LAB ACTION PLAN FOR WEEK 6

Objectives:

Students will able to:

- Learn how to define and run multiple interdependent services (e.g., web server, database) in a single configuration file.
- Gain skills in writing and interpreting docker-compose.yml files for service setup.
- Deploy the same setup across different machines without manual configuration.
- Configure container networking and persistent storage within Compose.
- Reduce setup time and enable faster iteration during application development.

Students will perform Hands-on running multi container applications using docker compose and make document that include the execution of each step along with scenario-based question solutions after completing the tasks.

What is Docker Compose?

Docker Compose is a tool used to define and run multi-container Docker applications. It allows you to define services, networks, and volumes that your application needs, all in a single file. This makes it easier to manage complex applications that require multiple containers (e.g., a web server and a database).

Structure of a Docker Compose File

A Docker Compose file is written in YAML (YAML Ain't Markup Language), a human-readable data format used for configuration. In the Docker context, it helps define the services, networks, and volumes needed for an application.

Basic Components of a Docker Compose YAML File

1. **Version**: Specifies which version of the Docker Compose file format you're using. Each version comes with its own set of features.

   ```
   version: '3.8'
   ```

2. **Services**: This section defines all the containers you want to run for your application. Each service represents a container (like a web server or database).

   Each service includes:

   - **Image**: The Docker image to use.
   - **Ports**: Ports to map between the container and your host machine.
   - **Environment**: Environment variables required by the service.
   - **Dependencies**: Which other services a container depends on.

Example:

```
services:
  web:
    image: nginx
    ports:
      - "8060:80"
  db:
    image: tomee
    ports:
      - "8050:8080"
```

3. **Networks**: Define networks to allow different services to communicate with each other. Docker Compose automatically creates a default network if not specified.

4. **Volumes**: Used for persistent storage, allowing data to persist even after the container stops or is removed.

Example Docker Compose File (Simple)

Here's a basic example of a Docker Compose file that runs WordPress and MySQL together:

```
version: '3.8'  # Docker Compose file format version

services:
  wordpress:  # WordPress service
    image: wordpress:latest
    ports:
      - "8080:80"  # Map port 80 of the container to port 8080 of the host
    environment:
      WORDPRESS_DB_HOST: db:3306  # Database host
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
      WORDPRESS_DB_NAME: wordpress
    depends_on:
      - db  # Ensures the db service starts first

  db:  # MySQL service
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress
```

How to Run It

To run a multi-container setup like the one above:

1. **Save the file as docker-compose.yml.**

   Or

   docker-compose.yaml

2. **To Start the compose**

   ```
   docker-compose up –d
   ```

3. **To stop the containers**

   ```
   docker-compose down
   ```

4. **To scale the container**

   ```
   docker-compose up --scale <service name>=2 -d
   ```

1.Define and run multiple interdependent services

Task:

I. Create a new folder compose-lab

Inside it, create a file docker-compose.yml with the following content:

```
version: "3.9"
services:
  web:
```

```
    image: nginx:latest
    ports:
      - "8080:80"

  db:
    image: postgres:15
    environment:
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: demo
      POSTGRES_DB: demo_db
```

II. Run the setup:

```
docker compose up -d
```

III. Open your browser and visit: http://localhost:8080.

IV. Expected Output:

Nginx welcome page is displayed.

db container runs in the background.

2.Write and interpret docker-compose.yml files

Task:

I. Modify docker-compose.yml to add a Redis cache:

```
  redis:
    image: redis:alpine
```

II. Add a depends_on so web waits for Redis:

```
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    depends_on:
      - redis
```

III. Restart the setup:

```
docker compose up -d
docker compose ps
```

IV. Expected Output:

Three services (web, db, redis) are listed as running.

3.Deploy across different machines

Task:

I. Zip your compose-lab folder.

Transfer it to another machine with Docker Compose installed.

II. Run:

```
docker compose up -d
```

Check that Nginx and Postgres work there as well.

III. Expected Output:

The same services run on the new machine without changes.

4.Networking and persistent storage

Task:

I. **Update your docker-compose.yml to add a custom network and volume:**

```
networks:
  app-net:

volumes:
  db-data:

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    networks:
      - app-net
    depends_on:
      - db

  db:
    image: postgres:15
    environment:
      POSTGRES_USER: demo
      POSTGRES_PASSWORD: demo
      POSTGRES_DB: demo_db
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - app-net
```

II. Run:

```
docker compose up -d
```

III. Insert some data into Postgres (optional with psql).

IV. Remove containers:

```
docker compose down
```

V. Start again:

```
docker compose up -d
```

VI. Expected Output:

Database data persists across restarts.

Services communicate via the app-net network using service names.

5.Faster iteration during development

Task:

I. Create a simple Flask app in app.py:

```
from flask import Flask
app = Flask(__name__)
@app.route("/")
def home():
    return "Hello from Flask + Docker!"
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

II. Add a Dockerfile in the same folder:

```
FROM python:3.10-slim
WORKDIR /app
COPY app.py /app/
RUN pip install flask
CMD ["python", "app.py"]
```

III. Update docker-compose.yml:

```
services:
web:
  build: .
  ports:
    - "5000:5000"
  depends_on:
    - db


db:
  image: postgres:13
  environment:
    POSTGRES_USER: user
    POSTGRES_PASSWORD: password
    POSTGRES_DB: mydb
```

IV. Run:

```
docker compose up --build
```

Visit http://localhost:5000.

Change the return text in app.py (e.g., "Hello Docker Compose!").

V. Rebuild:

```
docker compose up --build
```

VI. Expected Output:

New message appears instantly after rebuild.

Scenario based Questions:

1. You have two applications — a Node.js backend and a Python script that handles scheduled jobs. You're considering running both inside the same container. Should you run multiple applications in the same Docker container?

2. You have a Flask API and an Nginx server. You want to run both containers on the same host and expose them on ports 5000 and 80 respectively. How can you expose both applications from different containers?

3. You want to run a React frontend, an Express.js backend and MongoDB – all together.

How do you run and manage all three together?

4. You try to run two containers that both expose port 8080 on the host.

What happens, and how can you run both apps?

5. You updated some code and want to restart your frontend, backend, and DB containers quickly. What is the easiest way to restart all services?

6. Your frontend has a new release, but you don’t want to restart the backend or DB.
How can you update and rebuild only the frontend container?

7. Tomcat is listening on port 8080, but you get a "Connection Refused" error in the browser. Why?

8. You're not sure which container is using port 3000. How do you check?

9. How do you stop and clean up your running Tomcat container and image?

10.You want to share your app with a teammate. What do you do with the Docker image?

Conclusion:

Docker Compose simplifies multi-container application management by allowing developers to define, configure, and run services in a single YAML file. It streamlines deployment, ensures consistent environments, and reduces manual setup, making it an essential tool for orchestrating complex, container-based applications efficiently.