

Activity 3: Building and Publishing Custom Docker Images

Objective

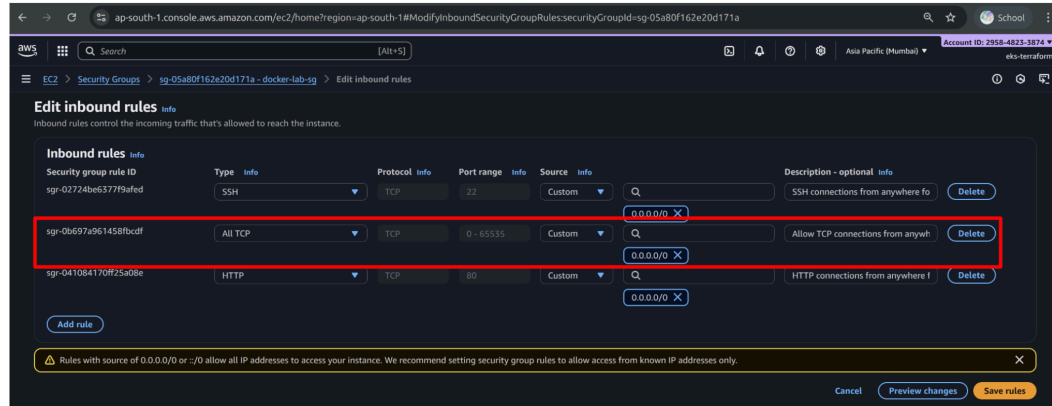
- Learn how to write **Dockerfile** to containerize an application.
- Understand a real **Flask** API's basic endpoints so you know what you're containerizing.
- Understand each instruction/flag and why we use it.
- Build, run and test **image** locally and then push to **Docker Hub** (public Docker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

```
Shell
bash system-clean-init.sh
```

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

```
Shell
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

```
None
$ 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:

Shell

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

2. Create and activate a virtual environment:

Shell

```
python3 -m venv venv
source venv/bin/activate
```

3. Install pip and dependencies:

Shell

```
sudo apt-get install python3-pip
pip install -r requirements.txt
```

4. Run the application:

Shell

```
python3 src/app.py
```

5. 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:

```
Shell
deactivate
rm -rf venv
```

- Purge extra packages if desired:

```
Shell
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/`)

```
None
__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:

```
None
FROM ubuntu:24.04
SHELL ["/bin/bash", "-c"]
```

3. Build the image:

```
Shell
docker build -t lab3/step1-base-shell .
```

4. Verify by running a quick command:

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

Student Action (ans.json)

```
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:

None

```
ENV APP_HOME=/app
ENV PORT=5000
WORKDIR ${APP_HOME}
```

3. Build the image:

Shell

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

4. Verify the working directory:

Shell

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

Student Action (ans.json)

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:

None

```
RUN apt-get update
RUN apt-get install -y --no-install-recommends python3 python3-pip
RUN apt-get install -y --no-install-recommends curl
```

3. Build the image:

Shell

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

4. Verify the installations:

Shell

```
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)

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:

```
None
COPY . .
ADD https://upload.wikimedia.org/wikipedia/sa/d/da/IIT\_Bombay\_logo.png
src/iit-bombay-logo.png
```

3. Build the image:

```
Shell
docker build -t lab3/step4-app-code .
```

4. Verify that the files exist inside the container:

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

Student Action (ans.json)

```
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 sequence for 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:

None

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

3. Build the image:

Shell

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

4. Verify Flask is installed:

Shell

```
docker run --rm lab3/step5-install-deps python3 -c "import flask;
print(flask.__version__)"
```

Student Action (ans.json)

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:

```
Shell
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:

None

```
EXPOSE ${PORT}
```

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

```
RUN chown -R appuser:nogroup ${APP_HOME}
```

```
USER appuser
```

3. Build the image:

Shell

```
docker build -t lab3/step6-nonroot .
```

4. Verify user and permissions:

Shell

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

Student Action (ans.json)

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](#), 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):

```
None
LABEL maintainer="<Your Name> <your email>"
LABEL version="1.0"
LABEL description="A simple Product Inventory API."
```

3. Build the image:

Shell

```
docker build -t lab3/step7-labels .
```

4. Verify labels in the built image:

Shell

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

Student Action (ans.json)

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.

```
Shell
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:

```
Shell
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:

```
None
HEALTHCHECK --interval=30s --timeout=5s --start-period=5s \
    CMD curl -f http://localhost:${PORT}/health || exit 1

ENTRYPOINT ["python3"]
CMD ["src/app.py"]
```

3. Build the image:

```
Shell
docker build -t lab3/step8-runtime .
```

4. Run the container:

```
Shell
docker run --rm -p 5000:5000 lab3/step8-runtime
```

5. In another EC2 ssh terminal, test endpoints:

```
Shell
curl -s http://localhost:5000/api/products | head -n 5
curl -s -o /dev/null -w "%{http_code}\\n" http://localhost:5000/health
```

6. 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??`

7. Inspect health status:

Shell

```
CONTAINER_ID=$(docker ps -q --filter ancestor=lab3/step8-runtime)
docker inspect --format='{{json .State.Health}}' $CONTAINER_ID | jq
```

Student Action (ans.json)

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.

None

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"]

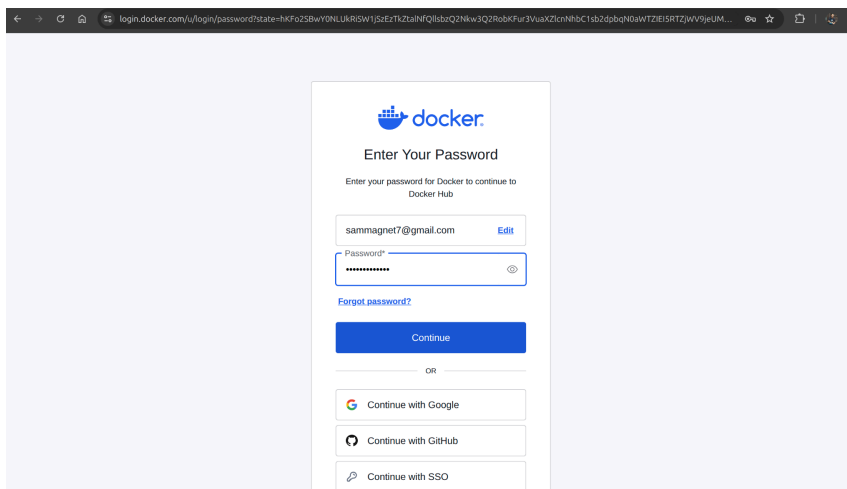
# CMD provides default arguments.
CMD ["src/app.py"]

```

Logging into Docker Hub and Creating a Repository

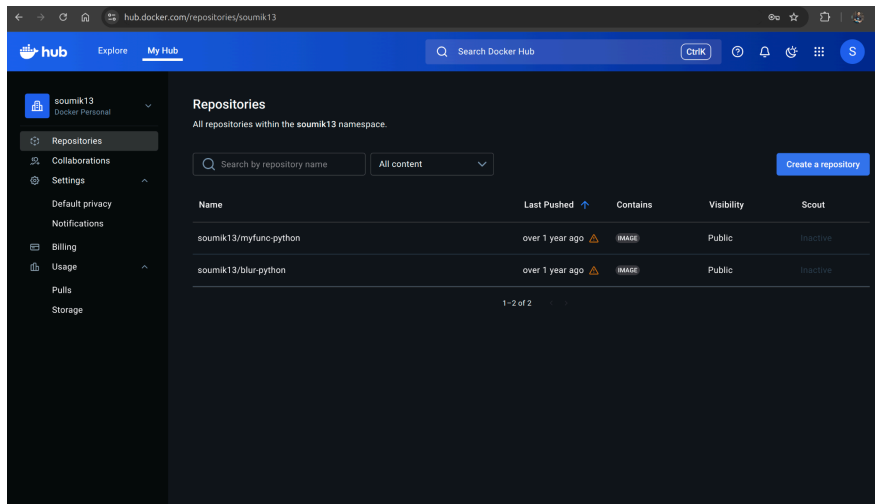
Before pushing your image, you need a **public repository** on Docker Hub. Follow the steps below:

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



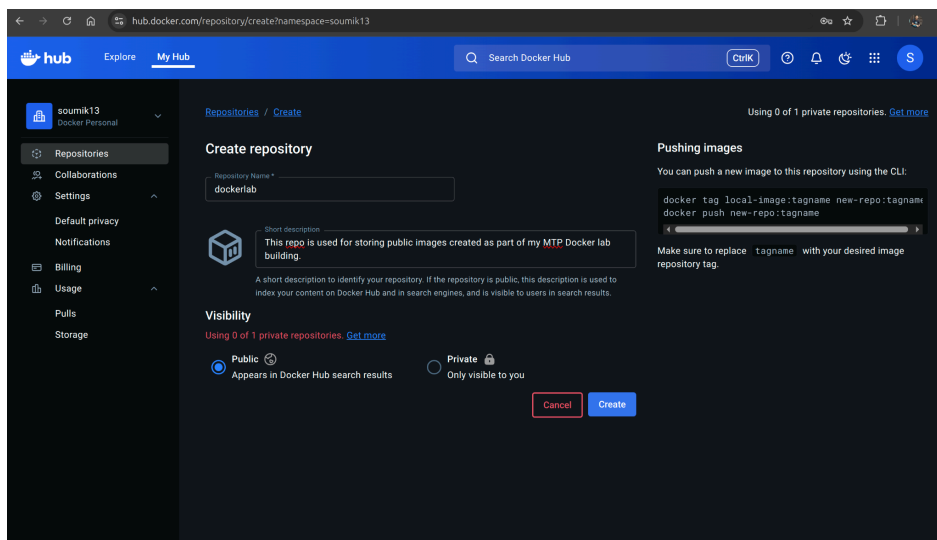
The screenshot shows the Docker Hub login page in a web browser. The page has a light blue header with the Docker logo and the text "Enter Your Password". Below this, it says "Enter your password for Docker to continue to Docker Hub". There is a text input field for the email address, which contains "sammagnet7@gmail.com", and a blue "Edit" link next to it. Below the email field is a password input field with a blue border and a blue "Show/Hide" icon on the right. Below the password field is a blue link that says "Forgot password?". Below the password field is a blue "Continue" button. Below the "Continue" button is a horizontal line with the word "OR" in the center. Below the line are three buttons: "Continue with Google" (with the Google logo), "Continue with GitHub" (with the GitHub logo), and "Continue with SSO" (with a key icon).

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

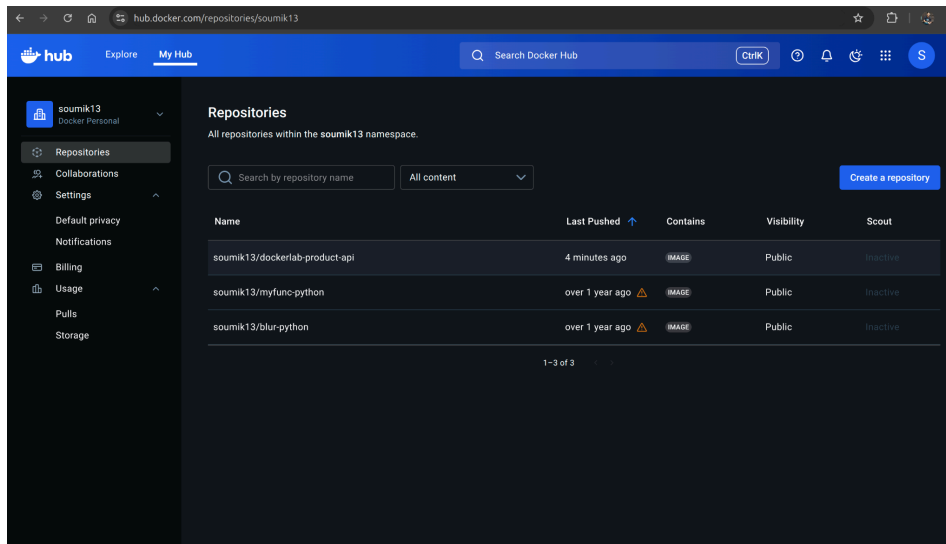


3. **Create a New Repository**

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



4. **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:

```
Shell
docker build -t lab3/final .
```

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

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

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

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

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

```
Shell
curl -s http://localhost:5000/api/products | head -n 5
curl -s -o /dev/null -w "%{http_code}\n" http://localhost:5000/health
```

5. **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)

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

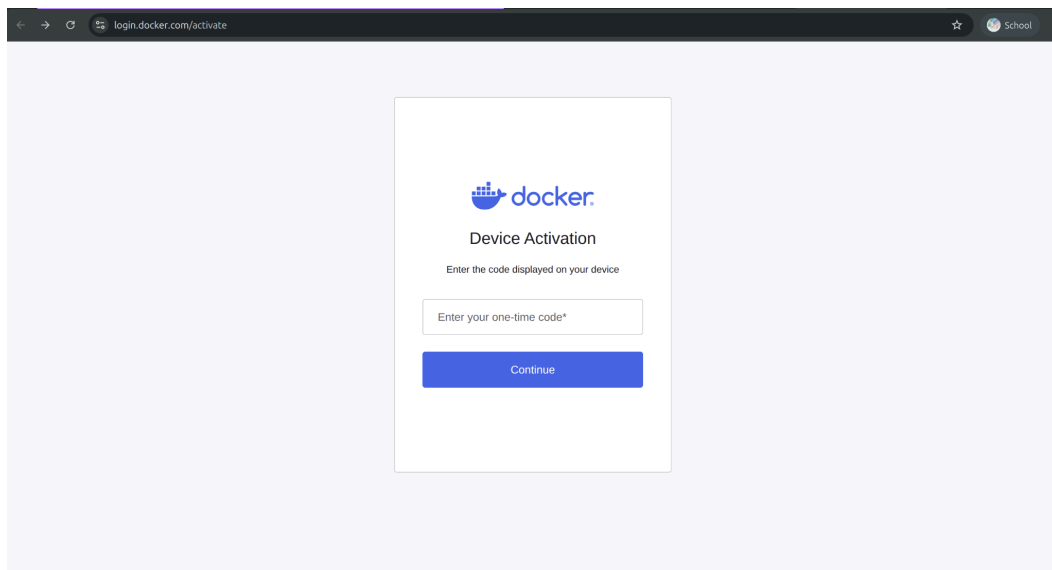
- **docker login** → Authenticates your local Docker CLI with Docker Hub. You will be prompted for your username and password (or access token).
- **docker push** → Uploads the tagged image layers to Docker Hub.
- **Digest** → Docker assigns a unique immutable identifier (**sha256:<hash>**) to the pushed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

```
Shell  
docker login
```

```
ubuntu@ip-172-31-13-192:~/docker_lab/product-api-flask$ docker login  
  
USING WEB-BASED LOGIN  
  
Info → To sign in with credentials on the command line, use 'docker login -u <username>'  
  
Your one-time device confirmation code is: HFLJ-LHVM  
Press ENTER to open your browser or submit your device code here: https://login.docker.com/activate  
Waiting for authentication in the browser...  
█
```



2. Push the image to your repository:

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

3. Verify the push:

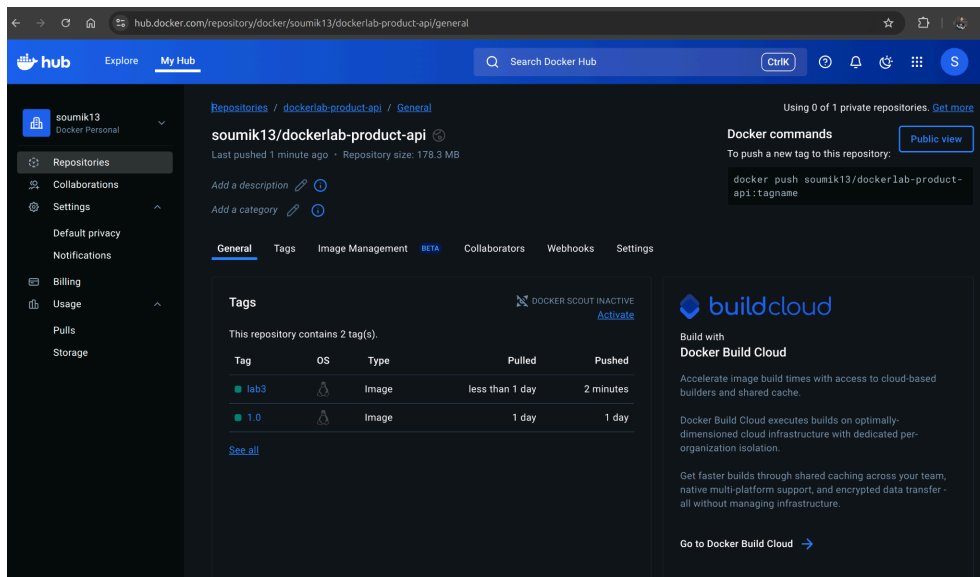
Shell

```
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...**).

4. Check on Docker Hub

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



The screenshot shows the Docker Hub interface for the repository `soumik13/dockerlab-product-api`. The left sidebar contains navigation links for Repositories, Collaborations, Settings, Default privacy, Notifications, Billing, Usage, Pulls, and Storage. The main content area displays the repository details, including the tag `lab3` and `1.0`. The `lab3` tag is highlighted in green. The right sidebar shows Docker commands and a section for Docker Build Cloud.

Tag	OS	Type	Pulled	Pushed
lab3	linux	Image	less than 1 day	2 minutes
1.0	linux	Image	1 day	1 day

Student Action (ans.json)

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-entriypoint`
 - `lab3-cmd`

- logo-spotted
- lab3-dockerhub-repo
- lab3-tag
- lab3-digest



Cleanup After the Activity

1. Stop and remove all containers/images:

Shell

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

2. Remove/Prune all Caches (optional):

Shell

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

3.  Remove all files from EC2 instance if needed:

Shell

```
rm -rf ~/docker_lab/*
```



Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:


```
Shell  
exit
```

2. 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 <code>docker build .</code> (controlled by <code>.dockerignore</code>).
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).

Term / Instruction	Meaning & Usage
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 (<code>0</code>) = healthy; failure (<code>!=0</code>) = unhealthy.
Digest	Content-addressable ID (<code>algo:hex</code>) 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 <code>:lab3</code> or <code>:latest</code>) pointing to an image ID.
Repository	A named collection of related images (e.g., <code>username/product-api</code>).
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 (<code>dockerd</code>) that builds, runs, and manages containers.

Term / Instruction	Meaning & Usage
Docker CLI	Command line interface (docker) that interacts with the Docker daemon.



Congratulations – You’ve completed Activity 3!

===== END OF DOCUMENT =====