David Christian C. Olimberio

IV – ACSAD

ASSIGNMENT 3 -
DOCKER AND CONTAINERIZATION

What is Docker and Containers

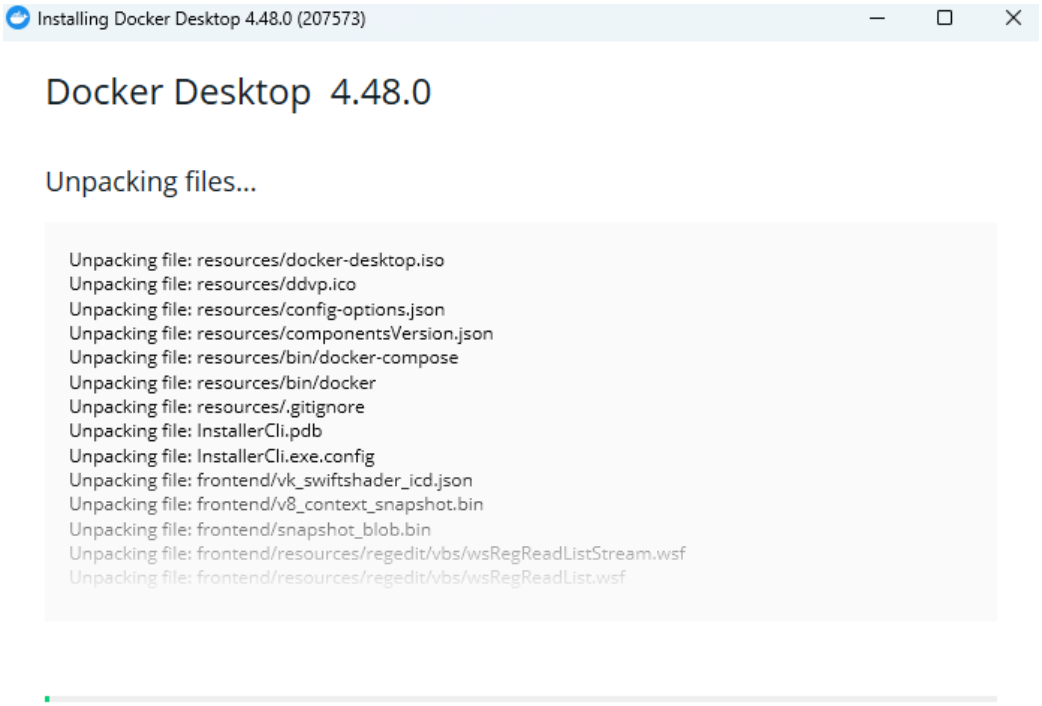
- Docker is a tool that lets you run applications inside containers. A *container* is like a small, isolated computer inside your system that holds your app and everything it needs to run.

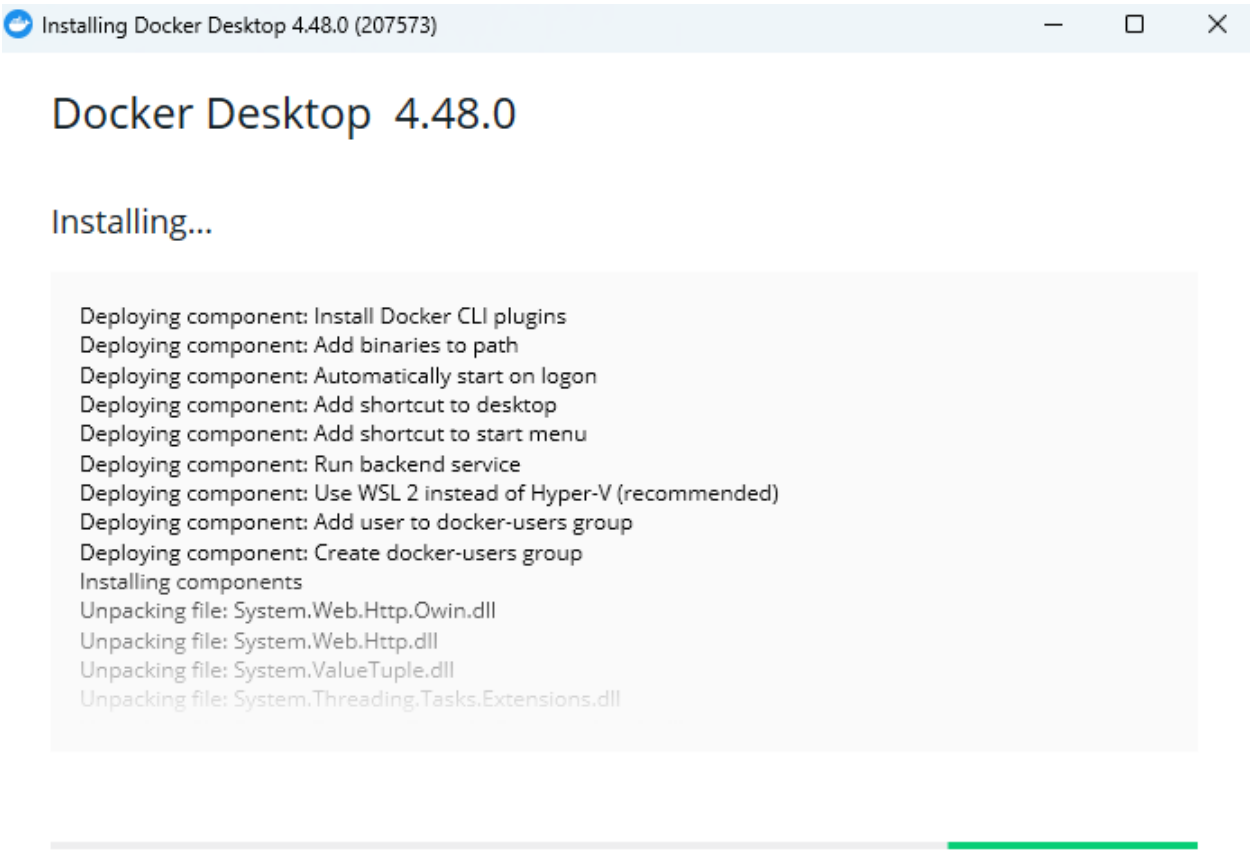
Example:

- Instead of installing Python or Node.js on your PC, Docker runs them inside containers so your main system stays clean.

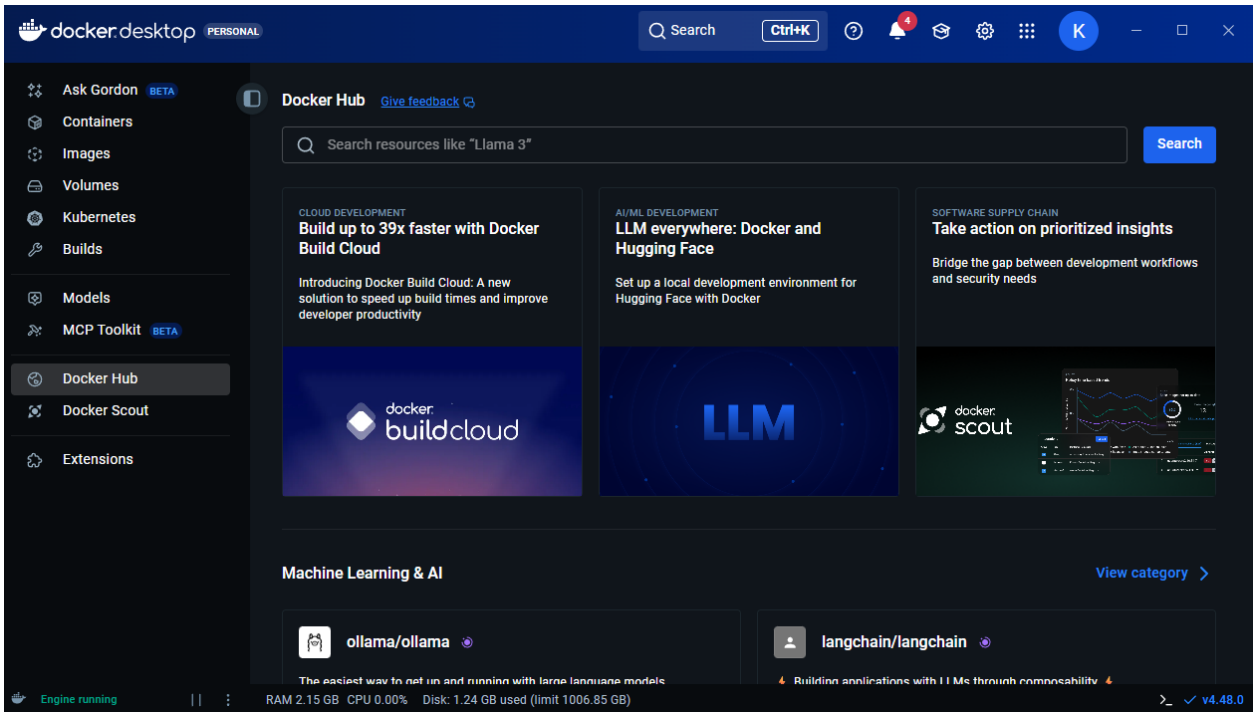
Installing Docker

- Download Docker Desktop: https://www.docker.com/get-started/
- Install and open it.





- Asking to Log-in your Work Account either Google or Github Account
- Checking the version of the Windows Subsystem for Linux
- Opening the Docker Desktop



Run Your First Container

- Open your Terminal
- Let's test Docker with the official *hello-world* image:

```
PS C:\Users\david> docker run hello-world
```

- Output

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

PS C:\Users\david>
```

What it does:

- Downloads a small image from Docker Hub.
- Runs it inside a container.
- Prints a "Hello from Docker!" message if everything works.

Exploring Docker Commands

- docker ps # Show running containers
- docker ps -a # Show all containers
- docker images # Show downloaded images
- docker stop <id> # Stop a container
- docker rm <id> # Remove a container
- docker rmi <image> # Remove an image

```
PS C:\Users\david> docker ps -a
CONTAINER ID   IMAGE         COMMAND    CREATED         STATUS                     PORTS     NAMES
e247bc6c86f3   hello-world   "/hello"   5 minutes ago   Exited (0) 5 minutes ago             sweet_chatelet
3e97a7843a8b   hello-world   "/hello"   6 minutes ago   Exited (0) 6 minutes ago             admiring_noether
PS C:\Users\david> docker rmi hello-world
Error response from daemon: conflict: unable to delete hello-world:latest (must be forced) - container 3e97a7843a8b is using its referenced image 6dc565aa630
PS C:\Users\david> docker rm e247bc6c86f3
e247bc6c86f3
PS C:\Users\david> docker rm 3e97a7843a8b
3e97a7843a8b
PS C:\Users\david> docker rmi hello-world
Untagged: hello-world:latest
Deleted: sha256:6dc565aa630927052111f823c303948cf83670a3903ffa3849f1488ab517f891
```

Create a Simple App Using Docker

- Create a folder

```
PS C:\Users\david\Desktop\Docker> mkdir docker-demo


    Directory: C:\Users\david\Desktop\Docker


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        10/17/2025   9:56 PM                docker-demo


PS C:\Users\david\Desktop\Docker> cd docker-demo
PS C:\Users\david\Desktop\Docker\docker-demo>
```

- Create a simple web app file
    - Create a file named **app.py**:

app.py 1

app.py > ...

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def home():
    return "Hello from Docker Container!"

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

```

- Create a file named **Dockerfile** (no extension):

app.py 1 DockerFile

DockerFile > ...

```
# Use official Python image
FROM python:3.9

# Set working directory
WORKDIR /app

# Copy all files to the container
COPY . .

# Install dependencies
RUN pip install flask

# Run the app
CMD ["python", "app.py"]

```

Build and Run the Container

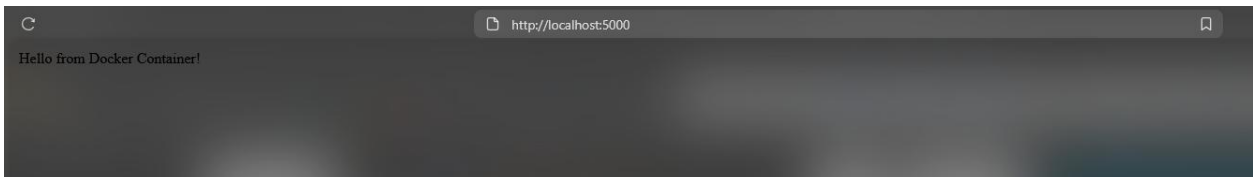
- Build the image: **docker build -t my-flask-app .**

```
PS C:\Users\david\Desktop\Docker\docker-demo> docker build -t my-flask-app .
[+] Building 46.7s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 265B
 => [internal] load metadata for docker.io/library/python:3.9
 => [auth] library/python:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/4] FROM docker.io/library/python:3.9@sha256:ace56aa808ac8bea1ebbb13b0eb99fb15b135b98ed2a90e6fdeb66aca1ccfd3b
 => => resolve docker.io/library/python:3.9@sha256:ace56aa808ac8bea1ebbb13b0eb99fb15b135b98ed2a90e6fdeb66aca1ccfd3b
 => => sha256:48fcb9e811570ebe79723445e94ea554cbc0d9e860df83638ab80722c2a5d92d 6.10MB / 6.10MB
 => => sha256:0001e22f10a7e8471449ba2c2a536614d76f8f1fce2110ac674ccdcb40efb8c0 20.37MB / 20.37MB
 => => sha256:792a4763ca0d414bef79efd8a3c8f4b9cfe72553a5edc73bcfdc075f314b3a1e 248B / 248B
 => => sha256:f0c9d6d993ac93f222ba87ca01097d40f632be9b48f6b5e399f2c5da1b3133d1 67.78MB / 67.78MB
 => => sha256:bd090f42c4b7844c5846f9b4c719994f496fac3befe1d30f0ff20794e742370a 25.61MB / 25.61MB
 => => sha256:a2ade626d67af90eb146ef31070d6021beb378ab38f6477493190d674c44a1e9 235.93MB / 235.93MB
 => => sha256:cae3b572364a7d48f8485d67baee38e4e44e299b8c8c4d020ff7fb5fdd97f88c 49.28MB / 49.28MB
 => => extracting sha256:cae3b572364a7d48f8485d67baee38e4e44e299b8c8c4d020ff7fb5fdd97f88c
 => => extracting sha256:bd090f42c4b7844c5846f9b4c719994f496fac3befe1d30f0ff20794e742370a
 => => extracting sha256:f0c9d6d993ac93f222ba87ca01097d40f632be9b48f6b5e399f2c5da1b3133d1
 => => extracting sha256:a2ade626d67af90eb146ef31070d6021beb378ab38f6477493190d674c44a1e9
 => => extracting sha256:48fcb9e811570ebe79723445e94ea554cbc0d9e860df83638ab80722c2a5d92d
 => => extracting sha256:0001e22f10a7e8471449ba2c2a536614d76f8f1fce2110ac674ccdcb40efb8c0
 => => extracting sha256:792a4763ca0d414bef79efd8a3c8f4b9cfe72553a5edc73bcfdc075f314b3a1e
 => [internal] load build context
 => => transferring context: 495B
 => [2/4] WORKDIR /app
 => [3/4] COPY . .
 => [4/4] RUN pip install flask
 => exporting to image
 => => exporting layers
 => => exporting manifest sha256:17c7c35bf842ef230183ec4f6056ab0e614d59f25d840c0d03ab6779eea89c12
 => => exporting config sha256:0525788cde4f4379fac1b5e61b710d9729e8f68b20106b711fa06a6a56cb8e7b
 => => exporting attestation manifest sha256:b85f9903a960a1a24548dd396c5e7e39e9686d46a63c366d869d933d3af245e6
 => => exporting manifest list sha256:9cafab972a5c681d4baece1a35fd005983237226584 2b048aad2a61c6a4cb04d
 => => naming to docker.io/library/my-flask-app:latest
 => => unpacking to docker.io/library/my-flask-app:latest
```

- Run the container: **docker run -p 5000:5000 my-flask-app**

```
PS C:\Users\david\Desktop\Docker\docker-demo> docker run -p 5000:5000 my-flask-app
 >>
  * 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
172.17.0.1 - - [17/Oct/2025 14:07:17] "GET / HTTP/1.1" 200 -
172.17.0.1 - - [17/Oct/2025 14:07:17] "GET /favicon.ico HTTP/1.1" 404 -
```

- Open your browser and go to: http://localhost:5000
- You should see: **“Hello from Docker Container!”**



Manage and Stop the Container

- See all containers: **docker ps -a**

```
PS C:\Users\david\Desktop\Docker\docker-demo> docker ps -a
CONTAINER ID   IMAGE          COMMAND           CREATED          STATUS                          PORTS     NAMES
f78dd1c4725b   my-flask-app   "python app.py"   4 minutes ago    Exited (0) About a minute ago             awesome_edison
f2b2e7231560   hello-world    "/hello"          17 minutes ago   Exited (0) 17 minutes ago                 xenodochial_blackwell
```

- Stop it: docker stop <container_id>

```
PS C:\Users\david\Desktop\Docker\docker-demo> docker stop f78dd1c4725b
f78dd1c4725b
```

History of Docker and Containerization

1. Early Beginnings (1970s–2000s)

- The concept of containerization began with **chroot** in **1979** on Unix systems, allowing processes to run in isolated directory trees.
- Later technologies like **FreeBSD Jails (2000)** and **Linux Containers (LXC, 2008)** improved isolation and resource control.
- These early methods laid the foundation for modern containers, but they were complex to manage.

2. The Birth of Docker (2013)

- **Docker, Inc.** (originally dotCloud) released **Docker** in **2013**, revolutionizing container technology by making it simple, portable, and developer-friendly.
- Docker introduced easy-to-use tools for packaging applications and dependencies into lightweight containers, ensuring consistency across environments (development, testing, and production).

3. Container Ecosystem Expansion (2014–2018)

- Docker's popularity grew rapidly, leading to widespread adoption by developers and enterprises.
- The introduction of **Docker Compose**, **Docker Swarm**, and **Docker Hub** made container management and sharing easier.
- **Google** introduced **Kubernetes (2014)** — an open-source orchestration system — which became the standard for managing large-scale containerized applications.

4. Modern Era (2019–Present)

- Containers are now integral to **DevOps**, **Cloud Computing**, and **Microservices Architecture**.
- Platforms like **AWS**, **Azure**, and **Google Cloud** natively support Docker and Kubernetes.
- Docker continues evolving, focusing on performance, security, and developer productivity.

Best Practices in Implementing Docker and Containerization

1. Use Lightweight Base Images

- Choose minimal images (e.g., alpine, slim) to reduce size and attack surface. Example:

  FROM python:3.9-alpine

2. Keep Dockerfiles Simple and Clean

- Follow a clear order: **Base image** → **Dependencies** → **Copy** → **Commands**.
- Combine related commands to reduce image layers.

  RUN apt-get update && apt-get install -y curl

3. Use .dockerignore Files

- Exclude unnecessary files (like .git, logs, temp files) to speed up builds and keep images clean.

4. Avoid Running as Root

- Create a non-root user inside your container for better security.

  RUN useradd -m appuser

  USER appuser

5. Use Environment Variables for Configurations

- Store settings and secrets outside your image:

  docker run -e APP_MODE=production my-app

6. Tag and Version Your Images

- Always tag your builds to track versions easily:

  docker build -t myapp:v1.0 .

7. Manage Data with Volumes

- Use **Docker volumes** for persistent data instead of storing data inside containers:

  docker run -v mydata:/app/data myapp

8. Keep Containers Single-Purpose

- Each container should run **one process or service** (e.g., web server, database). Multi-container apps should use **Docker Compose**.

9. Regularly Clean Up Unused Resources

- Remove unused containers and images to save space:

  docker system prune

10. Implement Security Scanning

- Regularly scan your images for vulnerabilities using tools like:

  docker scan myapp