ASSIGNMENT CLASS-17

Title

Docker Assignment Report

SUBMITTED ON

30/05/2025

Docker Assignment Report

This report details the completion of Docker-related tasks, including practical examples for running Docker containers, creating Dockerfiles for Flask and Node.js applications, and setting up Docker Compose configurations for single and multi-container applications. The tasks are presented in a structured format, including commands, code, and explanations.

Practical Example 1: Basic Docker Commands

Objective

Run basic Docker commands to pull and execute container images on an AWS server.

Example 1: Hello-World Container

Steps and Commands:

1. **Install Docker on AWS Server**
   Install Docker on an Ubuntu-based AWS server using the following commands:
   - sudo apt update
   - sudo apt install -y docker.io
   - sudo systemctl start docker
   - sudo systemctl enable docker
2. **Pull the hello-world image**
   - sudo docker pull hello-world
3. **List running containers**
   - sudo docker ps
4. **List Docker images**
   - sudo docker images
5. **Run the hello-world container**
   - sudo docker run hello-world

Explanation:

- The hello-world image is a lightweight test image that outputs a confirmation message when run.

- docker ps shows running containers (none initially).
- docker images lists available images, including hello-world after pulling.

Example 2: Nginx Container

Steps and Commands:

1. **Pull the nginx image**
    - sudo docker pull nginx
2. **List running containers**
    - sudo docker ps
3. **List Docker images**
    - sudo docker images
4. **Run the nginx container**
    - sudo docker run -p 5000:80 nginx

Explanation:

- The nginx image runs a web server.
- The -p 5000:80 flag maps port 5000 on the host to port 80 in the container, making the Nginx server accessible at http://<my-ip>:5000.

Practical Example 2: Flask Application

Objective

Create and run a Flask application in a Docker container.

Files

1. **app.py**

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

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

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

2. **requirements.txt**

   flask

3. **Dockerfile**

```
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 8000
CMD ["python", "app.py"]
```

Steps and Commands

1. **Create project directory**
   - mkdir app-python
   - cd app-python
2. **Create Dockerfile**
   - sudo vi Dockerfile
   - Paste the Dockerfile content above.
3. **Build the Docker image**
   - sudo docker build -t python_app .
4. **List Docker images**
   - sudo docker images
5. **Run the Flask container**
   - sudo docker run -p 8000:8000 python_app
6. **List running containers**
   - sudo docker ps

Explanation:

- The Flask application serves a simple message at http://<server-ip>:8000.
- The Dockerfile uses a slim Python 3.11 image, installs dependencies, and runs the Flask app.
- Port 8000 is exposed and mapped to the host for access.

Practical Example 3: Node.js Application

Objective

Create and run a Node.js Express application in a Docker container.

Files

1. **index.js**

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello from Node.js Express!');
});

app.listen(3000, () => {
    console.log('Node app listening on port 3000');
});
```

2. **package.json**

```
{
    "name": "node-app",
    "version": "1.0.0",
    "main": "index.js",
    "scripts": {
        "start": "node index.js"
    },
    "dependencies": {
        "express": "^4.18.2"
    }
}
```

3. **Dockerfile**

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 3000
CMD ["node", "index.js"]
```

Steps and Commands

1. **Create project directory**
    - mkdir node-app
    - cd node-app

2. **Create Dockerfile**
    - sudo vi Dockerfile
    - Paste the Dockerfile content above.
3. **Build the Docker image**
4. sudo docker build -t node_app .
5. **List running containers**
6. sudo docker ps
7. **List Docker images**
8. sudo docker images
9. **Run the Node.js container**
10. sudo docker run -p 3000:3000 node_app

Explanation:

- The Node.js application uses Express to serve a message at http://<server-ip>:3000.
- The Dockerfile uses a lightweight Node.js 18 Alpine image, installs dependencies, and runs the app.
- Port 3000 is exposed and mapped to the host.

Docker Compose Tasks

Practical Example 1: Single Service (Flask)

docker-compose.yml

```
version: "3.8"
services:
  python-app:
    build: .
    container_name: myfirst_app
    ports:
      - "8000:8000"
```

Explanation:

- Defines a single service (python-app) built from the current directory's Dockerfile.
- Maps port 8000 on the host to 8000 in the container.
- Assigns a container name for easy reference.

Practical Example 2: Multi-Container (Flask + Node.js)

docker-compose.yml

```
version: "3.8"

services:

  python-app:

    build:

      context: ./app-python

    ports:

      - "8000:8000"

  node-app:

    build:

      context: ./node-app

    ports:

      - "3000:3000"
```

Commands to Manage Containers:

1. **Start containers**
2. docker-compose up

Add -d for background mode:

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

3. **Stop and remove containers**
4. docker-compose down
5. **View logs**
6. docker-compose logs

Explanation:

- Defines two services: python-app and node-app, built from their respective directories.
- Maps ports 8000 and 3000 to their respective containers.
- Allows simultaneous management of both applications.

Task 1: Simple Docker Compose File

Objective: Create a basic docker-compose.yml file for an Nginx service.

docker-compose.yml

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "8080:80"
```

Explanation:

- Uses Docker Compose version 3.
- Defines a web service using the nginx image.
- Maps port 8080 on the host to port 80 in the container, making Nginx accessible at http://<server-ip>:8080.

Task 2: Multi-Container Application (Web + DB)

Objective: Create a docker-compose.yml file for an Nginx web server and a MySQL database.

docker-compose.yml

```
version: '3'
services:
  web:
    image: nginx
    ports:
      - "8080:80"
  db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: root123
```

Explanation:

- Defines two services: web (Nginx) and db (MySQL).
- Nginx is accessible at http://<server-ip>:8080.
- MySQL is configured with a root password (root123) via an environment variable.
- No port mapping is specified for MySQL, as it's typically accessed internally or via tools like mysql client.

Conclusion

This assignment demonstrated the setup and execution of Docker containers for various applications:

- **Basic Docker Commands**: Ran hello-world and nginx containers.
- **Flask Application**: Built and ran a Python Flask app in a container.
- **Node.js Application**: Built and ran a Node.js Express app in a container.
- **Docker Compose**: Configured single and multi-container setups for Flask, Node.js, Nginx, and MySQL.

All tasks were completed with proper file configurations and commands, ensuring functionality on an AWS server.