University of Monastir
Higher Institute of Computer Science
and Mathematics of Monastir

Subject: DevOps
Study Level: ING2 Computer Science
Academic Year: 2024/2025

TP3

Docker Basics Lab: Containerizing a Simple Application

1 Objectives

By the end of this lab, you will be able to:

- Install and run Docker (locally or via an online playground such as Play with Docker Classroom).
- Pull images from Docker Hub and run containers.
- Write a Dockerfile to containerize a basic web application.
- Build and run a custom Docker image with port mapping.
- Use Docker volumes for persistent data storage.
- Create and manage a multi-container application using Docker Compose.
- **Understand how to integrate Docker images into CI/CD pipelines** for automated build, test, and deployment.

2 Prerequisites

- Basic command line (UNIX/Linux) skills.
- Docker installed on your machine or access to an online Docker playground (e.g., Play with Docker Classroom).
- A text editor (e.g., VS Code).

3 Lab Tasks

3.1 Task 1: Running a Simple Container

1. **Pull and Run an Official Image:**
   Open a terminal and run:

```
docker run --rm hello-world
```

   Expected Outcome: The container prints a ”Hello from Docker!” message, confirming that Docker is set up correctly.

2. **List Running Containers:**
   Run:

```
docker ps -a
```

   Discussion: Explain the difference between running containers and those that have exited.

3.2 Task 2: Building a Custom Docker Image

1. **Prepare a Simple Web Application:**
   Create a file named `index.html` in your working directory with the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Docker Web App</title>
  </head>
  <body>
```

```
    <h1>Hello, Docker!</h1>
  </body>
</html>
```

2. **Write a Dockerfile:**
   In the same directory, create a file named `Dockerfile` with the following content:

```
# Use the official Nginx image as the base image
FROM nginx:alpine

# Copy the custom web page into the Nginx default directory
COPY index.html /usr/share/nginx/html/index.html

# Expose port 80 for the web server
EXPOSE 80
```

3. **Build the Docker Image:**
   Run the following command:

```
docker build -t my-docker-webapp .
```

   Expected Outcome: Docker builds the image and tags it as `my-docker-webapp`.

4. **Run the Custom Image with Port Mapping:**
   Execute:

```
docker run -d -p 8080:80 --name webapp my-docker-webapp
```

   Verification: Open your browser and navigate to `http://localhost:8080` to view the ”Hello, Docker!” message.

3.3 Task 3: Working with Docker Volumes

1. **Run a Container with a Volume:**
   Execute:

```
docker run -it -v "$(pwd)/data":/data alpine sh
```

Activity: Inside the container, run:

```
echo "Persistent␣Data" > /data/info.txt
exit
```

Verification: Check the `data` folder in your working directory to ensure that `info.txt` exists.

2. **Discussion:**
   Docker volumes allow data to persist outside the container lifecycle.

3.4 Task 4: Multi-Container Application with Docker Compose

1. **Create a `docker-compose.yml` File:**
   In your project directory, create a file named `docker-compose.yml` with the following content:

```
version: ’3’
services:
  web:
    image: nginx:alpine
    ports:
      - "8081:80"
    volumes:
      - ./web-content:/usr/share/nginx/html
  redis:
    image: redis:alpine
    ports:
      - "6379:6379"
```

Note: Create a directory named `web-content` and add an `index.html` file if you wish to customize the web server content.

2. **Run Docker Compose:**
   Start the services with:

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

Verification: Check the running containers:

```
docker-compose ps
```

Then, navigate to `http://localhost:8081` in your browser.

3. **Clean Up:**
   Stop the services by running:

```
docker-compose down
```

3.5 Task 5: CI/CD Integration (Optional/Extension)

Overview:
In modern DevOps workflows, Docker images are the artifact that is built, tested, and deployed via a CI/CD pipeline. The pipeline is typically divided into three phases:

- **Build Phase:** On every code commit, a CI system builds a Docker image from the Dockerfile.
- **Test Phase:** The built image is deployed to a test/staging environment where automated tests are executed.
- **Deployment Phase:** If tests pass, the image is pushed to a Docker registry (e.g., Docker Hub) and later deployed to production using orchestration tools such as Kubernetes or Docker Swarm.

Example: GitHub Actions CI/CD Pipeline
Create a file at `.github/workflows/ci-cd.yml` with the following content:

```
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      # Checkout the repository code
      - name: Checkout Code
        uses: actions/checkout@v2

      # Build Phase: Build the Docker image from the Dockerfile
      - name: Build Docker Image
        run: docker build -t my-docker-app:latest .

      # Test Phase: Run the container and execute tests inside it
      - name: Run Container and Execute Tests
        run: docker run --rm my-docker-app:latest ./run-tests.sh

  deploy:
    runs-on: ubuntu-latest
    needs: build-and-test
    if: github.ref == ’refs/heads/main’
    steps:
      - name: Checkout Code
        uses: actions/checkout@v2

      # Deployment Phase: Log in to Docker Hub
      - name: Log in to Docker Hub
        run: echo "${{ secrets.DOCKER_PASSWORD }}" | docker login -u ${{ secrets.DOCKER_USERNAME }} --password-stdin

      # Tag the built image for Docker Hub
      - name: Tag Docker Image
        run: docker tag my-docker-app:latest mydockerhubuser/my-docker-app:latest

      # Push the Docker image to Docker Hub
      - name: Push Docker Image
        run: docker push mydockerhubuser/my-docker-app:latest

      # (Optional) Deploy to Kubernetes using your
```

```
         deployment manifests
      - name: Deploy to Kubernetes
        run: |
          echo "${{ secrets.KUBE_CONFIG }}" | base64 --decode > kubeconfig
          kubectl --kubeconfig=kubeconfig apply -f k8s/deployment.yaml
```

Note: Replace `mydockerhubuser` and `my-docker-app` with your Docker Hub username and repository name. Ensure your repository secrets (`DOCKER_USERNAME`, `DOCKER_PASSWORD`, and optionally `KUBE_CONFIG`) are configured.

3.6 Task 6: Cleanup and Reflection

1. **Clean Up Docker Environment:**
   Remove unused containers and images:

```
docker container prune -f
docker image prune -a -f
```

2. **Reflection Questions:**

   - What are the benefits of containerization compared to traditional virtual machines?
   - How does Docker ensure consistency across different environments?
   - How does integrating Docker into a CI/CD pipeline improve deployment reliability?