**Docker Lab Exercise**

**I Docker Basic Commands**

**Questions:**

1. Install Docker on your system and verify the successful installation.
2. Check the Docker service status and ensure it is running.
3. Pull the ubuntu base image from Docker Hub.
4. List all the Docker images available on your local system.
5. Inspect the downloaded ubuntu image in the repositories and display its low-level information.
6. Create a new Docker container using the ubuntu image.
7. Create another Docker container using the ubuntu image and name it test.
8. Create an interactive container named user1 using the ubuntu image and start a bash shell inside it.
9. Start the user1 container.
10. Attach to the running user1 container.
11. Stop the user1 container.
12. Restart the user1 container and attach to it interactively.
13. Write shell scripts for the following tasks:
    * Create 100 containers.
    * Delete 100 containers.
    * Create 10 containers named user1 to user10.
    * Start the 10 containers.
14. Monitor computing resource usage for all 10 running containers.
15. Monitor computing resource usage specifically for the user5 container.
16. Check the Docker logs for the user3 container.
17. Check the Docker logs for the user3 container in real-time.
18. Display the detailed runtime details, operational data, and configurations of the Docker environment.
19. Display the history of the ubuntu Docker image.
20. Search and display the details of the Jenkins Docker image.
21. Remove all existing containers with a single command.
22. Remove the ubuntu Docker image.
23. Create a file named user.txt in the host system and copy it to the user1 container. Then, create a file named test.txt in the user1 container and copy it back to the host system.
24. Verify that the files (user.txt and test.txt) are successfully transferred between the host and the user1 container.
25. Run a container named user3 in the background using the ubuntu image, and attach to the container to interact with it.
26. List all running containers by inspecting the /var/lib/docker/containers directory.
27. Check the log file for a specific running container in the /var/lib/docker/containers directory.

**Q27. Creating a User Container for Isolated Lab Environments**

**Objective:**

This lab helps create a user container on a host system. Users can work in an isolated container environment with root privileges, minimizing the risk of impacting the host system. Mentors can monitor user activities and debug issues effectively, making this ideal for online lab courses.

**Steps:**

1. **Create a User Account:** Create a new user named user1 on the host system: adduser user1
2. **Set Password for the User:** Assign a password for user1: passwd user1
3. **Grant Sudo Privileges:** Add user1 to the sudo group: usermod -aG sudo user1
4. **Create a Docker Container for user1:** Create and run a container named user1 using the ubuntu image: docker run -it --name user1 ubuntu /bin/bash
5. **Modify user1’s .profile File:** Edit the /home/user1/.profile file to include the following line: sudo docker start -a -i user1; exit
6. **Test Container Access via SSH:** SSH into the host system as user1: ssh <host\_ip>. Upon logging in as user1, you will be directly connected to the user1 container shell.

**Benefits:**

* **Isolation**: Mistakes made by users in the container will not affect the host system.
* **Monitoring**: Mentors can track user activities and debug container issues without exposing the host environment.
* **Ease of Use**: Logging in via SSH seamlessly redirects users to their container environment.

**Q28. Setting Up Apache2 in a Docker Container**

**Question:**

1. Start the user1 Docker container and perform the following tasks inside the container:
   * Update the package list.
   * Install the following packages: apache2, apache2-utils, and vim.
   * Edit the /var/www/html/index.html file using vim and modify the content at line number 208 to display a custom message.
   * Start the Apache2 service.
   * Use elinks with the container's IP address to verify the changes made to the index.html file.

**Expected Outcome:**

* Apache2 is successfully installed and running inside the user1 container.
* The custom message in the /var/www/html/index.html file is accessible through the elinks browser.

**Q29. Running a Web Server in Docker**

**Question:**

1. Execute the following steps to run a web server inside a Docker container:
   * Pull the Docker image thangaraju/webserver\_test from Docker Hub.
   * Run a container using the pulled image, map port 9999 on the host to port 80 in the container, and start the container interactively.
   * Start the Apache2 service inside the container.
2. Access the web server in a browser using the following URL:
   * http://<host\_ip\_address>:9999

**Expected Outcome:**

* The web server is successfully running and accessible via the browser at the specified host IP and port.

**Q30. Creating a Docker Image for C Program Development and Pushing to Docker Hub**

**Objective:**

This lab helps you create a custom Docker image tailored for C program development and push it to Docker Hub. The image will include necessary tools for compiling and running C programs, making it reusable across systems.

**Steps:**

1. **Create and Run a Docker Container**: Start a Docker container using the ubuntu image.
2. **Install Development Tools**: Inside the container, install the required packages for C program development (e.g., gcc, make, vim, etc.).
3. **Configure the Environment**: Set up a directory structure and include sample files for C program compilation.
4. **Commit the Container**: Commit the container to create a custom Docker image for C program development.
5. **Push the Docker Image to Docker Hub**: Log in to Docker Hub and push the custom image to your repository.
6. **Verify the Pushed Image**: Check your Docker Hub account to confirm the image is successfully uploaded.
7. **Pull and Test the Image**: On another system, pull the pushed image and verify its functionality by compiling and running a sample C program.

**Benefits:**

* **Reusable Development Environment**: Simplifies setup for compiling and running C programs.
* **Cross-System Compatibility**: The image can be used on any system with Docker installed.
* **Efficient Sharing**: Pushing the image to Docker Hub makes it accessible for collaboration.

**Q31. Working with Docker Volumes**

**Question:**

1. Perform the following tasks to understand and use Docker volumes effectively:
   * **Create a Docker Volume**: Create a volume named devtest.
   * **List Docker Volumes**: Verify the volume creation by listing all the available Docker volumes.
   * **Inspect the Docker Volume**: Inspect the devtest volume to check its properties.
   * **Mount the Volume**: Run a container using the ubuntu image and mount the devtest volume to /usr/src/app in the container.
   * **Store Files in the Volume**: Inside the container, navigate to /usr/src/app and create 100 files named abc1.txt to abc100.txt. Exit the container once done.
   * **Access the Volume from Another Container**: Start another container and mount the devtest volume to /app. Verify that all the 100 files are accessible in the /app directory.
2. **Delete the Volume**:
   * Remove the devtest volume after completing the above steps.

**Expected Outcome:**

* The devtest volume is successfully created, mounted, and shared between containers.
* The files stored in the volume are accessible across containers.

**Q32. Mini Project: Automated Docker Container and User Management System**

**Objective:**

Design a Bash script that automates the creation of Docker containers for different user groups, manages user accounts on the host system, and facilitates cleanup operations. This mini-project focuses on building an efficient script for managing containerized environments and user access.

**Requirements:**

1. **User Menu for Docker Images**:
   * Create a menu-based script to allow users to select one of the following Docker images:
     + unix1admin
     + debugger
     + java
     + developer
     + unix\_11\_assessment\_base
   * Based on the user's choice, start a container from the selected image.
2. **Dynamic User Account Creation**:
   * Prompt the user to input the number of user accounts to create.
   * Create user accounts (user1, user2, etc.) on the host system.
   * For each user:
     + Set up a home directory.
     + Add .bash\_profile configurations to enable direct login into their respective Docker containers.
3. **Container Management**:
   * For each user created, start a Docker container with the user's name, using the image selected in the menu.
   * Ensure the container is linked to the respective user account on the host.
4. **Cleanup Operations**:
   * Implement a script to delete all user accounts and associated Docker containers in bulk.

**Deliverables:**

1. A functional Bash script that:
   * Creates user accounts and containers dynamically.
   * Allows users to select Docker images via a menu interface.
   * Links user accounts to containers for seamless integration.
2. A cleanup script that:
   * Deletes all user accounts and containers created during the process.

**Expected Outcome:**

* The script should streamline the creation and management of user accounts and containers for multiple users.
* The cleanup script should remove all traces of user accounts and containers effectively.
* The project should demonstrate practical automation of containerized environments using Docker and Bash scripting.

\*\*\*\*\*\*\*\*\*\*

**II Dockerfile**

**Q1. Dockerizing a Python Application**

**Objective:** Write and run a Dockerized Python application.

**Task:**

1. Write a Python script named hello.py that prints:  
   Hello, World! Welcome to Docker!
2. Create a Dockerfile to containerize the Python script.
3. Build a Docker image named hello-world and run a container based on this image.
4. Use the docker ps -a command to verify the container creation and its status.

**Instructions:**

* Students must create the hello.py script and Dockerfile themselves.
* Demonstrate the successful execution of the container by showing the output and the container's details using docker ps -a.

**Q2.** **Deploying an Apache Web Server Using Docker**

**Objective:** Set up an Apache web server using Docker and demonstrate its functionality.

**Task:**

1. Write a Dockerfile that:
   * Uses the ubuntu base image.
   * Installs Apache2 and its utilities.
   * Exposes port 80.
   * Starts the Apache server in the foreground.
2. Build a Docker image named apache-server using the docker build command.
3. Run a container from the apache-server image and verify:
   * The container is running.
   * Apache is serving the default page.
4. Use the docker ps -a command to verify the status and details of the container.
5. Use a tool like elinks or a browser to access http://localhost and confirm the Apache default page is displayed.

**Notes:**

* Address any warnings about the server's fully qualified domain name (FQDN) as described in the slides.

**Q3. Deploying an Apache Web Server with FQDN Configuration**

**Objective:** Set up an Apache web server using Docker, ensuring there are no warnings about the server's fully qualified domain name (FQDN).

**Task:**

1. Write a Dockerfile that:
   * Uses the ubuntu base image.
   * Installs Apache2 and its utilities.
   * Updates the Apache configuration file to set the ServerName directive to localhost to suppress FQDN warnings.
   * Exposes port 80.
   * Starts the Apache server in the foreground.
2. Build a Docker image named my-image using the docker build command with the -f option to specify the Dockerfile.
3. Run a container from the my-image image using the docker run command, mapping port 80 of the container to port 80 of the host.
4. Verify that:
   * The container is running (docker ps -a).
   * The Apache default page can be accessed at http://localhost using a browser or a text-based tool like elinks.

**Additional Task:**

* Demonstrate that no warnings related to FQDN appear in the container logs by ensuring the ServerName localhost directive is correctly configured in the Apache configuration file.

**Q4. Deploying an NGINX Web Server with External Access**

**Objective:** Set up an NGINX web server using Docker, run it in the background, and access it from outside the virtual machine.

**Expected Result:**

* The custom HTML page with "Hello, World!" and a welcome message should be displayed when accessed via a browser at http://<VM\_IP>:8080.

**Notes:**

* Ensure the VM's network settings allow external access (e.g., set to bridged mode or port forwarding is configured).

**Q5. Multi-Stage Build for a C Application**

**Question:** Create a multi-stage Docker build for a simple C application. Write a Dockerfile to:

1. Use a GCC image to compile a hello.c program into a static binary during the first stage.
2. Use an Alpine image in the second stage to copy the compiled binary and run it as the container's entry point.

Build the Docker image as hello-c-app and run the container to verify the output.

\*\*\*\*\*\*\*\*\*\*

**III Docker Compose**

**Q1. Installing Docker Compose**

**Question:** Install Docker Compose on your system by following these steps:

1. Download the latest Docker Compose binary from the official repository.
2. Apply execute permissions to the downloaded binary.
3. Create a symbolic link to make docker-compose globally accessible.
4. Verify the installation by checking the Docker Compose version.

Demonstrate the successful installation by showing the output of the **docker-compose --version** command.

**Q2. NGINX Web Server with Docker Compose**

**Question:**

1. Write a docker-compose.yml file to:
   * Use the nginx:latest image.
   * Map port 8080 on the host to port 80 in the container.
   * Bind the ./html directory on the host to /usr/share/nginx/html/ in the container to serve a custom HTML file.
2. Create a custom HTML file (index.html) in the ./html directory.
3. Start the service using Docker Compose and verify:
   * Access the NGINX web server at http://<host\_ip>:8080.
   * Show the running containers using docker ps.
4. Stop the service and remove the containers using docker-compose down.

Demonstrate the process and the expected results during the lab session.

**Q3. Live Update with NGINX and Docker Compose**

**Question:**

1. Write a docker-compose.yml file to:
   * Use the nginx:latest image.
   * Map port 8080 on the host to port 80 in the container.
   * Bind the ./html directory on the host to /usr/share/nginx/html/ in the container to serve custom HTML content.
2. Create a custom HTML file (index.html) in the ./html directory.
3. Start the service using Docker Compose and verify:
   * Access the NGINX web server from outside the virtual machine using a browser or tool like curl at http://<VM\_IP>:8080.
4. While the container is running, update the index.html file.
5. Refresh your browser to confirm that the changes are reflected immediately without restarting the service or the container.
6. Stop the service and clean up the containers using **docker-compose down**.

**Expected Outcome:**

* Verify that the initial and updated HTML content is accessible from outside the virtual machine, and live updates are visible without restarting the container.

**Q4. Dynamic Scaling with Docker Compose**

**Question:**

1. Write a docker-compose.yml file to:
   * Define an nginx service using the nginx:latest image.
   * Bind the ./html directory on the host to /usr/share/nginx/html/ in the container.
   * Expose port 80 for internal communication (no direct binding to the host).
2. Perform the following scaling operations:
   * **Scale Up**: Start the services and scale the NGINX service to 3 instances.
     + Use docker ps -a to confirm that 3 NGINX containers are running.
   * **Scale Down**: Reduce the number of NGINX instances to 1.
     + Use docker ps -a to confirm that only 1 NGINX container is running.
3. Stop all services and clean up the environment.

**Expected Outcome:**

* During scaling up and down, the correct number of NGINX instances are reflected in docker ps -a.

**Q5. Simple Microservices Implementation with Docker Compose**

**Question:**

1. Write a docker-compose.yml file to implement a microservices architecture with the following services:
   * **frontend**:
     + Build the service from the frontend directory.
     + Map port 5000 on the host to port 5000 in the container.
     + Define dependencies on the addition and multiplication services.
   * **addition**:
     + Build the service from the addition directory.
   * **multiplication**:
     + Build the service from the multiplication directory.
2. Define a custom network named app-network using the bridge driver to allow all services to communicate internally.
3. Start all the services using Docker Compose and verify:
   * Use docker ps to confirm that all three services (frontend, addition, and multiplication) are running.
   * Access the frontend service at http://localhost:5000 and confirm that it interacts with the addition and multiplication services.
4. Stop all services and clean up the environment.

**Expected Outcome:**

* All three services (frontend, addition, and multiplication) are running and can communicate internally using the app-network.
* The frontend service is accessible at http://localhost:5000.

\*\*\*\*\*\*\*\*\*\*