Network:

- Compose creates a default network
- All services join this network
- Services reach each other by name

Example:

```
services:
  app:
    # This service is reachable as "app"

  nginx:
    # Can connect to app using hostname "app"
```

In nginx.conf:

```
upstream app {
    server app:3000;  # "app" resolves to app container
}
```

Network Types:

Bridge (default):

- Isolated network on host
- Containers get internal IPs
- Port mapping required for external access

Host:

- Container uses host's network directly
- No network isolation
- Better performance, less security

None:

- No network access & Complete isolation

Network Commands:

```
# List networks
docker network ls

# Inspect network
docker network inspect bridge

# Create custom network
docker network create my-network

# Connect container to network
docker network connect my-network my-container
```

Volume Management and Data Persistence

Problem: Containers are Ephemeral

When Susie's container stops, all data inside is lost. Volumes solve this.

Volume Types:

1. **Bind Mounts** (like in Susie's nginx config):

```
volumes:
  - ./nginx.conf:/etc/nginx/nginx.conf
```

- Maps host file/directory to container
- Good for development and config files

2. Named Volumes:

Docker reuses a named volume by its name, which makes it perfect for persistent data sharing between container runs.

```
volumes:
  - app-data:/app/data # use the named volume

volumes:
  app-data: # declare it
```

- app-data is stored at /var/lib/docker/volumes/app-data/_data (Linux)
- Data persists even if the container is removed
- Reusing the same name restores the same data volume

3. Anonymous Volumes:

```
VOLUME /app/logs
```

- Docker creates unnamed volume
- Deleted when container removed

Volume Commands:

```
# List volumes
docker volume ls

# Create volume
docker volume create my-data

# Inspect volume
docker volume inspect my-data

# Remove volume
docker volume rm my-data

# Remove unused volumes
docker volume prune
```

Best Practices and Troubleshooting

Dockerfile Best Practices:

1. Order matters for caching:

```
# Bad: COPY changes invalidate npm install cache
COPY . .
RUN npm install # reinstall always

# Good: npm install cached unless package.json changes
COPY package.json .
RUN npm install # only reinstall package.json changes
COPY . .
```

2. Minimize layers:

```
# Bad: Multiple RUN commands
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y git

# Good: Single RUN command
RUN apt-get update && \
    apt-get install -y curl git && \
    rm -rf /var/lib/apt/lists/*
```

3. Use specific tags:

```
# Bad: Latest tag can change
FROM node:latest

# Good: Specific version
FROM node:18.17.0-alpine
```

Common Issues and Solutions:

Issue: "Cannot connect to Docker daemon"

```
# Check if Docker is running
docker version

# Start Docker daemon
sudo systemctl start docker  # Linux
# Or start Docker Desktop on Mac/Windows
```

Issue: "Port already in use"

```
# Find what's using the port
lsof -i :3000  # Mac/Linux
netstat -ano | findstr :3000  # Windows

# Use different port
docker run -p 8080:3000 susie-app
```

Issue: "Container exits immediately"

```
# Check exit code and logs
docker ps -a
docker logs container-name

# Run interactively to debug
docker run -it susie-app /bin/sh
```

Issue: "Changes not reflected"

```
# For code changes: Rebuild image
docker build -t susie-app .

# For development: Use volumes
docker run -v $(pwd):/app susie-app
```

Cleanup Commands:

```
# Remove stopped containers
docker container prune

# Remove unused images
docker image prune

# Remove everything unused
docker system prune -a

# See disk usage
docker system df
```

Summary

Docker transforms deployment from Susie's 4-hour manual process to a 30-second automated one by:

1. **Packaging** - Everything in one container
2. **Isolation** - No conflicts between apps
3. **Portability** - Same container everywhere
4. **Reproducibility** - Identical environments
5. **Scalability** - Easy to replicate

The key is understanding:

- **Dockerfile** defines what goes in the image
- **docker build** creates the image
- **docker run** creates and starts containers
- **Docker Compose** orchestrates multiple containers

Remember: Docker is not just about running containers, it's about creating reproducible, scalable, and maintainable deployment processes!