End-to-End Practical Guide — Containerize a Python App & Push to Docker Hub

---

Project layout (create one folder per student / group)

```
docker-python-lab/
├─ app/
│  ├─ app.py
│  ├─ requirements.txt
│  └─ .dockerignore
├─ Dockerfile
└─ docker-compose.yml
```

Put `app/` as a subfolder containing python sources. `Dockerfile` and `docker-compose.yml` sit at repo root.

Step 0 — Prerequisites (verify before class)

- Docker installed and running (Docker Desktop or Docker Engine on Linux). `docker --version` and `docker info` should succeed.
- Students have a Docker Hub account (username).
- Terminal access and a text editor.

Step 1 — Quick smoke tests: pull & run official images

1. Hello World:

```
docker run hello-world
```

- Verifies daemon, network, pull permissions.

2. Run nginx (port mapped):

```
docker run -d --name demo-nginx -p 8080:80 nginx
# Verify
docker ps
# Open http://localhost:8080 in browser
```

3. Stop & remove:

```
docker stop demo-nginx
docker rm demo-nginx
```

Step 2 — Images, Volumes, Containers (short hands-on)

1. Create a volume:

```
docker volume create demo-vol
docker volume ls
```

2. Run nginx with a named volume mounted at `/usr/share/nginx/html` (example):

   ```
   docker run -d --name nginx-vol -p 8081:80 -v demo-
   vol:/usr/share/nginx/html nginx
   ```

4. Testing the volume

   ```
   docker exec -it nginx-vol bash
   ```

   ```
   cd /usr/share/nginx/html
   ```

   `ls` <- index.html will be present

   http://localhost:8081/ <- nginx will be running

   exit

Step 3 — Create application code (app/app.py)

Create `docker-python-lab/app/app.py` with this simple Flask app:

```
# app/app.py
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def hello():
    return jsonify(message="Hello from containerized Flask app!")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

`host="0.0.0.0"` is mandatory so the container exposes the port.

Step 4 — requirements.txt

Create `app/requirements.txt`:

```
flask==2.2.5
```

(Use a pinned version to be deterministic in class.)

Step 5 — .dockerignore

Create `app/.dockerignore` (keeps build context small):

```
__pycache__/
*.pyc
*.pyo
*.pyd
*.log
.env
.git
venv/
.idea/
```

Step 6 — Dockerfile (root)

Create `docker-python-lab/Dockerfile`:

```
# Use slim Python base
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Copy requirements first (for caching)
COPY app/requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY app/ ./

# Expose port
EXPOSE 5000

# Run the app
```

```
CMD ["python", "app.py"]
```

Why this order? Copying `requirements.txt` first lets Docker cache pip install layer — faster rebuilds.

Step 7 — Build & Run the image locally (docker build + run)

1. From project root:

```
# Build image (local tag)
docker build -t studentname/flask-lab:v1 .
```

2. Run container mapping port:

```
docker run -d --name flask-lab -p 5000:5000 studentname/flask-lab:v1
```

3. Verify:

```
docker ps
curl http://localhost:5000/    # should return JSON
docker logs flask-lab
```

4. Stop & remove:

```
docker stop flask-lab && docker rm flask-lab
```

Step 8 — docker-compose.yml (single service)

Create `docker-python-lab/docker-compose.yml`:

```
version: "3.8"

services:
  web:
    build: .
    image: studentname/flask-lab:compose-v1
    ports:
      - "5000:5000"
    volumes:
      - ./app:/app:ro        # bind code read-only for local dev (optional)
    restart: unless-stopped
```

Notes:

- `build: .` uses the Dockerfile in root.
- `volumes` maps local code into container for quick edits (use `:ro` to prevent accidental writes).
- `image` sets the image name so `docker-compose push` can be used (if logged in).

Step 9 — Build & run with Docker Compose

From project root:

```
# Build (compose will use Dockerfile)
docker-compose build

# Start in detached mode
docker-compose up -d

# Check containers
docker-compose ps
curl http://localhost:5000/

# Stop
docker-compose down
```

`docker-compose logs -f` shows combined logs.

Step 10 — Finalize local verification

- Confirm endpoint returns JSON.
- Show `docker images` and explain IMAGE ID, TAG, SIZE.
- Show `docker inspect <container>` briefly to point students to configuration details.

Step 11 — Tag & Push to Docker Hub (publish)

1. Login to Docker Hub (interactive):

```
docker login
# Enter username (studentname) and password
```

2. Tag the local image (if not already named with your Docker Hub username):

```
docker tag studentname/flask-lab:compose-v1 studentname/flask-lab:latest
```

3. Push:

```
docker push studentname/flask-lab:latest
```

4. Verify (students can pull on another machine or cloud VM):

```
docker pull studentname/flask-lab:latest
docker run -d -p 5000:5000 studentname/flask-lab:latest
curl http://localhost:5000/
```

Step 12 — Cleanup commands (classroom reset)

```
# Stop and remove all containers created by compose
docker-compose down --rmi local --volumes

# Remove image
docker image rm studentname/flask-lab:latest

# Remove dangling images
docker image prune -f

# Remove stopped containers
docker container prune -f

# Remove unused volumes
docker volume prune -f
```

(Show caution: `prune` removes unused resources.)

Step 13 — How files stitch together (explain to students, succinct)

- `app/` contains code and dependency list.
- `.dockerignore` reduces build context, speeds builds.
- `Dockerfile` defines image build steps: base → deps → copy code → CMD.
- `docker-compose.yml` defines how to run multi-service stacks; it can build or use image, map ports and volumes.
- `docker run` creates containers from images; `docker push` uploads images to a registry (Docker Hub) so others can `docker pull`.