Docker Lab 1 — Multi-Stage Docker Builds

Author: Dr. Sandeep Kumar Sharma

Lab Description

This lab introduces **Multi-Stage Docker Builds**, an advanced Docker image optimization technique used by professionals to build lightweight, secure, and production-ready images. Multi-stage builds allow you to use multiple `FROM` instructions in a single Dockerfile, helping you separate the build environment from the runtime environment.

In simple terms:

> Multi-stage builds let you compile heavy applications in one stage and copy only the required output into a smaller final image.

This keeps the final image **clean**, **small**, **secure**, and **fast to deploy**.

Topics Covered in This Lab

- What are multi-stage Docker builds?
- Why multi-stage builds are required?
- Benefits of multi-stage builds
- Anatomy of a multi-stage Dockerfile
- Hands-on: Build a Go/Node/Python sample app using multi-stage builds
- Comparing image sizes (normal build vs multi-stage build)

Learning Objectives

By the end of this lab, you will be able to: - Understand how multi-stage builds work internally - Write Dockerfiles using multiple stages - Reduce final image size drastically - Build production-ready images without exposing build tools or secrets - Improve build performance and maintainability

---

Learning Outcomes

After performing this lab, you will achieve the following: - Ability to create highly optimized Docker images - Ability to separate build and runtime environments - Ability to use `COPY --from=<stage>` effectively - Ability to follow industry best practices for image optimization

---

Section 1 — Understanding Multi-Stage Builds

A typical Dockerfile installs build tools (like compilers, SDKs, dependencies) along with the application runtime. This leads to: - Very large images - Security issues (extra packages) - Slow deployments - Hard-to-maintain Dockerfiles

Multi-stage builds solve this by using:

```
FROM builder-image AS build-stage
... compile or build ...

FROM runtime-image
COPY --from=build-stage /app/output /app
```

The first stage contains heavy tools; the final stage contains only the necessary output.

Section 2 — Hands-On Multi-Stage Build (Go Application Example)

We will build a simple Go application using multi-stage builds.

Step 1: Create a Project Directory

```
mkdir ~/docker-multistage-lab
cd ~/docker-multistage-lab
```

Step 2: Create a Simple Go Application

```
nano main.go
```

Paste:

```
package main
import "fmt"
func main() {
    fmt.Println("Hello from a Multi-Stage Docker Build!")
}
```

Step 3: Create the Multi-Stage Dockerfile

```
nano Dockerfile
```

Paste:

```
# Stage 1: Build the application
FROM golang:1.21 AS builder
WORKDIR /app
COPY . .
RUN go build -o helloapp main.go

# Stage 2: Create the final lightweight image
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/helloapp .
CMD ["./helloapp"]
```

Section 3 — Build the Image

```
docker build -t sandeep/multistage:v1 .
```

Expected output: - First stage compiles the Go app - Second stage copies the compiled output - Final image size should be very small

---

Section 4 — Run the Container

```
docker run --name sandeep-multistage-container sandeep/multistage:v1
```

Output:

```
Hello from a Multi-Stage Docker Build!
```

Section 5 — Compare Image Sizes

Run:

```
docker images | grep multistage
```

Observe that: - Standard Go image ~1GB - Final multi-stage image ~7MB (Alpine-based)

This demonstrates how multi-stage builds reduce image size drastically.

Section 6 — Cleanup

```
docker rm sandeep-multistage-container
docker rmi sandeep/multistage:v1
```

Section 7 — Practice Tasks

1. Modify the Dockerfile to use `scratch` instead of `alpine`.
2. Add environment variables and rebuild.
3. Create a multi-stage build for a Node.js application.
4. Create a multi-stage build for a Python application using `pyinstaller`.

Summary

- Multi-stage builds create efficient, production-ready Docker images.
- They reduce size, increase security, and improve performance.
- This method is used by all modern enterprise containerized applications.

You can now say: **“Create Lab 2”** and I will generate the next lab in a new canvas.