**🐳 Assignment 1: Introduction to Containerization and Docker Fundamentals**

**🔹 1. What is Containerization?**

Containerization is a lightweight form of virtualization that packages applications and their dependencies into containers. Unlike traditional virtual machines (VMs), containers share the host OS kernel and are much more efficient.

**🔹 2. Benefits of Containerization**

* **Portability:** Run anywhere – on-premise, cloud, or hybrid.
* **Isolation:** Each container runs independently.
* **Scalability:** Easily scale applications horizontally.
* **Faster Deployment:** Containers start in seconds.

**🐳 3. What is Docker?**

Docker is an open-source platform used for automating the deployment, scaling, and management of containerized applications.

* **Docker Engine:** Core component for running containers.
* **Docker Hub:** Cloud registry to store and share container images.

**🔹 4. Docker vs Virtual Machines**

| **Feature** | **Docker (Containers)** | **Virtual Machines** |
| --- | --- | --- |
| Boot Time | Seconds | Minutes |
| Resource Usage | Low | High |
| Portability | High | Medium |
| Isolation | Process Level | OS Level |

**🔹 5. Docker Components**

* **Docker Image:** Read-only template (like a snapshot).
* **Docker Container:** A running instance of an image.
* **Dockerfile:** Script to build images.
* **Docker Hub:** Repository to share and pull images.

**📘 Basic Docker Commands with Example**

**🔹 1. Check Docker Version**

bash

docker --version

**🔹 2. Run a Container (Example: Ubuntu)**

bash

docker run -it ubuntu

-it: interactive terminal  
This opens an Ubuntu shell inside a container.

**🔹 3. List Running Containers**

bash

docker ps

**🔹 4. List All Containers (including stopped ones)**

bash

docker ps -a

**🔹 5. List Downloaded Docker Images**

bash

docker images

**🔹 6. Pull an Image from Docker Hub**

bash

docker pull nginx

**🔹 7. Stop a Running Container**

bash

docker stop <container\_id>

**🔹 8. Remove a Container**

bash

docker rm <container\_id>

**🔹 9. Remove an Image**

bash

docker rmi <image\_id>

**🔹 10. Build Image from Dockerfile**

bash

docker build -t myapp .

**📝 Example: Run a Simple Web Server**

bash

docker run -d -p 8080:80 nginx

* -d: detached mode
* -p: map port 8080 on host to 80 in container  
  Now visit http://localhost:8080 to see the NGINX welcome page.

**✅ Conclusion**

Docker revolutionizes the way we build and ship applications. Its speed, simplicity, and efficiency make it the industry standard for containerization.

## 🐳 ****Assignment 2: Docker Installation, Basic Container Operations & Building an Image from Dockerfile****

### 🔹 1. Docker Installation (Ubuntu Linux)

**Step 1: Update Packages**

bash

sudo apt-get update

**Step 2: Install Dependencies**

bash

sudo apt install apt-transport-https ca-certificates curl software-properties-common

**Step 3: Add Docker GPG Key**

bash

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

**Step 4: Add Docker Repository**

bash

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] \

https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable" | \

sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

**Step 5: Install Docker Engine**

bash

sudo apt update

sudo apt install docker-ce docker-ce-cli containerd.io

**Step 6: Verify Installation**

bash

docker --version

### 🔹 2. Basic Docker Container Operations

✅ **Run a Container (Ubuntu)**

bash

docker run -it ubuntu

✅ **List Running Containers**

bash

docker ps

✅ **List All Containers**

bash

docker ps -a

✅ **Stop a Container**

bash

docker stop <container\_id>

✅ **Start a Container**

bash

docker start <container\_id>

✅ **Remove a Container**

bash

docker rm <container\_id>

✅ **Remove an Image**

bash

docker rmi <image\_id>

✅ **Pull an Image from Docker Hub**

bash

docker pull httpd

✅ **Run a Container in Detached Mode**

bash

docker run -d -p 80:80 httpd

### 🔹 3. Build an Image from Dockerfile

#### Step 1: Create a Dockerfile

Dockerfile

# Use official Python image

FROM python:3.9-slim

# Set working directory

WORKDIR /app

# Copy application files

COPY . /app

# Install dependencies

RUN pip install flask

# Set environment variable

ENV FLASK\_APP=app.py

# Expose port

EXPOSE 5000

# Run the Flask app

CMD ["flask", "run", "--host=0.0.0.0"]

#### Step 2: Build the Image

bash

docker build -t flask-app .

#### Step 3: Run the Image

bash

docker run -d -p 5000:5000 flask-app

Open http://localhost:5000 to view the running app.

### ✅ Conclusion:

In this assignment, you learned:

* How to install Docker on a Linux system.
* Run, stop, and manage containers.
* Build a custom image using a Dockerfile.

# Assignment3: Docker Registry, DockerHub, and Multi-Stage Build

## 🔹 1. What is a Docker Registry?

A **Docker Registry** is a storage and distribution system for Docker images. It allows you to **push**, **pull**, and **store** container images.

### ✅ Types of Docker Registries:

* **Public Registry**: Anyone can access it.  
  ➤ Example: DockerHub
* **Private Registry**: Restricted access, used within organizations.  
  ➤ Example: Azure Container Registry, Amazon ECR, Google Container Registry

## 🔹 2. DockerHub – The Official Public Docker Registry

**DockerHub** is the default registry used by Docker. It contains:

* Official images (like nginx, mysql, etc.)
* Community images
* Your personal or organization's image repositories

### 🛠️ Common DockerHub Commands:

✅ **Login to DockerHub**

bash

docker login

✅ **Pull an Image from DockerHub**

bash

docker pull ubuntu

✅ **Tag a Local Image for DockerHub**

bash

docker tag flask-app yourusername/flask-app:latest

✅ **Push an Image to DockerHub**

bash

docker push yourusername/flask-app:latest

✅ **Logout from DockerHub**

bash

docker logout

## 🔹 3. What is a Multi-Stage Docker Build?

In **Multi-Stage Builds**, we use multiple FROM instructions in a single Dockerfile to:

* Compile or build in one stage (heavy)
* Copy only the final executable to a smaller image (lightweight)

### ✅ Benefits:

* Reduces final image size
* Improves security (removes build-time dependencies)
* Clean separation of build and runtime environments

## 🔨 4. Example: Multi-Stage Dockerfile for a Go App

### 📁 Project Structure:

go

/go-app

├── Dockerfile

└── main.go

### 🧾 Dockerfile with Multi-Stage Build:

Dockerfile

# Stage 1: Build the Go binary

FROM golang:1.20 AS builder

WORKDIR /app

COPY . .

RUN go mod init example.com/app && go build -o myapp main.go

# Stage 2: Create lightweight container

FROM alpine:latest

WORKDIR /app

COPY --from=builder /app/myapp .

EXPOSE 8080

CMD ["./myapp"]

### 🧪 Build & Run the Image:

bash

docker build -t go-multistage-app .

docker run -p 8080:8080 go-multistage-app

## 🔎 Real-world Use Case of Multi-Stage Build

* **React + Node Backend:**  
  Stage 1: Build React frontend  
  Stage 2: Serve it using NGINX
* **Java App:**  
  Stage 1: Compile .java to .jar using Maven  
  Stage 2: Run the .jar using JRE

## 📝 Conclusion

In this assignment, you learned:

✅ What is a Docker Registry and DockerHub  
✅ How to pull/push images from/to DockerHub  
✅ What is a Multi-Stage Build and why it’s useful  
✅ Hands-on Dockerfile example with a Go app

# **Assignment 4: Creating Docker Images Using Multiple Methods (Dockerfile & Running Containers)**

## 🔹 Objective:

Understand and demonstrate how to **create Docker images** using:

1. **Dockerfile**
2. **Running Container (commit method)**

## ✅ 1. ****Creating Docker Image Using a Dockerfile****

The **Dockerfile** method is the most widely used and recommended approach.

### 🧱 What is a Dockerfile?

A Dockerfile is a script containing a list of instructions used to build a Docker image.

### 📁 Example Project Structure:

bash

/flask-app

├── app.py

└── Dockerfile

### 🐍 Example: Flask App (app.py)

python

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route("/")

def home():

return "Hello from Flask inside Docker!"

### 🧾 Dockerfile Example:

Dockerfile

# Use official Python base image

FROM python:3.9

# Set working directory

WORKDIR /app

# Copy application code

COPY . .

# Install dependencies

RUN pip install flask

# Expose app port

EXPOSE 5000

# Run the Flask app

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

### 🔨 Build the Docker Image:

bash

docker build -t flask-dockerfile-app .

### ▶️ Run the Container:

bash

docker run -p 5000:5000 flask-dockerfile-app

Visit: http://localhost:5000

## ✅ 2. ****Creating Docker Image from a Running Container****

Sometimes, you might manually configure a container (e.g., install software or change config), and want to **save its current state** as a new image.

### 🔁 Steps:

### 1. Run a Base Container (e.g., Ubuntu):

bash

docker run -it ubuntu

### 2. Install Software (inside container):

bash

apt-get update

apt-get install -y nginx

exit

### 3. View Container ID:

bash

docker ps -a

### 4. Commit Changes to New Image:

bash

docker commit <container\_id> my-nginx-image

### 5. Run Image:

bash

docker run -d -p 8080:80 my-nginx-image

Visit: http://localhost:8080

### 📊 Comparison

| **Feature** | **Dockerfile Method** | **Commit Method** |
| --- | --- | --- |
| **Reproducibility** | High – fully scripted | Low – manual steps |
| **Best Practice** | ✅ Yes | ❌ No (used for quick snapshots) |
| **Version Control Friendly** | ✅ Yes (can be committed) | ❌ No |
| **When to Use** | Production builds | Quick testing / temporary setup |

## 📝 Conclusion

In this task, we learned how to create Docker images using:

1. **Dockerfile** – recommended for structured, repeatable image creation
2. **Running container with commit** – quick and dirty method to capture live state

# **Assignment 5: Push and Pull Docker Images to DockerHub and Azure Container Registry (ACR)**

## 🔹 Objective:

Understand how to **push** (upload) and **pull** (download) Docker images to/from:

1. **DockerHub (Public Registry)**
2. **Azure Container Registry (Private Cloud Registry)**

## ✅ 1. ****Push and Pull Image with DockerHub****

### 🔧 Prerequisites:

* Docker installed
* DockerHub account created at https://hub.docker.com

### 🪪 Step 1: Login to DockerHub

bash

docker login

Enter your DockerHub **username** and **password/token**.

### 🔨 Step 2: Tag Your Image

Suppose you already built an image locally:

bash

docker build -t flask-app .

Now tag it for DockerHub:

bash

docker tag flask-app your\_dockerhub\_username/flask-app:latest

### 🚀 Step 3: Push Image to DockerHub

bash

docker push your\_dockerhub\_username/flask-app:latest

You’ll see layers being uploaded to your DockerHub repository.

### ⬇️ Step 4: Pull Image from DockerHub

From any other machine:

bash

docker pull your\_dockerhub\_username/flask-app:latest

### ▶️ Run the Pulled Image:

bash

docker run -p 5000:5000 your\_dockerhub\_username/flask-app

## ✅ 2. ****Push and Pull Image with Azure Container Registry (ACR)****

### 🔧 Prerequisites:

* Azure account
* Azure CLI installed
* Docker installed and logged in

### 🛠️ Step 1: Create an ACR (Azure Container Registry)

bash

az login

az acr create --resource-group <resource-group-name> --name <acrName> --sku Basic

Example:

bash

CopyEdit

az acr create --resource-group dev-rg --name mydevregistry --sku Basic

### 🔍 Step 2: Login to ACR

bash

az acr login --name mydevregistry

This logs Docker in to use the ACR.

### 🏷️ Step 3: Tag Docker Image for ACR

Get the full login server name:

bash

az acr list --resource-group dev-rg --query "[].{acrLoginServer:loginServer}" --output table

Example Output:

markdown

acrLoginServer

--------------------------

mydevregistry.azurecr.io

Then tag the image:

bash

docker tag flask-app mydevregistry.azurecr.io/flask-app:v1

### 🚀 Step 4: Push Image to ACR

bash

docker push mydevregistry.azurecr.io/flask-app:v1

### ⬇️ Step 5: Pull Image from ACR

bash

docker pull mydevregistry.azurecr.io/flask-app:v1

Then run it:

bash

docker run -p 5000:5000 mydevregistry.azurecr.io/flask-app:v1

## 📊 Summary Comparison

| **Operation** | **DockerHub (Public)** | **Azure Container Registry (ACR)** |
| --- | --- | --- |
| Authentication | DockerHub login | Azure CLI + Docker login |
| URL Format | username/image:tag | acrname.azurecr.io/image:tag |
| Access Control | Public or Private | Fully Private (RBAC Support) |
| Ideal For | Open Source, Personal Projects | Enterprise-level, secure environments |

## 📝 Conclusion

In this task, we learned:

* How to **tag**, **push**, and **pull** Docker images to/from **DockerHub**
* How to interact with **Azure Container Registry (ACR)** for enterprise deployment

# **Assignment 6: Create a Custom Docker Bridge Network**

## 🎯 ****Objective****

Learn how to create and use a **custom Docker bridge network** to enable **container-to-container communication** using container names.

## 🔍 ****What is a Docker Bridge Network?****

A **bridge network** in Docker is a **private internal network** where containers can **communicate with each other** by name if they're on the same bridge.

## ✅ 1. ****Default vs Custom Bridge Network****

| **Feature** | **Default Bridge (docker0)** | **Custom Bridge (user-defined)** |
| --- | --- | --- |
| DNS-based container name resolution | ❌ No | ✅ Yes |
| Isolation from other networks | ❌ Less | ✅ Better |
| Customizable | ❌ No | ✅ Yes |

## 🛠️ 2. ****Create a Custom Bridge Network****

bash

CopyEdit

docker network create --driver bridge my\_custom\_network

🔎 To verify:

bash

CopyEdit

docker network ls

## 📦 3. ****Run Containers on the Custom Network****

### 🧪 Example: Run two containers (web and db) and connect them

### Step 1: Run a simple web container (e.g., nginx)

bash

docker run -dit --name web --network my\_custom\_network nginx

### Step 2: Run a second container (e.g., Alpine Linux)

bash

docker run -dit --name debugger --network my\_custom\_network alpine

### Step 3: Access one container from the other

bash

docker exec -it debugger sh

ping web

✅ You'll see responses – proving both containers are connected by **name**.

## 🧱 4. ****Use Case: Flask App talking to Redis over Custom Bridge****

### Step 1: Create a Network

bash

CopyEdit

docker network create flask\_net

### Step 2: Start Redis Container

bash

CopyEdit

docker run -d --name redis --network flask\_net redis

### Step 3: Flask App Dockerfile

Dockerfile

FROM python:3.9

WORKDIR /app

COPY . .

RUN pip install flask redis

EXPOSE 5000

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

### app.py

python

from flask import Flask

import redis

app = Flask(\_\_name\_\_)

r = redis.Redis(host='redis', port=6379)

@app.route('/')

def hello():

r.incr('hits')

return f"Hello! You have visited this page {r.get('hits').decode()} times."

### Step 4: Build Flask Image

bash

docker build -t flask-redis-app .

### Step 5: Run Flask App on Same Network

bash

docker run -d --name flask-app --network flask\_net -p 5000:5000 flask-redis-app

Visit: http://localhost:5000 to test the app.

## 📤 5. Clean Up (Optional)

bash

docker stop flask-app redis

docker rm flask-app redis

docker network rm flask\_net

## 📝 ****Conclusion****

In this task, you:

✅ Created a custom Docker bridge network  
✅ Launched containers within it  
✅ Enabled inter-container communication using names  
✅ Used it in a real-world Flask + Redis microservice setup

# **Assignment 7: Create a Docker Volume and Mount it to a Container**

## 🎯 ****Objective****

Understand how to use **Docker volumes** for **persistent storage** and how to **mount** them into containers so data is retained even after container deletion.

## 🔍 ****What is a Docker Volume?****

A **Docker volume** is a storage mechanism managed by Docker that allows **data persistence** across container restarts and recreations.

### ✅ Why Use Volumes?

* Store databases and user data
* Share data between containers
* Keep container images lightweight
* Backup/restore container data

## 🧱 1. ****Create a Docker Volume****

bash

docker volume create my\_volume

🔎 To list all volumes:

bash

docker volume ls

🔍 To inspect volume details:

bash

docker volume inspect my\_volume

## 📦 2. ****Mount Volume to a Container****

Let’s use an **nginx** container to demonstrate volume mounting.

### 🔨 Step 1: Create and Mount Volume

bash

CopyEdit

docker run -d --name web-server \

-v my\_volume:/usr/share/nginx/html \

-p 8080:80 nginx

✅ This mounts my\_volume to the web root directory of nginx.

## ✏️ 3. ****Write Data to the Volume****

Let’s add a custom HTML file inside the volume.

### Step 1: Open a shell in the running container:

bash

docker exec -it web-server bash

### Step 2: Add a web page:

bash

echo "<h1>Hello from Docker Volume!</h1>" > /usr/share/nginx/html/index.html

exit

## 🌐 4. ****Access the Web Page****

Visit http://localhost:8080 in your browser.

✅ You should see:  
**“Hello from Docker Volume!”**

## 🔁 5. ****Test Persistence of Data****

### Step 1: Stop and remove container

bash

docker stop web-server

docker rm web-server

### Step 2: Run a new container with the ****same volume****:

bash

docker run -d --name new-web-server \

-v my\_volume:/usr/share/nginx/html \

-p 8080:80 nginx

✅ Visit http://localhost:8080 again — the HTML content is **still there!**

This proves the data was **persisted** outside the container in the volume.

## 📤 6. Clean Up (Optional)

bash

docker stop new-web-server

docker rm new-web-server

docker volume rm my\_volume

## 📊 Summary

| **Feature** | **Description** |
| --- | --- |
| Command to create volume | docker volume create my\_volume |
| Mount to container | -v my\_volume:/container/path |
| Persistent after container delete | ✅ Yes |
| Ideal for | DBs, Web content, config files, backups |

## 📝 ****Conclusion****

In this assignment, you learned:

✅ What Docker volumes are  
✅ How to create and mount them  
✅ How data persists even after container deletion  
✅ How to use volumes for real-world use cases like NGINX web pages

# Assignment 8: Docker Compose for Multi-Container Applications & Docker Security Best Practices

## 🧩 Part 1: Docker Compose for Multi-Container Applications

### 🎯 ****Objective:****

Learn how to use **Docker Compose** to define and run multiple containers for a complete application stack (e.g., web + database).

### 🧾 What is Docker Compose?

**Docker Compose** is a tool that allows you to define and run **multi-container applications** using a simple YAML file (docker-compose.yml).

### ✅ Use Case: Flask Web App + Redis Backend

### 📁 Project Structure:

flask-redis-app/

├── app.py

├── requirements.txt

└── docker-compose.yml

### 🔹 app.py (Flask App)

python

from flask import Flask

import redis

app = Flask(\_\_name\_\_)

cache = redis.Redis(host='redis', port=6379)

@app.route('/')

def index():

count = cache.incr('hits')

return f'Hello! This page has been viewed {count} times.'

### 🔹 requirements.txt

nginx

CopyEdit

flask

redis

### 🔹 docker-compose.yml

yaml

version: '3.8'

services:

web:

build: .

ports:

- "5000:5000"

depends\_on:

- redis

redis:

image: redis:alpine

### 🔹 Dockerfile

FROM python:3.9

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

EXPOSE 5000

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

### 🚀 Steps to Run:

1. **Navigate to project folder**

bash

cd flask-redis-app

1. **Start the containers**

bash

docker-compose up --build

1. **Access the app** at http://localhost:5000
2. **Stop the services**

bash

docker-compose down

### ✅ Benefits of Docker Compose:

* Centralized service definition
* Easy to manage complex multi-container apps
* Volumes, networks, and environment configs handled easily

## 🛡️ Part 2: Docker Security Best Practices

### 🔐 Why Docker Security Matters

Containers share the host kernel, so one compromised container could lead to host compromise. You must follow best practices.

### 🧱 1. Use Official and Trusted Images

✅ Always pull images from verified sources like:

bash

docker pull nginx

Avoid using unknown or unverified public images.

### 🔍 2. Scan Images for Vulnerabilities

Use tools like:

* docker scan (built-in)
* **Trivy**: trivy image myapp
* **Clair**, **Anchore**

### 🔒 3. Limit Container Privileges

Avoid running containers as root unless absolutely necessary.

Dockerfile

USER nonrootuser

Or run with --user flag:

bash

docker run --user 1001 myimage

### 🔐 4. Use Docker Secrets for Sensitive Data

Instead of environment variables:

bash

docker secret create db\_pass ./password.txt

Then use docker service with secrets in Swarm mode.

### 🧱 5. Use Read-Only File Systems

Limit file access using:

bash

docker run --read-only ...

### 🔄 6. Keep Docker & Dependencies Updated

Always update:

* Docker Engine
* Host OS packages
* Base images

bash

apt update && apt upgrade

docker pull python:3.9

### 🔍 7. Enable Logging and Auditing

Use:

* docker logs
* Integration with syslog, Fluentd, ELK

Track container behavior.

### 🔥 8. Use Resource Limits

Prevent container abuse with:

yaml

deploy:

resources:

limits:

cpus: "0.5"

memory: "512M"

### 📜 9. Avoid Hardcoded Secrets

Do **not** do this:

yaml

environment:

- MYSQL\_PASSWORD=supersecret

Instead, use .env files or Docker secrets.

### 🧱 10. Isolate Containers with Custom Networks

Use docker network create and Docker Compose networks to control traffic flow.

## 📝 Conclusion

In this assignment, you learned:

✅ How to use **Docker Compose** to deploy multi-container apps  
✅ How to build a Flask + Redis app using docker-compose.yml  
✅ **10+ Docker security best practices** to secure containers and hosts

## 📁 Optional: Clean Up Resources

bash

docker-compose down --volumes

docker network prune

docker image prune