TU856/3
INTRODUCTION TO DEVOPS

LAB 6 - DOCKER AND CONTAINERS

12/03/2025

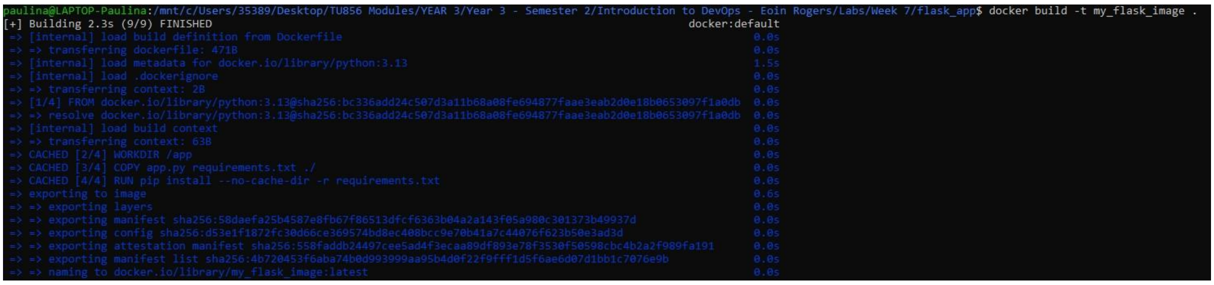
PAULINA CZARNOTA C21365726

1. BUILDING THE DOCKER IMAGE

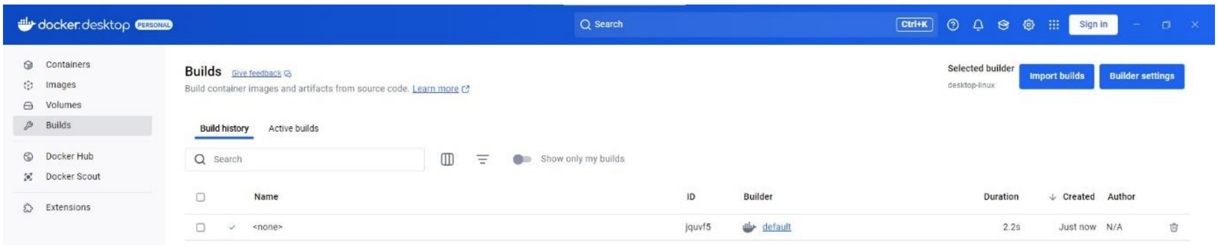
This screenshot shows the execution of:

- `docker build -t my_flask_image .`

The image build process is displayed, including successful completion.



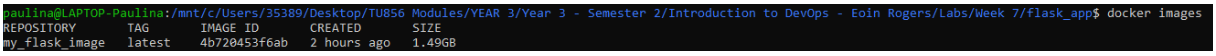
Docker Desktop displays the successfully built image in the `Build history`.



2. VERIFYING THE DOCKER IMAGE EXISTS

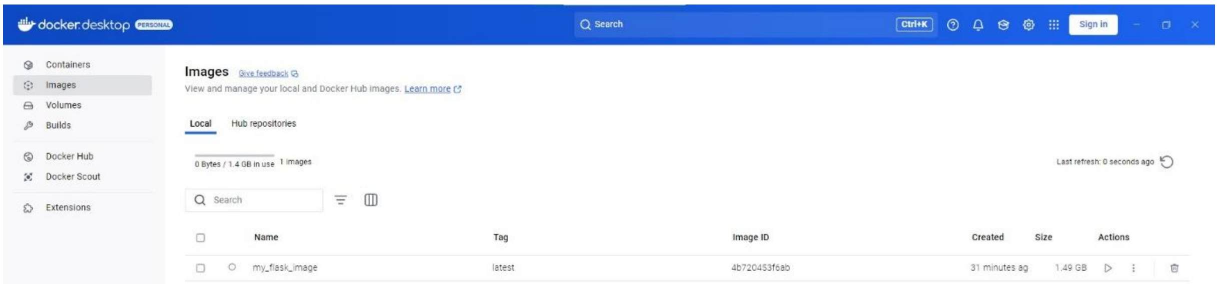
This screenshot captures the execution of:

- `docker images`



It lists all available Docker images, including `my_flask_image`.

A corresponding screenshot from Docker Desktop confirms the presence of `my_flask_image` under `Images`.



3. RUNNING THE FLASK CONTAINER ON PORT 8080

This screenshot displays the execution of:

- `docker run -p 8080:5000 --name my_flask_container my_flask_image`

```
paulina@LAPTOP-Paulina:/mnt/c/Users/35389/Desktop/TU856 Modules/YEAR 3/Year 3 - Semester 2/Introduction to DevOps - Eoin Rogers/Labs/Week 7/flask_app$ docker run -p 8080:5000 --name my_flask_container my_flask_image
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.0.2:5000
Press CTRL+C to quit
```

The terminal output confirms that the container `my_flask_container` has started and is running.

4. VERIFYING RUNNING CONTAINERS

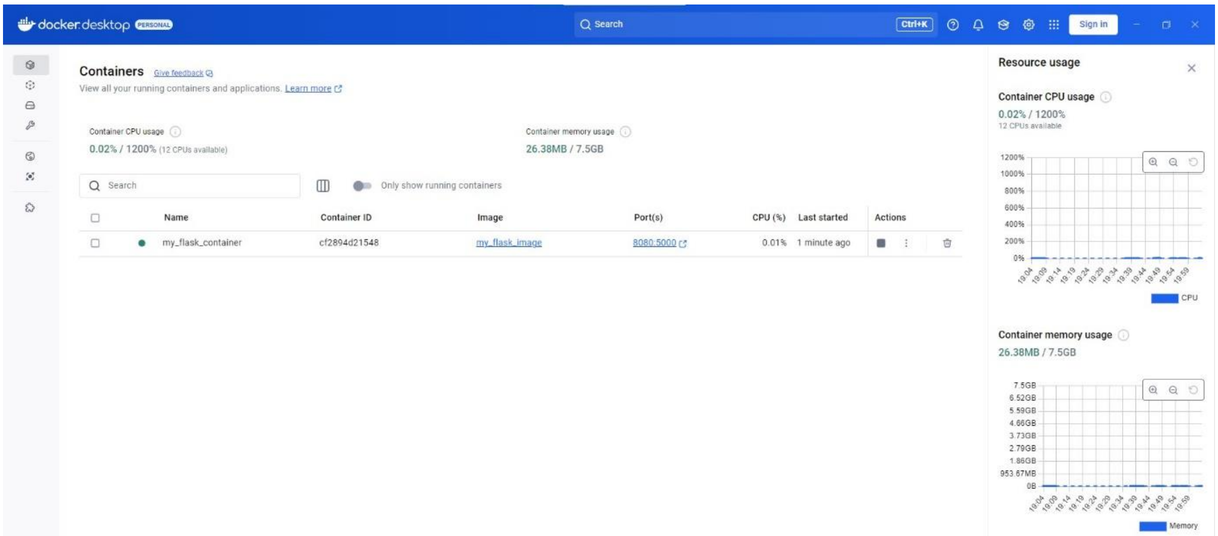
This screenshot captures the execution of:

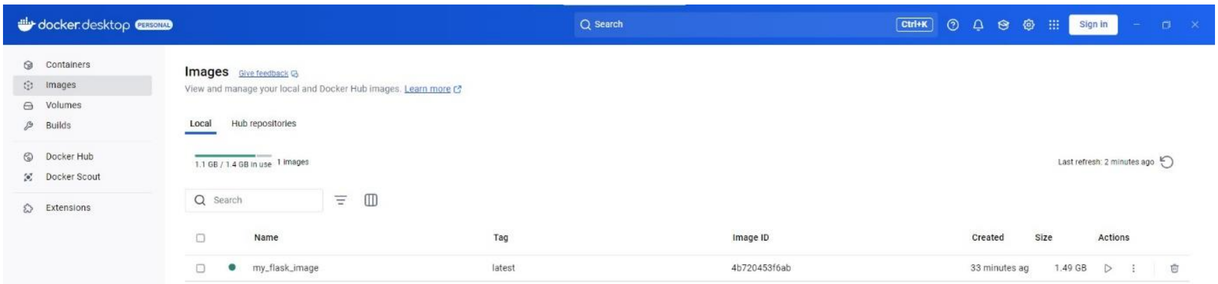
- `docker ps`

```
paulina@LAPTOP-Paulina:/mnt/c/Users/35389/Desktop/TU856 Modules/YEAR 3/Year 3 - Semester 2/Introduction to DevOps - Eoin Rogers/Labs/Week 7/flask_app$ docker ps
CONTAINER ID   IMAGE            COMMAND           CREATED              STATUS              PORTS                    NAMES
cf2894d21548   my_flask_image   "python app.py"   About an hour ago    Up About an hour    0.0.0.0:8080->5000/tcp   my_flask_container
```

It lists active containers, showing that `my_flask_container` is running.

The following screenshots from Docker Desktop confirms that the container and its image are active.

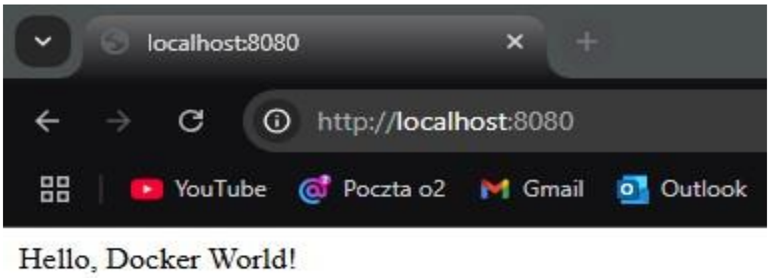




5. TESTING THE FLASK APPLICATION

The screenshot of the browser shows the output "Hello, Docker World!", confirming that the Flask app is successfully running inside the container.

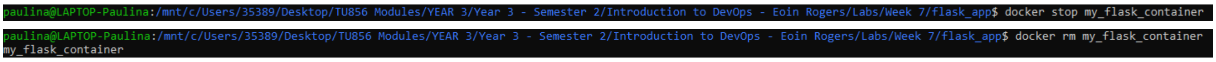
To access the application, the browser was directed to http://localhost:8080.



6. STOPPING AND REMOVING CONTAINERS (CLEANUP)

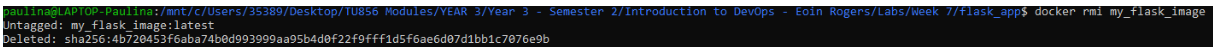
After testing, the container was stopped and removed using:

- `docker stop my_flask_container`
- `docker rm my_flask_container`



The removal of the image `my_flask_image` was confirmed using:

- `docker rmi my_flask_image`



7. INSPECTING AN EXISTING CONTAINER

The Dockerfile for this container:

https://github.com/docker-library/python/blob/37a6827e0b7a9ef099cfdec5de305e3d4cea7331/3.13/bookworm/Dockerfile

Q1: The image is itself derived from `buildpack-deps:bookworm`. What is this derived from? Can you trace and list all of the layers used in the container?

The `buildpack-deps:bookworm` image is derived from `debian:bookworm`, which serves as the base operating system image. This image is built on top of the Debian Linux distribution, a widely used, stable Linux-based OS.

`buildpack-deps:bookworm` includes additional development tools, such as:

- **Git** – A version control system for tracking code changes.
- **curl** – A tool for transferring data from URLs.
- **build-essential** – A package that provides key compilation utilities, including `gcc` and `make`.

These tools make `buildpack-deps:bookworm` suitable for building applications that require compiling dependencies.

Layers in the container:

1. **Debian Base Layer** – Contains the core Debian Bookworm operating system, providing a stable foundation.
2. **Buildpack-Dependencies Layer** – Installs essential development tools (`git, curl, build-essential`) to support software compilation and installation.
3. **Python Source Layer** – Downloads and extracts the Python source code from the official Python repository.
4. **Compilation Layer** – Builds Python from source using `make`, ensuring customization and performance optimizations.
5. **Installation Layer** – Installs the compiled Python binary and necessary components into the system.
6. **Cleanup Layer** – Removes unnecessary files, temporary build directories, and test folders to optimize image size.
7. **Final Configuration Layer** – Sets environment variables, creates symbolic links, and configures the runtime environment for proper execution.

Each of these layers contributes to building a functional, efficient, and optimized Python runtime within the container.

Q2: Given that the container image is based on Debian, why do you think the Dockerfile is downloading the source code on line 28 of the Dockerfile? Why not just install the Debian package with `apt`?

Line 28:

```
wget -O python.tar.xz
"https://www.python.org/ftp/python/${PYTHON_VERSION%%[a-z]*}/Python-
$PYTHON_VERSION.tar.xz"; \
```

Reasons for downloading and compiling from source instead of using `apt`:

1. **Latest Version Availability** – The Debian `apt` package manager provides a stable but often outdated version of Python. By downloading the source code, the container can use the latest available version.
2. **Customization and Optimization** – Compiling Python from source allows for build optimizations such as:
    - `--enable-optimizations` (enables better performance)
    - `--with-lto` (uses Link Time Optimization to reduce binary size and improve speed)
3. **Ensuring Compatibility** – Some features or bug fixes may not be available in the pre-packaged Debian version. By building from source, developers ensure that all required features and patches are included.
4. **Dependency Control** – The pre-built Debian Python package may omit certain libraries or compile options needed by developers. Building from source allows for greater control over dependencies and configurations.

Q3: Docker will build the Python interpreter when it builds the image. Python is built using `make`, which we discussed two weeks ago. Can you see the `make` invocation(s) in the Dockerfile? What line(s) does it appear on?

The `make` command is used to compile Python from source.

The relevant make invocations in the official Python Dockerfile occur at:

Lines 54-56:

```
make -j "$nproc" \
    "EXTRA_CFLAGS=${EXTRA_CFLAGS:-}" \
    "LDFLAGS=${LDFLAGS:-}" \
```

- This command compiles Python using multiple CPU cores `(-j "$nproc")` to speed up the build process.

Lines 61-64:

```
make -j "$nproc" \
      "EXTRA_CFLAGS=${EXTRA_CFLAGS:-}" \
      "LDFLAGS=${LDFLAGS:--Wl},-rpath='\$\$ORIGIN/../lib'" \
      python \
```

- This ensures that Python is built with proper library paths and linking options `(LDFLAGS).`

Line 66:

```
make install; \
```

- This command installs the compiled Python binaries into the system after they have been built. It ensures that the compiled version is correctly placed in the system directories and available for use.

Purpose of `make` in the Dockerfile:

1. Compiles Python from source with parallel execution `(-j "$nproc")` to optimize build time.
2. Ensures proper linking and library paths, so that Python can run correctly without conflicts with system-installed versions.
3. Installs the compiled Python binaries into the system, making them available for execution.