Chapter 4: The Ship — Containerization with Docker

Introduction: What You Will Learn

Your code works on your laptop. But the professor's laptop is different. The TA's laptop is different. How do we ensure it works everywhere?

By the end of this chapter, you will understand: 1. What containerization is and why we need it 2. The difference between containers and virtual machines 3. Every line of a Dockerfile 4. Multi-stage builds (and why they matter) 5. Docker Compose for orchestration 6. How to troubleshoot common issues

---

Part 1: The Problem — "It Works on My Machine"

The Scenario

You build an amazing app. You test it. It works perfectly.

You show it to the professor.

"It doesn't work."

Why? Because: - You have Go 1.24, they have Go 1.21 - You have Node.js, they don't - Your OS is macOS, theirs is Windows - Your environment variables are different

This is the #1 problem in software deployment.

The Solution: Ship the Environment

Instead of saying: "Here's my code, figure out how to run it"

We say: "Here's a box containing my code AND everything needed to run it"

That box is called a **container**.

Part 2: Containers vs Virtual Machines

Virtual Machines (VMs)

A VM is like having a second computer inside your computer:

```
          Your Physical Computer

         Host Operating System

            Hypervisor (VMware)

   VM 1              VM 2
```

```
Guest OS          Guest OS
(Ubuntu)          (Windows)

Your App          Another App
```

Problem: Each VM has a complete operating system (2-10 GB each). Slow to start.

Containers

Containers share the host OS but isolate the application:

```
          Your Physical Computer

         Host Operating System

            Docker Engine

  Container 1   Container 2

   App +         App +
   Libs          Libs
```

Benefits: - No guest OS (containers are MBs, not GBs) - Start in milliseconds - Share resources efficiently

Aspect	VM	Container
Size	Gigabytes	Megabytes
Startup	Minutes	Seconds
Isolation	Complete (own OS)	Process-level
Use case	Different OS	Same OS, different apps

Part 3: Docker Concepts

Images vs Containers

Image: A recipe/template. Read-only. Created from a Dockerfile. **Container**: A running instance of an image. Has state.

Analogy: - **Image** = Cookie cutter - **Container** = Cookie

You can make many cookies (containers) from one cutter (image).

The Dockerfile

A Dockerfile is a text file that defines how to build an image.

```
FROM ubuntu:22.04          # Start with Ubuntu
RUN apt-get update         # Run a command
COPY ./myapp /app          # Copy files into the image
WORKDIR /app               # Set the working directory
CMD ["./myapp"]            # What to run when container starts
```

Part 4: Our Backend Dockerfile — Line by Line

Here's our `Dockerfile.backend`:

```
# Stage 1: Build
FROM golang:1.24-alpine AS builder
```

Line 1-2: - `FROM golang:1.24-alpine`: Start with an image that has Go installed - `golang`: Official Go image - `1.24`: Version of Go - `alpine`: A tiny Linux variant (5 MB vs 800 MB for full Ubuntu) - `AS builder`: Name this stage "builder" (for multi-stage build)

```
WORKDIR /app
```

Line 4: - Set the working directory inside the container to `/app` - All following commands run from this directory - Like `cd /app` but persistent

```
COPY go.mod ./
```

Line 6: - Copy `go.mod` from your computer into the container - `./` means current directory in container (`/app`)

```
RUN go mod download
```

Line 8: - Download Go dependencies defined in `go.mod` - `RUN` executes a command during build

Why copy go.mod first? Docker caches layers. If `go.mod` hasn't changed, it reuses the cached dependencies. This makes rebuilds FAST.

```
COPY service/ ./service/
```

Line 10: - Copy your source code into the container

```
RUN go build -o /app/api service/cmd/api/main.go
```

Line 12: - Compile the Go code - `-o /app/api`: Output the binary to `/app/api` - This produces a single executable file

Now we have a compiled binary. But the image also contains: - The entire Go compiler (~500 MB) - All source code - All dependencies

We don't need any of that to **run** the app!

```
# Stage 2: Run
FROM alpine:latest
```

Line 14-15: - Start a NEW image from Alpine Linux - This is the "runner" stage - Alpine is ~5 MB compared to ~800 MB for golang

```
WORKDIR /root/
```

Line 17: - Set working directory in the new image

```
COPY --from=builder /app/api .
```

Line 19: - `--from=builder`: Copy from the first stage (named "builder") - `/app/api`: The compiled binary - `.`: Copy to current directory (`/root/`)

This is the magic of **multi-stage builds**: 1. Stage 1: Big image with compiler, build the app 2. Stage 2: Tiny image, copy ONLY the compiled binary

Final image size: ~10 MB instead of ~800 MB!

```
EXPOSE 3000
```

Line 21: - Document that this container listens on port 3000 - This is informational; you still need to map ports when running

```
CMD ["./api"]
```

Line 23: - When the container starts, run `./api` - This starts your Go server

---

Part 5: Our Frontend Dockerfile — Line by Line

The frontend is much simpler:

```
FROM nginx:alpine
```

Line 1: - Use Nginx, a web server - Alpine variant for small size

```
COPY service/web /usr/share/nginx/html
```

Line 3: - Copy your frontend files (HTML, CSS, JS) into Nginx's default directory - Nginx automatically serves files from this directory

```
EXPOSE 80
```

Line 5: - Nginx listens on port 80 by default

That's it! Nginx handles everything: - Serving static files - Caching - HTTP compression - Efficient file delivery

We don't need Node.js because we're not using React or a build process. Our HTML/CSS/JS files are ready to serve as-is.

Part 6: Docker Compose — Orchestrating Multiple Containers

The Problem

We have TWO containers: 1. Backend (Go) on port 3000 2. Frontend (Nginx) on port 80

How do we: - Build both? - Start both? - Make them talk to each other?

The Solution: Docker Compose

`docker-compose.yml` defines multiple services:

```yaml
services:
  backend:
    build:
      context: .
      dockerfile: Dockerfile.backend
    ports:
      - "3000:3000"
    container_name: wasatext-backend

  frontend:
    build:
      context: .
      dockerfile: Dockerfile.frontend
    ports:
      - "8081:80"
    depends_on:
      - backend
    container_name: wasatext-frontend
```

Let me explain each part:

The `services` Section

```yaml
services:
```

A "service" is a container definition. We define two: `backend` and `frontend`.

Backend Service

```yaml
  backend:
    build:
      context: .
      dockerfile: Dockerfile.backend
```

- `build`: How to build the image
- `context: .`: Build from current directory
- `dockerfile: Dockerfile.backend`: Use this file

```yaml
    ports:
      - "3000:3000"
```

- Port mapping: `HOST:CONTAINER`
- `3000:3000`: When I access `localhost:3000` on my laptop, forward to port `3000` inside the container

```
container_name: wasatext-backend
```

- Give the container a specific name (instead of auto-generated)

Frontend Service

```
frontend:
  build:
    context: .
    dockerfile: Dockerfile.frontend
  ports:
    - "8081:80"
```

- Map port `8081` on host to port `80` in container
- Why 8081? Maybe port 80 is already used on your laptop

```
depends_on:
  - backend
```

- Start the backend BEFORE starting the frontend
- Ensures the API is ready when the UI loads

Running Docker Compose

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

# Run in background (detached)
docker compose up -d --build

# Stop all services
docker compose down

# View logs
docker compose logs -f

# View running containers
docker compose ps
```

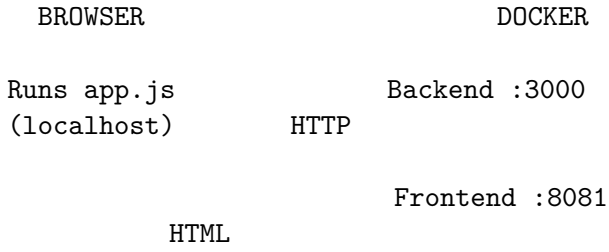
Part 7: Networking Between Containers

How Does the Frontend Talk to the Backend?

When both containers run via Docker Compose, they're on the same virtual network.

However, our frontend JavaScript runs **in the browser**, not in the container!

```
YOUR LAPTOP
```



1. Browser loads page from `localhost:8081` (Frontend container)
2. `app.js` runs IN THE BROWSER
3. `app.js` makes fetch requests to `localhost:3000` (Backend container)

This is why we use `localhost:3000` in `app.js`, not some Docker network address.

Part 8: Troubleshooting Docker

Common Error: "Cannot connect to Docker daemon"

```
Cannot connect to the Docker daemon at unix:///var/run/docker.sock
```

Cause: Docker Desktop isn't running. **Fix**: Open Docker Desktop application.

Common Error: "Port already in use"

```
Error: bind: address already in use
```

Cause: Something else is using port 3000 or 8081. **Fix**:

```
# Find what's using the port
lsof -i :3000

# Kill it
kill -9 <PID>

# Or change the port in docker-compose.yml
ports:
  - "3001:3000"  # Use 3001 instead
```

Common Error: Build fails

Cause: Usually syntax errors in your code. **Fix**: Read the error message carefully. It usually points to the exact line.

Useful Commands

```
# See all containers
docker ps -a

# See all images
docker images

# Remove all stopped containers
docker container prune

# Remove all unused images
docker image prune

# Nuclear option: remove EVERYTHING
docker system prune -a
```

Part 9: Why Multi-Stage Builds Matter

Without Multi-Stage

If we built everything in one stage:

```
FROM golang:1.24
COPY . .
RUN go build -o /app/api main.go
CMD ["./api"]
```

The final image contains: - Alpine Linux (~5 MB) - Go compiler (~500 MB) - All Go source code (~1 MB) - All dependencies (~100 MB) - Your compiled binary (~10 MB)

Total: ~616 MB

With Multi-Stage

The final image contains: - Alpine Linux (~5 MB) - Your compiled binary (~10 MB)

Total: ~15 MB

Why Size Matters

1. **Faster deployment**: Smaller images upload/download faster
2. **Security**: Less code = less attack surface
3. **Cost**: Cloud storage and transfer costs scale with size
4. **Disk space**: 100 containers × 600 MB = 60 GB vs 100 × 15 MB = 1.5 GB

Oral Exam Defense: "I use multi-stage Docker builds to minimize image size. The build stage has the Go compiler; the run stage has only the compiled binary. This reduces the image from ~600 MB to ~15 MB, improving deployment speed and security by removing unnecessary tools from production."

Summary Checklist

Before the oral exam, make sure you understand:

- ☐ The difference between containers and VMs
- ☐ What an image vs container is
- ☐ Every Dockerfile instruction (FROM, COPY, RUN, CMD, EXPOSE, WORKDIR)
- ☐ What multi-stage builds are and why they matter
- ☐ How docker-compose.yml orchestrates multiple containers
- ☐ How port mapping works
- ☐ Why our frontend uses localhost:3000 to reach the backend

---

Complete Command Reference

```
# Build and run everything
docker compose up --build

# Run in background
docker compose up -d --build

# Stop everything
docker compose down

# View logs
docker compose logs -f backend
docker compose logs -f frontend

# Restart a specific service
docker compose restart backend

# Rebuild just one service
docker compose build backend

# Open a shell inside a running container
docker exec -it wasatext-backend /bin/sh
```

Oral Exam Questions

Q: Why Docker instead of just instructions? "Docker guarantees reproducibility. The environment is identical everywhere—my laptop, the TA's laptop, the grading server. 'It works on my machine' becomes 'it works everywhere' because we ship the machine."

Q: Explain multi-stage builds. "Multi-stage builds separate compilation from execution. The first stage has build tools (compilers, package managers). The second stage has only the runtime. We

copy artificial from stage one to stage two, leaving behind GBs of unnecessary tools."

Q: Why Alpine Linux? "Alpine is ~5 MB versus ~100 MB for Ubuntu. It uses musl libc and busybox, providing just enough to run our app. For Go, which compiles to a static binary, we need almost nothing from the OS."

Q: How do the frontend and backend communicate? "The frontend JavaScript runs in the user's browser, not in Docker. It makes HTTP requests to localhost:3000, which Docker maps to the backend container. CORS headers allow cross-origin requests from the frontend origin."