Docker Diaries: Building and Shipping Apps

What is Docker?

Imagine you're moving to a new house. Instead of moving everything piece by piece, you pack everything in a shipping container. This container:

- Has everything you need
- Can be moved anywhere easily
- Works the same way wherever it goes
- Doesn't interfere with other containers

Docker is like that shipping container for your software!

Why Use Docker?

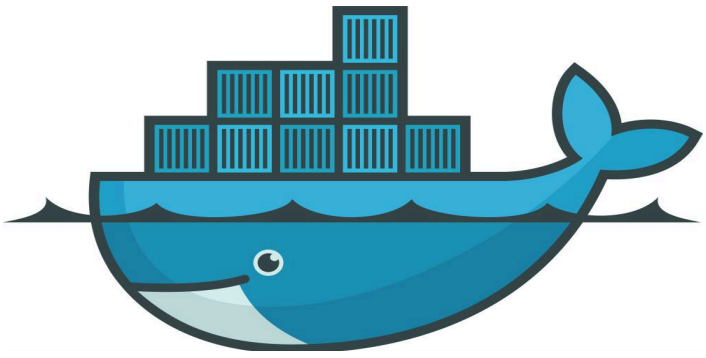
- **Consistency**: "Works on my machine" becomes "Works everywhere"
- **Isolation**: Applications don't interfere with each other
- **Efficiency**: Uses fewer resources than virtual machines
- **Speed**: Quick to start, stop, and deploy
- **Scalability**: Easy to scale up or down as needed

Core Concepts

1. Container 📦

A lightweight, standalone package containing your application and everything it needs to run.

- **What it includes**:
  - Application code
  - Runtime environment
  - System tools
  - Libraries
  - Settings



2. Image 🖼️

A blueprint for creating containers.

- **Characteristics**:
  - Read-only template
  - Contains application code and dependencies
  - Can be versioned (e.g., myapp:1.0, myapp:latest)
  - Stored in registries

3. Dockerfile 📝

A text file with instructions for building a Docker image.

- **Components**:
  - Base image to start from
  - Commands to run
  - Files to copy
  - Ports to expose

4. Docker Registry 🏢

Storage for Docker images.

- **Examples**:
  - Docker Hub (public repository)
  - Private registries (AWS ECR, Google Container Registry)
  - Local registry on your computer

Getting Started

Basic Commands

```
# Download an image
docker pull nginx

# List downloaded images
docker images

# Run a container
docker run -d -p 8080:80 nginx

# List running containers
docker ps
```

```
# Stop a container
docker stop container_id

# Remove a container
docker rm container_id

# Remove an image
docker rmi image_name
```

Creating Docker Images

Simple Dockerfile Example

```
# Use official Python runtime
FROM python:3.8-slim

# Set working directory
WORKDIR /app

# Copy requirements first (for better caching)
COPY requirements.txt .

# Install dependencies
RUN pip install -r requirements.txt

# Copy application code
COPY . .

# Tell Docker what port the application uses
EXPOSE 5000

# Run application
CMD ["python", "app.py"]
```

Explanation of Docker Dockerfile

Let me explain each line of this Dockerfile and what happens behind the scenes in simple language:

```
FROM python:3.8-slim
```

What it means: This is like saying "Start with a pre-made computer that already has Python 3.8 installed."

Behind the scenes: Docker downloads a minimal (slim) version of Python 3.8 from Docker Hub (a public library of container images). This image contains a basic operating system (Debian) with Python 3.8 already installed. It's the foundation for your container.

```
WORKDIR /app
```

What it means: This creates and sets up a folder called "/app" as your working area.

Behind the scenes: Docker creates a directory called "/app" inside the container and makes it the default location where future commands will run. It's like opening a new folder on your computer and telling the terminal "work in this folder from now on."

```
COPY requirements.txt .
```

What it means: This copies your list of Python libraries needed for your application.

Behind the scenes: Docker takes the requirements.txt file from your local computer (where you're building the Docker image) and copies it into the /app directory in the container. The "." means "copy to the current working directory" (which is /app).

```
RUN pip install -r requirements.txt
```

What it means: This installs all the Python libraries your application needs.

Behind the scenes: Docker runs the pip command inside the container to read the requirements.txt file and install all the listed Python packages. This creates a new layer in your Docker image containing all these installed packages. Any errors during installation would stop the build process here.

```
COPY . .
```

What it means: This copies all your application code into the container.

Behind the scenes: Docker copies all files and folders from the build context (the directory where you're running the docker build command) into the /app directory in the container. This includes your Python files, any data files, configuration files, etc. The first "." means "copy everything from the current directory on your local machine" and the second "." means "to the current working directory in the container (/app)".

```
EXPOSE 5000
```

What it means: This tells Docker that your application will use port 5000.

Behind the scenes: This doesn't actually open any ports - it's just documentation that helps other developers understand how your application works. It's like adding a note saying "This application expects to communicate on port 5000." To actually make the port accessible when running the container, you'll need to use the -p option with docker run.

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

What it means: This tells Docker what command to run when the container starts.

Behind the scenes: When someone runs your container (using docker run), this is the command that will execute automatically. In this case, it will run your app.py Python script. This doesn't happen during the build phase but only when the container starts. If this command ends (e.g., if your app crashes), the container will stop running.

The Build Process Explained

When you run `docker build -t myapp .` using this Dockerfile:

1. **Layer 1:** Docker starts with the python:3.8-slim base image
2. **Layer 2:** Creates the /app directory and makes it the working directory
3. **Layer 3:** Copies requirements.txt into the container
4. **Layer 4:** Runs pip install to install dependencies (this can take time)
5. **Layer 5:** Copies your application code into the container
6. **Layer 6:** Sets metadata indicating port 5000 will be used
7. **Layer 7:** Sets metadata for the default command to run

Each step creates a new "layer" in the Docker image. These layers are cached, so if you change your code but not your requirements.txt, Docker can reuse the cached layers for steps 1-4 and only rebuild from step 5, saving time.

The Running Process Explained

When someone runs `docker run -p 5000:5000 myapp`:

1. Docker creates a container from your image
2. It sets up a virtual environment isolated from the host computer
3. It maps port 5000 from the container to port 5000 on the host computer
4. It runs your application using the command `python app.py`
5. Your application starts and listens for connections on port 5000

The container continues running as long as your Python application is running. If your application stops or crashes, the container stops too.

Building and Running

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

# Run a container from the image
docker run -d -p 5000:5000 myapp:1.0
```

Sample Dockerfiles for Different Languages

1. Python Web Application

```
FROM python:3.9-slim

WORKDIR /app

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

COPY . .

EXPOSE 5000

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

2. Node.js Application

```
FROM node:16-alpine

WORKDIR /app

# Copy package.json and package-lock.json first for better caching
COPY package*.json ./
RUN npm install

# Copy the rest of the application
COPY . .

EXPOSE 3000

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

3. Go Application

```
FROM golang:1.18-alpine AS builder

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN go build -o main .

# Use a smaller image for the final container
FROM alpine:latest

WORKDIR /app

COPY --from=builder /app/main .

EXPOSE 8080

CMD ["./main"]
```

4. Ruby on Rails Application

```
FROM ruby:3.0

WORKDIR /app

# Install dependencies
COPY Gemfile Gemfile.lock ./
RUN bundle install

# Copy application code
COPY . .

EXPOSE 3000

CMD ["rails", "server", "-b", "0.0.0.0"]
```

5. Java Spring Boot Application

```
FROM openjdk:17-jdk-slim

WORKDIR /app

COPY build/libs/*.jar app.jar

EXPOSE 8080

CMD ["java", "-jar", "app.jar"]
```

Multi-Stage Builds

Multi-stage builds allow you to use multiple FROM statements in your Dockerfile. Each FROM instruction starts a new stage of the build. This is particularly useful for:

1. **Reducing final image size** by leaving build tools behind
2. **Improving security** by not including build-time vulnerabilities in your final image
3. **Simplifying the build process** with a single Dockerfile

Simple Multi-Stage Example

```
dockerfile
# Build stage
FROM node:16 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine
# Copy only the built app from the previous stage
COPY --from=builder /app/build /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

This Dockerfile:

1. Uses Node.js to build a JavaScript application
2. Takes only the compiled output and puts it in a lightweight Nginx image
3. Discards all the Node.js tools and dependencies

Multi-Stage Benefits

- **Smaller Images**: Only keep what's needed to run the application
- **Better Security**: Fewer components means fewer vulnerabilities
- **Faster Deployments**: Smaller images download faster
- **Cleaner Process**: No need for separate build scripts

Multi-Stage Example with Go

```
dockerfile
# Build stage
FROM golang:1.18 AS builder
WORKDIR /app
COPY go.mod go.sum ./
RUN go mod download
COPY . .
# Build the Go app with static linking
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

# Final stage - from scratch is the smallest possible image
```

```
FROM scratch
# Copy the binary from builder
COPY --from=builder /app/main /main
# Run the binary
ENTRYPOINT ["/main"]
```

This creates a tiny image containing only your compiled Go binary.

Setting Environment Variables at Runtime

```
# Override environment variables when running container
docker run -d -p 5000:5000 \
  -e APP_ENV=development \
  -e LOG_LEVEL=debug \
  -e DATABASE_URL=postgres://dev:dev@localhost:5432/devdb \
  myapp:1.0
```

Using Environment Files

Create a .env file:

APP_ENV=staging

LOG_LEVEL=warning

DATABASE_URL=postgres://staging:pwd@db:5432/stagingdb

Then run:

```

# Load all environment variables from file
docker run -d -p 5000:5000 --env-file .env myapp:1.0
```

Docker Compose: Managing Multiple Containers

Docker Compose lets you define and run multi-container applications.

Simple Docker Compose Example

```yaml
version: '3'
services:
  web:
    build:
      context: ./web
    ports:
      - "5000:5000"
    depends_on:
      - db
      - userservice
      - notification

  userservice:
    build:
      context: ./userservice
    ports:
      - "5001:5001"
    depends_on:
      - db

  notification:
    build:
      context: ./notification
    ports:
      - "5002:5002"

  db:
    image: postgres:13
    environment:
      - POSTGRES_PASSWORD=example
      - POSTGRES_DB=myapp
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Environment Variables in Docker Compose

```yaml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      - APP_ENV=development
      - LOG_LEVEL=debug
      - DATABASE_URL=postgres://dev:dev@db:5432/devdb
    # Alternative: use env file
    # env_file:
    #   - .env.development
```

Running Docker Compose

```
# Start services
docker-compose up

# Start in background
docker-compose up -d

# Stop services
docker-compose down
```

Advanced Topics Explained Simply

1. Docker Networking

Docker creates virtual networks that allow containers to communicate.

Network Types:
Bridge: Default network. Containers can talk to each other if on the same bridge.

```
 # Create a network
docker network create my-network

# Run containers on the network
docker run -d --network my-network --name container1 nginx
docker run -d --network my-network --name container2 alpine
```

- **Host**: Container shares the host's network - faster but less isolated.

  docker run -d --network host nginx

- **None**: Container has no network access - maximum isolation.

  docker run -d --network none alpine

2. Docker Volumes and Storage

Volumes let containers store data that persists even after containers are deleted.

Types of Docker Storage:
Volumes: Managed by Docker, stored in a special location.

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

# Use a volume
docker run -d -v my-data:/app/data nginx
```

- **Bind Mounts**: Connect a container to a specific folder on your machine.

  # Map local folder to container

  docker run -d -v /path/on/host:/path/in/container nginx

- **tmpfs Mounts**: Temporary storage in memory - fast but disappears when container stops.

  docker run -d --tmpfs /app/temp nginx

3. Container Orchestration

For managing many containers across multiple machines.

Docker Swarm (Docker's built-in orchestration):

```
# Initialize a swarm
docker swarm init

# Deploy a service (replicated across the swarm)
docker service create --replicas 3 --name my-web nginx
```

Kubernetes Basics:

Kubernetes is more powerful than Docker Swarm but more complex. A simple example:

```
# Simple Kubernetes deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx:1.14
        ports:
        - containerPort: 80
```

4. Docker Health Checks

Health checks monitor if your container is working properly.

```
FROM nginx:alpine

# Add health check
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost/ || exit 1

# Rest of Dockerfile...
```

Practical Examples

Example 1: Simple Web Server

```
FROM nginx:alpine
COPY index.html /usr/share/nginx/html/
EXPOSE 80
```

Run with:

docker build -t my-website .
docker run -d -p 8080:80 my-website

Now visit http://localhost:8080 in your browser.

Example 2: Python Flask Web App with Redis

Docker Compose file:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - redis

  redis:
    image: redis:alpine
```

Dockerfile:

```
FROM python:3.9-slim

WORKDIR /app

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

COPY . .

EXPOSE 5000

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

app.py:

```
from flask import Flask
import redis

app = Flask(__name__)
r = redis.Redis(host='redis', port=6379)
```

```
@app.route('/')
def hello():
    r.incr('hits')
    return f'Hello World! I have seen {r.get("hits").decode("utf-8")} times.'

if __name__ == "__main__":
    app.run(host="0.0.0.0")
```

requirements.txt:

```
flask
redis
```

Run with:

docker-compose up

Best Practices for Beginners

1. **Start Simple**: Begin with single containers before multi-container apps
2. **Use Official Images**: Prefer official images from Docker Hub
3. **Keep Images Small**: Use Alpine-based images when possible
4. **One Service Per Container**: Each container should do one thing well
5. **Use .dockerignore**: Exclude unnecessary files from your builds
6. **Version Your Images**: Always tag your images (don't rely on 'latest')
7. **Read Logs**: Use docker logs to troubleshoot issues
8. **Clean Up Regularly**: Remove unused containers and images with docker system prune

Common Issues and Solutions

1. **Container won't start**
    - Check logs: docker logs container_id
    - Verify ports aren't already in use
2. **Can't connect to container**
    - Check port mappings with docker ps
    - Make sure the application is listening on 0.0.0.0, not just localhost

3. **Out of disk space**
    - Clean up with docker system prune -a
4. **Image build fails**
    - Check syntax in your Dockerfile
    - Verify that all files referenced in COPY commands exist

Next Steps

1. Learn more about Docker Compose for multi-container applications
2. Experiment with different programming languages and frameworks
3. Explore container orchestration with Docker Swarm or Kubernetes
4. Set up CI/CD pipelines with Docker
5. Learn about Docker security best practices

Remember: The best way to learn Docker is through practice. Start small, build confidence, and gradually tackle more complex scenarios.