#90DaysOfDevOps Challenge - Day 18 - Docker Project for DevOps Engineers (Docker Compose)

Welcome to Day 18 of the #90DaysOfDevOps challenge. Today, we will explore an exciting Docker project specifically designed for DevOps Engineers. Our focus will be on Docker Compose, a powerful tool that simplifies the management of multi-container applications. By leveraging Docker Compose, we can easily orchestrate and deploy complex application stacks. Let's get started.

Docker Compose

Docker Compose is a command-line tool that allows us to define and manage multi-container applications. With Docker Compose, we can describe our application's services, networks, and volumes using a simple YAML file. It provides a declarative approach to defining the infrastructure requirements of our application, making it easier to manage and reproduce our environment across different stages of development, testing, and production.

Install Docker Compose

To begin, we need to ensure that Docker Compose is installed on our system. Here are the steps to install Docker Compose on Ubuntu:

1. Update the package index, and install the latest version of Docker Compose:

```
sudo apt-get update
sudo apt-get install docker-compose-plugin
```

2. Verify that Docker Compose is installed correctly by checking the version:

```
docker compose version
```

If the installation was successful, you should see the version information for Docker Compose.

Docker Compose CLI Reference

The docker-compose command provides a number of subcommands to manage Docker containers with docker-compose. Please find below details of the subcommands. Before reading the below commands, remember that you passed the service name as an argument (not the container name)

build

The build option is used to build images for services for which build is defined.

```

```

```
$ docker-compose build              ## Build all services
$ docker-compose build web          ## Build single service
```

up

Use to create docker containers with available services in `docker-compose.yml` file in the current directory. Use `-d` switch to launch containers in daemon mode.

```
$ docker-compose up -d              ## Create all containers
$ docker-compose up -d web          ## Create single container
```

down

This will stop and delete all containers, network and associated images for the services defined in a config file

```
$ docker-compose down           ## Restart all containers
$ docker-compose down web       ## Restart single container
```

ps

This will list all containers created for the services defined in a config file with their status, port bindings and command.

```
$ docker-compose ps
```

exec

This will execute a command to the running container. For example, list files in a container associated with a **web** service.

```
$ docker-compose exec web ls -l
```

start

This will start stopped containers of the services defined in the config file

```
$ docker-compose start              ## Start all containers
$ docker-compose start web          ## Start single container
```

stop

This will stop running containers for the services defined in the config file

```
$ docker-compose stop             ## Stop all containers
$ docker-compose stop web         ## Stop single container
```

restart

This will restart containers of the services defined in the config file

```
$ docker-compose restart           ## Restart all containers
$ docker-compose restart web       ## Restart single container
```

pause

This will pause running containers for the services defined in the config file.

```
$ docker-compose pause            ## Pause all containers
$ docker-compose pause web        ## Pause single container
```

unpause

This will start paused containers for the services defined in the config file.

```
$ docker-compose pause            ## Start all paused containers
$ docker-compose pause web        ## Start single paused container
```

rm

This will remove stopped containers for the services defined in the config file.

```
$ docker-compose rm               ## Start all paused containers
$ docker-compose pause web        ## Start single paused container
```

How to Run Docker Commands Without Sudo?

By default, running Docker commands requires sudo or root privileges. However, it is possible to grant our user permission to run Docker commands without sudo. Here's how you can achieve this:

1. Add the user to the `docker` group by running the following command (replace `<username>` with your actual username):

```
sudo usermod -aG docker <username>
```

2. After adding the user to the `docker` group, it is necessary to reboot the system for the changes to take effect. Restart your machine by running:

```
sudo reboot
```

After rebooting, you should be able to run Docker commands without sudo.

What is YAML?

YAML, which stands for **"YAML Ain't Markup Language"** is a popular data serialization format widely used in the DevOps field for creating configuration files. It provides a human-readable and machine-friendly syntax that allows developers and system administrators to define structured data clearly and concisely.

One of the primary **use cases** of YAML in DevOps is for configuring and defining application settings and environments. For example, in Docker Compose, a YAML file is used to specify the **services**, **networks**, **volumes**, and **other configurations** required to run a multi-container application.

YAML files utilize **indentation** and **simple syntax** to represent data hierarchically, making it easy to understand and work with. This readability is one of YAML's key strengths, as it enables both humans and machines to parse and interpret the data effortlessly.

Additionally, YAML supports various **data types** such as **scalars** (strings, numbers, booleans), **sequences** (arrays, lists), and **mappings** (key-value pairs). This flexibility allows for the representation of complex configurations and relationships between different components of an application.

Task 1

Learn how to use the docker-compose.yml file, to set up the environment, configure the services and links between different containers, and also to use environment variables in the docker-compose.yml file

Let's go through the `docker-compose.yml` file and understand how it sets up the environment, configures services, establishes links between containers, and uses environment variables:

```
version : "3.3"
services:
  web:
    image: nginx:latest
    ports:
      - "80:80"
  db:
    image: mysql
    ports:
      - "3306:3306"
```

```
    environment:
      - "MYSQL_ROOT_PASSWORD=test@123"
```

- The `version: "3.3"` line specifies the version of the Docker Compose file syntax being used.
- Under the `services` section, two services are defined: `web` and `db`. Each service represents a container.
- The `web` service uses the `nginx:latest` image, which is a popular web server. It is accessible through port mapping defined as `ports: - "80:80"`, where port 80 of the container is mapped to port 80 of the host machine. This allows accessing the web server running inside the container via `http://localhost` on the host machine.
- The `db` service uses the `mysql` image, which is a widely used relational database management system. Similar to the `web` service, it also has port mapping defined as `ports: - "3306:3306"`, allowing access to the MySQL database server running inside the container via `localhost` on port 3306 of the host machine.
- Additionally, the `db` service sets the `MYSQL_ROOT_PASSWORD` environment variable to `test@123`. This sets the root password for the MySQL database inside the container to ensure secure access.

By running `docker-compose up` command in the directory where the `docker-compose.yml` file is located, Docker Compose will create and start the defined services. It automatically manages the networking between containers and sets up the specified environment variables.

Task 2

Pull a pre-existing Docker image from a public repository (e.g. Docker Hub) and run it on your local machine. Run the container as a non-root user. Make sure you reboot the instance after giving permission to the user.

Use the docker pull command followed by the image name and tag to pull the image.

```
docker pull nginx:latest
```

```
esteban@ubuntu-server-template:~$ docker pull nginx:latest
latest: Pulling from library/nginx
Digest: sha256:af296b188c7b7df99ba960ca614439c99cb7cf252ed7bbc23e90cfda59092305
Status: Image is up to date for nginx:latest
docker.io/library/nginx:latest
```

Inspect the container's running processes and exposed ports using the docker inspect command.

```
docker inspect <container-id>
```

```
esteban@ubuntu-server-template:~$ docker run -d -p 80:80 nginx
7c36e1c9dbbdd9745dedb53f7b456852f3aae78a2b138c68cc006bfb23953c91
esteban@ubuntu-server-template:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                                   NAMES
7c36e1c9dbbd   nginx     "/docker-entrypoint.…"   5 seconds ago   Up 4 seconds   0.0.0.0:80->80/tcp, :::80->80/tcp   quirky_lamport
esteban@ubuntu-server-template:~$ docker inspect 7c36e1c9dbbd
[
    {
        "Id": "7c36e1c9dbbdd9745dedb53f7b456852f3aae78a2b138c68cc006bfb23953c91",
        "Created": "2023-06-08T18:19:38.465514265Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
            "nginx",
            "-g",
            "daemon off;"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 1578,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2023-06-08T18:19:38.95765052Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
```

Use the docker logs command to view the container's log output.

```
docker logs <container-id>
```

```
esteban@ubuntu-server-template:~$ docker logs 7c36e1c9dbbd
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2023/06/08 18:19:39 [notice] 1#1: using the "epoll" event method
2023/06/08 18:19:39 [notice] 1#1: nginx/1.25.0
2023/06/08 18:19:39 [notice] 1#1: built by gcc 10.2.1 20210110 (Debian 10.2.1-6)
2023/06/08 18:19:39 [notice] 1#1: OS: Linux 5.15.0-73-generic
2023/06/08 18:19:39 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2023/06/08 18:19:39 [notice] 1#1: start worker processes
2023/06/08 18:19:39 [notice] 1#1: start worker process 29
2023/06/08 18:19:39 [notice] 1#1: start worker process 30
```

Use the docker stop and docker start commands to stop and start the container.

Use the `docker stop` command followed by the container ID or name to stop the running container. For example:

```
docker stop <container-id>
```

Use the `docker start` command followed by the container ID or name to start the stopped container. For example:

```
docker start <container-id>
```

```
esteban@ubuntu-server-template:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                                   NAMES
7c36e1c9dbbd   nginx     "/docker-entrypoint.…"   3 minutes ago   Up 3 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp   quirky_lamport
esteban@ubuntu-server-template:~$ docker stop 7c36e1c9dbbd
7c36e1c9dbbd
esteban@ubuntu-server-template:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
esteban@ubuntu-server-template:~$ docker start 7c36e1c9dbbd
7c36e1c9dbbd
esteban@ubuntu-server-template:~$ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS         PORTS                                   NAMES
7c36e1c9dbbd   nginx     "/docker-entrypoint.…"   4 minutes ago   Up 1 second    0.0.0.0:80->80/tcp, :::80->80/tcp   quirky_lamport
esteban@ubuntu-server-template:~$
```

Use the docker rm command to remove the container when you're done.

```
docker rm <container-id>
```

```
esteban@ubuntu-server-template:~$ docker stop 7c36e1c9dbbd
7c36e1c9dbbd
esteban@ubuntu-server-template:~$ docker rm 7c36e1c9dbbd
7c36e1c9dbbd
esteban@ubuntu-server-template:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
esteban@ubuntu-server-template:~$
```

Task 3

Running my docker-compose file to create Jenkins and Portainer containers

```
version: "3.9"
services:
  portainer:
    container_name: portainer
    image: portainer/portainer-ce:latest
    restart: always
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - portainer_data:/data
    ports:
      - 9000:9000

  jenkins:
    container_name: jenkins
```

```
    image: jenkins/jenkins:lts
    restart: always
    volumes:
      - jenkins_data:/var/jenkins_home
    ports:
      - 8081:8080
      - 50000:50000

volumes:
  portainer_data:
  jenkins_data:
Congratulations on completing Day 18 of the #90DaysOfDevOps challenge.
Today, we dived into Docker Compose and explored its features for
orchestrating multi-container applications. We learned how to define
services, configure dependencies, and utilize environment variables using
a docker-compose.yml file. Additionally, we discovered essential Docker
Compose CLI commands and discussed how to run Docker commands without
sudo.
```

- The `version` field specifies the version of the Docker Compose file format being used (in this case, version 3.9).
- Under the `services` section, we define two services: `portainer` and `Jenkins`.
- For the `portainer` service:
    - The `image` field specifies the Portainer image to be used.
    - The `volumes` field mounts the Docker socket and a volume for Portainer data persistence.
    - The `ports` field maps port 9000 of the container to port 9000 of the host machine.
- For the `Jenkins` service:
    - The `image` field specifies the Jenkins image to be used.
    - The `volumes` field mounts a volume for Jenkins data persistence.
    - The `ports` field maps ports 8081 and 50000 of the container to the same ports on the host machine.

This example sets up Portainer for managing Docker containers and Jenkins for continuous integration and continuous delivery (CI/CD) processes. You can access Portainer on port 9000 of the host machine and Jenkins on port 8081.

Keep up the great work, and get ready for Day 19, where we will continue exploring Docker for DevOps Engineers

Congratulations on completing Day 18 of the #90DaysOfDevOps challenge. Today, we delved into Docker Compose and explored its features for orchestrating multi-container applications. We learned how to define services, configure dependencies, and utilize environment variables using a docker-compose.yml file.

Additionally, we discovered essential Docker Compose CLI commands and discussed how to run Docker commands without sudo.

Get ready for Day 19, where we will continue exploring Docker for DevOps Engineers. Stay tuned!