How can you help CodeCrafters Inc. manage their three servers (Alpha, Beta, Gamma) remotely and deploy a web app using Docker and Nginx?

Let’s dive into the task:

Setting up Virtual Machines:

You need to create three virtual machines named Server, Client1, and Client2.

1.Steps to create VM:

1. Open VirtualBox and click on "New" to create a new virtual machine.
2. Enter a name for the server and choose the latest version of the ISO image of ubuntu downloaded from the internet. Click "Next".
3. Optionally, select or skip unattended installation options as per your preference, then click "Next".
4. Set the base memory (RAM) to 2048 MB and choose the number of processors (1 or as desired). Click "Next".
5. Leave virtual hardware settings as default (adjust if needed), then click "Next".
6. Review all the details and click "Finish" to create your VM. Repeat these steps for each additional VM.

By following these steps, you'll successfully set up multiple virtual machines in VirtualBox.

2. Now you need to connect all VM’s to the same Network for seamless communication.

I want our virtual machines (VMs) to have internet access along with the ability to communicate with each other so am using a combination of NAT (Network Address Translation) and host-only networking.

How to setup NAT and Host-only Networking:

- Open VirtualBox, click on "File", then go to "Tools" and select "Network Manager".

- In the Network Manager window, navigate to "Host-only Networks" and click on "Create". This creates a new host-only network.
- Ensure "Configure adapter automatically" is selected and click "Apply".
- Still in Network Manager, go to "Nat Networks" and click on "Create". Configure general options as needed and click "Apply".
- Now, for each VM:

1. Go to VM Settings, then navigate to "Network".
2. In Adapter 1, enable the network adapter and select "NAT Network" from the "Attached to" dropdown.
3. In Adapter 2, enable the network adapter and select "Host-only Adapter" from the "Attached to" dropdown.

- Adjust display settings if necessary:

1. Go to VM Settings, then navigate to "Display".
2. Set "Video Memory" to 128MB, "Monitor Count" to 1, and select "VBoxSVGA" as the "Graphics Controller".

- Ensure the VM is powered off to apply any changes made in settings. Start the VM after making changes to reflect them.
- Repeat these steps for each of the other two VMs to ensure they are also connected to both the NAT network and the host-only network for seamless communication.

By combining these two types of networks, we can achieve a setup where VMs can interact locally (using the host-only network) and also access the internet (via NAT). This configuration is commonly used in virtualization environments to provide both isolation and internet connectivity to VM.

Generating SSH Key Pairs and Authentication

In My VM's I don't have ssh installed, you can verify that using

Systemctl status ssh

Steps to install ssh:

To install SSH (Secure Shell) on a Linux system, typically Ubuntu or Debian-based distributions, follow these steps:

1.It's good practice to update the package lists to ensure you are installing the latest available version of SSH:

sudo apt update

2.If there are any packages which need to be upgrade, you can use below command

sudo apt upgrade

3. **Install SSH Server**: Install the SSH server package (openssh-server).

sudo apt install openssh-server

4. **Verify SSH Service Status**: Once installed, the SSH service should start automatically. You can verify its status to ensure it's running:

sudo systemctl status ssh

This command will show you if SSH is active and running. If it is not running, you can start it by using **sudo systemctl start ssh.**

5. **Enable SSH Service (if necessary)**: If SSH is not enabled to start automatically at boot, you can enable it using:

sudo systemctl enable ssh

6. SSH is successfully set up on your VM! Now you can securely connect to your VM remotely using SSH.

This web application, powered by Docker and Nginx, will be spread across the client1and client2 servers, and the server will be the mastermind orchestrating the entire operation.

1. **Generate SSH Keys on the Server**:
    - Open a terminal on your server.
    - Use the command ssh-keygen to generate SSH keys.
    - You'll be prompted to choose a location for the keys (default is ~/.ssh/id_rsa) and optionally set a passphrase.
2. **Edit the Hosts File on the Server**:
    - Use sudo nano /etc/hosts to edit the hosts file on your server.
    - Add entries for client1 and client2 in the format client_ip_address client_hostname.

    - Save the changes (Ctrl+O, Enter) and exit (Ctrl+X) the editor.
3. **Copy Public Key to Client1 and Client2 VMs**:
    - Use ssh-copy-id to copy the public key (id_rsa.pub) from the server to client1 and client2.
    - Replace username with the actual username on each VM and client_ip_address with the IP address of each VM.
    - Example commands:

      ```
      bash
      Copy code
      ssh-copy-id -i ~/.ssh/id_rsa.pub username@client1_ip_address
      ssh-copy-id -i ~/.ssh/id_rsa.pub username@client2_ip_address
      ```

4. **SSH into Client1 (and Client2)**:
    - Use SSH to connect to client1 (and client2) from the server without a password prompt, if SSH keys are properly set up.
    - Example command:

      ```
      bash
      Copy code
      ssh username@client1
      ```

    - Replace username with the actual username on each VM.

These steps will help you set up SSH keys and configure access between your server and the client1 and client2 VMs. Ensure SSH is installed and running on all machines, and that you have the necessary permissions to edit files and copy keys.

4.Write a Bash script that remotely executes commands on the client1 and client2 VMs from the server.

```
#!/bin/bash



# Define your VM's hostname or IP Address

client1="10.0.2.4"

client2="10.0.2.6"
```

```
# Function to execute commands on a remote machine via SSH
execute_remote_commands() {
    local user="$1"
    local host="$2"

    echo "Executing commands on $user@$host..."

    # Execute 'uptime' command
    echo "Uptime:"
    ssh "$user@$host" uptime
    if [ $? -ne 0 ]; then
        echo "Failed to execute 'uptime' on $user@$host"
    fi
    echo ""

    # Execute 'uname -a' command
    echo "System Information:"
    ssh "$user@$host" uname -a
    if [ $? -ne 0 ]; then
        echo "Failed to execute 'uname -a' on $user@$host"
    fi
    echo ""


    # Execute 'date' command
    echo "Current Date and Time:"
    ssh "$user@$host" date
    if [ $? -ne 0 ]; then
        echo "Failed to execute 'date' on $user@$host"
    fi
    echo ""
}


# Execute commands on client1 as user vardhan
execute_remote_commands "vardhan" "$client1"
echo ""


# Execute commands on client2 as user dhanush
execute_remote_commands "dhanush" "$client2"
```

5. **Use scp to securely transfer files between the VMs.**

To securely transfer files between virtual machines (VMs) using scp (Secure Copy Protocol), follow these steps:

1. **Create a File with Random Content**: First, create a file named important_information on the source server VM with random content. You can do this using the following commands:

touch important_information

vi important_information

Inside vi, press I for insert mode, paste your random content, then press Esc followed by :wq to save and quit.

2. **Transfer the File Using scp**: Use scp to securely copy the file from the source VM to the destination VM. Replace /home/mkanisetty/scripts/important_information with the actual path to your file on the source VM, and dhanush@10.0.2.6:/home/dhanush with the destination address and path:

scp /home/mkanisetty/scripts/important_information dhanush@10.0.2.6:/home/dhanush

By using the above command, you can securely transfer the file between VM's. Automating file transfers ensures consistency by reducing errors, saves time with minimal supervision needed, and enhances reliability through error handling. It also provides detailed logs for tracking and integrates seamlessly into workflows for efficient data management. These benefits make automation crucial even for one-time tasks like database backups.

3. **Automate the File Transfer**:

create a Bash script (secure_transfer.sh) with the scp command and execute it automatically using a scheduler:

So below is the script

```
#!/bin/bash

# Define variables for source and destinations

source_file="/home/mkanisetty/scripts/important_information"


# Destination 1

destination1_user="dhanush"

destination1_ip="10.0.2.6"

destination1_dir="/home/dhanush"
```

```
# Destination 2
destination2_user="vardhan"
destination2_ip="10.0.2.4"
destination2_dir="/home/vardhan"

# Function to perform SCP transfer
perform_scp_transfer() {
  local user="$1"
  local ip="$2"
  local dir="$3"

  echo "Transferring file to $user@$ip:$dir..."
  scp "$source_file" "$user@$ip:$dir"

  # Check transfer status
  if [ $? -eq 0 ]; then
    echo "File transferred successfully to $user@$ip."
  else
    echo "Error: Failed to transfer file to $user@$ip."
  fi
}

# Perform SCP transfer to Destination 1
perform_scp_transfer "$destination1_user" "$destination1_ip"
"$destination1_dir"
```

Perform SCP transfer to Destination 2

perform_scp_transfer "$destination2_user" "$destination2_ip" "$destination2_dir"

Containerizing and Deploying the Web Application

1. **On the server VM, write a Dockerfile to containerize a simple web application.**
2. **Build the Docker image on the server VM.**
3. **Write the Bash script deploy_app.sh on client1 and client2 VMs to deploy the web application using Docker and Nginx.**

First you need to install docker in all the three VM's

Installing Docker on a Server VM

1. Update Package Index:

First, update the package index on your server VM:

sudo apt update

2. Install Dependencies:

Install the packages necessary to allow the use of Docker's repository over HTTPS:

sudo apt install apt-transport-https ca-certificates curl software-properties-common

3. Add Docker's Official GPG Key:

Add Docker's official GPG key to your system:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

4. Add Docker Repository:

Add the Docker repository to your APT sources:

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

5. Install Docker Engine:

Update the package index again, then install the latest version of Docker Engine and containerd:

sudo apt update

sudo apt install docker-ce docker-ce-cli containerd.io

6. Verify Docker Installation:

Check that Docker is installed correctly by running the hello-world container:

sudo docker run hello-world

This command downloads a test image and runs it in a container. If Docker is set up correctly, you'll see a message confirming that Docker is working.

7. Manage Docker as a Non-Root User (Optional):

If you want to avoid using sudo with Docker commands, add your user to the docker group:

sudo usermod -aG docker $USER

After executing this command, log out and log back in to apply the group membership.

8. Start Docker Service:

Start the Docker service if it's not already started:

sudo systemctl start docker

9. Enable Docker Service to Start on Boot (Optional):

To ensure Docker starts automatically when your server VM boots up, enable the service:

sudo systemctl enable docker

Success! Docker setup complete on your server VM. Repeat for the other two VMs

Any web application will work, but in my case, I'm using a Python Flask application

Prepare Your Flask Application

Create Flask Application Structure:

First, make sure you have a Flask application set up. Here's a simple example of a Flask application structure:

```
/flask-app

  |-- app.py

  |-- requirements.txt

  |-- /static

  |-- /templates
```

Create the directory and navigate into it:

Open your terminal or command prompt and execute the following commands one by one:

mkdir flask-app

cd flask-app

```
# Create app.py file

touch app.py

# Create requirements.txt file

touch requirements.txt

# List files in the current directory

ls
```

After running these commands, you will have app.py and requirements.txt files created in your current directory, and you can verify their existence by the output of ls.

Edit app.py with Nano:

```
nano app.py
```

Paste the following code:

```
from flask import Flask, render_template



app = Flask(__name__)



@app.route('/')

def home():

    return render_template('index.html')



if __name__ == '__main__':

    app.run(debug=True, host='0.0.0.0'
```

Save and exit Nano (Ctrl+O, Enter, Ctrl+X).

Edit requirements.txt with Nano:

Add the line Flask and save (Ctrl+O, Enter, Ctrl+X).

 Create Directories: Navigate to your flask-app directory and execute the following commands to create the necessary directories:

```
mkdir static
mkdir templates
```

This will create static and templates directories inside your flask-app directory.

 Create index.html in Templates Directory: Navigate into the templates directory:

```
cd templates
```

Create a new file named index.html and edit it to contain the following content:

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Flask App</title>
</head>
<body>
   <h1>Welcome to my Flask App! </h1>
   <p>This is a simple Flask application. </p>
</body>
</html>
```

Save and exit the editor.

 Summary of Files and Directories:

- **app.py**: Contains your Flask application code.

- **requirements.txt**: Lists all Python packages your Flask application depends on.
- **/static**: Directory for static files (e.g., CSS, JavaScript).
- **/templates**: Directory for HTML templates (e.g., index.html).

Dockerfile for Containerizing the Flask Application

Now, let's create a Dockerfile in the flask-app directory to build a Docker image for your Flask application.

Create a file named Dockerfile (no file extension) in the flask-app directory with the following content:

```
# Base image

FROM python:3.9-slim


# Set working directory within the container

WORKDIR /app


# Copy and install Python dependencies

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt


# Copy the Flask application code to the container

COPY . .


# Expose port 5000 (Flask default)

EXPOSE 5000
```

# Command to run the Flask application

CMD ["python", "app.py"]

Build the Docker Image: Run the following command to build the Docker image. Make sure to replace flask-app with your actual directory name if different:

docker build -t my-flask-app-image ./flask-app

- docker build: This command builds a Docker image from a Dockerfile.
- -t my-flask-app-image: Tags the built image with the name my-flask-app-image.
- ./flask-app: Specifies the path to the directory containing your Dockerfile.

This process will install all dependencies and set up your Flask application inside the Docker image.

Here's a detailed step-by-step guide on how to push a Docker image to Docker Hub:

Step 1: Docker Login

First, log in to Docker Hub using your Docker Hub username and password. Open your terminal or command prompt and enter:

docker login

You will be prompted to enter your Docker Hub username and password. After successful authentication, you'll see a message confirming "Login Succeeded".

Step 2: Tag Your Docker Image

Before pushing your Docker image to Docker Hub, you need to tag it with your Docker Hub username and repository name.

Assuming you have a Docker image named my-flask-app-image and your Docker Hub username is mkanisetty, tag the image like this:

docker tag my-flask-app-image mkanisetty/my-flask-app-image

This command associates your local Docker image (my-flask-app-image) with the repository on Docker Hub (mkanisetty/my-flask-app-image).

Step 3: Push Your Docker Image

Once your image is tagged correctly, you can push it to Docker Hub using the docker push command:

```
docker push mkanisetty/my-flask-app-image
```

This command uploads your tagged Docker image (mkanisetty/my-flask-app-image) to Docker Hub's registry. The push process may take some time depending on the size of your image and your internet connection speed.

Step 4: Verify the Push

After the push completes successfully, you can verify that your image is available on Docker Hub by visiting your repository page on Docker Hub's website. You can find it at https://hub.docker.com/r/mkanisetty/my-flask-app-image.

Make sure to replace mkanisetty with your actual Docker Hub username and adjust the image name (my-flask-app-image) accordingly based on your local Docker image's name.

Pushing your Docker image to Docker Hub provides a centralized location for your application's container image, simplifying deployment, version control, and collaboration among team members. It's an essential step in Docker-based application development and deployment workflows.

Configure Nginx as Reverse Proxy

1. Install Nginx

First, ensure Nginx is installed on the VM where you plan to set it up as a reverse proxy, am installing it in my server VM. Here's how to install Nginx:

```
sudo apt update

sudo apt install nginx
```

After installing Nginx, you need to configure it to proxy incoming requests to your Flask application running in Docker containers.

Create a Configuration File:

Navigate to Nginx's configuration directory (commonly /etc/nginx/sites-available) and create a new configuration file specific to your Flask application (e.g., my-flask-app).

```
sudo nano /etc/nginx/sites-available/my-flask-app
```

```
server {

   listen 80;  # Listen for incoming HTTP requests on port 80

   server_name 10.0.2.5;  # Replace with your server VM's IP address or domain name



   location /app1/ {

       proxy_pass http://10.0.2.4:5000;  # Forward requests to Client1's Flask application on port 5000

       proxy_set_header Host $host;  # Preserve the original Host header

       proxy_set_header X-Real-IP $remote_addr;  # Pass client's IP as X-Real-IP header

       proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  # Append client's IP to X-Forwarded-For header

       proxy_set_header X-Forwarded-Proto $scheme;  # Set X-Forwarded-Proto to the request's protocol (HTTP or HTTPS)

   }



   location /app2/ {

       proxy_pass http://10.0.2.5:5000;  # Forward requests to Client2's Flask application on port 5000
```

```
        proxy_set_header Host $host;  # Preserve the original Host header

        proxy_set_header X-Real-IP $remote_addr;  # Pass client's IP as X-Real-IP header

        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  # Append client's IP to X-Forwarded-For header

        proxy_set_header X-Forwarded-Proto $scheme;  # Set X-Forwarded-Proto to the request's protocol (HTTP or HTTPS)

    }

}
```

Enable the Configuration

Create a symbolic link from sites-available to sites-enabled to enable your new configuration:

```
sudo ln -s /etc/nginx/sites-available/my-flask-app /etc/nginx/sites-enabled/
```

Test and Restart Nginx

Check the Nginx configuration syntax for errors:

```
sudo nginx -t
```

If the syntax is OK, restart Nginx to apply the new configuration:

```
sudo systemctl restart nginx
```

SSH into Client VMs:

- Connect to your Client VMs (client1 and client2) where you want to deploy the Docker container.
- Execute the following commands on each Client VM (client1 and client2):

  ```
  ssh vardhan@10.0.2.4 # Replace client1_ip_address with your client1's IP address
  ```

Create and Edit deploy_app.sh Script:

- In the home directory (or any directory you prefer) of the Vardhan and dhanush users on each Client VM (client1 and client2), create and edit the deploy_app.sh script:

  ```
  nano deploy_app.sh
  ```

  Paste the following code into the file

  ```
  #!/bin/bash

  # Pull the latest version of the Docker image from Docker Hub (if applicable)
  docker pull my-web-app-image

  # Stop and remove the existing container if it exists
  docker stop my-web-app-container || true
  docker rm my-web-app-container || true

  # Run the Docker container
  docker run -d --name my-web-app-container -p 80:80 my-web-app-image
  ```

  Save and exit the text editor (nano or another editor).

- Set execute permissions for the script:

  chmod +x deploy_app.sh or chmod 700 deploy_app.sh

Execute the Deployment Script:

- Run the deploy_app.sh script on each Client VM (client1 and client2) to deploy your Dockerized web application:

  ```
  ./deploy_app.sh
  ```

  This script will pull the latest Docker image (my-web-app-image), stop/remove any existing container (my-web-app-container), and start a new container with your updated image, making your web application available on port 80 of each Client VM (client1 and client2).

  Create a Python virtual environment (recommended but optional):

```
python -m venv venv
```

This command creates a virtual environment named venv inside your flask-app directory. Virtual environments are good practice because they isolate dependencies for your project.

Activate the virtual environment:

```
source venv/bin/activate
```

Install Flask (assuming you have Python installed):

```
pip install Flask
```

Run the Flask application:

- Make sure you are in the flask-app directory where app.py is located.
- Run the application by executing:

```
python app.py
```

You should see output similar to:

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

This means your Flask application is running locally.

Access Your Flask Application

You should be able to access your Flask application by navigating to http:// 10.0.2.5:5000 in a web browser (replace 5000 with your actual port if different) and 10.0.2.5 is my server Ip.