Docker Lab 2 — Advanced Image Caching, Layers & BuildKit

Author: Dr. Sandeep Kumar Sharma

Lab Description

This lab provides a deep exploration of **Docker image layers**, **caching**, and **BuildKit** — three essential components that every Docker expert must understand. Caching and layering determine the performance, rebuild time, security, and efficiency of Docker images.

BuildKit, the next-generation Docker build engine, dramatically improves build performance, parallelism, secret management, and caching mechanisms.

This lab blends theory + hands-on examples to give you a complete mastery over advanced image build optimization.

Topics Covered in This Lab

- How Docker image layers are created
- Union filesystem and layer ordering
- How caching works during `docker build`
- Cache invalidation rules
- Build context optimization
- BuildKit fundamentals
- Using BuildKit features (inline cache, mount=type=cache, build secrets)
- Benchmarking image builds

Learning Objectives

By the end of this lab, you will be able to: - Understand exactly how Docker layers are generated and stored - Predict when Docker will use cache vs rebuild a layer - Optimize Dockerfiles to reuse cache effectively - Reduce image build time and image size - Use BuildKit to enable advanced caching and secret handling

---

Learning Outcomes

After completing this lab, you will gain: - The ability to write Dockerfiles that build 10× faster - Understanding of how layer changes trigger cache invalidation - Hands-on experience using BuildKit features - Confidence to debug, optimize, and tune complex enterprise Docker builds

---

Section 1 — Understanding Docker Layers

Every instruction in a Dockerfile creates a new layer:

- `FROM` → base layer
- `RUN` → command layer
- `COPY` → filesystem layer
- `ENV` → metadata layer
- `CMD` / `ENTRYPOINT` → configuration layer

Layers are **immutable** and cached unless something changes.

View layers of an image:

```
docker history ubuntu:latest
```

Section 2 — Demonstration: Cache Invalidation

Create a directory:

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

Create a Dockerfile:

```
nano Dockerfile
```

Paste:

```
FROM ubuntu:22.04
RUN apt-get update
RUN apt-get install -y curl
```

```
COPY file1.txt /data/file1.txt
RUN echo "Build complete"
```

Create file:

```
echo "hello" > file1.txt
```

Build:

```
docker build -t cache-demo:v1 .
```

Build again:

```
docker build -t cache-demo:v1 .
```

You will notice caching is used.

Now modify file1.txt:

```
echo "new content" > file1.txt
```

Rebuild:

```
docker build -t cache-demo:v1 .
```

Layer after `COPY file1.txt` will rebuild.

Section 3 — Optimizing the Dockerfile for Better Caching

Bad Dockerfile:

```
RUN apt-get update && apt-get install -y curl vim git
COPY . /app
RUN make build
```

This destroys caching because COPY happens too early.

Optimized Dockerfile:

```
COPY go.mod go.sum /app/
RUN go mod download
COPY . /app
RUN go build -o app
```

This ensures dependencies are cached separately.

Section 4 — Introduction to BuildKit

Enable BuildKit globally:

```
export DOCKER_BUILDKIT=1
```

Test BuildKit:

```
docker build -t buildkit-test .
```

You will see a new, modern output format.

Section 5 — BuildKit Cache Mount Example

Create new folder:

```
mkdir ~/docker-buildkit-cache
cd ~/docker-buildkit-cache
```

Create Dockerfile:

```
nano Dockerfile
```

Paste:

```
# Use build cache for apt
# syntax=docker/dockerfile:1.3

FROM ubuntu:22.04
RUN --mount=type=cache,target=/var/cache/apt
    apt-get update && apt-get install -y curl
CMD ["curl", "--version"]
```

Build:

```
DOCKER_BUILDKIT=1 docker build -t buildkit-cache-demo .
```

Build again:

```
DOCKER_BUILDKIT=1 docker build -t buildkit-cache-demo .
```

Notice that apt cache is reused — build is much faster.

Section 6 — BuildKit Secrets Example

(Used for API keys, SSH keys, private repos)

Create a secret file:

```
echo "my_secret_password" > secret.txt
```

Create Dockerfile:

```
nano Dockerfile
```

Paste:

```
# syntax=docker/dockerfile:1.3
FROM alpine
```

```
RUN --mount=type=secret,id=mysecret
    sh -c "echo Secret inside container: $(cat /run/secrets/mysecret)"
CMD ["echo", "BuildKit secret demo"]
```

Build with secret:

```
DOCKER_BUILDKIT=1 docker build --secret id=mysecret,src=secret.txt -t secret-demo .
```

Secrets never appear in image layers.
This is a critical security feature.

Section 7 — Cleanup

```
docker rmi cache-demo:v1 buildkit-test buildkit-cache-demo secret-demo
```

Section 8 — Practice Tasks

1. Optimize a Python `requirements.txt` Dockerfile for caching.
2. Use `mount=type=cache` for npm or pip cache.
3. Benchmark build performance with and without BuildKit.
4. Create a BuildKit-based multi-stage image.

Summary

- Docker layers and caching dramatically affect performance.
- Cache invalidation rules depend on the order of Dockerfile instructions.
- BuildKit enhances builds with parallelism, caching, and secret handling.
- Professional Docker engineers rely heavily on BuildKit for production builds.

Lab 2 is complete. You may now say: **“Create Lab 3”**.