Docker on AWS EC2 — Overall Problem Statement

Status Completed

Timing Jun 30, 2025 to Sep 23, 2025

Owners Soumik Dutta

Welcome to the **Docker Lab Series** built for cloud-based, production-grade learning.
Every activity is performed on your own **AWS EC2 instance (Ubuntu 24.04 LTS)** and is automatically graded via secure SSH access.
By the end of this series, you will have a **holistic and in-depth understanding of Docker** — from foundational usage to advanced concepts and production best practices.

Activity 0: Environment Setup

🎯 **Objective:** Provision and configure an AWS EC2 instance as a dedicated environment for running Docker-based labs.

📝 **Tasks:**

1. **Launch EC2 instance** — Create an Ubuntu 24.04 LTS instance (`t3.medium`) with the required CPU, RAM, and storage specifications.

2. **Configure networking & SSH access** — Set up security groups, open required ports (SSH, HTTP), and ensure secure key-based login.
3. **Install Docker** — Use the provided setup script to install Docker Engine and configure non-root access.
4. **Verify installation** — Run `docker run hello-world` to confirm Docker is functional on the instance.

📁 **Folder:** `Activity0/`

Activity 1: Core Docker Operations

🎯 **Objective:** Gain hands-on experience with running containers.

📝 **Tasks:**

- **Run basic containers** — Launch `hello-world`, `nginx`, and `ubuntu` containers to understand image execution.
- **Use interactive mode** — Work inside containers using `docker run -it` and explore container environments.
- **Learn and use key CLI commands** — Practice `docker ps`, `stop`, `rm`, `logs`, `inspect` for container monitoring and management.
- **Run containers as non-root** — Improve security by configuring and verifying `non-root` user execution inside containers.
- **Image housekeeping** — Identify unused images and free disk space using `docker image prune` and `docker system prune`.

📁 **Folder:** `Activity1/`

Activity 2: Docker Image Lifecycle & Persistence

🎯 **Objective:** Learn to manage Docker images, control container runtime behavior, and ensure persistence of application data.

📝 **Tasks:**

- **Image management** — Pull and run common images (`mysql`, `redis`, `node`) to explore usage patterns.
- **Inspect and tag images** — Use `docker image inspect`, `docker history`, and `docker tag` to analyze and manage image metadata.
- **Save and load images** — Practice distributing images with `docker save` and `docker load`.
- **Commit container state** — Capture a modified container as a new image using `docker commit`.
- **Apply runtime controls** — Add `HEALTHCHECK` instructions, restrict CPU/memory with `--cpus` and `--memory`, and enforce `ulimits`.
- **File backup with docker cp** — Copy files between containers and the host.
- **Data persistence** — Create and use named `volumes`, `bind mounts`, and `tmpfs mounts` to preserve data across container restarts.

📁 **Folder:** `Activity2/`

Activity 3: Building and Publishing Custom Docker Images

🎯 **Objective:** Learn how to design a `Dockerfile` from scratch, containerize a real Flask application, and publish the final image to Docker Hub.

📝 **Tasks:**

- **Understand the app** — Explore a simple Python Flask API (`product-api-flask`) with endpoints like `/api/products`, `/api/products/1`, `/health`, and `/logo`.
- **Write a Dockerfile step by step** — Build up the image incrementally:
  - Choose a base image and configure the shell (`FROM`, `SHELL`).
  - Define environment variables and working directory (`ENV`, `WORKDIR`).
  - Install required system packages (`RUN`).
  - Copy application code and add assets (`COPY`, `ADD`).
  - Install Python dependencies with pip (`RUN pip install`).

    - Expose the application port, create non-root users, and set permissions (`EXPOSE`, `USER`, `chown`).
    - Add metadata labels (`LABEL`).
    - Implement health checks and runtime defaults (`HEALTHCHECK`, `ENTRYPOINT`, `CMD`).
- **Optimize with `.dockerignore`** — Use `.dockerignore` to exclude unnecessary files from the build context.
- **Build and test locally** — Compile the image on the EC2 instance, run the container, and test API endpoints from both EC2 (via `curl`) and your local browser using the EC2 public IP.
- **Image retagging** — Learn why and how to tag images properly before pushing to Docker Hub.
- **Publish to Docker Hub** — Log in via CLI, push your image to a public repository, and verify digest integrity.
- **Final verification** — Ensure the image runs correctly, endpoints respond as expected, health check passes, and Docker Hub repo shows the published image.
- **Cleanup** — Stop containers, remove images, prune cache, and terminate the EC2 instance when done.

📁 **Folder:** `Activity3/`

Activity 4: Networking in Docker (without Compose)

🎯 **Objective:** Explore how containers communicate and integrate using Docker's native networking features — without relying on Compose.

📝 **Tasks:**

- **Default bridge network** — Launch multiple containers (e.g., `nginx` + `alpine` curl client) and test connectivity.
- **User-defined bridge networks** — Create a custom network, attach containers, and resolve names automatically.
- **Multi-container integration** — Run `mysql` and `phpmyadmin` as separate containers and connect them manually using networks and environment variables.

- **Debug connectivity** — Use `ping`, `curl`, and `docker logs` to troubleshoot inter-container communication.
- **Inspect networks** — Use `docker network inspect`, `docker events`, and `DNS` resolution to analyze connected containers.
- **Port publishing** — Understand `-p` vs `EXPOSE` and map container ports to host.
- **Experiment with none/host networks** — Learn special modes and their implications.

📁 **Folder:** `Activity4/`

Activity 5: Multi-Container Applications with Compose

🎯 **Objective:** Learn step-by-step how to containerize and orchestrate a **3-tier demo application** (React frontend, Spring Boot backend, PostgreSQL database) using Docker Compose.

📝 **Tasks:**

- **Service containers** — Containerize frontend (React + Nginx), backend (Spring Boot + JRE), and database (Postgres).
- **Compose basics** — Build each service, then integrate them in a single `docker-compose.yml`.
- **Environment configs** — Externalize DB credentials and API base URL with `.env` files.
- **Persistence & logs** — Use named volumes for Postgres data and backend logs.
- **Healthchecks & dependencies** — Ensure DB, backend, and frontend start in correct order.
- **Scaling & overrides** — Scale backend replicas and manage dev vs prod configs with override files.

📁 **Folder:** `Activity5/`

Activity 6: Make Dockerfiles Production Ready

🎯 **Objective:** Learn techniques to build smaller, faster, and more secure images using advanced Dockerfile patterns.

📝 **Tasks:**

- **Multi-stage builds** — Build a Go application using a builder image, then copy only the binary to a slim final image.
- **Scratch images** — Use `FROM scratch` to create minimal images.
- **Slimming images** — Compare `ubuntu`, `alpine`, and `distroless` base images.
- **Build cache optimization** — Use `--mount=type=cache` to cache dependencies (Go modules, pip, npm).
- **Layer ordering** — Reorder instructions to maximize cache reuse.
- **Reduce attack surface** — Install only minimal dependencies, drop root privileges, add labels and healthchecks.
- **Security scanning** — Use `docker scan` or `trivy` for vulnerability checks.
- **Benchmarking** — Compare build times and image sizes before vs after optimization.

📁 **Folder:** `Activity6/`

Activity 7: Advanced Docker Concepts (Bonus)

🎯 **Objective:** Deep dive into advanced topics for students who want to go beyond everyday Docker usage.

📝 **Topics:**

- **Docker Daemon & API** — How `dockerd` works, socket communication, and using the Docker Remote API (`curl` against `/var/run/docker.sock`).
- **Container internals** — Namespaces, cgroups, capabilities, and Linux kernel integration.
- **Storage drivers** — AUFS, OverlayFS, device-mapper; inspect layers in `/var/lib/docker`.
- **Rootless Docker** — Running Docker entirely as non-root for extra security.

- **OCI standards** — Docker vs containerd vs CRI-O.
- **Hands-on** — Explore `/proc`, `/sys/fs/cgroup`, `lsns`, and `unshare`; run `runc` directly to start containers.
- Sync for scanning images.

📁 **Folder:** `Activity7/`

═══════════ END OF DOCUMENT ═══════════