DOCKER | PART-1

Traditional Deployment Challenges:

Before Docker and containerization technologies became popular, applications were typically deployed on dedicated physical or virtual machines. Some of the key problems with this approach were:

1. **Resource Inefficiency**: Running multiple applications on a single machine often caused resource contention.
2. **Dependency Hell**: Different applications required different dependencies, leading to version conflicts when installed on the same machine.
3. **Slow Development to Production Pipeline**: Moving applications from development to production environments was often a nightmare due to differences in environments.
4. **Complex Setup & Maintenance**: Developers had to maintain separate environments for development, testing, and production.
5. **Scalability Issues**: Scaling applications required scaling entire machines, leading to inefficient use of resources.

How Docker Solves These Problems:

Docker solves these problems using **containerization**. Containers package an application and its dependencies in a standardized unit for development, ensuring consistency across environments. Docker offers:

1. **Isolation**: Containers run isolated from each other, preventing conflicts in dependencies.
2. **Resource Efficiency**: Containers share the same OS kernel, allowing for lightweight and efficient resource usage compared to VMs.
3. **Portability**: Docker containers run consistently across any environment—be it development, testing, or production.
4. **Simplified Deployment**: Docker enables faster CI/CD pipelines by simplifying deployments, rollbacks, and scaling.

What is Docker? Docker vs. Virtualization

What is Docker?

Docker is a **containerization platform** that allows developers to package applications into lightweight, portable containers. Containers include everything an application needs to run—code, libraries, dependencies, and configuration files—making it easy to move across different environments.

Key Benefits of Docker:

- **Portability**: Containers can run across any system that has Docker installed.
- **Lightweight**: Containers are much smaller than virtual machines (VMs) because they share the host system's OS kernel.
- **Scalability**: Docker makes it easy to scale applications by running multiple instances of containers.

Docker vs. Virtualization

Docker (Containerization)	**Virtualization (VMs)**
Lightweight: Containers share the host OS kernel	Heavyweight: Each VM runs a full OS instance
Faster startup (seconds)	Slower startup (minutes)
Efficient resource utilization	Requires more resources
More portable	Less portable due to OS overhead
Shared OS, isolated app environments	Isolated OS and apps in each VM

In summary, Docker is more lightweight and portable than traditional virtualization, as containers share the host operating system’s kernel, while VMs run separate OS instances.

Docker & Its Architecture

Docker Architecture:

Docker's architecture is based on the client-server model, comprising of the following components:

1. **Docker Client**:

    - The Docker client is used to interact with the Docker daemon (server). Commands like docker build, docker run, and docker pull are sent via the Docker client.

2. **Docker Daemon (Docker Engine)**:
    - The Docker daemon is the server that performs the heavy lifting of building, running, and managing Docker containers. It listens for Docker API requests and manages Docker objects (containers, images, volumes, networks).
3. **Docker Images**:
    - A Docker image is a read-only template that contains instructions for creating a container. Images are built from Dockerfiles.
4. **Docker Containers**:
    - A container is a runnable instance of a Docker image. Containers are isolated environments that include everything the application needs to run.
5. **Docker Registry**:
    - A Docker registry is where Docker images are stored. Docker Hub is a popular public registry, but you can also set up a private registry. Docker client interacts with the registry to pull and push images.

Dockerfile, Docker Image, Docker Container

Dockerfile:

A **Dockerfile** is a script with a set of instructions used to create a Docker image. Each instruction in the Dockerfile adds a layer to the image, making Docker images lightweight and efficient.

Key Dockerfile Instructions:

- **FROM**: Specifies the base image for the container (e.g., FROM ubuntu).
- **RUN**: Executes commands in the container during image build time.
- **COPY/ADD**: Copies files from the host to the container.
- **CMD**: Sets the default command to run when a container starts.
- **ENTRYPOINT**: Configures a container to run as an executable.
- **EXPOSE**: Defines the ports on which the container will listen for requests.

Docker Image:

- A **Docker image** is a read-only template containing the application and its dependencies. Images are created using Dockerfiles and are portable across environments.

Docker Container:

- A **Docker container** is a running instance of a Docker image. Containers are isolated environments but share the host OS kernel. They can be created, started, stopped, moved, or deleted using Docker commands.

Dockerfile: Writing & Keywords

A **Dockerfile** is essentially a blueprint or script that defines how to create a Docker image. It contains a set of instructions that Docker reads to build an image layer by layer. Every line in a Dockerfile is a command that contributes to constructing the final image.

Docker follows a layered approach where each instruction in the Dockerfile adds a layer to the image. This enables Docker to reuse layers to optimize both storage and efficiency.

Let's explore how to write a Dockerfile and what each key instruction does in detail:

---

1. FROM

Purpose: Specifies the base image from which your Docker image will be built. This is always the **first instruction** in any Dockerfile. It serves as the starting point for your container environment.

Example:

```
FROM ubuntu:20.04
```

This line tells Docker to use the official ubuntu:20.04 image as the base layer. All subsequent commands will be applied on top of this image.

Multi-stage Builds:

In more complex scenarios, we can use multiple FROM statements to create **multi-stage builds**, where we copy files from one image to another, reducing the final image size.

```
# First stage - build the application
FROM node:14 AS build
WORKDIR /app
COPY . .
RUN npm install && npm run build

# Second stage - create the final image
FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
```

In this example, the first FROM builds the application, and the second FROM uses a smaller nginx image to serve the application.

2. WORKDIR

Purpose: Sets the working directory for any subsequent instructions in the Dockerfile. This avoids using long directory paths and ensures the container's commands run in the intended folder.

Example:

```
WORKDIR /app
```

This sets the working directory inside the container to /app. Any COPY, RUN, or CMD commands following this will be executed relative to this directory.

3. COPY and ADD

Both COPY and ADD copy files from the host machine into the Docker container. However, they have different use cases and capabilities.

COPY:

- **Purpose**: Used to copy files and directories from the host filesystem into the Docker container.
- **Syntax**:

```
COPY <source_path> <destination_path>
```

<source_path> is the path on the host machine, and <destination_path> is the target location inside the container.

ADD:

- **Purpose**: ADD works like COPY but with two additional features:
    1. If the source is a **tar archive** (e.g., .tar, .gz), Docker will automatically extract it into the destination.
    2. ADD can also download files from remote URLs.
- **Syntax**:

```
ADD <source_path> <destination_path>
```

Example of COPY:

```
COPY ./local-file.txt /app/
```

This command will copy the local-file.txt from the current directory on the host to the /app/ directory in the container.

Example of ADD:

```
ADD ./archive.tar.gz /app/
```

This will extract the contents of archive.tar.gz into the /app/ directory.

```
ADD http://example.com/file.txt /app/
```

This downloads the file file.txt from the given URL and places it inside /app/.

Best Practice:

- **Use COPY when copying local files** and directories.
- **Use ADD only if you need** its special features like downloading files from URLs or auto-extracting archives.

4. RUN

Purpose: Used to execute commands in a new layer on top of the current image. It's commonly used to install packages, set environment variables, or perform system-level configurations.

Syntax:

```
RUN <command>
```

Example:

```
RUN apt-get update && apt-get install -y curl
```

This command updates the package index and installs the curl package in the container.

Chaining RUN Commands:

To keep the image size small, it's a good practice to chain multiple commands in a single RUN instruction to avoid creating unnecessary layers.

```
RUN apt-get update && apt-get install -y \
  git \
  curl \
  && rm -rf /var/lib/apt/lists/*
```

In this example, we're installing multiple packages in one layer and cleaning up afterward to reduce image size.

5. CMD vs ENTRYPOINT

Both CMD and ENTRYPOINT define what command should be executed when the container starts. However, their behavior differs slightly.

CMD:

- **Purpose**: Specifies the default command to run when the container starts. It can be overridden by passing arguments when running docker run.
- **Syntax**:

```
CMD ["executable", "param1", "param2"]
```

ENTRYPOINT:

- **Purpose**: Defines the command that will always run when the container starts. It is less flexible than CMD because it can't be easily overridden by docker run unless you use the --entrypoint flag.
- **Syntax**:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

Difference:

- CMD provides **default arguments** that can be overridden.
- ENTRYPOINT makes the container **behave like an executable**, meaning it will always run the specified command.

Examples:

CMD Example:

```
FROM ubuntu
CMD ["echo", "Hello World"]
```

When you run the container, it will execute echo "Hello World". However, if you pass a different command during docker run, it will override the default CMD.

```
docker run my-image echo "Hi"
```

This will print "Hi" instead of "Hello World".

ENTRYPOINT Example:

```
FROM ubuntu

ENTRYPOINT ["echo"]

CMD ["Hello World"]
```

In this case, ENTRYPOINT always runs echo, and CMD provides the default argument "Hello World". If you override CMD, the ENTRYPOINT will still execute:

```
docker run my-image "Hi"
```

This will print "Hi" because the ENTRYPOINT command is echo, and "Hi" replaces the default CMD.

When to Use CMD or ENTRYPOINT:

- Use **CMD** if you want to provide default behavior but allow users to override it.
- Use **ENTRYPOINT** if you want to enforce a specific command and only allow the arguments to change.

---

6. ENV

Purpose: Sets environment variables in the container. These variables can be accessed by the application running inside the container.

Syntax:

```
ENV <key>=<value>
```

Example:

```
ENV NODE_ENV production
```

This sets the environment variable NODE_ENV to production.

7. EXPOSE

Purpose: Informs Docker that the container will listen on a specific port at runtime. This does not actually publish the port but serves as metadata that Docker uses.

Syntax:

```
EXPOSE <port_number>
```

Example:

```
EXPOSE 3000
```

This informs Docker that the container will listen on port 3000. You still need to map this port to the host using the -p flag in docker run:

```
docker run -p 3000:3000 my-image
```

EXPOSE vs Publishing Ports:

- EXPOSE just **declares** the port but doesn't actually expose it outside the container.
- To make the port accessible outside, you must use the -p or --publish option when running the container.

8. VOLUME

Purpose: Declares a directory as a volume to be shared between the host and the container, enabling **persistent storage**.

Syntax:

```
VOLUME ["/path/in/container"]
```

Example:

```
VOLUME ["/app/data"]
```

This makes /app/data a volume where changes in the container are reflected on the host, and vice versa.

9. ENTRYPOINT and CMD Combined

In some cases, you can use both ENTRYPOINT and CMD together. ENTRYPOINT defines the base command, while CMD provides additional default parameters.

Example:

```
FROM ubuntu
ENTRYPOINT ["ping"]
CMD ["localhost"]
```

- ENTRYPOINT is set to always run the ping command.
- CMD provides the default argument localhost.

When you run the container:

```
docker run my-image
```

It will execute ping localhost.

If you override CMD:

```
docker run my-image google.com
```

It will execute ping google.com.

Conclusion

Understanding the structure and key instructions of a Dockerfile is essential for efficiently building Docker images. Dockerfile instructions like FROM, COPY, ADD, CMD, ENTRYPOINT, and others allow you to customize your image according to the application's requirements.

Key Takeaways:

1. **COPY vs ADD**: Use COPY for basic file copying, and ADD when you need to download files from URLs or extract archives.
2. **CMD vs ENTRYPOINT**: Use CMD to provide default commands or arguments, and ENTRYPOINT to enforce a command that always runs.
3. **EXPOSE**: Declares the port your container will listen on, but to make it accessible, you need to publish it with the -p option.

Docker's layered approach helps create optimized, reusable images that can be run across different environments with consistent behaviour.

Commands

Writing and Using a Dockerfile

The first step in the Docker workflow is creating a **Dockerfile**, which defines how your Docker image will be built. Once the Dockerfile is ready, you can use Docker commands to build the image.

1.1. Sample Dockerfile

Here's an example of a simple Dockerfile for a Node.js application:

```
# Use the official Node.js image as the base image
FROM node:14
# Set the working directory inside the container
WORKDIR /app
# Copy package.json and package-lock.json
COPY package*.json ./
# Install Node.js dependencies
RUN npm install
# Copy the rest of the application files
COPY . .
# Expose the application port
EXPOSE 3000
# Define the default command to run when the container starts
CMD ["npm", "start"]
```

Step 2: Docker Image Commands

A Docker image is a read-only template that contains the instructions and dependencies to create a container. After writing the Dockerfile, you can use Docker commands to build, manage, and inspect Docker images.

2.1. Building the Docker Image

Once you've created the Dockerfile, you use the docker build command to create a Docker image from it.

Syntax:

```
docker build -t <image_name>:<tag> <path_to_dockerfile>
```

Example:

```
docker build -t mynodeapp:1.0 .
```

- -t specifies the tag (mynodeapp:1.0), which names and optionally tags the image.
- . specifies the path to the Dockerfile (the current directory in this case).

Explanation:

- Docker reads the Dockerfile in the current directory and builds an image named mynodeapp with the tag 1.0.
- Each line of the Dockerfile creates a new layer in the image.
- You can view the build progress for each step (e.g., downloading base images, installing dependencies, etc.).

2.2. Listing Docker Images

To see the list of all Docker images on your system, use:

Command:

```
docker images
```

Example Output:

```
REPOSITORY    TAG      IMAGE ID      CREATED        SIZE
mynodeapp     1.0      3456789abcde  1 minute ago   910MB
ubuntu        20.04    d13d34e5abc1  2 weeks ago    72MB
```

- **REPOSITORY**: The name of the image (e.g., mynodeapp).
- **TAG**: The tag for different versions of the image (e.g., 1.0).
- **IMAGE ID**: The unique ID of the image.
- **CREATED**: When the image was created.
- **SIZE**: The size of the image.

2.3. Removing Docker Images

To remove a Docker image, use the docker rmi command followed by the image name or ID.

Command:

```
docker rmi <image_name>
```

Example:

```
docker rmi mynodeapp:1.0
```

This removes the mynodeapp:1.0 image from your system. Make sure no running containers are using the image, or you will need to stop them first.

Step 3: Docker Container Commands

A **Docker container** is a running instance of a Docker image. Once an image is built, you can create and manage containers from that image.

3.1. Running a Docker Container

The docker run command creates and starts a container based on a Docker image. If the image does not exist locally, Docker will pull it from the Docker registry (e.g., Docker Hub).

Syntax:

```
docker run -d -p <host_port>:<container_port> --name <container_name> <image_name>:<tag>
```

Example:

```
docker run -d -p 8080:3000 --name mynodecontainer mynodeapp:1.0
```

- -d: Runs the container in detached mode (in the background).
- -p 8080:3000: Maps port 8080 on the host to port 3000 in the container.
- --name: Assigns a name (mynodecontainer) to the running container.
- mynodeapp:1.0: Specifies the image and tag to use for the container.

Once the container is running, you can access the application in the container via http://localhost:8080.

3.2. Listing Docker Containers

You can see all running containers with the docker ps command:

Command:

```
docker ps
```

Example Output:

```
CONTAINER ID   IMAGE          COMMAND       STATUS         PORTS                NAMES
6bc123456789   mynodeapp:1.0   "npm start"    Up 10 seconds  0.0.0.0:8080->3000/tcp  mynodecontainer
```

- **CONTAINER ID**: The unique ID of the running container.
- **IMAGE**: The image used to create the container.
- **COMMAND**: The command running in the container.
- **STATUS**: The current status of the container (e.g., Up 10 seconds).
- **PORTS**: Shows the port mappings (e.g., 8080 on the host to 3000 in the container).
- **NAMES**: The name of the container.

To list all containers (both running and stopped), use:

docker ps -a

3.3. Stopping and Restarting Containers

To stop a running container, use the docker stop command followed by the container name or ID.

Command:

docker stop <container_name_or_id>

Example:

docker stop mynodecontainer

To restart a stopped container:

Command:

docker start <container_name_or_id>

Example:

docker start mynodecontainer

3.4. Removing Containers

To remove a container (whether it's running or stopped), use the docker rm command.

Command:

docker rm <container_name_or_id>

Example:

docker rm mynodecontainer

If you want to force-remove a running container, use the -f flag:

Command:

docker rm -f mynodecontainer

3.5. Viewing Logs of a Running Container

To view the logs of a running container, use the docker logs command.

Command:

docker logs <container_name_or_id>

Example:

docker logs mynodecontainer

This command shows the logs from the container's standard output.

3.6. Accessing a Running Container's Shell

Sometimes you need to access the shell inside a running container for debugging or manual inspection. Use the docker exec command to run commands inside a running container.

Command:

docker exec -it <container_name_or_id> /bin/bash

Example:

docker exec -it mynodecontainer /bin/bash

This command opens an interactive shell inside the container. You can now run shell commands just as if you were inside a regular Linux system.

3.7. Inspecting Containers

The docker inspect command gives detailed information about the container, including its configuration, environment variables, networking, and more.

Command:

docker inspect <container_name_or_id>

Example:

docker inspect mynodecontainer

This returns a JSON object containing all the metadata about the container.

3.8. Exporting and Importing Containers (continued)

You can export a container's filesystem as a tarball and import it later. This is useful for moving containers between environments.

Exporting a Container:

docker export <container_name_or_id> > container.tar

This command will export the entire filesystem of the container into a container.tar file.

Example:

docker export mynodecontainer > mycontainer.tar

The container's filesystem is saved in mycontainer.tar, and you can transfer this file to another environment.

Importing a Container:

To import the exported container tarball into another Docker environment, use the docker import command:

cat mycontainer.tar | docker import - myimage:1.0

This command will import the container's filesystem into a new Docker image called myimage with the tag 1.0.

3.9. Committing Changes to a Running Container

If you make changes to a running container (e.g., install new software or modify configurations), you can save these changes by creating a new Docker image from the container using the docker commit command.

Command:

docker commit <container_name_or_id> <new_image_name>:<tag>

Example:

docker commit mynodecontainer mymodifiedimage:1.0

This creates a new Docker image (mymodifiedimage:1.0) that contains all the changes made in the running container mynodecontainer. You can then use this new image to run containers in the future.

3.10. Copying Files from a Running Container

You can copy files from a running container to your host system using the docker cp command.

Command:

docker cp <container_name_or_id>:<container_path> <host_path>

Example:

docker cp mynodecontainer:/app/logs/app.log ./logs/

This command copies the app.log file from the /app/logs/ directory inside the running container (mynodecontainer) to the ./logs/ directory on the host system.

Step 4: Managing Docker Networks

Docker containers can communicate with each other through Docker networks. Docker automatically creates a default bridge network, but you can create custom networks for more control.

4.1. Creating a Network

To create a custom Docker network, use the docker network create command:

Command:

docker network create <network_name>

Example:

docker network create mynetwork

This creates a network named mynetwork.

4.2. Running a Container in a Custom Network

To attach a container to a custom network, use the --network flag with the docker run command:

Command:

docker run -d --name <container_name> --network <network_name> <image_name>:<tag>

Example:

docker run -d --name myapp --network mynetwork mynodeapp:1.0

This runs the mynodeapp:1.0 container and attaches it to the mynetwork network.

4.3. Connecting Containers Across Networks

You can connect an existing running container to another network using the docker network connect command.

Command:

docker network connect <network_name> <container_name_or_id>

Example:

docker network connect mynetwork mynodecontainer

This connects the mynodecontainer to the mynetwork network, allowing it to communicate with other containers on that network.

Step 5: Docker Volumes

Volumes are the best way to persist data generated by and used by Docker containers. Volumes are stored outside the container's filesystem and can be shared among multiple containers.

5.1. Creating a Volume

To create a volume, use the docker volume create command:

Command:

docker volume create <volume_name>

Example:

docker volume create myvolume

This creates a volume named myvolume.

5.2. Using Volumes in a Container

To use a volume in a container, use the -v or --mount flag when running the container. This mounts the volume inside the container's filesystem.

Command:

docker run -d -v <volume_name>:<path_in_container> <image_name>:<tag>

Example:

docker run -d -v myvolume:/app/data mynodeapp:1.0

This command mounts the myvolume volume to the /app/data directory inside the container.

5.3. Inspecting Volumes

To view the details of a specific volume, use the docker volume inspect command:

Command:

docker volume inspect <volume_name>

Example:

docker volume inspect myvolume

This shows details about the volume, such as its mount point and configuration.

5.4. Removing Volumes

To remove an unused volume, use the docker volume rm command:

Command:

docker volume rm <volume_name>

Example:

docker volume rm myvolume

If the volume is still in use by a running container, you'll need to stop the container first or use the -f (force) flag.

Step 6: Docker Cleanup Commands

Over time, your Docker environment may accumulate unused images, containers, networks, and volumes, consuming disk space. Docker provides cleanup commands to remove unused resources.

6.1. Pruning Unused Images, Containers, Networks, and Volumes

To remove all stopped containers, dangling images, unused networks, and unused volumes, use the docker system prune command:

Command:

docker system prune

This command cleans up all unused resources. If you want to remove **all images** and containers (not just stopped ones), use the --all flag:

```
docker system prune --all
```

6.2. Removing Dangling Images

Dangling images are those that are not tagged and have no containers associated with them. To remove dangling images, use:

Command:

```
docker image prune
```

This removes all dangling images.