howest
hogeschool

Lecture VII
A DevOps Journey : Container Essentials

Expanding your Containers
Into a full ecosystem
Guy Van Eeckhout
2025-2026

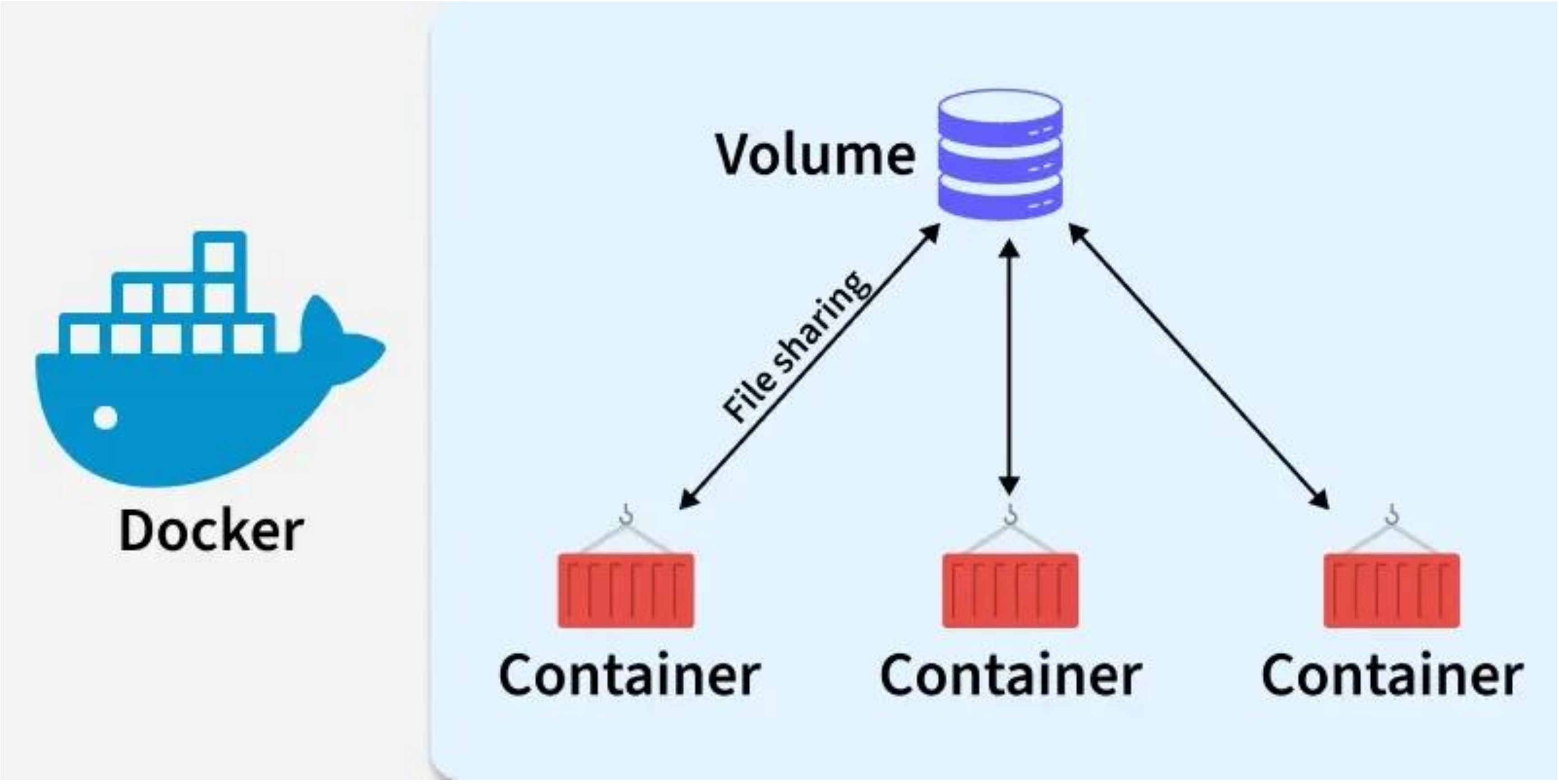
(inspired by Alex Desmedt)

Docker volumes

- No persistent storage on your containers
- If you remove your container, you lose your data !
- Adding Persistent data
  - Volumes
  - Bind mounts

Docker volumes

- Managed by the Docker daemon
- Seperate directory to store data persistently
- Can be used to
  - Load Configuration files
  - Mount Code files
  - Store Database files



Docker volumes – command Cheat Sheet

Command	Purpose
docker volume ls	List all available volumes
docker volume create <volume_name>	Create a new volume (the name is optional. If you do not specify one, docker generates a hash for a volumename
docker volume inspect <volume_name>	Gives infirmation about the volume
docker volume rm <volume_name>	Deletes the volume from your (docker) host
docker volume prune	Removes all unused volumes

Docker volumes

- You can attach a volume to a container at start time (**docker run**)
- You can use --mount or --volume
  - --mount: Everything you can do with a mount on your host !
  - --volume: more limited in options

- **docker run --mount type=volume,src=<volume-name>,dst=<mount-path>**
- **docker run --volume <volume-name>:<mount-path>**

Docker volumes --mount

Option	Description
source, src	The source of your mount
destination, dst, target	The path in your container where your want the volume to be mounted
volume-subpath	Option to mount a part of a volume in your container. The path must already exist in that volume
readonly, ro	Mount the volume as readonly in your container
volume-nocopy	The data on the destination must not be copied to the volume if it’s empty
volume-opt	Key-value pair to pass extra parameters

Docker volumes --volume

- Specific options besides *src* and *dst:*

Option	Description
readonly, ro	Mount the volume as readonly in your container
volume-nocopy	The data on the destiation must not be copied to the volume if it’s empty

Docker volumes

- Beware: the mount point in your container will be overwritten by your mounted volume!
  - **docker run --mount type=volume,src=my_volume,dst=/etc**
  - full /etc directory will be overwritten

Bind mounts

Bind mounts

- Not managed by Docker daemon
- Directory directly linked to your container
- **docker run --mount type=bind,src=<host-path>,dst=<container-path>**

howest
hogeschool

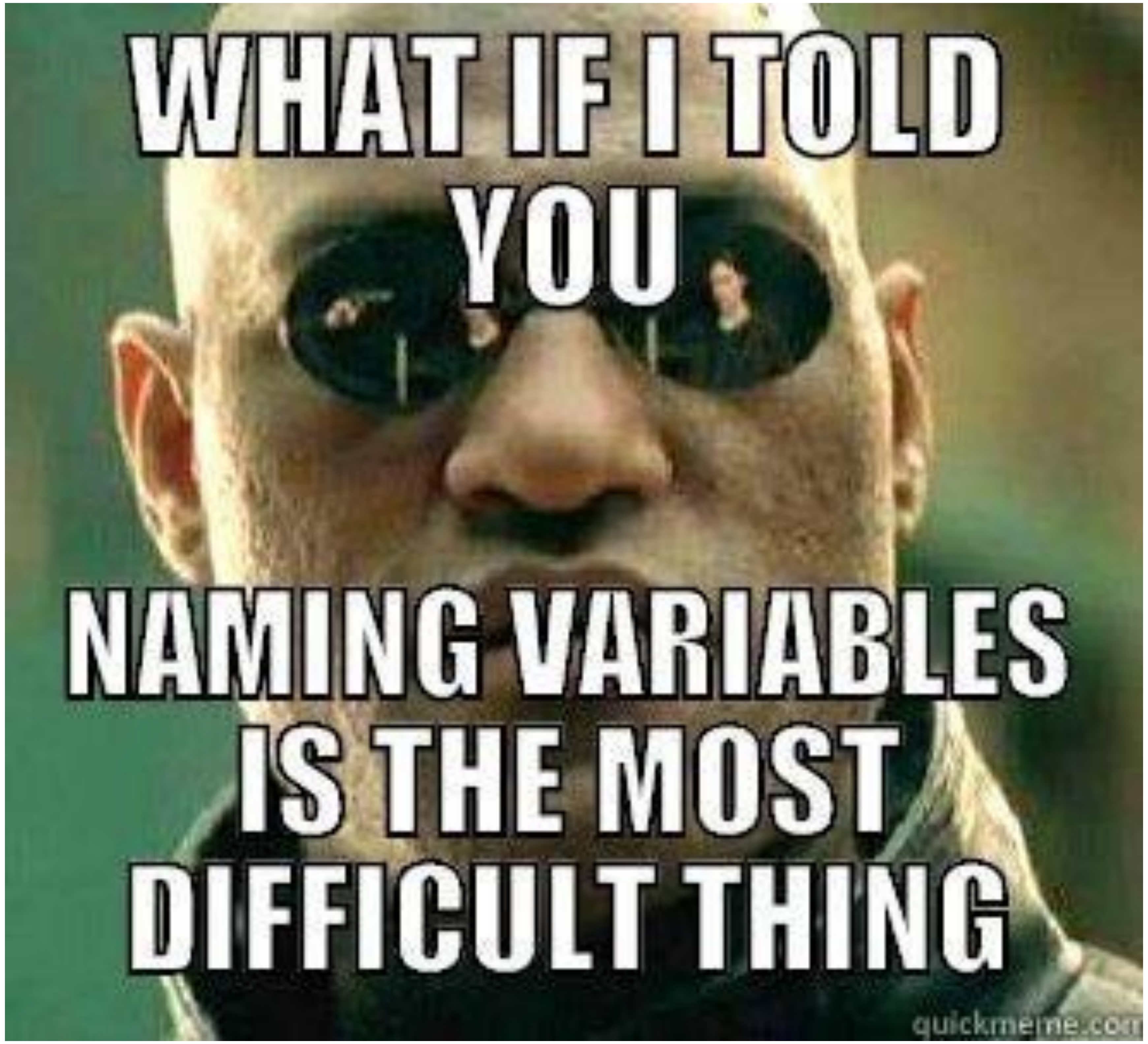
Environment variables

Environment variables

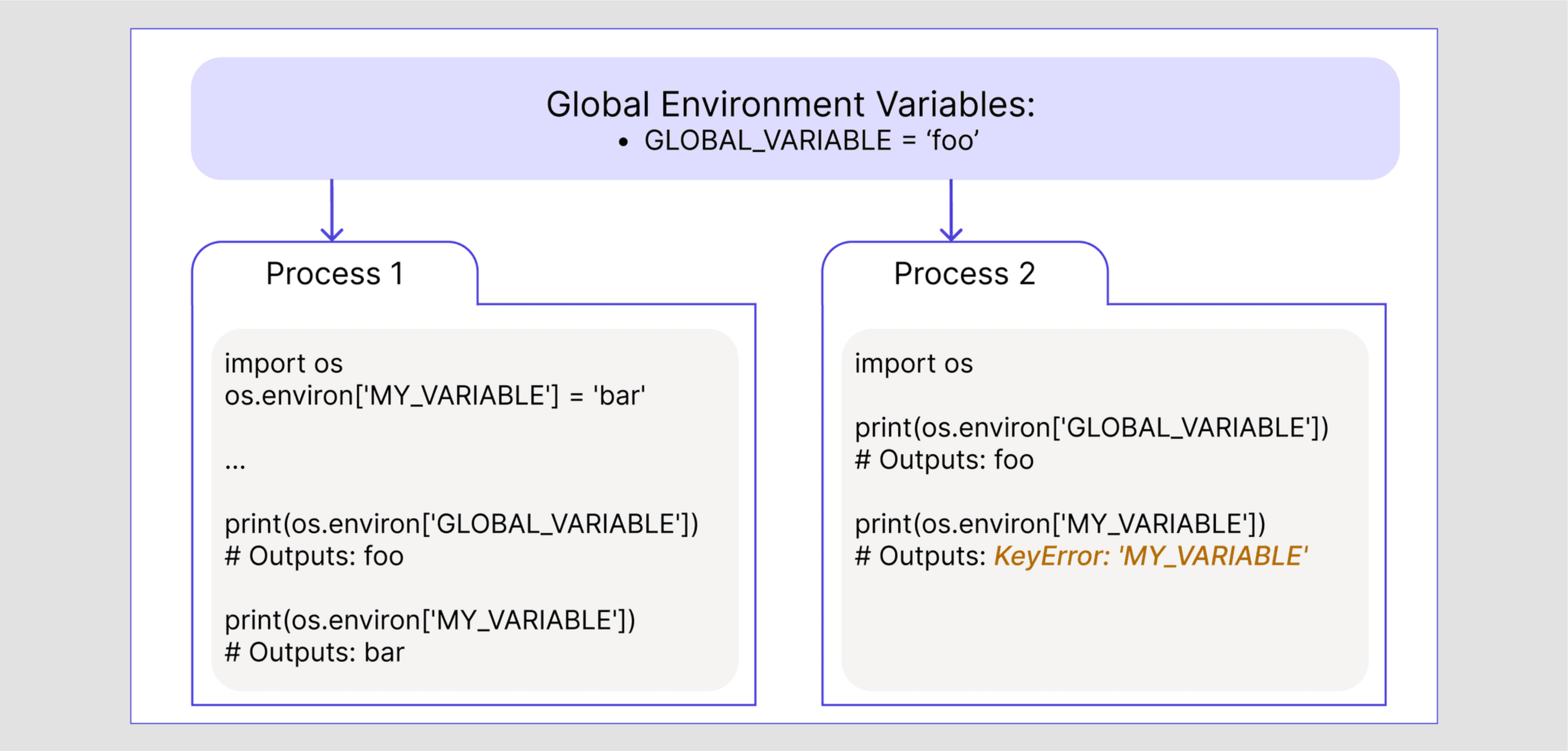
- Variables in your OS
- Key-value pairs
- Modify behaviour / configuration
- Use echo to check the value of a variable : e.g. **echo $HOSTNAME**
- Want to see all currently set variables ? **echo $<tab><tab>**

Environment variables in Docker

- You can pass them along your **docker run** using -e parameter
- E.g.: **docker run -e MYENV=content httpd**
- These variables can then by read by processes in the container



Environment variables in code



Global Environment Variables:

- GLOBAL_VARIABLE = ‘foo’

Process 1

```
import os
os.environ['MY_VARIABLE'] = 'bar'

...

print(os.environ['GLOBAL_VARIABLE'])
# Outputs: foo

print(os.environ['MY_VARIABLE'])
# Outputs: bar
```

Process 2

```
import os

print(os.environ['GLOBAL_VARIABLE'])
# Outputs: foo

print(os.environ['MY_VARIABLE'])
# Outputs: KeyError: 'MY_VARIABLE'
```

howest
hogeschool

Container registries

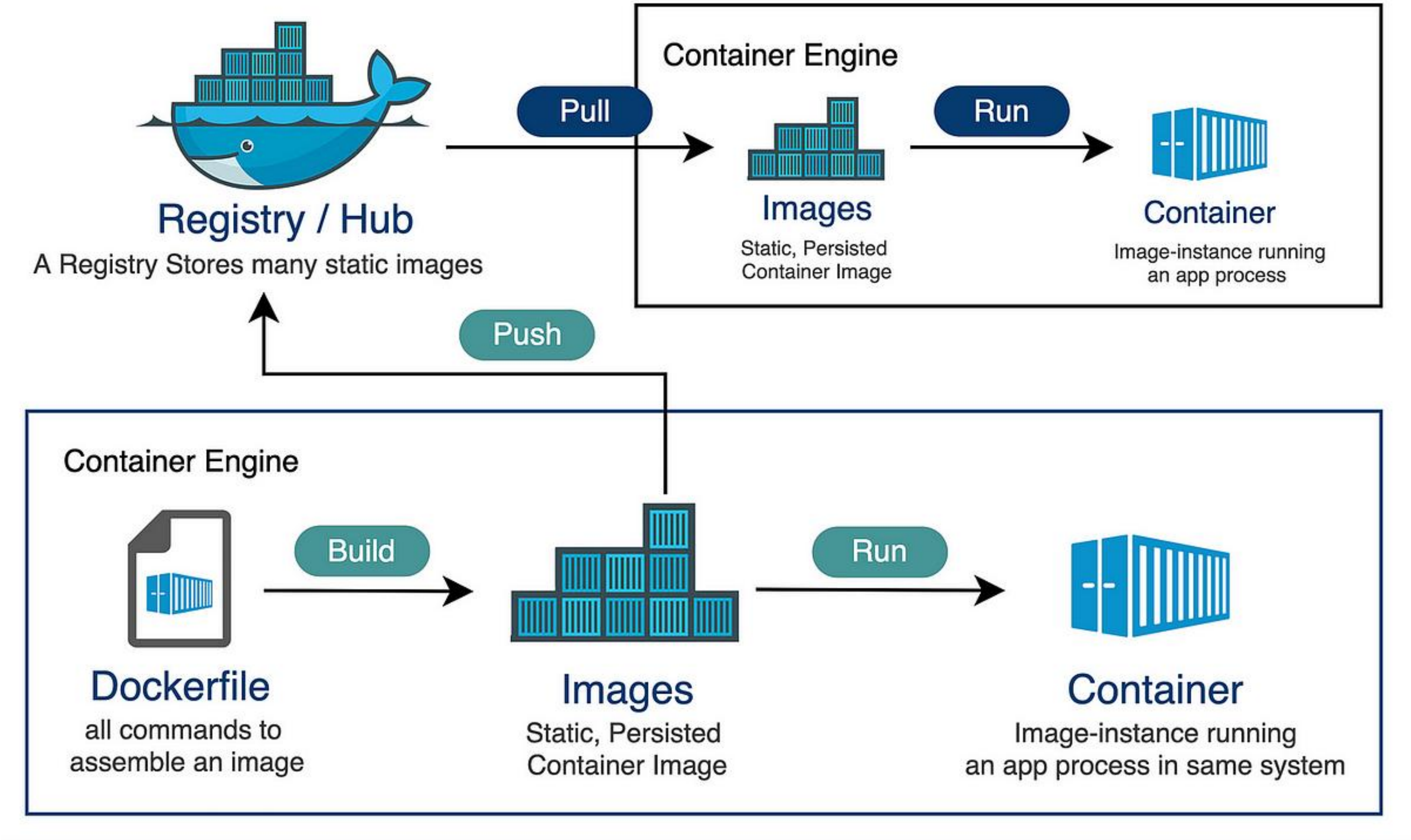
Container registries

- Store container **images**
- Public registries:
  - Dockerhub: hub.docker.com or docker.io
  - Quay.io: RedHat container registry
  - Linuxserver.io: Open Source container registry
  - …
- Docker default = Dockerhub

Private container registries

- Private = limited access
- You can build your own !
  - *Private container registry container* ☺
- Existing Products:
  - Registry container
  - Harbor
  - Private Gitea
  - Private Gitlab

Container registries



Anatomy of a Docker Image Name

[registry]/[username]/[repository]:[tag]

- **Registry:** Where the image is stored (docker.io, ghcr.io, etc.)
- **Username:** Your (Docker Hub) account name
- **Repository:** Project name
- **Tag:** Version identifier (semantic versioning recommended, see tag rant)

Examples

- `nginx:latest` (official image, latest version)
- `postgres:15-alpine` (specific version, alpine variant)
- `username/my-app:v1.0` (personal repository)
- `ghcr.io/username/app:main` (GitHub Container Registry)

Working with Docker Hub: Basic Flow

```
docker login              # Authenticate to Docker Hub
docker pull image:tag     # Download image
docker push image:tag     # Upload image
docker logout             # Remove authentication
```

Important:

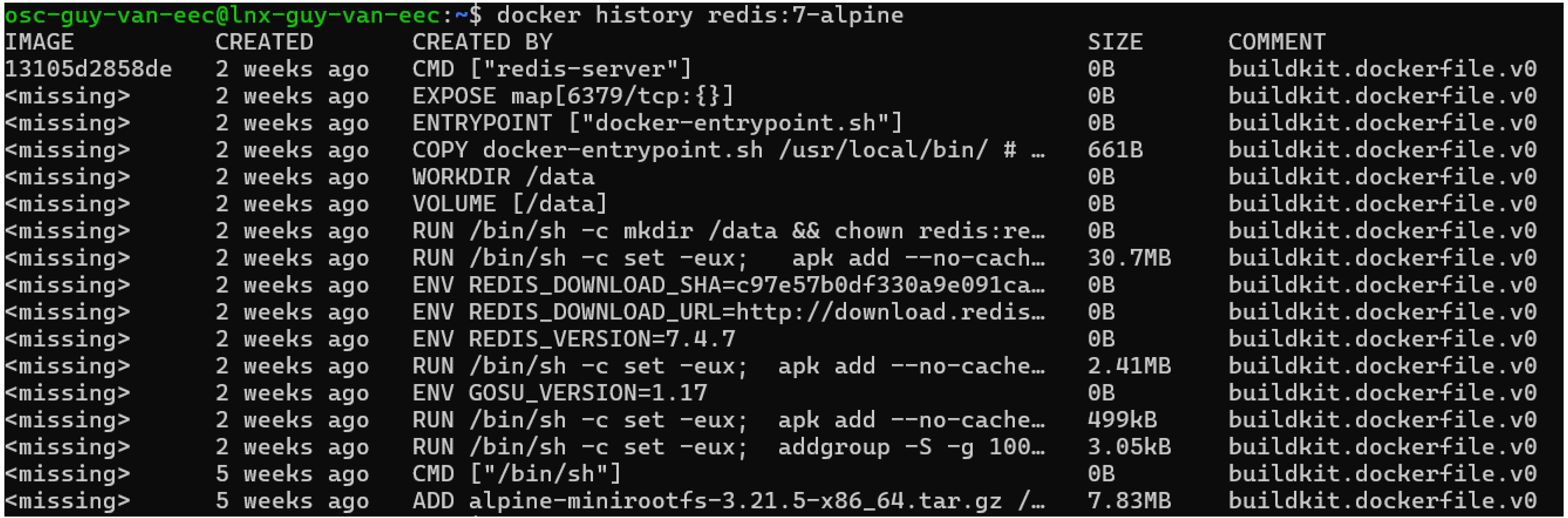
- `docker login` requires Docker Hub account
- Authentication needed to push images
- Pull doesn't always require authentication (public images)
- Credentials stored in `~/.docker/config.json`

howest
hogeschool

Tweaking / Modifying images

Docker Image Architecture: Layers

- Docker images consist of multiple read-only layers stacked on top of each other
- Each layer represents a set of filesystem changes
- Layers are immutable and reusable
- Bottom layer: base OS, top layer: application

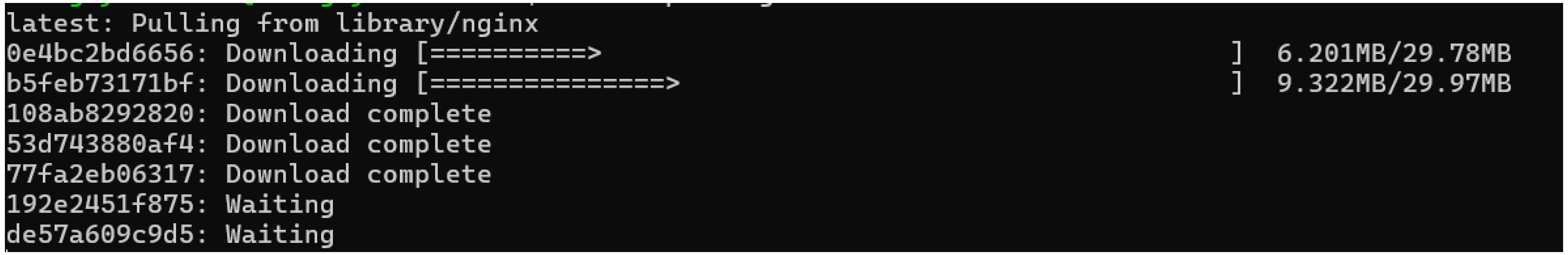


```
osc-guy-van-eec@lnx-guy-van-eec:~$ docker history redis:7-alpine
IMAGE          CREATED       CREATED BY                                      SIZE      COMMENT
13105d2858de   2 weeks ago   CMD ["redis-server"]                            0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   EXPOSE map[6379/tcp:{}]                         0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   ENTRYPOINT ["docker-entrypoint.sh"]             0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   COPY docker-entrypoint.sh /usr/local/bin/ # …   661B      buildkit.dockerfile.v0
<missing>      2 weeks ago   WORKDIR /data                                   0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   VOLUME [/data]                                  0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   RUN /bin/sh -c mkdir /data && chown redis:re…   0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   RUN /bin/sh -c set -eux;   apk add --no-cach…   30.7MB    buildkit.dockerfile.v0
<missing>      2 weeks ago   ENV REDIS_DOWNLOAD_SHA=c97e57b0df330a9e091ca…   0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   ENV REDIS_DOWNLOAD_URL=http://download.redis…   0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   ENV REDIS_VERSION=7.4.7                         0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   RUN /bin/sh -c set -eux;  apk add --no-cache…   2.41MB    buildkit.dockerfile.v0
<missing>      2 weeks ago   ENV GOSU_VERSION=1.17                           0B        buildkit.dockerfile.v0
<missing>      2 weeks ago   RUN /bin/sh -c set -eux;  apk add --no-cache…   499kB     buildkit.dockerfile.v0
<missing>      2 weeks ago   RUN /bin/sh -c set -eux;  addgroup -S -g 100…   3.05kB    buildkit.dockerfile.v0
<missing>      5 weeks ago   CMD ["/bin/sh"]                                 0B        buildkit.dockerfile.v0
<missing>      5 weeks ago   ADD alpine-minirootfs-3.21.5-x86_64.tar.gz /…   7.83MB    buildkit.dockerfile.v0
```

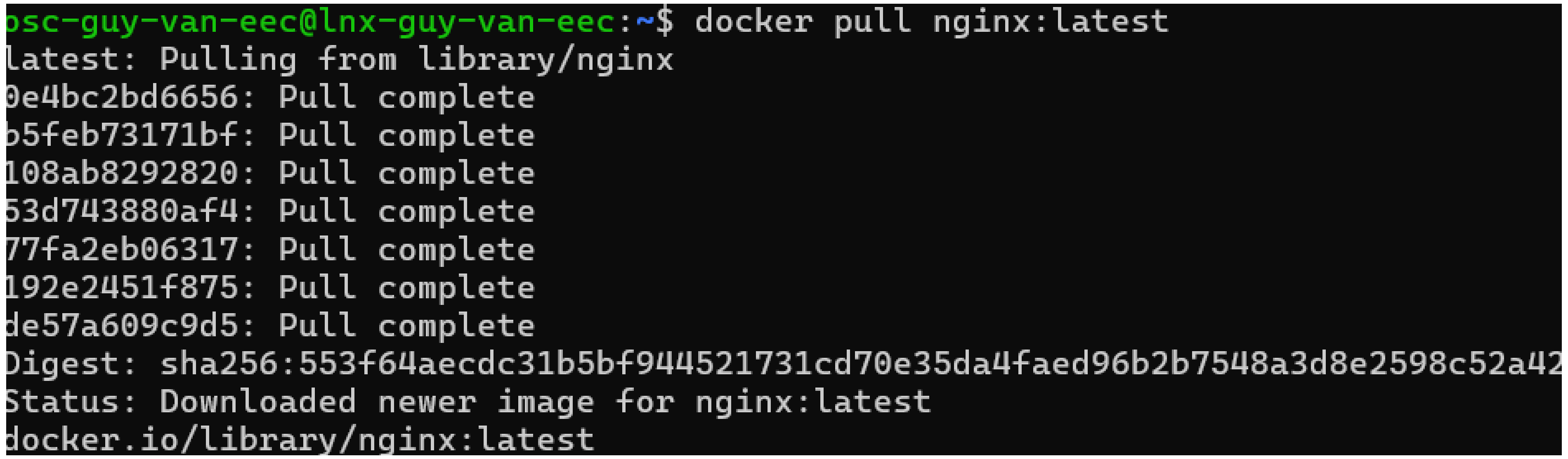
Docker Image Architecture: Layers

docker pull nginx:latest

1. Contact Docker Hub
2. Verify image exists
3. Download each layer
4. Verify checksums
5. Store locally



```
latest: Pulling from library/nginx
0e4bc2bd6656: Downloading [==========>                                        ]  6.201MB/29.78MB
b5feb73171bf: Downloading [================>                                  ]  9.322MB/29.97MB
108ab8292820: Download complete
53d743880af4: Download complete
77fa2eb06317: Download complete
192e2451f875: Waiting
de57a609c9d5: Waiting
```



```
psc-guy-van-eec@lnx-guy-van-eec:~$ docker pull nginx:latest
latest: Pulling from library/nginx
0e4bc2bd6656: Pull complete
b5feb73171bf: Pull complete
108ab8292820: Pull complete
53d743880af4: Pull complete
77fa2eb06317: Pull complete
192e2451f875: Pull complete
de57a609c9d5: Pull complete
Digest: sha256:553f64aecdc31b5bf944521731cd70e35da4faed96b2b7548a3d8e2598c52a42
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

Docker Image Architecture: Layers

docker run -d -p 8080:80 nginx:latest

- Creates container
  - From image nginx:latest
  - With a writable layer on top of image
- Starts the process defined in image
- Container exits when main process stops

Modify a running container

Basic Container modifications

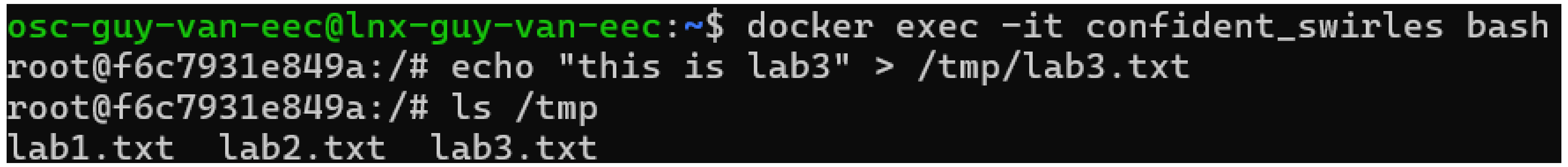
How to modify a running container ?

1) docker exec -it container_name bash

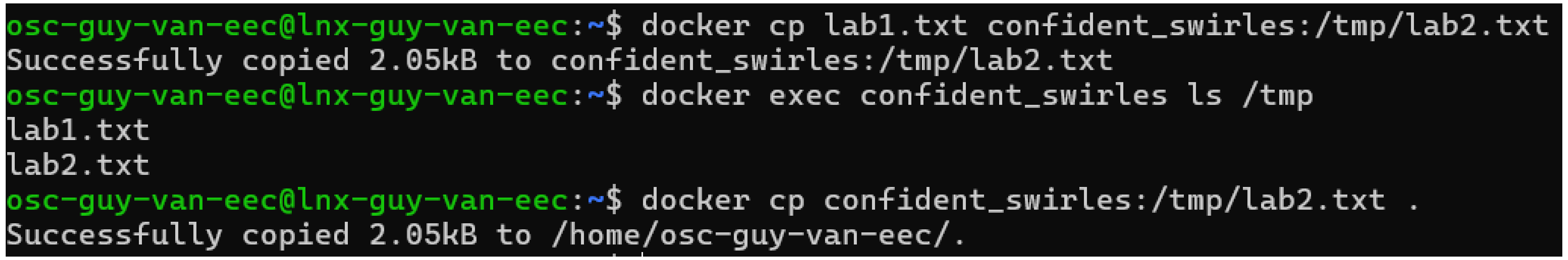
- Enters container shell
- Can inspect, debug, make changes

2) docker cp file container:/path

2) docker cp container:/path file

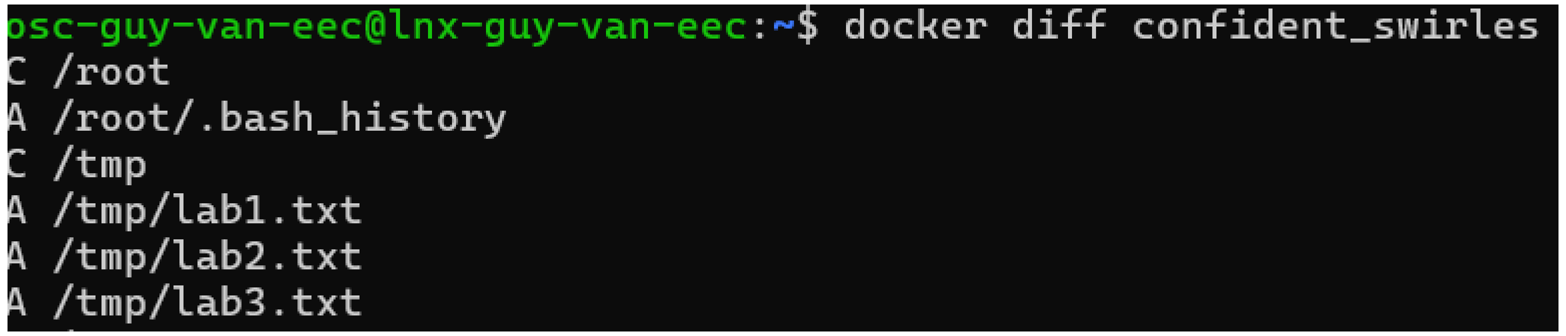


```
osc-guy-van-eec@lnx-guy-van-eec:~$ docker exec -it confident_swirles bash
root@f6c7931e849a:/# echo "this is lab3" > /tmp/lab3.txt
root@f6c7931e849a:/# ls /tmp
lab1.txt  lab2.txt  lab3.txt
```



```
osc-guy-van-eec@lnx-guy-van-eec:~$ docker cp lab1.txt confident_swirles:/tmp/lab2.txt
Successfully copied 2.05kB to confident_swirles:/tmp/lab2.txt
osc-guy-van-eec@lnx-guy-van-eec:~$ docker exec confident_swirles ls /tmp
lab1.txt
lab2.txt
osc-guy-van-eec@lnx-guy-van-eec:~$ docker cp confident_swirles:/tmp/lab2.txt .
Successfully copied 2.05kB to /home/osc-guy-van-eec/.
```

Basic Container modifications



```
osc-guy-van-eec@lnx-guy-van-eec:~$ docker diff confident_swirles
C /root
A /root/.bash_history
C /tmp
A /tmp/lab1.txt
A /tmp/lab2.txt
A /tmp/lab3.txt
```

Changes are LOST when container is removed !

Modify a container image

Option 1 : docker commit

Docker commit : Workflow and usage

Basic Workflow:

1. Start a container from base image
2. Make changes in that running container (install packages, create files)
3. Exit container
4. Use `docker commit` to save your changes as new image

When to Use:

Quick experimentation,Learning Docker,One-off fixes (not recommended for production)

When NOT to Use:

Production deployments,Team environments,Long-term maintenance,Complex applications

Docker commit syntax

docker commit [OPTIONS] CONTAINER **[REPOSITORY[:TAG]]**

Image

Key Options:

- `-a, --author` - Set image author
- `-m, --message` - Describe what changed
- `-p, --pause` - Pause container during commit (default: true)

Docker commit example

1. Start base container:

```
docker run -it --name dev-env ubuntu:latest /bin/bash
```

2. Make modifications inside:

```
apt update; apt install -y python3 pip curl
echo "Setup complete" > /app/status.txt
exit
```

3. Commit:

```
docker commit -a "Guy" -m "Python3 with utilities" dev-env my-python-env:v1.0
```

Image

4. Verify:

```
docker history my-python-env:v1.0
```

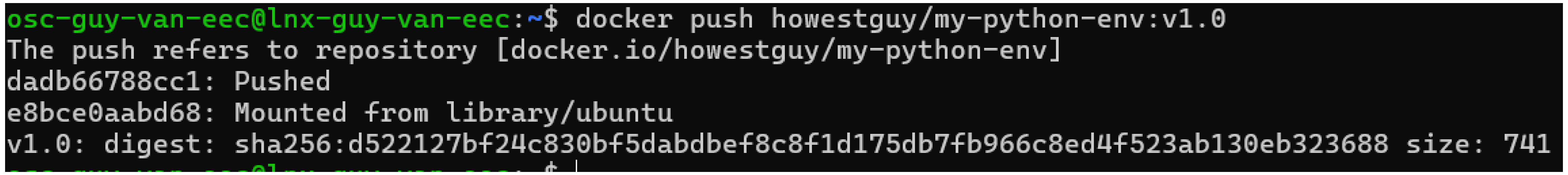
Pushing an image to the repository

docker tag ***my-python-env:v1.0 hubusername/my-python-env:v1.0***

docker push hubusername/my-python-env:v1.0

Image tagged to repository format

1. Checks authentication
2. Verifies image exists locally
3. Uploads layers (only new ones!)
4. Updates registry metadata
5. Creates/updates repository tags

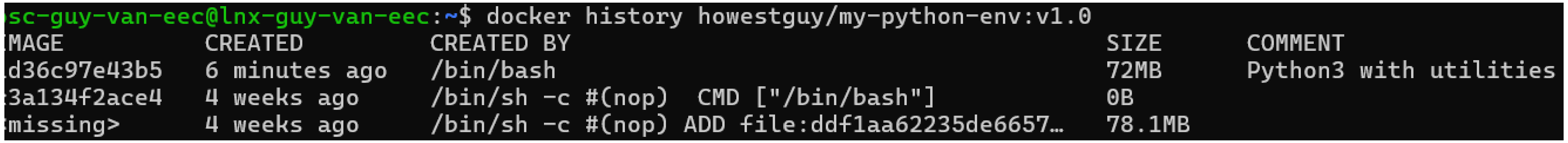


```
osc-guy-van-eec@lnx-guy-van-eec:~$ docker push howestguy/my-python-env:v1.0
The push refers to repository [docker.io/howestguy/my-python-env]
dadb66788cc1: Pushed
e8bce0aabd68: Mounted from library/ubuntu
v1.0: digest: sha256:d522127bf24c830bf5dabdbef8c8f1d175db7fb966c8ed4f523ab130eb323688 size: 741
```

Can now be pulled anywhere with

docker pull hubusername/my-python-env:v1.0

Docker commit caveats



```
sc-guy-van-eec@lnx-guy-van-eec:~$ docker history howestguy/my-python-env:v1.0
MAGE           CREATED          CREATED BY                                      SIZE      COMMENT
d36c97e43b5    6 minutes ago    /bin/bash                                       72MB      Python3 with utilities
3a134f2ace4    4 weeks ago      /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
missing>       4 weeks ago      /bin/sh -c #(nop) ADD file:ddf1aa62235de6657…   78.1MB
```

Not reproducible

- Manual steps not documented, Another developer can't recreate exact image, Difficult to debug what changed

No version control

- Binary artifacts can't be tracked in git,No revision history, Can't compare changes between versions

Poor documentation

- No clear record of what was installed/configured, Hard to maintain long-term, Difficult for team collaboration

Inefficient builds

- Layer caching doesn't work well, Can't easily modify specific steps, Large image sizes common

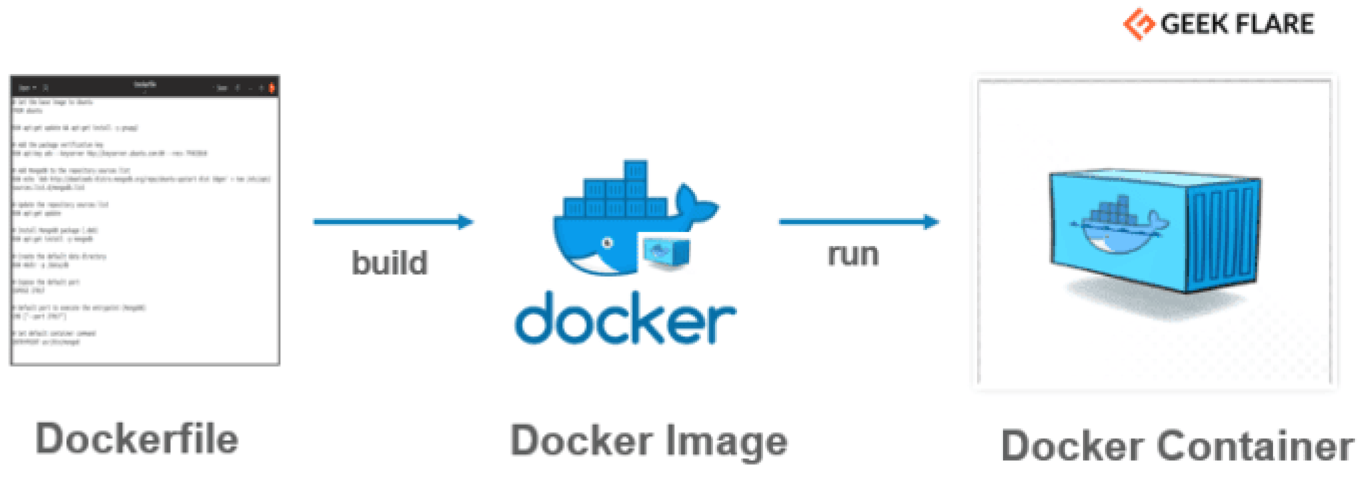
Modify a container image

Option 2 : docker build !

All hail the Dockerfile : Infrastructure as Code

Why Dockerfiles ?

- Reproducible: Same Dockerfile = same image every time
- Version controlled: Track changes in git
- Documented: Each instruction is self-documenting
- Efficient: Layer caching speeds up builds
- Maintainable: Easy to modify and debug
- Team-friendly: Fellow Team Members can understand and modify



Anatomy of a Dockerfile

```
# Comments start with #
# Specify base image
FROM ubuntu:22.04
# Set working directory
WORKDIR /app
# Copy files
COPY requirements.txt .
# Run commands (install, configure)
RUN apt update && apt install -y python3 pip
# Set environment variables
ENV APP_ENV=production
# Expose ports
EXPOSE 8000
# Set default command
CMD ["python3", "app.py"]
```

Each instruction creates a layer in the container image you are building

FROM: Specifying Base Image

Purpose: Define the starting point for your image

Examples:

```
FROM ubuntu:22.04          # Ubuntu base
FROM python:3.11-slim       # Python runtime
FROM node:18-alpine        # Node with minimal footprint
FROM scratch               # Empty image (advanced)
```

Best Practices:

- Pin specific version (avoid `latest`)
- Use `-alpine` variants for small images
- Choose minimal base images (alpine, slim)

WORKDIR: Setting Working Directory

Purpose: Set the working directory for subsequent commands

Examples:

```
FROM python:3.11
WORKDIR /app
# Now all subsequent commands run in /app
COPY . .
RUN pip install -r requirements.txt
# These run in /app context
```

Benefits:

- Cleaner paths in RUN commands
- Container starts in correct directory
- Prevents accidental file overwrites

COPY and ADD: Adding Files to Image

Purpose: insert new files into the image you are building

Examples:

```
# COPY: Simple File Copy from the build context directory to a directory in the image
# COPY <source> <destination>
COPY requirements.txt /app/
# ADD: Advanced (File + Extraction)
# ADD <source> <destination>
ADD app.tar.gz /app/    # Automatically extracts
ADD https://example.com/file /app/  # Downloads file
```

Order matters! Put frequently-changed files last for better caching.

RUN: Executing Commands During Build

Purpose: Execute commands in container during build

Examples:

```
# Shell form
RUN apt update && apt install -y python3
# Exec form (preferred for consistency)
RUN ["apt", "install", "-y", "python3"]
# Multi-line (using && to chain commands)
RUN apt update && \
    apt install -y python3 && \
    apt clean && \
    rm -rf /var/lib/apt/lists/*
```

****Optimization Tips:**
Chain commands with \`&&\` to reduce layers
Clean up after installations (remove cache)
Order from least to most frequently changed**

ENV: Environment Variables

Purpose: Set environment variables in image

Examples:

ENV APP_ENV=production
ENV PYTHON_UNBUFFERED=1
ENV NODE_ENV=development
ENV PORT=3000

Variables can then be used in a running container :

e.g. echo $NODE_ENV # outputs “development”

EXPOSE: Documenting Ports

Purpose: Document which ports the application listens on

Examples:

```
EXPOSE 80       # HTTP
EXPOSE 443      # HTTPS
EXPOSE 3000     # Node.js
EXPOSE 5432     # PostgreSQL
```

Important Notice : EXPOSE doesn't actually publish ports. It only documents which ports are used. You must still use `-p` flag when running the actual container to map ports

```
docker run -p 8080:80 myimage # Maps port 80 (from EXPOSE) to 8080
```

CMD and ENTRYPOINT: Default Process

Purpose: Default process when container starts

Simple Example:

CMD ["node", "app.js"] # Can be overridden! E.g. by docker run image python script.py

or

ENTRYPOINT ["python", "app.py"]

# Can't be overridden with different command, docker run args are passed to ENTRYPOINT

***Combined Usage**:*

ENTRYPOINT ["python"]

CMD ["app.py"]

docker run image → python app.py

docker run image script.py → python script.py

USER: Running as Specific User

Purpose: Specify which user is the default user inside the container

Example:

```
FROM python:3.11
WORKDIR /app
# Create non-root user
RUN useradd -m appuser
# Run as appuser (not root)
USER appuser
CMD ["python", "app.py"]
```

Security Best Practice:

- **Don't run containers as default user root**
- **Create dedicated unprivileged user**
- **Reduces damage if container is compromised**

Building your image (syntax)

docker build [OPTIONS] <context>

docker build -t image:tag .

Key Concepts:

Context: Directory containing Dockerfile and files to copy

Tag: Name and version for resulting image

Common Options:

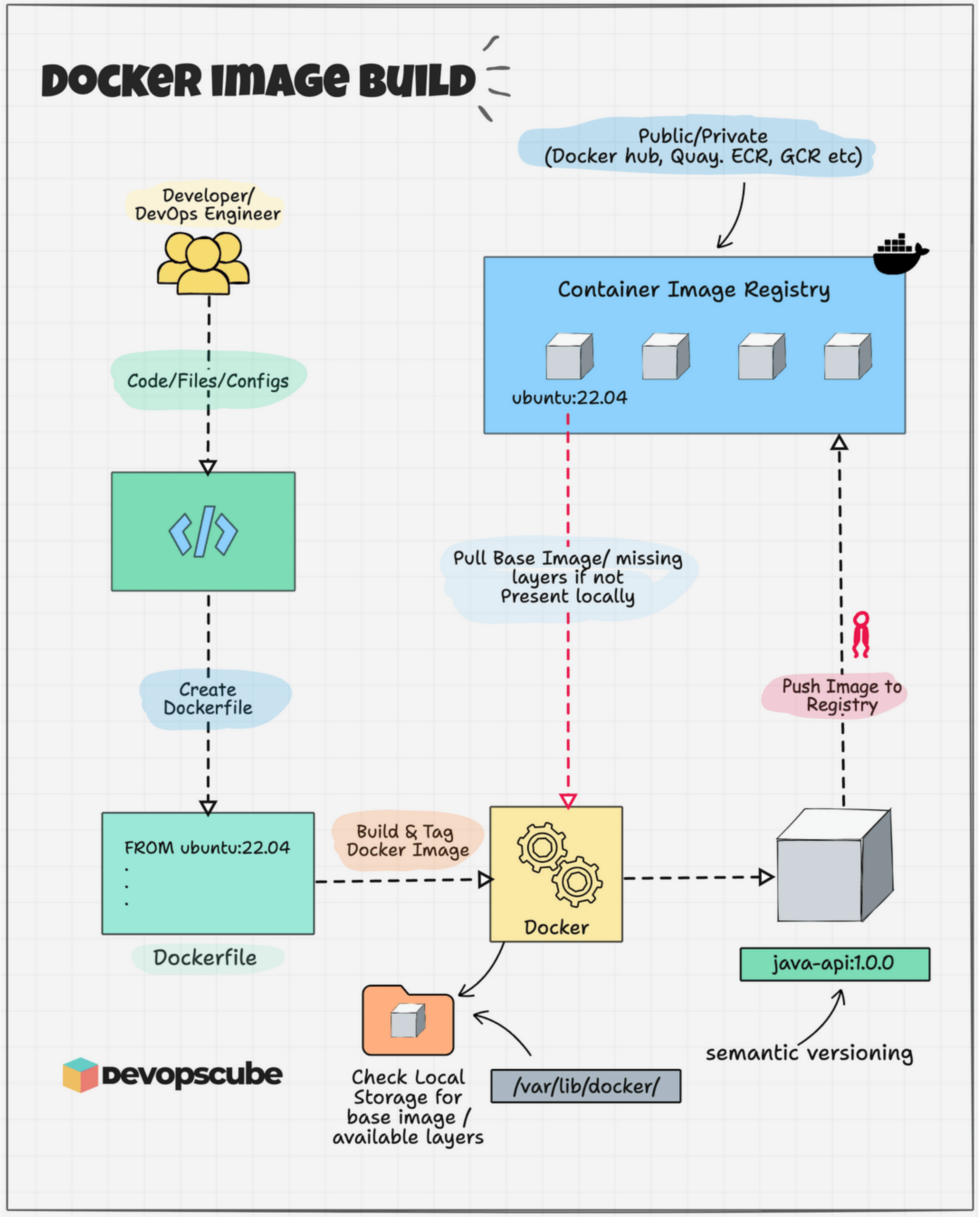
```
-t image:tag          # Tag the image
--no-cache            # Rebuild without using cache
-f path/to/Dockerfile # Use different Dockerfile
--build-arg VAR=value # Pass build arguments
```

docker build -t my-app:v1.0 --build-arg ENV=production .

The context is . so in other words :
The context is the current directory

Building your image : the build process

1. Docker reads Dockerfile line by line
2. For each instruction:
    1. Check if layer exists in cache
        1. If yes, reuse cached layer
        2. If no, execute instruction and create layer
3. Final image = stack of all layers
4. Tag the resulting image
5. Ready to push to the repository



When one container isn't enough

Docker compose gets you all the containers you need

Docker Compose: Multi-Container Orchestration

- One container?
  - Easy ☺
- Multiple containers ?
  - (web app + database + cache) ?
  - Must start each container separately
  - Manage networking manually
  - Remember all flags and options
  - Difficult for teams to reproduce

How Docker Compose Works

1. Services:

   Individual containers in your application

   e.g. web, database, cache

2. Networks:

   Internal communication between services

   Services auto-discover each other by name

   No need to hardcode IP addresses !

3. Volumes:

   Persistent storage shared between services

   Database data persists between restarts

   Shared configuration files

4. Orchestration:

   Start/stop entire stack with one command

   Manages startup order

   Handles dependencies

   Monitors health

Docker Compose = Define entire multi-container application in single YAML file

Anatomy of the docker-compose.yml

Example:

```
services:
    web:
        image: nginx:latest
        ports:   - "8080:80"
    database:
        image: postgres:15
        environment:
            - POSTGRES_PASSWORD=secret
volumes:
    db-data:
networks:
    app-net:
        driver: bridge
```

Services: Defining Member Containers

Each service = one container in your application

In yaml :

```
servicename:
        image: imagename:tag     # Pre-built image
        # OR
        build: ./path            # Build from Dockerfile
        container_name: name     # Custom container name
        ports:
                - "host:container"     # Port mapping
        environment:
                - VAR=value            # Environment variables
        volumes:
                - /host:/container     # mounts
        depends_on:
                - otherservice         # Start order
```

= pulled from repository

= Path to Dockerfile, build **context**

Expose container ports to host machine

So this does MORE than EXPOSE in the Dockerfile !

Example :

```
services:
        web:
                image: nginx:latest
                ports:
                        - "8080:80"       # HTTP
                        - "443:443"       # HTTPS
                        - "5000:5000/udp"  # UDP
```

Note: The container port is defined in Dockerfile; host port for access in .yaml file

Environment: Setting Variables

```
services:
    database:
        image: postgres:15
        environment:
            - POSTGRES_USER=appuser
            - POSTGRES_PASSWORD=secret
            - POSTGRES_DB=myapp
    app:
        image: myapp:latest
        environment:
            - DATABASE_HOST=database
            - DATABASE_USER=appuser
```

Volumes: Persistent Storage

1. Named Volumes (recommended)

```
services:
    database:
            volumes:
            - db_data:/var/lib/postgresql/data
volumes:  db_data:  # Defined at top level, outside services: scope
```

2. Bind Mounts (direct file access)

```
services:
    app:
            volumes:
            - ./app:/app  # Docker host local ./app → container /app
```

Ensure services start in correct order

```
services:
    web:
        depends_on:
            - database
            - cache
    database:
        image: postgres:15
    cache:
        image: redis:7
```

↔

```
services:
  web:
     depends_on:
        database:
           condition: service_healthy
        cache:
           condition: service_healthy
```

“depends_on” waits for container to START not for service to be READY.

Healthcheck: Service Readiness

```
services:
    database:
        image: postgres:15
        healthcheck:
            test: ["CMD", "pg_isready", "-U", "postgres"]
            interval: 10s      # Check every 10 seconds
            timeout: 5s        # Wait 5 seconds for result
            retries: 5         # Fail after 5 failed checks
            start_period: 20s  # Wait 20s before first check
```

Networks: Service Communication

```
services:
    web:
        networks:
            - app_network
    database:
        networks:
            - app_network
networks:
    app_network:
        driver: bridge
```

Service Discovery: Inside `web` container: `ping database` works! Docker internal DNS resolves `database` to its internal IP so no hardcoding IPs needed

Set Container Restart Behavior

```yaml
services:
  web-app:
    image: my-app:latest
    restart: unless-stopped # Recommended for most production web apps
  database:
    image: postgres:16
    restart: always # Ideal for critical services that must run indefinitely
  migration-job:
    image: my-migration-tool
    restart: on-failure:3 # Only restart on error, max 3 times
  test-container:
    image: my-test-image
    restart: no # For one-off tasks (default behavior)
```

Docker Compose Commands Cheat Sheet

Command	Description	Example Usage
up	Create and start containers, networks, and volumes defined in the file. Use -d for detached mode (background).	*docker compose up -d*
down	Stop and remove containers, networks, and images created by up. Use -v to also remove volumes.	*docker compose down (-v)*
ps	List the containers in the current project, showing their status and ports.	*docker compose ps*
logs	View output logs from containers. Use -f to follow logs in real-time, or specify a service name.	*docker compose logs -f web*
exec	Run a command inside a running container. Used for getting a shell prompt.	*docker compose exec web bash*
build	Build or rebuild service images defined in the Dockerfile within the compose file.	*docker compose build web*
start	Start existing containers without recreating them (if they were previously stopped).	*docker compose start web database*
stop	Stop running containers without removing them (so they can be started again quickly).	*docker compose stop*
restart	Restart all or specific service containers.	*docker compose restart web*
rm	Remove stopped containers. Use -f to force removal without confirmation.	*docker compose rm -f*

Today’s LAB

1) Play around with docker volumes
2) Explore the docker hub
3) Pull and Push your container images
4) Create your own container images / container stack
    1) Commit
    2) Build
    3) Compose

Note : all source code is included in the lab document and SHOULD work. If a syntax error happens after copy/paste : check the usual suspects : " became “, -- became – etc