4-Nov-2024

Training Day - 72 Report:

Introduction of Docker

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications using containerization technology. It provides a standardized unit of software, known as a container, that packages an application and all its dependencies, enabling it to run consistently across various computing environments.

Key Concepts of Docker

1. Containerization

Definition: Containerization is the process of encapsulating an application and its dependencies into a container. This allows applications to run in isolated environments without interfering with each other.

Isolation: Each container operates independently, sharing the host OS kernel but maintaining its own file system, libraries, and configuration files. This isolation helps prevent conflicts between applications.

2. Docker Images

Definition: A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the application code, libraries, dependencies, and runtime.

Layered Architecture: Docker images are built in layers, where each layer represents a set of file changes. This layered structure allows for efficient storage and sharing of common layers across different images.

Dockerfile: Images are created from a Dockerfile, a text document that contains a series of commands and instructions to assemble the image. The Dockerfile specifies the base image, application code, dependencies, and configuration.

3. Docker Containers

Definition: A Docker container is a running instance of a Docker image. It is a lightweight, standalone environment where the application runs.

Lifecycle: Containers can be created, started, stopped, and removed. They are ephemeral by nature, meaning they can be easily destroyed and recreated.

Resource Efficiency: Containers share the host OS kernel, making them more efficient in terms of resource usage compared to traditional virtual machines (VMs).

4. Docker Engine

Definition: The Docker Engine is the core component of Docker that enables users to create, run, and manage containers. It consists of a server (the Docker daemon), a REST API, and a command-line interface (CLI).

Architecture: The Docker daemon manages the containers and images on the host system, while the Docker CLI allows users to interact with the Docker daemon through commands.

5. Docker Hub

Definition: Docker Hub is a cloud-based registry service for sharing and managing Docker images. It allows users to store and distribute images publicly or privately.

Pre-built Images: Docker Hub hosts a wide range of pre-built images, making it easy for developers to find and use existing software packages.

Advantages of Using Docker

1. **Consistency Across Environments:** Docker ensures that applications run the same way in development, testing, and production environments, reducing "it works on my machine" issues.

2. **Isolation:** Each container runs in its own environment, preventing conflicts between applications and making it easier to manage dependencies.

3. **Portability:** Docker containers can run on any system that supports Docker, allowing for seamless movement of applications across different environments.

4. **Scalability:** Docker makes it easy to scale applications by adding or removing containers based on demand. This is particularly useful in microservices architectures.

5. **Resource Efficiency**: Containers are lightweight and can start quickly, leading to better resource utilization compared to traditional virtual machines.

6. **Rapid Deployment:** Docker allows for faster application deployment by enabling developers to package applications and their dependencies together.

7. **Microservices Support:** Docker is well-suited for microservices architectures, where applications are broken down into smaller, independent services that can be developed, deployed, and scaled independently.

8. **Version Control:** Docker images can be versioned, allowing developers to track changes, roll back to previous versions, and maintain multiple versions of an application.

9. **Integration with CI/CD:** Docker integrates well with Continuous Integration and Continuous Deployment (CI/CD) pipelines, enabling automated testing and

deployment of applications.

10. **Community and Ecosystem:** Docker has a large community and a rich ecosystem of tools, libraries, and extensions that enhance development workflows.

Use Cases for Docker

1) **Development Environments:** Developers can create consistent environments for their applications, reducing setup time and conflicts.

2) **Microservices:** Docker facilitates the development and deployment of microservices by allowing each service to run in its own container.

3) **Continuous Integration/Continuous Deployment:** Docker enables automated testing and deployment processes in CI/CD pipelines.

4) **Cloud Deployments:** Docker containers can be easily deployed on cloud platforms, making it an ideal choice for cloud-native applications.

5) **Legacy Application Migration:** Docker can be used to containerize legacy applications, making them easier to deploy and manage in modern environments.

Conclusion

Docker is a powerful tool that has transformed how applications are developed, deployed, and managed. Its containerization technology provides a consistent, efficient, and scalable

5-Nov-2024

Training Day - 73 Report:

Containers and Docker

1. **Through Docker Container deploy a project:**

   Docker containers are lightweight, standalone packages of software that include everything needed to run an application (code, libraries, dependencies, etc.). This makes deploying projects fast and consistent across different environments.

2. **Make folders and put Vagrant file:**

   - Vagrant is a tool for managing virtual machine environments.
   - A Vagrantfile is a configuration file that defines the setup for the virtual environment, such as specifying the base operating system, networking, and provisioning tools like Docker.

3. **Search docker.com:**

   - Visit Docker's official website (https://www.docker.com/) to download and install Docker, and explore its features, documentation, and tutorials.

4. **Install Docker:**

   - Docker installation steps include setting up Docker's repository, adding its GPG key, and installing Docker packages (as shown in your earlier script).

5. **Vagrant up:**

   - The vagrant up command initializes and starts the virtual environment defined in the Vagrantfile. If Docker provisioning is included, Docker will be installed and configured within this virtual environment.

Docker Build Cloud:

- Docker Build Cloud is likely referring to tools like **Docker Build** or **Docker Hub**:
  - **Docker Build**: Command to build Docker images locally or in CI (Continuous Integration) environments.
  - **Docker Hub**: A cloud-based registry service that allows you to host and share container images.
- These tools make it easier to create, store, and deploy Docker images.

Commands Explanation:

1. **vagrant ssh:**
   - Connects to the Vagrant-managed virtual machine using SSH. Once connected, you can run commands inside the VM.

2. **systemctl status docker:**
   - Checks the current status of the Docker service (active, inactive, or failed).

```
service docker status
```

3. **sudo -i:**
   - Switches to the root user for elevated privileges, necessary to execute certain Docker commands.

4. **docker images:**
   - Lists all Docker images currently stored on the system, including the image name, tag, and size.

```
root@ubuntu-jammy:~# docker images
REPOSITORY   TAG       IMAGE ID   CREATED   SIZE
```

5. **docker ps:**
   - Displays all currently running Docker containers.

```
root@ubuntu-jammy:~# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

6. **docker ps -a:**
   - Shows a list of all containers (both running and stopped).

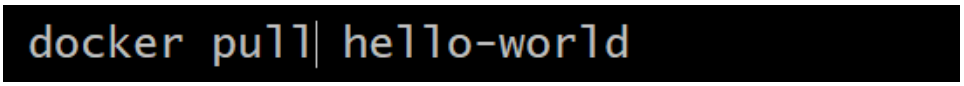
```
root@ubuntu-jammy:~# docker ps -a
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

7. **docker run hello-world:**
   - Runs the hello-world container, which verifies that Docker is installed and functioning correctly.

```
root@ubuntu-jammy:~# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:305243c734571da2d100c8c8b3c3167a098cab6049c9a5b066b6021a60fcb966
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Alternatively:

- **docker pull hello-world**: Downloads the hello-world image without running it.



6-Nov-2024

Training Day - 74 Report:

Commands and Explanations

1. **docker run --name web01 -p 9080:80 nginx:**

   - **docker run**: Starts a new container.
   - **--name web01**: Assigns the name web01 to the container for easy reference.
   - **-p 9080:80**: Maps port 9080 on the host machine to port 80 inside the container. This allows accessing the Nginx server running in the container via http://localhost:9080.
   - **nginx**: Specifies the Nginx image to run.

   Additional Note:

   - If the -d flag is used, the container will run in the background.

```
root@ubuntu-jammy:~# docker run --name web01 -d -p 9080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
2d429b9e73a6: Pull complete
9b1039c85176: Pull complete
9ad567d3b8a2: Pull complete
773c63cd62e4: Pull complete
1d2712910bdf: Pull complete
4b0adc47c460: Pull complete
171eebbdf235: Pull complete
Digest: sha256:bc5eac5eafc581aeda3008b4b1f07ebba230de2f27d47767129a6a905c84f470
Status: Downloaded newer image for nginx:latest
4ae207f45c5a502b01f59e89a555d687fb3a159d9cfc5fba2b0e4d8684c4f5d7
root@ubuntu-jammy:~#
```

2. **docker inspect web01:**

   - Provides detailed information about the web01 container, such as its configuration, network settings, and status.

```
root@ubuntu-jammy:~# docker inspect web01
[
    {
        "Id": "4ae207f45c5a502b01f59e89a555d687fb3a159d9cfc5fba2b0e4d8684c4f5d7",
        "Created": "2024-11-16T09:20:29.463295072Z",
        "Path": "/docker-entrypoint.sh",
        "Args": [
```

3. **curl http://172.17.0.2:**

   - Sends an HTTP request to the container's IP address (172.17.0.2 is a typical IP for Docker containers using the default bridge network). This checks if the Nginx server is running and serving content.

```
root@ubuntu-jammy:~# curl 172.17.0.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

4. **ip addr show:**

    - Displays network interfaces and their associated IP addresses on the host machine. Useful for verifying network configurations.

```
root@ubuntu-jammy:~# ip a s
```

5. **mkdir images:**

    - Creates a new directory named images. Typically used for organizing Docker-related files (like Dockerfiles or scripts).

6. **cd images:**

    - Changes the working directory to images.

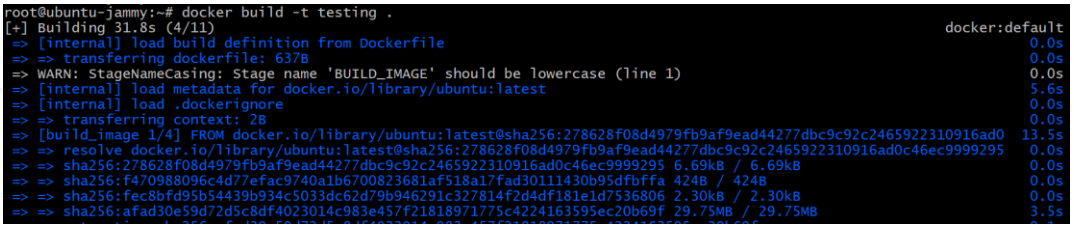
7. **vim Dockerfile:**

    - Opens or creates a Dockerfile using the vim text editor.
    - The Dockerfile contains instructions to build a custom Docker image.

```
FROM ubuntu:latest AS BUILD_IMAGE
RUN apt update && apt install wget unzip -y
RUN wget https://www.tooplate.com/zip-templates/2128_tween_agency.zip
RUN unzip 2128_tween_agency.zip && cd 2128_tween_agency && tar -czf tween.tgz * && mv tween.tgz /root/tween.tgz

FROM ubuntu:latest
LABEL "project"="Marketing"
ENV DEBIAN_FRONTEND=noninteractive

RUN apt update && apt install apache2 git wget -y
COPY --from=BUILD_IMAGE /root/tween.tgz /var/www/html/
RUN cd /var/www/html/ && tar xzf tween.tgz
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
VOLUME /var/log/apache2
WORKDIR /var/www/html/
EXPOSE 80
```

8. **docker build -t testing .:**

   - **docker build**: Builds a Docker image from the current directory (denoted by .) containing the Dockerfile.
   - **-t testing**: Tags the built image with the name testing.



9. **docker run -p 80:80 testing:**

   - Runs a container from the testing image and maps port 80 on the host to port 80 in the container. This exposes the application in the container to the host machine on port 80.

```
docker run -p 80:80 testing
```

10. **docker stop web01:**

    - Stops the web01 container gracefully.

```
root@ubuntu-jammy:~# docker stop web01
web01
```

11. **docker rm web01:**

    - Removes the stopped container named web01. Containers must be stopped before they can be removed.

```
root@ubuntu-jammy:~# docker rm web01
web01
```

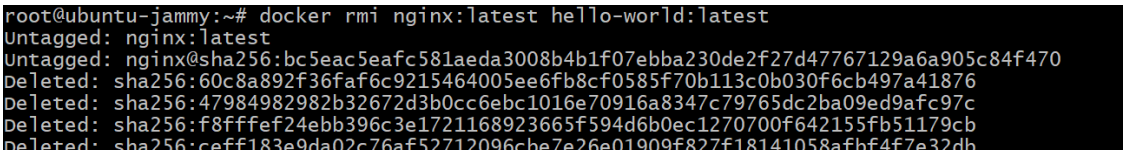
12. **docker images:**

    - Lists all locally available Docker images. Displays repository name, tag, image ID, creation date, and size.

```
root@ubuntu-jammy:~# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
nginx         latest    60c8a892f36f   6 weeks ago     192MB
hello-world   latest    d2c94e258dcb   18 months ago   13.3kB
```

13. **docker rmi <image_id>:**

    - Removes a Docker image by its ID. This helps free up space by deleting unused images.



Summary of Workflow:

1. Start by running a container (docker run) and mapping ports for accessibility.

2. Use docker inspect and curl to verify that the container is running and accessible.

3. Use mkdir, cd, and vim to create a directory and write a Dockerfile for custom image building.

4. Build a custom image with docker build, run it with docker run, and expose the application via port mapping.

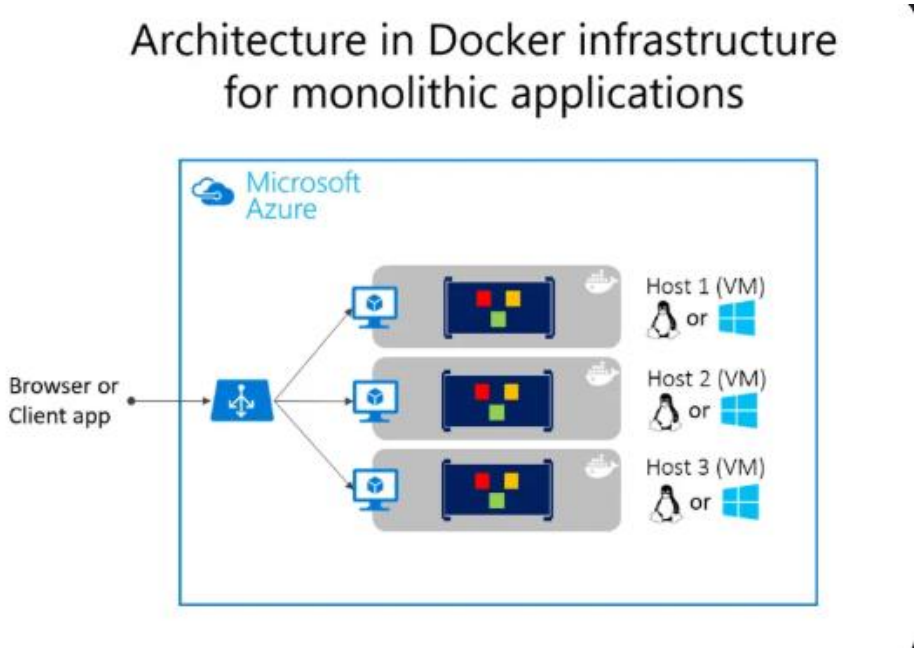
5. Stop and remove containers/images (docker stop, docker rm, docker rmi) as needed to manage resources.

7-Aug-2024

Training Day - 75 Report:

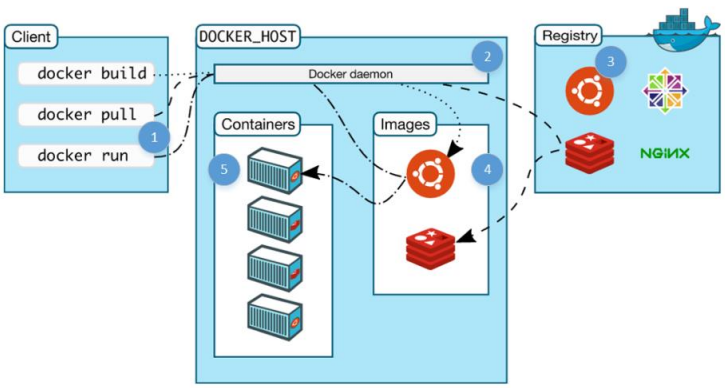
Monolithic Architecture in Docker:

- The entire application is packaged into a single container, which contains all components like UI, business logic, and database access.
- Scaling means replicating the entire container.
- Simpler to set up in Docker but harder to manage and update as the application grows.



Microservices Architecture in Docker:

- Each service (e.g., user service, payment service, inventory service) is deployed in its own container.
- Services communicate with each other using APIs (e.g., REST or gRPC).
- Allows independent scaling, deployment, and updates of each service.



Comparison:

Aspect	Monolithic Architecture	Microservices Architecture
Structure	All components (UI, business logic, DB) in one container.	Each service runs in its own container.
Deployment	Single container deployed as a whole application.	Multiple containers deployed for individual services.
Scaling	Entire application must be scaled.	Specific services can be scaled independently.
Fault Isolation	Failure in one part can affect the entire application.	Failures are isolated to specific services.
Development	Requires a tightly coupled development process.	Allows independent development by different teams.
Flexibility	Hard to change or update individual components.	Easier to update and redeploy specific services.
Resource Utilization	Resource-heavy due to one large container.	Efficient, as containers run only necessary services.
Technology Stack	Limited to a single technology stack for the entire app.	Different services can use different technology stacks.
Example in Docker	One `Dockerfile` for the whole app, e.g., monolithic.jar.	Multiple `Dockerfile`s, e.g., `user-service`, `auth-service`.
Docker Orchestration	Simpler, as there is only one container to manage.	Requires orchestration tools (e.g., Docker Compose, Kubernetes) to manage many containers.

Why Microservices Are More Popular Now:

1. **Scalability:**
   - Modern applications need to scale specific parts of the system independently. For instance, an e-commerce platform might scale its product catalog service during a sale, without scaling the user authentication service.
   - Microservices, combined with tools like Docker and Kubernetes, make independent scaling efficient.
2. **Flexibility in Technology Stack:**
   - Microservices allow developers to use different languages and technologies for different services. For instance, a company could use Python for machine learning services, Node.js for APIs, and Java for backend processing.
3. **Cloud-Native Applications:**
   - With the rise of cloud platforms (AWS, Azure, GCP), applications are designed to be **cloud-native**. Microservices are better suited to this architecture, leveraging containerization and orchestration tools for deployment.

4. **DevOps and CI/CD:**

    - DevOps practices and Continuous Integration/Continuous Deployment pipelines align well with microservices, enabling rapid updates to specific services without affecting the entire application.

5. **Fault Tolerance:**

    - Microservices provide better fault isolation. If one service fails, it doesn't bring down the entire system.

6. **Containerization with Docker:**

    - Tools like Docker and Kubernetes have made deploying and managing microservices much easier, allowing businesses to fully embrace the microservices approach.

Where Monolithic Is Still Relevant:

While microservices dominate modern architectures, **monolithic architecture** is still used in certain scenarios:

1. **Small Applications or Startups**:

    - For simple applications with limited scope and resources, a monolithic approach is faster and easier to develop and deploy.

2. **Legacy Systems:**

    - Many organizations still run legacy systems built on monolithic architecture, as transitioning to microservices can be costly and time-consuming.

3. **Teams with Limited Expertise:**

    - Small teams or those without expertise in managing distributed systems might prefer monoliths for simplicity.

Trends in Industry Usage:

1. **Microservices + Docker**:

    - Companies like Netflix, Amazon, and Uber rely heavily on microservices and Docker to run their scalable and distributed systems.

2. **Hybrid Approach:**

   - Some organizations adopt a **modular monolith** as a middle ground, where they keep the system unified but use Docker to package different modules for easier management.

3. **Kubernetes and Orchestration:**

   - With Kubernetes becoming the standard for container orchestration, microservices are easier to manage and deploy, further accelerating their adoption.

8-Aug-2024

Training Day - 76 Report:

Micro services Using Docker

Project: E-Mart Project Deploy Using Docker

1. Steps to Deploy

- **Vagrant up**:
  - Likely the command to initialize and provision a Vagrant virtual machine.
- **Vagrant ssh → sudo -i**:
  - Connect to the Vagrant virtual machine using SSH and switch to the root user with sudo -i.
- **mkdir compose**:
  - Create a directory named compose.

```
mkdir compose
```

- **cd compose**:
  - Change into the newly created compose directory.

```
cd compose/
```

- **touch docker-compose.yml**:
  - Create an empty docker-compose.yml file.

```
touch docker.compose.yml
```

- **vim docker-compose.yml**:

- Open the

```yaml
version: '3.8'
services:
  vprodb:
    image: vprocontainers/vprofiledb
    ports:
      - "3306:3306"
    volumes:
      - vprodbdata:/var/lib/mysql
    environment:
      - MYSQL_ROOT_PASSWORD=vprodbpass

  vprocache01:
    image: memcached
    ports:
      - "11211:11211"

  vpromq01:
    image: rabbitmq
    ports:
      - "15672:15672"
    environment:
      - RABBITMQ_DEFAULT_USER=guest
      - RABBITMQ_DEFAULT_PASS=guest

  vproapp:
    image: vprocontainers/vprofileapp
    ports:
      - "8080:8080"
    volumes:
      - vproappdata:/usr/local/tomcat/webapps

  vproweb:
    image: vprocontainers/vprofileweb
    ports:
      - "80:80"
volumes:
  vprodbdata: {}
  vproappdata: {}
```

2. Docker Compose Commands

- **docker-compose up -d**:
    - Start all the services defined in the docker-compose.yml file in detached mode (-d means run in the background).

```
root@ubuntu-jammy:~/compose# docker-compose up -d
Creating network "compose_default" with the default driver
Creating volume "compose_vprodbdata" with default driver
Creating volume "compose_vproappdata" with default driver
Pulling vprodb (vprocontainers/vprofiledb:)...
latest: Pulling from vprocontainers/vprofiledb
49bb46380f8c: Extracting [=========>                                         ]  8.716MB/44.92MB
aab3066bbf8f: Download complete
d6eef8c26cf9: Download complete
0e908b1dcba2: Download complete
480c3912a2fd: Download complete
89a648ecb3cf: Download complete
6313eed00780: Downloading [==============================>                    ]  36.93MB/58.6MB
668fe2d98404: Download complete
d3f8a843b813: Downloading [===>                                               ]  4.315MB/56.57MB
c80ab9fc8db5: Download complete
1b8b6b073273: Waiting
11ba0b468a55: Waiting
```

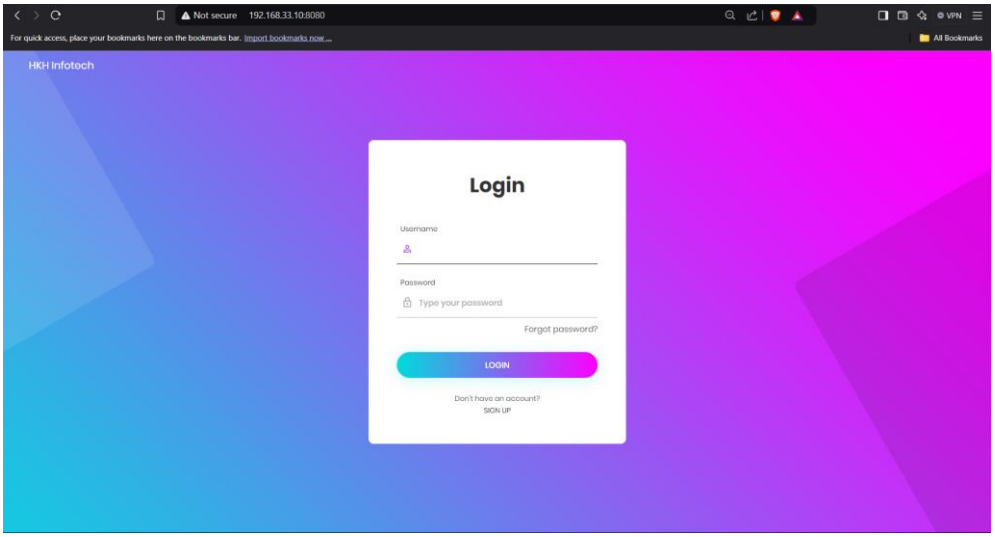
    - Note: "not a folder name, keyword" is likely a clarification to avoid confusion about the up command.

- **docker-compose ps**:
    - Show the status of the running containers/services.

```
root@ubuntu-jammy:~/compose# docker compose ps
WARN[0000] /root/compose/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
NAME                IMAGE                                COMMAND                  SERVICE       CREATED          STATUS          PORTS
compose_vproapp_1   vprocontainers/vprofileapp           "catalina.sh run"        vproapp       20 seconds ago   Up 18 seconds   0.0.0.0:8080->8080/tcp, :::8080->8080/tcp
compose_vprocache01_1   memcached                        "docker-entrypoint.s…"   vprocache01   20 seconds ago   Up 18 seconds   0.0.0.0:11211->11211/tcp, :::11211->11211/tc
compose_vprodb_1    vprocontainers/vprofiledb            "docker-entrypoint.s…"   vprodb        20 seconds ago   Up 18 seconds   0.0.0.0:3306->3306/tcp, :::3306->3306/tcp, 3
compose_vpromq01_1  rabbitmq                             "docker-entrypoint.s…"   vpromq01      20 seconds ago   Up 18 seconds   4369/tcp, 5671-5672/tcp, 15691-15692/tcp, 25
72->15672/tcp
compose_vproweb_1   vprocontainers/vprofileweb           "/docker-entrypoint.…"   vproweb       20 seconds ago   Up 18 seconds   0.0.0.0:80->80/tcp, :::80->80/tcp
```

- **ip addr show**:
    - Display the network interface details, which can be used to find the IP address of the machine.

- Project is live:



- **docker-compose down**:

  - Stop and remove all the containers, networks, and services started by docker-compose up.

```
root@ubuntu-jammy:~/compose# docker-compose down
Stopping compose_vprodb_1      ... done
Stopping compose_vproapp_1     ... done
Stopping compose_vproweb_1     ... done
Stopping compose_vpromq01_1    ... done
Stopping compose_vprocache01_1 ... done
Removing compose_vprodb_1      ... done
Removing compose_vproapp_1     ... done
Removing compose_vproweb_1     ... done
Removing compose_vpromq01_1    ... done
Removing compose_vprocache01_1 ... done
Removing network compose_default
```

3. General Notes

- The notes emphasize basic Docker Compose commands for managing the lifecycle of services.

- A focus on structured steps ensures proper configuration and deployment.