Docker

> Docker is an open-source platform used to automate the deployment, scaling, and management of applications within containers.

▼ How docker solves the problem in modern development/deployment:

- Solves the typical problem “It works on my computer”
- Picture this: you've built an application on your local machine, and it works perfectly there. But when you try to move it to a different environment, chaos strikes!
- Docker solves this by ensuring that your application runs consistently across different environments – be it your laptop, a testing server, or a production server. It streamlines the development process and enhances portability, making your life as a developer much easier.
- Docker package the bundles everything your software needs to run: code, runtime, system tools, libraries – neatly packaged together for easy portability.
- **Isolation:** Containers provide a high level of isolation, allowing multiple applications or services to run independently on the same host without interfering with each other. This isolation enhances security and prevents conflicts between different software components.
- **Portability:** Docker containers can run on any system that supports Docker, regardless of the underlying infrastructure. This portability streamlines the deployment process and makes it easier to move applications between different cloud providers or local environments.
- **Resource Efficiency:** Containers share the host system's kernel and use fewer resources compared to traditional virtual machines. They start up quickly and have less overhead, allowing for more efficient use of computing resources.

- **Scalability:** Docker makes it easier to scale applications by quickly creating or removing containers as needed. This elasticity is crucial for managing varying workloads and handling increased user demand without significant infrastructure changes.

▼ Docker vs Virtual Machines

Aspect	Virtual Machines (VMs)	Docker Containers
Technology	Hypervisor-based	Containerization (OS-level virtualization)
Resource Usage	High resource usage (separate OS per VM)	Lower resource usage (share host OS kernel)
Startup Time	Slower	Faster
Isolation	Strong isolation	Less isolated (share host OS kernel but separate user space)
OS Compatibility	Supports various OS types	Best when host and container OS are similar
Portability	Less portable	Highly portable between environments running Docker
Scalability	Slower scaling due to resource needs	Rapid scaling, efficient deployment and scaling

▼ Images

Docker images are the blueprints for containers. They're lightweight, standalone, executable packages that contain everything needed to run a piece of software, including the code, runtime, libraries, tools, and system dependencies.

Pulling an Image:

```
docker pull <image_name>:<tag>
# Example: docker pull ubuntu:latest
```

Listing Images:

```
docker images
docker images <image_name>
```

Removing Images:

```
docker rmi <image_id or image_name>:<tag>
# Example: docker rmi ubuntu:latest
```

▼ Containers

A Docker container is a runtime instance of a Docker image. It encapsulates an application and its dependencies, including libraries and other binaries, to ensure consistency and portability across different environments.

Key Points:

- **Image**: A blueprint for containers, including the application code, libraries, dependencies, and configuration needed to run the application.
- **Container**: A runtime instance of an image, isolated from the host system and other containers, but sharing the same kernel as the host OS.

Container Lifecycle:

```
# Create and start a container
docker run -d --name <container_name> <image_name>

# List running containers
docker ps

# List all containers (including stopped)
docker ps -a

# Stop a running container
docker stop <container_name>

# Start a stopped container
docker start <container_name>
```

```
# Remove a container
docker rm <container_name>
```

Interaction and Information:

```
# Get logs from a container
docker logs <container_name>

# Execute a command in a running container
docker exec -it <container_name> <command>

# View container details (including IP, ports)
docker inspect <container_name>

# View real-time resource usage of containers
docker stats
```

▼ Volumes

A Docker volume is a directory or filesystem that is managed by Docker but is outside the container's filesystem. It allows data to persist beyond the lifecycle of a container. Volumes can be used for various purposes like sharing data between containers, persisting database files, or storing configuration files.

Key Points:

- **Persistence**: Volumes persist data even if the container using them is stopped or deleted.
- **Shared Data**: Volumes enable sharing of data between containers or between the host and containers.
- **Performance**: Volumes provide faster I/O than bind mounts due to their design.

Volume Lifecycle:

```
# List Docker volumes
docker volume ls

# Create a named volume
docker volume create <volume_name>

# Inspect details of a volume
```

```
docker volume inspect <volume_name>

# Remove a volume
docker volume rm <volume_name>
```

Using Volumes with Containers:

```
# Start a container with a named volume
docker run -v <volume_name>:<container_mount_path> <image_name>

# Start a container with an anonymous volume
docker run -v <container_mount_path> <image_name>

# Mount multiple volumes to a container
docker run -v <volume1_name>:<container_mount_path1> -v <volume2_name>:<container_moun
t_path2> <image_name>
```

Removing Volumes Associated with Containers:

```
# Remove all unused volumes
docker volume prune

# Remove a volume even if it's in use by a container
docker volume rm -f <volume_name>
```

Backup and Restore Volumes:

```
# Backup a volume to a tar archive
docker run --rm -v <volume_name>:/data -v $(pwd):/backup busybox tar czf /backup/<volu
me_name>_backup.tar.gz /data

# Restore a volume from a tar archive
docker run --rm -v <volume_name>:/data -v $(pwd):/backup busybox tar xzf /backup/<volu
me_name>_backup.tar.gz -C /data
```

▼ Network

A Docker network is a communication pathway that allows containers within the same network to communicate with each other. Networks enable containers to discover and interact with each other via a virtual network interface. Docker supports different network drivers, each suitable for specific use cases such as bridge networks for single host communication or overlay networks for spanning multiple hosts.

Key Points:

- **Default Networks**: Docker creates a default bridge network for containers to communicate on the same host.
- **Custom Networks**: Users can create custom networks to isolate or group containers based on specific needs.
- **Networking Drivers**: Docker offers various drivers like bridge, overlay, macvlan, etc., each with its own features and use cases.

Network Lifecycle:

```
# List Docker networks
docker network ls

# Inspect details of a network
docker network inspect <network_name>

# Create a bridge network
docker network create <network_name>

# Remove a network
docker network rm <network_name>
```

Creating Containers on Specific Networks:

```
# Start a container and attach it to a specific network
docker run --network <network_name> <image_name>

# Connect a running container to a network
docker network connect <network_name> <container_name>

# Disconnect a container from a network
docker network disconnect <network_name> <container_name>
```

▼ Docker Engine:

The Docker Engine is a software application that enables the creation, management, and running of Docker containers. It's the core component of the Docker platform and provides the necessary tools and interfaces to interact with containers and their underlying infrastructure.

Components of Docker Engine:

1. **Docker Daemon (dockerd)**:
   - The Docker daemon is the background service that manages the container lifecycle, handling container creation, running, and stopping.
   - It listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.
2. **REST API:**
   - Docker exposes a REST API that allows users to interact with the Docker daemon, enabling operations like container creation, network management, and more.
   - This API provides a standardized interface for controlling and querying Docker functionality.
3. **Command-Line Interface (CLI):**
   - The Docker CLI (docker) is the command-line tool used to interact with the Docker Engine. It provides a user-friendly interface for running Docker commands to manage containers, images, networks, volumes, and more.

▼ How Docker Makes Deployment Easy

Challenges in Traditional Deployment flow:

- Environment/Package Mismatch
- Scalability
- Slow Deployment
- Manual Works
- Conflict between developer and devops team.

Example of Building Image using Docker File and Docker Compose:

https://github.com/tejas-gudulekar/docker-mean.git

Resources:

Use containers for Node.js development

Learn how to develop your Node.js application locally using containers.

https://docs.docker.com/language/nodejs/develop/

▼ Publishing your Image on Docker

1. Docker Login

Open your terminal or command prompt and log in to Docker Hub using the `docker login` command:

```
docker login
```

2. Create a Docker Image from a Container

a. Find the Container ID:

```
docker ps
```

Identify the CONTAINER ID of the container you want to convert into an image.

b. Commit the Container to an Image:

Use the `docker commit` command to create an image from the running container:

```
docker commit <CONTAINER_ID> <IMAGE_NAME>:<TAG>
```

3. Tag Your Image

After creating the image, tag it appropriately for Docker Hub:

```
docker tag <IMAGE_NAME>:<TAG> <DOCKERHUB_USERNAME>/<REPOSITORY_NAME>:<TAG>
```

4. Push the Image

Push the tagged image to Docker Hub:

```
docker push <DOCKERHUB_USERNAME>/<REPOSITORY_NAME>:<TAG>
```

5. Confirm on Docker Hub

Log in to Docker Hub in your web browser and navigate to your account. You should see your repository with the pushed image and its tags listed.

Tips:

- **Versioning**: Consider using version numbers in your tags (`<USERNAME>/<REPOSITORY>:v1.0`, `<USERNAME>/<REPOSITORY>:latest`, etc.) to manage different versions.
- **Security**: Ensure your Docker image follows security best practices before pushing it to Docker Hub.
- **README and Description**: Add a README and description on Docker Hub to explain your container's usage and purpose.