BCIT, COMP 4912 - LAB #2

Winter 2025

1. Objectives

At the end of this lab, you will learn how to create a simple multi-container online service via Docker Compose. In addition, you will understand features of LXC (Linux Containers) and LXD by working with a lightweight image.

2. Tasks

The lab consists of two major tasks, which are as follows:

2.1. Deploy a Simple Online Service with Docker Compose

In this assignment, you will create and deploy a simple multi-container online service using Docker Compose. You will:

- Write a Dockerfile for a simple web application (Python/Flask).
- Use an official database image (e.g., Postgres or MySQL).
- Define services in a **docker-compose.yml** file so both containers can communicate.
- Verify your application is up and running locally.

By completing this assignment, you will learn how to manage and deploy multiple containers using a single YAML configuration file. Indeed, you create a simple Python/Flask application and Dockerize it with a Dockerfile. Then you will Orchestrate multiple containers (web application + database) using Docker Compose. Using environment variables and port mappings in Docker Compose, you make the application accessible from your web browser and finally you will run and verify it.

2.1.1. Create a folder for your project, for example:

```
$ mkdir ~/docker-compose-assignment
$ cd docker-compose-assignment
```

2.1.2. Inside the folder, create two files and one app folder.

- o Dockerfile (for your web application)
- o docker-compose.yml (for composing your containers)

Your final structure might look like this:

```
docker-compose-assignment/
├── app/
│   ├── app.py
│   └── requirements.txt
├── Dockerfile
└── docker-compose.yml
```

Note: We'll use app/ subdirectory to store the application code.

2.1.3. Write a simple Flask Web Application

Inside the app folder, create a file called **app.py** with the following content:

```
from flask import Flask
import os

app = Flask(__name__)

@app.route("/")
def hello():
   return "Hello from Flask! Database is at host: " +
os.environ.get("DB_HOST", "not set")

if __name__ == "__main__":
   # Run on port 5000
   app.run(host="0.0.0.0", port=5000, debug=True)
```

Also create **requirements.txt** file (in the same app folder) to specify Flask:

```
Flask==2.3.2
```

2.1.4. Create a Dockerfile

The Dockerfile describes how to build an image for your Flask web application. Create a file named **Dockerfile** (at the root of your project, not inside app folder created at step 2.1.1) with the following content:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy the requirements file
COPY app/requirements.txt ./

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code
COPY app/ .

# Expose port 5000 for the Flask app
EXPOSE 5000

# Set environment variable for database host
ENV DB_HOST=database

# Define the command to run the app
CMD ["python", "app.py"]
```

Explanation of the items used in the Dockerfile is follows:

- **FROM python:3.9-slim**
  Uses a lightweight Python 3.9 image.
- **WORKDIR /usr/src/app**
  Sets the working directory in the container.
- **COPY app/requirements.txt ./**
  Copies requirements.txt into the container so we can install dependencies.
- **RUN pip install --no-cache-dir -r requirements.txt**
  Installs required Python packages.
- **COPY app/ .**
  Copies your Flask application files into the container.
- **EXPOSE 5000**
  Exposes port 5000, the default port your Flask app will run on.
- **ENV DB_HOST=database**
  Sets an environment variable that tells our app that the database host is named database.
- **CMD ["python", "app.py"]**
  Command to run the Flask application.

2.1.5. Create the docker-compose.yml file

Now we'll define our multi-container setup (one for the Flask app and one for a database) in a file called **docker-compose.yml.**

```
version: '3.8'

services:
  web:
    build: .
    container_name: flask_app
    ports:
      - "5000:5000"
    depends_on:
      - database
    environment:
      - DB_HOST=database

  database:
    image: postgres:14-alpine
    container_name: postgres_db
    environment:
      - POSTGRES_USER=myuser
      - POSTGRES_PASSWORD=mypassword
      - POSTGRES_DB=mydatabase
    ports:
      - "5432:5432"
```

Explanation of the items used in the docker-compose.yml file is follows:

- **version: '3.8'**
  Specifies which Docker Compose file format version we're using.
- **services:**
  We define two services, web and database.
- **web** service:
  - **build: .** uses Dockerfile in current directory to build the image.
  - **container_name: flask_app** names the container (optional but helpful).
  - **ports:** 5000:5000 maps host port 5000 to container port 5000.
  - **depends_on:** ensures that database starts before web.
  - **environment:** passes the DB_HOST=database variable to the container.
- **database** service:
  - **image: postgres:14-alpine** pulls the lightweight PostgreSQL 14 Alpine image.
  - **container_name: postgres_db** names the container.

    - **environment:** sets up user, password, and database.
    - **ports:** 5432:5432 to allow external connections.

2.1.6. Running your services

Go to the docker-compose-assignment folder (where docker-compose.yml is located) and run the following command to build/start your containers:

```
$ docker-compose up -d --build
```

In this command, --build flag ensures the images are rebuilt if Dockerfile has changed and runs in detached mode (-d). Docker Compose will:

- Pull the postgres:14-alpine image (if not on your machine).
- Build the web image from your Dockerfile.
- Start both containers (flask_app and postgres_db).
- Display logs for each container in real time.

Before running this command, ensure docker-compose has been installed:

```
$ apt update
$ apt install docker-compose
```

Open the browser and navigate to **http://localhost:5000**. You should see:

Hello from Flask! Database is at host: database

This confirms your Flask app is running and reading the environment variable for the database host.

2.1.7. Verifying and Inspection

- Check container status by running ‘docker-compose ps’ command, which shows the status of your web and database containers.
- Check logs (if running detached) for your Flask app and the database:

```
$ docker-compose logs web
$ docker-compose logs database
```

- Stop containers by running ‘docker-compose down’ command. This stops and removes the containers. By replacing down with up, you can bring up the container again.

2.2. Exploring LXC/LXD features

The purpose of this task is to help students understand and experiment with the key features of LXC (Linux Containers) and LXD (the container management tool) by working with a lightweight image such as Alpine Linux. Students will deploy, configure, and explore container functionalities while gaining hands-on experience with LXD commands and container lifecycle management.

2.2.1. Setting up LXD:

- Install LXD on your Ubuntu by running ‘snap install lxd’ command.
- Initialize LXD with default settings by ‘sudo lxd init --auto’.
- Verify that LXD is running by running ‘lxc list’ command.
- Launch a container using the Alpine Linux image. Then, verify it is running and get further info using ‘lxc list’ and ‘lxc info’ commands.

```
$ lxc launch images:alpine/edge alpine-test
$ lxc list
$ lxc info alpine-test
```

- If you are using Ubuntu on WSL, you can check the size of the running Alpine container by running the following command:

```
$ du -sh /var/snap/lxd/common/lxd/storage-pools/default/containers/alpine-test
```

2.2.2. Exploring Container Functionality:

- Enter the Alpine container shell by ‘lxc exec alpine-test -- /bin/sh’
- View the operating system version by running:

```
$ cat /etc/os-release
```

- Check the available memory and disk by running:

```
$ free -m
$ df -h
```

- Install curl package using apk command on the Alpine container:

```
$ apk add curl
```

- Now, you can check your public IP using curl by running:

```
$ curl ipinfo.io
```

- Install Apache2 web server and start it on the container by:

```
$ apk add apache2
$ rc-service apache2 start
```

- If the service is running successfully, you should see running processes in the output of '`ps -A | grep httpd`' command.
- Open the following index file using vi editor and replace "**it works!**" string with "**This is my running web server!!!**".

```
$ vi /var/www/localhost/htdocs/index.html
```

  Once it is done, exit from the container shell by **exit** command.

2.2.3. Configure port forwarding on the host machine:

- To allows external access to the container via port forwarding, you need to run the following command on the host machine:

```
$ lxc config device add alpine-test myport80 proxy listen=tcp:0.0.0.0:7070 connect=tcp:127.0.0.1:80
```

- The previous command adds a proxy device named "myport80" to the "alpine-test" container, which listens for incoming TCP connections on the host at port 7070 and forwards them to the container's localhost address on port 80. Note that the Apache2 package you installed on the container is listening on port 80.
- To verify the port forwarding is configured correctly, find the IP address of the host machine that your container is running on it. It can be done using the '`ip add show`' command. If you see multiple IP addresses in the output of this command, pick the one assigned to the **eth0** interface. Now, to test the port forwarding using a web browser, navigate **http://<host-ip>:7070**.
- If everything has been configured properly, you should be able to see **"This is my running web server!!!"** on your browser.

2.2.4. Manage container resources:

- To limit the CPU to 1 core and memory usage of the container run:

```
$ lxc config set alpine-test limits.cpu 1
$ lxc config set alpine-test limits.memory 64MB
```

- To verify the new resource limits, you can run the following command and try to find the new values in the output:

```
$ lxc config show alpine-test
```

2.2.5. Taking snapshots from the container:

- Create a snapshot (backup) from the container by running:

```
$ lxc snapshot alpine-test snap1
$ lxc info alpine-test
```

- Now login to the container and make a change by overwriting the content of /etc/os-release file. As you have already checked, this file contains the version of the OS which is Alpine Linux. To do this, run the following commands:

```
$ lxc exec alpine-test -- /bin/sh
$ echo "ABCD" > /etc/os-release
```

- After changing this file, you can check the content of the file again using '`cat /etc/os-release`' command and exit the container.
- Now, by running '`lxc restore alpine-test snap1`' you will restore the container to the snapshot 'snap1' that you took before overwriting the content of /etc/os-release file.
- Login to the container again and check/record the content of /etc/os-release file after restoring the snapshot.

3. Expected Outcome

3.1. Individual Report Requirement

Each student must submit an individual report for every lab. Collaboration on reports is not allowed, but you can discuss concepts with classmates.

3.2. Report Content

- **Screenshots**: Include screenshots of each step, including the commands you used and the corresponding output.
- **Challenges and Solutions**: If you encounter challenges while following the instructions on your computer, describe it in your report along with the solutions you applied to resolve them.

3.3. Organization and Quality

- Divide your report into sections corresponding to each task in the lab.
- At the end of your report, write a summary highlighting the tasks completed and your observations.
- Ensure your report is clear, well-written, and well-organized.

3.4. Deadline

Submit your report before the deadline. Late submissions may not be accepted unless prior arrangements are made.

3.5. Troubleshooting Guidance

- If you face an issue, start by researching solutions using Google, community forums, or tools like ChatGPT.
- Take time to investigate and understand the problem thoroughly.
- If you cannot resolve the issue, share your question along with a detailed explanation of what you've already tried with your classmates and me.
- Remember, It is very important to attempt some independent research before asking for help with any project-related questions.

3.6. Tips

- Use nano editor to create configuration files and copy/paste the content.
- If you faced any error during building the container due to cancelling a previous command/job, you may try two first following commands first and then compose the docker image again.

```
$ cd ~/docker-compose-assignment
$ docker-compose down --volumes --rmi all
$ docker system prune -af
$ docker-compose up -d --build
```