⬇️ Activity 00: Docker\_on\_AWS\_EC2



# **Docker on AWS EC2: Overall Problem Statement**

Status Completed

Timing Jun 30, 2025 to Sep 23, 2025

Owners [Soumik Dutta](mailto:23m0826@iitb.ac.in)

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.
* Docker init
* **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 ═══════════════

0️⃣ Activity 0: Environment Setup

# Activity 0: Environment Setup

## 🎯 Objective

Launch an AWS EC2 instance (Ubuntu 24.04 LTS) and install Docker so that all subsequent lab activities can be executed on this environment.

By the end of this activity you will have:

* A running Ubuntu VM on AWS EC2.
* Docker installed and verified.
* SSH access working from your lab environment.

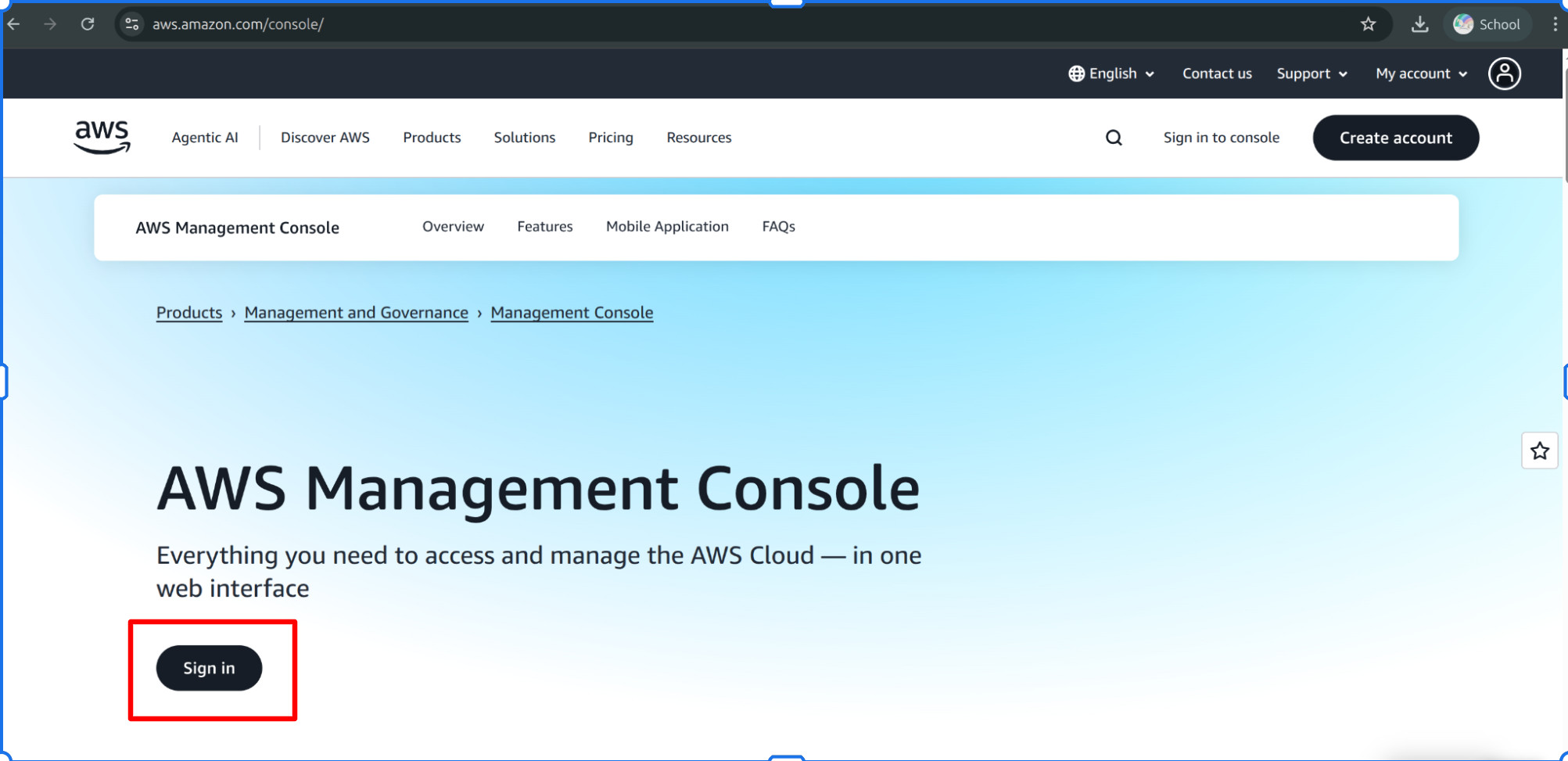
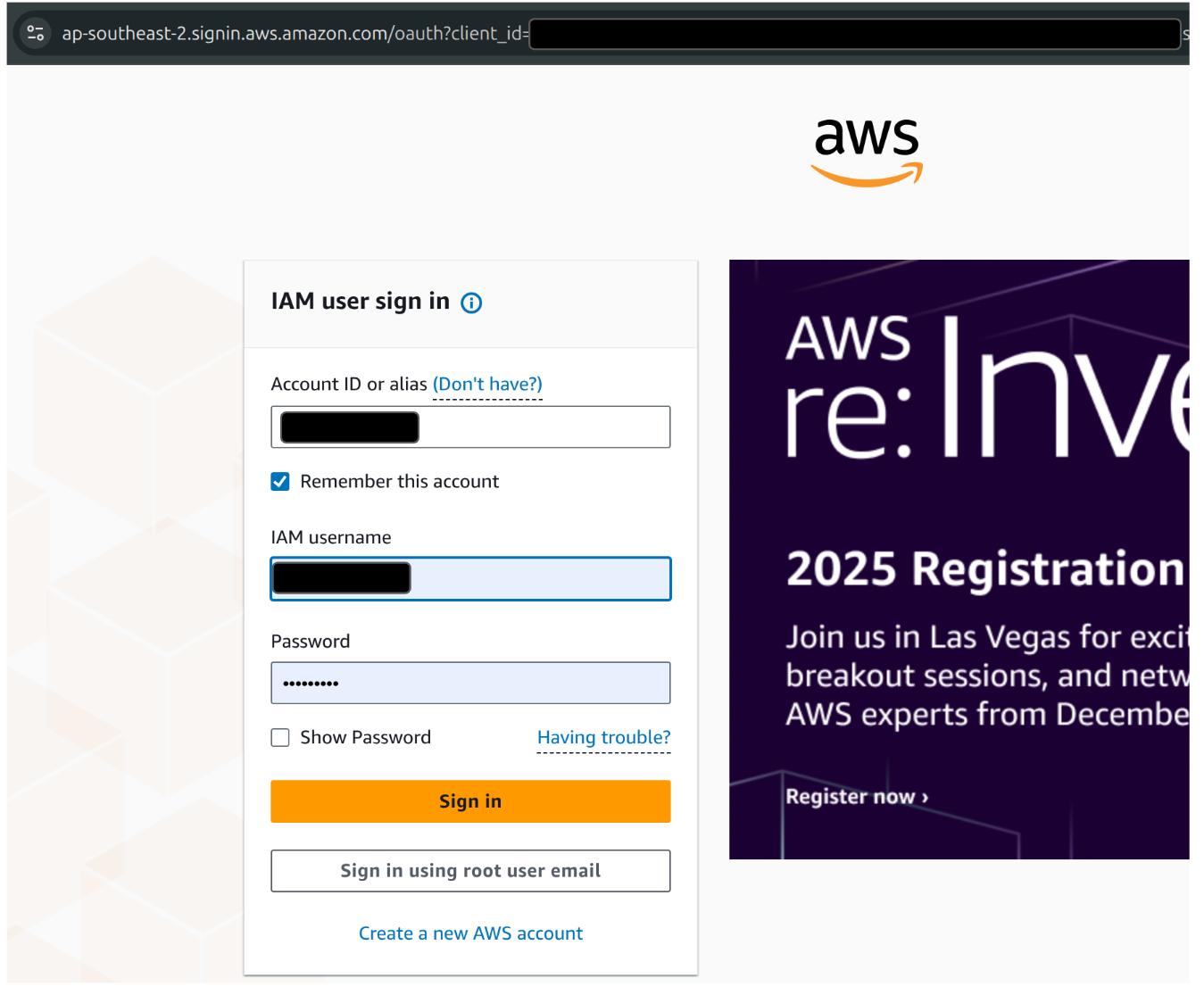
## ✅ Prerequisites

* An AWS account with EC2 permissions.
* Local SSH client (Linux/macOS: builtin; Windows: WSL, PowerShell + OpenSSH, or Git Bash).
* Minimum VM requirements: **2 vCPU**, **4 GiB RAM**, **20 GiB disk**.
* Recommended instance type: t3.medium (stop/terminate when not in use to avoid charges).

## 📜 Step-by-step Guidance

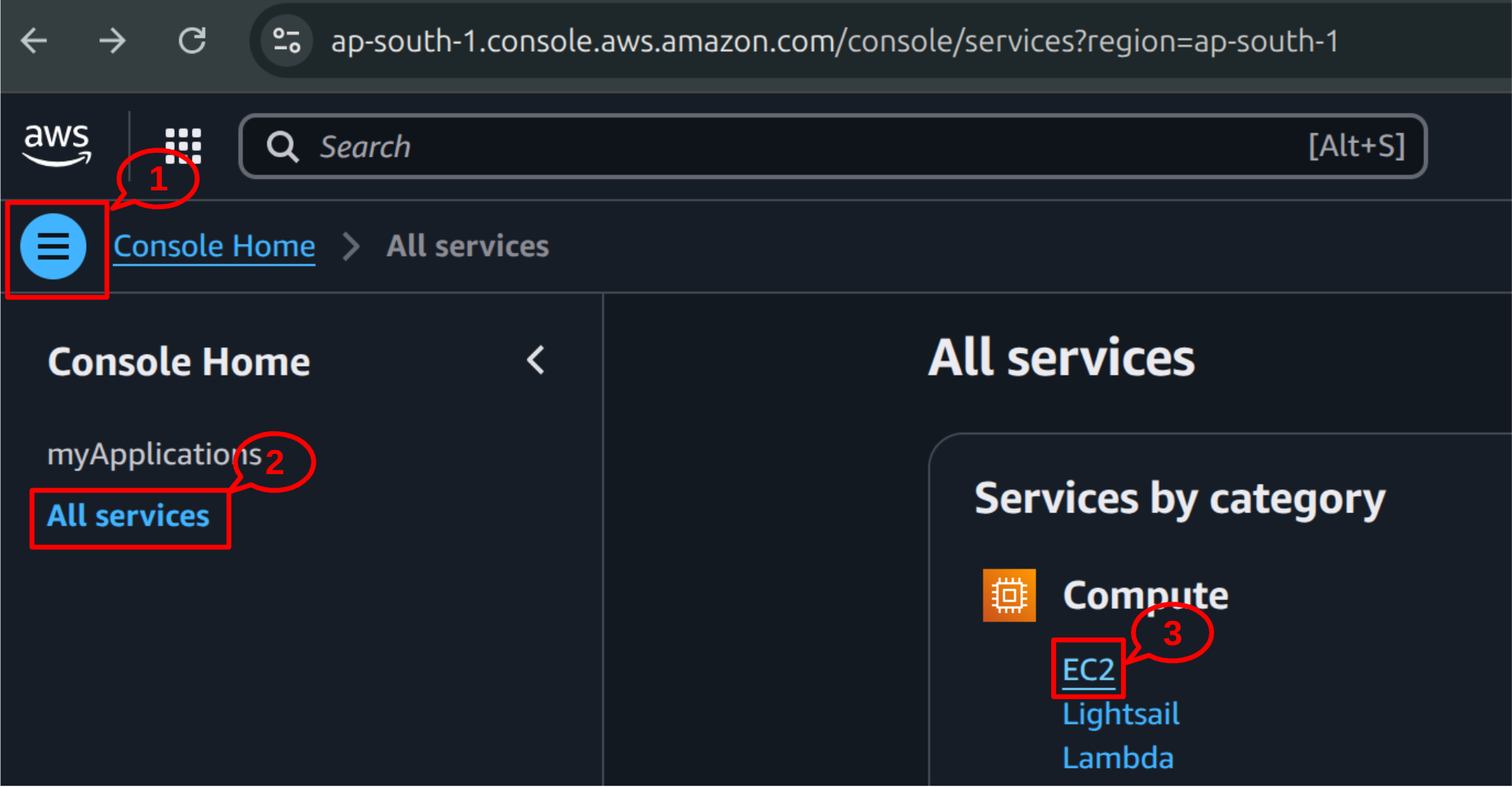
### Step 1: Sign in to AWS Management Console

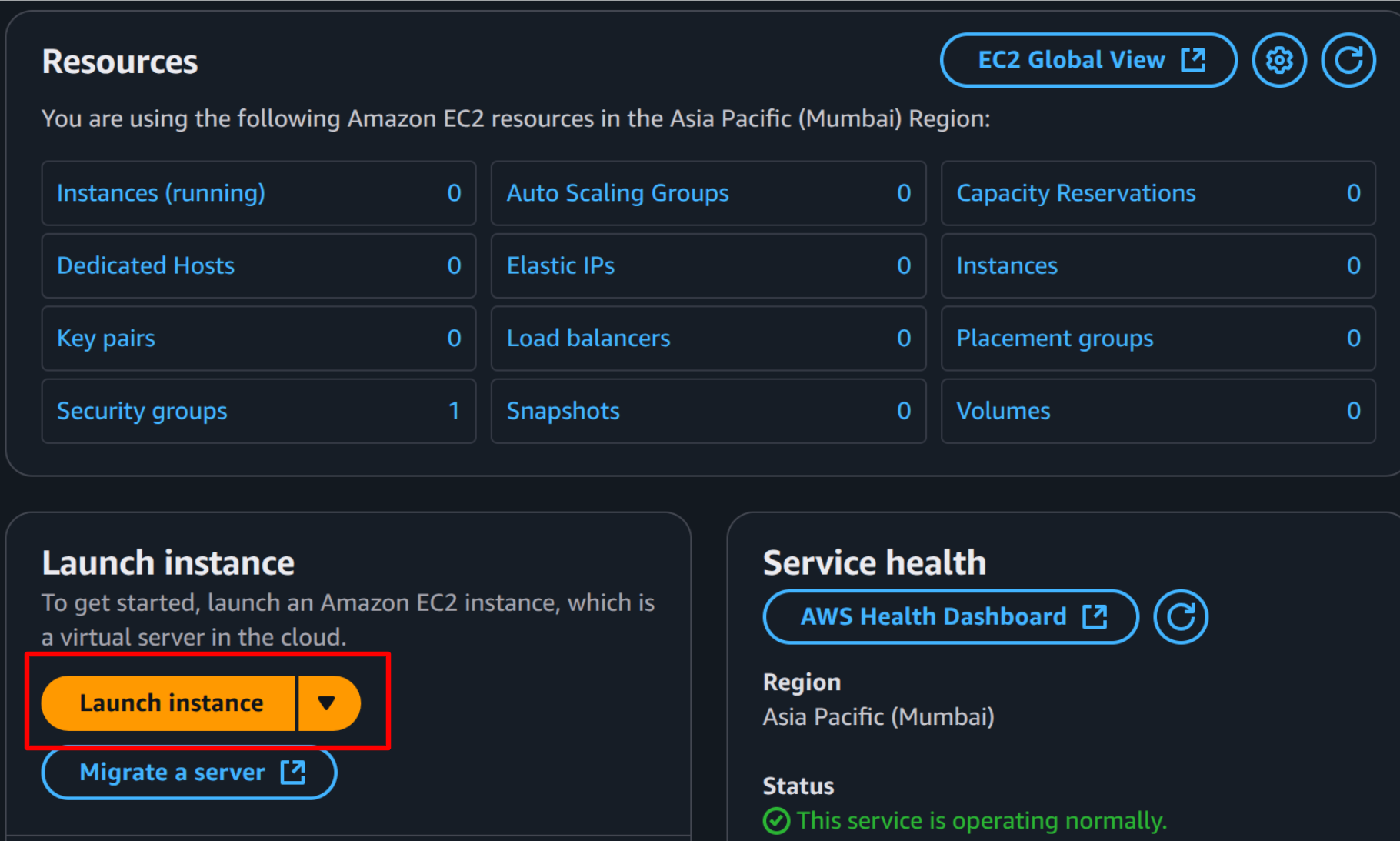
1. Open: <https://aws.amazon.com/console/>
2. Click **Sign in to the console** and log in with your AWS account.

### Step 2: Navigate to EC2 and Launch Instance

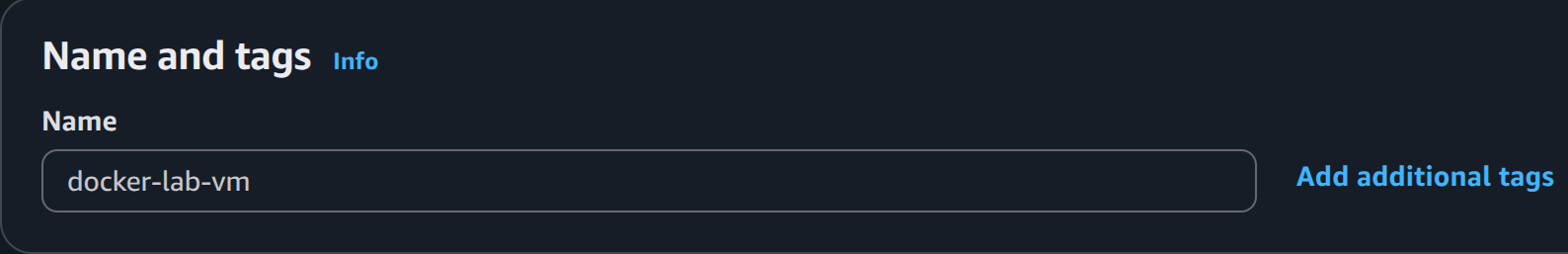
1. In the Console, go to **All Services → Compute → EC2**.
2. Click **Launch instance**.





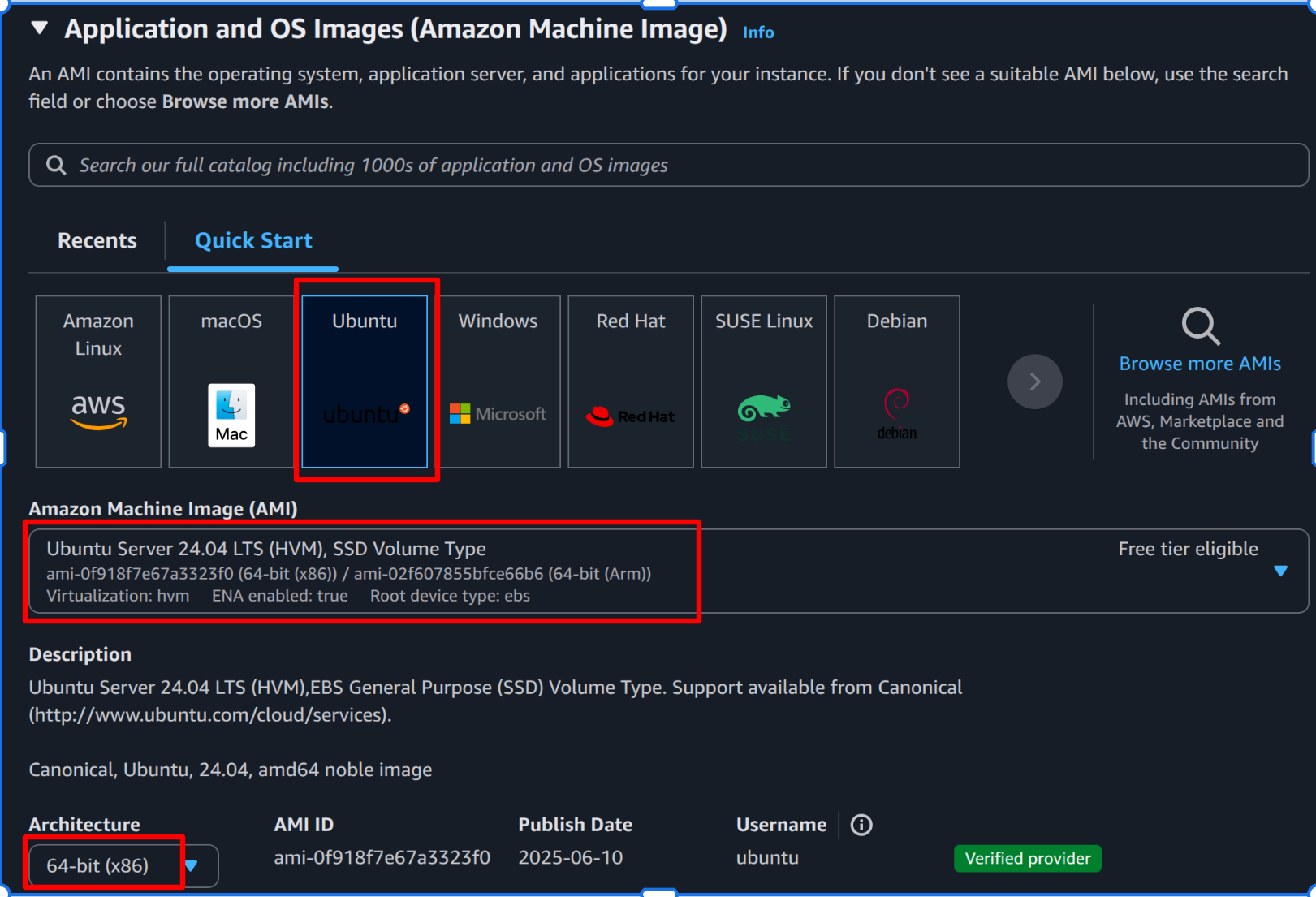
### Step 3: Name and Tags

1. In the **Name and Tags** section, set Name to something like: docker-lab-vm.
2. (Optional) Add additional tags for owner, course, etc.



### Step 4: Choose Amazon Machine Image (AMI)

1. Select **Ubuntu Server 24.04 LTS (HVM), SSD Volume Type**.
2. Make sure Architecture is selected as **64-bit(x86)**.



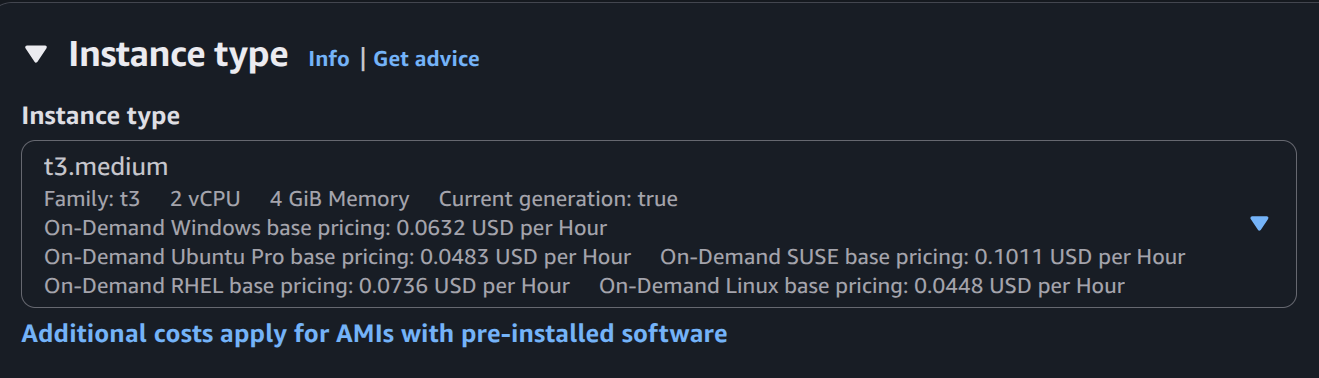
### Step 5: Choose Instance Type

1. Choose **t3.medium** (2 vCPU, 4 GiB RAM).

[*This choice ensures one has enough resources for the lab as mentioned in prerequisite.*]

⚠️ This instance type may incur charges. Stop/terminate when not required.

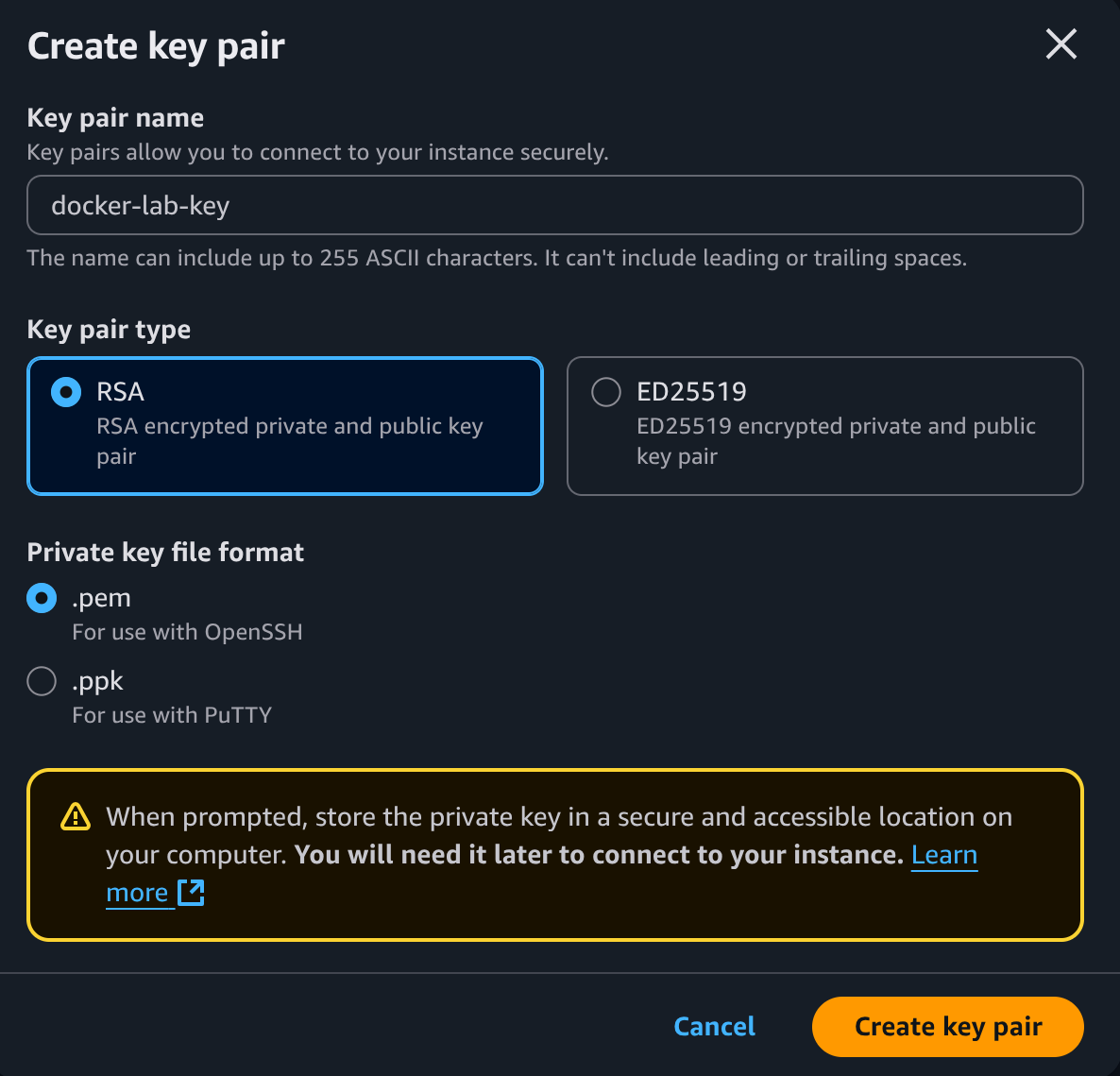
1. Prefer an equivalent Free-tier eligible AMI if available.



### Step 6: Create (or Use) Key Pair

1. Under **Key pair (login)** select **Create new key pair**.
2. Choose RSA and **.pem** format. Name it (e.g., docker-lab-key).
3. Click **Create key pair** and download the .pem file.

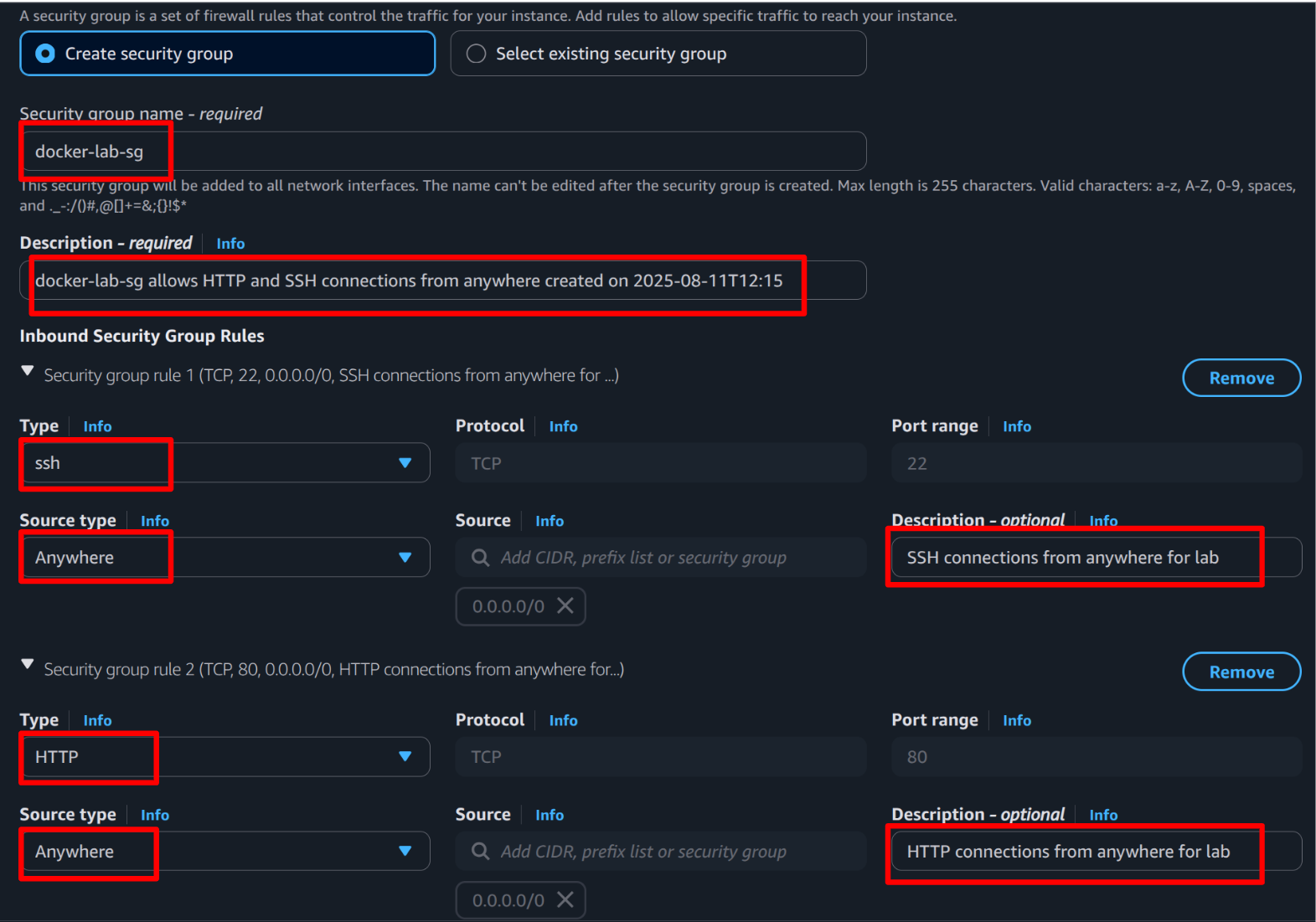
⚠️ Save this .pem file securely: this is your only download opportunity.



### Step 7: Configure Network / Security Group

1. Create a new security group (e.g., docker-lab-sg).
2. Add rules:
   * **SSH**: TCP 22: Source 0.0.0.0/0 (or restrict to your IP)
   * **HTTP**: TCP 80: Source 0.0.0.0/0   
     [*Here, we want to allow SSH and HTTP to the VM from* ***Anywhere***]
3. (Optional) Add additional rules as required by your lab.

⚠️ For real production or tighter security, restrict SSH to your IP only.

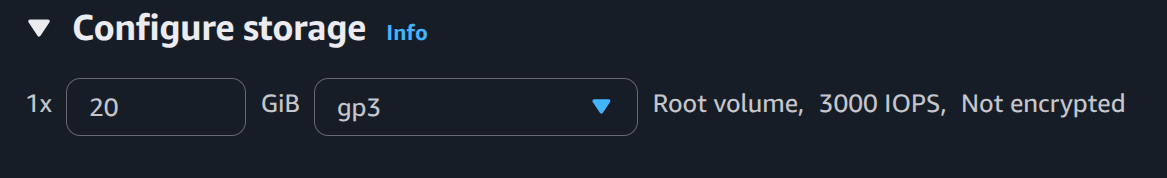


### Step 8: Configure Storage

1. Set root volume to **20 GiB** (**gp3**).

[*AWS EBS(Elastic Block Storage) gp3 volumes are generally cheaper and offer better performance than gp2 volumes*]

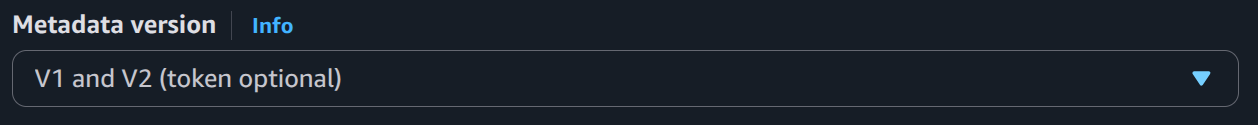
1. Confirm and proceed.



### Step 9: Make Metadata accessible

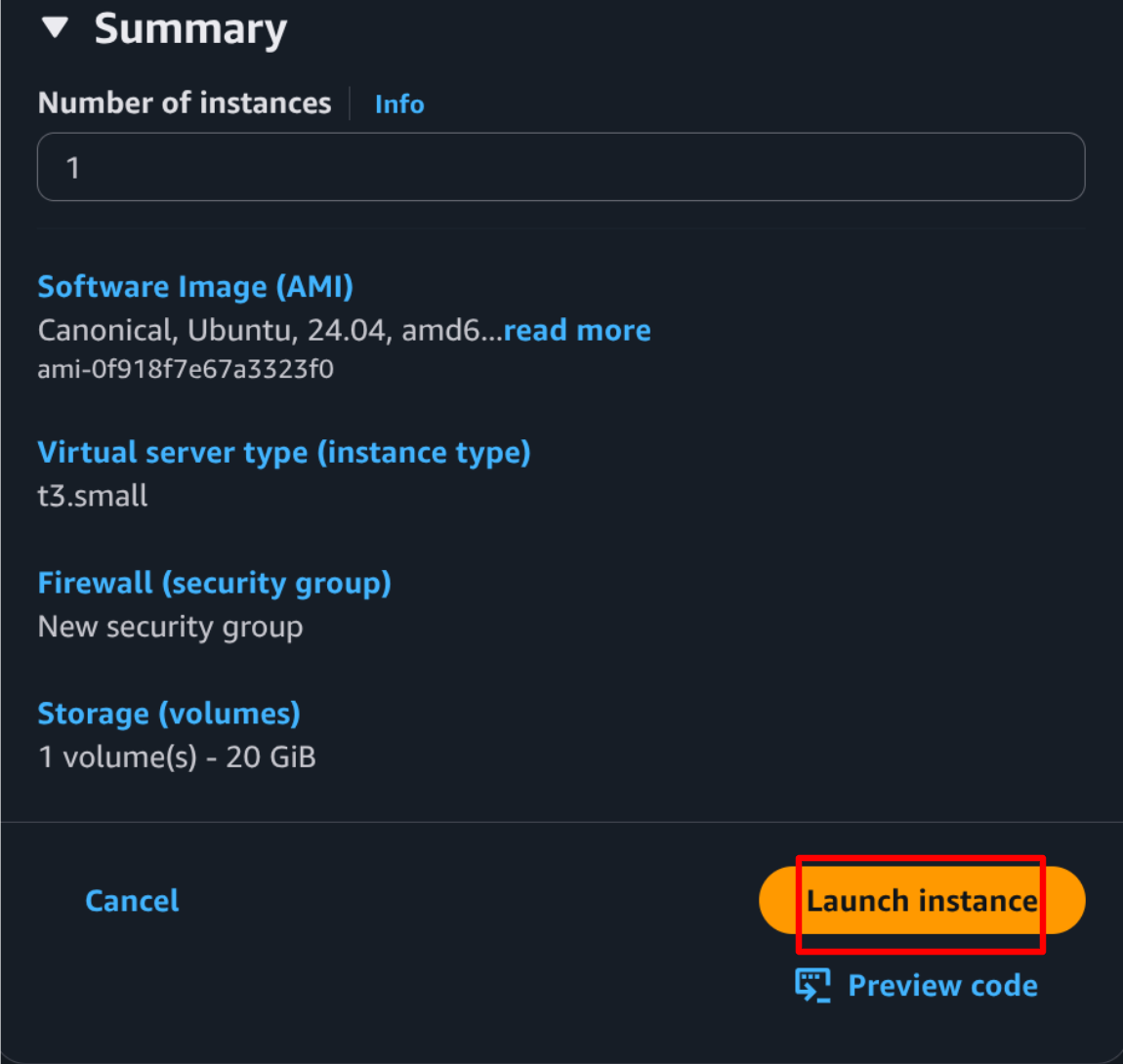
1. In the **Advanced details** Info select **Metadata Version**
2. Choose **V1 and V2 (token optional)**

[*V2**enforces token-based access for security where in lab setting we intentionally make token optional using V1 so that no need to attach a token with any ssh request into our ec2 instance*]



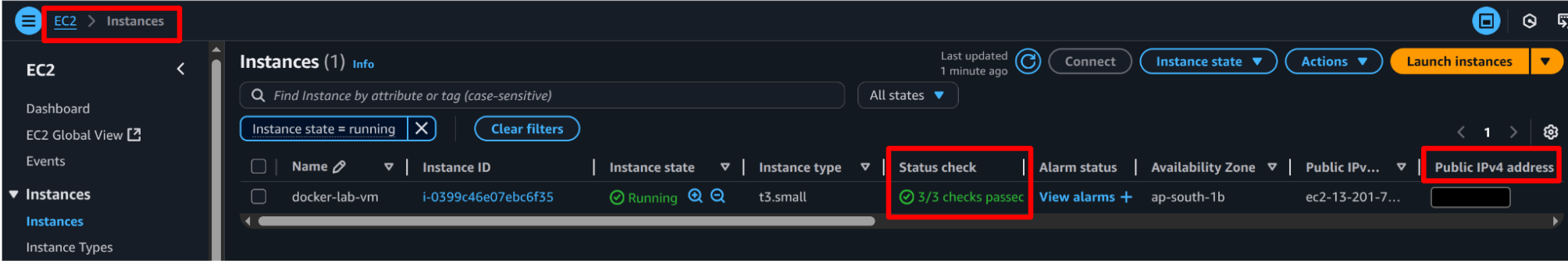
### Step 10: Launch Instance

1. Review configuration and click **Launch instance**.



### Step 11: Note Public IP

1. In **Instances** view wait until the instance state is **running** and status checks show **3/3 checks passed**.
2. In **EC2 → Instances**, copy the **Public IPv4 address** of your instance.



### Step 12: Prepare SSH Key Locally and Connect

Since cLab is a containerised application, direct file transfer from your local machine is limited. Follow these steps:

1. Open the **cLab** app and enter the current lab → current activity.
2. Click the **POP TERMINAL** button (opens a terminal at labDirectory).
3. Run:

chmod 770 secret-key.pem

1. Click **OPEN DIRECTORY** in cLab (opens labDirectory).
2. Open the downloaded .pem file (docker-lab-key.pem) from your local machine, copy its full content, and paste it into secret-key.pem inside labDirectory. Save the file.
3. Again, in the popped terminal, run:

chmod 400 secret-key.pem

* This ensures the private key has the correct permissions.

1. From the **popped terminal** in cLab (labDirectory), run:

ssh -i secret-key.pem ubuntu@<public-ip-address>

Replace <public-ip-address> with the value noted in **Step 11**.

### Step 13: Install Docker on the EC2 Instance

Step 13.0: Login to the EC2 instance (ubuntu)

ssh -i secret-key.pem ubuntu@<public-ip-address>



Step 13.1: Update Ubuntu packages and install prerequisites

* apt-transport-https : for https package retrieval
* curl : for data transfer
* conntrack : for network connection tracking

sudo apt-get update -y

sudo apt-get upgrade -y

sudo apt-get install -y apt-transport-https curl conntrack



Step 13.2: Download Docker installation script

curl -fsSL https://get.docker.com -o get-docker.sh



Step 13.3: Run the Docker installation script

sudo sh get-docker.sh



Step 13.4: Allow current user to run Docker without sudo

sudo usermod -aG docker $USER && newgrp docker



⚠️ If docker commands require sudo after usermod, **log out** and log back in or use newgrp docker to refresh group membership.

### Step 14: Verify Docker Installation

Check Docker version

docker --version



Expected output (sample):

Docker version 28.3.3, build 980b856



⚠️ If you see docker: command not found, the installation was unsuccessful.

### Step 15: Verify Docker with Hello World

docker run hello-world



Expected output (excerpt):

Hello from Docker!

This message shows that your installation appears to be working correctly.



### Step 16: Configure for Evaluation

* Open the data.json file in labDirectory.
* Paste your AWS EC2 **Public IPv4 address** into the appropriate field.
* Save the file.

## 🔍 Evaluation (what the grader checks)

The evaluation script will:

1. SSH into your EC2 instance using the provided public-ip and secret-key.pem.
2. Verify:
   * The AWS **EC2 instance** is set up correctly.
   * **Docker** is installed and configured correctly.

## 🧩 Notes & Tips

* Keep secret-key.pem private and do not share it publicly.
* If you use a restricted SSH source IP for better security, ensure the grader's IP (or the mechanism used by the autograder) can connect.

## 🛑 Stop / Terminate Instance (to avoid charges)

* After completing evaluation, it is important to stop the instance so that no extra costs are incurred.
* To **stop** the instance (preserve disk/data): In EC2 Console → select instance → **Instance state → Stop instance**.
* To **terminate** (delete resources): **Instance state → Terminate instance**.

## 🏆 Congratulations: You’ve completed Activity 0!

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

1️⃣ Activity 1: Core Docker Operations

# Activity 1: Core Docker Operations

## 🎯 Objective

Gain hands-on experience with running containers on your **AWS EC2 (Ubuntu 24.04 LTS)** instance prepared in Activity 0.

By the end of this activity, you should be able to:

* Run and inspect containers, images confidently.
* Work interactively inside containers.
* Run containers as non-root for better security.
* Keep the host clean using housekeeping commands.

## ✅ Prerequisites

* You have completed **Activity 0** and can SSH into the EC2 instance as ubuntu.
* Docker Engine is installed and you can run docker without sudo (user in docker group).

## **🐳 Task 1: Run Basic Containers**

**Goal:** Understand how images are executed and how containers behave in both **short-lived** and **long-running** scenarios.

**Do's:**

* Run a short-lived container (hello-world) and long-running ones (nginx, ubuntu).
* Assign **meaningful names** to all containers.
* Apply a consistent **label** to all containers for easy filtering.
* Practice running in **foreground** and **detached** modes.
* For the web server, **publish ports** so it is accessible externally.

### **Step-by-step actions**

#### **1. Run the short-lived container (hello-world)**

This tests image pulling and one-shot container execution with meaningful names.

* **Name:** lab1-hello
* **Label:** lab=act1

docker run --name <?> --label <?> hello-world

docker ps

docker logs <?>



**What this does & why:**

* docker run pulls the image if not available locally and runs it.
* --name <container-name> sets fixed container name (important for later inspection).
* --label <key>=<value> adds metadata for filtering.
* hello-world runs once and exits after printing a confirmation message.
* docker ps → lists only **running** containers.
* docker logs lets us view container logs.

#### **2. Run the long-running nginx service (Foreground vs Detached) with published port**

Start Nginx in detached mode, accessible on host port 8080.

* **Name:** lab1-nginx
* **Label:** lab=act1

**Foreground** example:

docker run --name <?> --label <?> -p <?> nginx:latest

  
**What this does & why:**

* -p <host-port>:<container-port> maps *container-port* to *host-port*
* Runs the latest Nginx web server image.
* Runs in the terminal, showing logs live.
* Useful for debugging and reading logs directly.
* Can be tested from another terminal by curl http://localhost:8080
* Stop with Ctrl+C.
* Not ideal for long running apps because it ties up your terminal.

**Detached** example (recommended for services):

docker run -d --name <?> --label <?> -p <?> nginx:latest

docker ps



**What this does & why:**

* **-d** runs the container in the **background**.
* Can be tested from the same terminal by curl http://localhost:8080
* With docker ps command verify the port mapping is done properly.
* Preferred for long-running or production-like scenarios.
* Standard for persistent services.

#### **3. Run an ubuntu container that stays alive (for interactive work)**

* **Name:** lab1-ubuntu
* **Label:** lab=act1

**Option A: short-lived keep-alive**

docker run -d --name <?> --label <?> ubuntu:latest sleep 300

docker ps

docker exec -it <?> bash

echo ok > /lab1.txt

exit

docker exec -it lab1-ubuntu cat /lab1.txt



* sleep 300 keeps container alive for 5 minutes.
* Allows temporary interactive work.
* docker exec runs a command inside an already running container without restarting it.
* -i (interactive) keeps STDIN open so you can provide input to the command.
* -t (tty) allocates a pseudo-terminal, making the session behave like a normal terminal.
* exec -it <container> bash opens Bash shell inside the running container.
* exec -it <container> cat /lab1.txt runs cat inside the container to display the contents of /lab1.txt on your *host’s* terminal.

**Option B: long-lived keep-alive (use for autograding)**

# Observe container 'lab1-ubuntu' exists

docker ps -a

# Remove it to create container with same name

docker rm lab1-ubuntu

docker run -d --name <?> --label <?> ubuntu:latest tail -f /dev/null

# Returns "No such file or directory"

docker exec -it lab1-ubuntu cat /lab1.txt docker exec <?> bash -c 'echo ok > /lab1.txt'

docker exec -it lab1-ubuntu cat /lab1.txt # Returns "ok"



* docker ps -a lists **all** containers, including stopped and exited.
* docker rm <container-name> → removes the container.
* tail -f /dev/null keeps container running indefinitely (common trick).
* Ideal for scenarios where the grader will connect later.
* bash -c '<command>' tells Bash inside container to execute the quoted command.

### **📌 Verification checklist for successful autograding**

* lab1-hello exists, exited successfully, logs contain the hello message.
* lab1-nginx is **Up** and serves HTTP on port 8080.
* lab1-ubuntu is **Up**, labeled lab=act1, and /lab1.txt contains ok.
* All containers have label lab=act1.

### **Useful commands for this task**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| docker run [OPTIONS] IMAGE [CMD] | Create and start a container | docker run --name lab1-hello --label lab=act1 hello-world |
| --name NAME | Assign fixed name | --name lab1-nginx |
| --label key=value | Add metadata | --label lab=act1 |
| -d / --detach | Run in background | docker run -d ... nginx |
| -p host:container | Publish container port | -p 8080:80 |
| docker ps / docker ps -a | List running / all containers | docker ps --filter "label=lab=act1" |
| docker logs CONTAINER | View container logs | docker logs lab1-hello |
| docker inspect CONTAINER | Detailed metadata | docker inspect lab1-nginx |
| docker exec -it CONTAINER CMD | Run inside container | docker exec -it lab1-ubuntu bash |
| docker port CONTAINER | Show port mappings | docker port lab1-nginx |
| docker stop / docker rm | Stop/remove container | docker stop lab1-nginx && docker rm lab1-nginx |
| curl URL | Test HTTP from host | curl -sI http://localhost:8080 |

## **🐳 Task 2: Master Key CLI for Lifecycle & Inspection**

**Goal:** Learn to list, filter, inspect containers/images in detail, as well as manage their lifecycle.

**Do's:**

* Use docker ps and docker images with **filters** and **custom formats** to extract exactly what you need.
* **Inspect** specific metadata fields using docker inspect w/o dumping the entire JSON.
* Explore container internals: **IP addresses**, **mounts**, **environment variables**, **commands**, **running processes**.
* Use docker top, docker stats to monitor live resource usage.

### **Step-by-step actions**

#### **1. List containers**

**Filter containers by label:**

docker ps -a --filter "label=lab=act1"



**Custom output format (name + status only):**

docker ps -a --format "table {{.Names}}\t{{.Status}}"



**Discover available fields for formatting:**

docker ps --format '{{json .}}' | head -n 3 | jq .



**What this does & why:**

* --filter → narrow down results (e.g., by label, status, name).
* --format → output only the fields you care about (avoids clutter).
* --format '{{json .}}' | jq . → shows all available keys for custom formatting.
* -n 3 shows only top 3 containers’ details from the list of all containers.

**🖊️ Student Action:**

Find the **CreatedAt** value of container lab1-nginx and write it in ans.json as:

{ "lab1-nginx-CreatedAt": "<value>" }



#### **2. Manage images List all images:**

#### 

docker images

#### **Filter by repository name:**

#### 

docker images --filter=reference="nginx:\*"

#### **Custom format (repository + size):**

#### 

docker images --format "table {{.Repository}}\t{{.Size}}"

#### **Discover available fields for images:**

#### 

docker images --format '{{json .}}' | head -n 3 | jq .

#### **Remove unused image by ID:**

#### 

docker rmi <image-id>

#### **What this does & why:**

#### docker images → list local images.

#### --filter=reference → match specific repository/tags.

#### docker rmi → removes image from local cache (only if unused).

#### --format key discovery helps automate reporting.

#### **🖊️ Student Action:** Find the **Size** of the nginx image and append it to ans.json:

#### 

{ "lab1-nginx-size": "<value>" }

#### 

#### **3. Inspect container metadata (targeted fields)**

* **Name:** lab1-ubuntu (from Task 1)

**Inspect all metadata (JSON output):**

docker inspect lab1-ubuntu



**Get container’s IP address only:**

docker inspect --format '{{ .NetworkSettings.IPAddress }}' lab1-ubuntu



**Get container’s mount points:**

docker inspect --format '{{ json .Mounts }}' lab1-ubuntu | jq



**Get container’s command:**

docker inspect --format '{{ .Config.Cmd }}' lab1-ubuntu



**What this does & why:**

* docker inspect → complete metadata in JSON.
* --format '{{ ... }}' → extract only the specific field you want (Go syntax).
* Useful for automation, scripts, and clean CLI outputs.

**🖊️ Student Action:** Record the **Image Sha256** of lab1-ubuntu in ans.json:

{ "lab1-ubuntu-sha256": "<value>" }



#### **4. Inspect runtime environment**

**List processes inside a running container:**

docker top lab1-ubuntu



**View environment variables:**

docker inspect --format '{{ json .Config.Env }}' lab1-ubuntu | jq



**View container filesystem layout from host:**

docker inspect --format '{{ .GraphDriver.Data.MergedDir }}' lab1-ubuntu



**Monitor resource usage (live stats):**

docker stats lab1-ubuntu



**What this does & why:**

* docker top → shows running processes inside the container.
* .Config.Env → retrieves environment variables at start.
* .GraphDriver.Data.MergedDir → shows where the container filesystem is mounted on host.
* docker stats → real-time CPU, memory, and network usage monitoring.

**🖊️ Student Action:** Find the value of the environment variable PATH from lab1-ubuntu and save to ans.json:

{ "lab1-ubuntu-path": "<value>" }



### **📌 Verification checklist for successful autograding**

* Listed lab1-nginx-CreatedAt, lab1-nginx-size, lab1-ubuntu-sha256, lab1-ubuntu-path values in ans.json.

### **Useful commands for this task**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| docker ps / docker ps -a | List running / all containers | docker ps -a --filter "label=lab=act1" |
| docker images | List images | docker images --filter=reference="nginx:\*" |
| --filter key=value | Narrow listing | --filter "name=lab1" |
| --format '{{...}}' | Custom output fields | --format '{{.Names}}' |
| docker inspect CONTAINER | View container metadata | docker inspect lab1-ubuntu |
| docker top CONTAINER | Show running processes | docker top lab1-ubuntu |
| docker stats [NAME] | Live resource usage | docker stats lab1-ubuntu |
| jq | Pretty-print JSON | ...|jq |

## **🐳 Task 3: Run Containers as Non-Root**

**Goal:** Improve security by avoiding the default root user inside containers.

**Do's:**

* Verify the running user inside a container.
* Use --user flag to specify a non-root UID/GID.
* Avoid granting unnecessary privileges to containers.
* Use official images that support non-root operation (or modify them if needed).

### **Step-by-step actions**

#### **1. Check the default user inside a container**

Run a temporary Ubuntu container and check the user:

docker run --rm ubuntu:latest whoami



**What this does & why:**

* whoami inside container prints the current user (usually root by default).
* --rm removes the container after it exits (no leftover).
* This helps confirm the container’s default privileges.

**🖊️ Student Action:** Record the **default user** for ubuntu:latest in ans.json:

{ "ubuntu-default-user": "<value>" }



#### **2. Run a container as a specific non-root user**

Run with UID 1000 (typical first non-root user on Linux):

docker run --rm \  
 --name lab1-nonroot --label lab=act1 \  
 --user 1000:1000 ubuntu:latest whoami



**What this does & why:**

* --user <UID>:<GID> sets the user and group inside the container.
* Prevents processes from having root privileges.
* Reduces risk if the container is compromised.

**🖊️ Student Action:** Record the **user** shown when running with UID 1000 in ans.json:

{ "lab1-nonroot-user": "<value>" }



#### **3. Verify user for an interactive container**

Start a container and verify UID/GID:

docker run -d --name lab1-nonroot-int --label lab=act1 \

--user 1000:1000 ubuntu:latest \

tail -f /dev/null

docker exec -it lab1-nonroot-int bash

whoami

id

exit



**What this does & why:**

* -it opens an interactive terminal inside the container.
* whoami shows the username (may show UID if no /etc/passwd entry).
* id shows the UID/GID explicitly.

**🖊️ Student Action:** Record the **UID** from id output in ans.json:

{ "lab1-nonroot-uid": "<value>" }



#### **4. Check processes and permissions**

From the **host**, verify the running container’s process owner:

# Expected: UID column should not show roo#

docker top lab1-nonroot-int



From inside the container verify the user’s permissions:

docker exec -it lab1-nonroot-int bash

ls -ld /root

touch /root/testfile # Expected: Permission denied



**What this does & why:**

* docker top shows **host-level** process info (here should be owned by the non-root UID).
* ls → list information
* -l → long format (permissions, owner, group, size, date, name)
* -d → show the directory entry itself, not what’s inside it
* touch create empty file
* Non-root user should not have permission to write to /root.
* Confirms reduced privileges are enforced.

**🖊️ Student Action:** If /root/testfile creation fails, autograder test case will pass.

### **📌 Verification checklist for successful autograding**

* ans.json contains:
  + ubuntu-default-user
  + lab1-nonroot-user
  + lab1-nonroot-uid
* Default userid for the lab1-nonroot-int container must be non-root.

### **Useful commands for this task**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| docker run --user UID:GID | Run as specific user/group | docker run --user 1000:1000 ... |
| whoami | Show current username | whoami |
| id | Show UID/GID | id |
| docker top CONTAINER | Show container processes from host | docker top lab1-nonroot |

## **🐳 Task 4: Image & System Housekeeping**

**Goal:** Keep your Docker environment clean by identifying and removing unused resources.

**Do's:**

* Identify **dangling** and **unused** images.
* Review disk usage for images, containers, and volumes.
* Use pruning commands carefully: know **exactly** what will be removed before running them.

### **Step-by-step actions**

#### **1. List all images & identify dangling ones**

docker image ls



* Shows all local images.
* **Dangling images**: untagged (<none>) images: often left behind after rebuilds.
* These can usually be removed without breaking anything.

#### **2. Review Docker disk usage**

docker system df



**What this does & why:**

* Shows how much space is used by images, containers, volumes, and build cache.
* Helps decide if cleanup is needed.

#### **3. Remove dangling images**

docker image prune



* Removes all dangling images (prompts for confirmation).
* Add --force to skip confirmation:

#### **4. Remove stopped containers**

docker container prune



* Deletes all stopped containers.
* Add --filter until=24h to remove only those older than 24 hours:

docker container prune --filter until=24h

  
❗Note: *If you deleted the hello-world container, you must recreate it before running all the test cases to ensure they pass.*

#### **5. Remove unused volumes**

docker volume prune



* Removes **unused** volumes (not referenced by any container).
* ⚠️ Be careful: this can delete persistent data.

#### **6. Full cleanup (images + containers + networks + build cache)**

docker system prune



* Removes all unused containers, networks, and dangling images.
* Add --volumes to also remove unused volumes:

docker system prune --volumes



* This removes all dangling **build cache**:

docker buildx prune



### **📌 Verification checklist for successful autograding**

* After cleanup, docker image ls should **not** list any <none> images.
* docker system df should show reduced usage compared to Step 2.

### **Useful commands for this task**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| docker image ls | List local images | docker image ls |
| docker system df | Show disk usage | docker system df |
| docker image prune | Remove dangling images | docker image prune --force |
| docker container prune | Remove stopped containers | docker container prune --filter until=24h |
| docker volume prune | Remove unused volumes | docker volume prune |
| docker system prune | Full cleanup | docker system prune --volumes |

## **📝 Final Verification Checklist for successful autograding**

**Task 1: Basic Containers**

* lab1-hello: exists, exited successfully, logs show hello message.
* lab1-nginx: **Up**, serving HTTP on port 8080, labeled lab=act1.
* lab1-ubuntu: **Up**, labeled lab=act1, /lab1.txt contains ok.

**Task 2: CLI & Inspection**

* ans.json contains:
  + lab1-nginx-CreatedAt
  + lab1-nginx-size
  + lab1-ubuntu-sha256
  + lab1-ubuntu-path

**Task 3: Non-Root Containers**

* ans.json contains:
  + ubuntu-default-user
  + lab1-nonroot-user
  + lab1-nonroot-uid
* lab1-nonroot-int runs as non-root, /root/testfile cannot be created.

**Task 4: Housekeeping**

* No <none> images remain (docker image ls).

## 🧹 Cleanup After the Activity

Stop/remove any containers you no longer need and reclaim space using the **housekeeping** tools you explored in **Task 4**.

**Do's:**

* Prefer **stop** over **kill** for a graceful shutdown.
* Always prune resources you no longer need to keep the host clean.

#### **5. Stop and remove containers (lifecycle control)**

**Gracefully stop a running container:**

docker stop lab1-ubuntu



**Force kill (immediate termination):**

docker kill lab1-ubuntu



**Remove a stopped container:**

docker rm lab1-ubuntu



**What this does & why:**

* stop → sends **SIGTERM**, allowing graceful shutdown.
* kill → sends **SIGKILL**, forcing immediate exit (use only if stop fails).
* Removing stopped containers frees up space and avoids clutter.

### **Useful commands for this task**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| docker stop CONTAINER | Graceful stop | docker stop lab1-ubuntu |
| docker kill CONTAINER | Force stop | docker kill lab1-ubuntu |
| docker rm CONTAINER | Remove container | docker rm lab1-ubuntu |
| docker rmi IMAGE | Remove image | docker rmi nginx:latest |

## 🛑 Stop / Terminate Instance (to avoid charges)

* After completing evaluation, it is important to stop the instance so that no extra costs are incurred.
* To **stop** the instance (preserve disk/data): In EC2 Console → select instance → **Instance state → Stop instance**.
* To **terminate** (delete resources): **Instance state → Terminate instance**.

## 🏆 Congratulations: You’ve completed Activity 1!

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

2️⃣ Activity 2: Docker Image Lifecycle & Persistence

# Activity 2: Docker Image Lifecycle & Persistence

## 🎯 Objective

In this activity, you will learn how to:

* Manage Docker images using **inspect**, **history**, **save**/**load**, and **commit**.
* Control container runtime behavior with **health checks**, **CPU/memory limits**, and **ulimits**.
* Use **volumes**, **bind** mounts, and **tmpfs** to persist data.
* Demonstrate the real-world usefulness of **persistence** with a MySQL database.

By the end, you will have hands-on experience managing images, applying runtime constraints, and configuring persistence in containers.

## ✅ Prerequisites

* Completion of **Activity 1** (basic container operations).
* A running **Ubuntu 24.04 LTS EC2 instance** with Docker installed.
* SSH access to EC2 instance using public-ip and secret-key.pem provided by you.
* Internet access on the EC2 instance to pull images.

## **🐳 Task 1: Image Management & Lifecycle**

### **🎯 Goal**

Work with Docker images to pull, inspect, analyze history, save/load images, and commit container changes.

### **📋 Do’s**

* Always verify image metadata with inspect.
* Use history to understand image layering.
* Demonstrate save and load workflow.
* Show how container state can be committed into a new image.

### **🛠️ Step-by-Step**

### **1. Pull required images**

****docker pull mysql:latest

docker pull redis:latest

docker pull node:latest

**What this does & why:**

* docker pull downloads images from Docker Hub into the local repository.
* These images (MySQL, Redis, Node) represent real-world services commonly used in applications.
* Pulling ensures you can run containers from these images even without internet later.

🖊️ **Student Action:** Note down the **Image IDs** for mysql:latest and redis:latest into ans.json.

### **2. Inspect an image**

****docker image inspect node:latest

**What this does & why:**

* docker image inspect shows detailed metadata of an image in JSON format.
* Useful to view configuration like environment variables, entrypoints, exposed ports, architecture, and labels.

🖊️ **Student Action:** Extract the **Architecture** field value from the metadata of redis:latest image and record it in ans.json.

### **3. View history**

****docker history redis:latest

**What this does & why:**

* Displays the **layer history** of the image.
* Each line represents a layer created by a command in the Dockerfile (e.g., RUN apt-get install).
* Helps understand **image size contributors** and optimization opportunities.

🖊️ **Student Action:** Identify the **largest layer size** from the history output and note it in ans.json.

### **4. Save & load image**

****docker save -o /home/ubuntu/redis.tar redis:latest

docker rmi redis:latest

docker load -i /home/ubuntu/redis.tar

**What this does & why:**

* docker save exports an image as a .tar archive file.
* Useful for moving images between systems **without Docker Hub**.
* docker rmi removes the image locally.
* docker load restores the image from archive.

🖊️ **Student Action:**

* Record the **Node image ID** before save into ans.json.
* Ensure node.tar is created in /home/ubuntu.
* Confirm that the local repo no longer has node:latest after removal step.

### **5. Commit container state**

****docker run -it --name test-commit ubuntu:latest bash

echo "persistent note" > /note.txt

exit

docker commit test-commit ubuntu-committed:lab2

**What this does & why:**

* docker run -it starts an interactive Ubuntu container.
* We create a new file (/note.txt) inside the container.
* docker commit creates a new Docker image from a modified or running container. This command captures the **current state of a container**, including any changes made to its **filesystem**, and saves it as a new image.
* This demonstrates how changes inside a running container can be preserved.

🖊️ **Student Action:** Record the **SHA256 digest** of the committed image (ubuntu-committed:lab2) in ans.json.

### **✅ Verification Checklist**

* Pulled mysql:latest, redis:latest, node:latest and noted their Image IDs.
* Inspected redis:latest and recorded **Architecture**.
* Viewed history of redis:latest and noted **largest layer size**.
* Exported Node image → confirmed node.tar exists in /home/ubuntu.
* Removed Node image locally.
* Committed changes to Ubuntu container → recorded SHA256 digest.

{

"mysql-image-id": "sha256:<full\_sha256>",

"redis-image-id": "sha256:<full\_sha256>",

"redis-arch": "<x86\_64/arm64/amd64>",

"redis-largest-layer-size": "<x.yMB/x.yKB>",

"node-image-id-tar": "sha256:<full\_sha256>",

"ubuntu-commit-sha256": "sha256:<full\_sha256>"

}



### **📑 Useful Commands for this Task**

| **Purpose** | **Command** |
| --- | --- |
| Pull MySQL, Redis, Node images | docker pull mysql:latest redis:latest node:latest |
| Inspect image metadata | docker image inspect <image> |
| View image layer history | docker history <image> |
| Save image as tar | docker save -o <file>.tar <image> |
| Remove local image | docker rmi <image> |
| Load image from tar | docker load -i <file>.tar |
| Run container interactively | docker run -it --name <name> <image> bash |
| Commit container to new image | docker commit <container> <new-image> |

## 🐳 Task 2: Container Behavior & Runtime Controls

### 🎯 Goal

Learn how to control runtime behavior of containers with **health checks**, **resource constraints**, and **file backups**.

### 📋 Do’s

* Use HEALTHCHECK to monitor container state.
* Limit CPU and memory to simulate resource-constrained environments.
* Backup data from a container using docker cp.

### 🛠️ Step-by-Step

1. **Run Redis with a healthcheck**

docker run -d --name redis-health \

--health-cmd="redis-cli ping || exit 1" \

--health-interval=10s \

--health-retries=3 \

redis:latest

docker inspect --format='{{json .State.Health}}' redis-health

**What this does & why:**

* Adds a **healthcheck** to Redis so Docker can monitor its status.
* Docker periodically runs redis-cli ping. If it doesn’t return PONG, the container is marked **unhealthy**.
* This is essential for automated recovery in real-world systems (e.g., restart unhealthy containers).

🖊️ **Student Action:** Record the **health status** (healthy or unhealthy) of redis-health in ans.json.

1. **Apply resource limits**

docker run -d --name limited-redis --cpus="0.5" --memory="256m" redis:latest

docker inspect limited-redis

**What this does & why:**

* Restricts the container to **0.5 CPU** and **256 MB memory**.
* Prevents a single container from consuming all system resources.
* This is useful in multi-container workloads or shared environments.

**Other resources that can be limited with Docker:**

* --pids-limit → restrict number of processes.
* --blkio-weight → control disk I/O priority.
* --device-read-bps / --device-write-bps → limit block device read/write rates.

🖊️ **Student Action:** From the container inspect output, find and record the following for limited-redis: Memory value; NanoCpus value. Add them to ans.json.

1. **Apply ulimit restriction**

docker run -d --name limited-ulimit --ulimit nofile=1000:1000 redis:latest

docker inspect limited-ulimit

**What this does & why:**

* --ulimit sets **kernel resource limits** for containers.
* Here nofile=1000:1000 restricts max open files (soft=1000, hard=1000) per process.
* **Soft limit** = the **current limit** the process is allowed to reach.
* **Hard limit** = the **maximum ceiling** that the soft limit can be raised to (only by root or privileged processes).
* Ulimits prevent resource exhaustion by rogue applications.

**Meaning of --ulimit:**

* It controls system resource limits for processes inside a container.
* Flags supported include:
  + nofile → max open files.
  + nproc → max number of processes.
  + fsize → max file size.
  + cpu → max CPU time.
  + as → max virtual memory.
  + core → max core dump size.

🖊️ **Student Action:** Record the **nofile ulimit value** from container inspect in ans.json.

**🌟 Extra Note: Resource Limits vs Ulimits**

* **Resource Limits (--cpus, --memory, etc.)**
  + These control how much **system resources** (CPU, RAM, I/O) a container can consume.
  + Implemented via Linux **cgroups (control groups)**.
  + Ensures fair distribution of resources across containers.
  + Example: --memory=256m prevents a container from using more than 256MB RAM.
* **Ulimits (--ulimit nofile=1000:1000, etc.)**
  + These restrict **per-process limits** inside the container.
  + Implemented via **kernel-level limits** that apply to processes.
  + Prevents applications inside a container from exhausting OS-level resources.
  + Example: --ulimit nofile=1000:1000 restricts open file descriptors per process.

**Key Difference:**

* *Resource limits* → control **how much** system resources a container can take.
* *Ulimits* → control **how processes behave** inside a container.

Together, they form a **two-layer defense**:

1. **Container-level control** (don’t starve the host).
2. **Process-level safety** (don’t let one app inside the container break everything).

### **Comparison Table**

| Feature | --cpus, --memory, --pids-limit | --ulimit |
| --- | --- | --- |
| **Control Mechanism** | Linux Cgroups (Control Groups) | Linux Ulimits (User Limits) |
| **Resource Type** | CPU, Memory, PIDs | Process-level resources (CPU time, open files, process count, file size, etc.) |
| **Scope of Limit** | Applies to the entire container | Applies per-process within the container |
| **Example for CPU** | --cpus=1.5 (limits CPU core usage) | --ulimit cpu=10 (limits process runtime in seconds) |
| **Primary Use Case** | Managing system-wide resource allocation for containers to ensure fair usage and prevent host degradation. | Controlling individual process behavior to prevent resource leaks or denial-of-service from a misbehaving application. |

1. **Backup file from container**

docker exec limited-redis sh -c 'echo "backup data" > /data.txt'

docker cp limited-redis:/data.txt ./copied\_data.txt



**What this does & why:**

* docker exec creates a test file /data.txt inside the running container.
* docker cp copies that file from container to host.
* This is useful for **backups, debugging, or exporting logs/data** from containers without setting up volumes.

🖊️ **Student Action:** Verify the backed up file is present on the host at /home/ubuntu. Record the name of the file present on the host in ans.json.

### **✅ Verification Checklist**

* Redis container (redis-health) reports health status (healthy/unhealthy).
* limited-redis container created with CPU and memory restrictions.
* limited-ulimit container runs with nofile=1000 ulimit.
* File data.txt copied successfully to host.

{

"redis-health-status": "<value>",

"limited-redis-nanocpus": "<value>",

"limited-redis-memory": "<value>",

"limited-ulimit-nofile": "soft={<value>},hard={<value>}",

"backup-host-file-name": "<filename.ext>",

}



## 🐳 Task 3: Volumes & Persistence (MySQL Use Case)

### 🎯 Goal

Understand that containers are **ephemeral** by default (data is lost after container removal).

Learn how to persist data using **volumes**, and compare with **tmpfs** and **bind mounts**.

### 📋 Do’s

* Show that data does **not** persist without volumes.
* Use a **named volume** for MySQL data.
* Insert and query data from MySQL.
* Prove persistence after container recreation.
* Compare with **tmpfs** and **bind mounts**.

### 🛠️ Step-by-Step

1. **Start MySQL without a volume (non-persistent run)**

****docker run -d --name mysql-temp \

-e MYSQL\_ROOT\_PASSWORD=rootpass \

-e MYSQL\_DATABASE=school \

mysql:latest



docker exec -it mysql-temp \

mysql -uroot -prootpass \

-e "USE school; CREATE TABLE students (id INT, name VARCHAR(50)); INSERT INTO students VALUES (1, 'Eve');"



docker exec -it mysql-temp \

mysql -uroot -prootpass \

-e "SELECT \* FROM school.students;"



docker rm -f mysql-temp

Then start a **fresh MySQL container** (again without a volume):

docker run -d --name mysql-temp \

-e MYSQL\_ROOT\_PASSWORD=rootpass \

-e MYSQL\_DATABASE=school \

mysql:latest



docker exec -it mysql-temp \

mysql -uroot -prootpass \

-e "SHOW TABLES IN school;"

**What this shows & why:**

* After removing the container, the data is **gone**.
* Confirms that container filesystems are ephemeral.

🖊️ **Student Action:** After recreating the container without volume, run SHOW TABLES IN school; Record whether the students table was found (exists / notfound) in ans.json.

1. **Create a named volume**

****docker volume create mysql\_data

**What this does & why:**

* Creates a persistent Docker-managed storage location.
* Find the volume at host: sudo ls /var/lib/docker/volumes/
* Volumes survive container removal.

🖊️ **Student Action:** Record the **volume name** in ans.json.

1. **Run MySQL with the named volume**

****docker run -d --name mysql-persist \

-e MYSQL\_ROOT\_PASSWORD=rootpass \

-e MYSQL\_DATABASE=school \

-v mysql\_data:/var/lib/mysql \

mysql:latest

**What this does & why:**

* Runs MySQL with a named volume mysql\_data attached at container’s /var/lib/mysql.
* Ensures all database files are stored persistently.

🖊️ **Student Action:** Run docker inspect mysql-persist and record both: Mounts.Source (host **full** path of the volume); Mounts.Destination (container **full** path of the volume) in ans.json.

1. **Insert data into MySQL**

****docker exec -it mysql-persist \

mysql -uroot -prootpass \

-e "USE school; CREATE TABLE students (id INT, name VARCHAR(50)); INSERT INTO students VALUES (1, 'Alice'), (2, 'Bob');"



1. **Query the data**

****docker exec -it mysql-persist \

mysql -uroot -prootpass \

-e "SELECT \* FROM school.students;"

🖊️ **Student Action:** Run SELECT COUNT(\*) FROM school.students; and record the **row count** in ans.json.

1. **Test persistence with the volume**

****docker rm -f mysql-persist # Remove container

docker run -d --name mysql-persist \ # Re-run container

-e MYSQL\_ROOT\_PASSWORD=rootpass \

-e MYSQL\_DATABASE=school \

-v mysql\_data:/var/lib/mysql \

mysql:latest

docker exec -it mysql-persist \ # Check if data persists

mysql -uroot -prootpass \

-e "SELECT \* FROM school.students;"

**What this does & why:**

* Container removed, but volume kept.
* On restart, the data is still there → proves persistence.

🖊️ **Student Action:** After recreating the container with the same named volume, run the same row count query. Record whether the **row count before and after** is the same (same / different) in ans.json.

1. **Compare with tmpfs and bind mount**

**Tmpfs Mount (memory-only):**

****docker run -d --name mysql-tmpfs \  
 -e MYSQL\_ROOT\_PASSWORD=rootpass \  
 -e MYSQL\_DATABASE=school \  
 --mount type=tmpfs,target=/var/lib/mysql \  
 mysql:latest

 **What this does & why:**

* Stores MySQL data in **RAM** only (no disk write).
* Very fast, good for **caching or temporary workloads**.
* But all data is lost once the container stops/restarts.
* Not suitable for databases that require durability.

🖊️ **Student Action:** Insert rows into mysql-tmpfs, restart the container, then re-run the row count query. Record whether the **row count after restart** is 0 or >0 in ans.json.

**Bind Mount (host filesystem):**

****docker run -d --name mysql-bind \  
 -e MYSQL\_ROOT\_PASSWORD=rootpass \  
 -e MYSQL\_DATABASE=school \  
 -v /home/ubuntu/mysql\_data:/var/lib/mysql \  
 mysql:latest

 **What this does & why:**

* Maps a **host directory** into the container.
* Data is visible directly on the host filesystem (/home/ubuntu/mysql\_data).
* Useful when you need to inspect/edit files directly, or for development/debugging.
* But less portable → depends on host directory structure and permissions.

**🌟 Extra Note: Why Volumes are better (default choice for DBs):**

* Managed by Docker, independent of host filesystem structure.
* More portable across environments.
* Backed up/restored easily with docker volume commands.
* Recommended for **production databases** like MySQL/Postgres.

🖊️ **Student Action:**

* Record the **bind mount path** Mounts.Source; Mounts.Destination in ans.json.

### 

### **✅ Verification Checklist**

* Recorded **no-volume-case** indicating table existence status after recreation.
* Recorded **mysql-volume-name** for the created named volume.
* Recorded **mysql-named-volume-mount-source** and **mount-destination**.
* Recorded **mysql-row-count** before recreation.
* Recorded **mysql-row-count-persistence-check** after recreation..
* Recorded **mysql-tmpfs-row-count-after-restart** recorded after restarting.
* Recorded **mysql-bind-mount-source** and **mount-destination**.

****{

"no-volume-case": "exists/notfound",

"mysql-volume-name": "",

"mysql-named-volume-mount-source": "",

"mysql-named-volume-mount-destination": "",

"mysql-row-count": "",

"mysql-row-count-persistence-check": "same/different",

"mysql-tmpfs-row-count": "0/>0",

"mysql-bind-mount-source": "",

"mysql-bind-mount-destination": ""

}



## 📝 Final Verification Checklist

**Task 1: Basic Containers**

* Pulled mysql:latest, redis:latest, node:latest and noted their Image IDs.
* Inspected redis:latest and recorded **Architecture**.
* Viewed history of redis:latest and noted **largest layer size**.
* Exported Node image → confirmed node.tar exists in /home/ubuntu.
* Removed Node image locally.
* Committed changes to Ubuntu container → recorded SHA256 digest.

**Task 2: CLI & Inspection**

* Redis container (redis-health) reports health status (healthy/unhealthy).
* limited-redis container created with CPU and memory restrictions.
* limited-ulimit container runs with nofile=1000 ulimit.
* File data.txt copied successfully to host.

**Task 3: Non-Root Containers**

* Recorded **no-volume-case** indicating table existence status after recreation.
* Recorded **mysql-volume-name** for the created named volume.
* Recorded **mysql-named-volume-mount-source** and **mount-destination**.
* Recorded **mysql-row-count** before recreation.
* Recorded **mysql-row-count-persistence-check** after recreation..
* Recorded **mysql-tmpfs-row-count-after-restart** recorded after restarting.
* Recorded **mysql-bind-mount-source** and **mount-destination**.

## 🧹 Cleanup After the Activity

1. Stop and remove all containers/images:

docker ps -aq | xargs docker rm -f  
docker images -aq | xargs docker rmi -f



1. Remove/Prune all Caches (optional):

docker buildx prune -f  
docker image prune -a -f  
docker system prune -a --volumes -f



1. 🚨 Remove all files from EC2 instance if needed:

rm -rf ~/docker\_lab/\*



## 🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

exit



1. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

## 🏆 Congratulations: You’ve completed Activity 2!

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

3️⃣ Activity 3: Custom Docker Images

# Activity 3: Building and Publishing Custom Docker Images

🎯 Objective

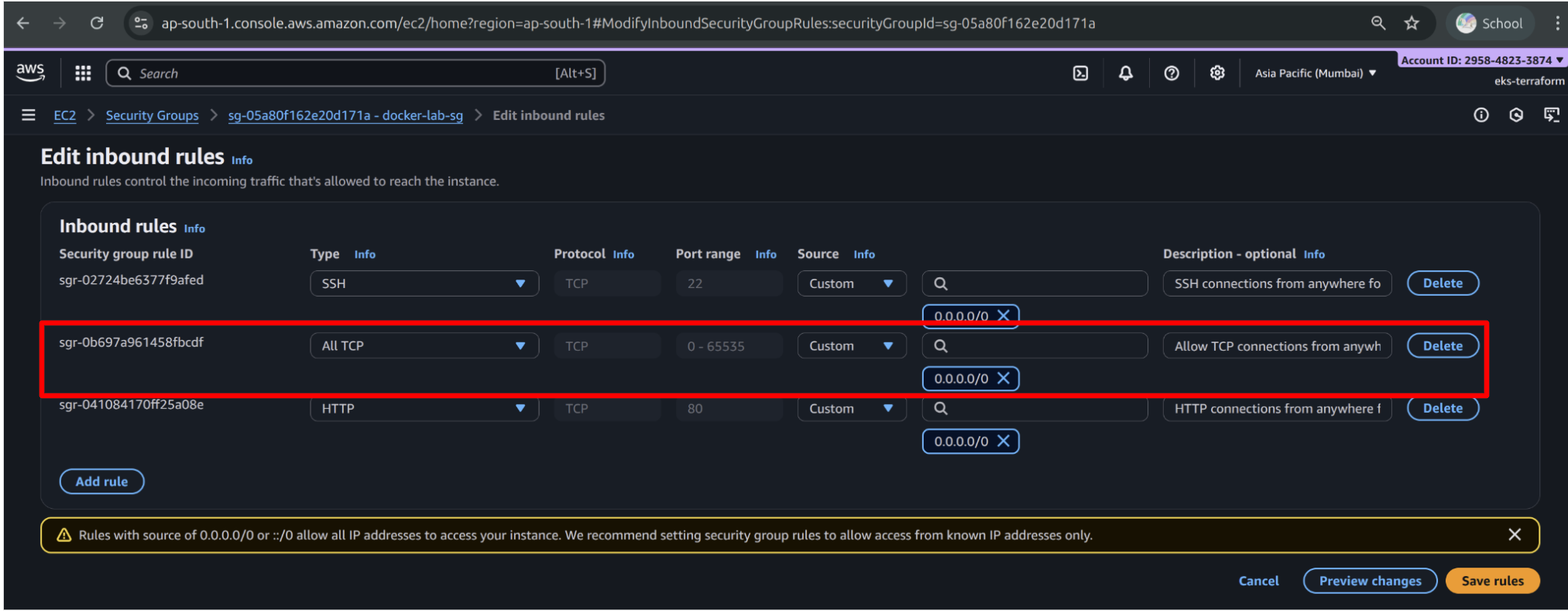
* Learn how to write **Dockerfile** to containerize an application.
* Understand a real **Flask** API’s basic endpoints so you know what you’re containerizing.
* Understand each instruction/flag and why we use it.
* Build, run and test **image** locally and then push to **Docker Hub** (public Docker repo).

✅ Prerequisites

* Completion of **Activity 2** (Docker Image Lifecycle & Persistence).
* A running **Ubuntu 24.04 LTS EC2 instance** with Docker installed.
* SSH access to EC2 instance using public-ip and secret-key.pem provided by you.
* Internet access on the EC2 instance to pull images.

📂 Lab Setup

* Configure **AWS EC2 security group**:
  1. Add another rule in docker-lab-sg to allow TCP inbound connections for our rest apps. For quick resolution allow all inbound TCP port range.



* Your lab directory contains two important files:
  1. product-api-flask.tar.xz → compressed archive of the Flask application that you will containerize.
  2. system-clean-init.sh → helper script that prepares your EC2 environment for this activity.

⚠️ **Important Note about system-clean-init.sh:** Running this script will **reset/clean** your EC2 instance environment and then **extract and copy** the Flask application into the correct working directory. Use it carefully, since it wipes existing Docker containers/images and resets the workspace. For more understanding go through the well commented script.

🛠️ Steps

1. Run the setup script:

bash system-clean-init.sh



1. Log in to your EC2 instance and move into the lab directory:

ssh -i docker-lab-key.pem ubuntu@<public-ip>

cd docker\_lab/product-api-flask/



## 📦 About the Application (what you’re containerizing)

This is a small **Python Flask** service. Key endpoints:

* GET /api/products/<int:product\_id>
* GET /api/products
* GET /health
* GET /logo

The app listens on port **5000** and uses standard Flask logging.

**Why this matters**: when you containerize, you must know **which port** to expose, what files must be present. For more info you can go through the product-api-flask folder provided in clab workspace and run the **Flask** application inside your **EC2 instance**.

📂 **Folder Structure**

$ pwd

/home/ubuntu/docker\_lab/product-api-flask

$ tree -a

.

├── .dockerignore

├── Dockerfile

├── requirements.txt

└── src

├── app.py

└── products.json

2 directories, 5 files



🛠️ **Run the Flask app on your EC2 instance (optional test)**

1. Update system and install Python:

sudo apt-get update

sudo apt-get install python3

sudo apt install python3.12-venv # to create isolated virtual environments



1. Create and activate a virtual environment:

python3 -m venv venv

source venv/bin/activate



1. Install pip and dependencies:

sudo apt-get install python3-pip

pip install -r requirements.txt



1. Run the application:

python3 src/app.py



1. Open your **local PC browser** and access the EC2 public IP to test endpoints:
   * http://<public-ip>:5000/api/products
   * http://<public-ip>:5000/api/products/1
   * http://<public-ip>:5000/health
   * **http://<public-ip>:5000/logo** **=> checkout this API just for fun!**

⚠️ **Drawbacks of this approach**

* Manual installation of Python, pip, and dependencies can be error-prone.
* Dependency conflicts may occur with system packages.
* Every developer must repeat the same steps on their EC2 instance or local machine.
* Difficult to ensure consistent runtime environments.

➡️ This is why we prefer **containerization**: one Docker image bundles the app and dependencies, ensuring **portability** and **reproducibility**.

🧹 **Cleanup Local Setup**

* Stop the running Flask app with Ctrl+C.
* Deactivate and remove the virtual environment:

deactivate

rm -rf venv



* Purge extra packages if desired:

sudo apt-get remove --purge python3-pip python3.12-venv -y

sudo apt-get autoremove -y



## 🧭 How we’ll learn in this activity

We will **start simple** and build our Dockerfile in small, understandable steps.  
Each step introduces one or two new Dockerfile instructions, explains their meaning, and shows why they are needed.

After each step you will:

1. **Update the Dockerfile** with the new instruction(s).
2. **Build the Docker image** with a new tag (e.g., lab3/stepX).
3. **Run the container** to test and verify its behavior.
4. **Record the outputs** in your ans.json file.

By the end of this activity you will arrive at the **final Dockerfile** version of the Flask API container.

We will cover:

* FROM and SHELL (base image and shell configuration).
* ENV and WORKDIR (environment variables and working directory).
* RUN (installing system dependencies).
* COPY and ADD (adding requirements and app source code).
* EXPOSE, USER, and LABEL (networking, security, and metadata).
* HEALTHCHECK, ENTRYPOINT, and CMD (making the container self-managed and runnable).

Each of these steps will help you understand **what the command does**, **why it is needed**, and how it contributes to making a portable, reliable Docker image.

## 📄 **The .dockerignore File**

When building Docker images, the **build context** (all files in your project directory) is sent to the Docker daemon.  
Unnecessary files in the context can **slow down builds**, **increase image size**, or even leak sensitive data into the image.

The .dockerignore file works just like a .gitignore: it tells Docker which files and folders to **exclude** from the build context.  
This ensures faster builds and cleaner, smaller images.

👉 **Why this matters:**

* Prevents copying of temporary or cache files.
* Avoids sending unnecessary large directories (like .venv) to Docker.
* Reduces security risks (e.g., excluding .git history or IDE configs).

🛠️ **Populate the .dockerignore file present in your project root (product-api-flask/)**

\_\_pycache\_\_/

\*.pyc

.venv/

venv/

.idea/

.vscode/

\*.log

\*.tar

\*.tar.gz

\*.zip

.git/



## 🐳 Task 1: Start from a base image & set the shell

🎯 Goal

* Choose a sensible **base image** (the starting filesystem for our container).
* Set the default shell for subsequent commands explicitly to **bash**.

📋 Explanation

* **FROM ubuntu:24.04** → Every Dockerfile begins with a FROM. This specifies the **base image** upon which your image layers will be built. We use **Ubuntu 24.04 LTS** because it is the same distribution as your EC2 instance and ensures long-term stability and compatibility.
* **SHELL ["/bin/bash", "-c"]** → By default, Docker executes commands with /bin/sh -c. Setting SHELL makes sure that all future RUN, CMD, and ENTRYPOINT instructions use **bash**, which supports advanced scripting features (e.g., &&, environment variable expansion).

🛠️ Steps

1. Open the existing Dockerfile (already present in your product-api-flask/ directory).
2. Append the following lines:

FROM ubuntu:24.04

SHELL ["/bin/bash", "-c"]



1. Build the image:

docker build -t lab3/step1-base-shell .



1. Verify by running a quick command:

docker run --rm -it lab3/step1-base-shell bash -lc "cat /etc/os-release | head -n1"



🖊️ Student Action (ans.json)

{

"lab3-base-image-PRETTY\_NAME": "<?>"

}



📌 Verification Checklist

* The image builds successfully without error.
* Running the container prints the Ubuntu release information.

## 🐳 Task 2: Define environment & working directory

🎯 Goal

* Use ENV to define environment variables for clarity and reuse.
* Use WORKDIR to set the default directory for subsequent instructions.

📋 Explanation

* **ENV APP\_HOME=/app** → Defines a variable APP\_HOME with value /app. This makes the Dockerfile easier to maintain because you can reuse ${APP\_HOME} instead of hardcoding paths.
* **ENV PORT=5000** → Defines the port on which the Flask app will run. Later, we will expose this port and use it in health checks.
* **WORKDIR ${APP\_HOME}** → Sets the working directory for all subsequent instructions (RUN, CMD, COPY, etc.). If the directory does not exist, Docker will create it automatically.

🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following lines:

ENV APP\_HOME=/app

ENV PORT=5000

WORKDIR ${APP\_HOME}



1. Build the image:

docker build -t lab3/step2-env-workdir .



1. Verify the working directory:

docker run --rm -it lab3/step2-env-workdir pwd



🖊️ Student Action (ans.json)

{

"lab3-app-home": "</?>",

"lab3-port": <?>

}



📌 Verification Checklist

* The image builds successfully without error.
* Running the container prints /app as the working directory.

## 🐳 Task 3: Install system packages (Python, pip, curl)

🎯 Goal

* Install the required system-level dependencies: **Python**, **pip**, and **curl**.
* Understand how to use RUN instructions to install packages inside the container.

📋 Explanation

* **RUN apt-get update** → Updates the package index inside the container so that we can install the latest available versions.
* **RUN apt-get install -y --no-install-recommends python3 python3-pip** → Installs Python3 and pip without extra recommended packages (keeps the image smaller). The -y flag automatically answers “yes” to prompts.
* **RUN apt-get install -y --no-install-recommends curl** → Installs curl, which we will later use for the container’s health check.

🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following lines:

RUN apt-get update

RUN apt-get install -y --no-install-recommends python3 python3-pip

RUN apt-get install -y --no-install-recommends curl



1. Build the image:

docker build -t lab3/step3-system-deps .



1. Verify the installations:

docker run --rm lab3/step3-system-deps python3 --version

docker run --rm lab3/step3-system-deps pip3 --version

docker run --rm lab3/step3-system-deps curl --version



🖊️ Student Action (ans.json)

{

"lab3-python-version": "<Python ?>",

"lab3-pip-version": "<pip ?>",

"lab3-curl-version": <curl ?>

}



📌 Verification Checklist

* The image builds successfully without error.
* Running the container shows the installed versions of Python and pip.
* Curl is available inside the container.

## 🐳 Task 4: Copy application code and add logo

🎯 Goal

* Copy the complete application source code into the container.
* Demonstrate the use of ADD to fetch a file from a remote URL.
* Understand the differences between COPY and ADD and when to use each.

📋 Explanation

* **COPY . .** → Copies everything from your current project directory (product-api-flask/) into the working directory inside the container (/app). This brings in the source code (src/), requirements.txt, and other files.
* **ADD <url> <destination>** → Unlike COPY, ADD can also:
  + Fetch files directly from remote URLs.
  + Automatically extract compressed archives (e.g., .tar.gz) into the destination directory.

🔍 **Comparison: COPY vs ADD**

* Use **COPY** when you only need to copy local files/directories into the image. This is the **recommended** and safer choice for most cases, as it is more explicit and predictable.
* Use **ADD** only when you need its extra features: fetching remote URLs or auto-extracting archives. Otherwise, stick to COPY.

🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following lines:

COPY . .

ADD <https://upload.wikimedia.org/wikipedia/sa/d/da/IIT_Bombay_logo.png> src/iit-bombay-logo.png



1. Build the image:

docker build -t lab3/step4-app-code .



1. Verify that the files exist inside the container:

docker run --rm -it lab3/step4-app-code ls -R /app | head



🖊️ Student Action (ans.json)

{

"lab3-app-files-present": <true/false>,

"lab3-logo-added": <true/false>

}



📌 Verification Checklist

* The image builds successfully without error.
* The application files (src/app.py, requirements.txt, etc.) are present inside /app.
* The logo file iit-bombay-logo.png is added under /app/src/.

## 🐳 Task 5: Install Python dependencies

🎯 Goal

* Install Python dependencies inside the container using pip.
* Understand why installing dependencies **after copying source code** may affect caching.

📋 Explanation

* **RUN pip install --no-cache-dir -r requirements.txt** → Installs all dependencies listed in the requirements.txt file.
  + The --no-cache-dir option tells pip not to store downloaded packages, keeping the image smaller.
  + The --break-system-packages flag is needed in containers to allow pip to override system-managed packages safely.
* 🔑 **Docker layer caching note:** Because we copied the whole directory in the previous step, any change in your source code will invalidate the cache for this step as well. This means dependencies will reinstall if any file changes. This is slightly less efficient than copying requirements.txt separately earlier, but we are keeping this sequencefor simplicity. Later we will discuss how to make these steps efficient.

🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following line:

RUN pip install --break-system-packages --no-cache-dir -r requirements.txt



1. Build the image:

docker build -t lab3/step5-install-deps .



1. Verify Flask is installed:

docker run --rm lab3/step5-install-deps python3 -c "import flask; print(flask.\_\_version\_\_)"



🖊️ Student Action (ans.json)

{

"lab3-flask-version": "<?>"

}



📌 Verification Checklist

* The image builds successfully without error.
* Flask and its dependencies install correctly.
* Running the container shows a valid Flask version number.

## 🐳 Task 6: Expose port, set permissions, and run as non-root user

🎯 Goal

* Make the container aware of the port on which the application runs.
* Create a dedicated non-root user and group with fixed IDs.
* Ensure correct permissions on the app directory.
* Improve security by running the application as a **non-root** user.

📋 Explanation

* **EXPOSE ${PORT}** → Documents that the container listens on the specified port (5000). This doesn’t publish the port itself but signals to others which port should be mapped.
* **RUN groupadd --gid 1001 appgroup && useradd --uid 1001 --gid 1001 -m appuser** → Creates a new group (appgroup) with group ID 1001 and a new user (appuser) with user ID 1001. Fixing IDs ensures consistency across environments. The -m flag creates a home directory for the user.
* **RUN chown -R appuser:nogroup ${APP\_HOME}** → Changes ownership of the application directory so the new appuser can access it.
* **USER appuser** → Ensures the container runs as a non-root user. This improves security because even if the container is compromised, the attacker has minimal privileges.

💡 **Note:** If the IDs 1001 already exist in your base image, you may get errors. In that case, run a temporary container with the base image and check existing users and groups:

docker run -it ubuntu:24.04 /bin/bash

cat /etc/passwd

cat /etc/group



🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following lines:

EXPOSE ${PORT}

RUN groupadd --gid 1001 appgroup && useradd --uid 1001 --gid 1001 -m appuser

RUN chown -R appuser:nogroup ${APP\_HOME}

USER appuser



1. Build the image:

docker build -t lab3/step6-nonroot .



1. Verify user and permissions:

docker run --rm lab3/step6-nonroot id



🖊️ Student Action (ans.json)

{

"lab3-exposed-port": <?>,

"lab3-user": "<?>",

"lab3-user-id": <100?>,

"lab3-group-id": <100?>

}



📌 Verification Checklist

* The image builds successfully without error.
* Running id inside the container shows the process is owned by appuser (UID 1001, GID 1001).
* Port 5000 is marked as exposed.

## 🐳 Task 7: Add metadata with labels

🎯 Goal

* Add descriptive metadata to the Docker image using LABEL.
* Understand how labels help with documentation, automation, and image management.

📋 Explanation

* **LABEL maintainer="<Your Name> <your email>"** → Specifies who maintains this image. Helpful for support and contact information.
* **LABEL version="1.0"** → Version number of the image/application. Useful for version tracking.
* **LABEL description="A simple Product Inventory API."** → Short description of what the image contains.

👉 Labels are stored as key-value pairs and can be queried using Docker commands. They follow the [OCI Image Specification](https://github.com/opencontainers/image-spec/blob/main/annotations.md), which defines standard label keys.

🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following lines (replace with your own details for maintainer):

LABEL maintainer="<Your Name> <your email>"

LABEL version="1.0"

LABEL description="A simple Product Inventory API."



1. Build the image:

docker build -t lab3/step7-labels .



1. Verify labels in the built image:

docker inspect lab3/step7-labels --format='{{json .Config.Labels}}' | jq



🖊️ Student Action (ans.json)

{

"lab3-version": "1.0"

}



📌 Verification Checklist

* The image builds successfully without error.
* Running docker inspect shows all three labels with correct values.

## 🐳 Task 8: Add healthcheck, entrypoint, and CMD

🎯 Goal

* Add a **healthcheck** to monitor the container’s runtime health.
* Define the container’s **entrypoint** (main executable).
* Provide default arguments with **CMD**.

📋 Explanation

* **HEALTHCHECK** → Defines a command that Docker runs periodically to test container health.
  + --interval=30s → run every 30 seconds.
  + --timeout=5s → fail if the command takes longer than 5 seconds.
  + --start-period=5s → give the container a short warm-up time before health checks begin.
  + CMD curl -f http://localhost:${PORT}/health || exit 1 → runs curl against the health endpoint; if it fails, container is marked unhealthy.
* **ENTRYPOINT ["python3"]** → Defines the main process that runs when the container starts. Here it always runs python3.
* **CMD ["src/app.py"]** → Provides default arguments to the entrypoint. Combined, the container runs. CMD arguments can be overridden at runtime if needed.

python3 src/app.py



🔍 **ENTRYPOINT vs CMD: When to use what**

* **ENTRYPOINT** → Use this when you want the container to *always* run a specific executable, regardless of user input. Example: in our case, we always want python3 to run.
* **CMD** → Use this for providing *default arguments* that can be overridden at runtime. Example: here app.py is the default script, but the student can override it by running:

docker run lab3/step8-runtime other\_script.py



* **Best practice** → Combine ENTRYPOINT (to fix the executable) with CMD (to provide flexible defaults).

🛠️ Steps

1. Open the existing Dockerfile in your product-api-flask/ directory.
2. Append the following lines:

HEALTHCHECK --interval=30s --timeout=5s --start-period=5s \

CMD curl -f http://localhost:${PORT}/health || exit 1

ENTRYPOINT ["python3"]

CMD ["src/app.py"]



1. Build the image:

docker build -t lab3/step8-runtime .



1. Run the container:

docker run --rm -p 5000:5000 lab3/step8-runtime



1. In another EC2 ssh terminal, test endpoints:

curl -s http://localhost:5000/api/products | head -n 5

curl -s -o /dev/null -w "%{http\_code}\\n" http://localhost:5000/health



1. From your local PC browser, use the EC2 public IP to check the endpoints:
   * http://<public-ip>:5000/api/products
   * http://<public-ip>:5000/api/products/1
   * http://<public-ip>:5000/health
   * **http://<public-ip>:5000/logo => check this API now. Can you spot the Logo??**
2. Inspect health status:

CONTAINER\_ID=$(docker ps -q --filter ancestor=lab3/step8-runtime)

docker inspect --format='{{json .State.Health}}' $CONTAINER\_ID | jq



🖊️ Student Action (ans.json)

{

"lab3-health-status": <?>,

"lab3-entrypoint": "<?>",

"lab3-cmd": "<?>"

"logo-spotted": <true/false>

}



📌 Verification Checklist

* The image builds successfully without error.
* Running container serves the API endpoints correctly.
* /health endpoint returns HTTP 200.
* docker inspect shows container health status as healthy.
* You understand the distinction between ENTRYPOINT and CMD, and when to use each.
* You can see the IIT Bombay logo in browser.

## 🏁 The Final Dockerfile

**Verify** that you have finally came up with the below dockerfile.

# Start with a base image for Python.

FROM ubuntu:24.04

# Use bash as the default shell for all subsequent commands.

SHELL ["/bin/bash", "-c"]

# The RUN, CMD, and ENTRYPOINT commands will now use bash.

# Set environment variables for clarity.

ENV APP\_HOME=/app

ENV PORT=5000

# Set the working directory for subsequent instructions.

WORKDIR ${APP\_HOME}

# Install Python, pip

RUN apt-get update

RUN apt-get install -y --no-install-recommends python3 python3-pip

RUN apt-get install -y --no-install-recommends curl

# Copy the application source code.

COPY . .

# Use ADD to download a file from a remote URL.

ADD https://upload.wikimedia.org/wikipedia/sa/d/da/IIT\_Bombay\_logo.png src/iit-bombay-logo.png

# Install dependencies from requirements file

RUN pip install --break-system-packages --no-cache-dir -r requirements.txt

# Expose the port the application will run on.

EXPOSE ${PORT}

# Add the 'appuser' user and group to the image.

RUN groupadd --gid 1001 appgroup && useradd --uid 1001 --gid 1001 -m appuser

# Change ownership of the application directory to the 'appuser' user

# so that the non-root user can read files.

RUN chown -R appuser:nogroup ${APP\_HOME}

# Set the user to run the application as a non-root user.

USER appuser

# Define labels for metadata and maintenance.

LABEL maintainer="Soumik Dutta/dutta.soumik.13@gmail.com"

LABEL version="1.0"

LABEL description="A simple Product Inventory API."

# Define a health check for the container.

HEALTHCHECK --interval=30s --timeout=5s --start-period=5s \

CMD curl -f http://localhost:${PORT}/health || exit 1

# ENTRYPOINT defines the primary command.

ENTRYPOINT ["python3"]

# CMD provides default arguments.

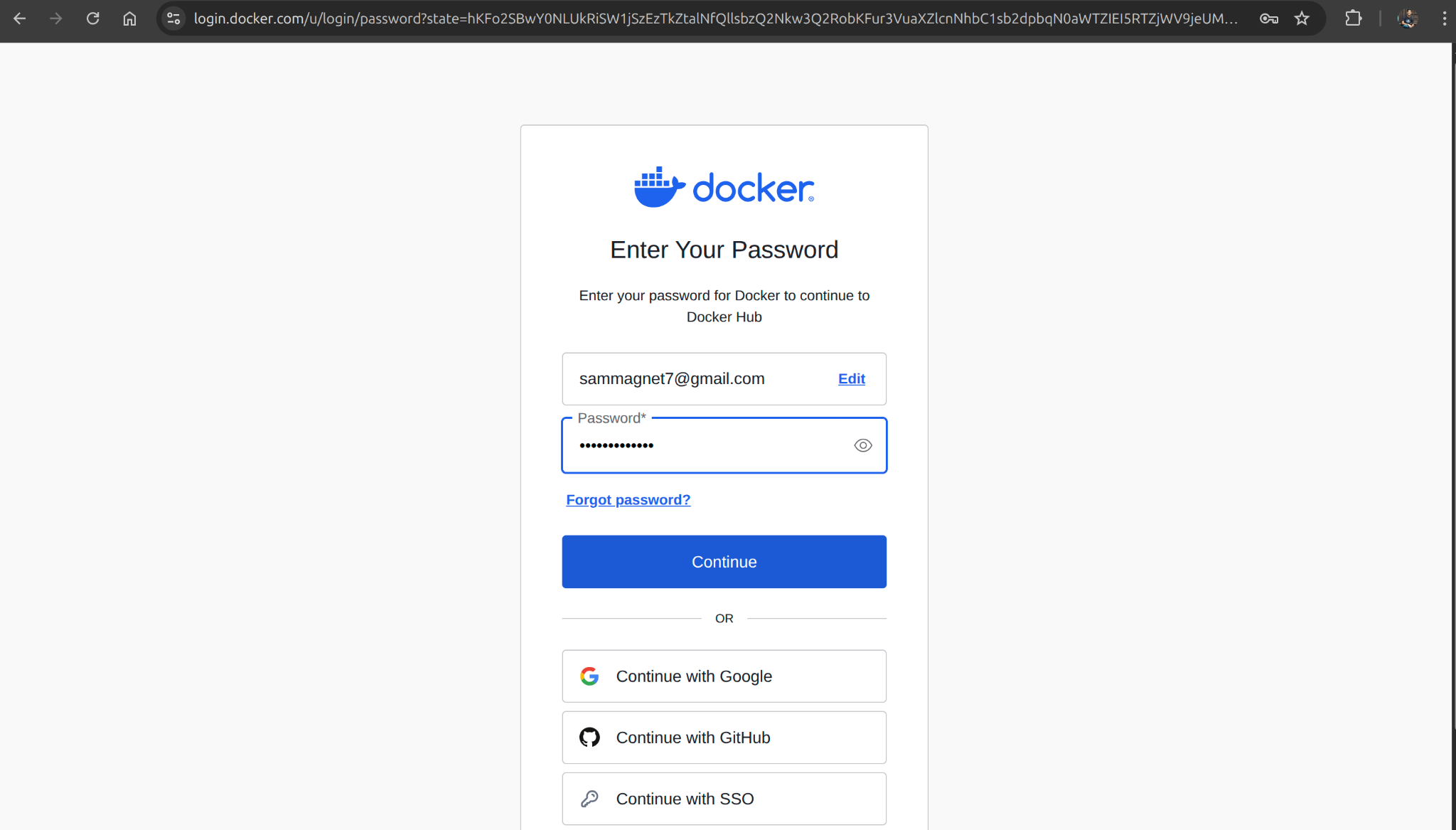
CMD ["src/app.py"]



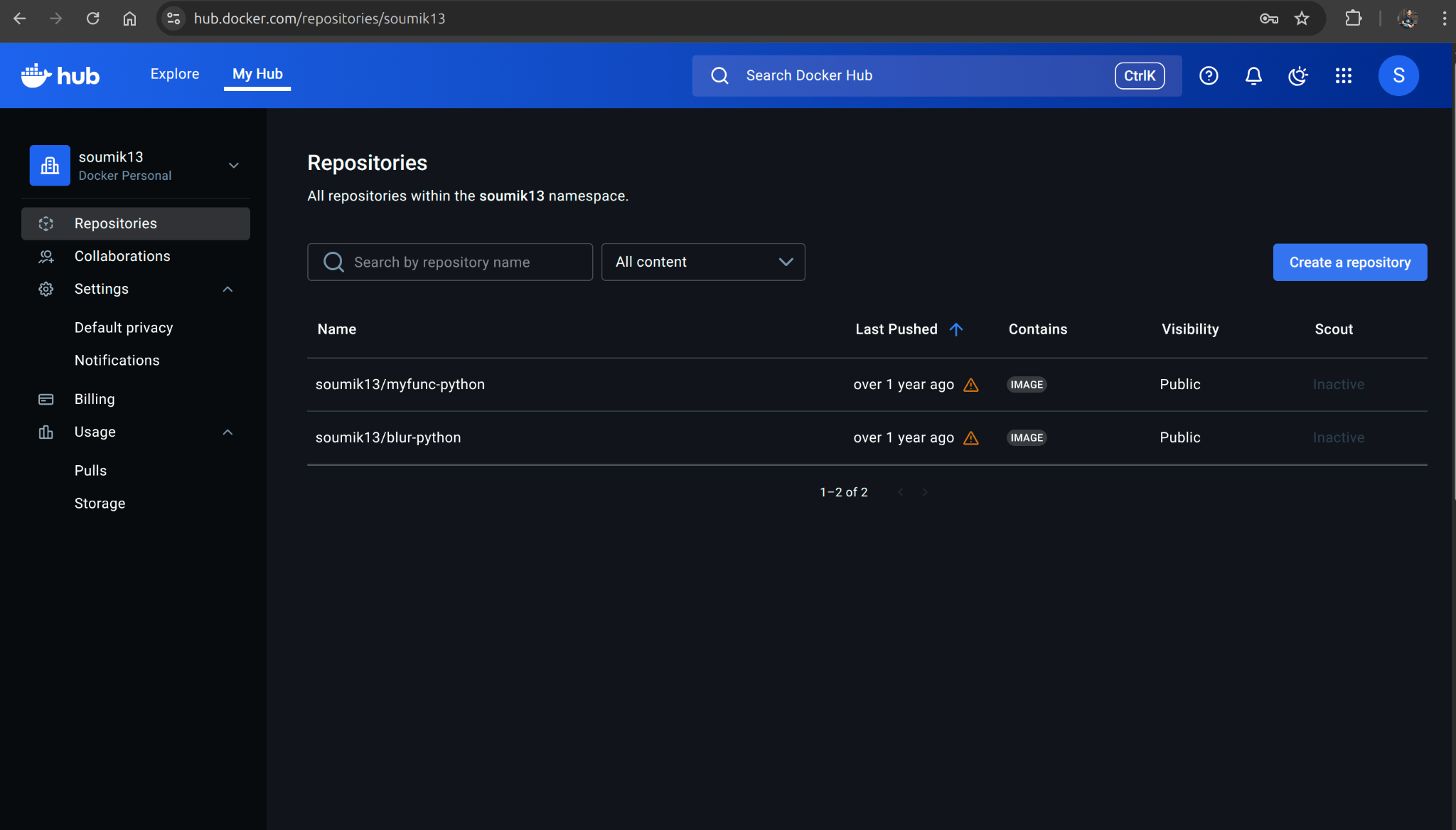
## 🌐 Logging into Docker Hub and Creating a Repository

Before pushing your image, you need a **public repository** on Docker Hub. Follow the steps below:

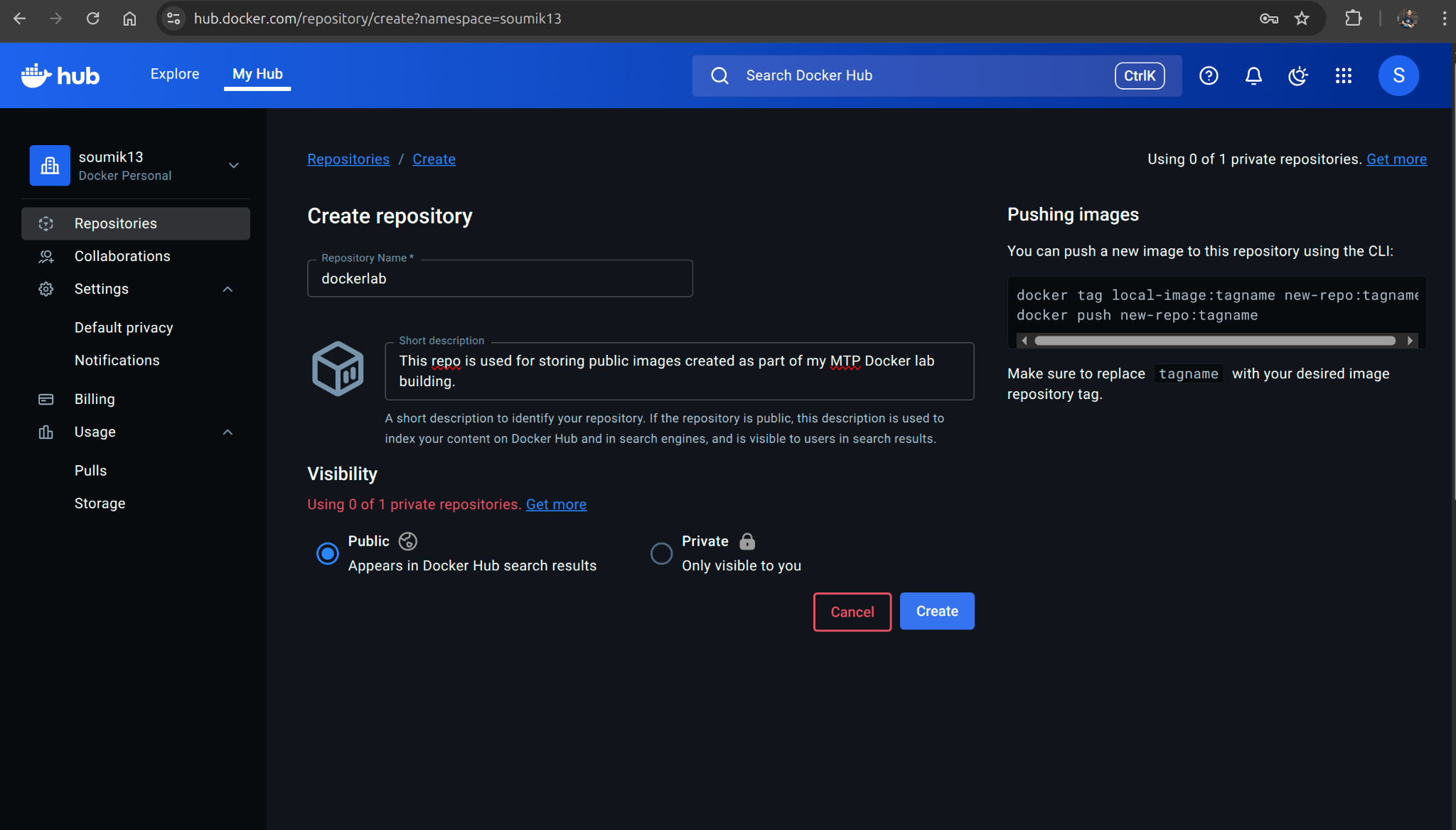
1. **Login to Docker Hub** Open your browser and go to <https://hub.docker.com>. Sign in with your Docker Hub credentials.



1. **Navigate to the Repository Dashboard** After login, click on **Repositories** in the top navigation bar.



1. **Create a New Repository**
   * Click **Create Repository**.
   * Provide a name, e.g., product-api.
   * Select **Public** visibility.
   * Optionally add a description.
   * Click **Create**.



1. **Verify Repository** You will be redirected to the repository dashboard showing your newly created repo.



✅ Now your Docker Hub repository is ready to receive the image you build and tag in the upcoming steps.

## 🧪 Run & Test Locally

Once you have your **final Dockerfile**, the next step is to **build, tag, and run** your image.

🎯 Goal

* Build the final Docker image from your Dockerfile.
* Retag the image with your Docker Hub repository name (so it can be pushed).
* Run the container and test its endpoints locally (via EC2 instance and your local PC browser).

📋 Explanation

* **Image Retagging** → When you build locally, you might use a name like lab3/final. However, Docker Hub requires the image to be tagged with your **Docker Hub username/repo name** (e.g., username/product-api:lab3). Retagging ensures Docker knows where to push the image when you run docker push.

🛠️ Steps

1. **Build the final image** inside your product-api-flask/ directory:

docker build -t lab3/final .



1. **Retag the image** with your Docker Hub repository name:

docker tag lab3/final <dockerhub-username>/product-api:lab3



1. **Run the container** on your EC2 instance:

docker run --rm -p 5000:5000 <dockerhub-username>/product-api:lab3



1. **Test endpoints from EC2** (open another terminal):

curl -s http://localhost:5000/api/products | head -n 5

curl -s -o /dev/null -w "%{http\_code}\\n" http://localhost:5000/health



1. **Test endpoints from your local PC browser** using EC2 public IP:
   * http://:5000/api/products
   * http://:5000/api/products/1
   * http://:5000/health
   * http://:5000/logo

🖊️ Student Action (ans.json)

{

"lab3-dockerhub-repo": "<dockerhub-username>/<dockerhub-reponame>",

"lab3-tag": "<tag-name>"

}



📌 Verification Checklist

* The image builds successfully without error.
* Container runs and serves endpoints correctly.
* Retagging works and the image is ready for pushing to Docker Hub.

## 🛫 Publish to Docker Hub (public)

Now that your image is built and tested, the final step is to **push it to Docker Hub** so it is publicly available.

🎯 Goal

* Authenticate with Docker Hub from your EC2 instance.
* Push the retagged image to your public repository.
* Verify that the image is published successfully.

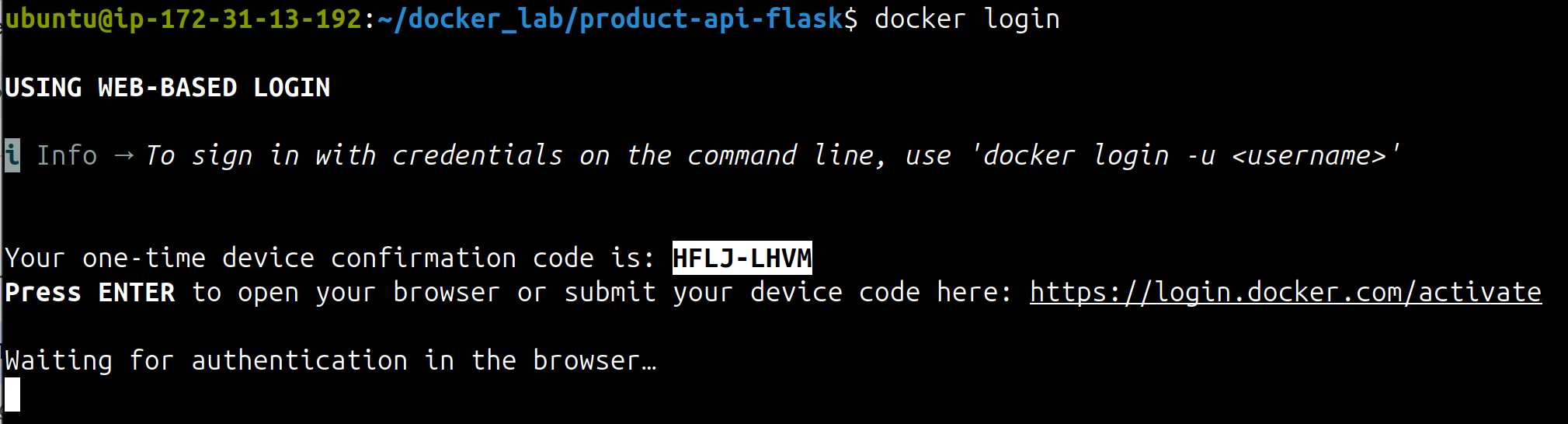
📋 Explanation

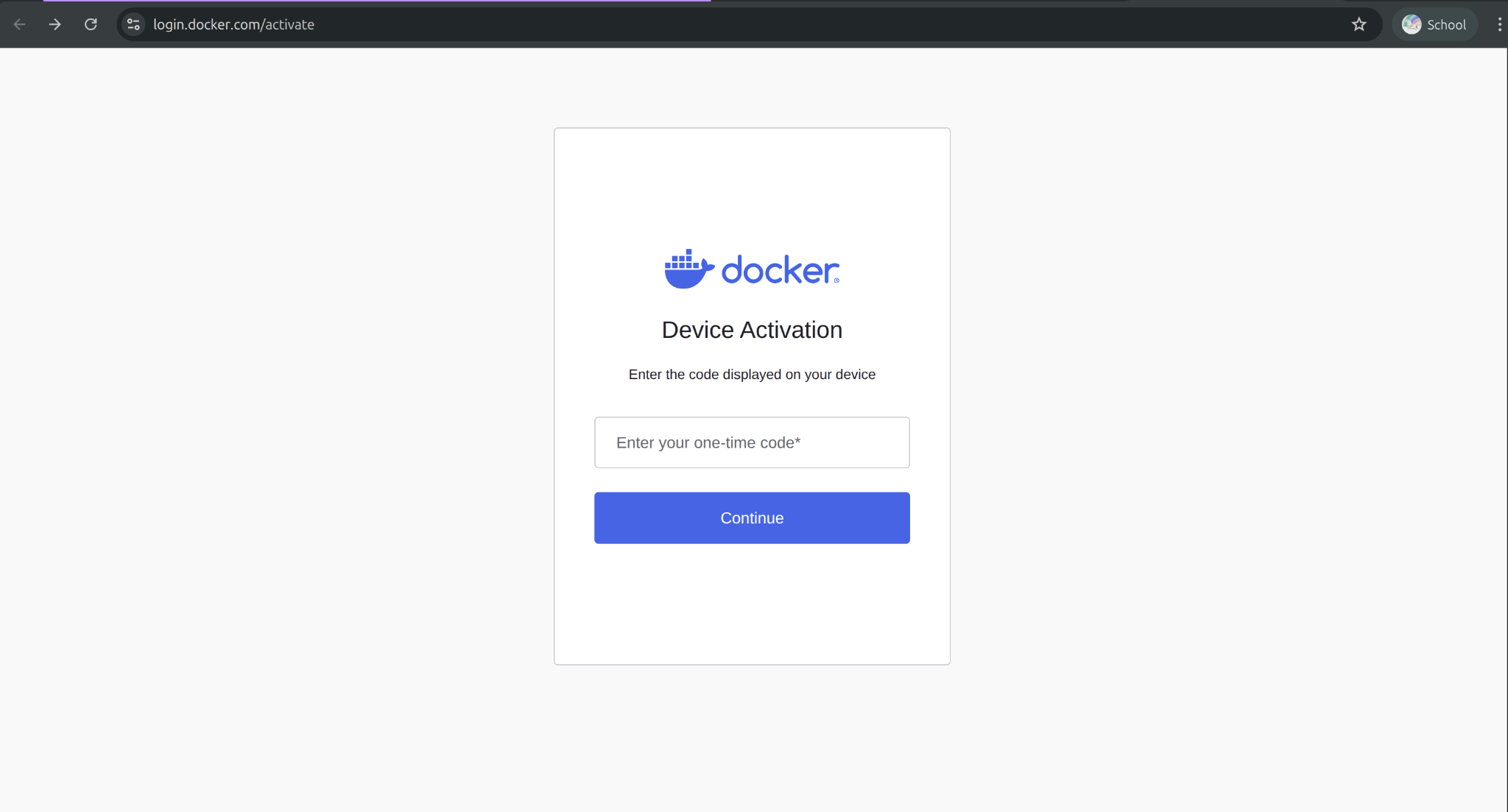
* **docker login** → Authenticates your local Docker CLI with Docker Hub. You will be prompted for your username and password (or access token).
* **docker push** → Uploads the tagged image layers to Docker Hub.
* **Digest** → Docker assigns a unique immutable identifier (sha256:<hash>) to the pushed image. This guarantees content integrity and can be used to reference the exact image version.

🛠️ Steps

1. **Login to Docker Hub** (if not already logged in):

docker login

   
 



1. **Push the image** to your repository:

docker push <dockerhub-username>/product-api:lab3



1. **Verify the push**:

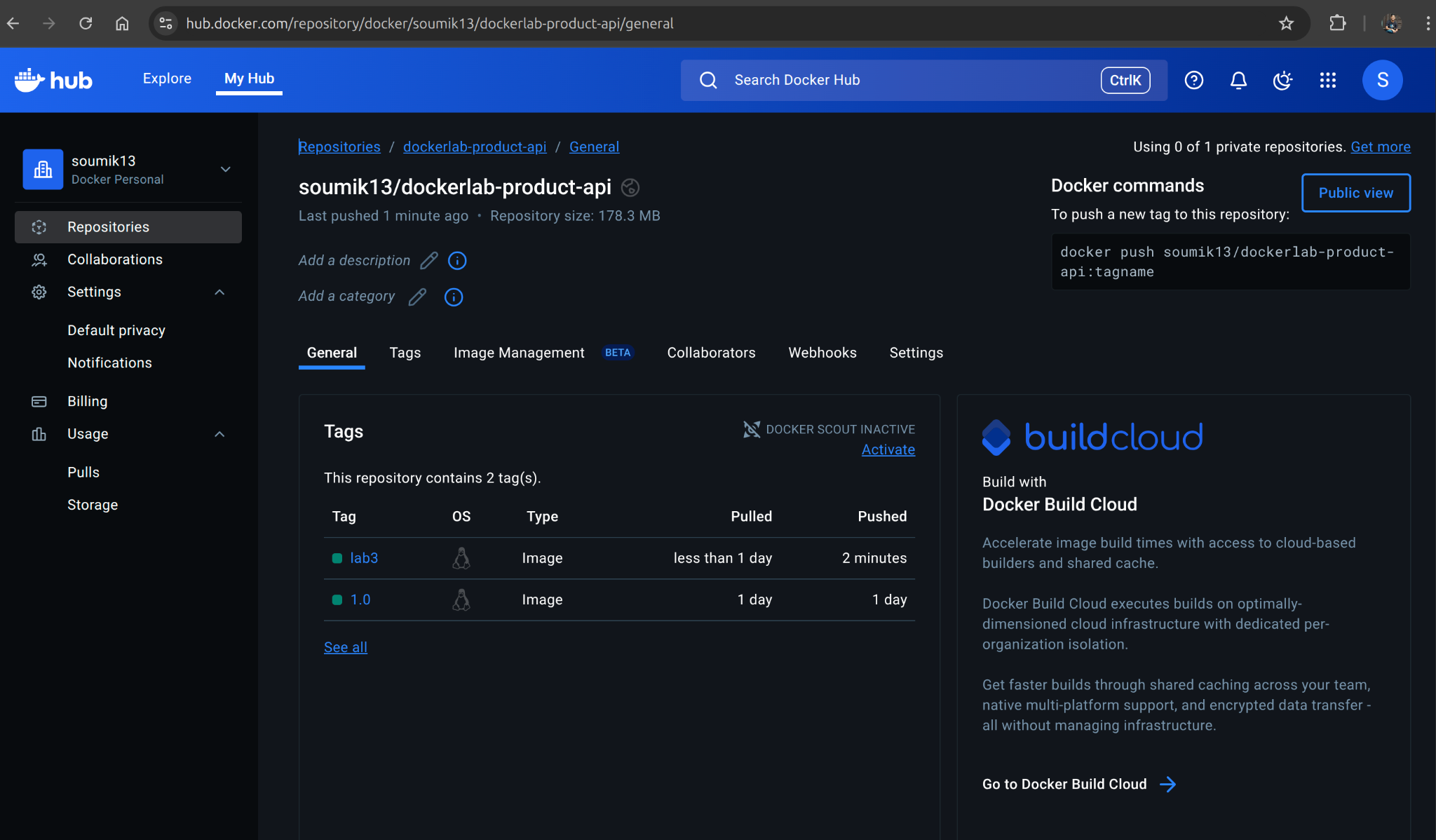
docker pull <dockerhub-username>/product-api:lab3

docker inspect <dockerhub-username>/product-api:lab3 --format='{{index .RepoDigests 0}}'



The RepoDigests field will display the digest (e.g., sha256:abcd123...).

1. **Check on Docker Hub**
   * Go to your repository page in Docker Hub.
   * Confirm the image with tag lab3 is listed.



🖊️ Student Action (ans.json)

{

"lab3-dockerhub-repo": "<dockerhub-username>/<dockerhub-reponame>",

"lab3-tag": "<tag-name>",

"lab3-digest": "<sha256:?>"

}



📌 Verification Checklist

* You can see the image with tag lab3 on your Docker Hub repository page.
* Pulling the image by name works from any system with Docker installed.
* The digest matches what you see in docker inspect.

## 📌 Final Verification Checklist

✅ Dockerfile matches exactly with the **Final Dockerfile** section.

✅ Image builds successfully without error.

✅ Running container answers on http://<public-ip>:5000/api/products (via browser).

✅ /health endpoint returns HTTP 200 and healthcheck flips to **healthy**.

✅ IIT Bombay logo visible at http://<public-ip>:5000/logo.

✅ Image is pushed to Docker Hub (public) at <dockerhub-username>/product-api:lab3.

✅ ans.json contains all required keys:

* lab3-base-image-PRETTY\_NAME
* lab3-app-home
* lab3-port
* lab3-python-version
* lab3-pip-version
* lab3-curl-version
* lab3-app-files-present
* lab3-logo-added
* lab3-flask-version
* lab3-exposed-port
* lab3-user
* lab3-user-id
* lab3-group-id
* lab3-version
* lab3-health-status
* lab3-entrypoint
* lab3-cmd
* logo-spotted
* lab3-dockerhub-repo
* lab3-tag
* lab3-digest

## 🧹 Cleanup After the Activity

1. Stop and remove all containers/images:

docker ps -aq | xargs docker rm -f  
docker images -aq | xargs docker rmi -f



1. Remove/Prune all Caches (optional):

docker buildx prune -f  
docker image prune -a -f  
docker system prune -a --volumes -f



1. 🚨 Remove all files from EC2 instance if needed:

rm -rf ~/docker\_lab/\*



## 🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

exit



1. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

## 🧠 Glossary (terms used above)

| **Term / Instruction** | **Meaning & Usage** |
| --- | --- |
| **Base Image (FROM)** | The starting filesystem for your image (e.g., Ubuntu). Determines available package manager and defaults. |
| **Layer** | Each Dockerfile instruction creates a cached, immutable layer. Smart ordering improves rebuild times. |
| **Build Context** | The files sent to the Docker daemon when you run docker build . (controlled by .dockerignore). |
| **RUN** | Executes shell commands at **build time**; results captured into a new layer. |
| **COPY / ADD** | Brings files into the image. Prefer COPY unless you need ADD features (URL fetch, auto-untar). |
| **ENV** | Defines environment variables available during build and at runtime. |
| **WORKDIR** | Sets the current directory for subsequent instructions and default runtime. |
| **EXPOSE** | Documents the port your app listens on (does not publish by itself). |
| **USER** | Sets the user for subsequent instructions and at runtime. |
| **LABEL** | Attaches metadata for tooling and inventory. |
| **ENTRYPOINT / CMD** | Define the container’s executable and default arguments. |
| **HEALTHCHECK** | A command Docker runs periodically; success (0) = healthy; failure (!=0) = unhealthy. |
| **Digest** | Content-addressable ID (algo:hex) for an image version; independent of human-readable tags. |
| **Image ID** | A unique identifier for a built image; changes if Dockerfile content changes. |
| **Tag** | A human-readable alias (like :lab3 or :latest) pointing to an image ID. |
| **Repository** | A named collection of related images (e.g., username/product-api). |
| **Container** | A runnable instance of an image; lightweight and isolated. |
| **Volume** | Persistent storage that exists outside container lifecycle; used for data durability. |
| **Bind Mount** | Mounting a local host path into a container; useful for development. |
| **Registry** | A service that stores and distributes images (e.g., Docker Hub). |
| **Docker Daemon** | Background service (dockerd) that builds, runs, and manages containers. |
| **Docker CLI** | Command line interface (docker) that interacts with the Docker daemon. |

## 🏆 Congratulations: You’ve completed Activity 3!

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

4️⃣ Activity 4: Networking in Docker

# Activity 4: Networking in Docker (without Compose)

🎯 Objective

* Understand how Docker networking enables communication between containers and host.
* Compare **default bridge**, **user-defined bridge**, and special modes (none, host).
* Practice manual multi-container integration (**MySQL + phpMyAdmin**) without Compose.
* Apply **inspection tools** and **security best practices** for container networking.

✅ Prerequisites

* Completion of **Activity 3** (Building & Publishing Images).
* A running **Ubuntu 24.04 LTS EC2 instance** with Docker installed.
* SSH access to EC2 instance (public-ip and secret-key.pem).
* Internet access on EC2 instance to pull images.
* AWS Security Group (docker-lab-sg) updated to allow inbound traffic for:
  + **Port 8080** (phpMyAdmin).
  + Any additional ports you choose for testing.

📂 Lab Setup

* Your lab directory contains below important file:
  + system-clean-init.sh → helper script that prepares your EC2 environment for this activity.
* Required Docker images (to be pulled during tasks):
  + nginx (web server).
  + alpine (lightweight client container with ping/curl).
  + mysql (database service).
  + phpmyadmin (database admin UI).
* No source code needed for this activity → focus is purely on **networking**.

⚠️ **Important Note about system-clean-init.sh:** Running this script will **reset/clean** your EC2 instance environment. Use it carefully, since it wipes existing Docker containers/images and resets the workspace. For more understanding go through the well commented script.

📦 About the Setup

Instead of coding an application, this activity uses **ready-made service containers**:

* **nginx** → to simulate a web service.
* **alpine** → as a lightweight debugging client.
* **mysql** → relational database.
* **phpmyadmin** → web UI for MySQL.

By combining these containers manually, you will see how Docker networking allows:

* Containers to talk **within the same network**.
* The host machine (and external browser) to access containers via **port publishing**.
* DNS resolution and inspection of **network topology**.

⚠️ Without Compose, you will manually connect containers to networks, define environment variables, and publish ports. This helps you understand how **Compose** simplifies these steps later (Activity 5).

🧭 How we’ll learn in this activity

We will progress step by step:

1. Explore the **default bridge network** (containers talk only via **IP**).
2. Create a **user-defined bridge network** (containers resolve each other by **name**).
3. Run **MySQL + phpMyAdmin** in the same network (manual integration + port publishing).
4. **Inspect networks** with Docker CLI (**subnets**, **DNS**, **connect**/**disconnect**).
5. Experiment with **special modes** (none, host) and review networking best practices.

By the end, you will have a solid grasp of **Docker’s native networking model**, how to troubleshoot connectivity, and how to apply **secure networking practices** in real-world setups.

📝 Notes

* **Default vs User-defined bridge:**
  + Default bridge is limited (no DNS).
  + User-defined bridge adds DNS + easier service discovery.
* **Port Publishing:**
  + Publishing is only needed for host/external access.
  + Containers in the same network don’t need ports exposed.

## 🐳 Task 1: Explore the Default Bridge Network

🎯 Goal

* Understand how Docker’s **default bridge network** works.
* Observe that containers in the default bridge can only talk to each other using **IP addresses**, not container names.
* Learn how to inspect container **networking** details using docker inspect.

📋 Explanation

* **Default Bridge Network** → When Docker is installed, it automatically creates a network named **bridge**.
  + All containers started **without specifying --network** are attached to this network.
  + Containers here can talk to each other, but **only via IP addresses** (no built-in DNS name resolution).
  + This is different from user-defined bridge networks, which we will explore in Task 2.
* **docker inspect <container>** → Used to fetch container details, including:
  + IP address.
  + Network name it is connected to.
  + Gateway and subnet details.
* **ping and curl inside containers** → Used to test connectivity.
  + ping <ip> → Tests basic ICMP reachability.
  + curl http://<ip> → Tests HTTP connectivity to nginx running inside a container.

🛠️ Steps

1. **Run an nginx container** (default bridge, port 80 exposed to container only, not host):

docker run -d --name web nginx



1. **Run an alpine container** (lightweight, attach to default bridge, interactive shell):

docker run -it --name client alpine sh



⚠️ Alpine does not have curl by default. Install it inside the container:

apk add --no-cache curl



1. **Find nginx container’s IP address** (from another terminal, outside client):

docker inspect -f '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' web



Suppose the IP is 172.17.0.2.

1. **From alpine container, ping nginx**:

ping -c 3 172.17.0.2



✅ Expected: you should see ICMP replies.

1. **From alpine container, curl nginx**:

curl http://172.17.0.2



✅ Expected: HTML response from nginx’s default welcome page.

1. **Try to curl using container name**:

ping -c 3 web

curl http://web



❌ Expected: This fails in default bridge (no DNS).

🖊️ Student Action (ans.json)

{

"lab4-defaultnetwork-type": "<bridge/host/none>",

"lab4-defaultnetwork-nginx-ip": "<?>",

"lab4-defaultnetwork-ping-success": "<true/false>",

"lab4-defaultnetwork-curl-works-with-ip": "<true/false>",

"lab4-defaultnetwork-curl-works-with-name": "<true/false>"

}



📌 Verification Checklist

* ✅ Nginx container (web) is running in background.
* ✅ Alpine container (client) is running and has curl installed.
* ✅ You can ping nginx container **by IP**.
* ✅ You can curl nginx container **by IP** and see HTML output.
* ❌ You cannot curl nginx container by **name** (web).

## 🐳 Task 2: Create and Use a User-Defined Bridge Network

🎯 Goal

* Create a **user-defined bridge network** and understand why it is preferred over the default bridge.
* Attach containers to the custom network and verify **name-based DNS resolution** (container names resolve automatically).
* Inspect the network to observe DNS entries and container IP assignments.

📋 Explanation

* **docker network create --driver bridge <name>** → Creates a new user-defined bridge network. User-defined bridge networks provide built-in DNS, automatic name resolution, and better isolation/segmentation than the default bridge.
  + **Why use it:** containers attached to the same user-defined bridge can resolve each other by container **name** (e.g., web), enabling convenient service discovery without needing IP addresses.
  + **Driver**: bridge is the common driver for single-host networking. Other drivers (overlay, macvlan) exist for multi-host or advanced scenarios, but are out of scope here.
* **--network <network>** flag on docker run → Attaches the container to a specific network at creation time.
* **DNS resolution behavior:** Docker injects entries into the **embedded DNS server** for containers connected to a user-defined network. This is what enables curl http://web to succeed (from another container on the same network).
* **docker network inspect <name>** → Shows **subnet**, **gateway**, and Containers map with container names and IPs: useful to verify DNS entries and IP assignments.

🛠️ Steps

1. **Create a user-defined bridge network** called lab4-net:

docker network create --driver bridge lab4-net



1. **Run nginx** attached to lab4-net (detached, name web):

docker run -d --name web2 --network lab4-net nginx



1. **Run alpine** attached to the same network for testing (interactive shell):

docker run -it --name client2 --network lab4-net alpine sh



Inside the alpine shell, install curl (and optionally iputils for ping on some alpine variants):

apk add --no-cache curl iputils



1. **Verify nginx IP and network membership** from the host (outside the client container):

docker inspect -f '{{range $k,$v := .NetworkSettings.Networks}}{{$k}} -> {{$v.IPAddress}}{{end}}' web2

# or

docker network inspect lab4-net



1. **From inside the alpine client shell, test name resolution and connectivity**:

# DNS by name (expected to succeed on user-defined bridge)

curl -sI http://web2 | head -n 5

# Ping by name (ICMP)

ping -c 3 web2



1. **Compare with curl by IP (optional):**

# find IP (host or via docker inspect) and curl:

curl -sI http://172.18.0.2 | head -n 5



1. **From the host, inspect the network mapping of containers**:

docker network inspect lab4-net --format='{{json .Containers}}' | jq



🖊️ Student Action (ans.json)

{

"lab4-userdefinednetwork-name": "<?>",

"lab4-userdefinednetwork-driver": "<?>",

"lab4-userdefinednetwork-web-ip": "<?>",

"lab4-userdefinednetwork-curl-works-with-name": "<true/false>",

"lab4-userdefinednetwork-ping-works-with-name": "<true/false>",

"lab4-userdefinednetwork-subnet": "<?>",

"lab4-userdefinednetwork-gateway": "<?>",

}



📌 Verification Checklist

* ✅ lab4-net network exists (docker network ls shows lab4-net).
* ✅ web2 container is attached to lab4-net (docker inspect web2 shows lab4-net under NetworkSettings).
* ✅ From client2, curl http://web2 returns HTTP headers / success (HTTP 200 or default nginx headers).
* ✅ From client2, ping web2 receives ICMP replies.
* ✅ docker network inspect lab4-net shows both web2 and client2 under Containers with assigned IP addresses.
* ✅ Student provided lab4-userdefinednetwork-\* key-values in ans.json.

## 🐳 Task 3: Multi-Container Integration (MySQL + phpMyAdmin)

🎯 Goal

* Run a **MySQL** server and **phpMyAdmin** as separate containers and connect them manually on the same user-defined network.
* Expose phpMyAdmin to the host so you can access the UI from a browser.
* Learn practical port-publishing patterns (-p 8080:80 and dynamic mapping -p :80) and security considerations (AWS SG + minimal published ports).
* Capture runtime evidence (container IDs, IPs, host port) needed by the autograder.

📋 Explanation

* **MySQL container**
  + Use the official mysql image. Provide MYSQL\_ROOT\_PASSWORD (required) and optionally MYSQL\_DATABASE, MYSQL\_USER, MYSQL\_PASSWORD to create a non-root DB user at first start.
  + MySQL initializes on first run; the container may take several seconds to become ready. You can check readiness with docker logs (look for “ready for connections”) or docker exec <mysql> mysqladmin ping -uroot -p"$MYSQL\_ROOT\_PASSWORD".
  + Persisting data with a volume is recommended in production but not required for this lab; the autograder will validate ephemeral containers.
* **phpMyAdmin container**
  + Use official phpmyadmin image. Configure connection target using PMA\_HOST=<mysql\_container\_name> (or IP) and PMA\_USER/PMA\_PASSWORD if needed.
* **PMA\_HOST** → target MySQL container name (required).
* **PMA\_USER / PMA\_PASSWORD** → optional defaults to auto-fill login (otherwise entered manually).
  + To access phpMyAdmin from your laptop/browser, publish the container port to the host with -p <host\_port>:80. Example: -p 8080:80.
  + **Container-to-container** traffic (phpMyAdmin → MySQL) happens on the **user-defined bridge network** without any -p. Only **host→container** requires publishing.
* **Port publishing patterns**
  + -p 8080:80 → deterministic host port 8080 maps to container 80. Use this when you want a known URL (http://<public-ip>:8080).
  + -p :80 (or -p 0:80) → Docker assigns a random host port; discoverable via docker ps or docker port. Useful when avoiding port conflicts; must query which host port was chosen.
  + **Security note:** Publish only the port(s) you need (in this task, only phpMyAdmin). Ensure the **EC2 security group** allows the chosen host port (e.g., 8080).
* **Networking**
  + Attach both containers to the same user-defined network (e.g., lab4-net) so phpMyAdmin can resolve MySQL by container name (e.g., mysql-db).
  + If MySQL and phpMyAdmin are on different networks, you must use docker network connect or publish ports—both are suboptimal for local service discovery.
* **Troubleshooting tips**
  + If phpMyAdmin shows “Error” or cannot connect:
    - Check docker logs mysql-db for initialization errors.
    - Verify PMA\_HOST exactly matches the MySQL container name (**case-sensitive**).
    - Confirm MySQL readiness (look for “ready for connections”). If not ready, phpMyAdmin will fail until MySQL finishes init.
  + Use docker exec -it mysql-db mysql -uroot -p"$MYSQL\_ROOT\_PASSWORD" -e "SELECT 1;" to verify DB access from host.
  + Use docker logs phpmyadmin to inspect phpMyAdmin errors (e.g., unable to resolve host).

🛠️ Steps

**Precondition:** Ensure lab4-net exists (created in Task 2). If not, create it:

docker network create --driver bridge lab4-net



1. **Run MySQL container** (named mysql-db) on lab4-net:

docker run -d \

--name mysql-db \

--network lab4-net \

-e MYSQL\_ROOT\_PASSWORD='rootpass123' \

-e MYSQL\_DATABASE='labdb' \

-e MYSQL\_USER='labuser' \

-e MYSQL\_PASSWORD='labpass123' \

mysql:8.0



1. **Wait for MySQL to initialize** (you can poll readiness or watch logs):

# Option A: tail logs until ready (in separate terminal)

docker logs -f mysql-db

# Option B: poll mysqladmin (returns "mysqld is alive" when ready)

docker exec mysql-db mysqladmin ping -uroot -prootpass123 --silent

# repeat until exit code 0



1. **Run phpMyAdmin** attached to the same lab4-net and publish **host** port 8080:

docker run -d \

--name phpmyadmin \

--network lab4-net \

-e PMA\_HOST='mysql-db' \

-e PMA\_USER='labuser' \

-e PMA\_PASSWORD='labpass123' \

-p 8080:80 \

phpmyadmin/phpmyadmin:latest



1. **Demonstrate dynamic/random port mapping**: run a second phpMyAdmin instance with Docker assigning a **host** port:

docker run -d \

--name phpmyadmin-random \

--network lab4-net \

-e PMA\_HOST='mysql-db' \

-e PMA\_USER='labuser' \

-e PMA\_PASSWORD='labpass123' \

-p :80 \

phpmyadmin/phpmyadmin:latest



Discover the assigned host port:

docker ps --format "table {{.Names}}\t{{.Ports}}"

# or

docker port phpmyadmin-random 80



1. **Verify phpMyAdmin from your browser** (on your laptop):
   * Visit: http://<EC2-public-ip>:8080 → You should see phpMyAdmin page.
2. **Verify container-to-container connectivity (no -p needed)** from inside phpmyadmin container shell (optional):

# Install ping and curl inside phpMyAdmin (Debian-based)

docker exec -it phpmyadmin sh -c "apt-get update && apt-get install -y curl iputils-ping netcat-openbsd >/dev/null 2>&1"

# 1. Ping MySQL by container name (tests DNS + ICMP reachability)

docker exec -it phpmyadmin ping -c 3 mysql-db

# 2. Check DNS resolution entry for mysql-db

docker exec -it phpmyadmin getent hosts mysql-db

# 3. Test MySQL port is open. Use netcat to check TCP connect.

docker exec -it phpmyadmin nc -vz mysql-db 3306



1. **Capture runtime evidence** (commands to run on host to fill ans.json):

# MySQL container IP

docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' mysql-db

# phpMyAdmin container IP

docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' phpmyadmin

# Which host port maps to phpMyAdmin (deterministic)

docker port phpmyadmin 80

# For dynamic mapping

docker port phpmyadmin-random 80

# Verify mysql readiness quickly

docker exec mysql-db mysqladmin ping -uroot -prootpass123



**Do not remove containers**: autograder will validate the running setup.

🖊️ Student Action (ans.json)

{

"lab4-mysql-container-name": "<?>",

"lab4-mysql-ip": "<?>",

"lab4-phpmyadmin-container-name": "<?>",

"lab4-phpmyadmin-ip": "<?>",

"lab4-phpmyadmin-host-port": "<?>",

"lab4-phpmyadmin-random-host-port": "<?>",

"lab4-phpmyadmin-accessible-from-host": "<true/false>",

"lab4-container-to-container-connection-works": "<true/false>"

}



*Hints for filling fields:*

* lab4-phpmyadmin-host-port: result of docker port phpmyadmin 80 (e.g., 0.0.0.0:8080). Fill only the numeric host port (8080).

📌 Verification Checklist

* ✅ mysql-db container is running and attached to lab4-net.
* ✅ phpmyadmin container is running and attached to lab4-net.
* ✅ docker logs mysql-db shows initialization completed / “ready for connections”.
* ✅ docker port phpmyadmin 80 shows 0.0.0.0:8080.
* ✅ From your browser, http://<EC2-public-ip>:8080 shows phpMyAdmin login page.
* ✅ Container-to-container resolution works: phpMyAdmin can resolve mysql-db (no host port needed).
* ✅ ans.json populated with the requested keys and correct values.

## 

## 📝 Notes:

**Port Reuse Across Networks**

* + Each Docker network provides an isolated namespace.
  + This means **containers on different networks can reuse the same container port** (e.g., multiple MySQL containers, each on a different network, all listening on port 3306).
  + No conflict occurs internally because each network maintains its own **routing table** and IP address space.
* **How Different Networks Communicate**
  + By default, **containers on different user-defined networks cannot reach each other directly**, even if they expose the same port number.
  + To enable communication, you have two main options:

**Connect one container to multiple networks**

A container can be attached to more than one network using:

docker network connect <network-name> <container-name>



This gives the container **an IP in each network**, allowing it to talk to services in both.

frontend-app <---> api-server <---> backend-db

(frontend-net) (dual-homed) (backend-net)



Example sequence:

docker network create frontend-net

docker network create backend-net

docker run -d --name frontend-app --network frontend-net nginx

docker run -d --name backend-db --network backend-net mysql:8.0

docker run -d --name api-server --network frontend-net my-api-image

docker network connect backend-net api-server

docker inspect api-server



Now api-server can reach both frontend-app and backend-db, acting as a **bridge** between the two isolated networks.

**Publish ports** and access via the **host’s IP**

* Example: publish MySQL with -p 3307:3306 and access it from another container via http://<host-ip>:3307.
* This approach is less efficient (traffic goes through host) but works if direct network connection isn’t possible.

**Best Practice**

* + Keep related services (like MySQL and phpMyAdmin) inside the **same user-defined network**.
  + Use multi-network attachments only when a service must interact with multiple isolated environments (e.g., reverse proxy bridging frontend and backend).
  + Avoid unnecessary port publishing: it expands the host’s attack surface. Prefer **internal networking with container names** for service-to-service communication.

## 🐳 Task 4: Inspect and Analyze Networks

🎯 Goal

* Use Docker CLI tools to inspect, analyze, and monitor networks.
* Understand what metadata Docker stores about networks (subnet, gateway, connected containers).
* Verify how DNS-based service discovery is working under the hood.

📋 Explanation

* **docker network ls** → Lists all networks on your host. You will see:
  + The **default bridge**, **host**, and **none** networks. (Will discuss more in Task5)
  + Any **user-defined networks** you created (e.g., lab4-net).
* **docker network inspect <network>** → Shows details about a network, including:
  + **Driver** (e.g., bridge).
  + **Subnet and Gateway** IPs.
  + **Containers** currently attached (with names, IDs, IPs).
  + Useful for verifying whether containers are correctly attached.
* **docker inspect <container>** → Can also be used at container level to confirm which networks it belongs to and what IPs it has. (Here search and explore field named: NetworkSettings.Networks )
* **DNS verification**
  + Containers in a user-defined bridge are registered with Docker’s **embedded DNS server**.
  + This allows commands like ping **mysql-db** or getent hosts phpmyadmin inside a container to resolve names.
  + **getent hosts <container>** → shows how the container name resolves to an IP address. (must run from inside the connected container)
* **docker events** (optional advanced) → Streams live Docker events (including when containers connect/disconnect from networks). Helpful for debugging.

🛠️ Steps

1. **List all networks on your host:**

docker network ls



Expected: you should see at least these:

* bridge (default)
* host
* none
* lab4-net (from Task 2/3)

1. **Inspect the user-defined network (lab4-net):**

docker network inspect lab4-net | jq



Look at:

* "Subnet"
* "Gateway"
* "Containers" section (should list mysql-db, phpmyadmin, and phpmyadmin-random).

1. **Inspect MySQL container (mysql-db) networking info:**

docker inspect mysql-db --format='{{json .NetworkSettings.Networks}}' | jq



1. **Verify DNS resolution inside a container (phpmyadmin):**

docker exec -it phpmyadmin getent hosts mysql-db

docker exec -it phpmyadmin getent hosts phpmyadmin-random



Expected: each command returns IP + name mappings.

1. **(Optional) Monitor Docker events** while disconnecting/reconnecting containers:

docker events --filter type=network



Open another terminal, then:

docker network disconnect lab4-net phpmyadmin

docker network connect lab4-net phpmyadmin



Watch how docker events logs these changes.

🖊️ Student Action (ans.json)

{

"lab4-networkls-names": "<?>",

"lab4-getent-mysql-db": "<?> (resolved IP from phpmyadmin)",

"lab4-getent-phpmyadmin-random": "<?> (resolved IP from phpmyadmin)"

}



📌 Verification Checklist

* ✅ docker network ls shows all default + custom networks.
* ✅ docker network inspect lab4-net shows correct subnet, gateway, and containers.
* ✅ docker inspect mysql-db confirms it is attached to lab4-net with correct IP.
* ✅ getent hosts mysql-db from inside phpMyAdmin resolves to MySQL’s IP.
* ✅ getent hosts phpmyadmin-random resolves to the random phpMyAdmin container’s IP.
* ✅ All ans.json keys filled with correct values.

## 🐳 Task 5: Special Network Modes (none and host) + Best Practices

🎯 Goal

* Learn the behavior and implications of Docker’s special network modes: --network none and --network host.
* Observe how host mode gives the container direct access to the host network stack (and the risks that entails).
* Observe how none mode gives the container no network at all (strict isolation).

📋 Explanation

* **--network none**
  + Creates a container with **no network interfaces** (except lo in some kernels). The container cannot reach other containers or the outside world.
  + Use-cases: tight isolation for **compute-only** workloads, **security sandboxes**, or when you want a container that must not have network access.
  + Limitations: you cannot ping, curl, or getent out of a none-networked container unless you explicitly docker network connect later.
* **--network host** (Linux behavior: applies to EC2 Ubuntu hosts)
  + The container **shares the host’s network namespace**. It sees the same network interfaces and IP addresses as the host.
  + Pros: **lowest latency**, no port mapping required (-p is ignored). Good for performance-sensitive networking workloads (raw sockets, network sniffers).
  + Cons / Security risks:
    - **No network isolation**: if the container is compromised, the attacker has direct access to host network interfaces.
    - **Port conflicts**: a service inside the container listening on port 80 will bind the host port 80 directly and may conflict with host services.
    - **Service discovery differences**: container cannot rely on Docker embedded DNS for name resolution the same way user-defined bridges do; name-based discovery between containers does not apply.
  + Note: On non-Linux platforms (Docker Desktop for Mac/Windows), --network host behaves differently or is limited: here we assume an Ubuntu EC2 host.
* **When to use which**
  + Use none for the strictest level of network isolation (e.g., untrusted workload, compute-only tasks).
  + Use host only when you explicitly need host-level networking performance or must bind to host network interfaces; otherwise prefer user-defined bridge networks for isolation and service discovery.

🛠️ Steps

**Warning:** In --network host mode, services bind directly to host ports. Be careful not to collide with essential host services (SSH, systemd services). Prefer test ports (e.g., 18080) or stop conflicting host services first on disposable lab VMs.

1. **Test none network: container should have no external network**

# Start a container with no network

docker run -d --name none-test --network none --rm alpine sleep 999999

# Exec into it and try network commands

docker exec -it none-test sh -c "ip addr || true"

docker exec -it none-test sh -c "apk add --no-cache iproute2 iputils || true"

docker exec -it none-test sh -c "ping -c 1 8.8.8.8 || true"

docker exec -it none-test sh -c "getent hosts google.com || true"

# Inspect container network settings (note: NetworkMode shows none)

docker inspect none-test --format '{{json .HostConfig.NetworkMode}}'



**Expected:**

* ip addr shows minimal interfaces (often only lo) or none.
* apk add,ping and getent fail (no network).

1. **Test host network: container shares host networking**

# Run a simple HTTP server in host mode binding to port 18080

docker run -d --name host-test --network host --rm alpine sh -c "apk add --no-cache python3 && python3 -m http.server 18080 & sleep 999999"

# From host, confirm the server is reachable (no -p required)

curl -sS http://localhost:18080 || true

# Inspect container network settings (note: NetworkMode shows host)

docker inspect host-test --format '{{json .HostConfig.NetworkMode}}'



**Expected:**

* curl http://localhost:18080 (run from host or another host process) returns HTTP response.
* docker inspect shows HostConfig.NetworkMode is host.
* docker ps will not show a published port for this container (because -p is ignored), but the service is available on the host IP/port.

1. **Demonstrate port conflict risk (do not run on SSH / critical ports!)**

# Attempt to run a container binding to host port 22 (SSH): usually fails if host SSH already bound

docker run -d --name dangerous --network host alpine sh -c "apk add --no-cache python3 && python3 -m http.server 22 & sleep 99999" || true

#verify error with

docker logs dangerous



**Expected:**

* If host already has SSH bound on 22, your container will fail to bind and may exit with an error; this demonstrates risk of port conflicts. Don’t run this on production hosts.

🖊️ Student Action (ans.json)

{

"lab4-none-container-name": "<?>",

"lab4-none-ip-interfaces": "<?>",

"lab4-none-can-ping": "<true/false>",

"lab4-host-container-name": "host-test",

"lab4-host-networkmode": "host",

"lab4-host-service-port": "<?>",

"lab4-host-accessible-from-host": "<true/false>",

"lab4-host-port-conflict-observed": "<true/false>"

}



*Hints:*

* lab4-none-ip-interfaces: capture the list like lo / none.
* lab4-host-accessible-from-host: result of curl http://localhost:18080 on the host (true/false).
* lab4-host-port-conflict-observed: true if any container failed to bind a host port because the host already had it in use.

📌 Verification Checklist

* ✅ none-test exists and shows no network connectivity (ip addr minimal, ping fails).
* ✅ host-test is running with NetworkMode set to host.
* ✅ Service started inside host-test is reachable from the host without -p (e.g., curl http://localhost:18080).
* ✅ docker inspect outputs contain the expected NetworkMode values for each container.
* ✅ ans.json populated with required keys.

🧠 Security & Best Practices (summary)

* Prefer **user-defined bridge** networks for application containers: they provide DNS-based service discovery and isolation.
* Use --network none for containers that must not have network access (untrusted tasks).
* Use --network host **only when necessary** (performance reasons or direct access to host interfaces). Understand that host mode removes network isolation and increases attack surface.
* Avoid running containers in host mode on production infrastructure unless you have a clear justification and compensating controls (firewalls, minimal privileges).
* Minimize port publishing (-p) to what is strictly required and keep your EC2 security group rules restrictive (only allow inbound on required host ports).

## 📌 Final Verification Checklist

✅ Containers on **default bridge** communicate only by IP (DNS fails).

✅ Containers on **user-defined bridge (lab4-net)** communicate by name (DNS works).

✅ mysql-db, phpmyadmin, and phpmyadmin-random run correctly on lab4-net.

✅ phpMyAdmin is accessible from browser at http://<EC2-public-ip>:8080.

✅ DNS resolution inside containers works using getent hosts.

✅ none-test container shows no network connectivity.

✅ host-test container shares host networking (service reachable at localhost:18080).

✅ ans.json contains all required keys:

* lab4-defaultnetwork-type
* lab4-defaultnetwork-nginx-ip
* lab4-defaultnetwork-ping-success
* lab4-defaultnetwork-curl-works-with-ip
* lab4-defaultnetwork-curl-works-with-name
* lab4-userdefinednetwork-name
* lab4-userdefinednetwork-driver
* lab4-userdefinednetwork-web-ip
* lab4-userdefinednetwork-curl-works-with-name
* lab4-userdefinednetwork-ping-works-with-name
* lab4-userdefinednetwork-subnet
* lab4-userdefinednetwork-gateway
* lab4-mysql-container-name
* lab4-mysql-ip
* lab4-phpmyadmin-container-name
* lab4-phpmyadmin-ip
* lab4-phpmyadmin-host-port
* lab4-phpmyadmin-random-host-port
* lab4-phpmyadmin-accessible-from-host
* lab4-container-to-container-connection-works
* lab4-lab4net-connected-containers
* lab4-getent-mysql-db
* lab4-getent-phpmyadmin-random
* lab4-none-container-name
* lab4-none-ip-interfaces
* lab4-none-can-ping
* lab4-host-container-name
* lab4-host-networkmode
* lab4-host-service-port
* lab4-host-accessible-from-host
* lab4-host-port-conflict-observed

## 🧹 Cleanup After the Activity

1. Stop and remove all containers/images:

docker ps -aq | xargs docker rm -f  
docker images -aq | xargs docker rmi -f



1. Remove custom network:

docker network rm lab4-net



1. Remove/Prune all Caches (optional):

docker buildx prune -f  
docker image prune -a -f  
docker system prune -a --volumes -f



1. 🚨 Remove all files from EC2 instance if needed:

rm -rf ~/docker\_lab/\*



## 🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

exit



1. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

## 🧠 Glossary (terms used above)

| **Term / Instruction** | **Meaning & Usage** |
| --- | --- |
| **bridge Network** | Default network driver; isolates containers but allows manual IP connectivity. |
| **User-defined Bridge** | Custom bridge with built-in DNS, supports container name resolution. |
| **Port Publishing (-p)** | Maps a host port to a container port, enabling external access. |
| **PMA\_HOST / PMA\_USER / PMA\_PASSWORD** | phpMyAdmin environment variables to configure MySQL connection target and default login. |
| **docker network ls** | Lists all networks available on the Docker host. |
| **docker network inspect** | Shows details of a network (subnet, gateway, connected containers). |
| **docker inspect <container>** | Displays container details, including network assignments. |
| **DNS Resolution (getent hosts)** | Verifies container name resolves to its IP within a user-defined network. |
| **--network none** | Starts container without any network access (except loopback). |
| **--network host** | Shares host’s network namespace; no isolation, but high performance. |
| **Port Conflict** | When two services attempt to bind the same host port; leads to failure. |
| **Embedded DNS** | Internal Docker DNS server for resolving container names in user-defined networks. |

Here’s a **Note** section you can plug in after Task 5 (or at the very end before Glossary). It briefly mentions other Docker network drivers beyond bridge, host, and none:

## 📝 Notes: Other Docker Network Drivers

While this activity focused on **bridge**, **host**, and **none**, Docker provides several other drivers for advanced use-cases:

* **Overlay**
  + Connects containers across **multiple Docker hosts** using a distributed network.
  + Commonly used with **Docker Swarm** or orchestrators like **Kubernetes**.
  + Enables multi-host service discovery and communication without manual tunneling.
* **Macvlan**
  + Assigns containers their **own MAC address**, making them appear as physical devices on the local network.
  + Useful when you want containers to integrate directly into an existing physical LAN, each with its own IP.
  + Common in legacy systems that expect direct L2/L3 connectivity.
* **Ipvlan**
  + Similar to macvlan but more lightweight; shares the host’s MAC address while assigning unique IPs at L3.
  + Useful in environments where MAC address limits exist (e.g., some cloud providers).
* **Overlay (Swarm/K8s)** and **Macvlan/Ipvlan** are less commonly needed in basic labs, but are crucial in **production-scale or enterprise networking scenarios**.

## 🏆 Congratulations: You’ve completed Activity 4!

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

5️⃣ Activity 5: Multi-Container with Compose

# Activity 5: Multi-Container Applications with Docker Compose

🎯 **Objective**

* Learn how Docker Compose simplifies orchestration of multi-container applications.
* Containerize and orchestrate a real **3-tier demo application** (React frontend, Spring Boot backend, PostgreSQL database).
* Understand Compose concepts step by step:
  + Networks
  + Environment variables
  + Secrets
  + Volumes & persistence
  + Healthchecks & startup sequencing
  + Restart policies
  + Scaling
  + Override files
* Finally, build a **complete production-ready docker-compose.yml** from scratch.

✅ **Prerequisites**

* Completion of **Activity 4** (Networking in Docker).
* A running **Ubuntu 24.04 LTS EC2 instance** with Docker + Docker Compose installed.
* SSH access to EC2 instance (public-ip and secret-key.pem).
* Internet access on EC2 to pull images.

📂 **Lab Setup**

* Your lab directory contains:
  + dockerlab\_activity5\_3TierApp.tar.xz → compressed archive of the 3-tier app.
  + system-clean-init.sh → helper script that prepares your EC2 environment for this activity.

⚠️ **Important Note about system-clean-init.sh:** Running this script will **reset/clean your EC2 instance environment** (remove all containers, images, volumes) and then **extract the 3-tier demo app** into the correct working directory. Use it carefully.

🛠️ Run the setup script (in clab terminal):

bash system-clean-init.sh



Then log in to your EC2 instance and move into the extracted app folder:

ssh -i secret-key.pem ubuntu@<public-ip>

cd /home/ubuntu/docker\_lab/dockerlab\_activity5\_3TierApp



📦 **About the Application**

This is a simple **3-tier demo app**:

* **Frontend** → React app served via Nginx.
* **Backend** → Spring Boot REST API that exposes course-related endpoints.
* **Database** → PostgreSQL storing courses, registrations, and grades.

The backend initializes schema & seed data at startup. The frontend consumes backend APIs and shows animated UI.

## 🐳 Task 1: Understand the 3-Tier App

🎯 **Goal** Get familiar with the application codebase, its structure, and verify that each component (Postgres, backend, frontend) works when run manually using docker build and docker run.

📂 **Project Tree**

$ pwd

/home/ubuntu/docker\_lab/dockerlab\_activity5\_3TierApp

$ tree -a

.

├── iitb-course-backend

│ ├── .dockerignore

│ ├── .vscode

│ │ ├── launch.json

│ │ └── settings.json

│ ├── Dockerfile

│ ├── README.md

│ ├── pom.xml

│ └── src

│ └── main

│ ├── java

│ │ └── org

│ │ └── iitb

│ │ └── demo

│ │ ├── CourseBackendApplication.java

│ │ ├── config

│ │ │ └── WebConfig.java

│ │ ├── controller

│ │ │ └── CourseController.java

│ │ ├── model

│ │ │ ├── Course.java

│ │ │ ├── Grade.java

│ │ │ └── Registration.java

│ │ └── repository

│ │ ├── CourseRepository.java

│ │ ├── GradeRepository.java

│ │ └── RegistrationRepository.java

│ └── resources

│ ├── application.yml

│ ├── db

│ │ └── migration

│ │ └── V1\_\_init\_schema\_and\_data.sql

│ └── logback-spring.xml

└── iitb-course-frontend

├── .dockerignore

├── Dockerfile

├── README.md

├── docker-entrypoint.sh

├── package.json

├── public

│ ├── env-config.template.js

│ └── index.html

└── src

├── App.js

├── api.js

├── components

│ ├── CourseList.js

│ └── CourseModal.js

├── index.css

└── index.js

20 directories, 31 files



🛠️ **Steps**

Run the following commands step by step to manually build and run the app components.

👉 Replace <EC2\_IP> with your EC2 public IP.

# (1) Run Postgres (persistent volume, exposed on host port 5432)

docker run -d --name postgres \

-e POSTGRES\_DB=demo\_db \

-e POSTGRES\_USER=demo\_user \

-e POSTGRES\_PASSWORD=demo\_pass \

-v pgdata:/var/lib/postgresql/data \

-p 5432:5432 \

postgres:15

# (2) Build and run backend (Spring Boot)

cd iitb-course-backend

docker build -t iitb-course-backend .

docker run -d --name backend \

-p 8080:8080 \

-e SPRING\_DATASOURCE\_URL=jdbc:postgresql://<EC2\_PRIVATE\_IP>:5432/demo\_db \

-e SPRING\_DATASOURCE\_USERNAME=demo\_user \

-e SPRING\_DATASOURCE\_PASSWORD=demo\_pass \

iitb-course-backend

# (3) Build and run frontend (React + Nginx)

cd ../iitb-course-frontend

docker build -t iitb-course-frontend .

docker run -d --name frontend \

-p 3000:80 \

-e REACT\_APP\_API\_BASE\_URL="http://<EC2\_PUBLIC\_IP>:8080" \

iitb-course-frontend

# (4) Verify endpoints

# From EC2 host:

curl -i http://localhost:8080/api/health

curl -i http://localhost:8080/api/courses

# From your laptop browser:

# - http://<EC2\_PUBLIC\_IP>:3000 (frontend UI)

# - http://<EC2\_PUBLIC\_IP>:8080/api/health

# - http://<EC2\_PUBLIC\_IP>:8080/api/courses

# (5) Inspect logs

docker logs -f backend

docker logs -f frontend

docker logs -f postgres

# (6) Cleanup when done

docker stop frontend backend postgres

docker rm frontend backend postgres



📌 **Notes**

* Here we use **published ports (-p)** so services are directly reachable on <EC2\_PUBLIC\_IP>.
* No Docker network is needed at this stage because all traffic flows via host ports.
* This task is for **understanding only**: *no test cases are provided*.

⚠️ Before starting Task 2, **stop all running containers** and rerun (in clab terminal):

bash system-clean-init.sh



This ensures a clean environment and avoids port conflicts.

## 📝 Note: Why Docker Compose?

When working with multi-container applications (like our **3-tier app** with frontend, backend, and database), manually running containers with docker run becomes **tedious and error-prone**. You would have to remember:

* Each container’s docker run command.
* Port mappings, environment variables, volumes, and networks.
* Startup order (DB before backend, backend before frontend).

To solve this, **Docker Compose** was created. It lets you:

* Define all services in a **single YAML file (docker-compose.yml)**.
* Use simple commands like docker compose up and docker compose down to start/stop everything.
* Automatically create a network so services can talk by name.
* Handle configuration in a **reproducible** and **portable** way.

⚡ **Why it came to market:** Teams needed a **standardized tool** to manage multi-container apps easily, without custom shell scripts or manual orchestration. Compose filled that gap and became the de facto tool for local development and small deployments.

## 🐳 Task 2: Compose Basics (Single-Service Postgres)

🎯 **Goal** Learn the basics of Docker Compose by running a single Postgres service. This task introduces the docker-compose.yml format and basic commands.

🛠️ **Steps**

1. Create a new file named docker-compose.yml inside the dockerlab\_activity5\_3TierApp folder with the following content:

services:

postgres:

image: postgres:15

container\_name: postgres

ports:

- "5432:5432"

environment:

POSTGRES\_DB: demo\_db

POSTGRES\_USER: demo\_user

POSTGRES\_PASSWORD: demo\_pass



📖 **Explanation of keywords**

* **services** → Top-level key defining all containers (services) in the app.
* **postgres** → Name of the service (also acts as its DNS hostname on the Compose network).
* **image** → The Docker image to use (postgres:15).
* **container\_name** → Explicitly names the container (otherwise Compose generates one).
* **ports** → Publishes container’s port 5432 to host port 5432 so we can connect from outside.
* **environment** → Environment variables passed into the container:
  + POSTGRES\_DB → Creates a database named demo\_db.
  + POSTGRES\_USER → Creates a user demo\_user.
  + POSTGRES\_PASSWORD → Password for that user.

🛠️ **Run and Verify**

1. Start the service in detached mode:

docker compose up -d



1. Check running containers:

docker compose ps

docker network ls

docker volume ls



1. View logs for Postgres:

docker compose logs postgres



1. Stop the service:

docker compose down



📌 **Notes**

* The **version** attribute is now **obsolete** in Docker Compose v2.
* At this stage we are not using **volumes**, **networks**, or **healthchecks**.
* Docker Compose automatically creates a **default network**, but since only one service is running, this doesn’t matter yet.
* *No test cases are provided for this task*: it is for learning and practice only.

⚠️ Before moving on, ensure you stop services with docker compose down to keep your environment clean.

## 🐳 Task 3: Custom Networks in Compose

🎯 **Goal** Learn how to define and use a **user-defined bridge network** in Docker Compose so services can communicate by name and be isolated from other containers.

🛠️ **Steps**

1. Create or update docker-compose.yml in dockerlab\_activity5\_3TierApp to declare a named network and attach the postgres service to it. Example minimal file:

services:

postgres:

image: postgres:15

container\_name: postgres

ports:

- "5432:5432"

environment:

POSTGRES\_DB: demo\_db

POSTGRES\_USER: demo\_user

POSTGRES\_PASSWORD: demo\_pass

networks:

- app-net

networks:

app-net:

driver: bridge



1. Start the service (the network will be created by Compose):

docker compose up -d



1. Verify the network exists and inspect its details:

docker network ls

docker network inspect dockerlab\_activity5\_3tierapp\_app-net

# or inspect by the name shown in `docker network ls`



1. Run another container on the same network to test name resolution (example: lightweight alpine client):

docker run -it --rm --network dockerlab\_activity5\_3tierapp\_app-net alpine sh

# inside the alpine shell, install tools and test DNS/HTTP:

apk add --no-cache curl iputils

ping -c 2 postgres

curl -sI http://postgres:5432 || true



Note: Replace dockerlab\_activity5tierapp\_app-net above with the actual network name reported by docker network ls. Compose-derived network names are usually <folder>\_<network> unless name: is provided explicitly under the networks: block.

📖 **Explanation of keywords**

* **networks:** → Top-level key defining networks that Compose will create.
* **app-net:** → Logical network name you choose (used by services to attach).
* **driver: bridge** → Creates a user-defined bridge network (default and suitable for single-host deployments).
* **services.networks** → Lists networks the service will join. Joining the same user-defined network enables **DNS-based service discovery** by service name (postgres).

📌 **Verification Checklist**

* ✅ docker network ls shows a new network for this Compose project.
* ✅ docker network inspect <network> lists the postgres container under Containers.
* ✅ From another container attached to the same network, ping postgres resolves and responds.

📌 **Notes & Tips**

* Compose automatically creates networks declared in the networks: section when you run docker compose up.
* If you want to **control the exact network name**, add a name: under the network definition:

networks:

app-net:

name: iitb-app-net

driver: bridge



* Using a user-defined network is generally **preferable to the default bridge** because it provides built-in DNS and easier service discovery.
* For Task 3 we attached only Postgres to the custom network: later tasks will attach backend and frontend so they communicate internally by service name.
* *No test cases are provided for this task*: it is for learning and practice only.

⚠️ Before moving on, keep the Compose services running or stop them with:

docker compose down



## 🐳 Task 4: Environment Variables in Compose

🎯 **Goal** Learn how to externalize configuration for containers using a .env file so you don’t hardcode values in docker-compose.yml.

🛠️ **Steps**

1. Create a .env file at the project root (dockerlab\_activity5\_3TierApp/.env) with the following contents:

# .env

POSTGRES\_DB=demo\_db

POSTGRES\_USER=demo\_user

POSTGRES\_PASSWORD=demo\_pass



1. Update your docker-compose.yml to reference these variables:

services:

postgres:

image: postgres:15

container\_name: postgres

environment:

POSTGRES\_DB: ${POSTGRES\_DB}

POSTGRES\_USER: ${POSTGRES\_USER}

POSTGRES\_PASSWORD: ${POSTGRES\_PASSWORD}

ports:

- "5432:5432"

networks:

- app-net

networks:

app-net:

driver: bridge



1. Validate substitution and final config:

docker compose config



This shows the resolved configuration with variables substituted.

1. Start the service:

docker compose up -d



1. Verify Postgres is running with your custom values:

docker compose logs postgres



📖 **Explanation of keywords & behavior**

* **.env file** → Compose automatically loads variables from .env if present in project **root**.
* **${VAR} syntax** → Substitutes environment variables inside docker-compose.yml.
* **docker compose config** → Debug tool to render full YAML after substitution.

📌 **Verification Checklist**

* ✅ .env file exists and contains the database variables.
* ✅ docker compose config shows POSTGRES\_DB, POSTGRES\_USER, POSTGRES\_PASSWORD substituted.
* ✅ Postgres starts successfully and logs show the DB and user created with your values.

📋 **Notes**

* At this stage only **Postgres** is part of the Compose file: backend and frontend will come later.
* This task focuses purely on externalizing config for database service.
* If you want to override a value temporarily, export it in your shell before docker compose up (shell env has higher precedence):

export POSTGRES\_PASSWORD="temp\_pass"; docker compose up -d



* For reproducible labs, keep .env in the repo for default values but **never commit real production secrets**.
* *No test cases are provided for this task*: it is for learning and practice only.

⚠️ Before moving on, keep the Compose services running or stop them with:

docker compose down



## 🐳 Task 5: Secrets in Compose (Compose-native)

🎯 **Goal** Secure sensitive values by using Docker Compose **secrets** (Compose can mount secret files into containers). Move the database password out of .env into a secret file so it is not stored alongside other configuration. We will keep POSTGRES\_DB and POSTGRES\_USER in .env and **remove** POSTGRES\_PASSWORD from .env.

🛠️ **Steps**

1. **Remove password from .env** (edit dockerlab\_activity5\_3TierApp/.env):

# .env (updated)

POSTGRES\_DB=demo\_db

POSTGRES\_USER=demo\_user

# POSTGRES\_PASSWORD is intentionally removed from .env



1. **Create a directory and secret file on the host** (with strict permissions):

mkdir -p ./secrets

printf "demo\_pass" > ./secrets/postgres\_password.txt

chmod 400 ./secrets/postgres\_password.txt



1. **Update docker-compose.yml** to use the secret and still supply DB name/user via environment:. Because the official postgres image expects POSTGRES\_PASSWORD in the environment, we use a small wrapper that reads the mounted secret and exports POSTGRES\_PASSWORD before starting Postgres:

services:

postgres:

image: postgres:15

container\_name: postgres

environment:

POSTGRES\_DB: ${POSTGRES\_DB}

POSTGRES\_USER: ${POSTGRES\_USER}

# DO NOT set POSTGRES\_PASSWORD here; it will come from the secret

secrets:

- postgres\_password

networks:

- app-net

entrypoint: [ "sh", "-c" ]

command: >

"export POSTGRES\_PASSWORD=$(cat /run/secrets/postgres\_password) &&

exec docker-entrypoint.sh postgres"

secrets:

postgres\_password:

file: ./secrets/postgres\_password.txt

networks:

app-net:

driver: bridge



1. **Start the stack**:

docker compose up -d



1. **Verify the secret is mounted and Postgres started**:

# Check mounted secret inside container

docker exec -it postgres sh -c "ls -l /run/secrets && cat /run/secrets/postgres\_password"

# Check Postgres logs for successful startup

docker logs postgres



1. **(Optional)** Remove the secret file from host after use if needed, or **keep it secure outside the repository**.

📖 **Explanation of keywords & behavior**

* **secrets:**: Top-level Compose key declaring secrets. Using file: tells Compose to use a local file as the secret source and mount it into containers at /run/secrets/<name>.
* **service.secrets:**: The service consumes the secret; at runtime the secret is mounted read-only to /run/secrets/<name>.
* **Why environment: still needed:** POSTGRES\_DB and POSTGRES\_USER must be provided so Postgres can initialize the database and user. We keep those in .env and reference them via environment:.
* **Why wrapper is used:** The official postgres image expects POSTGRES\_PASSWORD via an environment variable. Since Compose mounts secrets to files, we use a minimal wrapper command that reads /run/secrets/postgres\_password and exports POSTGRES\_PASSWORD before invoking the standard entrypoint. This keeps the password out of .env and avoids leaking it in the process command or image layers.
* **Permissions:** Keep the host secret file with tight permissions (chmod 400) so it is not world-readable.

📌 **Verification Checklist**

* ✅ .env file has POSTGRES\_DB and POSTGRES\_USER but **no** POSTGRES\_PASSWORD.
* ✅ ./secrets/postgres\_password.txt exists on host and has chmod 400.
* ✅ docker compose up -d starts the postgres service successfully.
* ✅ Inside the postgres container /run/secrets/postgres\_password exists and contains the expected password.
* ✅ Postgres logs show database initialization and readiness.

📌 **Notes & Best Practices**

* Removing POSTGRES\_PASSWORD from .env prevents accidental commit of secrets to VCS.
* Compose-native secrets (with file:) allow you to keep secret files out of the main repo while still mounting them at runtime.
* Do **not** commit ./secrets/postgres\_password.txt into source control. Add ./secrets/ to .gitignore.
* For images that natively read secrets, the wrapper is unnecessary: you could then read /run/secrets/<name> directly in the app. For the official postgres image, the small wrapper is the simplest safe approach.
* *No test cases are provided for this task*: it is for learning and practice only.

⚠️ Before moving on, keep the Compose services running or stop them with:

docker compose down



## 🐳 Task 6: Add Backend to Compose

🎯 **Goal** Wire the Spring Boot backend into your Compose setup so it connects to Postgres (using the custom network and envs/secrets configured earlier). The backend will be built from the local source (build:) for this learning phase and will mount a volume for persistent logs.

🛠️ **Steps**

1. Ensure you are in the project root and .env + ./secrets/postgres\_password.txt exist (from previous tasks).
2. Update docker-compose.yml to add the backend service (merge this snippet into your existing Compose file):

services:

postgres:

image: postgres:15

container\_name: postgres

environment:

POSTGRES\_DB: ${POSTGRES\_DB}

POSTGRES\_USER: ${POSTGRES\_USER}

secrets:

- postgres\_password

networks:

- app-net

entrypoint: [ "sh", "-c" ]

command: >

"export POSTGRES\_PASSWORD=$(cat /run/secrets/postgres\_password) &&

exec docker-entrypoint.sh postgres"

volumes:

- pgdata:/var/lib/postgresql/data

backend:

build:

context: ./iitb-course-backend

image: iitb-course-backend:local

container\_name: backend

ports:

- "8080:8080" # expose backend for host testing

environment:

SPRING\_DATASOURCE\_URL: ${SPRING\_DATASOURCE\_URL:-jdbc:postgresql://postgres:5432/${POSTGRES\_DB}}

SPRING\_DATASOURCE\_USERNAME: ${POSTGRES\_USER}

# Do NOT set SPRING\_DATASOURCE\_PASSWORD here; backend will read DB password from secret file if implemented

secrets:

- postgres\_password

networks:

- app-net

volumes:

- backend-logs:/app/logs # persist application logs (Logback configured to write here)

depends\_on:

- postgres

volumes:

pgdata:

backend-logs:

secrets:

postgres\_password:

file: ./secrets/postgres\_password.txt

networks:

app-net:

driver: bridge



1. Build and start services:

docker compose up --build -d



1. Watch startup logs (backend will apply migrations/seed data using the DB):

docker compose logs -f backend



1. Verify backend is reachable from host:

# health endpoint

curl -i http://localhost:8080/api/health

# list courses

curl -i http://localhost:8080/api/courses



1. Inspect persisted logs (inside container):

docker exec -it backend sh -c "ls -la /app/logs && tail -n 100 /app/logs/\*.log || true"



1. Stop services when done:

docker compose down



📖 **Explanation of snippet keywords**

* **build.context** → Points Compose to the backend source directory so the image is built locally from ./iitb-course-backend.
* **image** → Names the locally built image. Helpful for caching and later switching to Docker Hub images.
* **ports** → Maps backend port 8080 in container to 8080 on host, so you can curl it from EC2 or your laptop (http://<EC2\_IP>:8080).
* **environment** → Passes DB connection details. SPRING\_DATASOURCE\_PASSWORD is excluded for security; backend must read it from the secret file.
* **secrets** → Mounts the Postgres password at /run/secrets/postgres\_password. The backend must be configured (via wrapper or code) to use it.
* **volumes** → Persists logs at /app/logs so they survive container restarts.
* **depends\_on** → Ensures postgres starts before backend. Does not wait for readiness: proper sequencing via healthchecks will be added in Task 8.

📌 **Verification Checklist**

* ✅ docker compose up --build -d starts postgres and backend successfully.
* ✅ Backend logs show successful DB connection and migrations.
* ✅ curl http://localhost:8080/api/health returns ok.
* ✅ curl http://localhost:8080/api/courses returns seeded course data.
* ✅ Backend logs are persisted in the backend-logs volume across restarts (docker compose restart backend).

📌 **Notes**

* We use docker compose up --build -d to ensure the backend image is **rebuilt each time** you change the source code. If you skip --build, Compose will reuse the last built image and you may not see your changes.
* Later, when we switch to prebuilt Docker Hub images in the **final exercise**, you’ll only need docker compose up -d.
* If backend fails to authenticate, confirm it can read /run/secrets/postgres\_password. As a fallback, you can temporarily add SPRING\_DATASOURCE\_PASSWORD in .env for debugging, but secrets must be used for the graded exercise.
* *No test cases are provided for this task*: it is for learning and practice only.

## 🐳 Task 7: Add Frontend to Compose

🎯 **Goal** Integrate the React-based frontend into the Compose setup. The frontend will be built from local source (build:) and will connect to the backend API using an environment variable passed at container startup. Unlike Postgres, the backend and frontend **must** expose a port to the host so you can access it in your browser.

🛠️ **Steps**

1. Ensure you are in the project root (dockerlab\_activity5\_3TierApp).
2. Update docker-compose.yml to add the frontend service:

services:

postgres:

image: postgres:15

container\_name: postgres

environment:

POSTGRES\_DB: ${POSTGRES\_DB}

POSTGRES\_USER: ${POSTGRES\_USER}

secrets:

- postgres\_password

networks:

- app-net

entrypoint: [ "sh", "-c" ]

command: >

"export POSTGRES\_PASSWORD=$(cat /run/secrets/postgres\_password) &&

exec docker-entrypoint.sh postgres"

volumes:

- pgdata:/var/lib/postgresql/data

backend:

build:

context: ./iitb-course-backend

image: iitb-course-backend:local

container\_name: backend

ports:

- "8080:8080" # expose backend for outside api call

environment:

SPRING\_DATASOURCE\_URL: ${SPRING\_DATASOURCE\_URL:-jdbc:postgresql://postgres:5432/${POSTGRES\_DB}}

SPRING\_DATASOURCE\_USERNAME: ${POSTGRES\_USER}

secrets:

- postgres\_password

networks:

- app-net

volumes:

- backend-logs:/app/logs

depends\_on:

- postgres

frontend:

build:

context: ./iitb-course-frontend

image: iitb-course-frontend:local

container\_name: frontend

ports:

- "3000:80" # map container’s Nginx port 80 to host port 3000

environment:

REACT\_APP\_API\_BASE\_URL: "http://<EC2\_PUBLIC\_IP>:8080"

networks:

- app-net

depends\_on:

- backend

volumes:

pgdata:

backend-logs:

secrets:

postgres\_password:

file: ./secrets/postgres\_password.txt

networks:

app-net:

driver: bridge



1. Build and start all services:

docker compose up --build -d



1. Verify frontend logs:

docker compose logs -f frontend



1. Access the app in your browser using the EC2 public IP:

http://<EC2\_PUBLIC\_IP>:3000



1. The frontend should display the course list and interact with the backend API.
2. Stop services when done:

docker compose down



📖 **Explanation of snippet keywords**

* **build.context** → Builds the frontend image from ./iitb-course-frontend.
* **image** → Names the local frontend image (helps with caching).
* **ports "3000:80"** → Maps host port 3000 to container port 80 (Nginx). This is **necessary** because the frontend must be accessible from your laptop browser.
* **environment.REACT\_APP\_API\_BASE\_URL** → Configures the frontend to call backend APIs at http://<EC2\_PUBLIC\_IP>:8080. This is needed because a React application is served to the client's browser, the JavaScript code runs **on the client's machine**. Any API calls in the code will be executed by the browser. The browser doesn't know what 'backend' is, since it's a hostname that only exists within your Docker network..
* **jdbc:postgresql://postgres:5432/${POSTGRES\_DB}**} → In contrast, for DB calls DNS resolves postgres to the specific Postgres container IP in the same network.
* **networks** → Puts all three services (postgres, backend, frontend) on the same custom network. This removes the need for exposing ports for **backend** and **postgres**: they communicate internally via service names (backend, postgres).
* **depends\_on** → Ensures backend starts before frontend. Does not wait for readiness yet (healthchecks will be added later).

📌 **Verification Checklist**

* ✅ docker compose up --build -d starts postgres, backend, and frontend together.
* ✅ Logs show Nginx serving frontend build.
* ✅ Visiting http://<EC2\_IP>:3000 shows the React UI.
* ✅ Frontend fetches data from backend (/api/courses) without CORS issues.
* ✅ No explicit host port for **postgres** is needed since communication happens via the internal network.

📌 **Notes**

* **Frontend** and **Backend** must expose ports (3000:80,8080:8080) because your browser is outside Docker.
* **Postgres** does not need a port exposed: it only communicates with the backend through the Compose network.
* Keep using docker compose up --build -d while working with local source code. Later, when switching to prebuilt Docker Hub images in the **final exercise**, --build will not be required.
* *No test cases are provided for this task*: it is for learning and practice only.

## 🐳 Task 8: Healthchecks & Startup Sequencing

🎯 **Goal** Make service startup robust by adding **healthchecks** for Postgres, backend, and frontend, and ensure Compose starts dependent services only after their dependencies are healthy. This avoids race conditions (backend trying DB before DB is ready, frontend calling backend that hasn't finished booting).

🛠️ **Steps**

1. Ensure you are in the project root (dockerlab\_activity5\_3TierApp).
2. Update docker-compose.yml to add healthcheck endpoints::

services:

postgres:

image: postgres:15

container\_name: postgres

environment:

POSTGRES\_DB: ${POSTGRES\_DB}

POSTGRES\_USER: ${POSTGRES\_USER}

secrets:

- postgres\_password

networks:

- app-net

entrypoint: [ "sh", "-c" ]

command: >

"export POSTGRES\_PASSWORD=$(cat /run/secrets/postgres\_password) &&

exec docker-entrypoint.sh postgres"

volumes:

- pgdata:/var/lib/postgresql/data

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U ${POSTGRES\_USER} -d ${POSTGRES\_DB} -h localhost"]

interval: 10s

timeout: 5s

retries: 10

start\_period: 10s

backend:

build:

context: ./iitb-course-backend

image: iitb-course-backend:local

container\_name: backend

ports:

- "8080:8080" # expose backend for outside api call

environment:

SPRING\_DATASOURCE\_URL: ${SPRING\_DATASOURCE\_URL:-jdbc:postgresql://postgres:5432/${POSTGRES\_DB}}

SPRING\_DATASOURCE\_USERNAME: ${POSTGRES\_USER}

secrets:

- postgres\_password

networks:

- app-net

volumes:

- backend-logs:/app/logs

depends\_on:

postgres:

condition: service\_healthy

healthcheck:

test: ["CMD-SHELL", "curl -f http://localhost:8080/api/health || exit 1"]

interval: 10s

timeout: 5s

retries: 10

start\_period: 20s

frontend:

build:

context: ./iitb-course-frontend

image: iitb-course-frontend:local

container\_name: frontend

ports:

- "3000:80" # map container’s Nginx port 80 to host port 3000

environment:

REACT\_APP\_API\_BASE\_URL: "http://<EC2\_PUBLIC\_IP>:8080"

networks:

- app-net

depends\_on:

backend:

condition: service\_healthy

healthcheck:

test: ["CMD-SHELL", "curl -f http://localhost/ || exit 1"]

interval: 15s

timeout: 5s

retries: 8

start\_period: 15s

volumes:

pgdata:

backend-logs:

secrets:

postgres\_password:

file: ./secrets/postgres\_password.txt

networks:

app-net:

driver: bridge



🛠️ **Steps**

1. Save the docker-compose.yml (merge with your existing file; ensure .env and ./secrets/postgres\_password.txt exist).
2. Start the stack:

docker compose up --build -d



1. Watch services and their health status:

docker compose ps

# For detailed health JSON:

docker inspect --format='{{json .State.Health}}' postgres | jq

docker inspect --format='{{json .State.Health}}' backend | jq

docker inspect --format='{{json .State.Health}}' frontend | jq



1. Tail logs if any service becomes unhealthy:

docker compose logs -f postgres

docker compose logs -f backend

docker compose logs -f frontend



📖 **How condition: service\_healthy helps**

* healthcheck defines a command Docker runs periodically inside the container (e.g., pg\_isready for Postgres, curl to the backend health endpoint). Docker tracks the container’s health status (starting, healthy, or unhealthy).
* depends\_on with condition: service\_healthy tells Compose to wait to start the dependent service until the dependency’s health status is healthy. In the snippet above:
  + backend depends on postgres: { condition: service\_healthy }: Compose will delay starting the backend service until Postgres reports healthy.
  + frontend depends on backend: { condition: service\_healthy }: Compose will delay starting the frontend service until the backend reports healthy.
* This sequence reduces startup races: backend won’t attempt DB migration until DB is ready; frontend won’t send requests to backend before it’s ready.

📌 **Verification Checklist**

* ✅ docker compose up --build -d starts services.
* ✅ After a short period, docker compose ps shows services with healthy state.
* ✅ docker inspect .State.Health shows "Status": "healthy" for Postgres, backend, and frontend.
* ✅ Backend logs indicate migrations applied and readiness.
* ✅ Frontend UI is responsive once frontend becomes healthy.

📌 **Notes & Caveats**

* condition: service\_healthy relies on Compose honoring that field. In many environments and Compose versions this is supported; if you observe depends\_on not waiting, check your Docker Compose version and behavior.
* If your Compose version does **not** support condition: service\_healthy, simply remove the condition block and keep plain depends\_on (services will start in order, but not wait for health).

Example update:

backend:

depends\_on:

- postgres

frontend:

depends\_on:

- backend



* Tune interval, retries, and start\_period to match observed startup times: overly aggressive healthchecks can mark services unhealthy prematurely.
* Avoid placing secrets or sensitive data directly in healthcheck commands.
* *No test cases are provided for this task*: it is for learning and practice only.

## 🐳 Task 9: Restart Policies

🎯 **Goal** Learn how to configure container restart behavior in Compose using restart: policies so services recover automatically from failures or behave predictably after host reboots.

🛠️ **Steps**

1. **Add restart policies to your services** in docker-compose.yml. Common policies:

services:

postgres:

image: postgres:15

restart: unless-stopped

# ... (other config)

backend:

build: ./iitb-course-backend

restart: on-failure

# ... (other config)

frontend:

build: ./iitb-course-frontend

restart: always

# ... (other config)



1. **Meaning of common restart policies**

* no (default): Do not automatically restart the container.
* always: Always restart the container if it stops. If the container is stopped **manually**, it will be restarted when **Docker daemon restarts**.
* unless-stopped: Like always, but does **not** restart the container if it was stopped via docker stop (useful for interactive debugging).
* on-failure[:max-retries]: Restart only if the container exits with a non-zero exit code. Optionally supply :max-retries (e.g., on-failure:5).

1. **Apply and observe behavior**

# Start services

docker compose up -d

# Simulate failures:

# Example: kill the backend process inside container (or stop the container)

docker exec -it backend sh -c "kill 1" || true

# Or stop a service from host

docker stop backend

# Observe restart behavior

docker ps --filter name=backend --format "table {{.Names}}\t{{.Status}}"

docker logs -f backend



1. **Test host reboot behavior (optional / caution)** Rebooting the host will show how always vs unless-stopped behave:

* With always, containers come back up after Docker daemon starts.
* With unless-stopped, containers that were not manually stopped will restart; containers manually stopped remain stopped.

📖 **Explanation & Rationale**

* Use **on-failure** for services that should automatically retry transient errors (e.g., temporary DB connection issues).
* Use **unless-stopped** as a sensible default for long-running services you want to persist across host reboots but still allow manual stop during maintenance.
* Use **always** for critical services that must be running regardless of how they were stopped (use carefully: this can complicate debugging).

📌 **Verification Checklist**

* ✅ Services configured with restart: behave as expected after container termination (they restart or remain stopped according to policy).
* ✅ After simulating a crash (kill or docker stop), docker ps shows the container restarted per policy.
* ✅ After a host reboot, containers with always or unless-stopped return to running state if they were not previously manually stopped (verify after reboot).

📌 **Notes & Best Practices**

* For production orchestration (Kubernetes/Swarm), restart semantics differ: restart policies here apply to single-host Docker Compose setups.
* Be careful with restart: always in development: it can make iterative debugging harder because containers automatically respawn. Use unless-stopped for a friendlier development experience.
* Combine restart policies with robust **healthchecks** (Task 8) so failing services don’t repeatedly restart without addressing root causes.
* *No test cases are provided for this task*: it is for learning and practice only.

## 🐳 Task 10: Scaling Backend

🎯 **Goal** Understand the idea of scaling services with multiple replicas in Docker Compose, its challenges in this 3-tier demo, and what can be achieved in real systems.

📖 **Concept**

Scaling allows you to run **multiple replicas of a service**. Docker’s embedded DNS will return multiple IPs for the same service name, enabling internal load distribution across replicas.

⚠️ **Challenges in our 3-tier app**

1. **Container names** → You cannot use container\_name: with scaling (each replica must have a unique name).
2. **Port publishing** → Only one container can bind to a host port (e.g., 8080). When scaling replicas, they cannot all expose the same host port.
3. **Testing load distribution** → Our backend has no endpoint that returns replica-specific info (like container ID or hostname), so you cannot observe which replica served the request.
4. **Frontend dependency** → The frontend expects to call backend:8080 inside the Docker network, so scaling is only useful internally unless a reverse proxy/load balancer is added.

💡 **What scaling enables in real systems**

* Run multiple replicas of the backend → distribute load and improve fault tolerance.
* Use an internal DNS round-robin (Compose built-in) or add a reverse proxy (NGINX, Traefik) for external load-balancing.
* Combine scaling with **stateless services** (no in-memory session state) to make replicas interchangeable.
* Useful for **high availability**: if one replica crashes, others continue serving.

📌 **Notes**

* Scaling in Compose is configured at runtime only:

docker compose up --build -d --scale backend=2



* The deploy.replicas field exists in the Compose spec but is **ignored** by plain Docker Compose (only works in Swarm/Kubernetes).
* For production-grade scaling and load balancing, pair this with a reverse proxy or move to an orchestrator (Swarm/K8s).

*No test cases are provided for this task: it is for learning and practice only.*

## 🐳 Task 11: Logging Drivers

🎯 **Goal** Understand what Docker logging drivers are, and how they differ from application log files written to mounted volumes.

📖 **Concept**

* **Logging drivers** manage how Docker captures a container’s **stdout/stderr** output.

services:

backend:

logging:

driver: "json-file"

options:

max-size: "10m"

max-file: "3"



* + This stores container logs in JSON files on the host and rotates them after 10 MB.
* Other drivers forward logs to external systems (syslog, fluentd, none).
* **Mounted log volumes** (like /app/logs in our backend) are **different**:
  + The application itself writes files to disk.
  + Rotation/retention is handled by the app’s logging framework (Logback in our case), not by Docker.

📌 **Notes**

* Use **logging drivers** to control and forward container output (good for central log collection).
* Use **mounted volumes** for application-specific log files (good for persistence and analysis).
* In practice, many teams configure apps to log to **stdout** and let Docker drivers handle collection + forwarding.

*No test cases are provided for this task: it is for learning and practice only.*

## 🐳 Task 12: Override Files (Dev vs Prod)

🎯 **Goal** Learn how to manage different configurations for **development** and **production** using Docker Compose override files, so you don’t need to constantly edit the base docker-compose.yml.

📖 **Concept**

Docker Compose allows you to **combine multiple YAML files**. The rules are simple:

* **docker-compose.yml** → always the **base file**.
* **docker-compose.override.yml** → if present, it is **applied automatically** on top of the base when you run docker compose up.
* **Any other override file** (e.g., docker-compose.prod.yml, docker-compose.dev.yml) → used only when explicitly passed with -f.

👉 Order matters: later files **override or extend** settings from earlier files.

🛠️ **Steps**

1. **Create a base docker-compose.yml** (production-like defaults):

services:

backend:

image: <dockerhub-username>/iitb-course-backend:latest

ports:

- "8080:8080"

restart: unless-stopped



1. **Create a docker-compose.override.yml** (development tweaks). Since Compose loads this file automatically, you don’t need to pass -f in commands.

services:

backend:

build: ./iitb-course-backend

volumes:

- ./iitb-course-backend/src:/app/src # mount local code

environment:

SPRING\_PROFILES\_ACTIVE: dev

restart: "no"



✅ Here, build: from override.yml **replaces** the image: from the base file. ✅ The volumes: and environment: are **added** on top.

1. **Create a docker-compose.prod.yml** (explicit production overrides). This is not auto-loaded; you must specify it manually.

services:

backend:

environment:

SPRING\_PROFILES\_ACTIVE: prod

logging:

driver: "json-file"

options:

max-size: "10m"

max-file: "3"



1. **Run in development** (base + override automatically):

docker compose up --build -d



1. **Run in production** (base + prod override):

docker compose -f docker-compose.yml -f docker-compose.prod.yml up -d



📖 **Which file overrides which?**

* Default behavior:

docker-compose.yml ← base

docker-compose.override.yml ← applied automatically on top



* With custom files:

docker-compose.yml ← base

docker-compose.prod.yml ← applied on top when passed with -f



* When multiple files are passed: the **last one wins** for conflicting keys.

Example run order:

docker compose -f docker-compose.yml -f docker-compose.prod.yml up -d



Here, docker-compose.prod.yml overrides both the base and the default override.

📌 **Verification Checklist**

* ✅ Development run uses **override.yml** automatically → builds from source, mounts volumes, no auto-restart.
* ✅ Production run with -f docker-compose.prod.yml uses prebuilt images, enables logging driver, and restart policy.

📌 **Notes**

* Use **docker-compose.override.yml** for developer convenience (auto-applied).
* Use **docker-compose.prod.yml** (or dev, staging) for explicit environment configs.
* Compose never merges arrays like ports:: they are replaced. Keys like environment: and volumes: are merged.
* *No test cases are provided for this task*: it is for learning and practice only.

## 🏁 Task 13: Final Exercise: Write a Complete Compose File

🎯 **Goal** Bring together all concepts from this activity by writing a complete docker-compose.yml from scratch. This file should run the **3-tier course app** using prebuilt images from Docker Hub, with all best practices applied.

🛠️ **Instructions**

1. **Clean your environment** (from your local clab terminal): Reset your lab environment to avoid conflicts:

# run the provided setup which resets EC2 and extracts the app

bash system-clean-init.sh

# now SSH into EC2

ssh -i secret-key.pem ubuntu@<EC2\_PUBLIC\_IP>

cd /home/ubuntu/docker\_lab/dockerlab\_activity5\_3TierApp



1. **Write a new docker-compose.yml** in the project root. Follow these requirements **exactly** so that the autograder can validate:
   * **Services and images**
     + course-db → postgres:15
     + course-backend → soumik13/iitb-course-backend:v2
     + course-frontend → soumik13/iitb-course-frontend:v2
   * **Network**
     + All services must be attached to a custom network → course-net.
   * **Volumes**
     + course-db-data → for Postgres data.
     + course-backend-logs → for backend logs.
   * **Secrets**
     + Secret name: course\_db\_password.
     + File path: ./secrets/course\_db\_password.txt.
   * **Environment variables**
     + Define POSTGRES\_DB and POSTGRES\_USER in .env.
     + Use the secret for POSTGRES\_PASSWORD.
     + Backend must connect to DB using the Compose service name (course-db).
     + Frontend must use env variable REACT\_APP\_API\_BASE\_URL pointing to backend (http://<EC2\_PUBLIC\_IP>:8080).
   * **Ports**
     + Expose only:
       - course-frontend → 3000:80
       - course-backend → 8080:8080
       - course-db → **no host port**
   * **Add the following for each service:**
     + **Healthchecks** → ensure each service is actually “ready” (DB → pg\_isready, backend → /api/health, frontend → /).
     + **Restart policies** → e.g., unless-stopped for DB/frontend, on-failure for backend.
     + **depends\_on with service\_healthy** → backend waits for DB to be healthy, frontend waits for backend to be healthy.
2. **Skeleton (fill in the TODOs yourself):**

services:

course-db:

# TODO: add image (postgres:15)

# TODO: add environment variables (POSTGRES\_DB, POSTGRES\_USER)

# TODO: mount secret for password

# TODO: mount volume for db data

# TODO: attach to custom network

# TODO: add restart policy

# TODO: add logging driver with rotation

# TODO: add healthcheck using pg\_isready

course-backend:

# TODO: add image (soumik13/iitb-course-backend:v2)

# TODO: add environment variables (JDBC URL, DB username)

# TODO: mount secret for db password

# TODO: expose port 8080

# TODO: mount volume for backend logs

# TODO: attach to custom network

# TODO: add depends\_on with service\_healthy for course-db

# TODO: add restart policy

# TODO: add logging driver with rotation

# TODO: add healthcheck (curl /api/health)

course-frontend:

# TODO: add image (soumik13/iitb-course-frontend:v2)

# TODO: add environment variable (REACT\_APP\_API\_BASE\_URL)

# TODO: expose port 3000:80

# TODO: attach to custom network

# TODO: add depends\_on with service\_healthy for course-backend

# TODO: add restart policy

# TODO: add logging driver with rotation

# TODO: add healthcheck (curl /)

volumes:

# TODO: define course-db-data volume

# TODO: define course-backend-logs volume

secrets:

# TODO: define course\_db\_password secret with file path ./secrets/course\_db\_password.txt

networks:

# TODO: define course-net network (bridge driver)



📌 **Verification Checklist**

* ✅ course-db is healthy.
* ✅ course-backend is healthy and responds at http://<EC2\_PUBLIC\_IP>:8080/api/courses.
* ✅ course-frontend loads at http://<EC2\_PUBLIC\_IP>:3000 and fetches courses from backend.
* ✅ Data persists in course-db-data, and backend logs persist in course-backend-logs.
* ✅ Restart policies and logging drivers are correctly configured.
* ✅ Only required ports are exposed (3000, 8080).

### 🔐 Required file names & exact EC2 paths (must match exactly)

All files must be placed inside the extracted lab folder on the EC2 instance:

/home/ubuntu/docker\_lab/dockerlab\_activity5\_3TierApp/docker-compose.yml

/home/ubuntu/docker\_lab/dockerlab\_activity5\_3TierApp/.env

/home/ubuntu/docker\_lab/dockerlab\_activity5\_3TierApp/secrets/course\_db\_password.txt



Make sure course\_db\_password.txt has permission chmod 400.

## 🔁 Copy files back to your local clab workspace for submission

After verifying everything on EC2, **copy the three files** **docker-compose.yml,.env,secret/course\_db\_password.txt** from EC2 to your **local clab workspace** (so you can submit / keep copies).

📌 **Final Step** : submit it for testing.

The **autograder will run it and validate each checkpoint automatically.**

## 🧹 Cleanup

Stop all services and reset workspace:

# Stops and removes all containers, networks, and services from your project

# Also deletes associated named volumes and orphaned containers for a complete cleanup.

docker compose down --volumes --remove-orphans

# From clab run it

bash system-clean-init.sh



## 🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

exit



1. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

## 📚 Appendix: Extra Compose Concepts

Here are some additional Docker Compose features that are not part of this lab but are useful to know for real-world projects:

| **Concept** | **What it Does** | **Why it’s Useful** |
| --- | --- | --- |
| **Profiles** | Allows you to group services and start only selected ones. | Avoids always running optional services (e.g., monitoring, debugging). |
| **External Networks & Volumes** | Lets containers join pre-created networks or use existing volumes. | Enables multiple Compose projects or standalone containers to share data or talk over the same network. |
| **Resource Limits** | Restricts how much CPU/memory a service can consume. | Prevents one service from consuming all host resources (works in Swarm/K8s, not plain Compose). |
| **Build Caching** | Controls whether Docker uses cached layers during builds. | Ensures a fresh build when debugging or avoiding stale files. |
| **Configs** | Mounts non-sensitive config files (e.g., Nginx conf) into containers. | Cleanly manages config files without baking them into images. Useful for apps needing flexible configs. |

## 🏆 Congratulations: You’ve completed Activity 5!

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

6️⃣ Activity 6: Make Dockerfiles Production Ready

# Activity 6: Make Dockerfiles Production Ready

🎯 **Objective:** Learn techniques to build smaller, faster, and more secure container images by applying production-ready Dockerfile patterns: using a compact, hands-on Go worked example and a short Node exercise.

## ✅ Prerequisites

* Local **clab** workspace (you’ll run ./system-clean-init.sh from your local clab terminal).
* An **Ubuntu 24.04 LTS EC2** instance with SSH access and Docker installed (Docker Engine + Docker Compose available).
* secret-key.pem (SSH key) and the EC2 public IP address.
* Basic familiarity with Docker CLI, docker build, docker run, and tar commands.
* Recommended: Enable BuildKit when building (DOCKER\_BUILDKIT=1) for cache & --mount=type=cache support.

## 📂 Lab Setup

**Goal:** Reset the EC2 docker\_lab workspace and automatically ship the **Activity6** source archive to the EC2 instance.

**Important:** From your **local clab terminal** (/home/labDirectory), simply run the provided script:

bash ./system-clean-init.sh



This script will:

1. **Clean/reset** your EC2 docker\_lab workspace (remove old containers, images, volumes).
2. **Transfer and extract** the Activity6\_source.tar.xz archive into the correct location on EC2.

After running it, SSH into your EC2 instance and verify:

ssh -i secret-key.pem ubuntu@<EC2\_PUBLIC\_IP>

cd /home/ubuntu/docker\_lab/Activity6



The EC2 instance should look like this:

$ pwd

/home/ubuntu/docker\_lab/Activity6

$ tree -a

.

├── go-app

│ ├── Dockerfile.naive

│ ├── README.md

│ ├── go.mod

│ └── main.go

└── node-exercise

├── Dockerfile.naive

├── README.md

├── app.js

└── package.json

3 directories, 8 files



If the tree above matches, you are ready to proceed.

## 📦 What’s in the tarball

* go-app/: tiny Go web app + **naive** Dockerfile (Dockerfile.naive) and README.md with build/run/test instructions.
* node-exercise/: tiny Express app + **naive** Dockerfile (Dockerfile.naive) and README.md with build/run/test instructions.

Students will use these naive Dockerfiles as the starting point for optimization.

## 📝 Tasks (overview)

This short activity contains two main items:

1. **Worked example (Go):** step-by-step build of a production-ready Dockerfile demonstrating:
   * Multi-stage builds
   * FROM scratch (and distroless) final stages
   * Base-image slimming (ubuntu vs alpine vs distroless)
   * Build cache optimization (--mount=type=cache)
   * Layer ordering & minimizing layers
   * Reducing attack surface (minimal deps + drop root)
   * Security scanning (docker scan / trivy)
   * Benchmarking & metrics (build time, image size, layer info, startup time)
2. **Student exercise (Node):** given a naive Dockerfile.naive, the student must produce Dockerfile.prod that applies similar optimizations (multi-stage, cache, layer ordering, non-root, healthcheck) and verify by building and running the optimized image.

## 🐳 Task 1: Worked example: Make the **Go app** production-ready (step-by-step)

🎯 **Goal:** Starting from the naive go-app/Dockerfile.naive, we will apply one optimization at a time so you can see *why* each change helps. By the end you’ll have a production-ready Dockerfile and a clear comparison (size, build speed, layers, startup) between the naive and optimized images.

### 📌 GO app: general setup & optimization targets

**Where you are (files / location):**

/home/ubuntu/docker\_lab/Activity6/go-app

├── Dockerfile.naive

├── README.md

├── go.mod

└── main.go



**How the Go build works (brief):**

* go build compiles your Go sources into a single binary.
* Go supports modules (go.mod / go.sum): module downloads are a significant cost during builds.
* Static builds are possible (CGO\_ENABLED=0) and desirable for scratch images because they don't depend on libc.
* The final artifact that needs to run is usually just a single binary: everything else (compiler, toolchain, caches) is only needed during build.

**Why optimize the naive Dockerfile (targets):**

1. **Image size**: naive images often contain the full Go toolchain (hundreds of MB). Shipping only the binary drastically reduces size.
2. **Build speed (iterative)**: separate dependency download and use caching (BuildKit --mount=type=cache) so repeated builds are fast.
3. **Layer caching**: ordering COPY to maximize cache re-use (copy go.mod first).
4. **Attack surface**: final image should not include compilers or shells; run as non-root.
5. **Debuggability vs minimality**: provide variants (alpine for debugging, scratch/distroless for production).
6. **Reproducibility & metadata**: add labels, pin base images, strip binary symbols.
7. **Health & observability**: include HEALTHCHECK and expose port in Dockerfile or compose.

### 🔬 Baseline: inspect the naive Dockerfile

Dockerfile.naive (baseline):

# Dockerfile.naive

FROM golang:1.20

WORKDIR /app

COPY . .

RUN go build -o server .

EXPOSE 8080

CMD ["./server"]



**Baseline drawbacks:**

* Final image contains full golang:1.20 runtime (toolchain, package manager, shells): large (~700–900MB depending on base) (Check image size).
* Layer cache is poor because COPY . . invalidates cache for dependency installation even if only unrelated **source files** changed (Check image build time).
* No stripping of binary; likely larger binary size.
* Runs as root by default.
* No BuildKit cache mounts used for modules.

### Step 1: Improve layer ordering & separate dependency download (for making docker build faster)

**Goal:** Copy go.mod or go.sum first, download modules (so this layer stay cached when only source changes), then copy the rest of the source.

**Dockerfile snippet (Dockerfile.step1):**

# step1: ordering to improve cache

FROM golang:1.20 AS builder

WORKDIR /src

# copy go.mod/go.sum first and download go dependencies

COPY go.mod ./

RUN go mod download

# copy the rest and build

COPY . .

RUN go build -o /server ./...



**Build & verify:**

docker build -t go-app:step1 -f Dockerfile.step1 .

# Change a source file and rebuild: dependency layer will remain cached

docker build -t go-app:step1 -f Dockerfile.step1 .



**Why this helps:** go mod download becomes cacheable. If you only change main.go, Docker can reuse the module-download layer and rebuild faster.

### Step 2: Use BuildKit Cache Mounts for Faster Builds (for faster iterative builds)

**Goal:** Leverage Docker's BuildKit to cache dependencies and build artifacts, making repeated builds much faster. Requires BuildKit enabled (DOCKER\_BUILDKIT=1).

**Dockerfile (Dockerfile.step2):**

# syntax=docker/dockerfile:1.4

FROM golang:1.20 AS builder

WORKDIR /src

# cache Go module downloads and build cache

COPY go.mod ./

RUN --mount=type=cache,target=/go/pkg/mod \

--mount=type=cache,target=/root/.cache/go-build \

go mod download

COPY . .

RUN --mount=type=cache,target=/go/pkg/mod \

--mount=type=cache,target=/root/.cache/go-build \

go build -o /server ./...



The **# syntax=docker/dockerfile:1.4** line is crucial as it enables the advanced BuildKit features like cache mounts. The --mount=type=cache flag creates a temporary, isolated cache volume that persists between builds. This is a significant improvement over traditional caching methods, which rely on layers and are less granular.

### How It Works

The go mod download and go build commands use these cache mounts to speed up the process:

* **--mount=type=cache,target=/go/pkg/mod**: This caches Go modules, so they only need to be downloaded once.
* **--mount=type=cache,target=/root/.cache/go-build**: This caches compiled objects and build artifacts.

Unlike the traditional Docker cache where a single change to go.mod invalidates the entire cache layer, **BuildKit's cache mounts are more intelligent and granular**, only rebuilding what's absolutely necessary. This results in more efficient and reproducible builds.

### Build Commands

# First build (downloads modules and populates the cache)

time DOCKER\_BUILDKIT=1 docker build -t go-app:step2 -f Dockerfile.step2 .

# Subsequent builds (much faster, as the cache is used)

time DOCKER\_BUILDKIT=1 docker build -t go-app:step2 -f Dockerfile.step2 .



### Step 3: Make the Binary Smaller: Static Build & Strip Symbols (Size Compression)

**Goal:** Reduce the final binary size by creating a statically linked executable and removing unnecessary metadata.

**Dockerfile (Dockerfile.step3):**

# syntax=docker/dockerfile:1.4

FROM golang:1.20 AS builder

WORKDIR /src

COPY go.mod ./

RUN --mount=type=cache,target=/go/pkg/mod \

go mod download

COPY . .

# build static, stripped binary

ENV CGO\_ENABLED=0

RUN --mount=type=cache,target=/go/pkg/mod \

go build -trimpath -ldflags="-s -w" -o /server ./...



This process creates a self-contained, lightweight binary that is easier to deploy and has a smaller attack surface.

### How It Works

This build process uses two key flags to reduce the binary size:

* **ENV CGO\_ENABLED=0**: This creates a **statically linked** binary. It tells the Go compiler not to use the host C library (libc) but instead to package all necessary C code directly into the final executable. The resulting binary is self-sufficient and does not require any external dependencies.
* **-ldflags="-s -w"**: These linker flags remove metadata from the binary.
  + **-s**: This flag strips the **symbol table**. The symbol table contains information used by debuggers to link addresses back to function names, which is unnecessary for a production binary.
  + **-w**: This flag strips the **DWARF debug information**, further reducing the size by removing data used for stack traces and debugging.
* **-trimpath** : This Go linker flag removes all file system paths from the compiled binary.
  + Without -trimpath, the final binary contains absolute file paths from the build machine.
  + This is significant because it makes your Go builds reproducible and more secure across different machines while using **statically linked binary**.

### Build & Check Binary Size (Optional)

DOCKER\_BUILDKIT=1 docker build --target=builder -t go-app:builder -f Dockerfile.step3 .

# Inspect builder container filesystem

docker run --rm --entrypoint ls go-app:step2 -l /server || true

# vs

docker run --rm --entrypoint ls go-app:builder -l /server || true

# Or extract binary from image to check size:

docker create --name tmp go-app:builder

docker cp tmp:/server ./server\_tmp

docker rm tmp

ls -lh server\_tmp



**Why this helps:** A statically linked and stripped Go binary can be extremely small—often just a few megabytes—as it no longer relies on system libraries like libc and has all non-essential debugging information removed. The **--target=builder** flag is used to stop the build at this stage, creating a temporary image that contains only the builder environment and the final binary. This allows you to inspect the binary's size before proceeding to the final, minimal image.

Note: *The docker create command is used to create a new Docker container from a specified image, without starting it. This differs from docker run, which creates the container and immediately starts it.*

### Step 4: Multi-stage Builds for Minimal Images (Size Compression)

**Goal:** This step uses a **multi-stage build** to create a final, production-ready image that contains only the necessary application binary, stripping away all build dependencies.

**Dockerfile (Dockerfile.step4):**

# syntax=docker/dockerfile:1.4

FROM golang:1.20 AS builder

WORKDIR /src

COPY go.mod ./

RUN --mount=type=cache,target=/go/pkg/mod go mod download

COPY . .

ENV CGO\_ENABLED=0

RUN --mount=type=cache,target=/go/pkg/mod \

go build -trimpath -ldflags="-s -w" -o /server ./...

# Final stage: scratch

FROM scratch AS runtime

COPY --from=builder /server /server

### **Alternative final stage (Dockerfile.step4\_distroless):**

# syntax=docker/dockerfile:1.4

FROM golang:1.20 AS builder

WORKDIR /src

COPY go.mod ./

RUN --mount=type=cache,target=/go/pkg/mod go mod download

COPY . .

ENV CGO\_ENABLED=0

RUN --mount=type=cache,target=/go/pkg/mod \

go build -trimpath -ldflags="-s -w" -o /server ./...

### # Final stage: distroless FROM [gcr.io/distroless/static](http://gcr.io/distroless/static) COPY --from=builder /server /server

###  How It Works

A multi-stage build uses multiple FROM instructions in a single Dockerfile. Each FROM starts a new stage, and you can selectively copy artifacts from a previous stage without bringing along its entire filesystem.

* **AS <stagename>**: This keyword assigns a name to a build stage (e.g., FROM golang:1.20 AS builder). This allows you to reference the stage later.
* **--from=<stagename>**: This flag, used with the COPY instruction, specifies that you want to copy files from a named stage instead of the current one. Here, COPY --from=builder /server /server copies only the compiled /server binary from the builder stage into the new runtime stage.
* **--target=<stagename>**: This flag is a build-time option (docker build --target=builder) that allows you to stop the build at a specific stage, which is useful for debugging or creating intermediate images.

### Minimal Base Images: scratch vs. distroless

The final stage uses an extremely small base image, either **scratch** or **distroless**, to minimize the final image size and reduce the attack surface.

* **scratch**: This is the smallest possible base image, an **empty image**. It contains no files, no directories, no operating system, and no shell. It's ideal for statically-linked binaries like the one we've built, as it requires no external dependencies.
* **distroless**: An alternative to scratch from Google. While slightly larger than scratch, it contains a minimal set of system libraries (glibc, ssl) and is useful for dynamically-linked binaries that might need these libraries. It provides a tiny, secure runtime environment with no package managers or shells, making it safer than traditional base images.

### Build Commands

# Build with the scratch final stage

DOCKER\_BUILDKIT=1 docker build -t go-app:prod-scratch -f Dockerfile.step4 .

# Build with the distroless final stage

DOCKER\_BUILDKIT=1 docker build -t go-app:prod-distroless -f Dockerfile.step4\_distroless .



**Why this helps:** The final image contains only your application binary and its required runtime dependencies, removing the large build environment (compiler, development tools, package manager, and shell). This drastically reduces the image size, improves security by eliminating unnecessary software, and makes the image faster to push and pull.

### Step 5: Add Metadata, Healthcheck, and a Non-Root User

**Goal:** This step focuses on adding crucial operational metadata and security configurations to the final production image.

**Complete Dockerfile (Dockerfile.semifinal):**

# syntax=docker/dockerfile:1.4

FROM golang:1.20 AS builder

WORKDIR /src

COPY go.mod ./

RUN --mount=type=cache,target=/go/pkg/mod \

go mod download

COPY . .

ENV CGO\_ENABLED=0

RUN --mount=type=cache,target=/go/pkg/mod \

go build -trimpath -ldflags="-s -w" -o /server ./...

# --------------------------------------------------------

# Final stage (scratch) with metadata and healthcheck

FROM scratch AS runtime

LABEL org.opencontainers.image.title="go-app" \

org.opencontainers.image.version="v1.0.0" \

org.opencontainers.image.description="Tiny Go service"

COPY --from=builder /server /server

# numeric non-root UID

USER 10001

EXPOSE 8080

HEALTHCHECK --interval=10s --timeout=2s --start-period=5s CMD ["/server","--healthcheck"]

# Add ENTRYPOINT to run the server binary

ENTRYPOINT ["/server"]



### How It Works

These final additions enhance the image's security and operability:

* **LABEL**: This adds **Open Container Initiative (OCI) labels**, providing standardized metadata like title, version, and description. This helps with image discoverability and management.
* **USER 10001**: This sets the container to run as a **non-root user**, a key security practice. Using a numeric UID like 10001 is standard for scratch images since they contain no /etc/passwd file.
* **EXPOSE 8080**: This declares that the container listens on port 8080. While it doesn't publish the port to the host, it serves as documentation and is used by other Docker features.
* **HEALTHCHECK**: This defines a command to check the container's health. For a scratch image, you cannot use a shell or tools like curl, so the health check must be a **statically compiled binary that returns a non-zero exit code on failure.** Our example assumes the /server binary has a --healthcheck flag for this.

### Build and Run the Final Image

With the complete Dockerfile saved as Dockerfile.semifinal, you can now build the image and run the container.

# Build the final image with a tag

DOCKER\_BUILDKIT=1 docker build -t go-app:semifinal -f Dockerfile.semifinal .

# Run the container, mapping the port

docker run -d --name go-server -p 8080:8080 go-app:semifinal

# Check the container's health status

docker inspect --format='{{.State.Health.Status}}' go-server



The **docker run** command launches the container, publishing the internal port 8080 to the host's 8080. The **docker inspect** command then checks the container's health status, which Docker determined by running the HEALTHCHECK command you defined in the Dockerfile.

### Step 6: Final: combine all steps into Dockerfile.prod

Now assemble the previous progressive steps into the final Dockerfile.prod. (You built it incrementally above; this file is the final product that glues everything together.)

# Dockerfile.prod (final)

# syntax=docker/dockerfile:1.4

###################################

# Builder

###################################

FROM golang:1.20 AS builder

WORKDIR /src

# Copy go.mod first to leverage cache

COPY go.mod ./

# Use BuildKit cache for module downloads and go build cache

RUN --mount=type=cache,target=/go/pkg/mod \

--mount=type=cache,target=/root/.cache/go-build \

go mod download

# Copy source and build static, stripped binary

COPY . .

ENV CGO\_ENABLED=0

RUN --mount=type=cache,target=/go/pkg/mod \

--mount=type=cache,target=/root/.cache/go-build \

go build -trimpath -ldflags="-s -w" -o /server ./...

###################################

# Runtime (scratch)

###################################

FROM scratch AS runtime

LABEL org.opencontainers.image.title="go-app" \

org.opencontainers.image.version="v1.0.0" \

org.opencontainers.image.licenses="MIT"

COPY --from=builder /server /server

# Run as non-root numeric UID

USER 10001

EXPOSE 8080

HEALTHCHECK --interval=10s --timeout=2s --start-period=5s CMD ["/server","--healthcheck"]

ENTRYPOINT ["/server"]



Note: If your binary does not accept --healthcheck, modify the HEALTHCHECK to call the binary appropriately or use a minimal alpine final stage with curl to /health.

### ✅ How to build & measure (commands you will run on EC2)

**Build the baseline (naive) image:**

# baseline

time docker build -t go-app:naive -f Dockerfile.naive .



**Build the final optimized image (use BuildKit):**

# optimized (first build may download modules)

DOCKER\_BUILDKIT=1 docker build -t go-app:prod -f Dockerfile.prod .

# second build (should be faster due to cached mounts)

time DOCKER\_BUILDKIT=1 docker build -t go-app:prod -f Dockerfile.prod .



**Measure image size and layers:**

# sizes

docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}" go-app:naive

docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}" go-app:prod

# layers (count)

docker history go-app:naive --no-trunc --format '{{.ID}} {{.Size}}' | wc -l

docker history go-app:prod --no-trunc --format '{{.ID}} {{.Size}}' | wc -l



**Startup / healthcheck / non-root verification:**

# run baseline and optimized on different ports

docker run -d --name go-naive -p 8085:8080 go-app:naive

docker run -d --name go-prod -p 8086:8080 go-app:prod

# wait a few seconds then check

curl -fsS http://localhost:8085/health || echo "naive no response"

curl -fsS http://localhost:8086/health || echo "prod no response"

# verify non-root (inspect image config)

docker image inspect go-app:prod --format='User={{.Config.User}}'

# for running container (if it has shell): docker exec -it go-prod id -u



**Security scanning (Trivy):**

# run trivy (containerized) and capture output

docker run --rm -v /var/run/docker.sock:/var/run/docker.sock aquasec/trivy:latest image go-app:prod > trivy\_goapp\_prod.txt  
  
docker run --rm -v /var/run/docker.sock:/var/run/docker.sock aquasec/trivy:latest image --skip-db-update go-app:naive > trivy\_goapp\_naive.txt



### ✅ Testing and Verification

Testing an optimized Docker image involves more than just ensuring it runs. You must verify that the optimizations, such as a reduced size and enhanced security, have been successfully applied. The process involves a **comparison test** between a **naive image** (built without any optimizations) and the **final optimized image**. This difference is the proof that your work has paid off.

* **Size & Layer Measurement**: The docker images and docker history commands allow you to directly compare the size and number of layers. The naive image will be significantly larger and have more layers due to its large golang base and a history of build commands, while the optimized image will be tiny and have only one or two layers.
* **Startup & Healthcheck**: Running the two containers side-by-side verifies that both are functional. The curl command confirms the application's availability, and docker inspect validates that the HEALTHCHECK from the Dockerfile.prod is correctly configured and passing.
* **Non-Root Verification**: The USER instruction is a critical security feature. docker image inspect is used to confirm that the Config.User is set to 10001, proving that the container will run with reduced privileges. This is a simple but vital security check.

### 🛡️ Security Scanning with Trivy

**Trivy** is a popular, open-source **container security scanner**. It analyzes container images for vulnerabilities, misconfigurations, and other security issues. The docker run command provided runs Trivy in a container, mounts the host's Docker socket, and then scans your go-app:prod image.

* **Why it's useful**: A tiny scratch-based image is inherently secure because it contains no operating system, shell, or package manager. This means Trivy will find almost zero vulnerabilities, as there are no packages with known CVEs to report. Scanning the naive image, however, would likely find hundreds of vulnerabilities from the golang:1.20 base image, demonstrating the massive security advantage of your optimized build. The difference in scan results is a powerful testament to the value of a multi-stage, static binary build.

### 

### Comparison: naive vs optimized: what to expect (how to interpret results)

Run the measurement commands above and compare. Below is an **example** comparison (sample numbers: your EC2 / network environment may differ). Use these as guidance for expected improvements.

| **Aspect** | **Baseline (Dockerfile.naive): example** | **Optimized (Dockerfile.prod): example** | **Why optimized is better** |
| --- | --- | --- | --- |
| Image size (human) | 850 MB (golang base) | 6 MB (scratch with static binary) | The final image contains only the stripped, statically linked binary, completely removing the heavy Go compiler and operating system toolchain. |
| Build time: 1st build | 28 s | 32 s | The optimized build may be slightly slower on the first run due to the extra build stage, but overall is similar. |
| Build time: 2nd build | 25 s | 2–5 s | The **BuildKit cache mounts** make subsequent builds **dramatically faster** by reusing downloaded modules and compiled build artifacts. |
| Layer count | 10+ layers | 4–6 layers | The multi-stage build discards all intermediate layers from the heavy builder stage, resulting in a significantly slimmer final image with fewer layers. |
| Startup time | ~0.2 s | ~0.05–0.2 s | A smaller binary with no OS overhead can lead to a slightly faster startup time, reducing latency. |
| Attack surface | High (toolchain + shell) | Minimal (only binary) | A smaller image means a drastically reduced number of installed packages, which translates to fewer potential CVE sources and a more secure image. |
| Debuggability | Easy (shell available) | Harder (scratch no shell) | The scratch base image has no shell or debugging tools. An alpine variant provides a trade-off, offering a minimal shell for debugging. |
| Trivy findings (where present) | Many base-image CVEs | Very few in runtime image | The golang base image carries hundreds of vulnerabilities from its operating system. Your final image has almost none, as it contains only your trusted binary. |

**Important:** The first-build time for the optimized Dockerfile can sometimes be slightly longer because the builder stage has to download modules and compile the binary. However, the **second (and subsequent) builds are dramatically faster** with BuildKit cache mounts. The image size reductions are typically enormous when moving from a full golang final image to a scratch/distroless final image.

### 🧹 Cleanup (after comparing)

docker rm -f go-prod go-naive go-server 2>/dev/null || true

docker rmi go-app:prod go-app:naive go-app:semifinal go-app:prod-distroless go-app:prod-scratch go-app:builder go-app:step1 go-app:step2 aquasec/trivy 2>/dev/null || true

docker system prune

rm -f trivy\_goapp\_prod.txt trivy\_goapp\_naive.txt || true

find . -maxdepth 1 -type f -name 'Dockerfile\*' -not -name 'Dockerfile.naive' -not -name 'Dockerfile.prod' -delete



## 🐳 Task 2: Student exercise : Optimize the **Node** app Dockerfile

🎯 **Goal:** Starting from the provided node-exercise/Dockerfile.naive, write an optimized Dockerfile.prod that applies the same classes of optimizations we used for the Go worked example.

### 📂 Node app: what is provided

Work in the EC2 path:

/home/ubuntu/docker\_lab/Activity6/node-exercise

├── Dockerfile.naive # provided (unoptimized)

├── README.md # provided (how to run naive image)

├── app.js # tiny Express app (GET / and GET /health)

└── package.json



**About the Node app:** A minimal Express application exposing two endpoints:

* GET /health → returns ok (used for health checks)
* GET / → returns a simple greeting

The app has no build step (plain JavaScript) and uses only a few dependencies listed in package.json. Your task is to produce a production-ready Dockerfile for this app (named Dockerfile.prod) that is smaller, faster to iterate on, and safer to run.

### 🔎 Baseline (already available)

Dockerfile.naive:

# Dockerfile.naive

FROM node:18

WORKDIR /app

COPY . .

RUN npm install

EXPOSE 8080

CMD ["npm", "start"]



This builds and runs, but leaves many optimization opportunities.

### ✳️ Where you should focus: *conceptual hints*

Below are **hints** pointing to the areas you should optimize.

* **Dependency isolation & cacheability**
  + Think about which files change frequently and which do not. Arrange steps so dependency installation is cached when only source code changes.
* **Reproducible installs**
  + Consider the difference between a simple install and a reproducible install. Which artifact ensures the same install every time?
* **Build vs runtime separation**
  + Can you separate the installation/build of dependencies from the minimal runtime image so the final image contains only what it needs to run?
* **Final image minimality**
  + The runtime image should be as small as possible while still running Node. What base image spectrum exists (full → slim → musl-based → distroless), and what tradeoffs do they bring?
* **Cache reuse across builds**
  + How can you make repeated local edits rebuild very fast? (Think caching mechanisms that persist dependency caches across builds.)
* **Least privilege execution**
  + How will you ensure the container process does not run as root at runtime?
* **Health probing in a minimal runtime**
  + If you choose a minimal runtime that lacks shell tools, how will the health probe be implemented?
* **Layer ordering & file copies**
  + Which files should be copied early to maximize cache reuse, and which should be copied later to avoid invalidating dependency layers?
* **Production vs development dependencies**
  + How do you keep dev-only packages out of the final runtime image?
* **Package manager cache location**
  + Where does the package manager cache live (inside the build), and how could reusing it speed builds?

Use these hints to design your Dockerfile.prod. The point of this exercise is that you decide the tradeoffs and implement them.

### ⚙ Minimal Dockerfile.prod scaffold (create this file and fill it in)

Create the file /home/ubuntu/docker\_lab/Activity6/node-exercise/Dockerfile.prod on EC2. Keep the file mostly empty as a scaffold: include only an initial header and comments describing where to add stages. Example scaffold (paste into Dockerfile.prod and implement below it):

# syntax=docker/dockerfile:1.4

# Dockerfile.prod (student to implement)

# --- Add a builder stage here (if needed) ---

# --- Add a minimal runtime stage here ---

# --- Ensure HEALTHCHECK, non-root USER, and pinned runtime base are handled ---

# End of scaffold



**Important:** keep your final file named exactly Dockerfile.prod in the node-exercise folder.

### ✅ Verify the same checks you ran for the Go production image

After building your optimized image on EC2, perform **all** the verification steps we used for the Go image. This is the checklist you must complete **before** submission:

1. **Build succeeds**
   * DOCKER\_BUILDKIT=1 docker build -t node-app:prod -f Dockerfile.prod . must exit 0.
2. **Runtime health**
   * Start the naive and optimized containers on different host ports (e.g., 8080 and 8081) and confirm:
     + curl -fsS http://localhost:<port>/health returns ok for both naive and optimized images.
3. **Non-root**
   * Verify the optimized image does not run as root:
     + Inspect Config.User in the image metadata or check container UID at runtime (id -u inside container): it should not be 0.
4. **Image size & layering**
   * Compare docker images and docker history for node-app:naive vs node-app:prod. Optimized image should show fewer unnecessary layers and a reduced size.
5. **HEALTHCHECK present and passing**
   * The optimized image should include a HEALTHCHECK instruction that passes when the container is started.
6. **Security scan (recommended)**
   * Optionally run a vulnerability scan (e.g., trivy) on both images to observe differences; save outputs locally for your records.
7. **Both images present on EC2**
   * Ensure both the naive and optimized images are present in the EC2 Docker image list (docker images), because the submission/test will be performed while both images exist on the EC2 host.

### 📤 Submission instructions (what to do when ready)

1. **Ensure both images are present on EC2**
   * Build the naive image (if not present): docker build -t node-app:naive -f Dockerfile.naive .
   * Build your optimized image: DOCKER\_BUILDKIT=1 docker build -t node-app:prod -f Dockerfile.prod .
   * Confirm with docker images | grep node-app.
2. **Copy your optimized Dockerfile to your local clab workspace** (for submission):
   * Open your local clab workspace editor and paste the contents of your optimized Dockerfile.prod into the file:

/home/labDirectory/Activity6/node-exercise/Dockerfile.prod



* + Save the file locally.

1. **Submit for testing**

## 🧹 Cleanup After the Activity

1. Stop and remove all containers/images:

docker ps -aq | xargs docker rm -f  
docker images -aq | xargs docker rmi -f



1. Remove/Prune all Caches (optional):

docker buildx prune -f  
docker image prune -a -f  
docker system prune -a --volumes -f



1. 🚨 Remove all files from EC2 instance if needed:

rm -rf ~/docker\_lab/\*



## 🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

exit



1. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

## 🏆 Congratulations: You’ve completed Activity 6!

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

7️⃣ Activity 7: Advanced Docker Concepts (Bonus)

# Activity 7: Advanced Docker Concepts (Bonus)

🎯 **Objective:** Explore advanced Docker internals and ecosystem topics with deeper theory and hands-on practice. This activity is designed for students who want to understand how Docker really works under the hood. **No submission or grading**: this is a bonus lab for self-learning.

## ✅ Prerequisites

* Ubuntu 24.04 LTS EC2 instance with Docker installed and running.
* SSH access (with secret-key.pem and EC2 public IP).
* Comfort with Linux CLI and sudo.
* Curiosity to explore how Docker integrates with Linux kernel and OCI standards.

## 📂 Lab setup

SSH into your EC2 instance:

ssh -i secret-key.pem ubuntu@<EC2\_PUBLIC\_IP>



Switch to root for system-level inspection:

sudo -i



All commands below are to be run on the EC2 host.

# 📌 Docker Daemon & Engine API

### Theory

Docker has two key components:

1. **Docker daemon (dockerd)**: long-running background process managing containers, images, networks, and volumes.
2. **Docker CLI (docker)**: a client tool that talks to the daemon.
3. **Docker API**: REST API exposed by the daemon over a Unix socket (/var/run/docker.sock) or optionally a TCP socket.

The CLI (docker ps, docker run, etc.) is just a wrapper that makes HTTP requests to the Docker API. For example, docker ps calls GET /containers/json.

This separation means **you can build your own Docker clients** using HTTP calls.

👉 Docs: [Docker Engine API](https://docs.docker.com/engine/api/)

### Hands-on

Check daemon status:

ps aux | grep dockerd

systemctl status docker --no-pager



Inspect daemon logs:

journalctl -u docker -n 20



Curl the API directly:

# list running containers (like docker ps)

curl --unix-socket /var/run/docker.sock http://localhost/containers/json | jq



Create a container via API:

# Pull image  
curl -s --unix-socket /var/run/docker.sock \

-X POST \

"http://localhost/images/create?fromImage=busybox&tag=latest"

#Create container

curl -s -X POST --unix-socket /var/run/docker.sock \

-H "Content-Type: application/json" \

-d '{"Image":"busybox","Cmd":["echo","hello-from-API"]}' \

http://localhost/containers/create?name=api-test | jq

# Start container

curl -s --unix-socket /var/run/docker.sock \

-X POST \

http://localhost/containers/api-test/start



Check logs:

curl -s --unix-socket /var/run/docker.sock \

"http://localhost/containers/api-test/logs?stdout=1&stderr=1&timestamps=0" | jq -R -s .



# 📌 Container Internals: Namespaces, Cgroups, Capabilities

### Theory

Linux kernel features make containers possible:

* **Namespaces**: isolate resources:
  + pid (process IDs),
  + net (network interfaces),
  + mnt (mount points),
  + ipc, uts, user. Each container has its own “view” of processes, networking, and filesystems.
* **Control groups (cgroups)**: control and account resources (CPU, memory, IO, pids). Cgroups ensure a container cannot exceed resource quotas.
* **Capabilities**: fine-grained kernel privileges (instead of full root). E.g., CAP\_NET\_ADMIN, CAP\_SYS\_ADMIN. By default, containers drop some dangerous capabilities for safety.

👉 Docs:

* [Namespaces in Linux](https://man7.org/linux/man-pages/man7/namespaces.7.html)
* [Cgroups v2](https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html)
* [Linux capabilities](https://man7.org/linux/man-pages/man7/capabilities.7.html)

### Hands-on

Run a container and find its PID:

docker run -d --name ns-test busybox sleep 300

PID=$(docker inspect -f '{{.State.Pid}}' ns-test)

echo $PID



Check its namespaces:

ls -l /proc/$PID/ns



Check its cgroups:

sudo cat /proc/$PID/cgroup



Use lsns (if installed) to view namespace assignments:

sudo lsns -p $PID



Try a container with dropped capabilities:

docker run -it --rm --cap-drop ALL busybox sh -c "id; ps aux"

# Observe difference in capability bitmask  
docker run --rm busybox sh -c 'echo "CapEff (no drop):"; awk "/CapEff/ {print \$2}" /proc/1/status'

docker run --rm --cap-drop ALL busybox sh -c 'echo "CapEff (cap-drop):"; awk "/CapEff/ {print \$2}" /proc/1/status'



Clean up:

docker rm -f ns-test



# 📌 Storage Drivers & Image Layers

### Theory

Docker images and containers are built on **union file systems** using storage drivers.

* Default driver: **overlay2** (on Ubuntu ≥ 18.04).
* Each image consists of **read-only layers**.
* A container adds a **thin writable layer** on top.

This layering is what allows caching, efficient pulls, and small incremental changes.

👉 Docs: [Docker storage drivers](https://docs.docker.com/storage/storagedriver/)

### Hands-on

Check driver in use:

docker info | grep "Storage Driver"



See disk usage:

docker system df



Inspect /var/lib/docker/overlay2 (do not modify):

sudo ls /var/lib/docker/overlay2 | head



View image history:

docker history alpine:3.18



Export a container’s filesystem:

docker export $(docker create alpine) | tar -tvf - | head



# 📌 Rootless Docker

### Theory

Rootless Docker runs the Docker daemon and containers entirely without root privileges.

**Why important?**

* Extra security (mitigates daemon privilege escalation).
* Useful in shared systems without root access.

**How it works:**

* Uses **user namespaces** to map container root → host non-root UID.
* Requires slirp4netns for networking and fuse-overlayfs for storage.

👉 Docs: [Rootless mode](https://docs.docker.com/engine/security/rootless/)

### Hands-on (conceptual only)

Check if your current daemon is rootless:

docker info | grep Rootless



To install rootless Docker (don’t run in production without care):

curl -fsSL https://get.docker.com/rootless | sh



Then run the daemon as your user:

systemctl --user start docker

export DOCKER\_HOST=unix:///run/user/$UID/docker.sock



(We won’t do full setup here; just awareness.)

# 📌 Docker Init (Scaffolding a Project)

### Theory

Since **Docker 25.0**, a new command docker init was introduced to quickly scaffold a **starter Dockerfile** and .dockerignore for your project.

It analyzes your project’s source code (Node.js, Python, Go, .NET, etc.) and generates a basic containerization setup.

👉 Docs: [docker init reference](https://docs.docker.com/reference/cli/docker/init/)

**Why useful?**

* Eases onboarding for beginners.
* Provides best-practice base images for common languages.
* Saves time writing boilerplate Dockerfiles.

**Cautions:**

* The generated Dockerfile is **generic**, not optimized (e.g., no multi-stage builds, no cache tuning).
* It may include unnecessary layers or packages.
* Treat it as a **starting point only**, not production-ready.
* Always review, optimize, and add security hardening before using in production.

### Hands-on

1. Create a sample Node project:

mkdir ~/demo-node && cd ~/demo-node

npm init -y

echo "console.log('Hello Docker Init!')" > app.js



1. Run Docker init:

docker init



It will prompt you with questions like:

* Which language? (Node, Python, Go, etc.)
* Which package manager?
* Default port?

It then generates:

* Dockerfile
* .dockerignore

1. Inspect the generated Dockerfile:

cat Dockerfile



1. Build & run the scaffolded image:

docker build -t demo-node .

docker run --rm demo-node



### Example Output (Node.js)

# Generated by docker init

FROM node:18

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]



### Why it matters

* For teaching/demo projects, docker init lowers the barrier.
* For production, it’s a **teaching tool**: you should gradually evolve the Dockerfile to include **multi-stage builds, non-root users, healthchecks, and slim base images** (as we did in Activity 6).

# 📌 OCI Standards: Docker vs Containerd vs CRI-O

### Theory

The **Open Container Initiative (OCI)** standardizes:

1. **Image format** (OCI Image Spec).
2. **Runtime format** (OCI Runtime Spec).

This lets multiple runtimes and tools interoperate.

* **Docker** (CLI + daemon + build + containerd + runc).
* **containerd**: lightweight core container runtime (used inside Docker and Kubernetes).
* **CRI-O**: Kubernetes-focused runtime, minimal.
* **runc**: reference implementation of the OCI runtime (low-level, executes containers).

👉 Docs:

* [OCI Image Spec](https://github.com/opencontainers/image-spec)
* [OCI Runtime Spec](https://github.com/opencontainers/runtime-spec)
* [Containerd](https://containerd.io/)
* [CRI-O](https://cri-o.io/)

### Hands-on

Check if containerd is running:

systemctl status containerd



Try ctr (containerd CLI):

sudo ctr images pull docker.io/library/alpine:3.18

sudo ctr run --rm -t docker.io/library/alpine:3.18 testctr sh -c "echo hello from containerd"



# 📌 Kernel Interfaces: /proc, /sys/fs/cgroup, unshare, runc

### Theory

Containers are just Linux processes with namespaces and cgroups.

* /proc shows process details and namespace links.
* /sys/fs/cgroup shows cgroup controllers and usage.
* unshare creates a new namespace manually.
* runc is the low-level binary that executes containers from OCI bundles (used under Docker).

👉 Docs:

* [runc OCI runtime](https://github.com/opencontainers/runc)
* [man 2 unshare](https://man7.org/linux/man-pages/man2/unshare.2.html)

### Hands-on

Inspect a container’s namespaces:

docker run -d --name alpine1 alpine sleep 60

PID=$(docker inspect -f '{{.State.Pid}}' alpine1)

ls -l /proc/$PID/ns



Inspect cgroups:

cat /proc/$PID/cgroup



Try unshare:

sudo unshare --fork --pid --mount-proc bash

ps aux # shows only processes inside new PID namespace

exit



Try runc (low-level run):

mkdir -p /tmp/bundle/rootfs

cd /tmp/bundle

docker export $(docker create alpine) | tar -C rootfs -xf -

runc spec

sudo runc run testcontainer



Clean up:

docker rm -f alpine1



# 🧹 Cleanup

docker ps -aq | xargs docker rm -f

docker system prune -af



# 📖 Suggested further reading

* [Docker Engine overview](https://docs.docker.com/get-started/overview/)
* [Namespaces and cgroups deep dive (LWN)](https://lwn.net/Articles/531114/)
* [Rootless Docker guide](https://docs.docker.com/engine/security/rootless/)
* [OCI Specifications](https://opencontainers.org/)

## 🏆 Conclusion

This bonus activity gives you a “peek under the hood” of Docker: daemon & API, kernel integration, storage drivers, rootless mode, OCI ecosystem, and low-level runtimes.

The goal is not to memorize commands but to **connect the dots**: Docker is just tooling on top of Linux features (namespaces, cgroups, capabilities) and OCI standards.