Lab 2 Submission — Docker Containerization

This application is fully containerized for easy and consistent deployment.

Build the Image Locally

To build the Docker image from the source code and the provided Dockerfile, use the docker build command with appropriate tags.

Run a Container

To run the application inside a container from the built image, use the docker run command, ensuring you map the container's exposed port to a port on your host machine.

Pull from Docker Hub

The pre-built image is also available on Docker Hub. You can pull it directly using docker pull with the correct repository name and tag, and then run it as described above.

Docker Best Practices Applied

- Using a non-root user (USER appuser):
    - Why it matters: Running containers as root is a significant security risk. If an attacker compromises the application, they gain root privileges in the container, which can be leveraged for further attacks (container escape, host system compromise). Creating and switching to a dedicated, unprivileged user (appuser) minimizes the potential impact of a security breach.

```
RUN useradd --create-home --shell /usr/sbin/nologin appuser
USER appuser
```

- Layer Caching & Ordering:
    - Why it matters: Docker caches the result of each layer. By copying only requirements.txt first and installing dependencies before copying the entire application code (app.py), we optimize the build cache. Changes to app.py will not trigger a re-install of all Python packages, leading to much faster rebuilds.

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .
```

- Using .dockerignore:

    - Why it matters: The .dockerignore file prevents unnecessary files (like local virtual environments venv/, IDE config .vscode/, cache **pycache**/, or secrets) from being sent to the Docker daemon during COPY. This results in a smaller build context, faster builds, and a more secure image by avoiding accidental inclusion of sensitive data.

- --no-cache-dir with pip:
    - Why it matters: This flag tells pip not to store the download cache. Since the installed packages are persisted in the Docker image layer, keeping the cache is redundant and only increases the final image size.

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

Image Information & Decisions

- Base Image: python:3.13-slim

Justification: The slim variant provides a balance. It contains the essential Python runtime and common system libraries needed for most Python apps but strips out unnecessary extra packages (like common documentation) found in the default python:3.13 image. This leads to a significantly smaller and more secure image compared to the full alpine version, which might require additional steps to compile some Python dependencies.

- Final Image Size: 49 MB

Assessment: This is a reasonable size for a Python application. The bulk comes from the python:slim base layer. Further reduction could be explored using multi-stage builds if the project had complex compilation steps, but for this simple Flask app, slim offers the best trade-off between size and ease of use.

- Layer Structure & Optimization:

The Dockerfile is structured to leverage caching. The most stable dependencies (Python package list from requirements.txt) are copied and installed first. The more frequently changed application code (app.py) is copied in the final COPY instruction. This ensures that code changes don't invalidate the cached pip install layer, speeding up development cycles.

Build & Run Process

1. Build command:

```
docker build -t lab02:latest .
```

2. Command for container running:

```
docker run --rm -it -p 5001:5000 lab02
```

3. Docker Hub Repository URL :

```
https://hub.docker.com/repository/docker/abrahambarrett228/lab02/general
```

P.S. Terminal output showing the successful push:

```
abraham_barrett@Abrahams-MacBook-Air app_python % docker push
abrahambarrett228/lab02:latest
The push refers to repository [docker.io/abrahambarrett228/lab02]
fe9a90620d58: Pushed
a6866fe8c3d2: Pushed
97fc85b49690: Pushed
ee8758c3eb7d: Pushed
4fa8698484f1: Pushed
3ea009573b47: Pushed
6fb77c4bfd96: Pushed
399cf5dc7b8b: Pushed
3de5fa5034b2: Pushed
latest: digest:
sha256:ff4a7b2b082f8fa68caa395f865e938ed30671d377439092b6ecefe3b2873007
size: 856
```

However, there were no difficult strategy of tagging images since for now I need to create only one push. However, in future, I may recreate a strategy with creating versioning (based on major/minor updates)

Technical Analysis

- Dockerfile Logic: The file follows a logical flow: define the environment (FROM), set up the workspace (WORKDIR), install system-level dependencies if any (none here), install Python dependencies, set up security (non-root user), copy application code, declare runtime configuration (EXPOSE, CMD). This order maximizes layer cache efficiency and security.
- Changing Layer Order: If we copied app.py before running pip install, every single change to the application code would force Docker to invalidate the cache for the pip install layer and all subsequent layers. This would result in re-downloading and re-installing all dependencies on every build, making the development process much slower.
- Security Considerations: The key security measure is running the process as a non-root user (appuser). Additionally, using an official, minimal base image (slim) reduces the attack surface by including fewer pre-installed packages that could contain vulnerabilities. The .dockerignore file is also a security feature, preventing local secrets from being baked into the image.

- .dockerignore Improvement: Beyond security, .dockerignore drastically speeds up the build process, especially for projects with large directories like node_modules/ or .git/. The Docker daemon doesn't have to process these files, leading to quicker context upload and layer creation.