SLIIT | DEPARTMENT OF COMPUTER SCIENCE & SOFTWARE ENGINEERING | FACULTY OF COMPUTING

Module - Current Trends in Software Engineering (SE4010) | 2025 | Semester 1

DevOps Lab – 1

Getting Started with Docker

- **Task:** Pull and run the official nginx image from Docker Hub.
- **Steps:**
  1. Open a terminal or command prompt.
  2. Run docker pull nginx to pull the latest nginx image.
  3. Run docker run -d -p 8080:80 nginx to start an Nginx server.
  4. Open a web browser and go to http://localhost:8080 to see the Nginx welcome page.
- **Goal:** Familiarize with pulling and running Docker images.

Containerizing a Simple Application

- **Task**: Create a Dockerfile for a simple "Hello World" application in any language of your choice (Node JS/ Python etc…). Sample repository for reference can be accessed at - https://github.com/rav94/node-docker-demo
- **Steps**:
  1. Navigate to the directory with the provided application (e.g., **app.py/ index.js**).
  2. Create a Dockerfile in the same directory.
  3. In the Dockerfile, start from a base image of chosen language, copy the application file/s, and set the command to run the application.
  4. Build the Docker image: **docker build -t hello-world-app .**
  5. Run the container: **docker run -d -p <host-port>:<container-port> hello-world-app**.
  6. Verify the application is running by accessing **http://localhost:<host-port>**.
  7. Create a DockerHub (https://hub.docker.com/signup) account and login to it via terminal - **docker login**
  8. Tag your image <your-dockerhub-username>/hello-world-app
  9. Push the image to DockerHub and view the image in DockerHub – docker push - **<your-dockerhub-username>/hello-world-app**
- **Goal**: Learn Dockerfile basics and build a custom Docker image.

Using Docker Volumes

1. **Objective**: Persist data using Docker-managed volumes.
2. **Explanation**: Volumes are stored in a part of the host filesystem managed by Docker (**/var/lib/docker/volumes/** on Linux). They are completely managed by Docker and are isolated from the host filesystem.
3. **Steps**:
   - Create a Docker volume: **docker volume create hello-world-volume**.

- Run the container with the volume attached for data storage: **docker run -d -p <host-port>:<container-port> -v hello-world-volume:/app/data hello-world-app**.
- Data written by the application to **/app/data (within container)** will persist across container restarts and can be accessed by other containers using the same volume.

Using Bind Mounts

1. **Objective**: Persist data using bind mounts.
2. **Explanation**: Bind mounts link a specific directory or file on the host to a directory or file in the container. They allow direct access to the host's filesystem and are suitable for development purposes.
3. **Steps**:
    - Choose a directory on the host machine (e.g., **/path/to/host/data**).
    - Run the container with the bind mount: **docker run -d -p <host-port>:<container-port> -v /path/to/host/data:/app/data hello-world-app**.
    - Data written by the application to **/app/data (within container)** will be directly visible in **/path/to/host/data** on the host machine and vice versa.
    - Challenge: docker exec into the created container and create a directory inside the given path. Observe the directory getting created on host within /path/to/host/data

Using Docker Compose

- **Task**: Run a multi-container application with a web server and database.
- **Steps**:
    1. Provide a **docker-compose.yml** file defining a simple web app and a database service (e.g., Node app with MongoDB). Sample repository for reference can be accessed at - https://github.com/rav94/node-mongodb-docker-compose
    2. Navigate to the directory containing **docker-compose.yml**.
    3. Run **docker-compose up** to start both services.
    4. Access the web application to ensure it's interacting with the database.
    5. View the created volumes for database service – **docker volume ls**
    6. View the created networks for the application deployed by docker-compose – **docker network ls**
    7. Stop docker compose – **docker compose down**
    8. Ensure that volume is not deleted as its managed by Docker and persisted – **docker volume ls**
- **Goal**: Experience managing multi-container applications with Docker Compose.

Understanding Docker Layers

- Create a simple python application that can be containerized.
- Create a file named app.py:

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route("/")
def hello():
    return "Hello, Docker!"

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

- Create file named requirements.txt and add following
  - flask==2.2.2

- Create a file named Dockerfile.naive:

```
# Naive Dockerfile
FROM python:3.9-slim

# Copy all files
COPY . /app
WORKDIR /app

# Install requirements
RUN pip install -r requirements.txt

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

- Build and Run the Container

```
# Build image from the naive Dockerfile
docker build -t naive-flask-app -f Dockerfile.naive .
docker run -p 5000:5000 naive-flask-app
```

- Examine the Layers
  - Run docker history naive-flask-app to see each layer of the created image.
  - Notice that each command (FROM, COPY, RUN pip install, etc.) appears as a separate layer.

- Observation:
  - Every time you change *any* file in your code, or add new dependencies, Docker has to rebuild the entire image, including the RUN pip install step.
  - The COPY step that copies your entire source directory (including requirements.txt) into the image occurs *before* pip installs dependencies, causing Docker to rebuild the dependency layer whenever *any* file changes.

- Optimize the Dockerfile

  The goal now is to re-order and reduce the steps that are frequently invalidated. We do so by copying only requirements.txt first, installing dependencies, and then copying the rest of the application. This way, any change to the code (except for changes to dependencies) will not invalidate the pip install layer.

- Create a file named Dockerfile.optimized:

```
# Optimized Dockerfile
FROM python:3.9-slim

# Set the working directory
WORKDIR /app

# Copy requirements.txt first (cache-friendly)
COPY requirements.txt requirements.txt

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Now copy the rest of the code
COPY . /app

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

- Build and Run the Optimized Container

```
docker build -t optimized-flask-app -f Dockerfile.optimized .
docker run -p 5000:5000 optimized-flask-app
```

- Compare and Contrast

  - Run docker history optimized-flask-app and compare the layers to naive-flask-app.
  - Make a small change in app.py (e.g., modify the "Hello, Docker!" message) and rebuild both images:
    - **Naive**: Notice that the pip install step runs again.
    - **Optimized**: You should see that pip install step is *not* re-run because requirements.txt didn't change, so Docker reuses the cached layer.