

Assignment 4: Orchestration and Contextualization using Docker containers

Task 1. Introduction to Docker containers and DockerHub

Questions:

1 - Explain the difference between contextualization and orchestration processes.

Contextualization: Contextualization in Docker refers to preparing and configuring an individual container so that it can run its application correctly in any environment. This process involves defining the container's environment, dependencies, and settings in a way that aligns with its operational requirements.

How it's done: In Docker, contextualization is primarily achieved through a Dockerfile and sometimes additional scripts. A Dockerfile is a text document containing all the commands a user could call on the command line to assemble an image. By using a Dockerfile, we can create a template that automates the process of creating a Docker image which includes the application and its environment.

Orchestration: Orchestration in the context of Docker is about managing the lifecycle of containers, especially when they are deployed in large numbers and possibly across multiple host machines or cloud environments. It involves not just the initial deployment of containers, but also their ongoing management, scaling, networking, and eventual termination.

How it's done: Orchestration is typically achieved through tools like Docker Compose, Kubernetes, or Docker Swarm. These tools allow for the definition of complex container setups, their relationships, and how they should be managed. They handle tasks like keeping containers running, scaling them according to load, managing network configurations, and ensuring fault tolerance and high availability.

2 - Explain the followings commands and concepts:

i) Contents of the docker file used in this task.

FROM ubuntu:22.04: This line specifies the base image. Here, it's using Ubuntu 22.04 as the base image for the Docker container.

RUN apt-get update and RUN apt-get -y upgrade: These commands update the package lists and upgrade any existing packages in the base image.

RUN apt-get install sl: This installs the sl package (Steam Locomotive) in the container. It's a small program that displays animations in the terminal for example we have seen a moving train on the terminal after running the command sl.

ENV PATH="\${PATH}:/usr/games/": Sets the environment variable PATH to include the directory where the sl command is located. This is necessary for the system to find and execute the sl command.

CMD ["echo", "Data Engineering-I."]: Specifies what command to run when the container starts. In this case, it's using echo to print "Data Engineering-I." to the console.

ii) Explain the command

```
# docker run -it mycontainer/first:v1 bash
```

Basically, this command starts a new container from the mycontainer/first:v1 image, attaches an interactive terminal to it, and opens a bash shell inside the container.

iii) Show the output and explain the following commands:

```
# docker ps
```

```
[ubuntu@a3-masudul-islam:~$ ls
A3-Apache-Spark A4 DE1-spark Dockerfile data-engineering-apache-spark snap
[ubuntu@a3-masudul-islam:~$ sudo bash
[root@a3-masudul-islam:/home/ubuntu# docker ps
CONTAINER ID   IMAGE                COMMAND             CREATED          STATUS            PORTS           NAMES
6a42400802a5   mycontainer/first:v1 "bash"             38 minutes ago   Up 38 minutes    0.0.0.0:22->0.0.0.0:22   dreamy_saha
root@a3-masudul-islam:/home/ubuntu#
```

docker ps: Lists all the currently running containers.

Output: It shows details of running containers like container ID, image being used, when it was created (38 minutes ago), status, ports, and names.

```
# docker images
```

```
[root@a3-masudul-islam:/home/ubuntu# docker images
REPOSITORY          TAG         IMAGE ID      CREATED          SIZE
myfirstcontainer/first v1          67e9f3df4835 45 minutes ago   130MB
mycontainer/first    v1          1ca1f6c7513f  About an hour ago 130MB
ubuntu               22.04      3db8720ecbf5 12 days ago      77.9MB
root@a3-masudul-islam:/home/ubuntu#
```

docker images: Lists all docker images available on our system.

Output: It displays information about the docker images we have in our system, including repository name, tag, image ID, creation time, and size.

```
# docker stats
```

```
masudulislam — root@a3-masudul-islam: /home/ubuntu — ssh 130.238.1.100
CONTAINER ID   NAME          CPU %       MEM USAGE / LIMIT   MEM %      NET I/O       BLOCK I/O     PIDS
6a42400802a5   dreamy_saha   0.00%      2.68MiB / 3.828GiB   0.07%      1.37kB / 0B    2.45MB / 0B    1
```

docker stats: Displays a live stream of resource usage statistics for running containers.

Output: Shows real-time information for each running container, such as CPU usage, memory usage, network I/O, and block I/O.

3-What is the difference between docker run, docker exec and docker build commands?

docker run: Used to create and start a new container from a docker image. It initializes and starts a container with a specified configuration.

docker exec: Used to execute a new command in an already running container. It's useful for interacting with an existing container.

docker build: Used to build a docker image from a Dockerfile. It processes the instructions in the Dockerfile to create an image.

4 - Create an account on DockerHub and upload your newly built container to your DockerHub area. (Watch the UI and think: is it uploading the entire image, or just your changes? How is this advantageous?) Briefly explain how DockerHub is used. Make your container publicly available and report the name of your publicly available container.

Uploading the image: Docker uploads only the layers of the image that have changed or don't already exist on Docker Hub. This is advantageous because it saves bandwidth and time, making the process more efficient.

How dockerHub is used: DockerHub is used as a cloud-based registry service to store and share docker images. Users can push images to Docker Hub to make them available to others. Docker Hub can also automatically build images from a GitHub or Bitbucket repository with source code.

Publicly available container: I made my container publicly available on Docker Hub under the repository name **masudulislam/a4**. The **latest** (see screenshot below) tag indicates the most recent version of the image.

```
root@a3-masudul-islam:/home/ubuntu# docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com to create one.
Username: masudulislam
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
root@a3-masudul-islam:/home/ubuntu# docker tag mycontainer/first:v1 masudulislam/a4
root@a3-masudul-islam:/home/ubuntu# docker push masudulislam/a4
Using default tag: latest
The push refers to repository [docker.io/masudulislam/a4]
58033c8b6890: Pushed
ae7959845ebe: Pushed
a48d8573fe4b: Pushed
d101c9453715: Mounted from library/ubuntu
latest: digest: sha256:2679a5efae06fa4470efe451508ab1bdca0bbb6d1f495190ae7a0e286be8695c size: 1161
root@a3-masudul-islam:/home/ubuntu#
```

masudulislam / [Repositories](#) / [a4](#) / [General](#)

General

Tags

Builds


Collaborators

Webhooks

Settings



masudulislam/a4 

Updated 2 minutes ago

Data engineering assignment 4. 

Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
 latest		Image	---	2 minutes ago

5 - Explain the difference between docker build and docker-compose commands.

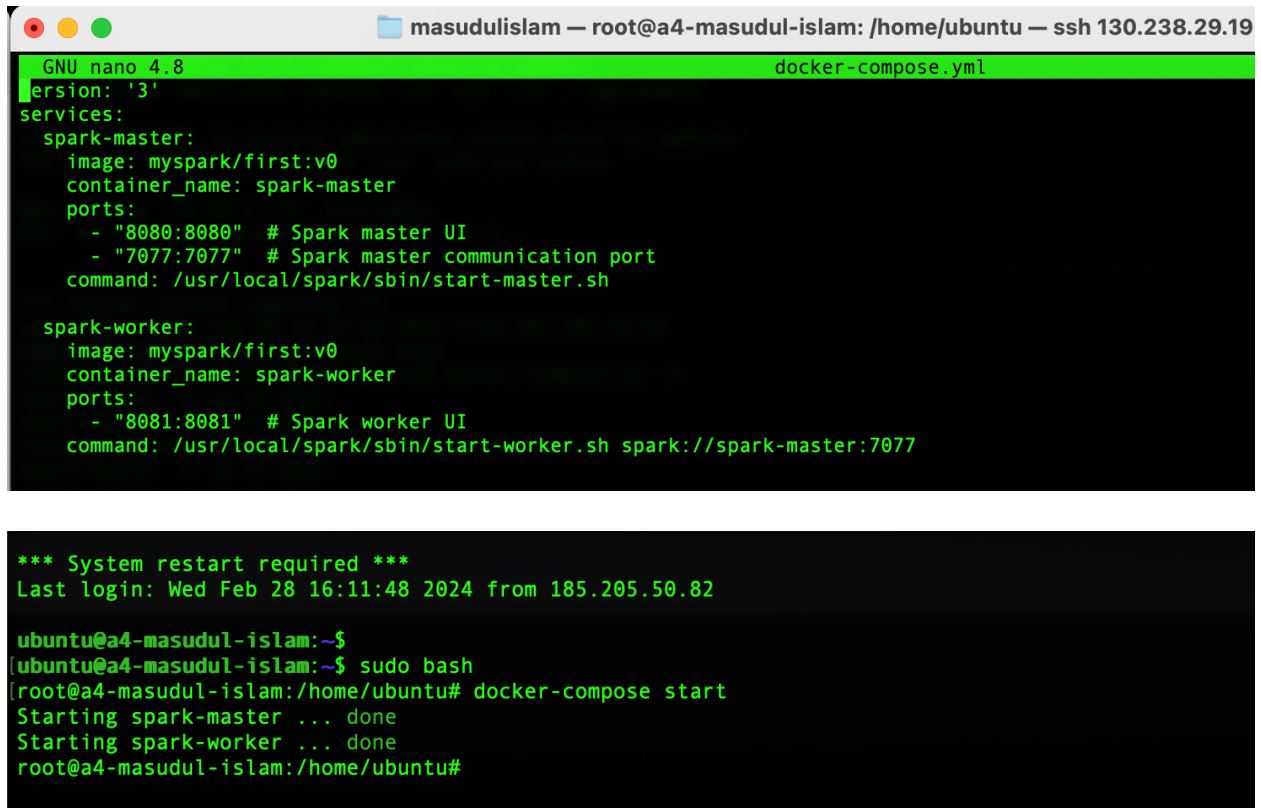
docker build: This command is used to create a docker image from a Dockerfile. It processes the instructions written in the Dockerfile, which typically include setting up the operating system, installing dependencies, copying project files into the image, and configuring execution commands. The result is a docker image that can be used to run containers.

docker-compose: This command is used with docker compose, a tool for defining and running multi-container docker applications. With a docker-compose.yml file, we can configure all aspects of the service stack, including which images to use, ports to expose, volumes to mount, and other services the application depends on. docker-compose allows us to manage the whole lifecycle of an application with a single command, handling the building of images if necessary, and the creation, starting, and stopping of containers based on the configurations defined in the docker-compose.yml file.

Task 2. Build a multi-container Apache Spark cluster using *docker-compose*

Questions

1 - Explain your *docker-compose* configuration file.



```
masudulislam — root@a4-masudul-islam: /home/ubuntu — ssh 130.238.29.19
GNU nano 4.8 docker-compose.yml
version: '3'
services:
  spark-master:
    image: myspark/first:v0
    container_name: spark-master
    ports:
      - "8080:8080" # Spark master UI
      - "7077:7077" # Spark master communication port
    command: /usr/local/spark/sbin/start-master.sh

  spark-worker:
    image: myspark/first:v0
    container_name: spark-worker
    ports:
      - "8081:8081" # Spark worker UI
    command: /usr/local/spark/sbin/start-worker.sh spark://spark-master:7077

*** System restart required ***
Last login: Wed Feb 28 16:11:48 2024 from 185.205.50.82

ubuntu@a4-masudul-islam:~$
[ubuntu@a4-masudul-islam:~$ sudo bash
[root@a4-masudul-islam:/home/ubuntu# docker-compose start
Starting spark-master ... done
Starting spark-worker ... done
root@a4-masudul-islam:/home/ubuntu#
```

My *docker-compose.yml* file is designed to orchestrate a basic Apache Spark cluster, consisting of a master node and a worker node. Here's an explanation of its configuration:

version:

version: '3': This specifies the version of the Docker Compose file format being used. Version 3 is a widely adopted version that supports most of Docker's features.

services:

This section defines the containers that make up the application. In my case, there are two services: spark-master and spark-worker.

spark master:

spark-master: This is the service name for the Spark master node.

image: myspark/first:v0: Specifies the Docker image to use for this service. The image tagged as v0 in the myspark/first repository is used.

container_name: spark-master: A custom name assigned to the container running the Spark master.

ports: This maps the container's ports to the host machine's ports.

"8080:8080": Exposes the Spark master web UI by mapping port 8080 of the container to port 8080 on the host.

"7077:7077": Exposes the Spark master's communication port.

command: The command to be executed when the container starts. Here, it starts the Spark master service.

spark worker:

spark-worker: The service name for the Spark worker node.

image: myspark/first:v0: Uses the same Docker image as the Spark master service.

container_name: spark-worker: A custom name for the worker container.

ports:

"8081:8081": Maps port 8081 of the container to port 8081 of the host, exposing the Spark worker's web UI.

command: The command to start the Spark worker and connect it to the Spark master. It specifies the master's URL (spark://spark-master:7077).

2-What is the format of the *docker-compose* compatible configuration file?

The format of my docker-compose compatible configuration file is YAML, which stands for *YAML Ain't Markup Language*. YAML is a human-readable data serialization standard that is particularly well-suited for configuration files. For example in my docker-compose file:

The **version** specifies the docker compose file version.

The **services** defines the containers (services) we want to run.

Each service (spark-master, spark-worker) has its configuration, like image, container_name, ports, and command.

3 - What are the limitations of *docker-compose*?

The limitations of docker-compose include:

Single-host focus: Docker compose is primarily designed for managing containers on a single host, making it less suitable for larger, distributed environments.

Limited scalability: While docker compose can manage multiple containers, it lacks the advanced scaling features needed for large-scale applications.

No built-in cluster management: Docker compose does not provide cluster management features, which are essential for managing containers across multiple hosts.

Limited fault tolerance: Docker compose does not offer built-in fault tolerance

Manual updates: Updating container configurations often requires manual actions, unlike systems with automatic rolling updates and rollbacks.

Task 3. Introduction to different orchestration and contextualization frameworks (1 point)

In the field of distributed computing, the orchestration and contextualization of applications play crucial roles, especially as systems scale and grow in complexity. Orchestration refers to the automated arrangement, coordination, and management of complex computer systems, middleware, and services. Contextualization, on the other hand, involves configuring these systems to operate effectively in a specific environment, taking into account dependencies, configurations, and interactions with other systems.

Orchestration and contextualization frameworks:

Kubernetes:

Features: Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers into logical units for easy management and discovery. Key features include horizontal scaling, self-healing (automatic restarts of failed containers), load balancing, and rollouts & rollbacks.

Design philosophy: Designed with the principles of declarative configuration and automation, Kubernetes focuses on building a platform for automating deployment, scaling, and operations of application containers across clusters of hosts.

Docker Swarm:

Features: Docker Swarm provides native clustering functionality for docker containers, turning a group of docker engines into a single, virtual docker engine. Key features include ease of use, high availability, load balancing, and secure by default.

Design philosophy: Docker Swarm is designed to be a simple and straightforward way to orchestrate docker containers. It is a lightweight alternative to Kubernetes and is deeply integrated into the docker ecosystem.

Apache Mesos:

Features: Apache Mesos is a project to manage computer clusters that provides efficient resource isolation and sharing across distributed applications or frameworks. Mesos is known for its scalability and fault-tolerant design.

Design philosophy: Mesos is built on the principles of abstraction and scalability, abstracting CPU, memory, storage, and other compute resources away from machines (physical or virtual).

Amazon ECS:

Features: Amazon Elastic Container Service (ECS) is a highly scalable, high-performance container orchestration service that supports docker containers. It allows us to run and scale containerized applications on AWS easily.

Design philosophy: ECS is designed to integrate deeply with the AWS ecosystem, providing a solution for container management that leverages the powerful capabilities of AWS services.

In Task 2, I have successfully orchestrated a multi-container Spark cluster using docker compose. The Kubernetes and Docker Swarm can significantly enhance and scale this solution for larger. Here's how:

Scalability: Kubernetes excels in scaling applications. It can automatically adjust the number of Spark worker nodes based on the workload, which is crucial for data-intensive tasks.

Simplified management: Both Kubernetes and Docker Swarm offer simplified and more robust management of containerized applications compared to docker compose, particularly in a production environment.

Resilience and high availability: Kubernetes self-healing feature automatically restarts failed containers, ensuring that the Spark cluster remains available and resilient to failures.

Load balancing: Kubernetes can intelligently distribute incoming network traffic across the Spark cluster, ensuring efficient use of resources.

Service discovery and networking: Kubernetes provides advanced service discovery and networking capabilities that can simplify the communication between Spark nodes.

In conclusion, while Docker Compose is suitable for development and small-scale deployments, Kubernetes or Docker Swarm are more robust solutions that provide to the needs of large-scale, distributed applications. They give the necessary tools for automatic scaling, self-healing, and efficient resource utilization, which are essential for complex applications like a Spark cluster.