PW SKILLS | DevOps and Cloud Computing

Docker Fundamentals

Objective

- Understand Docker's architecture and how it differs from traditional virtual machines.
- Learn about Docker’s core components, including images, containers, and the Docker ecosystem.
- Install and set up Docker on Linux, macOS, and Windows.
- Work with Docker CLI commands for managing containers and images.
- Understand Docker networking and persistent storage for containerized applications.

PW SKILLS

The concept of containerization and why it is widely used in modern software development

Let’s see

Containerization is a modern software deployment method that packages an application along with its dependencies, such as libraries and configuration files, into a lightweight, self-contained unit called a container.

- These containers are portable and can run consistently across various environments, including different operating systems and cloud platforms.

Let’s see

Why Containerization is Widely Used:

Containerization has become integral to modern software development due to the following benefits:

1. Portability
2. Resource Efficiency
3. Isolation
4. Scalability
5. Agility
6. Fault Tolerance
7. Security

PW SKILLS

Compare Virtual Machines (VMs) vs. Containers

Let’s compare

Aspect	Virtual Machines (VMs)	Containers
Architecture	Each VM runs a full guest operating system on top of a hypervisor.	Containers share the host OS kernel but isolate applications and dependencies in user space.
Resource Utilization	High resource usage as each VM requires its own OS, leading to significant overhead.	Lightweight, sharing the host OS kernel, resulting in efficient resource usage.
Performance	Slower due to the overhead of running full OS instances; higher latency and startup times.	Faster startup times (seconds) with lower latency and better overall performance.

Let’s compare

Isolation	Strong isolation as each VM operates independently with its own OS and hardware emulation.	Weaker isolation since containers share the host OS kernel, though still sufficient for many apps.
Portability	Less portable; VMs may require reconfiguration when moved across environments.	Highly portable; containers run consistently across different environments without modification.
Use Cases	Ideal for running multiple OSes on the same hardware, legacy applications, or secure environments.	Best suited for microservices, CI/CD pipelines, scalable cloud-native apps, and rapid deployments.

PW SKILLS

Introduce Docker's core components

Docker components

Docker is a containerization platform with several core components that work together to build, run, and manage containers. Here's a brief introduction to its main components:

1. **Docker Images:** These are read-only templates containing the instructions and dependencies needed to create containers. They serve as the blueprint for containerized applications.
2. **Containers:** Containers are running instances of Docker images. They provide isolated environments to execute applications, ensuring consistency across different systems.

Docker components

3. **Docker Daemon:** Also known as dockerd, this is a background process that manages Docker objects like images, containers, networks, and volumes. It listens for API requests and executes Docker commands.
4. **Docker Client:** The CLI tool (docker) used by users to interact with the Docker daemon. It sends commands (e.g., docker run) to the daemon, which performs the requested actions.
5. **Docker Hub:** A centralized registry where users can store, share, and retrieve Docker images. It serves as the default repository for pulling and pushing container images.

PW SKILLS

Explaining the Docker ecosystem and demonstrate pulling, building, and running Docker images

Let’s do it

Docker Ecosystem Overview

The Docker ecosystem is a comprehensive platform for building, shipping, and running containerized applications. It includes tools and components that streamline the container lifecycle, from development to deployment. Key elements include:

1. Docker Engine
2. Docker Hub
3. Docker Compose
4. Docker CLI
5. Docker Desktop

Let’s do it

Below is a step-by-step demonstration of basic Docker operations:

1. **Pulling an Image**

To download a pre-built image from Docker Hub:

```
docker pull nginx
```

This pulls the official Nginx image from Docker Hub.

Let’s do it

2. Building an Image

To create a custom image using a Dockerfile:

```
# Example Dockerfile
FROM ubuntu:latest
RUN apt-get update && apt-get install -y curl
CMD ["curl", "--version"]

# Build the image
docker build -t my-custom-image .
```

This creates an image named my-custom-image based on Ubuntu with curl installed.

Let’s do it

3. Running a Container

To run a container from an image:

```
docker run -d -p 8080:80 nginx
```

This starts an Nginx container in detached mode (-d) and maps port 8080 on the host to port 80 in the container.

PW SKILLS

Explaining the system requirements for installing Docker on different platforms.

Requirements

General Requirements

- 64-bit processor with virtualization support (e.g., Intel VT-x or AMD-V).
- Minimum RAM: 2GB, but 4GB or more is recommended for better performance.
- Disk Space: At least 10GB free, though 20GB or more is ideal for storing images and container data.

Requirements

Linux

- Kernel: 64-bit kernel with virtualization support.
- Virtualization: KVM enabled and QEMU version 5.2 or later.
- RAM: Minimum 4GB.
- Desktop Environment: Gnome, KDE, or MATE (tray icon support may require extensions).
- Systemd: Required as the init system.

Requirements

Windows

OS Versions: Windows 10 or 11 (64-bit) Home, Pro, Enterprise, or Education editions (22H2 or higher).

WSL2 Backend:

- WSL version 1.1.3.0 or later.
- Enable hardware virtualization in BIOS.

Hyper-V Backend:

- Requires SLAT-enabled processors and Hyper-V enabled in Windows features.

RAM: Minimum 4GB.

Requirements

MacOS

- Requires macOS Monterey (12.0) or later.
- Minimum of 4GB RAM and virtualization support.

PW SKILLS

Provide installation steps for: Linux, macOS & Windows

Docker installation

Linux: Installing Docker Engine using package managers:

1. Update package index:

```
sudo apt-get update
```

2. Install prerequisites:

```
sudo apt-get install ca-certificates curl gnupg
```

3. Add Docker's official GPG key:

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor
-o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Docker installation

4. Set up Docker repository:

```
 echo "deb [arch=$(dpkg --print-architecture) signed-
by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list
> /dev/null
```

5. Update package index again:

```
sudo apt-get update
```

6. Install Docker Engine:

```
 sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-
plugin docker-compose-plugin
```

Docker installation

macOS: Using Docker Desktop for Mac

- Download Docker Desktop for Mac from the official Docker website.
- Open the downloaded .dmg file.
- Drag Docker.app to the Applications folder.
- Launch Docker Desktop from Applications.
- Follow the on-screen instructions to complete the installation.

Docker installation

Windows: Installing Docker Desktop and enabling WSL2 backend

1. Enable WSL2:

```
 dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-
Linux /all /norestart
 dism.exe /online /enable-feature /featurename:VirtualMachinePlatform
/all /norestart
```

2. Download and install the WSL2 Linux kernel update package from Microsoft.

Docker installation

3. Set WSL2 as default:

```
wsl --set-default-version 2
```

4. Download Docker Desktop for Windows from the official Docker website.
5. Run the installer, ensuring to select the option to use WSL2 backend.
6. Follow the installation wizard to complete the setup.
7. Launch Docker Desktop and complete the initial configuration.

PW SKILLS

Discuss post-installation verification and troubleshooting

Let’s discuss

Verification Steps

1. **Check Docker Version:**

Run the following command to verify Docker installation:

```
docker --version
```

2. **Test Docker Daemon:**

Ensure the Docker daemon is running by executing:

```
docker info
```

3. **Run Test Container:**

Use the hello-world image to confirm Docker can pull and run containers:

```
docker run hello-world
```

Let's see

Troubleshooting Common Issues

1. **Daemon Not Running:**

Start the Docker service manually:

```
sudo systemctl start docker
```

2. **Permission Errors:**

Add your user to the docker group to avoid using sudo:

```
sudo usermod -aG docker $USER
```

3. **WSL2 Issues on Windows:**

Restart WSL kernel or regenerate certificates:

```
wsl --shutdown
docker-machine regenerate-certs
```

4. **Network Problems:**

Reset Docker network settings or restart the daemon:

```
docker network prune
sudo systemctl restart docker
```

Take A 5-Minute Break!

- Stretch and relax
- Hydrate
- Clear your mind
- Be back in 5 minutes

PW SKILLS

Explaining and demonstrating commonly used Docker commands

Let’s do it

Running a Container

The docker run command creates and starts a container from an image. You can use options to customize its behavior.

Example:

```
docker run --name my_container -d nginx
```

Let’s do it

Listing Active and Stopped Containers

The docker ps command displays running containers, while docker ps -a lists all containers, including stopped ones.

Examples:

```
docker ps          # Lists running containers
docker ps -a       # Lists all containers (running and stopped)
```

Let’s do it

Stopping, Restarting, and Removing Containers

You can manage containers using the following commands:

Stop a Running Container:

```
docker stop my_container
```

Restart a Container:

```
docker restart my_container
```

Remove a Container:

```
docker rm my_container
```

Removes a stopped container. Use -f to forcefully stop and remove a running container:

```
docker rm -f my_container
```

Let’s do it

Executing Commands Inside a Running Container

The docker exec command runs commands inside an active container.

Examples:

1. Run a command in the background:

```
docker exec -d my_container touch /tmp/newfile
```

2. Start an interactive shell session:

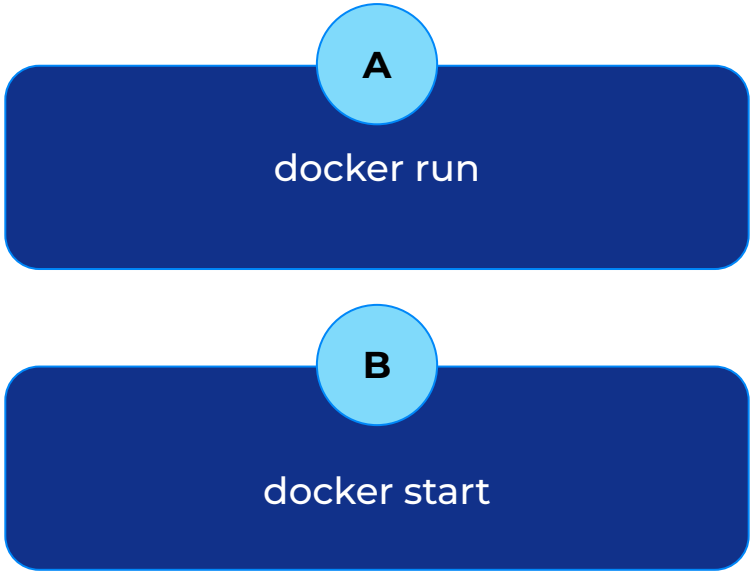
```
docker exec -it my_container sh
```

3. Set environment variables during execution:

```
docker exec -e VAR_A=1 -e VAR_B=2 my_container env
```

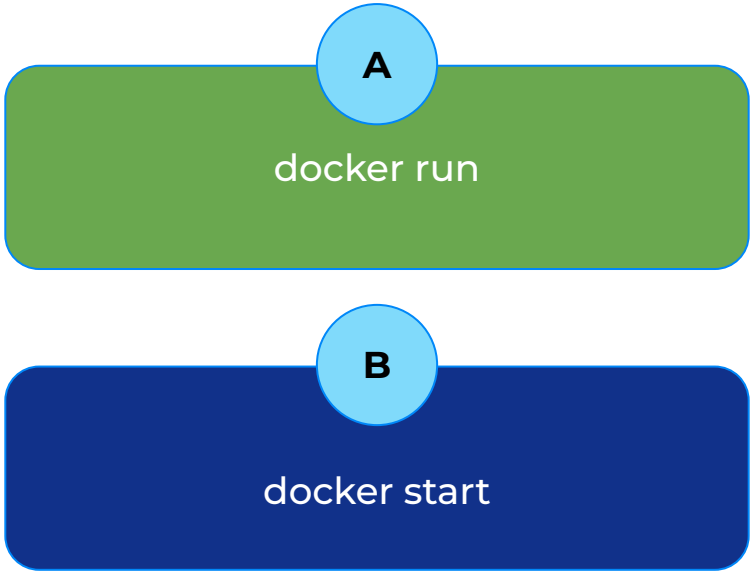
Pop Quiz

Q. Which command is used to run a Docker container?



Pop Quiz

Q. Which command is used to run a Docker container?



Pop Quiz

Q. What does the -d flag in the docker run command do?

A

Deletes the container after execution.

B

Runs the container in detached mode

Pop Quiz

Q. What does the -d flag in the docker run command do?

A

Deletes the container after execution.

B

Runs the container in detached mode

PW SKILLS

Explain the Dockerfile and its role in creating images.

Let's see

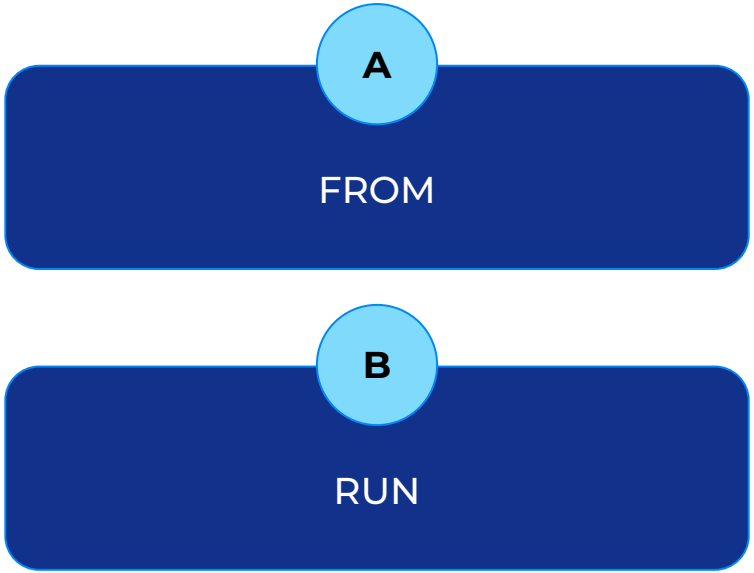
A Dockerfile is a text file containing a set of instructions used to define and build a Docker image. It automates the process of creating images by specifying the base image, software dependencies, configurations, and commands to run inside the container.

Role in Creating Images

1. Write a Dockerfile with desired instructions.
2. Build an image using: `docker build -t my_image .`
3. The resulting image can be used to create containers: `docker run my_image`

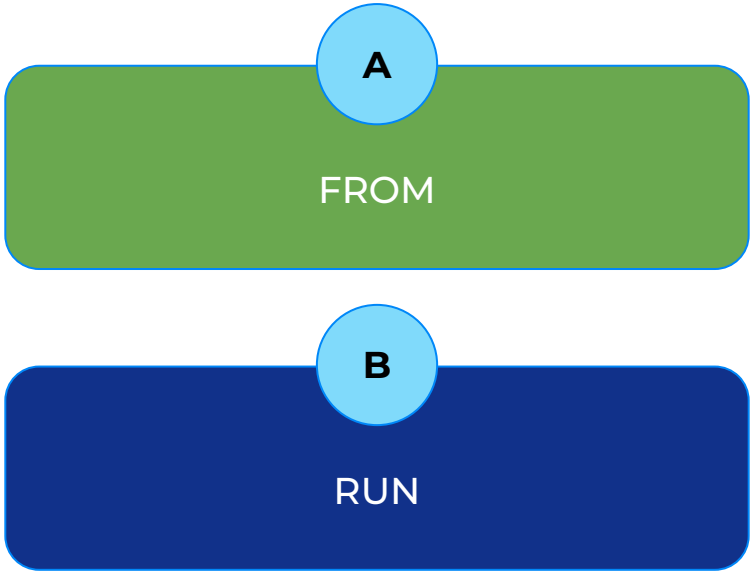
Pop Quiz

Q. Which instruction must start every Dockerfile?



Pop Quiz

Q. Which instruction must start every Dockerfile?



PW SKILLS

Discussing best practices for building optimized images

Let’s discuss

Using Smaller Base Images

- Start with minimal base images like Alpine Linux or distroless images.
- Alpine Linux base image can be as small as 5.59MB, while Nginx alpine base image is only 22MB.
- Use slim variants of official images, e.g., node:<version>-slim for Node.js applications.

Let’s discuss

Minimizing Layers

- Order Dockerfile instructions logically to avoid unnecessary cache invalidation.
- Combine related commands into a single RUN instruction using && to reduce layers.
- Use multi-stage builds to separate build and runtime environments, significantly reducing final image size.

Let’s discuss

Caching Dependencies

- Utilize cache mounts for persistent package caches during builds.

Example for npm:

```
RUN --mount=type=cache,target=/root/.npm npm install
```

This approach ensures only new or changed packages are downloaded in subsequent builds.

PW SKILLS

Explaining how Docker Hub is used for sharing and managing images

Let’s see

Docker Hub is a cloud-based registry service for sharing and managing Docker images.

Key features include:

- **Image storage:** Users can store both public and private repositories of Docker images.
- **Image distribution:** Easily share images with team members or the public.
- **Automated builds:** Integrate with GitHub or Bitbucket to automatically build images when code is updated.
- **Official and verified images:** Access to high-quality, pre-built images from Docker and trusted publishers.
- **Webhooks:** Trigger actions in other services when images are pushed.
- **Access control:** Manage team access to private repositories.
- **Image tagging:** Organize and version images using tags.

PW SKILLS

Demonstrating pushing a locally built image to Docker Hub

Let’s do it

Steps to Push an Image to Docker Hub

1. **Log in to Docker Hub:**

Use the docker login command to authenticate with your Docker Hub account.

```
docker login
```

2. **Tag the Image:**

Assign a tag that includes your Docker Hub username and repository name.

```
docker tag my_image dockerhubusername/my_image:latest
```

Let’s do it

3. **Push the Image:**

Use the docker push command to upload the tagged image to Docker Hub.

```
docker push dockerhubusername/my_image:latest
```

4. **Verify on Docker Hub:**

Log in to your Docker Hub account and check your repository to confirm the image was successfully uploaded.

Let’s do it

Example Workflow:

```
# Build the image locally
docker build -t my_image .

# Log in to Docker Hub
docker login

# Tag the image for Docker Hub
docker tag my_image dockerhubusername/my_image:latest

# Push the image to Docker Hub
docker push dockerhubusername/my_image:latest
```

PW SKILLS

Introducing different Docker networking modes and their use cases

Let’s discuss

1. **Bridge (Default Mode)**

Description: Containers are connected to an internal virtual network (bridge), isolated from the host network. Containers communicate using private IPs, and external access requires port mapping.

Use Case: Ideal for container-to-container communication on the same host and controlled external access.

Example:

```
docker run --network=bridge -p 8080:80 nginx
```

Let’s discuss

2. Host

Description: The container shares the host's network stack, using the host's IP and ports directly. No isolation is provided.

Use Case: Useful for performance-critical applications or when direct access to host resources is needed.

Example:

```
docker run --network=host nginx
```

Let’s discuss

3. None

Description: Disables networking entirely for the container, ensuring complete isolation from other containers and external networks.

Use Case: Suitable for containers that don't require any network connectivity, such as batch processing tasks.

Example:

```
docker run --network=none ubuntu
```

PW SKILLS

Explaining how containers communicate within the same network and across different networks

Let’s see

Within the Same Network

- Containers on the same user-defined bridge network can communicate directly using container names as hostnames.
- They can access each other's exposed ports without additional port mapping.
- Communication is isolated from containers not connected to the network.

Let’s see

Across Different Networks

- Containers on different networks cannot communicate directly by default.

To enable communication:

1. Connect containers to multiple networks using docker network connect.
2. Use the host network to route traffic between networks.
3. Implement overlay networks for multi-host communication.

PW SKILLS

Explaining Docker volumes and their role in persistent storage

Let’s see

Docker volumes are a mechanism for persistent storage in Docker containers, allowing data to outlive container lifecycles and enabling efficient sharing between containers. Here’s a concise explanation of their role:

Role of Docker Volumes

1. **Persistent Storage:** Volumes store data outside the container's ephemeral filesystem, ensuring it is retained even if the container is stopped, removed, or updated.
2. **Data Sharing:** Volumes enable sharing of data between multiple containers, such as configuration files or logs.
3. **Host-Container Interaction:** Volumes can link container paths to directories on the host machine for seamless data exchange.

PW SKILLS

Demonstrating creating and mounting volumes for data persistence

Let's see

1. Create a named volume:

```
docker volume create my_data
```

2. Run a container with the volume mounted:

```
docker run -d --name my_container -v my_data:/app/data nginx
```

3. Alternatively, use the --mount flag:

```
docker run -d --name my_container --mount source=my_data,target=/app/data nginx
```

Let's see

4. Verify the volume is mounted: `docker inspect my_container`

5. Data written to /app/data inside the container will persist on the host, even if the container is removed.

6. To use the same data in a new container:

```
docker run -d --name new_container -v my_data:/app/data nginx
```

PW SKILLS

Discussing the difference between bind mounts and named volumes

Let’s discuss

Bind mounts and named volumes are two methods for persistent storage in Docker, with key differences:

Bind Mounts

- Directly mount a host directory into the container.
- Host location is specified by the user.
- Provide direct access to host filesystem.
- Useful for development environments and sharing configuration files.
- Do not support Docker volume drivers.

Let’s discuss

Named Volumes

- Docker-managed virtual filesystems.
- Host location is managed by Docker.
- Provide better isolation from host system.
- Easier to back up, migrate, and share among containers.
- Support Docker volume drivers for advanced storage options.

Named volumes are generally preferred for persistent application data, while bind mounts are useful for development and when direct host access is needed.

PW SKILLS

Time for case study!

Important

- Complete the post-class assessment
- Complete assignments (if any)
- Practice the concepts and techniques taught in this sessio
- Review your lecture notes
- Note down questions and queries regarding this session ar
  consult the teaching assistants

Thanks !