Infrastructure as Code

Containers, Docker, and Docker Compose

Jason Kuruzovich

2026-01-16

Infrastructure as Code

System Definition and Reproducibility

Today's Agenda

Learning Objectives

1. Understand the **problems** that containers solve
2. Master **Docker fundamentals** - images, containers, volumes
3. Write effective **Dockerfiles** for Node.js applications
4. Use **Docker Compose** for multi-service orchestration
5. Implement **development workflows** with containers
6. Understand container **networking** and **data persistence**

The "Works on My Machine" Problem

Without Containers

```
Developer A: Node 18, MongoDB 6.0
Developer B: Node 20, MongoDB 7.0
Production:  Node 16, MongoDB 5.0

"But it works on my machine!"
```

- Different OS versions
- Different dependencies
- Different configurations

- Different data
- Hours debugging environment issues

With Containers

```
All environments run identical containers

Developer A: Docker
Developer B: Docker
Production:  Docker

"It works everywhere!"
```

- Identical runtime
- Identical dependencies
- Identical configuration
- Reproducible builds
- Focus on building features

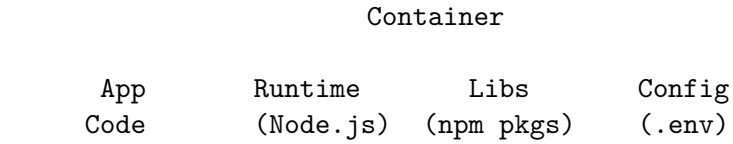
Container Fundamentals

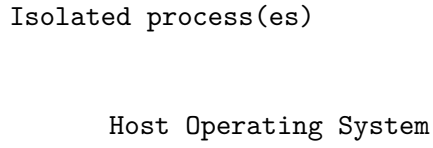
What is a Container?

A container is a **lightweight, standalone, executable package** that includes everything needed to run software:

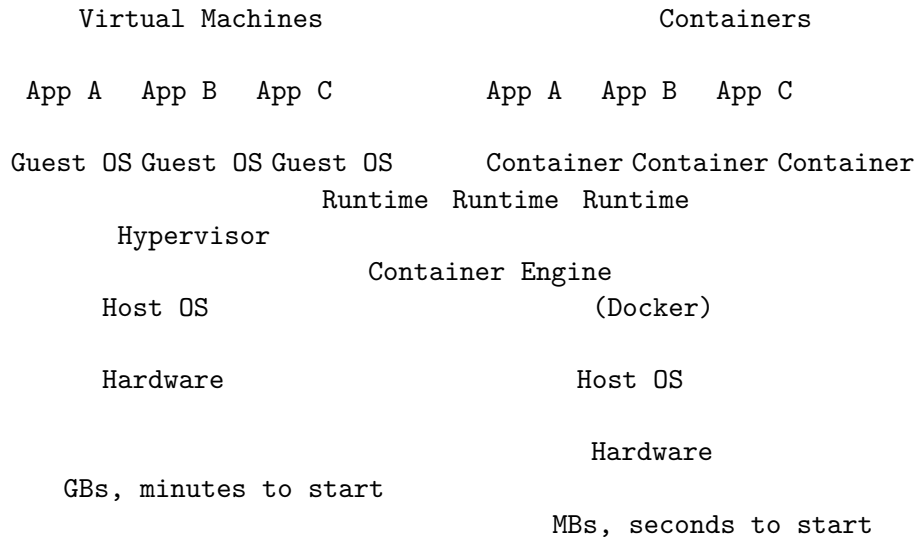
- Code
- Runtime
- System tools
- Libraries
- Settings

. . .





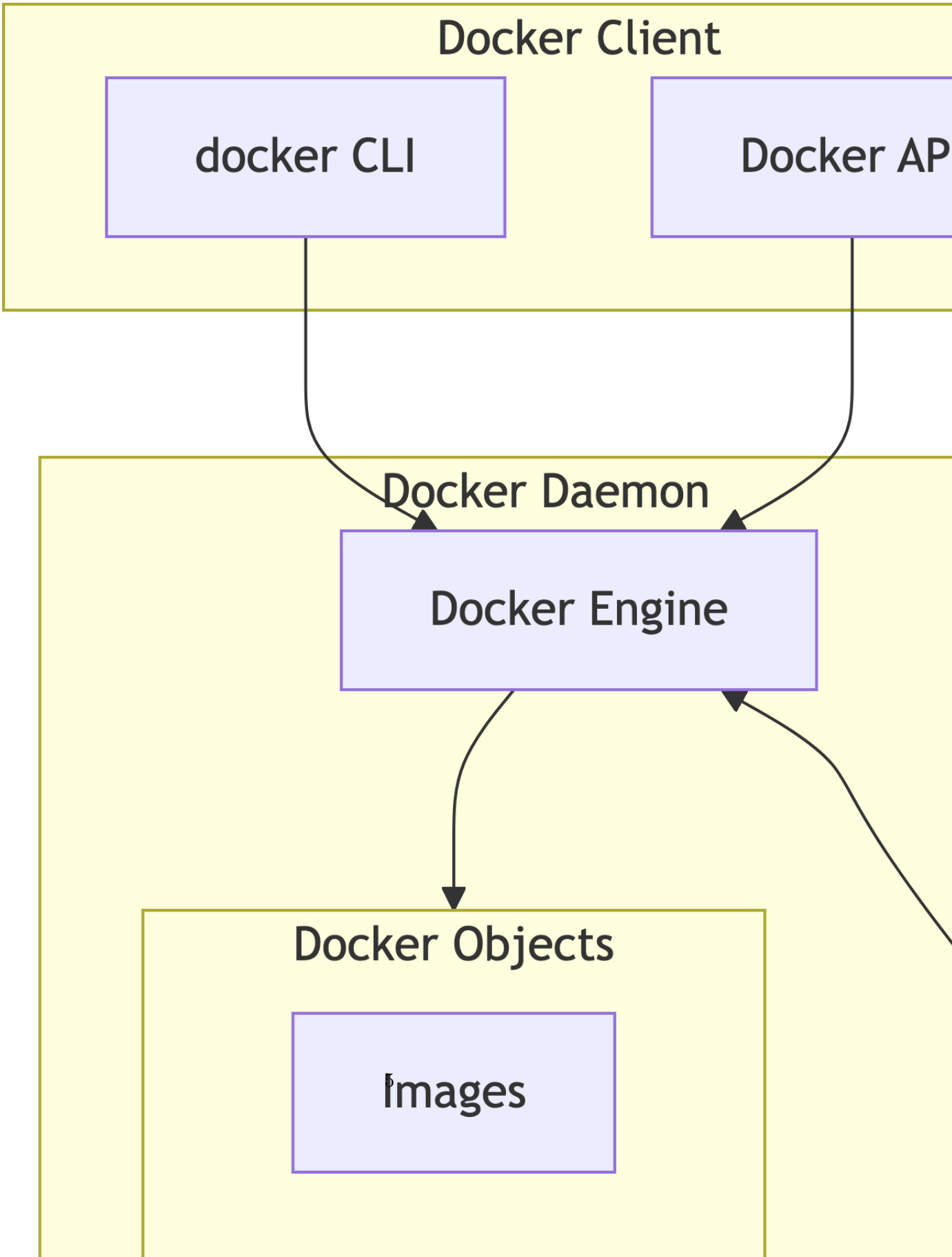
Containers vs. Virtual Machines



Key Differences

Aspect	Virtual Machines	Containers
Size	GBs	MBs
Startup	Minutes	Seconds
Isolation	Complete OS	Process-level
Performance	~5-10% overhead	Near-native
Density	10s per host	100s per host
Use Case	Different OS needs	Application packaging

Docker Architecture



Core Concepts

Image

- Read-only template
- Built from Dockerfile
- Composed of layers
- Versioned with tags
- Stored in registries

```
docker pull node:20-alpine
docker images
```

Container

- Running instance of image
- Isolated environment
- Has its own filesystem
- Can be started/stopped
- Ephemeral by default

```
docker run -it node:20-alpine
docker ps
```

Image Layers

```
  Layer 5: COPY . .                        Your code

  Layer 4: RUN npm install                 Dependencies

  Layer 3: COPY package*.json ./           Package files

  Layer 2: WORKDIR /app                    Directory

  Layer 1: FROM node:20-alpine             Base image


Each layer is cached - rebuild only changed layers
```

. . .

Key Insight: Order your Dockerfile commands from least to most frequently changing for optimal caching.

Writing Dockerfiles

Dockerfile Basics

```
# Start from a base image
FROM node:20-alpine

# Set working directory
WORKDIR /app

# Copy package files first (for caching)
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy application code
COPY . .

# Expose the port
EXPOSE 3000

# Define the command to run
CMD ["node", "server.js"]
```

Dockerfile Instructions

Instruction	Purpose	Example
`FROM`	Base image	`FROM node:20-alpine`
`WORKDIR`	Set working directory	`WORKDIR /app`
`COPY`	Copy files from host	`COPY . .`
`ADD`	Copy + extract archives	`ADD app.tar.gz /app`
`RUN`	Execute command (build time)	`RUN npm install`
`CMD`	Default command (run time)	`CMD ["npm", "start"]`

Instruction	Purpose	Example
`ENTRYPOINT`	Container entry point	`ENTRYPOINT ["node"]`
`ENV`	Environment variable	`ENV NODE_ENV=production`
`EXPOSE`	Document port	`EXPOSE 3000`
`VOLUME`	Mount point	`VOLUME ["/data"]`

Development vs. Production Dockerfile

Development

```
FROM node:20-alpine

WORKDIR /app

# Install all dependencies
COPY package*.json ./
RUN npm install

# Copy source (or use volume)
COPY . .

# Development command with hot reload
CMD ["npm", "run", "dev"]
```

Production

```
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM node:20-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
```

```
USER node
CMD ["node", "dist/server.js"]
```

Multi-Stage Builds

```
# Stage 1: Build
FROM node:20-alpine AS builder

WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: Production
FROM node:20-alpine AS production

WORKDIR /app

# Copy only what we need from builder
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/package*.json ./
RUN npm ci --only=production

# Security: non-root user
USER node

EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Benefits: Smaller images, no build tools in production

Best Practices for Dockerfiles

1. **Use specific tags** - `node:20-alpine` not `node:latest`
2. **Minimize layers** - Combine related RUN commands
3. **Order for caching** - Less changing → more changing
4. **Use .dockerignore** - Exclude unnecessary files
5. **Don't run as root** - Add `USER node`

6. **Multi-stage builds** - Separate build and runtime
7. **Health checks** - Add `HEALTHCHECK` instruction

. . .

```
# .dockerignore
node_modules
npm-debug.log
.git
.env
*.md
```

Docker Commands

Essential Commands

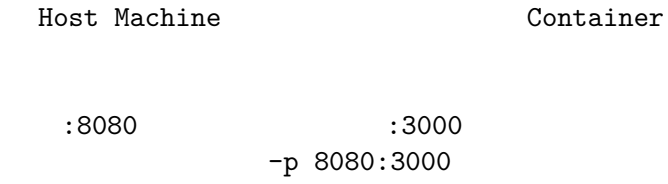
```
# Build an image
docker build -t myapp:1.0 .

# List images
docker images

# Run a container
docker run -d -p 3000:3000 --name myapp myapp:1.0

# List running containers
docker ps

# List all containers
docker ps -a

# View logs
docker logs myapp
docker logs -f myapp  # Follow

# Stop container
docker stop myapp

# Remove container
docker rm myapp
```

```
# Remove image
docker rmi myapp:1.0
```

Interactive Commands

```
# Run interactive shell
docker run -it node:20-alpine /bin/sh

# Execute command in running container
docker exec -it myapp /bin/sh

# Execute specific command
docker exec myapp npm run migrate

# Copy files to/from container
docker cp myapp:/app/logs ./logs
docker cp ./config.json myapp:/app/
```

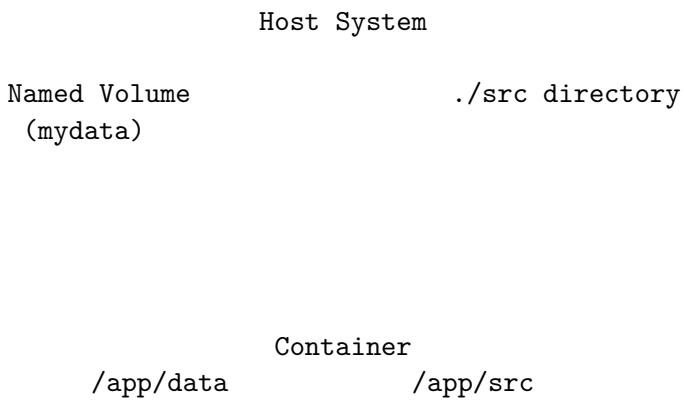
Port Mapping




```
docker run -p 8080:3000 myapp

# Multiple ports
docker run -p 8080:3000 -p 9229:9229 myapp

# All interfaces
docker run -p 0.0.0.0:8080:3000 myapp

# Random host port
docker run -p 3000 myapp
```

Volumes: Data Persistence

```
# Named volume (Docker-managed)
docker run -v mydata:/app/data myapp

# Bind mount (host directory)
docker run -v $(pwd)/src:/app/src myapp

# Read-only mount
docker run -v $(pwd)/config:/app/config:ro myapp
```



Docker Compose

What is Docker Compose?

Docker Compose is a tool for defining and running **multi-container applications**.

```
# docker-compose.yml
services:
  web:
    build: ./web
    ports:
      - "3000:3000"

  api:
```

```
    build: ./api
    ports:
      - "4000:4000"

  database:
    image: mongo:7
    volumes:
      - mongo-data:/data/db

volumes:
  mongo-data:
```

One command to start everything: `docker compose up`

Compose File Structure

```
# docker-compose.yml
version: "3.9"  # Optional in recent versions

services:          # Container definitions
  service_name:
    image: ...
    build: ...
    ports: ...
    volumes: ...
    environment: ...
    depends_on: ...

volumes:           # Named volumes
  volume_name:

networks:          # Custom networks
  network_name:

secrets:           # Sensitive data
  secret_name:
```

A Complete MERN Compose File

```yaml
services:
  # React Frontend
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile.dev
    ports:
      - "3000:3000"
    volumes:
      - ./frontend/src:/app/src
    environment:
      - REACT_APP_API_URL=http://localhost:4000
    depends_on:
      - api

  # Express API
  api:
    build:
      context: ./api
      dockerfile: Dockerfile.dev
    ports:
      - "4000:4000"
    volumes:
      - ./api/src:/app/src
    environment:
      - MONGODB_URI=mongodb://mongo:27017/myapp
      - NODE_ENV=development
    depends_on:
      - mongo

  # MongoDB Database
  mongo:
    image: mongo:7
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db
    environment:
      - MONGO_INITDB_DATABASE=myapp
```

```yaml
volumes:
  mongo-data:
```

Service Configuration Options

```yaml
services:
  api:
    # Build configuration
    build:
      context: ./api
      dockerfile: Dockerfile
      args:
        NODE_VERSION: 20

    # Or use existing image
    image: node:20-alpine

    # Container name
    container_name: myapp-api

    # Port mapping
    ports:
      - "4000:4000"
      - "9229:9229"  # Debug port

    # Volume mounts
    volumes:
      - ./api:/app
      - /app/node_modules  # Anonymous volume

    # Environment variables
    environment:
      - NODE_ENV=development
      - DB_HOST=mongo

    # Or from file
    env_file:
      - .env.development

    # Dependencies
```

```
    depends_on:
      - mongo
      - redis

    # Restart policy
    restart: unless-stopped

    # Resource limits
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
```

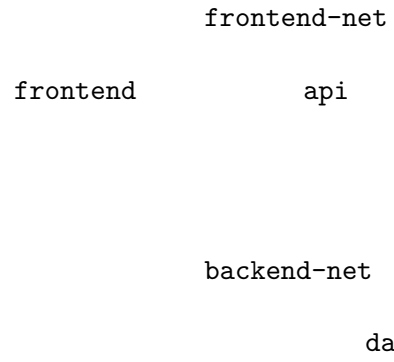
Docker Compose Commands

```
# Start all services
docker compose up

# Start in detached mode (background)
docker compose up -d

# Start specific service
docker compose up api

# Stop all services
docker compose down

# Stop and remove volumes
docker compose down -v

# Rebuild images
docker compose build
docker compose up --build

# View logs
docker compose logs
docker compose logs -f api

# Execute command in service
```

```
docker compose exec api npm run migrate

# List running services
docker compose ps

# Scale a service
docker compose up -d --scale api=3
```

Networking in Compose

```
services:
  frontend:
    networks:
      - frontend-net

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

  database:
    networks:
      - backend-net

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



Services can reach each other by name within a network.

Health Checks

```yaml
services:
  api:
    build: ./api
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:4000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 40s

  mongo:
    image: mongo:7
    healthcheck:
      test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]
      interval: 10s
      timeout: 5s
      retries: 5

  api-dependent:
    build: ./dependent
    depends_on:
      api:
        condition: service_healthy
```

Development Workflow

Hot Reloading with Volumes

```yaml
services:
  api:
    build:
      context: ./api
```

```
      dockerfile: Dockerfile.dev
    volumes:
      # Mount source code
      - ./api/src:/app/src
      # Preserve node_modules from container
      - /app/node_modules
    command: npm run dev  # nodemon
```

```
# api/Dockerfile.dev
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

# Don't copy source - use volume mount
# COPY . .

EXPOSE 4000

CMD ["npm", "run", "dev"]
```

Development vs. Production Compose

docker-compose.yml (dev)

```
services:
  api:
    build:
      context: ./api
      dockerfile: Dockerfile.dev
    volumes:
      - ./api/src:/app/src
    environment:
      - NODE_ENV=development
    command: npm run dev
```

docker-compose.prod.yml

```yaml
services:
  api:
    build:
      context: ./api
      dockerfile: Dockerfile
    environment:
      - NODE_ENV=production
    restart: always
    deploy:
      replicas: 2
```

```
# Development
docker compose up

# Production
docker compose -f docker-compose.prod.yml up
```

Debugging in Containers

```yaml
services:
  api:
    build: ./api
    ports:
      - "4000:4000"
      - "9229:9229"  # Node.js debug port
    command: node --inspect=0.0.0.0:9229 src/server.js
```

VS Code `launch.json`:

```
{
  "type": "node",
  "request": "attach",
  "name": "Docker: Attach",
  "port": 9229,
  "address": "localhost",
  "localRoot": "${workspaceFolder}/api/src",
  "remoteRoot": "/app/src"
}
```

Common Development Tasks

```
# Install new package
docker compose exec api npm install express-validator

# Run database migrations
docker compose exec api npm run migrate

# Access database shell
docker compose exec mongo mongosh myapp

# Run tests
docker compose exec api npm test

# View real-time logs
docker compose logs -f api

# Restart single service
docker compose restart api

# Rebuild and restart
docker compose up --build api
```

Live Demo

Building a MERN Stack with Docker Compose

Let's build this together:

1. Create project structure
2. Write Dockerfiles
3. Configure Docker Compose
4. Start the stack
5. Test hot reloading
6. Explore container networking

Project Structure

```
mern-app/
   docker-compose.yml
```

```
.env
frontend/
    Dockerfile
    Dockerfile.dev
    package.json
    src/
api/
    Dockerfile
    Dockerfile.dev
    package.json
    src/
.dockerignore
```

Step 1: API Dockerfile

```
# api/Dockerfile.dev
FROM node:20-alpine

WORKDIR /app

# Install dependencies
COPY package*.json ./
RUN npm install

# Install nodemon globally for hot reload
RUN npm install -g nodemon

EXPOSE 4000

CMD ["nodemon", "src/server.js"]
```

Step 2: Docker Compose

```
# docker-compose.yml
services:
  api:
    build:
      context: ./api
      dockerfile: Dockerfile.dev
```

```
    ports:
      - "4000:4000"
    volumes:
      - ./api:/app
      - /app/node_modules
    environment:
      - MONGODB_URI=mongodb://mongo:27017/mernapp
      - PORT=4000
    depends_on:
      - mongo

  mongo:
    image: mongo:7
    ports:
      - "27017:27017"
    volumes:
      - mongo-data:/data/db

  mongo-express:
    image: mongo-express
    ports:
      - "8081:8081"
    environment:
      - ME_CONFIG_MONGODB_URL=mongodb://mongo:27017/
    depends_on:
      - mongo

volumes:
  mongo-data:
```

Step 3: Start and Verify

```
# Build and start
docker compose up --build

# In another terminal
curl http://localhost:4000/health

# Check MongoDB
docker compose exec mongo mongosh mernapp --eval "db.stats()"
```

```
# View logs
docker compose logs -f api

# Stop everything
docker compose down
```

Best Practices Summary

Docker Best Practices

1. **Use official base images** with specific versions
2. **Minimize image size** with Alpine variants and multi-stage builds
3. **Don't run as root** - use non-privileged users
4. **Use .dockerignore** to exclude unnecessary files
5. **Leverage layer caching** by ordering Dockerfile commands
6. **Use health checks** for production containers
7. **Don't store secrets** in images or environment variables
8. **Tag images meaningfully** - use semantic versioning

Compose Best Practices

1. **Use named volumes** for persistent data
2. **Define networks explicitly** for service isolation
3. **Use depends_on with health checks** for startup order
4. **Separate dev and prod** configurations
5. **Use environment variables** for configuration
6. **Don't expose database ports** in production
7. **Set resource limits** in production
8. **Use restart policies** appropriately

Looking Ahead

Lab 1: Next Week (January 20)

Infrastructure as Code with Docker Compose

You will:

- Set up a complete MERN development environment

- Write Dockerfiles for frontend and backend
- Configure multi-service orchestration
- Implement volume mounting for development
- Test the complete stack

Preparation for Lab 1

Install

- Docker Desktop
- VS Code with Docker extension
- Git

Verify

```
docker --version
docker compose version
```

Review

- Today's slides
- Docker Docs: Get Started
- Compose File Reference

Summary

Today we covered:

1. Problems containers solve
2. Docker fundamentals - images, containers, volumes
3. Writing effective Dockerfiles
4. Multi-stage builds for optimization
5. Docker Compose for multi-service apps
6. Development workflows with containers
7. Container networking and data persistence

Questions?

Office Hours: Tuesday 9-11 AM, Pitt 2206

Email: kuruzj@rpi.edu

Appointments: bit.ly/jason-rpi

See you Tuesday for Lab 1!